SKYLINE QUERIES IN DYNAMIC ENVIRONMENTS

HUA LU

Master of Science
Peking University, China

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2006

Acknowledgement

First of all, my deep gratitude goes to my supervisors Prof. Beng Chin Ooi and Dr. Zhiyong Huang. I sincerely appreciate their guidance, patience and encouragement, which collectively helped me survive all challenges, pains and even desperation during the period of my candidature.

I would like to thank Prof. Christian S. Jensen, who warmheartedly offered valuable help to improve my research work. Also, he generously hosted me in Aalborg University for three months in 2005, which left me a memorable experience.

I would like to thank Prof. Kian-Lee Tan and Dr. Anthony K. H. Tung for their kind suggestions on my study and research. I am also thankful to Prof. Mong Li Lee and Prof. Chee Yong Chan. As my thesis advisory committee members, they provided constructive advice on my research work and thesis composition. I would also like to thank Prof. Bo Huang, who used to be my co-supervisor in 2003 and 2004 when he was still working at NUS.

Special thanks go to all fellow members of SoC database labs. Discussing, chatting and gathering with these smart and easy-going people gave me wonderful memories of those monotonous days.

Last but not least, I am deeply indebted to my family. They did so much for me that I could concentrate on my PhD work. Without their continuous support and encouragement, I would probably have given it up somewhere on my way to this point.

CONTENTS

Acknowledgement **ii**

List of Tables **viii**

List of Figures **ix**

Summary **xiii**

1 Introduction **1**

1.1 The Concept of Skyline Query 2

1.2 Motivation 4

1.3 Contributions 6

1.3.1 Continuous Skyline Queries for Moving Objects 6

1.3.2 Skyline Queries on Mobile Lightweight Devices 7

1.3.3 Skyline Queries Against Mobile Lightweight Devices in MANETs 8

1.4 Organization 9

1.5 An Overall Picture 11

2 Background **14**

2.1 Skyline Queries 14

2.1.1 Skyline Queries in Centralized Environments 15

2.1.2 Variants and Derivatives of Centralized Skyline Queries . . . 21

2.1.3 Skyline Queries on Data Streams 24

2.1.4 Skyline Queries in Distributed Environments 25

2.1.5 Skyline Cardinality Estimation 26

2.2 Continuous Queries in Relation to Moving Objects 27

2.3 Data Management in MANETs 30

2.3.1 Mobile Ad-Hoc Networks 30

2.3.2 Data Management in Mobile Ad-Hoc Networks 30

3 Continuous Skyline Queries for Moving Objects **32**

3.1 Introduction 33

3.2 Preliminaries 36

3.2.1 Problem Statement 36

3.2.2 Time Parameterized Distance Function 37

3.2.3 Terminologies 38

3.3 The Change of Skyline in Moving Context 38

3.3.1 Search Bound 38

3.3.2 Change in the Skyline 40

3.3.3 Continuous Skyline Query Processing 46

3.4 Data Structure and Algorithms 48

3.4.1 Data Structure 48

3.4.2 Algorithms 50

3.4.3 Updating the Moving Plan 56
3.5 Cost Analysis and Discussion 57
3.5.1 Cost Analysis 57
3.5.2 Possible Extensions 61
3.6 Performance Studies 62
3.6.1 Effect of Cardinality 63
3.6.2 Effect of Non-spatial Dimensionality 67
3.6.3 Effect of Movement Update 72
3.6.4 Effect of Speed Distribution 75
3.7 Summary 75

4 Skyline Queries on Mobile Lightweight Devices 77

4.1 Introduction 78
4.2 Problem Definition 80
4.3 Data Storage Scheme on Lightweight Devices 82
4.3.1 Existing Storage Schemes for Limited Space 82
4.3.2 Hybrid Storage Scheme 85
4.3.3 Discussion 86
4.4 Skyline Computation on Lightweight Device 88
4.4.1 Flat Storage Based Skyline Algorithm with Pre-computed Skyline Points 88
4.4.2 Hybrid Storage Based Skyline Computation 91
4.5 Performance Studies 94
4.5.1 Storage Space Cost 96
4.5.2 Skyline Query Processing Time 96
4.6 Summary 100

5 Skyline Queries Against Mobile Lightweight Devices in MANETs 102

5.1 Introduction . . . 103

5.2 Problem Definition . . . 105

5.3 Unsophisticated Distributed Skyline Processing in MANETs . . . 107

5.3.1 Naive Strategy . . . 107

5.3.2 Straightforward Strategy . . . 108

5.4 Efficient Distributed Skyline Processing in MANETs . . . 110

5.4.1 Filtering Tuple Based Strategy . . . 110

5.4.2 Estimated Dominating Region . . . 113

5.4.3 Adaptations to Wireless Ad Hoc Networks . . . 114

5.5 Local Configurations on Mobile Devices . . . 116

5.5.1 Dataset Storage . . . 116

5.5.2 Local Skyline Computing . . . 116

5.5.3 Assembly on Query Originator . . . 117

5.6 Performance Studies . . . 119

5.6.1 Experimental Settings . . . 119

5.6.2 Data Reduction Efficiency . . . 121

5.6.3 Response Time . . . 129

5.6.4 Query Message Count . . . 133

5.6.5 Data Reduction Efficiency with Multiple Filtering Tuples . . . 134

5.7 Summary . . . 138

6 Conclusions and Future Work 139

6.1 Conclusions . . . 140

6.1.1 Continuous Skyline Queries for Moving Objects . . . 140

6.1.2 Skyline Queries on Mobile Lightweight Devices . . . 141

6.1.3 Skyline Queries Against Mobile Lightweight Devices in MANETs 142
6.1.4 Discussion 143
6.2 Directions for Future Work 144

Bibliography **146**

LIST OF TABLES

3.1 Intersections and possible skyline changes 43

3.2 Certificates 50

3.3 Parameters used in experiments 62

4.1 Parameters used in experiments 94

4.2 Reasonable algorithm/storage combinations 97

5.1 Symbols used in discussion 107

5.2 Example relation R_1 112

5.3 Example relation R_2 112

5.4 Example relation R_3 114

5.5 Example relation R_4 114

5.6 Parameters used in experiments 119

5.7 Parameters used in MANET simulations 121

LIST OF FIGURES

1.1 A classical example of skyline of hotels 2

1.2 Skyline of Singapore city 3

1.3 An overall picture of this thesis 12

3.1 An example of skyline in a static scenario 34

3.2 Skylines in mobile environment 35

3.3 An example of distance function curves 41

3.4 An example of multiplex intersection 44

3.5 An example of evolving intersections 47

3.6 Initialization framework 52

3.7 Handle bound 52

3.8 Create events 54

3.9 Process $s_i s_j$ event 55

3.10 Process nsp_{ij} event 55

3.11 Process ord_{ij} event 55

3.12 An example of the change of moving plan 56

3.13 Handle the change of moving plan 58
3.14 Query costs v.s. cardinality of independent datasets 64
3.15 KDS overheads v.s. cardinality of independent datasets 65
3.16 Query costs v.s. cardinality of anti-correlated datasets 68
3.17 KDS overheads v.s. cardinality of anti-correlated datasets 69
3.18 Query costs v.s. non-spatial dimensionality 70
3.19 KDS overheads v.s. non-spatial dimensionality 71
3.20 Effect of update 73
3.21 Effect of speed distribution 74

4.1 Skyline query on a mobile device 79
4.2 Hybrid storage model 86
4.3 Skyline algorithm with pre-computed SK_{ns} 90
4.4 Comparison efficient on-device skyline query processing 92
4.5 Storage space costs 95
4.6 Skyline query processing time v.s. cardinality 98
4.7 Skyline query processing time v.s. dimensionality 99

5.1 Skyline query on mobile devices in a MANET 104
5.2 Dominating Region 111
5.3 Estimated Dominating Region 113
5.4 Local skyline query processing on M_i 118
5.5 Query request forwarding strategies 122
5.6 DRR on independent datasets in a static setting 124
5.7 DRR on anti-correlated datasets in a static setting 125
5.8 DRR on independent datasets in MANET simulation 127
5.9 DRR on anti-correlated datasets in MANET simulation 128

5.10 Response time on independent datasets in MANET simulation . . . 131
5.11 Response time on anti-correlated datasets in MANET simulation . . 132
5.12 Query message count 134
5.13 DRR_m on independent datasets 135
5.14 DRR_m on anti-correlated datasets 136

Summary

As a query type that is able to retrieve interesting points from a multi-dimensional dataset according to multiple criteria, skyline queries have gained considerable attention in database community in the past few years. However, so far most work on skyline queries has been accomplished in the context of static computing environments. The emergence and development of dynamic computing environments, including moving objects databases and mobile ad-hoc networks, present new stages and also challenges to skyline queries. In this thesis, we address skyline queries along three different but correlated aspects of dynamic computing environments.

First, we tackle continuous skyline queries for moving objects by taking into account the ever changing distances between a query point and points of interest. The continuously changing distances make it inefficient or even infeasible to handle a skyline query by reprocessing repeatedly. We instead turn to an incremental maintenance way that is focused on the changes of skyline. A permanent part of the skyline is identified and utilized to derive a search bound for further processing and maintenance. Then the preconditions for potential skyline changes are strictly discovered. Based on the thorough and solid analysis, we propose a kinetic-based data

structure and relevant processing algorithms for continuous skyline queries. Our proposal is targeted at the powerful server in a client/server architecture, within which the server stores all information coming from moving objects as clients. Our method does not re-compute the whole skyline query from scratch every time it changes, and it is also tolerant to moving plan updates reported by clients.

Second, we consider skyline query processing on mobile lightweight devices. We propose specific measures to efficiently process skyline queries on such resource-constrained devices. By comparing existing methods, we employ a hybrid storage scheme that deals differently with the distinct spatial coordinates and non-spatial attributes sharing duplicates. Raw coordinates are directly stored, while other attribute values are organized in linear domains and integer identifiers for domain values are stored instead of raw values. To help skyline computation, all domains and identifiers of one selected attribute are sorted. Based on the hybrid storage, we propose a skyline algorithm that executes less and more efficient value comparisons. Our hybrid storage saves storage space on device, and the skyline algorithm based on it runs faster than that without any specific measures.

Third, we address distributed skyline queries in a MANET formed by multiple mobile lightweight devices via peer-to-peer networking. To efficiently process such distributed skyline queries, we focus on cutting the data transmission time among mobile devices. We propose a filtering based query processing strategy, which identifies some unqualified points early and prevent them from being transmitted. Based on a probability model, the skyline point with the maximum capability to dominate other points is selected when the query is locally processed on its originating device. That point is called filtering point and attached to the query request sent out to other peers, where it is used to filter out unqualified points in local skylines otherwise to be sent back to the query originator. To maximize

the dominating capability the filtering point holds, it is dynamically changed on involved devices before the query is forwarded further in the MANET. This filtering base strategy reduces the amount of data to transmit, and consequently shortens the response time of distributed skyline queries.

In summary, this thesis studies three different but correlated problems of skyline queries in dynamic environments. All proposal are verified to be efficient through extensive experimental studies. At the end of this thesis, possible directions for further research are also discussed.

CHAPTER 1

Introduction

Due to the ability to model various data from structured, semi-structured to even unstructured, flexibility to support disparate architectures from strictly centralized to widely distributed, and accessibility to satisfy diverse users from amateurs to experts, databases have been applied in so many scenarios ranging from lightweight hand-held devices to supercomputers. With the wide deployment and accumulation of databases over all those years, people in modern days often need to query against databases for useful information. Such queries are composed according to the characteristics of information needed by users. Among variety of user needs, some can be modelled as ranking with respect to a single criterion. As a typical example of this category, a top-k query returns the k "best" records in the database. In a top-k query, each record is evaluated based on either a single attribute value or a scoring function output of several attribute values as inputs.

Nevertheless, more frequently user requirements on multiple dimensional data are too complex to be captured by straightforward measures like top-k ranking. Instead, users from time to time need to make decisions based on multiple criteria. To cater for such requirements, skyline query [19] has been proposed as an operator to be integrated into the existing database context.

1.1 The Concept of Skyline Query

A skyline query [19] returns a subset of *interesting* points from a large set of data points of multiple dimensionality. A point is said to be interesting if it is not *dominated* by any other points. A point pt_1 is said to dominate pt_2, if pt_1 is not worse than pt_2 in every single dimension but better than pt_2 in at least one dimension. The meaning of "better" varies in different situations, for example "smaller" or "larger" in value comparison, and "earlier" or "later" in date comparison. If a pair of points do not dominate each other, they are said to be *incomparable*.

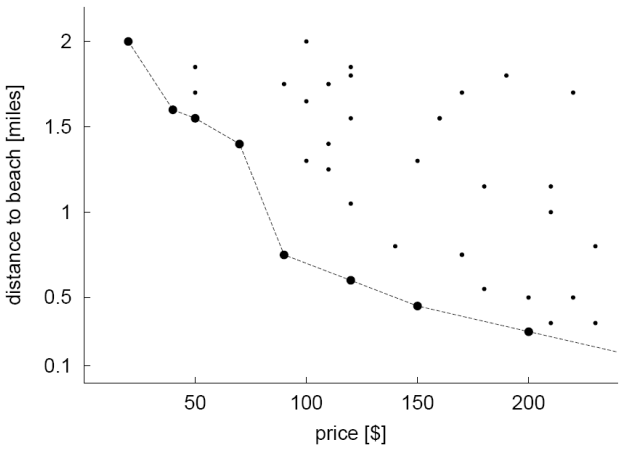


Figure 1.1: A classical example of skyline of hotels

Because of their powerful capability of retrieving interesting points from a large data set, skyline queries are well suitable for applications like decision making and optimization according to multiple criteria. Refer to Figure 1.1 from [19] for a classical example of skyline of hotels. Tourists prefer those hotels that are cheap and near to the beach, and a skyline query against the hotel sets identifies these ones on the poly-line. While a simple top-k query fails to find all these interesting ones and gives a user much less choices, e.g., either the one cheapest or the one nearest to the beach if k equals 1.

From a historical aspect of view, skyline queries can be traced to earlier topics

including contour problem [61], maximum vector [52] and convex hull [70] computations, and multi-objective optimization [81]. It also is interesting why the term "skyline" is used to name this kind of query. According to Merriam-Webster Online Dictionary [2], a skyline is "an outline (as of buildings or a mountain range) against the background of the sky". Figure 1.2 shows an example of this definition: skyline of Singapore city viewed from the sea. In this photo, only those buildings that are close to the sea or tall are visible. Here depth of field and height are two dimensions that need optimizing. Again, a top-k query is not be able to retrieve objects favorable in terms of both dimensions.

Figure 1.2: Skyline of Singapore city

Because of their power to handle multiple criteria, skyline queries has gained considerable attention in database community since its debut in [19]. Most work on skyline queries so far, however, has assumed a static centralized relational context. Whereas in this thesis we put skyline queries in dynamic environments, which are characterized by moving objects that can not be trivially accommodated by traditional databases or mobile devices that construct eccentric and challenging computing milieus. This is motivated by the emergence and popularity of such dynamic environments and the lack of research work on skyline queries in such

environments.

1.2 Motivation

We live in a dynamic world, which abounds with moves and changes. Modern computer scientists are enthusiastic about modelling this dynamic world in computers and providing efficient solutions for practical problems. Their efforts have brought about dynamic computing environments, which noticeably result from mobility related technologies. Within the past few decades, mobility related technologies have made marked progress in three dimensions.

First, positioning technology has been greatly improved in terms of both accuracy and availability based on the communication infrastructure or constellation of orbiting satellites like the global positioning system (GPS) [65], or a combination. And this trend of improvement will be pushed even further with the advent of the new positioning system Galileo [89].

Second, technology of computer hardware miniaturization has succeeded in providing variety of mobile hand-held devices with relatively acceptable computing capability. These mobile devices, including smart mobile phones and personal digital assistants (PDA), have freed users from the fixed environments of unmovable computers, making it possible to do computation anytime and anywhere.

Third, wireless communication technology has been significantly developed, so as to stimulate the worldwide upgrade of cellular networks and the prevalent deployment of IEEE 802.11-based LANs [59]. These developments have given mobile users opportunities to keep connected with the traditional fixed facilities like the Internet, and even with other mobile peers en route.

As a consequence of the confluence of all these advances, an application category

called geo-enabled mobile services [44] has surfaced with a promising prospect of convenient availability and great usefulness. As data management is still a key component of geo-enabled mobile services, database technologies undoubtedly can play a unique role in such applications. Relevant examples have been seen including moving objects databases [91], mobile databases [17], etc.

On the other hand, skyline queries have not gained enough attention in these dynamic computing environments, in spite of their suitability to multiple criteria based optimizations and decision makings which are also frequent issues in such environments. This neglect is attributed to at least two factors. One is that similar to the case in static environments, other problems like top-k or k nearest neighbors are pronounced and have attracted most efforts. The other is that the possibility of volatile values being involved in skyline queries in dynamic environments makes query processing very complex to handle, which probably retards the enrollment of active researchers. For instance, a tourist walking in a city may be interested in those hotels that are cheap and near to him. Here the distance is a dimension to be considered in the skyline query, but it is rather continuously changing than fixedly available in the database. This certainly requires specific query processing methods different from those in static settings.

The above motivation example will be more complicated if the points of interest are moving ones, like taxis, instead of static hotels. There also exist alternatives for data storage: either data are stored in a central server that is responsible for answering queries from mobile users, or data are distributed among mobile devices who collaborate on query processing or data sharing through wireless peer-to-peer communication.

Motivated by those observations above, in this thesis we carry out research on skyline queries in dynamic environments. Our research mainly covers two dy-

namic environments: moving objects databases and wireless mobile ad hoc networks (MANETs). We proceed to give an introductory presentation on the technical contributions achieved in this thesis.

1.3 Contributions

In this thesis, three skyline query problems are formalized within dynamic or mobile environments. First, we address the problem of continuous skyline queries for moving objects, where the continuously changing distances between a moving query point and other static/moving points form a particular dimension in skyline computation. Second, we address the problem of skyline queries against lightweight devices in a MANET setting, for which inter-device communication cost and intra-device computing cost are performance goals to minimize. Third, we modify the MANET setting into a hybrid mobile environment where mobile devices can contact both remote wireless server and mobile peers, and process skyline queries based on a collaborative data sharing scheme which also supports other query types. For each problem, we propose specific solution proposal and carry out extensive experimental studies to evaluate the proposal performance.

1.3.1 Continuous Skyline Queries for Moving Objects

We first formalize a continuous skyline query problem for moving objects. In a moving object context, any moving point may issue skyline queries that concern not only static dimensions but also the continuously changing distances between it and other points of interest either static or moving. As a result, the skyline result is also continuously changing as time elapses. Such continuous skyline queries involving volatile values pose a significant challenge, as most existing skyline algorithms

assume all values are constant in database for direct access, which makes them inapplicable in a moving context.

To avoid re-processing a query from scratch every time a point moves, we examine the spatiotemporal coherence existing in the problem, and propose an incremental query processing strategy accordingly. Our solution captures those spatial properties that do not change abruptly between continuous temporal scenes, and stores them in a specifically designed kinetic-based data structure [12]. Despite the changing skyline those data points that are permanently in the skyline are identified, and used to derive a search bound for further query processing. Then the connection between point locations and inter-point dominance relationship are uncovered, which implies where to find changes in the skyline and how to continuously maintain the skyline. Based on the analysis, a kinetic-based data structure is proposed, together with an efficient skyline query processing algorithm. We concisely analyze the space and time costs of the proposed method. Update issue on moving objects and its impact on our proposal is also addressed. Extensive experimental studies are conducted to evaluate the proposal performance.

1.3.2 Skyline Queries on Mobile Lightweight Devices

In step with the continued advances of electronics miniaturization, mobile lightweight devices like personal digital assistants (PDAs) and smart mobile phones are being increasingly popular. Equipped with such devices storing relevant data, mobile users may issue local queries to learn about their geographic surroundings. Skyline queries, for their ability in retrieving interesting points according to multiple criteria, are unsurprisingly of interest on such devices.

Transplanting the existing skyline algorithms directly into a lightweight mobile device is unlikely efficient, as computing resources including storage space and

processor power is considerably constrained on such devices. To speed up the on-device skyline query processing, we propose a hybrid storage of spatial data points. This storage handles spatial coordinates and other attributes in different specific ways. Coordinates are stored in raw format, while others are stored in a way that sorts value domains and keeps corresponding integer indexes in the storage instead of the raw values. We also pick one attribute to sort all integer indexes stored. This hybrid storage helps save space, and speed up the local skyline computation because of the sorting order and the representative integers.

Based on the hybrid storage scheme, we also propose corresponding skyline algorithm suitable for lightweight devices. We implement our hybrid storage scheme and on-device skyline computation on a real HP iPAQ pocket PC. We compare our proposal with other alternatives, and the results show that ours is more efficient.

1.3.3 Skyline Queries Against Mobile Lightweight Devices in MANETs

We next formalize another skyline query problem by going into a wireless mobile environment and allowing mobile devices to issue skyline queries against peers. In other words, we use wireless mobile ad hoc networks (MANETs) as the physical environment for this problem. Every mobile device is resource-constrained, i.e., equipped with limited storage space and computing capacity. Each device holds only a portion of the whole dataset, which consists of both geographic coordinates and other attributes. But devices can communicate with other peers through the MANET in which the connections are not so reliable and fast as those in wired situation. A query issued by a mobile device is attached with spatial constraints indicating the distance of interest, within which points in the skyline are expected. Because the whole dataset is distributed among all mobile devices, such a query

involves data from different peer devices.

The main challenges of the MANET environment is its relatively slow and unsteady wireless communication channels between mobile devices. This requires less amount of data to be transferred among mobile devices through a MANET, as well as efficient processing on any single device, the problem we have posed in Section 1.3.2. Therefore, our research goal on this problem is to find in a MANET some efficient distributed skyline query processing strategy that saves data communication.

To cut the inter-device communication costs, we propose a filtering based distributed query processing method. On the device issuing a skyline query, we choose from the initial local skyline a filtering point with the maximum estimated ability to dominate other points, and attach it to the query request sent out. On other devices, this filtering point is used to eliminate unqualified candidates that are transmitted otherwise.

We carry out extensive experiments to evaluate the performance of our proposal. We simulate the whole system proposal using a MANET simulator, JiST-SWANS [1]. The simulation results have confirmed the efficiency of our proposal, as our filtering based strategy incurs less data transmission and shorter query response time.

1.4 Organization

The thesis is organized as follows:

- In Chapter 2, we describe the background of the research work presented in this thesis, which covers three relevant research areas. The first area unsurprisingly is skyline queries, for which we mainly cover previous query process-

ing algorithms. The second area is moving objects databases, for which we mainly cover indexing and query processing techniques pertinent to our first problem. The third area is mobile peer-to-peer (P2P) and wireless mobile ad hoc networks (MANETs) which constitute the setting of our second and third problems.

- In Chapter 3, we formalize the continuous skyline query problem for moving objects, which involves both static dimensions and the continuously changing distance. We propose an incremental query processing strategy that does not re-process the query from scratch every time a point moves. Extensive experimental studies are also conducted.

- In Chapter 4, we shrink the environment into resource-constrained mobile lightweight devices and consider skyline queries on a single device of that kind. To speed up the on-device query processing, we propose a specific hybrid storage and a relevant algorithm. We experimentally compare our methods with others on a real pocket PC.

- In Chapter 5, we alter the previous problem by allowing mobile devices to communicate via a MANET and issue distribute skyline queries against peers in the MANET. To cut the communication costs via the relatively slow and unsteady MANET, we propose a filtering based distributed query processing method. We also carry out extensive experiments using a MANET simulator.

- We conclude this thesis in Chapter 6, which summarizes the contributions and limitations of our proposals in this work, and discusses some possible directions for future research.

Two research papers have been accepted for publication or published from the work presented in this thesis. The work on continuous skyline queries for moving

objects, presented in Chapter 3, has been accepted by TKDE for publication [42]. The work on skyline queries against mobile lightweight devices in MANETs, presented in Chapter 5 and part of Chapter 4, has been published in the proceedings of ICDE [41].

1.5 An Overall Picture

Our three main contributions are achieved by solving three problems that are different but correlated in some way. In this section we relate them together into an overall picture, which is believed to be helpful for readers to understand better the work done in this thesis.

The correlations between contributions are illustrated in Figure 1.3. All three problems/contributions are solved/accomplished within a big dynamic/mobile computing environment. Client/server architecture is an important paradigm in mobile computing environment [47]. It is also employed in moving objects databases, where mobile entities like pedestrians, vehicles act as clients and update their positions or/and movement information to a central server by sending appropriate messages. The central server is responsible for storing moving objects information in databases and processing queries in relation to those moving objects. Our first research problem, presented in Chapter 3, falls into this category. A powerful central server is assumed to process continuous skyline queries in relation to the moving objects, whose information is stored on the server together with static points of interest if applicable. Our query processing solution is focused on the server and does not involve any client side computing capability. Nevertheless, the query point can be a mobile client and points of interest can be other mobile clients.

Different from relying on the central server in a client/server system, mobile

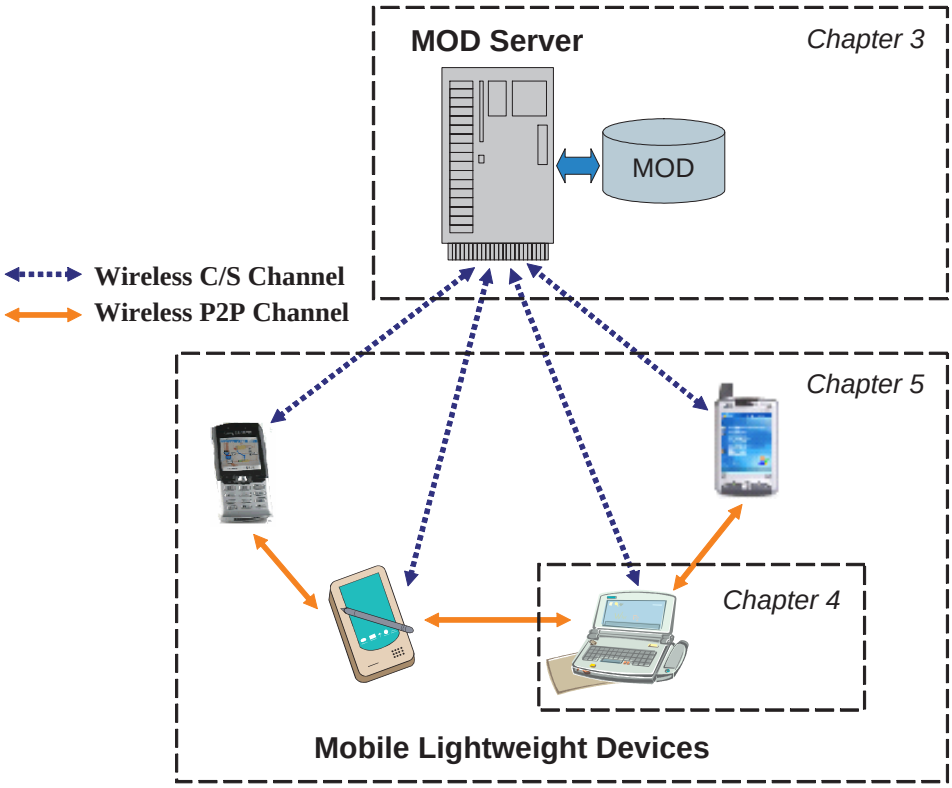


Figure 1.3: An overall picture of this thesis

devices themselves are also able to provide computing services to some extent. Our second research problem, presented in Chapter 4, falls into this category. By a specific storage scheme, we carefully store static points of interest on a resource-limited mobile device. Then we accordingly propose algorithms to process snapshot skyline queries locally on a device. Using only a mobile lightweight device without contacting the central server, a mobile user is enabled to issue skyline queries in relation to her/his surroundings.

Equipped with wireless peer-to-peer networking interfaces like infrared, Bluetoothe or even Wi-Fi, mobile devices can constitute ad-hoc networks. Within a MANET of this kind, any device can query against not only itself but also other peers reachable. Our third research problem, presented in Chapter 5, falls into this category. With points of interest properly stored on different mobile lightweight

devices in a MANET, we propose a distributed skyline query strategy that is efficient in terms of data transmission. This fashion, still without a central server, extends the capability of any single device in a mobile environment.

The three problems addressed in this thesis represent three different paradigms in dynamic or mobile computing environments. For each paradigm, we accordingly propose our solution for the specific research problem. Our proposals provide indications and choices for users who face their own practical situations.

CHAPTER 2

Background

In this chapter, we describe the background of this thesis. We present a comprehensive review on previous work that is relevant to ours presented in this thesis. As this thesis is targeted at skyline queries in dynamic environments, our related work comes from three significant aspects: skyline queries, continuous queries in moving objects databases, and wireless mobile ad-hoc networks (MANETs).

2.1 Skyline Queries

Skyline, as a new query type, was first introduced into database community by Börzönyi et al. [19], whereas its origin and similar precedents can be found in other areas different from database. Such examples includes some earlier topics: the contour problem [61], maximum vector [52] and convex hull [70] computations, and multi-objective optimization [81]. A skyline query is closest to maximum vector and multi-objective optimization, because all of them aim to find the "best" ones from a set of multiple-dimensional points according to some criteria directly involving more than a single dimension. Here we use "directly" to indicate that comparisons are carried out between values on a given dimension, instead of between values obtained using a function of several dimensions. On the other hand, a skyline

query differs from a convex hull as its result is not necessarily convex, which is illustrated in Figure 1.1. A skyline is not closed, which also distinguishes it from contour and convex hull.

In the remainder of this section, we mainly review relevant work on skyline query processing algorithms, which regard disk I/O or/and CPU time as the most important performance factors. The computing environments include centralized relational storage, data streams, and distributed settings, with the emphasis on centralized environments. We also cover work on estimation of skyline cardinality.

2.1.1 Skyline Queries in Centralized Environments

Börzönyi et al. [19] for the first time introduced the skyline as an operator into database systems. They gave the definition of a skyline query within the relational setting, and extended the SQL SELECT statement with an optional SKYLINE OF clause in the following way:

SELECT ... FROM ... WHERE ...

GROUP BY ... HAVING ...

SKYLINE OF [DISTINCT] d_1 [MIN | MAX | DIFF], ..., d_m [MIN | MAX | DIFF]

ORDER BY ...

In the statement, annotation MIN (MAX) on dimension d_i means smaller (larger) values on d_i are preferred in a skyline query, e.g., cheap hotel prices (number of stars a hotel has). Annotation DIFF on dimension d_i means no identical values on d_i are needed in a skyline query. Besides, the option DISTINCT is used to eliminate possible duplicates in the skyline. Throughout this thesis, we use the MIN annotation, i.e., smaller values are preferred in skyline computation.

The authors also proposed in that work two skyline query processing algorithms: *Block Nested Loop* (BNL) and *Divide-and-Conquer* (D&C). BNL is a straightfor-

ward approach that compares each pair of points in an iterative way. It sequentially scans the data relation on the disk and keeps a *window* of skyline candidates in memory. Initially the first point is put into the window. Then each subsequent point p is compared to every candidate in the *window* to check the dominance relationship. If p is dominated by a candidate, it is eliminated and will not be visited again. If p dominates one or more candidates, it is inserted into the window and all those candidates it dominates are deleted. Otherwise, p is inserted into the window. If the memory can not hold all the candidates, any new p to be inserted into *window* is written into a temporary disk file, which will be loaded into memory for processing in next iterations. Two variants of BNL are proposed. One keeps the window as a self-organizing list that automatically moves each point found dominating other points to the beginning of the window. The other is used for constrained memory situation, where BNL has to be executed for more than one pass and the most dominant candidates are kept in the memory according to some metric-based replacement policy. The D&C approach divides the whole dataset into several partitions each of which fits in memory. Then for each partition a local skyline is computed. The final skyline is obtained by correctly merging the local skylines.

Chomicki et al. [29] proposed an algorithm named *Sort-Filter-Skyline* (SFS) as a variant of BNL. SFS requires the dataset to be pre-sorted according to some monotone scoring function before the skyline computation. And then during the SFS algorithm, any point inserted into the *window* is ensured to be a skyline point and no removal operations are invoked on the *window*.

Tan et al. [82] proposed two progressive processing algorithms: *Bitmap* and *Index*. In Bitmap approach, for each point $x = (x_1, ..., x_d)$ in the range of [0,1], any of its x_i is represented by k_i bits where k_i is the number of distinct values on the ith

dimension in the whole dataset. If x_i is the qth distinct value on dimension i, bits 1 to q-1 are set to 0 and the remaining bits 1. In this way, point x is represented by a m-bit vector where $m = \sum_{i=1}^{d} k_i$. To check if a point x is in the skyline, a specific column is retrieved from each dimension's bit matrix. After that, each of those d bit columns is treated as a bit string and the *and* operation is applied to them all. Point x is in the skyline if and only if the resultant bit string contains only one 1. In Index approach, each point x is mapped into a single dimensional space by formula $y = d_{max} + x_{max}$ where x_{max} is the largest value among all dimensions of x, and d_{max} is the corresponding dimension. After the transformation a B^+-tree is used to index all the transformed values. Thus, all points in the dataset are also partitioned into d parts in such a way that point x is put into partition d_{max}. To compute the skyline progressively, the algorithm processes all points in different batches. Suppose that $m_1 > m_2 > ... > m_k$ are all distinct values on all dimension of the whole dataset. Then all these points are processed in k batches. Initially the algorithm gets from each dimension partition all those points with a maximum dimension value being m_1. Next a skyline algorithm is executed on this batch of points to get the local skyline. All points in the local skyline are inserted into the final skyline. Then the algorithm repeats the same steps for batches from m_2 to m_k, and each local skyline is merged to the final skyline with ineligible points correctly excluded. Within each dimension partition, locating points for the current batch is facilitated by the B^+-tree.

Kossmann et al. [51] proposed a *Nearest Neighbor* (NN) method to process skyline queries progressively. It first carries out a depth-first NN search [73] on the dataset indexed by an R*-tree [13], and then inserts the NN point into the skyline. The NN point also determines a region within which all those points are dominated by it. That region is pruned from subsequent processing. The rest

part of the dataset is partitioned into two parts based on the NN point, and both are inserted into a *to-do* list for further processing. Then the algorithm repeats removing a part from the *to-do* list and processing it recursively, until the list is empty. However, in each partitioning step different parts may overlap and the overlapping region might cause duplicates in the skyline result. This is a crucial factor that must be correctly considered in the NN method. To deal with this problem, several alternatives are proposed for the NN method to incorporate. Any of those alternatives either eliminates the duplicates after they happen or prevents them before they happen. Tradeoffs of these alternative ways are also discussed.

Lu et al. [58] proposed an IO optimal divide-and-conquer algorithm for 2D skyline queries. Their algorithm improves the NN algorithm [51], mainly by refining the search regions when recursive partitioning is carried out after a NN point is found. However, this contribution is not significant as the new algorithm applies to 2D skyline queries only.

Papadias et al. [66, 67] proposed another progressive algorithm named *Branch-and-Bound Skyline* (BBS) which is based on the best-first nearest neighbor (BF-NN) algorithm [38]. It initially enqueues all the entries of the R*-tree root into a priority heap that prioritizes entries based on their *mindist*s in a non-descending manner. *mindist* is computed according to L_1 distance, i.e., the *mindist* of a point is the sum of its coordinates and that of a MBR is the *mindist* of its minimum corner (e.g., bottom-left point in the 2-dimensional cases). Then the entry e on the heap top is dequeued for processing. It is discarded if it is dominated by some existing skyline point. Otherwise, it is either expanded with its entries inserted into the heap if it is an intermediate node, or inserted into the skyline if it is a point. This procedure is repeated until the heap is empty which indicates the whole dataset has been processed. A difference between BBS and BF-NN is that BBS

only enqueues into the heap those entries that are not dominated by any point in the current skyline list.

Godfrey et al. [34] provided a comparative analysis of previous maximal vector computation algorithms. It mainly compares theoretical algorithms on maximal vector problem, with those ones that compute skylines in a relational context without indexing supports. Algorithms are regarded as either divide-and-conquer or scan-based based on their computational nature. After a comprehensive analysis, the authors claimed that scan-based skyline algorithms outperform divide-and-conquer ones. Then they proposed a new hybrid algorithm that combines different scan-based skyline algorithms.

The related work aforementioned all considers skyline queries concerning attributes whose domains are totally ordered. As a matter of fact, attribute domains can be partially ordered. Such domains may include intervals and incomparable set elements. Motivated by this, Chan et al. [23] studied the problem of processing skyline queries with partially-ordered domains. Due to the lack of a total order on relevant attributes, previous index-based skyline algorithms such as Index [82], NN [51], and BBS [66] is no longer able to prune search space effectively. The authors proposed a solution that transforms every partially-ordered domain into a closed integer range, which makes it possible to use index-based algorithms on the transformed space. Then the authors proposed three algorithms: BBS$^+$, SDC and SDC$^+$. BBS$^+$ straightforwardly adapts BBS into the proposed transformation framework. While SDC and SDC$^+$ exploit the dominance relationship captured from integer ranges to organize the data into strata, and are optimized to reduce false positives and support progressive report. They differ in that SDC generates its strata on the fly, whereas SDC$^+$ does so offline.

Within the context of a spatial road network, Huang and Jensen [40] proposed a

in-route skyline query to be incorporated in location-based services. When moving along a pre-defined road route towards her/his destination, a user may visit points of interest in the network. Selection of points to visit is made in terms of multiple distance-related preferences like detour and total travelling distance. To optimize such kind of selections by skyline querying, specific algorithms are proposed based on disk-resident network data.

Sharifzadeh and Shahabi [77] defined a special skyline query in spatial databases. Given a set of query points $Q = \{q_1, \ldots, q_n\}$ and two point p and p', p is said to spatially dominates p' iff $dist(p, q_i) \leq dist(p', q_i)$ for any $q_i \in Q$ and $dist(p, q_i) < dist(p', q_i)$ for at least one $q_i \in Q$. With this definition, *spatial skyline* of a set of points P is defined as its subset containing all points that are not spatially dominated by any other point of P. To efficiently process such spatial skyline queries (SSQ), the authors proposed two algorithms, both of which employ the R-tree oriented best-first search framework of BBS algorithm [66], but adopt computational geometry knowledge to help processing. The first algorithm, called B^2S^2, takes advantage of convex hull to help skyline points determination. The second algorithm, called VS^2, also utilizes Voronoi diagram and Delaunay Graph [30] in addition to convex hull.

Our first problem on continuous skyline query (CSQ), presented in Chapter 3, differs from the spatial skyline problem in several ways. First, there is only one query point in our problem while a SSQ problem has multiple query points which are necessary for the skyline definition it uses. Note our problem is not a special case of SSQ, as for SSQ a sole query point makes it degrade to a nearest neighbor query instead of a real skyline query like ours. Second, spatial distance is the only factor considered in SSQ for most of the time, though the authors also give brief hints on how to include other attributes in SSQ. Whereas, CSQ takes into

account not only changing distances but also non-spatial attributes. Third, our CSQ solution is applicable to both static and moving set of data points, while both B^2S^2 and VS^2 algorithms apply to static data point set P.

2.1.2 Variants and Derivatives of Centralized Skyline Queries

In this section, we review a particular portion of existing work that is focused on problems of variants or derivatives of skyline queries in a centralized manner. Directly using previous skyline algorithms either cannot solve such problems or cannot solve them with acceptable efficiency.

I. Dominance Variants

Usually, the dominance relationship used in skyline computation is defined for a pair of data points. And this dominance relationship is transitive [19]. Rather than evaluating every single data point, Jin et al. [46] extended the usual skyline, called *thin skyline* by the authors, to a new concept named *thick skyline*. The thick skyline includes not only those skyline points returned by the usual skyline definition, but also their neighboring points within ε-distance. Three different approaches for thick skyline computation were proposed, based on statistics, indexes and clustering means respectively.

Koltun and Papadimitriou [49] introduced approximately dominating representatives to reduce skyline query result at the cost of slight accuracy loss. The approximation lies in that before a point is considered in the dominance relationship with others, it is first boosted by ϵ in all dimensions. They proposed for 2-dimensional datasets a linear algorithm using traditional skyline query result as input. And for 3-dimensional datasets, they proved the problem is NP-complete. This work is intended to theoretically generalize the skyline definition, based on an algorithmic standpoint.

II. Skyline Cube

Given a multi-dimensional dataset, a skyline query is usually processed by taking all dimensions into consideration. Actually, skyline query can be asked concerning any possible combination of individual dimensions.

Yuan et al. [98] addressed the problem of computing all skylines of all possible non-empty subsets of a given set of full dimensions. They called all these skylines *Skyline Cube* or *Skycube* for short. When computing for skylines for different subspaces, intermediate computation or partial results can be shared to reduce computation costs. Motivated by this observation, the authors proposed *Bottom-Up* and *Top-Down* algorithms to efficiently compute the skycube. For either algorithm, specific strategies are utilized to share different things like result and sorting or partitioning operations.

Pei et al. [68] addressed the same problem with a different approach. They analyzed the semantics of points' common membership in the skylines of subspaces, and further the structures of such subspace skylines. Based on the observations gained in the semantic and structure analysis, the authors proposed a top-down depth-first search framework to compute subspace skylines.

Tao et al. [87] proposed a technique that facilitates subspace skyline computation with B-tree. An anchor is defined as the maximal corner of the original space, and each point is converted to a value, the maximal difference of the anchor coordinate and the point coordinate on all dimensions in the full space. This converted value is used to easily determine if that point is in the skyline of a given subspace, with a B-tree indexing all points in terms of their converted values. Furthermore, the authors also discussed how to pick different and more efficient anchors in different point clusters, together with pruning algorithms with multiple anchors.

Xia and Zhang [93] discussed how to efficiently update the skycube in databases

facing concurrent, frequent and unpredictable updates. The core of their solution is a novel structure called *compressed skycube*, which represents the skycube in a concise but complete manner. Besides, a buffer is used to store the most frequently asked query results. Database updates are then modelled as incremental object-aware updates to the compressed skycube, which allows scalable updates. And query processing is facilitated by taking advantage of the query buffer. Thus, the overall query cost and update cost is balanced.

Based on the dominance relationship in skyline computation, Li et al. [53] introduced a new kind of analysis called *Dominant Relationship Analysis* (DRA). DRA aims to disclose the dominant relationship between products and potential consumers, and thus helping to make good business strategies. To efficiently answer different analysis queries of DRA, the authors proposed a data cube based structure, named *DADA*, which stores the dominant relationships in the way supporting ordered access and compressing.

III. Skyline Queries against High Dimensionality

Though skyline queries are powerful in retrieving interesting data according to multiple criteria, they are also hurt by the "dimensionality curse". For a dataset in a considerably high dimensional space, the size of its skyline can be very huge especially when the dataset itself is large. Research endeavors have been made to alleviate this negative effect.

Zhang et al. [100] proposed the concept of strong skyline points that frequently appear in small-sized subspace skylines in a high dimensional space. As such points are much less than the skyline of the full space, retrieving these strong skyline points does not cause the result explosion problem. Two search algorithms, depth-first and breadth-first, were proposed to search all subspaces for such interesting points.

By counting the frequency of points' membership in subspace skylines, Chan

et al. [25] converted the skyline query into a top-k ranking problem which gives priority to those points that appear in more subspace skylines than others. Such points, with their high skyline frequency, are interesting for analysis purpose in high dimensional spaces, as they are less than the full space skyline points but still hold some preferable values. Approximate algorithms were proposed to efficiently search the high dimensional subspaces for those points with reduced computational complexity.

Furthermore, Chan et al. [24] proposed the concept of k-dominant skyline for high dimensional spaces. The strict dominance regarding all dimensions is relaxed to only k dimensions in any subspace. A point p is said to *k-dominate* another point p' if p is better than or equal to p' and is better in at least one of these k dimensions. This k-dominant relationship is not transitive and also can lead to cycles, depending on the different k values. A new type of query called top-δ dominant skyline query was proposed, to decide the smallest k that produces more than δ k-dominant skyline points.

2.1.3 Skyline Queries on Data Streams

Compared to traditional databases, data streams [9] are very special because of their high arrival speed, continuous updates and in-memory processing. These characteristics make conventional skyline algorithms unsuitable for stream environments.

Lin et al. [54] proposed an efficient skyline computation method over sliding window data stream model, which keeps the most recent N elements only. Their goal is to compute the skyline for the most recent n ($\forall n \leq N$) elements. Their solution consists of three main parts. First, a pruning technique is used to identify and discard uninteresting elements from the memory. Second, a graph based

encoding scheme is proposed for elements in the memory, which allows fast skyline computation based on the encoding scheme. Third, a trigger based incremental algorithm is proposed to efficiently process continuous skyline queries over sliding windows.

Tao and Papadias [85] addressed the similar problem with different approaches. In their proposals, both query processing time cost and memory space overhead are targets to minimize. Two frameworks for tracking skylines over data streams were proposed. The first one delays most computations until some existing skyline point expires, which reduces the processing cost. The second one instead pre-computes some results by predicting future skyline changes, which minimizes the memory consumption.

2.1.4 Skyline Queries in Distributed Environments

In a distributed context, Balke et al. [10] addressed skyline operation over web databases where different dimensions are stored in different data sites. Their algorithm first retrieves values in every dimension from remote data sites using sorted access in round-robin on all dimensions. This continues until all dimension values of an object, called the terminating object, have been retrieved. Then all non-skyline objects will be filtered from all those objects with at least one dimension value retrieved.

For the same problem, Lo et al. [57] extended the previous solution with several modifications. By assuming that for any pair of points, they do not share duplicate value on any single dimension, the authors made the skyline point reporting progressive. To speed up dominance check, a memory resident R*-tree was used to store objects retrieved from remote sites. A heuristic was proposed to predict the terminating object that ends the data retrieval.

Our third problem on distributed skyline queries in a MANET, presented in Chapter 5, targets horizontal dataset partitions rather than vertical partitions in the work aforementioned.

Recently, Wu et al. [92] proposed a parallel execution of constrained skyline queries in a shared nothing distributed environment. By using the query range to recursively partition the data region on every data site involved, and encoding each involved (sub-)region dynamically, their method avoids accessing sites not containing potential skyline points and progressively reports correct skyline points. However, a overlay network is assumed to be available to ensure the correctness and efficiency of their method.

2.1.5 Skyline Cardinality Estimation

The size of skyline result, or skyline cardinality, is of interest for query optimizations in a relational database engine.

For the maximal vector problem [52], Bentley et al. [16] and Buchta [21] worked out upper-bounds for the average number of maxima in a set of vectors. Their results were gained with the assumption that all dimensions are totally ordered independent.

Godfrey [33] addressed the problem in the context of relational databases. His basic model requires two assumptions: no duplicate values exist on any dimension, and all dimensions are independent. Then, the basic model was generalized to deal with dense domains, domains following Zipfian distribution [101], and correlated and anti-correlated dimensions.

Chaudhuri et al. [27] proposed a uniform random sampling based probability model to estimate the skyline cardinality. Their model does not require dimension value independence assumption, and works for both totally ordered attribute

domains and categorical attribute domains.

2.2 Continuous Queries in Relation to Moving Objects

In this section, we briefly review existing work on continuous queries in relation to moving objects. For a comprehensive view of moving objects databases or spatial-temporal databases, readers would be directed to an early paper [91] and a panel report [72].

In the context of moving objects, both objects of interest and queries can be moving. The continuity movement makes it interesting to issue continuous queries [78] on moving objects. Such a continuous query either requires results for a period of time specified or keeps active until it is deactivated explicitly. Various techniques for efficient continuous query processing have been proposed.

Song and Roussopoulos [80] first studied k nearest neighbor search for a moving query point. The query moves on a predefined line, while all points of interest are static and indexed by a R-tree [36]. The continuous query is processed in a periodic sampling fashion: at each sampled position of the query point, the current correct k nearest points are returned by searching the R-tree. To reduce processing costs, different methods were used including limiting search distance, reusing previous results, and pre-computing results for future query positions. However, their approaches are very sensitive to the sampling rate. A low sampling rate can cause incorrect results while a high sampling rate can incur considerable processing costs.

To remedy this drawback, Tao et al. [86] proposed an approach that processes a continuous NN query in a single-pass and returns a set of $<point, interval>$ tuples as the result. When the query point is moving in the line segment determined

by *interval*, its NN is *point*. Their approach is based on a careful analysis on the geometry characteristics of the problem, which indicates the query positions where the NN changes.

Prabhakar et al. [69] proposed two techniques for efficient processing multiple continuous queries on moving objects. The first one, Query Indexing (QI), regards queries as data and indexes them using R-tree. Each object's safe region, a region within which an object moves without affecting any query, is computed by searching the query index. The second one, Velocity Constrained Indexing (VCI), assumes a maximum speed exists for each moving object, and stores those maximum speeds in the traditional index for moving objects. VCI itself does not facilitate query processing, but it can be combined with QI to accommodate query insertions and deletions in addition to active queries.

The idea of indexing queries was also employed by Mokbel et al. [62] to support scalable incremental processing of continuous queries in spatial-temporal databases. With grid based hashing for both moving objects and moving queries, concurrent queries in their work are executed in an incremental and shared manner: query results are gained by hash joining of queries and objects. Similar approaches were used by Xiong et al. [94] focusing on continuous kNN queries.

Gedik and Liu [32] proposed a mobile system architecture to process continuous moving monitoring queries on moving objects in a distributed way. In their proposal, queries with information of regions and additional predicates are installed on moving clients, where such information is used to delay updates to server and process queries locally. In this way, the server load is considerably reduced. Several optimizations like lazy query propagation, query grouping and safe periods were also introduced to reduce the computational costs on mobile clients.

The ideas of safe region and shifting load from server to clients were also applied

by Hu et al. [39] to monitor continuous spatial queries over moving objects. By sending safe regions to mobile clients in their proposal, location updates to the server via wireless communication are saved until they affect relevant queries.

Besides those proposals aforementioned, kinetic data structure [12] and its underlying ideas also have inspired some continuous query processing techniques that utilize events to maintain query results in moving objects database.

Mokhtar et al. [63] proposed an event-driven approach to maintain the results of k-NN queries on moving objects while time elapses. Their approach starts with a list of all moving objects sorted by their current distance to the query point. Then the events indicating when a moving object will change position in the list with its neighbor are computed based on the motion plans. The problem of maintaining k-NN query results is transformed into the problem of maintaining the list of moving objects. As time progresses, events are processed in time order and the order of the moving objects is maintained; this allows k-NN query results to be always available in the object list.

Instead of keeping all moving objects in ascending order of distance to query point, Iwerks et al. [43] presented another event-driven method to maintain continuous k-NN queries on moving objects. Based on the fact that window queries are cheaper to maintain on moving objects than k-NN queries, the authors proposed the Continuous Windowing (CW) k-NN algorithm. The CW k-NN algorithm first gets all those objects within a specific distance d around the query point. If at least k objects are found, all the final k nearest neighbors must be among these objects, and only they need to be checked. Otherwise, the search is extended outwards with the distance d adjusted. Here, events indicating when and which objects will move into the distance d around the query point are computed first and processed gradually to maintain the query result during the life time of the query.

2.3 Data Management in MANETs

In this section, we first give a brief introduction to the mobile ad-hoc networks (MANETs), and then review some work relevant to data management in such mobile environments.

2.3.1 Mobile Ad-Hoc Networks

Recent years have witnessed increasing variety of mobile handsets being equipped with wireless peer-to-peer (P2P) networking capabilities. This enables mobile handsets to become parts of self-organizing, wireless mobile ad hoc networks, or MANETs for short. Such MANETs allow seamless, inexpensive, and easily deployed communications [11, 20].

Schollmeier et al. [74] proposed a protocol framework based on current Internet protocols for P2P networking in the mobile environment. An inter-layer communication protocol named Mobile Peer Control Protocol is used to bridge the network layer and the application layer, which are coordinated to operate with minimum routing cost.

For a quality presentation of the state of the art on wireless mobile ad hoc networking, readers are referred to a book [11], which covers extensive relevant topics including location discovery, routing protocols, energy saving, security issues etc.

2.3.2 Data Management in Mobile Ad-Hoc Networks

Charas [26] reviewed the history of wireless mobile networks with an emphasis on system architecture, and claimed that mobile P2P would be the ultimate business model in mobile network. A new architecture was proposed, with conceptual

descriptions for each component inside.

Kortuem et al. [50] describe scenarios where mobile devices can collaborate to exchange information when they encounter each other. They also discuss challenges for such mobile ad hoc information systems based on a prototype mobile peer-to-peer platform named Proem.

Budiarto et al. [22] mainly discuss strategies for data replication in a mobile P2P environment, which is modelled by a hierarchy with wireless cells at the bottom and fixed networks at the top. Master databases of mobile peers are replicated in the fixed network for other peers to access.

Xu et al. [96] covered systemic topics on data management in mobile P2P networks. A data model and a data dissemination approach specific to mobile peers were proposed that include security and transaction mechanisms. These ideas have been applied to disseminate spatio-temporal resource information in mobile P2P networks, where a mobile object exchanges information with peer objects it encounters [90].

Lindemann et al. [56] proposed a distributed document search service named Passive Distributed Indexing (PDI) for applications in mobile ad hoc networks. Their proposal confines the wireless broadcast transmission of query requests and response messages to a limited region instead of globally, and utilizes local storage to cache results on each mobile device. With these optimizations, PDI is able to avoid flooding messages throughout the network and improve the hit rates of searches.

Based on a review of previous technologies, Fife and Gruenwald [31] categorized different research issues of interest for data communication in MANETs.

CHAPTER 3

Continuous Skyline Queries for Moving Objects

The literature on skyline algorithms has so far dealt mainly with queries of static query points over static datasets. With the increasing number of mobile service applications and users, however, the need for continuous skyline query processing has become more pressing. A continuous skyline query involves not only static dimensions but also the dynamic one. In this chapter, we examine the spatiotemporal coherence of the problem and propose a continuous skyline query processing strategy for moving query points. First, we distinguish the data points that are permanently in the skyline and use them to derive a search bound. Second, we investigate the connection between the spatial positions of data points and their dominance relationship, which provides an indication of where to find changes in the skyline and how to maintain the skyline continuously. Based on the analysis, we propose a kinetic-based data structure and an efficient skyline query processing algorithm. We concisely analyze the space and time costs of the proposed method and conduct an extensive experiment to evaluate the method.

3.1 Introduction

With rapid advances in electronics miniaturization, wireless communication and positioning technologies, the acquisition and transmission of spatiotemporal data using mobile devices are becoming pervasive. This fuels the demand for location-based services (LBS) [14, 75, 95, 99]. A skyline query retrieves from a given dataset a subset of interesting points that are not dominated by any other points [19]. Skyline queries are an important operator of LBS. For example, mobile users could be interested in restaurants that are near, reasonable in pricing, and provide good food, service and view. Skyline query results are based on the current location of the user, which changes continuously as the user moves.

The existing work on skyline queries assumes a static setting, where the distances from the query point to the data points do not change. Refer to the example of skyline in a static scenario shown in Figure 3.1. Assume there is a set of hotels, and for each hotel, we have its distance to the beach (x *axis*) and its price (y *axis*). The interesting hotels are all the points not worse than any other point in both the distance to the beach and the price. Hotels 2, 4 and 6 are interesting and can be derived by a skyline query, for their distances to the beach and their prices are preferable to those of any other hotels. Note that a point with minimum value in any dimension is a skyline point – hotels 2 and 6 for example. Also, skyline is different from convex hull in that it is not necessarily convex. In this example, hotel 4 makes the skyline not convex.

In the above query, the skyline is obtained with respect to a static query point; in this case, it is the origin of both axes. Now, let us change the example to the scenario of a tourist walking about to choose a restaurant for dinner. We consider three factors in the skyline operation, namely the distance to the restaurant, the average price of the food and the restaurant rank. Different from the previous

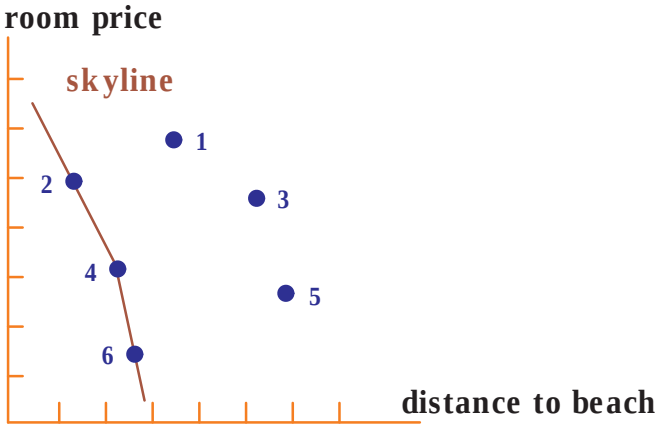
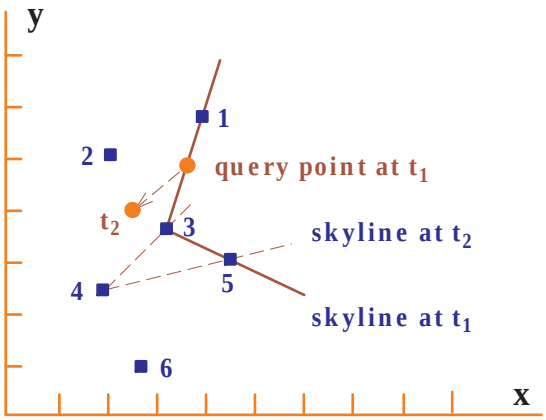


Figure 3.1: An example of skyline in a static scenario

example, the distance now is not fixed since the tourist is a moving object. Figure 3.2 shows the changes in the skyline due to the movement. In the figure, the positions of the restaurants are drawn in the X-Y plane while the table shows their prices and ranks. Lower values are preferred for all three dimensions. A tourist as the query point moves as the arrow indicates from time t_1 to t_2. The skyline – which refers to the interesting restaurants – changes with respect to the tourist's position. Skylines at different times are indicated by different line chains. The situation becomes more complex when all data points can move, which is frequent in real-time applications like e-games and digital war systems. For instance, one player in a field fighting game wants to keep track of those enemies who are close and most dangerous in terms of multiple aspects like energy, weapon, strategy and etc.

In this chapter, we address the problem of continuous skyline query processing, where the skyline query point is a moving object and the skyline changes continuously due to the query point's movement. We solve the problem by exploiting its spatiotemporal coherence. Coherence refers to properties that change in a relevant way from one part to other parts within a scene in computer graphics [37], which is



Restaurant	Price	Rank
1	6	4
2	5.8	4
3	4	1
4	2.8	3
5	1	2
6	2	4

Figure 3.2: Skylines in mobile environment

used to build efficient incremental processing for operations such as area filling and face detection. We use spatiotemporal coherence to refer to those spatial properties that do not change abruptly between continuous temporal scenes. The positions and velocities of moving points do not change by leaps between continuous temporal scenes, which enables us to maintain the changing skyline incrementally. First, we distinguish the data points that are permanently in the skyline and use them to derive a search bound to constrain the processing of the continuous skyline query. Second, we investigate the connection between the spatial locations of data points and their dominance relationship, which provides an indication of where to find changes in the skyline and update it. Third, to efficiently support the processing of continuous skyline queries, we propose a kinetic-based data structure and the associated efficient query processing algorithm. We present concise space and time cost analysis of the proposed method. We also report on an extensive experimental study, which includes a comparison of our proposed method with an existing method adapted for the application. The results show that our proposed method is efficient in terms of storage space, and is especially suited for continuous skyline queries. To the best of our knowledge, this is the first work on continuous skyline

queries in the mobile environment.

The rest of this chapter is organized as follows. In Section 3.2, we present the preliminaries including our problem statement and a brief review of related work. In Section 3.3, we present a detailed analysis of the problem. In Section 3.4, we propose our solution which continuously maintains the skyline for moving query points through efficient update. In Section 3.5, we present cost analysis and discussion on our method. The experimental results are presented in Section 3.6. Section 3.7 summarizes the chapter.

3.2 Preliminaries

3.2.1 Problem Statement

In LBS, most queries are continuous queries [95]. Unlike snapshot queries that are evaluated only once, continuous queries require continuous evaluation as the query results vary with the change of location and time. Continuous skyline query processing has to re-compute the skyline when the query location and objects move. Due to the spatiotemporal coherence of the movement, the skyline changes in a smooth manner. Notwithstanding this, updating the skyline of the previous moment is more efficient than conducting a snapshot query at each moment.

For intuitive illustration, we limit the data and the moving query points to a two-dimensional (2D) space. Our statement is however sufficiently general for high-dimensional space too. We have a set of n data points in the format $<x_i$, y_i, v_{xi}, v_{yi}, p_{i1}, ..., p_{ij}, ..., $p_{im}>$ $(i = 1, ..., n)$, where x_i and y_i are positional coordinate values in the space, v_{xi} and v_{yi} are respectively velocity in the X and Y dimensions while p_{ij}'s $(j = 1, ..., m)$ are static non-spatial attributes, which will not change with time.

For a moving object, x_i and y_i are updated using v_{xi} and v_{yi}. When it is stationary, v_{xi} and v_{yi} are zero. We use $Tuple(i)$ to represent the i-th data tuple in the database. Users move in the 2D plane. Each of them moves in velocity (v_{qx}, v_{qy}), starting from position (x_q, y_q). They pose continuous skyline queries while moving, and the queries involve both distance and all other static dimensions. Such queries are dynamic due to the change in spatial variables. In our solution, we only compute the skyline for (x_q, y_q) at the start time 0. Subsequently, continuously query processing is conducted for each user by updating the skyline instead of computing a new one from scratch each time. Moving points are allowed to change their velocities, which will be addressed in Section 3.4.3. Without loss of generality, we restrict our discussion to what follows the MIN skyline annotation [19], in which smaller values of distance or attribute p_{ij} are preferred in comparison to determine dominance between two points.

3.2.2 Time Parameterized Distance Function

In our problem, the distance between a moving query point and a data point is involved in the skyline operator. For a moving data point pt_i starting from (x_i, y_i) with velocity (v_{ix}, v_{iy}), and a query point starting from (x_q, y_q) moving with (v_{qx}, v_{qy}), the Euclidean distance between them can be expressed as a function of time t: $dist(q(t), pt_i(t)) = \sqrt{at^2 + bt + c}$, where a, b and c are constants determined by their starting positions and velocities: $a = (v_{ix} - v_{qx})^2 + (v_{iy} - v_{qy})^2$; $b = 2[(x_i - x_q)(v_{ix} - v_{qx}) + (y_i - y_q)(v_{iy} - v_{qy})]$; $c = (x_i - x_q)^2 + (y_i - y_q)^2$. For simplicity, we use function $f_i(t) = at^2 + bt + c$ to denote the square of the distance. When pt_i is static, a, b and c are still determined by the formulas above with $v_{ix} = v_{iy} = 0$. This time parameterized distance function has been used in literature to help processing queries in moving object databases [43, 71, 84].

3.2.3 Terminologies

For two points pt_1 and pt_2, if $dist(pt_1, q) \leq dist(pt_2, q)$ and $pt_1.p_k \leq pt_2.p_k, \forall k$, and at least one "$<$" holds, i.e., $\exists k$, such that $pt_1.p_k < pt_2.p_k$, we say pt_1 dominates pt_2. We say pt_1 and pt_2 are *incomparable* if pt_1 does not dominate pt_2, and pt_1 is not dominated by pt_2. We use $pt_1 \prec pt_2$ to represent that pt_1 dominates pt_2, and $pt_1 \preceq pt_2$ to represent that pt_1 dominates pt_2 for all static non-spatial dimensions.

In kinetic data structures, a *certificate* is a conjunction of algebraic conditions, which guarantees the correctness of some relationship to be maintained between mobile data objects. Readers are referred to [12] for the formal and detailed description of kinetic data structures (KDS). In this chapter, we use a certificate to ensure the status of a data point is valid within a period of time t. For example, a certificate of a point can guarantee it staying in the skyline for a period of time t. Beyond t, its certificate is invalid and an event will trigger a process to update the certificate, which may result in a change in the skyline.

3.3 The Change of Skyline in Moving Context

In this section, we analyze the change in skyline in continuous query processing. We first point out the search bound that can be used to filter out unqualified data points in determining the skyline for a moving query point. Then we carry out an analysis of the skyline change due to the movement, which reveals some insights for the algorithms in the next section.

3.3.1 Search Bound

Although in our problem the skyline operator involves both dynamic and static dimensions, some data points could be always in the skyline no matter how the

data points and query points move. This is because they have dominating static non-spatial values, which guarantee that no other objects can dominate them. We denote this subset of skyline points as SK_{ns} and the whole set of skyline points as SK_{all}. We call SK_{ns} the *static partial skyline*, and SK_{all} the *complete skyline*.

We call points in SK_{ns} *permanent skyline points*. In this way, we distinguish those points always in the complete skyline from the rest of the dataset. The benefit of this discrimination is threefold:

1. It extracts the unchanging part of a continuous skyline query result from the complete skyline SK_{all}. This allows efforts in query processing to be concentrated on the changing part only, i.e., $SK_{all} - SK_{ns}$. We name that part SK_{chg}, and call those points in it *volatile skyline points*. In continuous skyline query processing, only SK_{chg} needs tracking for each query. In this manner, we can reduce overall processing cost.

2. The discrimination can reduce the amount of data to be sent to clients. Since SK_{ns} is always in the final skyline result, we need to send it only once from server to client. This benefits mobile applications where clients and servers are usually connected via limited bandwidth.

3. Static partial skyline SK_{ns} also provides an indication of the search bound for processing a continuous skyline query. Since SK_{ns} is always contained in SK_{all}, for any point not in SK_{ns} to enter SK_{all}, it must be incomparable to any item in SK_{ns}. More specifically, it must have advantage in distance to the query point since it is dominated in all static dimensions by at least one point in SK_{ns}. This leads to Lemma 3.3.1.1.

Lemma 3.3.1.1 *At any time t, if sp_f is the farthest point in SK_{ns} to the query point, then any point pt not nearer to the query point than sp_f is not in the complete*

skyline.

Proof. Obviously $pt \notin SK_{ns}$, thus $\exists sp \in SK_{ns}$ s.t. $\forall k, sp.p_k \leq pt.p_k$ and at least one inequality holds. From $dist(q, sp) \leq dist(q, sp_f)$ and $dist(q, sp_f) \leq dist(q, pt)$, we get $dist(q, sp) \leq dist(q, pt)$ by transitivity. Because of its disadvantage in both spatial and non-spatial dimensions, pt is dominated by sp at time t so that it is not in the complete skyline. $\diamond$

Lemma 3.3.1.1 indicates a search bound for the complete skyline. This can be used to filter out unqualified points in query processing: those farther away than all points in SK_{ns} cannot be in the complete skyline. Refer to the example in Figure 3.2, $SK_{ns} = \{3, 5\}$. At time t_1, $SK_{chg} = \{1\}$ and restaurants 2, 4 and 6 are not in the skyline as they are farther to the query point than restaurant 5, which is the farthest permanent skyline point to the query point.

3.3.2 Change in the Skyline

When the query point q and data points move, their distance relationships may change. This causes the skyline to change as well. As discussed in Section 3.3.1, such changes only happen to SK_{chg}, i.e. $SK_{all} - SK_{ns}$. It is also mentioned in Section 3.2.2 that the square of the distance from each point to the query point can be described as a function of time t. Figure 3.3 illustrates an example of such functions of several points with respect to the moving query point.

Intuitively, a skyline point s_i in SK_{chg} before time t_x may leave the skyline after t_x. On the other hand, a non-skyline point nsp at time t_x may enter the skyline and become part of SK_{chg} after t_x. For the former, after time t_x, s_i must be dominated by a skyline point s_j in SK_{all}. For the latter, when nsp enters the skyline after time t_x, those points that used to dominate nsp before t_x will stop dominating it. That moment t_x is indicated by an intersection of two distance

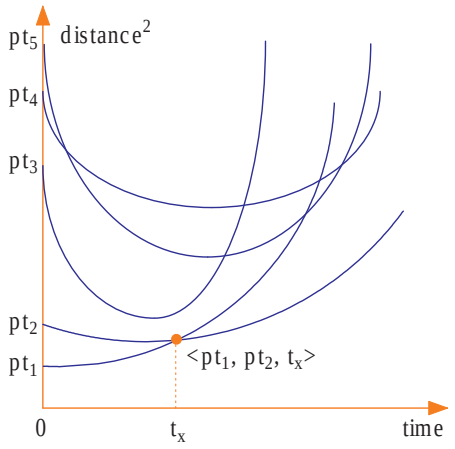


Figure 3.3: An example of distance function curves

function curves. We use $<pt_1, pt_2, t_x>$ to represent an intersection shown in Figure 3.3, where at time t_x point pt_2 is getting closer to the query than point pt_1, opposite to the situation before t_x. From the figure, we can see that such an intersection only alters pt_1 and pt_2's presence in or absence from SK_{chg} if it does cause change. This is because before and after the intersection, the only change of comparison is $dist(q, pt_1) < dist(q, pt_2)$ to $dist(q, pt_2) < dist(q, pt_1)$. If no intersections happen, the skyline does not change at all because the inequality relationship between the distances of all points to the query point remains unchanged. Nevertheless, not every intersection necessarily causes the skyline to change. Whether an intersection $<pt_1, pt_2, t_x>$ causes change is relevant to which set pt_1 and pt_2 belong to just before time t_x, i.e., SK_{ns}, SK_{chg} or $\overline{SK_{all}}$ (neither of the former two, i.e., not in SK_{all}). We have following lemmas to clearly describe these possibilities.

Lemma 3.3.2.1 *An intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) has no influence on the skyline if one of the following conditions holds before* t_x*:*

(1) $pt_1 \in SK_{ns}$ *and* $pt_2 \in SK_{ns}$

(2) $pt_1 \in SK_{ns}$ *and* $pt_2 \in SK_{chg}$

(3) $pt_1 \notin SK_{all}$ *and* $pt_2 \in SK_{ns}$

(4) $pt_1 \notin SK_{all}$ *and* $pt_2 \in SK_{chg}$

(5) $pt_1 \notin SK_{all}$ *and* $pt_2 \notin SK_{all}$

Proof. (1) This is obvious according to the definition of permanent skyline points.
(2) Obviously pt_1 does not leave the skyline. Assuming that pt_2 leaves the skyline after t_x, there must be another skyline point s dominating it, i.e., $dist(q, s) < dist(q, pt_2)$ for $t > t_x$ and $\forall k, s.p_k \leq pt_2.p_k$. Since intersection $<pt_1, pt_2, t_x>$ does not change the distance inequality relationship between s and pt_2, $dist(q, s) < dist(q, pt_2)$ also holds for $t < t_x$. Thus s dominates pt_2 before t_x, which contradicts $pt_2 \in SK_{chg}$ before t_x. Therefore pt_2 does not leave the skyline either, and there is no influence on the skyline.
(3) Since $pt_1 \notin SK_{all}$ before t_x, there must be at least one skyline point $s \in SK_{all}$ dominating it. Because $dist(q, s) < dist(q, pt_1)$ does not change after the intersection, s still dominates pt_1 and thus pt_1 will not enter the skyline. Since pt_2 is a permanent skyline point, it will not leave the skyline.
(4) Due to the same reasoning as in (3), pt_1 will not enter the skyline after t_x. Due to the same reasoning in (2), pt_2 itself will not leave the skyline after t_x.
(5) Due to the same reasoning as in (3), neither pt_1 nor pt_2 will enter the skyline after t_x. $\diamond$

Lemma 3.3.2.2 *An intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) may have influence on the skyline if one of the following conditions holds before* t_x*:*

(1) $pt_1 \in SK_{ns}$ *and* $pt_2 \notin SK_{all}$

(2) $pt_1 \in SK_{chg}$ *and* $pt_2 \in SK_{ns}$

(3) $pt_1 \in SK_{chg}$ *and* $pt_2 \in SK_{chg}$

(4) $pt_1 \in SK_{chg}$ *and* $pt_2 \notin SK_{all}$

Table 3.1: Intersections and possible skyline changes

$pt_1 \setminus pt_2$	SK_{ns}	SK_{chg}	$\overline{SK_{all}}$
SK_{ns}	—	—	$\surd$
SK_{chg}	$\surd$	$\surd$	$\surd$
$\overline{SK_{all}}$	—	—	—

Proof. (1) Obviously pt_1 will not leave the skyline after t_x. Since $pt_2 \notin SK_{all}$ before t_x there must be at least one skyline point in SK_{all} dominating it. If pt_1 is the only dominating pt_2 before t_x, after t_x, pt_1 will stop dominating pt_2 and no other skyline points will dominate it. Consequently, pt_2 will enter the skyline after t_x.

(2) Obviously pt_2 will not leave the skyline after t_x. But if $\forall k, pt_2.p_k \leq pt_1.p_k$ holds, pt_2 will dominate pt_1 and cause pt_1 to leave the skyline since $dist(q, pt_2) < dist(q, pt_1)$ holds after t_x.

(3) If $\forall k, pt_2.p_k \leq pt_1.p_k$ holds, pt_2 will dominate pt_1 and cause pt_1 to leave the skyline because $dist(q, pt_2) < dist(q, pt_1)$ holds after t_x. Due to the same reasoning as in (2) of Lemma 3.3.2.1, pt_2 itself will not leave the skyline since no other points will dominate it after t_x.

(4) Due to the same reasoning as in (1), pt_2 may enter the skyline after t_x. $\diamond$

Table 3.1 lists all possibilities attached to an intersection. For (4) in Lemma 3.3.2.2, an interesting issue is whether pt_2 can dominate pt_1 after time t_x.

Lemma 3.3.2.3 *For an intersection* $<pt_1, pt_2, t_x>$ *(*$dist(q, pt_1) < dist(q, pt_2)$ *before* t_x*) in which* $pt_1 \in SK_{chg}$ *and* $pt_2 \notin SK_{all}$ *before* t_x*,* pt_1 *will not be dominated by* pt_2 *and leave the skyline after* t_x*, if no other intersection happens at the same time and the static non-spatial values of* pt_1 *and* pt_2 *are not the same for all dimensions.*

Proof. Assume that pt_1 will be dominated by pt_2 and leave the skyline after t_x, we have $pt_2 \preceq pt_1$. Because pt_2 is not in SK_{all} before t_x, in SK_{all} there must exist

at least one pt_3 dominating pt_2, i.e. $pt_3 \prec pt_2$. For simplicity of presentation, we assume that pt_3 is the only one skyline point of such kind. By transitivity, we have $pt_3 \preceq pt_1$. But because pt_1 is in SK_{chg}, the distance from pt_3 to the query point must be larger than that from pt_1 before t_x; otherwise $pt_3 \prec pt_1$ means pt_1's absence from SK_{chg}. Thus for pt_2 to dominate pt_1 after t_x, it must first become incomparable to pt_3, which requires that an intersection between pt_1 and pt_3 must happen no later than t_x. If the time of intersection is earlier than t_x, however, pt_2 will be in SK_{chg} before t_x. Thus that time must only be t_x. Therefore, their three distance function curves must intersect at the same point, and $<pt_1, pt_2, t_x>$ is not the only intersection at time t_x.

Note that pt_3 cannot be pt_1 in the above proof. Otherwise, before t_x, we have $pt_1 \prec pt_2$. Thus, $\exists k$ such that $pt_1.p_k < pt_2.p_k$ because their static non-spatial attribute values are not the same for all dimensions. This means pt_2 cannot dominate pt_1 even after time t_x. $\diamond$

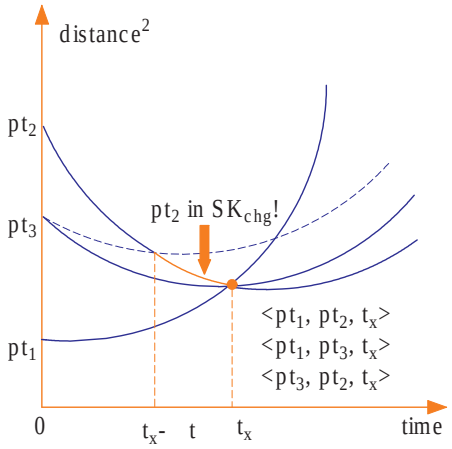


Figure 3.4: An example of multiplex intersection

Figure 3.4 shows such a scenario indicated by Lemma 3.3.2.3, and we call such an intersection *multiplex intersection*. One feasible processing strategy for this situation is to only consider if pt_2 has the chance to enter SK_{chg}. We need to

check if pt_1 is the only one that used to dominate pt_2. We ignore the possibility that pt_2 might enter the skyline and start dominating pt_1 at the same time. That possibility is indicated by other intersections at the same time, each of which is to be processed in isolation.

Accordingly, the intersection $<pt_1, pt_2, t_x>$ in Figure 3.4 will be ignored. After time t_x, both pt_2 and pt_3 are in SK_{all} but pt_1 is not. This result can be achieved as long as the three intersections are correctly processed one by one according to our discussion above, regardless of the order in which they are processed. Now, let us look at the processing of the intersections in the order listed in the figure. First, $<pt_1, pt_2, t_x>$ does not change the skyline because pt_1 does not dominate pt_2 and thus pt_2 will not enter SK_{chg} though it is getting closer to the query point than pt_1. Second, $<pt_1, pt_3, t_x>$ will cause pt_1 to leave SK_{chg} because pt_1 starts dominating it. Finally, $<pt_3, pt_2, t_x>$ will cause pt_2 to enter SK_{chg} because pt_3 is the only one that used to dominate pt_2 and now it stops dominating the point as its distance to the query point becomes larger. The procedures of other processing orders are similar and thus omitted due to space constraint.

An extreme situation is that many distance function curves are involved in the same multiplex intersection. Our processing strategy can also ensure the correct change as long as each legal intersection is processed correctly in isolation. In fact, this situation is rather special and seldom happens because it requires that all the points involved to be on the same circle centered at the query point. This situation usually happens to minority data points only, and it becomes more infrequent in the moving context.

To summarize the above analysis, we only need to take into account two primitive cases in which the skyline may change.

Case 1 *Just before time* t_x, $s_i \in SK_{chg}$ *and* $\exists s_j \in SK_{all}$ *s.t.* $s_j \preceq s_i$. *At time* t_x,

an intersection $<s_i, s_j, t_x>$ *between their distance function curves happens. Then from time* t_x *on,* $s_i \notin SK_{chg}$ *and leaves the skyline because* $s_j \prec s_i$*, and* $s_j \in SK_{all}$ *still.*

Case 2 *Just before time* t_x*,* $nsp \notin SK_{all}$ *and* $\exists s_i \in SK_{all}$ *s.t.* $s_i \prec nsp$*. At time* t_x*, an intersection* $<s_i, nsp, t_x>$ *between their distance function curves happens. Then from time* t_x *on,* $nsp \in SK_{chg}$ *because* $\nexists s_j \in SK_{all}$ *s.t.* $s_j \prec nsp$*.*

Case 1 determines a skyline change, whereas Case 2 suggests a possibility of change which requires further checking. For a period of time before the change in Case 1, s_j must be out of the circle determined by the query point q and s_i. We use $Cir(q, s_i)$ to denote the circle whose center is q and radius is $dist(q, s_i)$. In Case 2, the possible non-skyline point *nsp* is also out of circle $Cir(q, s_i)$ for a period of time before the change. Namely, the distance from each current skyline point (permanent or volatile) provides indication of future change in the skyline.

3.3.3 Continuous Skyline Query Processing

We now address the issues of continuous skyline query processing. A naive way is to pre-compute and store all possible intersections of any pair of distance function curves, and then process each one when its time comes according to the discussion in Section 3.3.2. This method produces many false hits which actually do not cause skyline to change as we have shown in Table 3.1.

Based on those observations, we compute and store intersections in an evolving way. We only keep those intersections with possibility to change the skyline according to Table 3.1. Specifically, first, we get the initial skyline and compute some intersections of the distance curves in terms of the current skyline points. Then, when some intersections happen and the skyline is changed, we further compute

intersections in terms of the updated skyline. By looking into the near future, we ensure that the skyline query result is kept updated, and more information will be obtained later for updating the skyline further into the future.

Besides, we keep all the current skyline points sorted based on their distance to the query point. At each evolving step, we only compute those possible intersections that involve points between two adjacent skyline points s_i and s_{i+1}, and will happen before s_i and s_{i+1} stop being adjacent. Therefore, we need to keep track of any intersection between two skyline points that are adjacent to each other in sorted order.

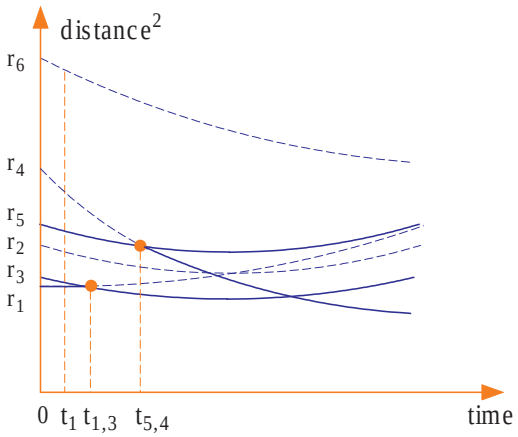


Figure 3.5: An example of evolving intersections

Figure 3.5 shows the distance curves of the restaurant example in Figure 3.2. At time t_1, restaurants r_1, r_3 and r_5 are three adjacent skyline points, and only those two dotted intersections are computed and stored for future processing. Then at time $t_{1,3}$, r_1 will leave the skyline as r_3 becomes dominating it. Next at time $t_{5,4}$, r_4 will enter the skyline as its only dominator r_5 stops dominating it. Not all intersections are stored for processing, e.g., the intersection between r_2 and r_4, and that between r_4 and r_1.

Note our method is a kind of sweeping algorithm but with two distinctive fea-

tures. We have a search bound which renders the search limited in some specific regions instead of the whole data space. The case study in Section 3.3.2 helps identify result changes and reduce processing in the maintenance. The next section addresses the data structure and relevant algorithms in detail.

3.4 Data Structure and Algorithms

3.4.1 Data Structure

We use a bidirectional linked list, named L_{sp} to store all current skyline points, which are sorted in ascending order of their distances to the query point. For each current skyline point s_i, we keep an entry of form $(flag, tuple\_id, a, b, c, t_v, t_{skip})$. $flag$ is a boolean variable indicating if s_i is in SK_{ns}. $tuple\_id$ is the tuple identifier of s_i which can be used to access the record. a, b, c are coefficients of the distance function between s_i and query point q, introduced in Section 3.2.2. t_v is only available to each changing skyline point, indicating its validity time. t_{skip} is the time when s_i will exchange its position with its successor in L_{sp}. Besides L_{sp}, a global priority queue Q_e is used to hold all events derived from certificates to represent future skyline changes, with preference being given to earlier events.

Based on the analysis in Section 3.3, we define three kinds of certificates used in the KDS, which are listed in Table 3.2. The first column is the name of a certificate, the second is what the certificate guarantees, and the third lists the data points involved in the certificate.

An event occurs when any certificate fails due to the distance change resulting from movement. Each event is in the form of $(type, time, self, peer)$, where $type$ represents the kind of its certificate; $time$ is a future time instance when the event will happen; and $self$ and $peer$ respectively represent skyline point and relevant

data point involved in the event.

Certificate $s_i s_j$ ensures for an existent volatile skyline point s_i that any other skyline point s_j with the potential to dominate s_i ($s_j \preceq s_i$) keeps being farther to query point q than s_i, therefore s_i is not dominated by any of them and stays in the skyline. Here $self$ and $peer$ respectively point to s_i's and s_j's entries in L_{sp}.

Certificate nsp_{ij} ensures for a non-skyline point nsp that all those skyline points currently dominating it keeps being closer to query point q than nsp, therefore nps is prevented from entering the skyline. When a certificate of this kind fails at $time$, nsp will get closer to query point q than one skyline point s_i, but whether it will enter the skyline or not depends on whether s_i is the only one that used to dominate it. This will be checked when an event of this kind is being processed. Here $self$ points to s_i's entry in L_{sp}, whereas $peer$ is the tuple identifier of data point nsp.

Certificate ord_{ij} ensures for an existent skyline point s_i that its successor s_j in L_{sp} keeps being farther to query point q than it. This s_j does not have the potential to dominate s_i, otherwise an $s_i s_j$ certificate will be used instead. Here $self$ points to the entry of the predecessor skyline point in the pair, and $peer$ to the successor. Certificate ord_{ij} not only keeps the order of all skyline points in L_{sp}, but also implies a way to simplify event computation and evolvement. For Case 1 described in Section 3.3.2, it also involves a position exchange in L_{sp}, i.e., just before s_j dominating s_i, s_j must be its successor. And we need to determine if an exchange in L_{sp} really results in $s_i s_j$ event. In this sense, we only need to check for s_i its successor to compute a possible $s_i s_j$ or ord_{ij} event. If s_i does have an $s_i s_j$ or ord_{ij} event, the event's $time$ value is exactly s_i's validity time t_v. If s_i has no such event, its validity time is set to infinity. An event of certificate nsp_{ij} with $self = s_i$ is supposed to have a time stamp no later than $s_i.t_v$, and those events with a later time are not considered.

Table 3.2: Certificates

Cert.	Objective	Data Points
$s_i s_j$	$\forall s_i \in SK_{chg}, s_j \in SK_{all}$, s.t. $s_j \preceq s_i \rightarrow dist(q, s_i) < dist(q, s_j)$	$self = s_i$ $peer = s_j$
nsp_{ij}	$\forall nsp_j \notin SK_{all}, \forall s_i \in SK_{all}$, s.t. $s_i \prec nsp_j \rightarrow$ $dist(q, s_i) \leq dist(q, nsp_j)$	$self = s_i$ $peer = nsp_j$
ord_{ij}	$\forall s_i \in SK_{all}$, s.t. $\exists s_j \in SK_{all} \wedge s_j \not\preceq s_i$ $\wedge s_j = s_i.next$ in L_{sp} $\rightarrow dist(q, s_i) < dist(q, s_j)$	$self = s_i$ $peer = s_j$

Initially, L_{sp} contains the current skyline points, and Q_e contains events that will happen in the nearest future. As time elapses, every due event is dequeued and processed based on its *type*. While processing due events and updating the skyline accordingly, our method also creates new events for future. Thus, Q_e evolves with due events being dequeued and new events being enqueued, providing information for correctly maintaining the skyline. At any time t after all due events are processed, L_{sp} is the correct skyline with respect to the query point q's current position.

3.4.2 Algorithms

For a given dataset, its SK_{ns} is pre-computed and stored as a system constant. Before maintaining skyline continuously, an initialization is invoked to compute the initial SK_{chg} and the earliest events. To compute SK_{chg} over static dataset for the query point's starting position, in order to use the search bound determined by SK_{ns} and reduce intermediate steps to access data tuples when computing events, we use the grid file to index all data points. Grid file provides a regular partition of space and at most two-disk-access feature for any single record [64]. In our solution for the static dataset, we use a simple uniform 2D grid file dividing the data space

into $h \times v$ cells to index D', and the data points within each cell are stored in one disk page.

For the similar reasons we use a hash based method [79] to index moving data points in D'. The data space is also divided into regular cells, with each representing a bucket to hold all those moving data points within its extent. Data points can move across adjacent cells with the velocities in its tuple, which is monitored by a pre-processing layer and declared in an explicit update request to the database. An update request can also change a data point's speed. How to deal with the updates of moving data points to maintain the correct skyline will be addressed in Section 3.4.3. Except for the difference on underlying indexing schemas, the initializations for static and moving datasets share the same framework and events creation algorithm.

The initialization framework is presented in Figure 3.6. First all permanent skyline points in SK_{ns} are inserted into L_{sp} according to their distance to query point q's starting position. The farthest distance is recorded in variable d_{bnd} as the search bound. Then starting from cell $cell_{org}$ where q's starting position lies, all grid cells are searched in a spiral manner that those on an inner surrounding circle are searched before those on an outer one. Cells beyond d_{bnd} are pruned, where *mindist* is computed as in [73] by regarding a cell as an MBR. Points in a cell not pruned are sequentially compared to the current skyline points in L_{sp}, which is adjusted with deletion or insertion if necessary. After all cells are searched or pruned, algorithm createEvents is invoked for each skyline point s_i from outermost to innermost, to compute all events for all skyline points except the last one s_{last}. Finally, algorithm handleBound is called to compute a possible nsp_{ij} event for those points farther than s_{last}.

Algorithm handleBound is presented in Figure 3.7. It does not involve all outer

Algorithm initialization(q)
Input: q is the query point
Output: the skyline for q's starting position
the event queue to be used in maintenance
// load SK_{ns} into skyline list
1. **for each** s_i in Sk_{ns}
2. Compute a, b, c in terms of q;
3. Insert an entry $(1, s_i, a, b, c, \infty, \infty)$ into L_{sp};
// search bound determined by SK_{ns}
4. $d_{bnd} = dist(L_{sp}.last, q)$;
// compute initial skyline
5. Search the grid cell $cell_{org}$ in which q lies;
6. **while** there still exist grid cells unsearched
7. **for each** cell $cell_i$ on next outer surrounding circle
8. **if** $(mindist(q, cell_i) \geq d_{bnd})$
9. **break;**
10. **else** Search $cell_i$;
// compute events
11. **for each** s_i from $L_{sp}.last.prev$ to $L_{sp}.first$
12. createEvents(s_i, q);
13. handleBound(q, t_{cur});

Figure 3.6: Initialization framework

Algorithm handleBound(q, t_{cur})
Input: q is the query point
Output: upcoming events for $L_{sp}.last$
1. $t_{next} = Q_e.first.time$;
2. $s_{last} = L_{sp}.last$;
3. $C = Cir(q(t_{next}), s_{last}) - Cir(q(t_{cur}), s_{last})$
4. **for each** point nsp in C
5. **for each** s_j from s_{last} to $L_{sp}.first$
6. t = time nsp will get closer to q than s_j;
7. **if** $((t \geq s_j.t_v)$ **or** $(t \geq s_j.t_{skip}))$ **continue**;
8. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
9. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
10. **break**;

Figure 3.7: Handle bound

non-skyline points of s_{last}'s, instead it is limited to an estimated region. This region C is the difference between the two circles determined by s_{last} and query point q at two different times, the current time and the earliest event time t_{next} in the future. Only those non-skyline points in C have chance to enter the skyline before t_{next}.

Algorithm createEvents is presented in Figure 3.8. For a given skyline point s_i in L_{sp}, the algorithm first computes the time t when s_i and the next skyline point s_j in L_{sp} will exchange their position in the list, i.e. when s_j will get closer to q than s_i. If t is later than s_j's skip time or s_i's validity time, it is ignored. Otherwise, it means an $s_i s_j$ event depending on s_j's validity time if $s_i \in SK_{chg}$, or it is a simple order change event. Then for each non-skyline point nsp between $Cir(q, s_i)$ and $Cir(q, s_j)$, the algorithm computes nsp_{ij} event by looping on all skyline points in the inner of nsp. Once an nsp event is derived, the loop on all inner skyline points breaks.

During the lifespan of a continuous skyline query, the skyline result is maintained by correctly processing due events and creating new events. In each maintenance step, the due events are dequeued and processed according to their types, and new events for further future are computed and enqueued based on new positions. As in the initialization, the event of those points out of the last skyline point is computed in a special way with an estimated search region by calling algorithm handleBound.

The actions to process each kind of events, respectively shown in Figures 3.9 to 3.11, are described as follows. For an $s_i s_j$ event, s_i is removed from the skyline list L_{sp} and new events are computed for s_i's predecessor because its successor skyline point in L_{sp} has been changed. For an nsp_{ij} event, the non-skyline point nsp will be checked against all those skyline points closer to the query point, to see if nsp will enter the skyline. If not, a possible new nsp event is computed an enqueued.

Algorithm createEvents(s_i, q)
Input: s_i is a skyline point in L_{sp}
q is the query point
Output: upcoming events for s_i

```
1.  peer = null;
    // compute events with next skyline point in L_sp
2.  s_j = s_i.next;
3.  t = time s_i and s_j will exchange position;
4.  if ((t < s_j.t_skip) and (t < s_j.t_v))
5.      if (!s_i.flag)
6.          if ((t < s_i.t_v) and (∀k, s_j.p_k ≤ s_i.p_k))
7.              s_i.t_v = t; peer = s_j;
8.      else s_i.t_skip = t;
    // enqueue relevant events
9.  if (peer ≠ null)
10.     Enqueue (s_i, s_i.t_v, rep, s_i s_j) to Q_e;
11. if (s_i.t_skip < s_i.t_v)
12.     Enqueue (s_i, s_i.t_skip, s_j, ord_ij) to Q_e;
    // compute events involving non-skyline points
13. for each point nsp between Cir(q, s_i) and Cir(q, s_j)
14.     for each s_k from s_i to L_sp.first
15.         t = time nsp will get closer to q than s_k;
16.         if ((t ≥ s_k.t_v) or (t ≥ s_k.t_skip)) continue;
17.         if (∀k, s_k.p_k ≤ nsp.p_k)
18.             Enqueue (s_k, t, nsp, nsp_ij) to Q_e;
19.             break;
```

Figure 3.8: Create events

Process $s_i s_j$ **event** e
1. $s_i = e.self$; $s_j = e.peer$;
2. Delete s_i from L_{sp};
3. $s_i = s_j.prev$;
4. createEvents(s_i, q);

Figure 3.9: Process $s_i s_j$ event

Process nsp_{ij} **event** e
1. $s_i = e.self$; $nsp = e.peer$
2. $dominated$ = FALSE;
   // check inner skyline points
3. **for each** s_j in L_{sp} from $s_i.prev$ to first
4. **if** $(\forall k, s_j.p_k \leq nsp.p_k)$
5. $dominated$ = TRUE;
6. **break**;
   //nsp does not enter the skyline this time
7. **if** ($dominated$)
8. $s_k = s_i.prev$;
9. t = time nsp will get closer to q than s_k;
10. **if** $((t < s_k.t_v)$ **and** $(t < s_k.t_{skip})$
    and $(\forall k, s_k.p_k \leq nsp.p_k))$
11. Enqueue (s_k, t, nsp, nsp_{ij}) to Q_e;
12. **else** // nsp enters the skyline
13. Insert nsp into L_{sp} before s_i;
14. $s_j = s_i.prev$;
15. createEvents(s_j, q);
16. createEvents($s_j.prev$, q);

Figure 3.10: Process nsp_{ij} event

Process ord_{ij} **event** e
1. $s_i = e.self$; $s_j = e.peer$;
2. Switch s_i and s_j's positions in L_{sp};
3. createEvents(s_i, q);
4. createEvents(s_j, q);
5. **if** $(s_j.prev \neq null)$
6. createEvents($s_j.prev$, q);

Figure 3.11: Process ord_{ij} event

Otherwise, nsp will be added into the skyline list L_{sp} and relevant events will be computed for it and its predecessor. For an ord_{ij} event, the skyline list L_{sp} is correctly adjusted by switching s_i and s_j, and relevant events are computed and enqueued for them and their predecessor if it exists.

3.4.3 Updating the Moving Plan

A moving data point mpt_i's distance function does not change unless its moving plan changes. When this happens, the intersections of its distance function and other points' will also be changed as a consequence, which invalidates those events computed based on mpt_i's old distance function. Figure 3.12 shows how a data point's velocity change causes the intersections of the function curves to change. Thus, it may cause the skyline to change in the future.

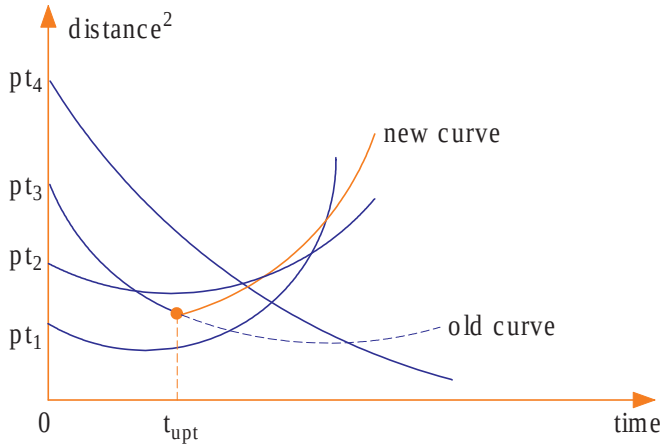


Figure 3.12: An example of the change of moving plan

To ensure correct process with updates, we need to add for each moving object's tuple a field t_{upt} indicating its last update time. We define an update request for any moving data point mpt_i in the form $update(id, x, y, v_x, v_y)$. id is mpt_i's identifier which can be used to locate its tuple directly. x and y represent its current position. v_x and v_y represent its current speed. The algorithm updateMotion in Figure 3.13 is

used to process such updates. When an update request comes in, it is first checked if mpt_i has moved to a new cell and if its speed has been changed since the last update. If x and y indicate that mpt_i has moved to a different cell, we need to remove it from the old one and insert it into the new one (line 1-5), which incurs 2 IOs. If v_x and v_y indicate that mpt_i's speed is not changed, the algorithm stops (line 6-7). Otherwise, we need to update the speed record for mpt_i (line 8-10), and adjust relevant events starting from the first skyline points till the first one out of mpt_i (line 17). If mpt_i is a skyline point, then its events will be re-computed and the algorithm stops (line 12-15). Otherwise, the algorithm continues to compute nsp_i events for mpt_i (line 19-24). With the independent distribution assumption, $(|SK_{all}| + 1)/2$ skyline points are expected to be accessed. To facilitate location of events involving a data point efficiently, the priority event queue is implemented using a B^+-tree, and each current skyline point s_i has a list of pointers to all those events whose $self$ is s_i.

It also can be seen in Figure 3.12 that right at the moment t_{upt} when an update request comes in, the skyline does not change abruptly. To keep the skyline correct, the update request is only processed after all due events are processed, i.e., updateMotion(req) at time t_{upt} executes after updateSkyline(t_{upt}) completes.

3.5 Cost Analysis and Discussion

3.5.1 Cost Analysis

The space cost incurred by our method consists of two components: the space used to keep the skyline and that used to store events. For a d-dimensional dataset with N points subject to independent distribution, the expected size of its skyline is $n_{sky} = O((\ln N)^{d-1})$ [16]. Since there are m static dimensions involved in skyline

Algorithm updateMotion(req)
Input: req is an update request
Output:updated hash index, tuple and Q_e
1. $cell_1 = Tuple(req.id).cell$;
2. $cell_2 = Hash(req.x, req.y)$;
3. **if** ($cell_1 \neq cell_2$)
4. $Tuple(req.id).cell = cell_2$;
5. remove $req.id$ from $cell_1$ and insert it to $cell_2$;
// Still in the same grid cell
6. **if** (($req.v_x == Tuple(req.id).v_x$) **and** ($req.v_y == Tuple(req.id).v_y$))
7. **return**;
8. $Tuple(req.id).v_x = req.v_x$
9. $Tuple(req.id).v_y = req.v_y$
10. $Tuple(req.id).t_{upt} = t_{cur}$
// Adjust relevant events
11. **for each** s_i in L_{sp} from $L_{sp}.first$
12. **if** ($s_i.tuple\_id == req.id$)
13. Delete all s_i's events;
14. createEvents(s_i, q);
15. **return**;
16. Delete all s_i's events with $peer == req.id$;
17. **if** ($dist(q, Tuple(req.id)) \leq dist(q, s_i)$) **break**;
18. $nsp = req.id$;
19. **for each** s_j from s_i to $L_{sp}.first$
20. t = time nsp will get closer to q than s_j;
21. **if** (($t \geq s_j.t_v$) **or** ($t \geq s_j.t_{skip}$)) **continue**;
22. **if** ($\forall k, s_j.p_k \leq nsp.p_k$)
23. Enqueue (s_j, t, nsp, nsp_{ij}) to Q_e;
24. **break**;

Figure 3.13: Handle the change of moving plan

operator in our assumption in Section 3.2.1, the size of skyline on static dimensions is $|SK_{ns}| = O((\ln N)^{m-1})$, and the size of skyline on all dimensions is $|SK_{all}| = O((\ln N)^m)$ at any time. Thus the size of changing part is $|SK_{chg}| = |SK_{all}| - |SK_{ns}| = O((\ln N)^m - (\ln N)^{m-1})$ at any time.

Now we consider the worst-case number of events, i.e., failure of certificates, at any time. In our method, any $s_i s_j$ event or ord_{ij} event is determined by an underlying intersection between two adjacent skyline points' distance function curves. They are *external events* because they affect the skyline result we maintain [12]. Therefore, the maximum number of events of these two kinds is $|SK_{all}|_{max}/2$, since we reduce multiplex intersections into simple ones and store only one at a time. In contrast, nsp_{ij} events are *internal events* because they are used to adjust internal data structure. As we at most keep one nsp_{ij} event for a non-skyline point at any time, the worst case is that every non-skyline point is involved in such an event, which means the number of nsp_{ij} events is not more than $N - |SK_{all}|_{max}$. By summing up all events, the total number of events in the worst case is $N - |SK_{all}|_{max}/2$. Hence, the ratio of total events to external events is $2N/|SK_{all}|_{max} - 1$. In the worst case where $|SK_{all}|_{max}$ is 1, the upper bound of this ratio is $2N - 1$ which is linear with the number of all points involved. This worst case ratio verifies that our KDS is efficient.

As we store datasets in hard-disk, our method needs to do IO when accessing data points. The main IO cost is incurred by createEvents, which accesses all non-skyline points between the circles of two adjacent skyline points in L_{sp}. This access can be regarded as a special region query over the dataset indexed by grid file, asking for points between two circles with same center but different radiuses. The IO cost of such a query can be estimated with a simple probabilistic model. Let the data space be a 2D unit space (as we use a 2D grid file to index all data points), and

the outer and inner circles have radii R_i and r_i respectively when we create events for the ith skyline in L_{sp}. Then the area of the query circle is $S = \pi(R_i^2 - r_i^2)$, and the query will access $SP = \pi(R_i^2 - r_i^2)P$ grid cells (pages), where P is the total number of grid cells. Next we estimate R_i, the distance from q to the i+1th skyline point in L_{sp}. Suppose we do an incremental kNN search for q, if we have met i+1 permanent skyline points, then we must have met the the i+1th skyline point already. With the assumption of independent distribution, $(i+1)N/|SK_{ns}|$ points are met before the i+1th permanent skyline point. Then in the 2D unit space, we have $\pi R_i^2 = ((i+1)N/|SK_{ns}|)/N$, which leads to an upper bound of R_i satisfying $R_i^2 = (i+1)/(\pi|SK_{ns}|)$. For r_i, which is the distance from query point q to the ith skyline point, we use a lower bound $min(\sqrt{i/(\pi|SK_{ns}|)}, i/(\sqrt{N}-1))$ to approximate it. In this way, we get an upper bound of SP.

Let us compare the time cost of continuous skyline query to that of snapshot skyline queries. Assume $\mathcal{N}$ snapshot queries are triggered within a time period $[t_1, t_2]$, and the cost of each is C_i. Then the total and average cost of that method are $\sum_{i=1}^{\mathcal{N}} C_i$ and $\sum_{i=1}^{\mathcal{N}} C_i/\mathcal{N}$ respectively. More snapshot queries incur higher total processing cost, while each single snapshot query's cost is expected to vary little from the average cost $\mathcal{C}$ because of the static processing fashion. For the same time period, our method computes the initial skyline and events at time t_1, and then updates the skyline only when some certificate fails before t_2. Suppose the number of certificate failures during $[t_1, t_2]$ is $\mathcal{N}'$ (including the initialization), and the cost of each is C_i', the total and average cost of our method are $\sum_{i=1}^{\mathcal{N}'} C_i'$ and $\sum_{i=1}^{\mathcal{N}'} C_i'/\mathcal{N}'$ respectively. The number of certificate failures $\mathcal{N}'$ is a constant in a fixed time period, therefore the average cost $\mathcal{C}'$ is determined by the total cost only. It makes little sense to compare the total costs of these two methods. If too many snapshot queries are triggered the total cost will be very high, while

few snapshot queries deteriorate the result accuracy. To ensure a fair comparison of average costs, we set $\mathcal{N} = \mathcal{N}'$ in our experiment. In other words, we trigger snapshot queries by assuming when the skyline changes is known, which is gained from our method. The experimental study results in next section show that our method even outperforms the privileged snapshot query method.

Our problem formulation assumes a linear movement model for both query point and data points (if they are moving), which is justified by the fact that linear movement model has so far been the most popular one in the literature of moving objects research [6, 48, 63]. This model itself assumes that moving objects hold their current velocities for a period of time, which is also usually considered as a system parameter in typical indexing structures such as TPR-tree [75] and B^x-tree [45]. In most cases, on the other hand, a user can change the speed but seldom changes it every time stamp while still issuing a continuous query. As long as the velocity keeps for a period of time, our method pays off because it saves much computation cost in the result maintenance for future, and it always reports result changes in time, which renders our method beneficial.

3.5.2 Possible Extensions

It is true that users may issue continuous skyline queries with constraints in SQL WHERE clauses. Our current solution can be adapted to deal with such constraints with some modifications of the kinetic data structures (the certificate) to tender the WHERE clauses. In brief, we first apply the given constraints to SK_{ns} so that an updated SK'_{ns} are gained for further use. Then, in the use of the kinetic data structures, only those data points satisfying the specified constraints will be considered and processed. Thus, our method is still effective to support the WHERE clauses.

Our current method is focused on processing single continuous skyline query efficiently, whereas it still provides helpful indications for concurrent continuous skyline queries. $|SK_{ns}|$ obviously is the common part for all concurrent queries, which means computation savings can be achieved with $|SK_{ns}|$. Besides, concurrent queries still can share volatile skyline points in some way. These indicate that with proper adaptations our current method can be used to handle this more complex case.

3.6 Performance Studies

We conducted our experiments on a desktop PC running on MS Windows XP professional. The PC has a Pentium IV 2.6GHz CPU and 1GB memory. All experiments were coded in ANSI C++. The parameters used in the experiments are listed in Table 5.6. We used both static datasets and moving datasets. For the former, we explored into the effects of cardinality and non-spatial dimensionality on the performance. For the latter, we investigated into the effect of points speed distribution and moving plan update.

Table 3.3: Parameters used in experiments

Parameter	**Setting**
Dataset cardinality	100K, 200K, ..., 1000K
Dimensionality of non-spatial attributes	$2, 3, 4, 5$
Distribution of non-spatial attributes	Independent, Anti-Correlated
Spatial range	10000×10000
Non-spatial attribute range	[0, 10000]
Point speed range	[10, 30]
Speed Zipf factor	0, 0.5, 1.0, 1.5, 2.0
Update interval	30, 60, 90, 120
Update ratio	4%, 6%, 8%, 10%

3.6.1 Effect of Cardinality

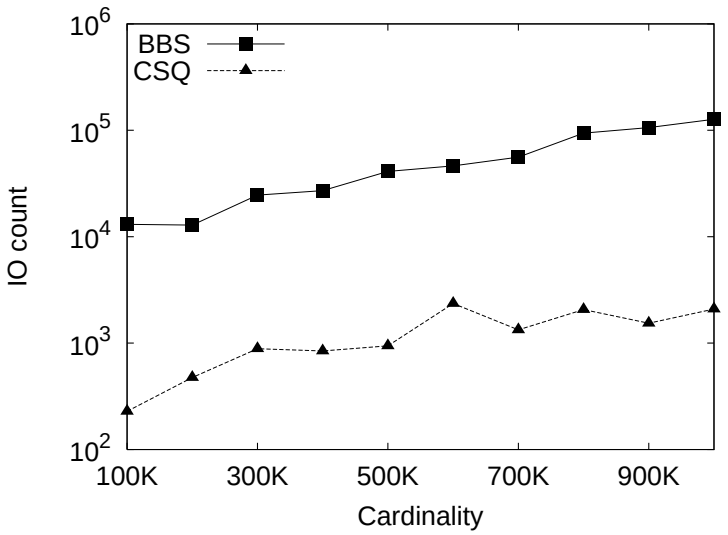
In this set of experiments, we used synthetic datasets of data points with spatial attributes (x and y) and two non-spatial attributes. For each dataset, all data points are distributed randomly within the spatial space domain of $10,000 \times 10,000$, and their non-spatial attribute values range from 1 to 100,000 according to either independent or anti-correlated distribution. The cardinality of datasets ranges from 100K to 1M. For each set of data we executed 100 continuous queries moving in random directions. For each query, we randomly generated a point within the data space as the starting position of the moving query point. The speed of each moving query point is also randomly determined and ranges from 10 to 30. Each query ends as soon as the query point moves out of the data space extent. The minimum, maximum and average validity time for all these queries are 1, 475 and 149 units respectively. The experimental results to be reported are the average values on those 100 queries.

Since BBS algorithm is the most efficient method for computing skyline in static settings (both dataset and query point are static) [66], we adapted it for comparison in our experiments. At each time instance, the BBS algorithm is invoked to re-compute the skyline in terms of the query point's new position. It is worth noting that BBS cannot correctly tell when the skyline changes as our method does.

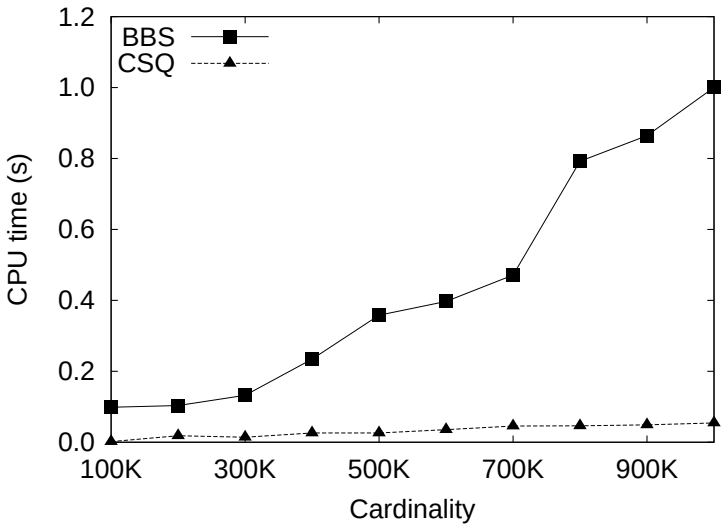
The comparison was carried out on a fair basis. The same set of randomly generated queries are used by both methods on the same series of datasets. Processing costs, IO count and CPU time, in both methods are amortized over the same number of time units when the skyline changes. For both indices, R*-tree and grid file, we set the data page size to 1K bytes.

I. Datasets of Independent Non-spatial Attribute Values

Figure 3.14(a) shows that as cardinality increases the logarithm of IO count of

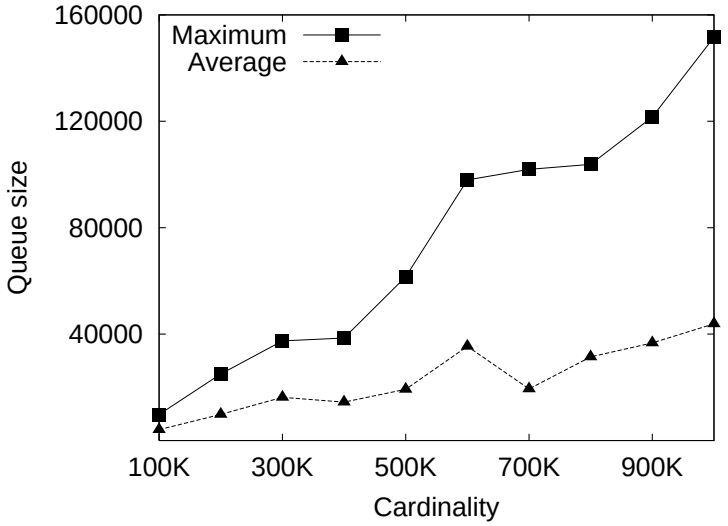


(a) IO

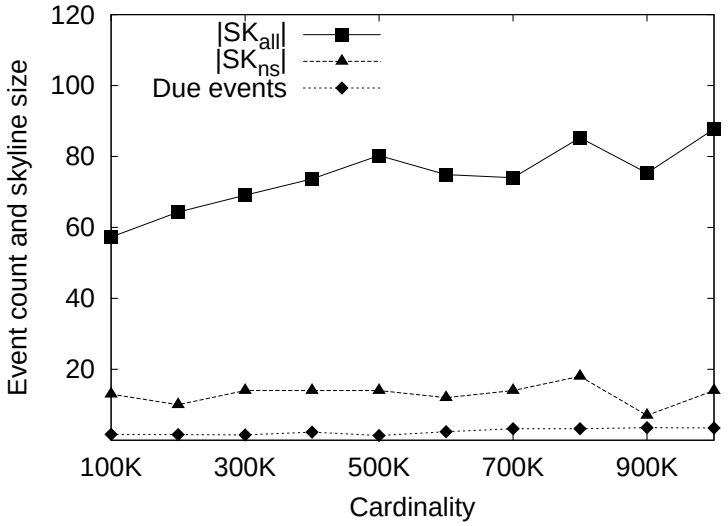


(b) CPU time

Figure 3.14: Query costs v.s. cardinality of independent datasets



(a) Event queue size



(b) Skyline size and due events

Figure 3.15: KDS overheads v.s. cardinality of independent datasets

our maintenance method grows steadily, and nearly 2 orders of magnitude less than that of BBS. Figure 3.14(b) shows that as cardinality increases the CPU time cost of our maintenance solution grows steadily, in a rate much less than that of BBS. At each time instance, our maintenance solution does not need to search the whole dataset again to re-compute the skyline from scratch, instead it mainly involves event processing which consists less computation of distance and comparison of attribute values than BBS, which does a totally new search via R*-tree. This processing behavior difference leads to the difference on processing costs.

Figure 3.15(a) shows the effect of cardinality on event queue size at any time unit. The maximum size is gained throughout all 100 queries. It can be seen that the queue event size increases as the cardinality increases, the average queue size is much smaller compared to the maximum size, and it does not exceed 6% of the cardinality.

Figure 3.15(b) shows the effect of cardinality on skyline size and the number of events being processed at any time unit. It can be seen that complete skyline size roughly increases as cardinality increases, but the average number of due events at any time unit of skyline change never exceeds 4, which indicates the efficiency of our maintenance strategy. Figure 3.15(b) also implies the percentage of the complete skyline points that are permanent skyline points. Furthermore, by contrasting Figure 3.15(b) with Figure 3.14, we can see how BBS and CSQ perform as the percentage roughly decreases. The similar implications and contrasts can also be gained from results on effect of anti-correlated data cardinality, as well as results on effect of non-spatial dimensionality.

By comparing Figures 3.15(a) and 3.15(b), we can see that some events are not processed before the query ends. In a real application, we can take advantage of this observation to further reduce the queue size. The lifetime of a query can be

estimated in a specific scenario, e.g., in 2 hours or this afternoon, and any event whose due time later than it will be prevented from being enqueued.

II. Datasets of Anti-Correlated Non-spatial Attribute Values

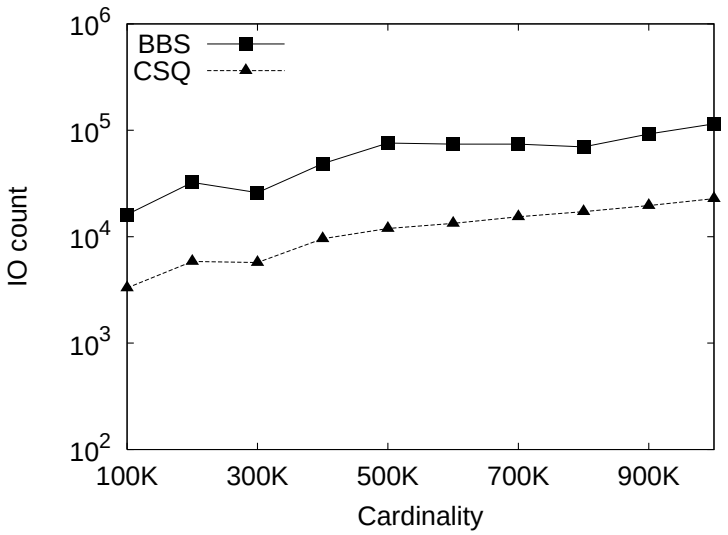
We also carried out experiments on datasets whose two non-spatial attributes are anti-correlated. We used the method in [19] to generate such datasets. Figures 3.16 and 3.17 show that our continuous skyline query processing still outperforms adapted BBS. The higher cost than that on independent datasets is attributed to the increase of skyline size of anti-correlated datasets. The anti-correlation between non-spatial attributes also makes the events number increases less unsteadily, as the dominance relationship of data points is more irregular compared to the independent datasets.

3.6.2 Effect of Non-spatial Dimensionality

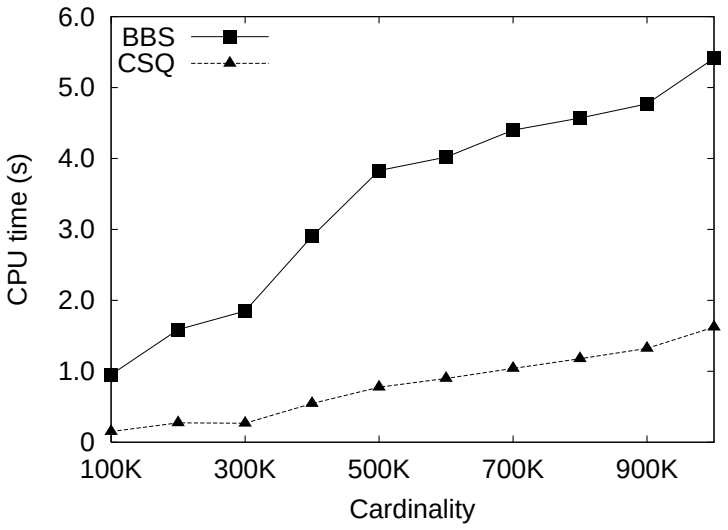
In this set of experiments, we used datasets of 500K points with non-spatial dimensionality ranging from two to five to evaluate the effect of non-spatial dimensionality on our solution. Values on those non-spatial dimensions are of independent distribution. Other settings are the same as in Section 3.6.1. Datasets with anti-correlated non-spatial values incur similar performance trends, except that every single cost is higher than its counterpart on the independent datasets. Hence we omit those figures here.

Figures 3.18(a) and 3.18(b) show the IO and CPU cost respectively. Again our maintenance method outperforms the BBS method. As the non-spatial dimensionality increases the gap of cost keeps steady.

Figure 3.19(a) shows that the event queue size decreases as the non-spatial dimensionality increases. The probability that one volatile skyline point will be dominated by others is lower when more dimensions are involved, because all di-

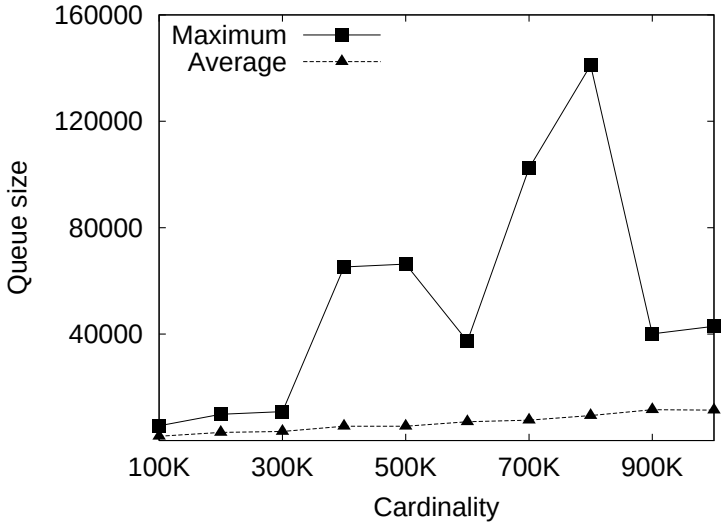


(a) IO

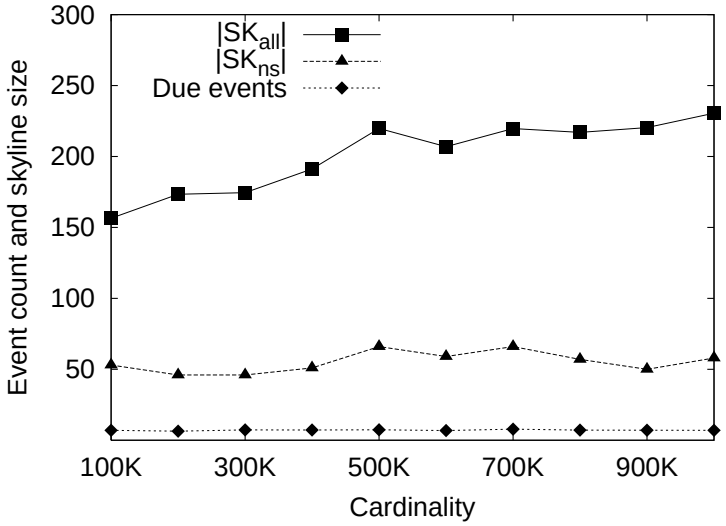


(b) CPU time

Figure 3.16: Query costs v.s. cardinality of anti-correlated datasets

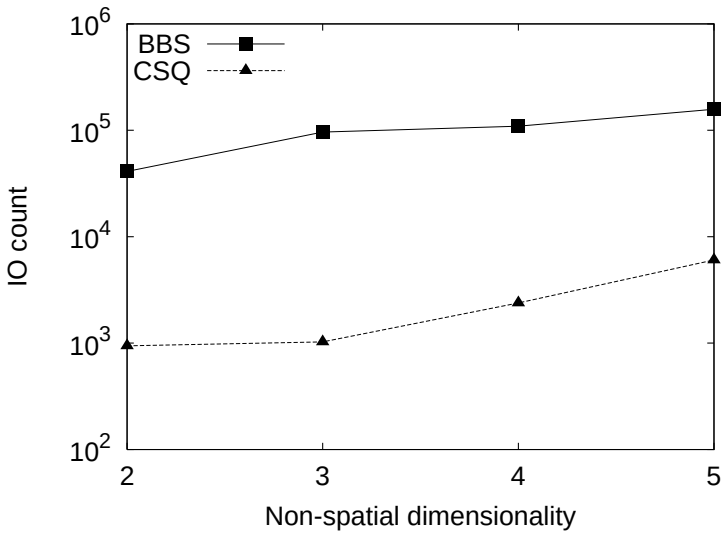


(a) Event queue size

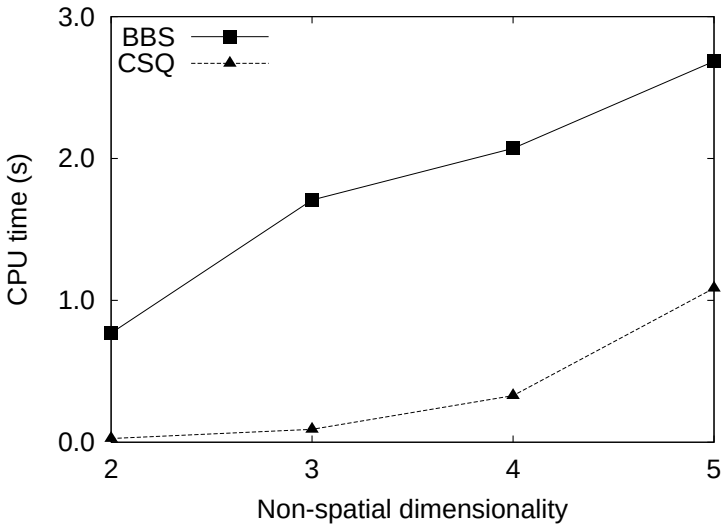


(b) Skyline size and due events

Figure 3.17: KDS overheads v.s. cardinality of anti-correlated datasets

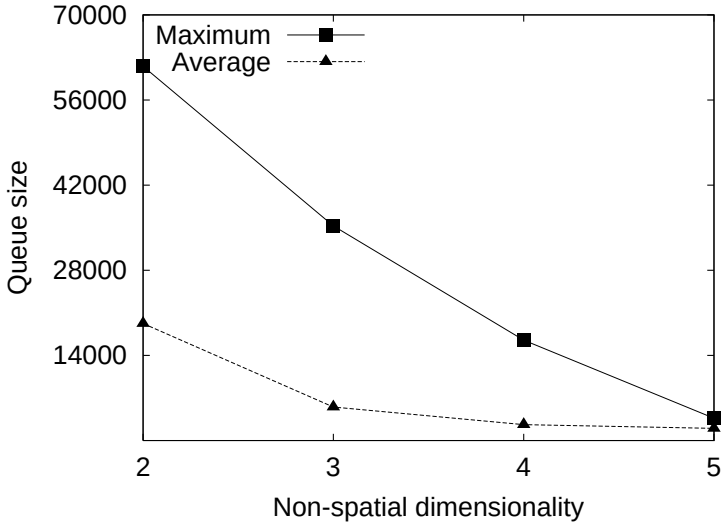


(a) IO

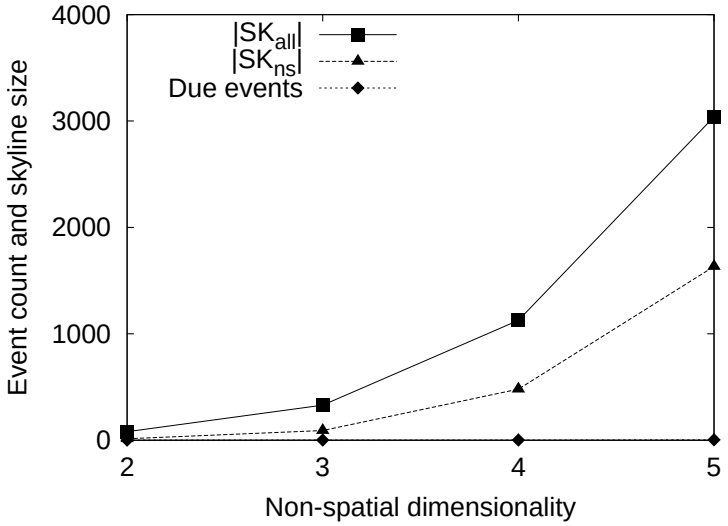


(b) CPU time

Figure 3.18: Query costs v.s. non-spatial dimensionality



(a) Event queue size



(b) Skyline size and due events

Figure 3.19: KDS overheads v.s. non-spatial dimensionality

mensions are independent in our dataset. This reduces the number of events in the queue.

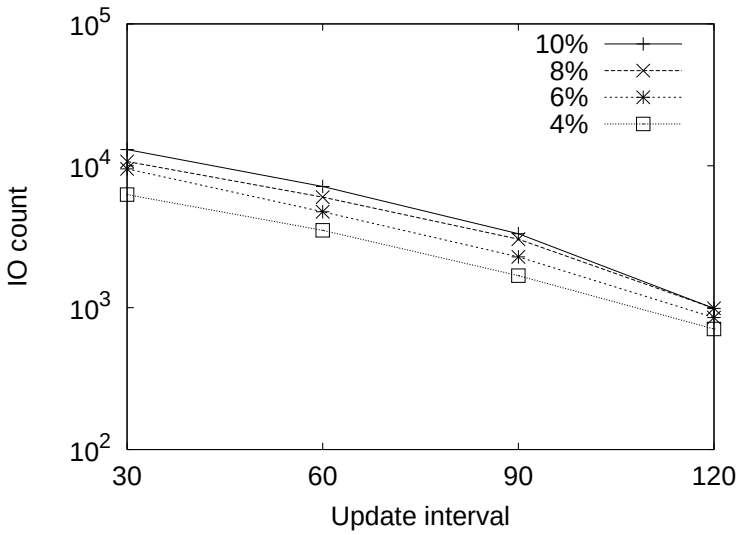
Figure 3.19(b) shows the effect of non-spatial dimensionality on skyline size and the number of events being processed at any time unit. It can be seen that both static partial skyline and complete skyline size increases as non-spatial dimensionality increases, but the average number of due events at any time unit is still drastically smaller. This indicates that our continuous query processing method still works efficiently.

3.6.3 Effect of Movement Update

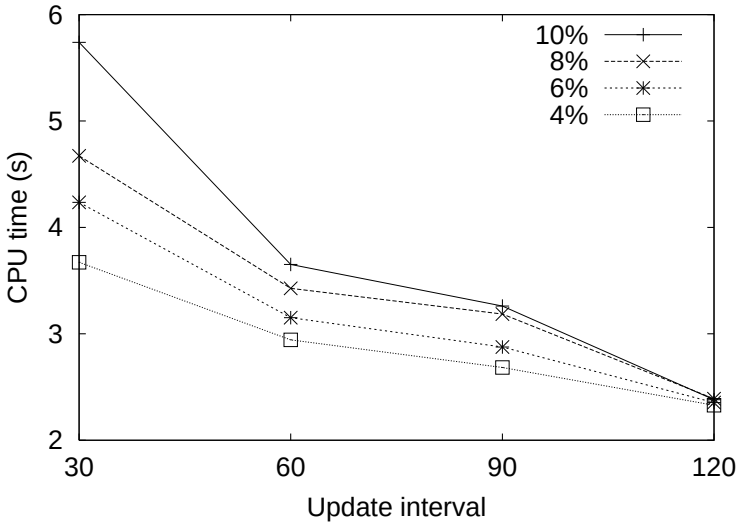
In this set of experiments, we used the dataset of 500K data points with spatial attributes (x and y) and two static non-spatial attributes. Every point in each dataset moves within the 2D extent with a speed ranging from 10 to 30. The hash mechanism is based on the same grid file used for static datasets, with each cell as a bucket containing moving data points. Periodically, a number of moving data points send in update requests. Queries are picked up in the same way as in Section 3.6.1.

In this set of experiments, the initial speeds of all 500K points were randomly distributed in the range of 10 to 30. We mainly explore into two aspects of moving data points update: update interval length and the ratio of points requesting update. We varied the update interval length from 30 to 120 time units and update ratio from 4% to 10%.

Figure 3.20(a) shows the IO count decreases as the update interval increases, and higher ratio of moving data update incurs more IO counts. The reason for this is as follows. Longer update intervals reduce the amortized update costs which involve changing point records in the database and recomputing events, and weaken

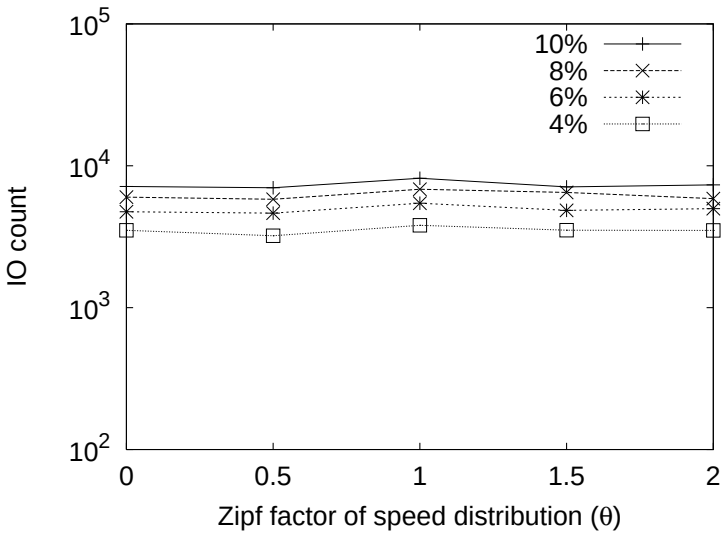


(a) IO

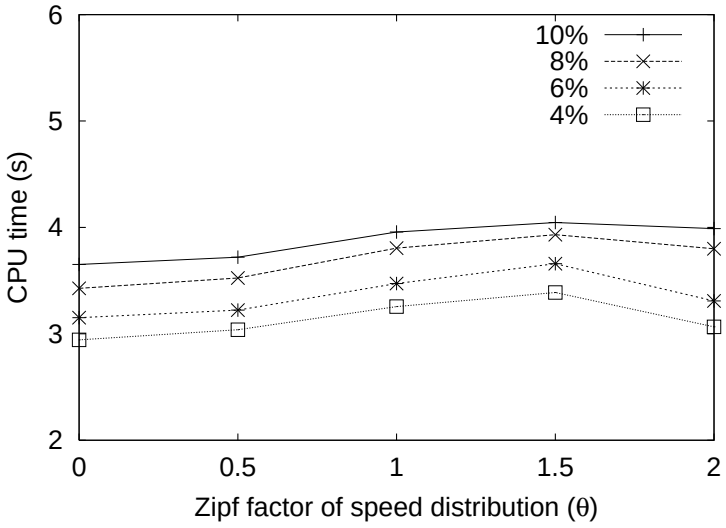


(b) CPU time

Figure 3.20: Effect of update



(a) IO



(b) CPU time

Figure 3.21: Effect of speed distribution

the effects of different update ratios. While higher update ratios increase update costs at every update time, as more point records are involved in modification and event re-computation. Referring to Figure 3.20(b), the similar changing trend is seen for the CPU time as the update interval increases.

3.6.4 Effect of Speed Distribution

In this set of experiments, we fixed the moving data points update interval to 60, varied the update ratio from 4% to 10% to see the effect of initial speed distributions. For all moving data points, their initial speeds follow the Zipfian distribution [101]. The Zipf factor θ of speed distribution varies from 0, which is a uniform distribution, to 2, which is a skewed distribution where 80% data points move slowly and the rest 20% move fast. Other settings are the same as those used in Section 3.6.3.

Figure 3.21(a) shows that the IO cost of the proposed method is not too sensitive to skewness on speed. Referring to Figure 3.21(b), CPU time increases slowly as Zipf factor θ increases from 0 to 1.5, and then decreases when θ increases from 1.5 to 2. For the same Zipf factor θ, a higher ratio of mobility data set incurs a higher processing cost in terms of both IO count and CPU time. The experiments show that our method performs well for the different distributions of moving speed. This is because speed distribution has no crucial impact on the event computation and process in our method.

3.7 Summary

In this chapter, we have addressed the problem of continuous skyline query processing for moving objects. In our problem, a continuous query is issued by a

continuously moving query point, and the data points of interest can be either static or moving too. The changing distance between the query point to any data point is considered as a unique dimension in the skyline computation. Based on a thorough analysis that exploits the spatiotemporal coherence of the problem, we propose our solution with an underlying kinetic data structure. Although in the moving setting, the skyline changes continuously due to the movement of the query point (and the data points if applicable), our solution does not need to compute the skyline from scratch at every time instance. Instead, the possible change from one time to another is predicted and processed accordingly, thus making the skyline query result updated and available continuously. The experimental studies conducted using different datasets and parameters demonstrate that the proposed method is robust and efficient.

Assuming a client/server architecture in this chapter, we process continuous skyline queries on the powerful server side. In a real dynamic setting where users have mobile devices, a user may store data on his or her device and want a query to be answered locally and instantly. In next chapter, we address efficient skyline query processing on mobile lightweight devices.

CHAPTER 4

Skyline Queries on Mobile Lightweight Devices

Due to the successful electronics miniaturization, mobile devices are becoming increasingly popular. By offering practical computing capability with great flexibility and mobility, such devices free computer users from fixed seats and awkward machines and set them into dynamic computing environments. Mobile device users can store their data on those devices, and carry out supported computations on the data with their devices anytime and anywhere. Like on normal computers, skyline queries on mobile devices may be interesting to users when the data are of multi-dimensional and the retrieval is based on multiple criteria. In this chapter, we consider skyline queries on spatial data with multiple attributes stored on mobile lightweight devices, which can provide sort of on-device location base service by locally solving user queries without contacting the remote wireless application server. However, such mobile devices are not so powerful as normal computers in terms of both storage space and computing capacity. Using existing skyline algorithms directly without any specific adjustments towards mobile lightweight device characteristics might be not efficient. To get good performance, we need to do skyline

queries on such lightweight devices appropriately. We in this chapter compare and propose suitable data storage schemes on such lightweight devices, and then adapt existing skyline query processing algorithms properly onto the resource-limited devices.

4.1 Introduction

Computer users get great freedom and mobility from the advances in electronics miniaturization, which has successfully made possible variety of mobile devices with computing capabilities. Data, and even data management modules or systems, are squeezed into such devices, which renders mobile computing anytime and anywhere. As a consequence, corresponding on-device data querying or retrieving is worth studying on such devices that usually are resource-constrained in terms of both storage space and computation power compared to normal computers.

Like on normal computers, a mobile device user may issue skyline queries when the data stored are of multi-dimensional and the retrieval is based on multiple criteria. In a practical situation, such a skyline query is likely to be related to the mobile user's current location, and the distance from that location to any point of interest is taken into account in the skyline computation.

Refer to Figure 4.1 for a motivation example. A mobile device stores a set of hotels in the city, each of which has its geographic *location*, together with *price* and *rating*. A simplest case is as follows. The mobile user wants to get the best choices from all hotels, by considering both price and rating. This is a typical skyline query. A more complex case is reached by introducing spatial constraints. The mobile device is supposed to be equipped with a GPS receiver to tell its current geographic location in longitude and latitude. Additionally, the user is only interested in those hotels within a specific distance to him, indicated by the

circle in the figure. For all those hotels covered by the distance, both price and rating are considered to retrieve best choices for the user.

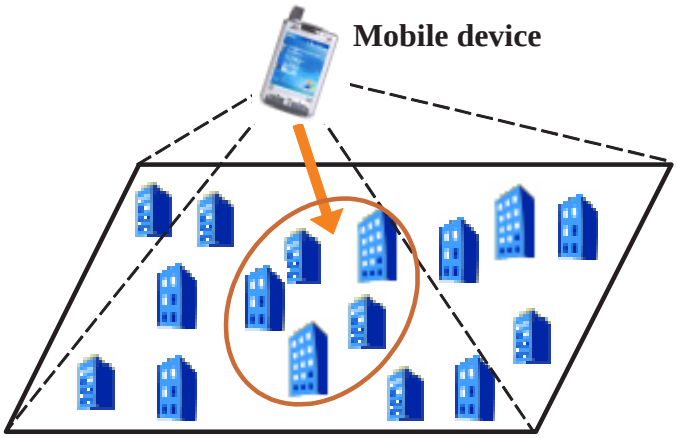


Figure 4.1: Skyline query on a mobile device

An intuitive way to address such on-device skyline queries is to directly squeeze existing skyline algorithms into the lightweight devices. As skyline query processing usually is more expensive than others like range query or k nearest neighbor query, this approach without any adjustments specific to the resource-limited characteristics of such devices is not expected to be efficient, though it is able to produce results. Proper adjustments and configurations, on the contrary, are likely to improve the performance.

In this chapter, we consider possible adjustments and configurations to be used on a mobile device to speed up the local skyline query processing. Because mobile devices are equipped with limited storage space, data should be stored in a way that takes into account this space constraint. Besides, it is favorable if the storage benefits the corresponding skyline query processing algorithm based on it by shortening the query processing time.

First, we carry out a reasonable analysis on the existing storage schemes for space-constrained devices, by taking into account the skyline computation char-

acteristics – dominance comparison. Based on the analysis, we propose a hybrid storage scheme that deals differently with spatial coordinates and non-spatial attributes. Furthermore, we adapt a simple skyline algorithm BNL to the hybrid storage scheme. Results of experiments on a real pocket PC show that our proposal not only saves data storage space but also speeds up the skyline query processing on the device.

The rest of this chapter is organized as follows. In Section 4.2, we give a brief definition of our problem. In Section 4.3, we propose the hybrid data storage scheme based on an analysis on existing schemes. In Section 4.4, we adapt skyline algorithms for normal computers to the proposed hybrid storage scheme on lightweight devices. Section 4.5 presents the experimental results, followed by the summary of this chapter in Section 4.6.

4.2 Problem Definition

We assume a mobile lightweight device M holds a database relation R that contains the data that pertains to sites of interest located in a geographical area. Relation R conforms to the schema $<x, y, p_1, p_2, \ldots, p_n>$, where (x, y) represents the location of a site and the p_1 to p_n are attributes describing a site.

In this setting, a mobile device M can ask a local skyline query, whose result consists of all sites s in relation R that satisfy these conditions: (a) site s is within distance d of M's current position; (b) site s is in the skyline of R' in terms of all attributes p_i, where R' is the set of all sites satisfying condition (a). Such a query can be defined as $Q_{sky} = (pos, d)$, pos is the current location of M, and d is the distance specifying the region of interest.

The queries we consider here differ from traditional skyline queries in that a

region of interest is specified in each query, making the query a constrained skyline query. However, this query differs from those obtained by placing constraints on the dimensions involved in the skyline operation [66]. In particular, we use spatial constraints that are not involved in the skyline operation. Such queries are meaningful in practice. For instance, a tourist in a city may want to know about inexpensive and highly rated restaurants within a certain range, in order to find a near-by place for dinner. If no spatial constraint is applied first to filter out far restaurants, a skyline query on all restaurants will return those that are very cheap but are too far away, which are not desirable to the tourist because to their long distances. Thus, the use of distance of interest as a spatial constraint is reasonable as it helps to filter out answers with low acceptance.

As a matter of fact, a mobile user usually is only interested in sites located within a region around the user's current position, rather than in the entire geographical space. For example, people are more likely to know about the restaurants nearby than to get complete information of all those ones throughout the city. While within the region of interest, the specific location of a site is often not particularly important as all sites are not far away. Instead, other multiple attributes of the sites are of importance for comparison and choice. As this is a multi-criteria decision making, a skyline query is perfectly capable of retrieving desired options for the user from those sites within the spatial range.

Typical skyline algorithms so far have been focused on efficient query processing on normal computers without any resource constraints. Whereas, our objective in this chapter is to find appropriate adjustments and configurations that are suitable for skyline computation on individual lightweight devices. We expect proper measures that not only consume less data storage space, but also speed up on-device skyline query processing.

4.3 Data Storage Scheme on Lightweight Devices

To support on-device skyline queries or any other queries, the first task of all is to store the data on a given hand-held device. In our problem, we need to store the relation R on a hand-held lightweight device. Although the storage space available on a hand-held device keeps increasing, it is still meaningful to find space-efficient storage scheme for data on hand-held devices for the following reasons:

1. First, the volume of data also keeps growing up with a considerable speed. This trend consumes more available storage spaces due to the fragments hard to avoid and the possible backup requirement for security purpose.

2. Second, applications for hand-held devices develops very quickly. On hand-held devices, application programs and even runtime codes usually compete with data for limited storage space, which makes the problem more pressing.

3. Last but never least important, proper data storage scheme can speed up data-centric tasks while straightforward schemes are likely to hold them back. As we will show in this chapter, an properly configured data storage does make the on-device skyline query processing faster.

For the reasons aforementioned, we proceed to find proper storage scheme for relation R on mobile hand-held device M.

4.3.1 Existing Storage Schemes for Limited Space

The most straightforward storage scheme is to directly squeeze a relational table into a lightweight device without any specific modifications, and therefore it is called *Flat Storage* (FS) [18, 4]. It simply stores all tuples of a table sequentially

on a device. As Flat Storage Scheme does not use any specific measures, it is the least desirable way in most cases.

Several alternatives so far have been proposed for relational data on lightweight devices with limited space [7, 18, 76]. Next, we carry out a brief review on these existing schemes, and show why they are not suitable for our on-device skyline query processing purpose.

An obvious problem with FS lies in that all duplicated values are stored repeatedly, which causes large space consumption. To remedy this, a pointer based scheme named *Domain Storage* (DS) is introduced for memory resident DBMS [7] and later is adopted into lightweight devices [18]. Basically, DS keeps all unique values of an attribute in a domain, and replaces original values of that attribute with pointers to the corresponding real values stored in the domain. Different attributes also can share a common domain. By grouping duplicated values in the domain(s) and using pointers with shorter length measured in bytes, DS is able to reduce storage space consumption, especially for enumerated types with small number of values.

To facilitate join operation on device with compact indices, a storage scheme called *Ring Storage* (RS) [18] is proposed as a modification of DS. RS does not replace every value of an attribute with a pointer to the value stored in the domain. Instead, it modifies the domain structure by adding to each domain value a value-to-tuple pointer, which points to the first tuple with that value on a specific attribute. Besides, within the table all tuples with the same value on the concerned attribute are linked by inter-tuple pointers, and the last tuple holds the only pointer to the domain value for all tuples. In this way a ring of pointers is formed across both table and domain, and this explains how this storage scheme is named.

To further reduce the space cost incurred by the pointers in DS, *ID Based*

Storage (IS) is proposed [76]. In IS, all distinct domain values are organized in an array, and those tuple-to-value pointers DS keeps in a relation are replaced by identifier values, which are exactly the positioning indexes of the corresponding domain values in the array. IS can be regarded as a dictionary-based value mapping scheme, which uses a compact integer domain $\{0, 1, 2, \ldots, k\}$ to refer to an attribute domain of $k + 1$ distinct values. All identifier values consume minimum number of bytes, which is determined by the length of the domain value array. In order to make storage space consumption more economically, IS even allows the byte length of identifiers to grow or shrink dynamically according to the updated domain size.

Now we consider those aforementioned storage schemes' suitability for the skyline query processing. The crucial part in the skyline computation is the dominance relationship determining for points, which is usually based on value comparison. This means that it is beneficial if values can be retrieved quickly and if comparisons can incur less computing time.

We first point out that RS scheme is not an appropriate for our skyline query processing on lightweight devices. This is because not every tuple has a pointer to value in RS scheme. Rather, on any attribute all tuples of the same value are linked by internal pointers, and only the last tuple has a direct pointer to the shared value. This causes expensive cost to access the real tuple values, since we have to traverse the internal pointer chain to reach the unique tuple with the external pointer. This seriously hurts skyline query processing, which needs tuple values in dominance determining based on value comparison.

We next argue that DS scheme is not an ideal choice either. Though DS provides tuple-to-value pointers for each attribute of each tuple, we also do not use this scheme as it still consumes extra time to use tuple-to-value pointers to access a tuple's attribute values from separate domains. In an operation involving

single or very few attributes, this extra cost might be tolerable. While skyline queries involves a number of attributes, the accumulated extra costs is not negligible. Besides, the relation R in our problem has spatial coordinates which unlikely share common values required by DS as sites of interest are located on different geographic positions.

IS scheme faces the similar difficulties as DS does, since the former is a modification of the latter. However, by differently storing the spatial coordinates and non-spatial attributes in R and properly adapting IS scheme, we are able to achieve a storage scheme suitable for skyline query processing.

4.3.2 Hybrid Storage Scheme

Based on the analysis presented previously, we adopt a hybrid storage scheme for the relation R stored on a mobile device M. Relation R consists of both both spatial and non-spatial attributes. All spatial coordinates are measured and digitized in floating-point numbers, and they seldom share common values. This is attributed to the digitization accuracy and the fact that usually no geographic entities share the same exact location in reality. Lack of duplicated values in spatial attributes make it not effective to store spatial attribute values separately from the tuples, as DS and IS do, since this does not save storage space. Hence, we store for each tuple its spatial coordinate values directly in R, as the FS scheme does. Non-spatial attribute values are likely to be shared among multiple tuples, making it beneficial to store them separately. To facilitate skyline query processing on mobile device M, we use ID based storage (IS) for the non-spatial attributes.

Our choice is justified by several specific measures applicable to IS and beneficial to skyline query processing. First, we sort the domain of every non-spatial attribute. Second, we also sort R on the identifiers of one attribute. This is inspired

by the SFS algorithm [29], and reduces the number of value comparisons in determining dominance during query processing, as the tuple with a smaller identifer also has a smaller real attribute value. We choose the attribute with the largest number of distinct values as the attribute whose identifiers in R are to be sorted. For simplicity and without loss of generality, we assume that p_1 is that attribute we look for. If tie happens on p_1, then the order is determined by comparing values on p_2. Possible ties on further dimensions will be handled in the same way.

As we store the spatial coordinates and non-spatial attributes in R differently, FS for the former and modified IS for the latter, we call this storage scheme *Hybrid Storage* (HS). Our hybrid storage scheme for relation R on a lightweight device is illustrated in Figure 4.2.

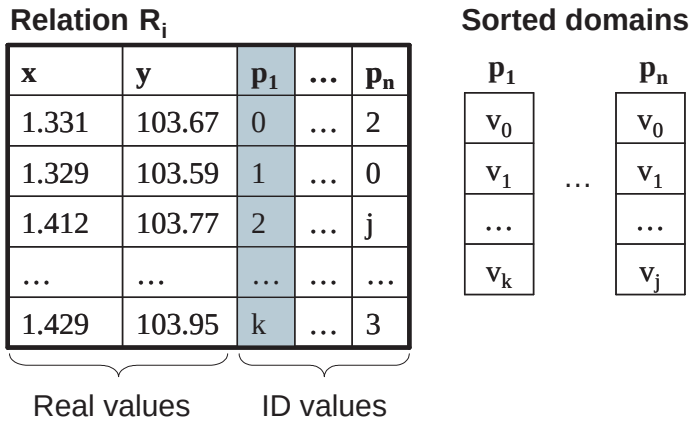


Figure 4.2: Hybrid storage model

4.3.3 Discussion

Those storage schemes presented above can be regarded as means of compression of original relations. The compression effect is accomplished as storage space is saved on lightweight devices. As a matter of fact, our IS-based HS scheme can be regarded as a dictionary-based order preserving (OP) compression scheme [8],

because each domain in HS is sorted and the whole relation is sorted on one selected attribute. As for other relation compression schemes in the literature [35, 88], they are aimed at large scale databases in traditional computing environments instead of mobile lightweight devices, the setting of our problem. In particular, they have a common performance characteristic: their compression reduces IO cost but increases CPU cost, and the former gain is ensured to be larger than the latter loss such that the overall query processing cost is still reduced on a compressed relation. Unfortunately, the IO reduction gain expected by those compression schemes is not available on lightweight devices, where no expensive IOs take place. As a result, the extra processing cost introduced by those relation compression schemes, which is overly offset on normal computers, is not affordable on lightweight devices. This platform particularity makes those compression relation schemes not applicable to our problem.

We intend to accommodate efficient skyline query processing on resource-limited devices. This requires that we consider not only storage space saving but also query processing efficiency. However, conventional database compression schemes [35, 88], which may produce higher compression ratio, need considerable extra processor costs to restore original values when tuples are accessed. Such extra costs are really expensive on lightweight devices, and therefore slow down the targeted skyline query processing. Our dual goals make the HS scheme a very reasonable and favorable choice. Our IS-based HS scheme not only saves storage space, but also helps to reduce processing cost in skyline computing. The latter will be further analyzed in Section 4.4.2.

Another issue is related to the update cost on the hybrid storage scheme. When a new tuple is to be inserted, two steps are to be done. First, a correct ID value should be decided for each attribute. This needs n binary searches, each on an

attribute domain. The average time cost is $n \log_2 k$, if k is the average domain size. Second, the tuple should be placed in a correct position in the relation with respect to the sorted attribute p_1. This on average needs to move $O(N/2)$ tuples, where N is the relation cardinality. When a tuple is to be deleted from the relation, it also needs to move $O(N/2)$ tuples on average. For bulk update, a usual way is to connect the lightweight device to a desktop computer where the update can be executed much more quickly. There are special softwares supporting such synchronization. A typical example is the ActiveSync program Microsoft provides for lightweight devices running Windows Mobile operating system.

4.4 Skyline Computation on Lightweight Device

For several reasons we do not consider index based skyline algorithms, like BBS based on R*-tree. First, considerable extra storage space may be consumed by indices like a R*-tree. Second, our skyline query in the problem definition is a dynamic one [66], which hurts the pruning efficiency of the index based skyline algorithm as the dynamic dimension is not indexed. Third, lightweight devices do not have secondary storage as normal computers do, which causes the benefit gained by the index much less significant than hard-disk based IO reductions on normal computers.

4.4.1 Flat Storage Based Skyline Algorithm with Pre-computed Skyline Points

Recall the *static partial skyline* SK_{ns} we define in Section 3.3.1, which is the skyline concerning all non-spatial attributes. There, we have shown that SK_{ns} is always a part of the final skyline result SK_{all} that takes both distance and non-spatial

attributes into account. For the problem we define in this chapter, SK_{ns} still can be utilized. If no spatial constraint (distance of interest d) is specified in the query Q_{sky}, SK_{ns} is undoubtedly a part of the query result, termed SK_Q. For this extreme case, pre-computing the SK_{ns} will be enough and no realtime query processing will be needed. For a skyline query Q_{sky} with a distance of interest d, some points in SK_{ns} are not wanted as they are too far away from the query's location; others are also part of the result SK_Q, as no other points can dominate them regarding all non-spatial attributes. Note that in SK_Q some points do not come from SK_{ns}. Those points used to be dominated by those skyline points out of the query distance d, but are not dominated when only near points are considered. These points must be found to get correct SK_Q.

Based on the flat storage (FS), we adapt the BNL [19] skyline algorithm to incorporate the SK_{ns} to help filter out dominated points in the scan. We pre-compute the skyline regarding all non-spatial attributes, and for each skyline point sp we add into SK_{ns} its identifier, i.e., its sequential index in the relation R. Given such an identifier we can retrieve the real tuple from the flat storage in $O(1)$ time. Besides, we use an extra bit string for all tuples in the relation R. Each bit in the string corresponds to a tuple based on the sequential index in both string and the relation. If a tuple belongs to SK_{ns} its corresponding bit is 1, otherwise 0.

The algorithm based on the arrangements above is shown in Figure 4.3, called *fltSkyline* algorithm. It first gets those qualified skyline points from the pre-computed SK_{ns} into SK, by checking the distances (line 2-4). Then it carries out a simple loop to determine for every tuple if it is in the result. Those points that belong to S_{ns} or are out of the query distance are skipped (line 6-7). Then S_{ns} is used to filter out dominated points (line 9-12). For a point that passes the filtering, it is compared to the temporary skyline points in SK_{tmp}. Note that during

Algorithm fltSkyline(pos, d, SK_{ns})
Input: pos is the location of query originator
d is the distance of interest
SK_{ns} is the static partial skyline
Output:skyline result of query Q_{sky}

```
1.  SK_tmp = ∅; SK = ∅;
    // Identify partial result from SK_ns
2.  for each tuple identifer i ∈ SK_ns
3.      if (dist(pos, tp_i) <= d)
4.          add tp_i into SK;
5.  for each tuple tp_j in R_i
        // A skyline point in SK_ns or, too far away from query point
6.      if ((bits_string[j] == 1) or (dist(pos, tp_j) > d))
7.          continue;
8.      dominated = FALSE;
        // Filtering with SK_ns
9.      for each tuple identifer i ∈ SK
10.         if (∀l ≥ 1, tp_i.id_l ≤ tp_j.id_l)
11.             dominated = TRUE;
12.             break;
13.     if (dominated)
14.         continue;
15.     for each tuple identifer k ∈ SK_tmp
16.         if (∀l ≥ 1, tp_k.p_l ≤ tp_j.p_l)   // tp_k dominates tp_j
17.             dominated = TRUE;
18.             break;
19.         if (∀l ≥ 1, tp_j.p_l ≤ tp_k.p_l)   // tp_j dominates tp_k
20.             remove k from SK_tmp;
21.     if (!dominated)
22.         add j into SK_tmp
23. return SK_tmp ∪ SK;
```

Figure 4.3: Skyline algorithm with pre-computed SK_{ns}

the loop a point in SK_{tmp} is not ensured to be a final skyline point. Finally the union of SK_{tmp} and SK is returned as the result.

By utilizing SK_{ns}, dominated points can be early detected and thus filtered out. Therefore, the total comparison between tuples are expected to be reduced. Suppose in relation R, a point pt is only dominated by a skyline point $sp \in SK_{ns}$. Consider a pure BNL skyline algorithm is used. If pt is met before sp is added into the temporary skyline list, pt will keep residing there and all subsequent points will be compared with it. This is avoided in the fltSkyline algorithm, which identifies pt early by filtering with Sk_{ns}. If pt is met after sp, two algorithms performs almost the same. Similar analysis also applies to the cases that pt is dominated by more than one point.

4.4.2 Hybrid Storage Based Skyline Computation

In a skyline algorithm without any index, value comparison is the crucial operation that determines the computation time. Our hybrid storage supports efficient value comparison. We adapt the SFS algorithm to the hybrid storage and propose a comparison efficient skyline algorithm, as shown in Figure 4.4. We call it *hsSkyline* algorithm.

In the algorithm, relation R is sequentially scanned to obtain the skyline result SK, which takes advantage of the sorted attribute p_1 and checks the rest of the dimensions only. Two aspects make this algorithm different from the SFS algorithm. First, a spatial distance check is used to exclude tuples too far away from the query position. Second, attribute identifiers are compared instead of real attribute values. Because the domain values of any dimension p_j is stored in a sorted way, the identifiers in relation R exactly reflect the inequality between the real p_j values of different tuples. For instance, suppose two tuples tp_1 and tp_2 have id_{j1} and id_{j2},

Algorithm hsSkyline(pos, d)
Input: pos is the location of query originator
d is the distance of interest
Output:skyline result of query Q_{sky}

```
1.  SK_Q = ∅;
    // Local ID-based skyline processing
2.  for each tuple tp_j in R
        // Too far away from query point
3.      if (dist(pos_org, tp_j) > d)
4.          continue;
5.      dominated = FALSE;
6.      for each skyline point sp_k in SK_i
            // sp_k dominates tp_j
7.          if (∀l > 1, sp_k.id_l < tp_j.id_l)
8.              dominated = TRUE;
9.              break;
10.     if (!dominated)
11.         add tp_j into SK_Q
12. return SK_Q;
```

Figure 4.4: Comparison efficient on-device skyline query processing

respectively, on dimension p_j, and that all domain values are stored in ascending order. Then simply comparing each pair of id_{j1} and id_{j2}, instead of accessing and comparing the real domain values, is enough to determine the dominance between tp_1 and tp_2. This has two benefits. One is that it saves access time since no offset based addressing is needed to access values. The other is that comparison of simple identifier integers generally costs less time than that of real domain values.

One may wonder why we do not use pre-computed SK_{ns} in the hsSkyline algorithm above as we do in the algorithm presented in Figure 4.3. We argue that using S_{ns} for filtering in the hsSkyline algorithm does not help, on the contrary it may cause more unnecessary value comparisons. Because in our hybrid storage the first non-spatial attribute is sorted in the non-descending order, a point pt_i cannot be dominated by any point pt_j that is met behind pt_i in the scan. Now suppose in the fltSkyline algorithm, a point pt_i is being considering and compared with sp_j, a skyline point gained from SK_{ns}. If sp_j is located before pt_i in the relation R, the hsSkyline algorithm will already have identified sp_j and added it into SK_Q. This means the comparison is not missed in hsSkyline algorithm and not saved in fltSkyline algorithm either. If sp_j is located behind pt_i in the relation R, sp_j cannot dominate pt_i and the comparison between them is unnecessary as it does not contribute to the result. If we introduce SK_{ns} for filtering into the hsSkyline algorithm, the algorithm will have to do many unnecessary comparisons of this kind. While in the hsSkyline algorithm above, this comparison between sp_j and pt_i does not happen when we determine if sp_j is a skyline point.

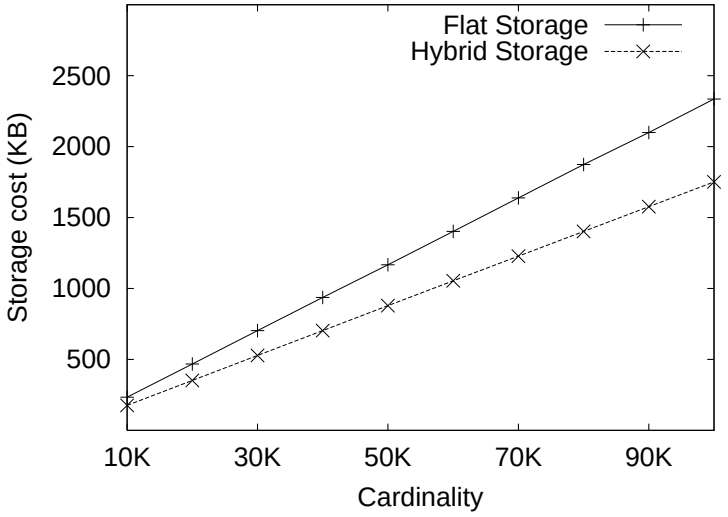
4.5 Performance Studies

In this section we present the results of performance studies on our proposals. All experiments are conducted on an HP iPAQ h6365 pocket PC running MS Windows Mobile 2003. The device has a 200MHz TI OMAP1510 processor and 64MB SDRAM (55MB user accessible). All programs are written using SuperWaba, a Java-based open-source platform for PDA and Smartphone application development [3]. The parameters and their corresponding settings used in the experiments are listed in Table 5.6. These settings in bold are the defaults, used when their corresponding parameters are not varied.

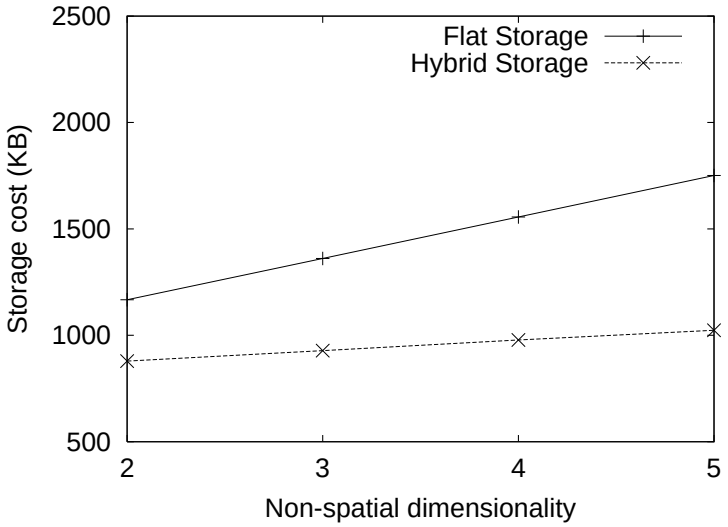
Parameter	**Setting**
Cardinality of local relations	10K, ..., **50K**, ..., 100K
Storage model for local relations	Flat, Hybrid
Number of non-spatial attributes	**2**, 3, 4, 5
Non-spatial attribute domain range	[0.0, 9.9]
Spatial extent of local relation	1000.0 × 1000.0
Attribute distribution	Independent, Anti-Correlated
Query distance of interest	100, 250, 500

Table 4.1: Parameters used in experiments

For tests on dataset cardinality, the datasets contain 10K to 100K points with two non-spatial attributes. For tests on non-spatial dimensionality, the datasets contain 50K points with two to five non-spatial attributes. Each non-spatial attribute is of float type and its domain is {0.0, 0.1, ..., 0.9, 1.0, ..., 9.0, ..., 9.9}. Since each domain contains 100 distinct values, we use byte type IDs in the hybrid storage implementation. Synthetic datasets with both independent and anti-correlated distributed attributes are used. We first investigate into the storage space cost of our hybrid storage scheme, and then study the time efficiency of the skyline query processing algorithm based on the hybrid storage scheme.



(a) Space cost v.s. cardinality



(b) Space cost v.s. dimensionality

Figure 4.5: Storage space costs

4.5.1 Storage Space Cost

For on-device storage space cost, we compare the hybrid storage scheme with the flat storage scheme, as the latter is very simple and used most possibly when users do not bother to do any on-device adjustments. The results on space cost are shown in Figure 4.5. Datasets with both independent and anti-correlated attributes are used, but tests on two distributions exhibit close results. Therefore we only show the results on the independent distribution.

It is seen from Figure 4.5(a) that our hybrid storage scheme saves storage space compared to the flat storage scheme. As the dataset cardinality grows up, the space saving by the hybrid storage scheme also goes up steadily. This is because a larger cardinality causes IDs to be used more often in the hybrid storage scheme, which consume less space than the raw data in the flat storage scheme.

Similar space savings and differences are observed when the dimensionality increases, as shown in Figure 4.5(b). The gap becomes larger as dimensionality increases, because a larger number of attributes also makes simple IDs to be used more frequently in the hybrid storage scheme.

4.5.2 Skyline Query Processing Time

For on-device skyline query processing time cost, we compare three algorithms. The first and simplest one is the BNL algorithm on the flat storage scheme. The second one is the algorithm fltSkyline presented in Figure 4.3, which is also based on the flat storage scheme. And the last one is the hsSkyline algorithm we propose for our hybrid storage scheme, i.e., the one presented in Figure 4.4. No extra index is used for either storage scheme.

Although it is theoretically possible to combine any of BNL, fltSkyline and hsSkyline with either storage scheme, we only compare the three combinations

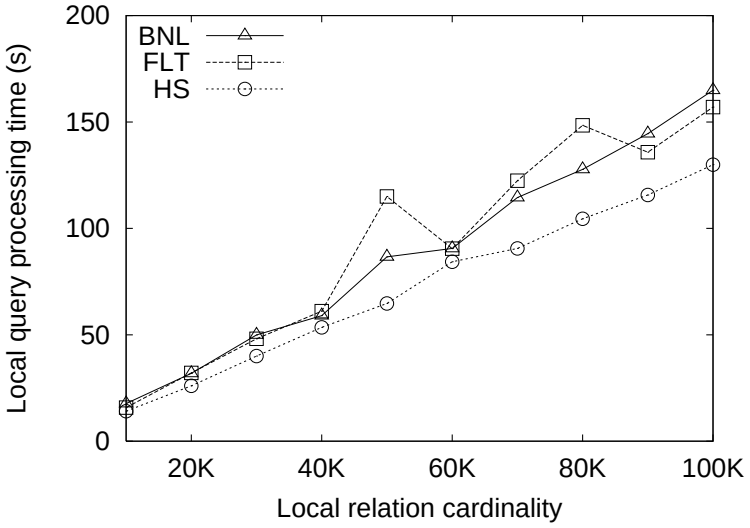
Table 4.2: Reasonable algorithm/storage combinations

Algorithm \ Scheme	FS	HS
BNL	$\surd$	–
fltSkyline	$\surd$	–
hsSkyline	$\times$	$\surd$

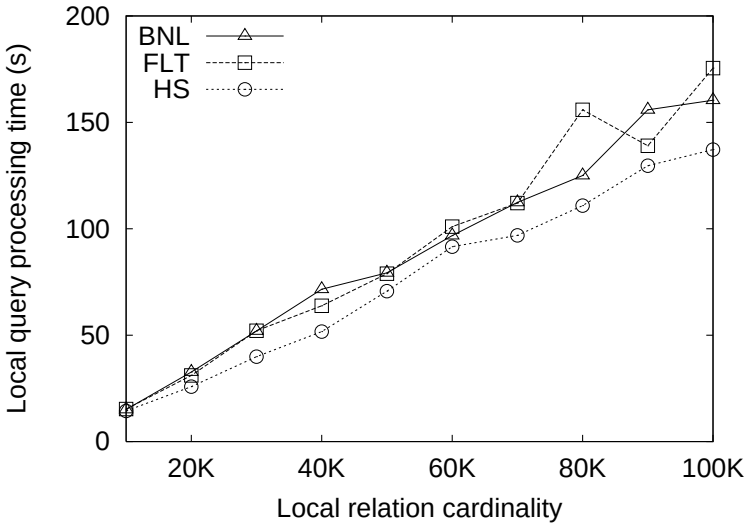
aforementioned, as shown in Table 4.2. The choices are justified by the reasons that follows. First, applying algorithm BNL on HS does not perform better than hsSkyline on HS. Given the same sorted relation processed by SFS based hsSkyline, BNL incurs much more unnecessary value comparisons and maintains intermediate skyline candidates that are not in the final skyline, according to the research on SFS [29]. Second, following the argument presented in the last paragraph of Section 4.4.2, applying algorithm fltSkyline or filtering with pre-computed skyline points on HS does not get any improvement but probably some unnecessary comparisons. Third, algorithm hsSkyline, which requires a relation to be sorted specifically, is definitely not applicable to FS that does not sort the relation at all.

We do not take into consideration those earlier main-memory algorithms for the maximal vector problem [16, 52, 15]. All those algorithms are divide-and-conquer based algorithms. Our decision is mainly justified by a detailed comparative research conducted by Godfrey et al. [34]. The authors disclose that such divide-and-conquer based algorithms do not perform better than the scan-based skyline algorithms such as BNL and SFS, either in the main-memory environment or on the secondary storage. Therefore, we here focus on comparing different scan-based skyline algorithms on specific storage schemes for lightweight devices.

We investigate into the impacts of dataset cardinality and dimensionality on the query processing efficiency. For each cardinality or dimensionality setting, 10 skyline queries are issued and the average processing cost is recorded. For each query, its position *pos* is randomly determined within the spatial extent, and its

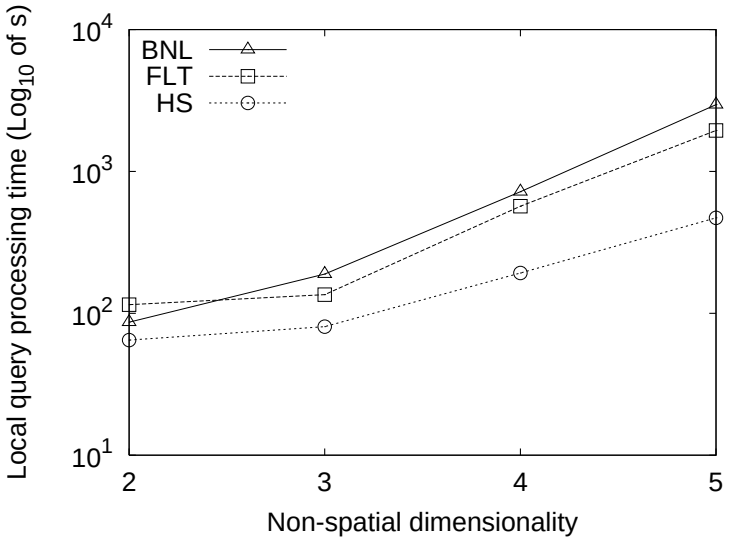


(a) Independent attributes

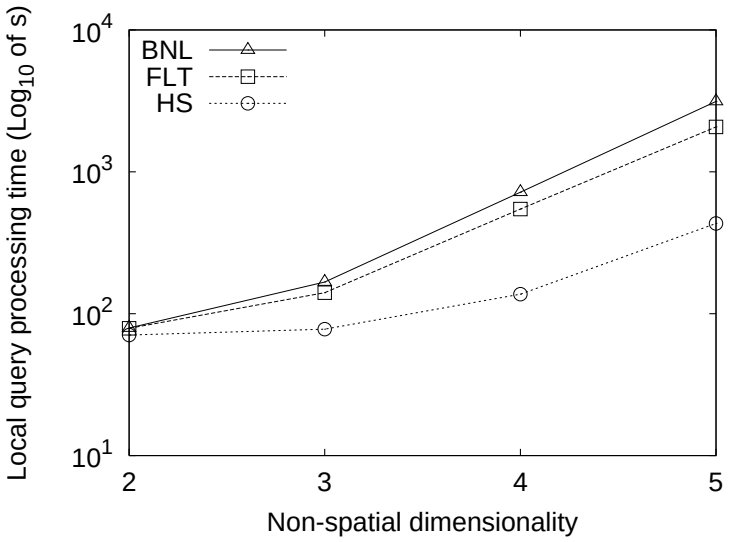


(b) Anti-correlated attributes

Figure 4.6: Skyline query processing time v.s. cardinality



(a) Independent attributes



(b) Anti-correlated attributes

Figure 4.7: Skyline query processing time v.s. dimensionality

distance of interest d is randomly picked from 100, 250 and 500.

Figure 4.6 shows how the query processing time of each algorithm is affected by the dataset cardinality, where both independent and anti-correlated datasets are used. It is not surprising that hsSkyline algorithm (denoted as HS) needs less query processing time than both BNL algorithm and fltSkyline algorithm(denoted as FLT). This is because in HS, most comparisons are between simple IDs of type byte, whereas in BNL and FLT, all comparisons are between float raw values which consume more time. In addition, HS keeps the domains and the first attribute sorted, which is also helpful in speeding up skyline computation.

Figure 4.7 shows how the query processing time of each algorithm is affected by the non-spatial dimensionality. It can be seen that HS still outperforms BNL and FLT, and the gap here is marked and steady. This indicates that our hybrid storage scheme and hsSkyline algorithm are robust to the increase of dimensionality.

Referring to Figure 4.6, the performance difference between pure BNL algorithm and fltSkyine algorithm is not significant. This is because for datasets with two non-spatial attributes the pre-computed skyline size is small, and therefore the filtering effect is not strong. When the dataset dimensionality increases the filtering effect becomes clear, as shown in Figure 4.7. For anti-correlated datasets, the gain by fltSkyline increases as the dimensionality goes up. This is attributed to the fact that a higher dimensionality produces more additional skyline points for an anti-correlated distribution than for an independent distribution.

4.6 Summary

In this chapter, we focus on processing skyline queries efficiently on individual resource-constrained mobile lightweight devices. A mobile device user may issue

skyline query with distance of interest to understand the surroundings, where sites of interest are abstracted as points with both spatial coordinates and non-spatial attributes stored on the device. Directly squeezing both data and algorithms from normal computers into such lightweight devices is not an efficient way. We propose some specific measures for efficient skyline query processing on such devices.

After an analysis on existing data storage schemes for resource-limited devices, we propose a hybrid storage scheme which deals differently with spatial coordinates and non-spatial attributes. Based on that, we are able to propose a on-device skyline algorithm that carries out efficient skyline computation with less and faster comparisons. Besides, we also adapt the BNL skyline algorithm for the simple flat storage case. We incorporate those pre-computed skyline points to early filter out dominated points. Through experiments on a real pocket PC device, we show that our hybrid storage proposal not only saves on-device storage cost but also speeds up on-device skyline query processing.

In this chapter we focus on skyline query processing on individual lightweight devices without any inter-connections. A bolder conception is to let multiple devices communicate with each other and share data through distributed queries. We will address this novel problem in the next chapter.

CHAPTER 5

Skyline Queries Against Mobile Lightweight Devices in MANETs

While skyline querying is gaining in interest, most previous work on skyline querying has assumed the traditional, centralized data storage. The growing popularity of mobile computing devices challenges this assumption. Assuming a mobile setting, this chapter considers skyline querying from a new and unexplored angle. In this setting, each mobile device is capable of holding only a portion of the whole dataset; devices communicate through mobile ad hoc networks (MANETs); and a query issued by a mobile user is interested only in a fixed area, although a query generally involves data stored on many mobile devices due to the storage limitations. We propose a distributed skyline query processing strategy that aims to reduce the communication costs among mobile devices in a MANET. In our strategy, skyline query requests are forwarded among mobile devices in a deliberate way, such that the amount of data to be transferred is reduced. Besides, specific measures are proposed for resource-constrained mobile devices, in order to make the whole system work properly. We conduct extensive experiments to show that our proposal performs efficiently in simulated wireless ad hoc networks.

5.1 Introduction

With the continued advances in electronics and wireless communications, more and more mobile handsets with computing and wireless networking capabilities are being deployed. For example, mobile handsets are being equipped with infrared, Bluetooth, or even Wi-Fi capabilities. Further, positioning capabilities are becoming available on handsets, based, e.g., on GPS, the communication infrastructure, or a combination.

In particular, handsets are now being equipped with wireless peer-to-peer (P2P) capabilities. This enables handsets to become parts of self-organizing, wireless mobile ad hoc networks (MANETs) that allow seamless, low-cost, and easily deployed communications [11, 20]. It is conceivable that ad hoc and P2P technologies will be combined to bring about wireless communications without the presence of central servers [5].

Within relational data management, most previous work on skyline queries [19, 82, 51, 29, 66] has assumed that the data is stored in a centralized fashion. However, the developments just outlined make it relevant to consider a much more distributed setting: processing skyline queries in a MANET.

This chapter considers skyline querying in a mobile context with the following assumptions: 1) each mobile device only holds a portion of the entire dataset; 2) devices communicate through MANETs; 3) and mobile users posing skyline queries are only interested in data pertaining to a limited geographical area, although the queries involve data stored on many mobile devices due to the storage limitations of the devices.

Consider the example skyline query shown in Figure 5.1. Here, M_1 to M_4 are four mobile devices, each having data corresponding to different portions of the geographical space within which these devices move. Device M_2 is interested in the

region represented by the circle, the data of which is held on all four mobile devices. Thus, the skyline query of M_2 involves all four mobile devices. The problem here is different from the one we address in Chapter 4, where a skyline query is issued on a device against its local data only.

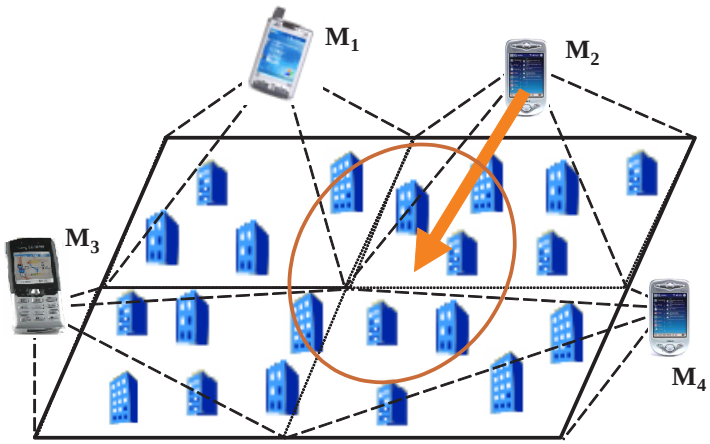


Figure 5.1: Skyline query on mobile devices in a MANET

To improve the efficiency of such distributed skyline queries, two most important costs need to be considered: the cost of the communication among the mobile devices and the cost of query execution on the mobile devices. To reduce the former cost, a skyline query request is sent to the mobile devices involved in a deliberate way such that the amount of data to be transferred is minimized. To reduce the latter, specific measures are proposed for resource-constrained mobile devices. Our experimental study shows that the proposed method is efficient in terms of both communication cost and response time.

The chapter makes the following contributions: First, it identifies and formalizes the problem of skyline querying in MANETs. Second, it proposes distributed processing strategies that aim to reduce the data to be transferred during the processing of queries. Third, it proposes necessary measures to ensure the proper query strategy and speed up local processing on resource-constrained devices. Fourth, it

reports on extensive experiments conducted on a MANET simulator.

The remainder of this chapter is organized as follows. Section 5.2 defines the problem to be solved. Section 5.3 presents two unsophisticated distributed skyline query processing strategy in a MANET. Section 5.4 proposes our data transmission efficient distributed skyline query processing strategy in a MANET. Section 5.5 presents necessary local configurations on each resource-constrained device. Section 5.6 experimentally evaluates the proposed techniques, followed by the summary of this chapter in Section 5.7.

5.2 Problem Definition

We assume a MANET setting with m mobile devices $M = \{M_1, M_2, \ldots, M_m\}$. Each device M_i holds a database relation R_i that contains the data that pertains to sites located in a small geographical area. All R_is on all devices conform to the same schema $<x, y, p_1, p_2, \ldots, p_n>$, where (x, y) represents the location of a site and the p_1 to p_n are attributes describing a site. The contents of different R_is may overlap, i.e., it is possible that $R_i \cap R_j \neq \emptyset$ for any $i \neq j$. We may also envision a global (and virtual) relation R, such that $\bigcup_{i=1}^{m} R_i = R$.

In this setting, a mobile device M_i can ask a distributed skyline query, whose result consists of all sites s in global relation R that satisfy these conditions: (a) site s is within distance d of M_i's current position; (b) site s is in the skyline of R' in terms of all attributes p_i, where R' is the set of all sites satisfying condition (a). Such a query can be defined as $Q_{ds} = (id, pos_{org}, d)$, where id is the identifier of the device M_{org} issuing the query, pos_{org} is the location of M_{org}, and d is the distance specifying the region of interest.

The query in the problem defined above has the same semantics as that in the

previous chapter. However, in this chapter we assume that the data of interest is distributed across multiple devices connected by a MANET, instead of being totally available on the single device issuing the query. At least two practical possibilities justify this assumption. On one hand, a single mobile device may not always have the complete information about the whole region of its possible activities. Reasons for this can be local space limitation, restricted data availability, or other possible ones. On the other hand, mobile devices are capable of exchange data via MANETs as long as they are equipped with appropriate networking interfaces..

The problem we consider differ from traditional skyline queries. First, a region of interest is specified in the query, which we have discussed in Section 4.2. Second, horizontal partitions of the global relation are distributed across different mobile devices, which is different from the distributed skyline queries on vertical partitioning [10]. Third, the wireless communication channels between mobile devices are slow and unreliable, in comparison to wired connections. Finally, because different R_is on different mobile devices may overlap, duplicate elimination must be applied before results are returned.

Together, the above features pose several challenges to our distributed skyline queries. Proper distributed skyline query strategy is expected for horizontal partitions. And because they are distributed in a dynamic MANET environment, the amount of data transferred between devices should be reduced. Besides, on those lightweight devices, necessary configurations are needed to ensure the success of the whole paradigm.

In the target environment, both communication costs and processing costs on the mobile devices are important to the overall query performance. Indeed, these two aspects determine the performance of a query. With this in mind, we provide processing methods that can reduce the amount of data to be transferred among

mobile devices and speed up the query processing on a single mobile device. Symbols to be used throughout this chapter are summarized in Table 5.1.

Symbol	Description
m	Total number of mobile devices
M_i	One mobile device
n	Number of attributes of a tuple
p_i	ith non-spatial attribute of a tuple
R_i	Local relation on M_i
R	Virtual global relation, union of all R_is
M_{org}	The mobile device originating query Q_{ds}
Q_{ds}	Distributed mobile skyline query
pos_{org}	Location of query originator M_{org}
d	Distance of interest in query Q_{ds}
SK_i	Local skyline on M_i w.r.t. Q_{ds}
SK_i'	Reduced local skyline on M_i w.r.t. Q_{ds}
SK	Final skyline w.r.t. Q_{ds}
FSK	$\bigcup_{i=1}^{m} SK_i - SK$
FSK_i	$SK_i - SK$
tp_{flt}	Tuple used for filtering
$dom(p_i)$	The set of values of attribute p_i

Table 5.1: Symbols used in discussion

5.3 Unsophisticated Distributed Skyline Processing in MANETs

In this section, we present two unsophisticated strategies for distributed skyline processing in MANETs, as preliminaries and motivations for our efficient strategy to be presented in Section 5.4.

5.3.1 Naive Strategy

A naive method for computing our skyline queries is for the query originator to get all local relations from all other devices and then perform the query locally.

The local query can be performed after all relations have been received, or it can be repeated each time a single relation has arrived. This method may work in a wired, distributed environment, although it is unlikely to yield good performance. In a mobile setting, at least two limitations make this method infeasible. One is that wireless connections between mobile devices have low bandwidths. Therefore, transferring an entire local relation R_i from M_i to M_{org} is too time consuming, especially when R_i is large. The other is that device M_{org} lacks the capacity needed to store all the relations received from the other devices. Even using the incremental approach, an extra relation may still use up the limited storage, which in some mobile devices is also used as memory for computations. The storage limitation is likely to slow down the naive method or even make it impractical. Thus, we need to look beyond the naive method.

5.3.2 Straightforward Strategy

We observe that it is not necessary to obtain all relations from all other mobile devices—only those tuples that may potentially appear in the final skyline are needed. Based on this observation, we can reduce the amounts of data to be transferred as follows: query originator M_{org} sends the query specification, $Q_{ds} = (id, pos_{org}, d)$, rather than simple data requests. On receiving the specification, device M_i does a skyline query on its relation R_i and then sends M_{org} the result only, not the entire R_i. Device M_{org} also computes a local skyline for its relation after sending out the query specification. Later, it merges each result it receives with the previous result in an incremental fashion, while also removing the non-qualifying tuples.

This method reduces the amount of data to be transferred at the cost of local computations at each M_i. Suppose for relation R_i on mobile device M_i that the

result of skyline query Q_{ds} is SK_i. Then the reduction ratio of the data transferred is $(|R_i| - |SK_i|)/(|R_i|) = 1 - |SK_i|/|R_i|$. Clearly, the more selective a local skyline query is, the smaller the communication cost.

Although the use of local skyline queries can reduce the amount of data sent back to the query originator, there is still a chance that a local result may contain tuples that do not belong to the final skyline. In other words, $\bigcup_{i=1}^{m} SK_i = SK$ does not hold. Instead, the union of all SK_is is a superset of SK. The set $\bigcup_{i=1}^{m} SK_i - SK$ contains all those tuples that appear in one or more local skylines SK_i, but not in the final skyline SK, and we use FSK to represent it. Transmitting $SK_i - SK$ from mobile device M_i to query originator M_{org} is a waste because it does not affect the final query result. We use FSK_i to represent that subset $SK_i - SK$ on each mobile device. If we can reduce each FSK_i, we can also reduce the communication cost. As an extreme, if each FSK_i is empty, $\bigcup_{i=1}^{m} SK_i = SK$ will hold, and communication cost is at its lowest. Unfortunately, this is almost impossible practically because it implies a very special data distribution: the final skyline SK must be perfectly partitioned among all mobile devices, and every tuple in M_i not belonging to SK has at least a dominator in SK within the same local relation R_i.

In distributed join processing, semi-join [97] works as follows. A smaller projection of one relation is first sent from one site to its peer site, where it is used to reduce the tuples to be sent to the first site, so that the total communication cost of joining the two relations is reduced. In our setting, FSK_i and FSK provide indications as to where and how to reduce the communication cost. We proceed to present a strategy that is based on the analysis above and is inspired by the semi-join processing approach.

5.4 Efficient Distributed Skyline Processing in MANETs

5.4.1 Filtering Tuple Based Strategy

Since in each local skyline SK_i there may exist non-qualifying tuples for the final skyline SK, it may be helpful to identify these and prevent them from being transmitted. If a tuple tp_i in SK_i does not appear in SK, there must be at least one tuple tp_j in SK and not in SK_i that dominates tp_i. If we can know tp_j when doing the local skyline query on device M_i, tp_i can be removed from the result. We use SK_i' to represent SK_i from which some (maybe not all) tp_is of this kind have been removed. In this way, the number of reduced tuples for transmission is $|SK_i| - |SK_i'|$.

The problem is now how to get such tp_js for a mobile device M_i. Such a tp_js must come from somewhere else than M_i. Since we do a local skyline query on query originator M_{org}, we can pick possible tp_js from its local result. We use SK_{org} to represent the initial, local skyline on M_{org}. Which tuple to choose from SK_{org} is then of interest. To address this issue, we need to consider for a tuple tp_j in SK_{org} its ability to dominate and then remove other tuples.

For a tuple tp_j ($<p_{j1}, \ldots, p_{jn}>$), its ability to dominate other tuples is determined by its own values and the boundaries of the data space, i.e., the hyper-rectangle whose diagonal is the line segment with tp_j and the maximum corner of the data space as coordinates. Any other tuple that resides in that region is dominated by tp_j and is excluded from the skyline. For this reason, we call that hyper-rectangle the *dominating region* of tuple tp_j. Intuitively, the larger tp_js dominating region is, the more other tuples are dominated by tp_j because a larger hyper-rectangle covers more tuples in the data space, especially when the tuples

are distributed independently. For simplicity, we use a 2-D illustration as shown in Figure 5.2 in our discussion.

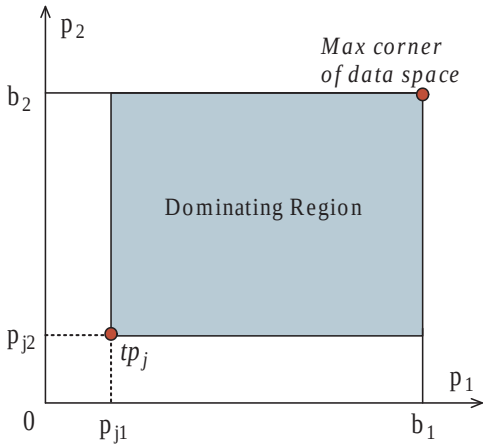


Figure 5.2: Dominating Region

Suppose the value range on dimension p_k is $[s_k, b_k]$ in the virtual global relation R. Then the volume of tuple tp_js dominating region is $VDR_j = \prod_{k=1}^{n}(b_k - p_{jk})$. We choose the tuple, termed tp_{flt}, from SK_{org} with the maximum VDR_j value and use this tuple to filter out non-qualifying tuples. Then instead of sending only the query specification to the mobile devices, we include also tp_{flt}. Each device M_i will then use tp_{flt} to prune non-qualifying tuples during its local skyline query processing. This tp_{flt} is called *filtering tuple* or *filtering point* interchangeably throughout the remainder of this thesis.

Because we add a tuple to what we send to the mobile devices from query originator M_{org}, the communication savings is $|SK_i| - |SK_i'| - 1$ for one M_i. It can be seen that if tp_{flt} fails to remove any non-qualifying tuples from SK_i, the communication cost is actually increased by one tuple. However, we expect that in total, this strategy will be competitive. That is, we expect that $\sum_{i=1,i\neq org}^{m}(|SK_i| - |SK_i'| - 1) = \sum_{i=1,i\neq org}^{m}(|SK_i| - |SK_i'|) - m + 1 > 0$.

As an example, two mobile devices M_1 and M_2 hold hotel relations R_1 and R_2,

hotel	**price**	**rank**
h_{11}	20	7
h_{12}	40	5
h_{13}	80	7
h_{14}	80	4
h_{15}	100	7
h_{16}	100	3

Table 5.2: Example relation R_1

hotel	**price**	**rank**
h_{21}	60	3
h_{22}	80	2
h_{23}	120	1
h_{24}	140	2
h_{25}	100	4

Table 5.3: Example relation R_2

respectively, as shown in Tables 5.2 and 5.3. Assume M_2 is the query originator that wants information on cheap and good hotels. In both relations, each hotel tuple has attributes for the price and the rank based on recommendations of previous visitors. The skyline on M_2 is $\{h_{21}, h_{22}, h_{23}\}$, whereas that on M_1 is $\{h_{11}, h_{12}, h_{14}, h_{16}\}$. If no filtering tuple is used, all four tuples in M_1's local skyline are transferred to M_2. To use a filtering tuple, assuming the global upper bounds of price and rank are 200 and 10, respectively. We need to pick a filtering tuple from M_2's local skyline $\{h_{21}$, h_{22}, $h_{23}\}$. Using the *VDR* definition from above, we have $VDR_{21} = (200-60)*(10-3) = 980$, $VDR_{22} = (200-80)*(10-2) = 960$, and $VDR_{23} = (200-120)*(10-1) = 720$. Because h_{21} has the largest *VDR* value, we choose it as the filtering tuple. This tuple eliminates h_{14} and h_{16} from M_1's local skyline. As a result, the amount of data transferred to M_2 is reduced by two, and the total savings are one tuple. As the cardinality of R_i increases, more non-qualifying tuples can be identified from the local skyline using a filtering tuple.

5.4.2 Estimated Dominating Region

In the above, we have assumed that the global domain range of any attribute p_j is known on mobile device M_i. This ensures exact computation of the dominating region. Sometimes, the global domain range may be unknown to M_i. In this case, we can compute over-estimated and under-estimated dominating regions for a given tuple tp_i.

Over-estimation of the dominating region for tuple tp_j is achieved using formula $VDR_o = \prod_{k=1}^{n}(\max_k - p_{jk})$, where $\max_k$ is a pre-specified value larger than the global domain upper bound b_k, or the largest possible value of the attribute value type. Underestimation is done using $VDR_u = \prod_{k=1}^{n}(h_k - p_{jk})$, where h_k is the local maximum value of attribute p_k known to M_i. The estimations of the dominating region are shown in Figure 5.3. Note that neither over- nor under-estimation affect the correctness of query results. They possibly pick different filtering tuples and therefore have different filtering abilities. Section 5.6 explores this aspect.

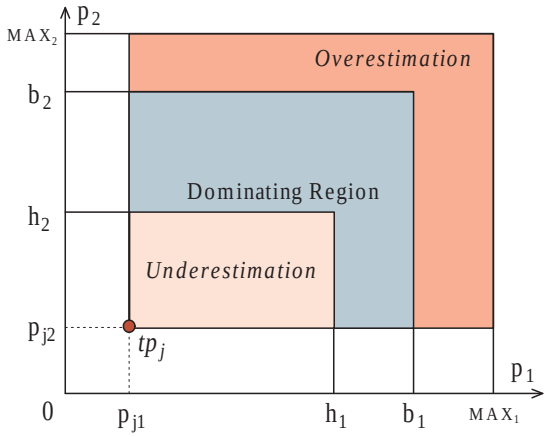


Figure 5.3: Estimated Dominating Region

5.4.3 Adaptations to Wireless Ad Hoc Networks

So far, we have assumed that the query originator can directly communicate with any other mobile device. In that setting, the filtering tuple decided by the originator is used for all mobile devices. In a real MANET setting, however, communication is more likely to be accomplished using multiple hops via intermediate mobile devices if source and destination cannot contact each other directly. In such a setting, we need to adapt our strategy to reduce the communication cost.

First, tp_{flt} is dynamically updated during the procedure of query relay to increase its pruning potential. After doing a local skyline query on mobile device M_i, the tuple tp'_{flt} with the maximum VDR value is obtained from the local skyline result SK_i. Then tp'_{flt} is compared with the current tp_{flt}, and the one with the larger pruning potential, i.e., the larger VDR value, will be used as the new filtering tuple for other mobile devices to which M_i will forward this query.

hotel	**price**	**rank**
h_{31}	60	3
h_{32}	80	5
h_{33}	120	4

Table 5.4: Example relation R_3

hotel	**price**	**rank**
h_{41}	80	2
h_{42}	120	1
h_{43}	140	2

Table 5.5: Example relation R_4

As an example consider the three mobile devices M_1, M_3, and M_4, whose relations are shown in Tables 5.2, 5.4 and 5.5. This time, we assume M_4 is the query originator and M_3 is the intermediate in-between M_4 and M_1. The local skyline on M_4 is $\{h_{41}, h_{42}\}$ and that on M_3 is $\{h_{31}\}$. Based on the VDR values, h_{41} on M_4

is chosen as the filtering tuple and is sent to M_3. If the filtering tuple is not dynamically adjusted, h_{41} will be sent to M_1 as well, where it will eliminate h_{16} only. If the filtering tuple is dynamically adjusted, h_{31} on M_3 will be used as the new filtering tuple because $VDR_{31} > VDR_{41}$. When h_{31} is sent to M_1, it will eliminate both h_{16} and h_{14}. This example shows how dynamic updates of the filtering tuple can make a difference.

Second, a check is made in every mobile device to avoid processing the same query more than once. To support this check, a tag *cnt* is added to every query, which then becomes (id, cnt, pos_{org}, d). This tag is a local count, generated by each query originator. In other words, each mobile device maintains a count for all queries it issues, and this count is attached to the corresponding query. When a mobile device M_i receives a query (id, cnt, pos_{org}, d), it will check its log to see if this query has been processed. If not, M_i processes this query and forwards it to others. Otherwise, the query is ignored.

To save communication cost, this count *cnt* can be defined as a byte, allowing a device to generate 256 queries. After a period of time, e.g., one day, this count can be reset. The log on a device keeps for every device its last arriving query's count *cnt*, which is implemented simply by a hash table that maps the device identifier *id* to a count. On each mobile device, the worst case space cost of such a hash table is $O(m)$, where m is the total number of mobile devices. The time cost of the check is $O(1)$. This mechanism works well under the assumption that a mobile device is only interested in its latest query.

5.5 Local Configurations on Mobile Devices

In this section, we present some necessary local configurations on mobile devices to make our distributed skyline query processing strategy work properly and efficiently.

5.5.1 Dataset Storage

Basically, we store every local relation R_i on each mobile device M_i according to the hybrid storage scheme we have proposed in Section 4.3.2. Additionally, to support fast spatial range check, the maximum and minimum spatial coordinates are kept as constants in x_{max}, y_{max}, x_{min} and y_{min}. These coordinates specify the minimum bounding rectangle MBR_i of all sites in R_i.

In the hybrid storage, all attribute domain values are stored in an array for each attribute. This arrangement can be used to facilitate local skyline query processing on a mobile device. Suppose on mobile device M_i the value range of attribute p_j is $[l_j, h_j]$. Such an l_j or h_j value can be fetched on a mobile device in $O(1)$ time, if we store all available values in each domain sequentially and sorted (either in ascending or descending order). We assume that smaller values are preferred for each attribute p_i, and all values in each attribute domain are stored in ascending order. In the local skyline computing presented next, we will show how to take advantage of these range values.

5.5.2 Local Skyline Computing

With the domain information and tp_{flt} received from the query originator, we can determine efficiently whether we need to do a local skyline on mobile device M_i. If on every attribute, we have $tp_{flt}.p_j \leq l_j$, which means that the best tuple

(potentially, maybe unavailable) on M_i is dominated by tp_{flt}, no tuple in relation R_i will be in final skyline SK. All these comparisons cost $O(n)$ time, where n is the number of attributes. In this case, M_i needs to do nothing but return a correct, short message to M_{org}. Thus, local computation is saved significantly. If the range check does not imply that the whole R_i is dominated by tp_{flt}, we need to do a local skyline query, as we have presented in Section 4.4.2.

The algorithm for a local skyline query on mobile device M_i is presented in Figure 5.4. Initially, a spatial range check is done to see if the spatial extent covered by mobile device M_i overlaps with the query region specified in Q_{ds}. If not, the processing stops. Otherwise, each attribute domain range's lower bound is checked against the filtering tuple tp_{flt}, to determine whether the entire local relation R_i is dominated by tp_{flt}. If not, R_i is sequentially scanned to obtain the local skyline SK_i, which takes advantage of the sorted attribute p_1 and checks the rest of the dimensions only. After getting the local skyline SK_i, the filtering tuple is used again to filter out non-qualifying tuples. While filtering, the skyline point with the maximum VDR is recorded. Finally, if the recorded VDR value is larger than that of the current filtering tuple, which indicates that the recorded skyline point has more powerful ability to dominate other points, that skyline point will be used as the new filtering point to replace the old one. Then, when the device continues to forward the query to other devices, the new filtering tuple will be attached in the request instead.

5.5.3 Assembly on Query Originator

When receiving results back from devices, the query originator needs to combine them properly with its own local skyline to obtain the correct global skyline. Assembly involves two tasks. The one is to remove all non-qualifying tuples that are

Algorithm local_skyline(pos_{org}, d, tp_{flt})
Input: pos_{org} is the location of query originator
d is the distance of interest
tp_{flt} is the filtering tuple
Output: reduced local skyline
updated filtering tuple

```
    // Check if R_i's MBR overlaps the query region
1.  if (mindist(pos_org, MBR_i) > d)    return;
    // Check if R_i is dominated by the filtering tuple
2.  skip = TRUE;
3.  for each attribute j of R_i
4.      if (tp_flt.p_j > l_j)
5.          skip = FALSE;   break;
6.  if (skip)   return;   else   SK_i = ∅;
    // Local ID-based SFS processing
7.  for each tuple tp_j in R_i
        // Too far away from query point
8.      if (dist(pos_org, tp_j) > d)    continue;
9.      dominated = FALSE;
10.     for each skyline point sp_k in SK_i
            // sp_k dominates tp_j
11.         if (∀l > 1, sp_k.id_l < tp_j.id_l)
12.             dominated = TRUE;   break;
13.     if (!dominated)
14.         add tp_j into SK_i
    // Filtering, and picking up maximum VDR
15. idx = null; VDR_m = 0;
16. for each skyline point sp_k in SK_i
17.     if (∀l, tp_flt.p_l < sp_k.pl)
18.         remove sp_k from SK_i
19.     else if (VDR_k > VDR_m)
20.         idx = k;    VDR_m = VDR_k
    // Update filtering tuple if necessary
21. if (VDR_m > VDR_flt)    tp_flt = tp_idx;
```

Figure 5.4: Local skyline query processing on M_i

present in both an incoming result and M_{org}'s own local result. The other is to remove all duplicate tuples in the final skyline result. Both tasks can be done within a simple nested loop, i.e., for each tuple tp_j in an incoming result SK'_i, every tuple tp_k in the local current result SK_{org} is checked. Duplicates can be identified by checking the x and y values only, since we assume that no two tuples represent the same geographic location. If tp_j and tp_k are not the same, the dominance between them is determined by checking all their non-spatial attributes. In this way, SK'_i and SK_{org} are merged together correctly to produce the updated SK_{org}.

5.6 Performance Studies

We proceed to offer insight into the properties of the method proposed in this chapter based on experimental studies. The parameters used in the experiments are listed in Table 5.6.

Parameter	**Setting**
Number of total mobile devices	$3^2, \ldots, 5^2, \ldots, 10^2$
Cardinality of global relation	100K, ..., 500K, ..., 1000K
Cardinality of local relations	10K, ..., 50K, ..., 100K
Storage model for local relations	Hybrid
Number of non-spatial attributes	$2, 3, 4, 5$
Non-spatial attribute domain range	[0, 1000], [0.0, 9.9]
Spatial extent of global relation	1000.0×1000.0
Attribute distribution	Independent, Anti-Correlated
Query distance of interest	100, 250, 500

Table 5.6: Parameters used in experiments

5.6.1 Experimental Settings

We test our proposed methods in a MANET environment simulated using JiST-SWANS [1], a Java-based wireless ad hoc network simulator. We consider three

main performance aspects: (1) the data reduction efficiency of the distributed query processing strategy; (2) the overall response time; and 3) the number of messages used to forward the query between mobile devices. The simulation experiments have been conducted on a Pentium IV desktop PC running MS Windows XP with a 2.99GHz CPU and 1GB main memory.

We use global relations of 100K to 1M. In each global relation, all tuples are distributed randomly within a 1000×1000 spatial domain. Based on a uniform grid on the spatial domain, a global relation R is divided into local relations (R_is), each containing all the tuples within its corresponding grid cell.

We vary the total number of mobile devices (m) as the squares of the numbers 3 to 10, i.e., $\{9, 16, 25, 36, 49, 64, 81, 100\}$, with each device containing the data of a grid cell. The number of non-spatial attributes is varied from 2 to 5. All non-spatial attributes are of integer type in the range $[1, 1000]$, and they conform to either independent or anti-correlated distributions. The total number of mobile devices indicates that we test our methods in environments of small-scale and moderate-scale MANETs, according to a recent classification [60]. All devices move within the spatial domain according to the random waypoint mobility model [20]. In that model, every device moves towards its own destination with its own speed, and when it reaches that destination it will stop there for a period of time (holding time) and then move to another destination with a new random speed. The mobility and wireless settings used are listed in Table 5.7. Every mobile device issues 1 to 5 queries at random times during the simulation. Queries of different devices can coexist, while a single device does not issue a new query if it has one in progress.

For query forwarding, we compared two different strategies. The first is a *breadth-first* strategy, where initially the query originator sends its query to all its neighbors. Each neighbor processes the query locally, sends the result back to

Parameter	Setting
Total simulation time	2h
Maximum speed	10m/s
Minimum speed	2m/s
Holding time	120s
Wireless routing protocol	AODV

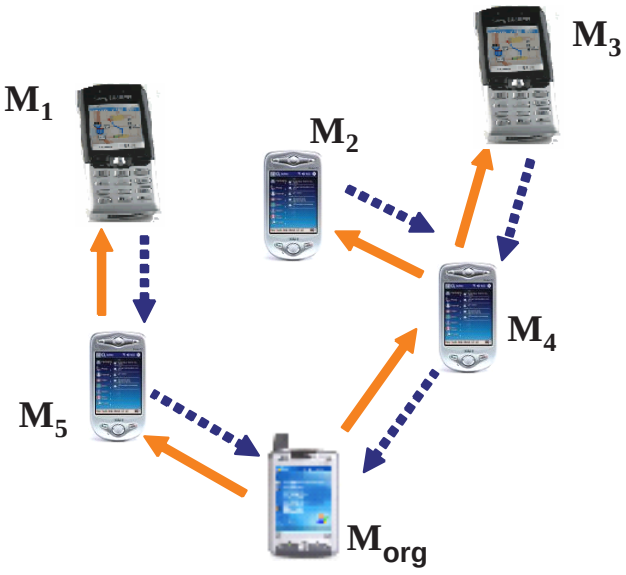
Table 5.7: Parameters used in MANET simulations

the query originator directly and then forwards the query to its own neighbors. The same procedure is repeated on every mobile device involved: processing the query locally, sending the local result directly back to the query originator and forwarding the query to its own neighbors. The second is a *depth-first* strategy, where a query is forwarded to only one neighbor to which the query has not been sent. The query result will only be returned when no further neighbor is available or all neighbors have processed it. Then the result will be forwarded back along the reversed path. Each mobile device M_i (including the originator) on the path merges the received result with its own result. The merge on any intermediate device M_i is similar to the assembly on the query originator, as introduced in Section 5.5.3. Besides, it updates the filtering tuple if necessary. Then, M_i either sends the result further back, or forwards the query (with either the old or the updated filtering tuple) to another available neighbor. These two strategies are illustrated in Figure 5.5.

5.6.2 Data Reduction Efficiency

In this set of experiments, we investigate the efficiency of distributed processing strategies in terms of their data reduction rate. The data reduction rate (DRR) is the proportion of tuples reduced by the filtering tuple to the number of in the unreduced skyline. To be specific, recall that in Section 5.4, for each mobile device M_i (except the query originator M_{org}), its unreduced skyline was given as SK_i and its reduced skyline was given as SK_i'. The data reduction rate with respect to the

(a) Breadth-first strategy



(b) Depth-first strategy

Figure 5.5: Query request forwarding strategies

whole system is then defined as:

$$DRR = \frac{\sum_{i=1,i\neq org}^{m}(|SK_i| - |SK_i'| - 1)}{\sum_{i=1,i\neq org}^{m}|SK_i|}. \quad (5.1)$$

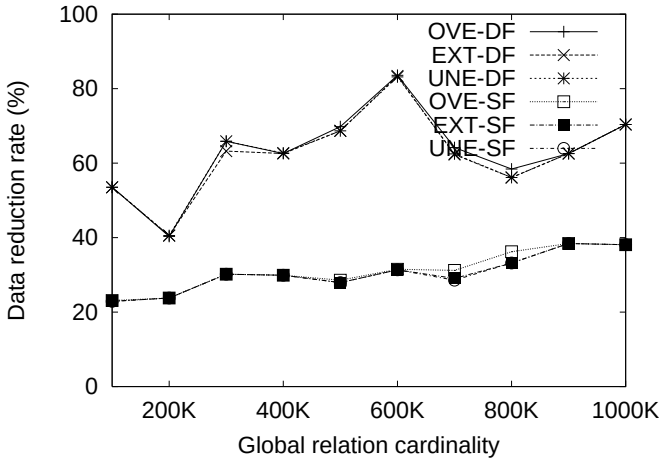
I. Pre-Tests in Static Setting

Before conducting the simulation, we tested the different filtering tuple selections in a static setting where no devices move and queries are forwarded recursively from the originator to the outer neighbors in the grid framework. We also ignore the distance constraint and use every device M_i as the query originator once. The final result is the average of a total of $m \times m$ queries for each single experiment.

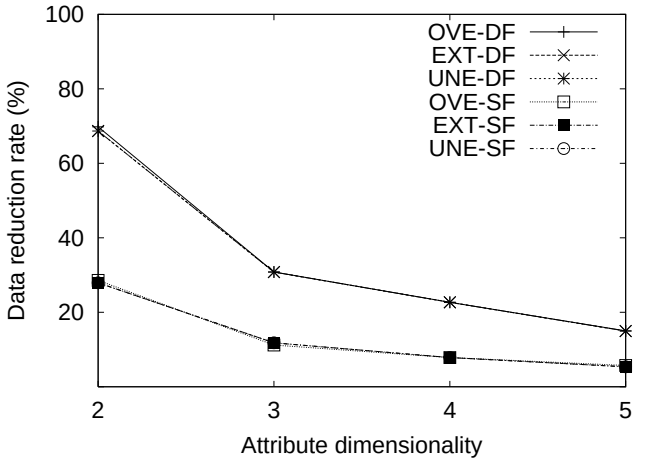
The experimental results on independent global relations are shown in Figure 5.6. The results show that different estimations of the dominating region (OVE for over-estimation, EXT for exact computation, and UNE for under-estimation) barely affect the filtering efficiency for uniform global relations. This justifies the use of estimation in a mobile device that does not require knowledge about the global bounds of each attribute.

In the experiment reported upon in Figure 5.6(a), all global relations have two non-spatial attributes and are partitioned among 5×5 mobile devices. For the strategy using a single filtering tuple (SF), the data reduction rate grows slowly as the global cardinality increases. A data space D of fixed size becomes denser if more tuples fall into D, which renders a given tuple tp_{flt} possibly dominate more tuples. The strategy using a dynamic filtering tuple (DF) has obviously higher efficiency than SF for all relations. This is because the filtering tuple is dynamically changed based on its pruning capacity such that a tuple with higher pruning potential (if there exists one) is picked for further processing. This dynamic adjustment also makes DF lack steadiness along the relation cardinality.

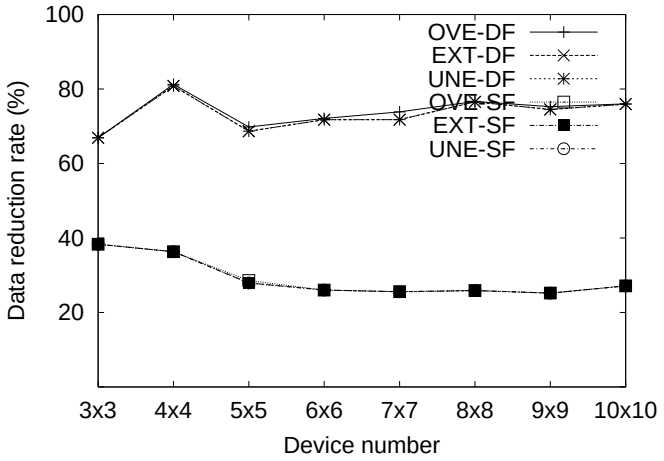
In the experiment covered by Figure 5.6(b), all global relations have 500K tuples



(a) DRR vs. global cardinality

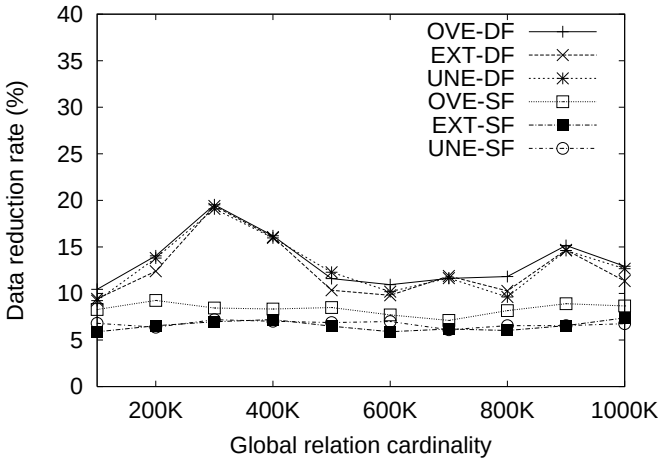


(b) DRR vs. attribute dimensionality

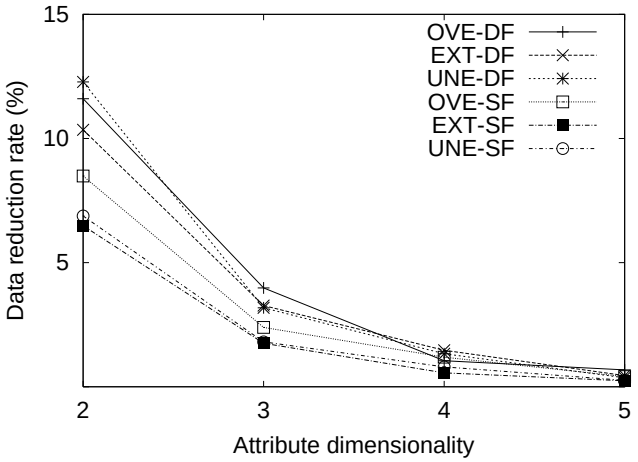


(c) DRR vs. device number

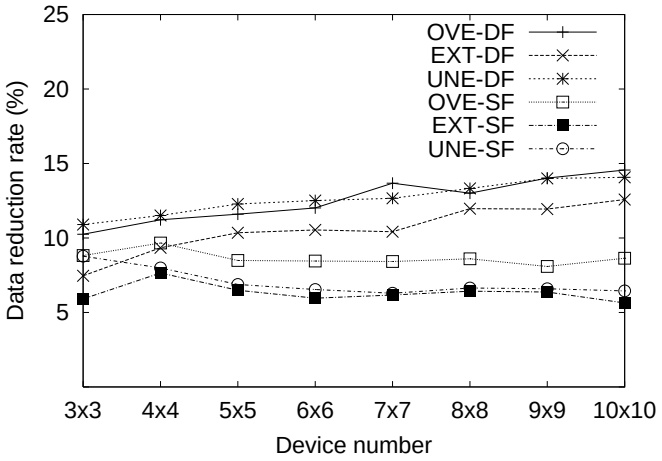
Figure 5.6: DRR on independent datasets in a static setting



(a) DRR vs. global cardinality



(b) DRR vs. non-spatial dimensionality



(c) DRR vs. device number

Figure 5.7: DRR on anti-correlated datasets in a static setting

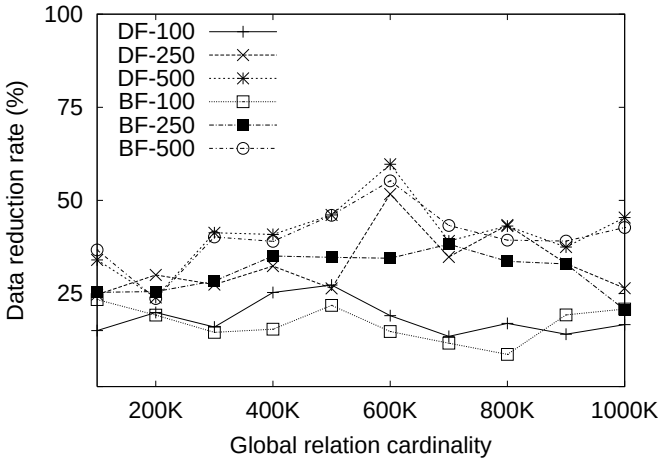
and are partitioned among 5×5 mobile devices. For both filtering strategies, the data reduction rate decreases as the attribute dimensionality increases. In contrast to Figure 5.6(a), a data space D of fixed cardinality becomes sparser if the dimensionality increases, which leaves a given tuple tp_{flt} possibly dominating fewer tuples.

In the experiment reported upon in Figure 5.6(c), all global relations have 500K tuples with two non-spatial attributes. For the SF strategy, the data reduction rate decreases slightly as the number of mobile devices increases. As the global relation is partitioned among more mobile devices, the denominator $\sum_{i=1,i\neq org}^{m} |SK_i|$ in Formula 5.1 possibly becomes larger while the single filtering tuple strategy cannot prune more tuples, which leads to a smaller DRR value. The pruning capacity of the dynamic strategy is not affected, as the filtering tuple is dynamically changed according to the local skyline on every mobile device.

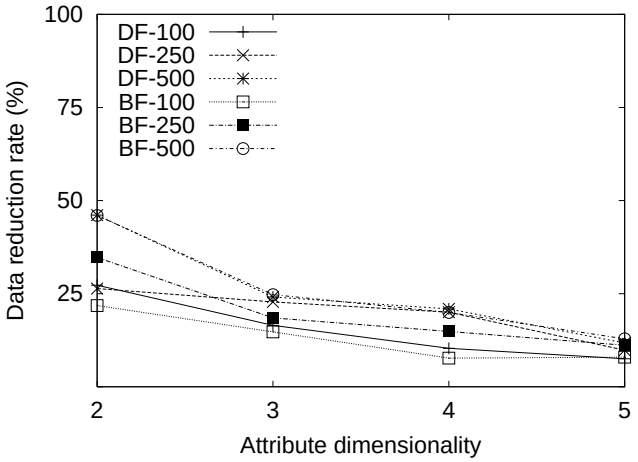
The experimental results on anti-correlated global relations are shown in Figure 5.7. We can see that for the SF strategy, over-estimation of the dominating region exhibits the best filtering efficiency for anti-correlated global relations in almost all cases. However, for every single experiment, the filtering efficiency here is lower compared to that of its counterpart for independent global relations. This is not surprising because the filtering tuples are chosen based on the assumption of an independent distribution.

II. Tests in MANET Simulation

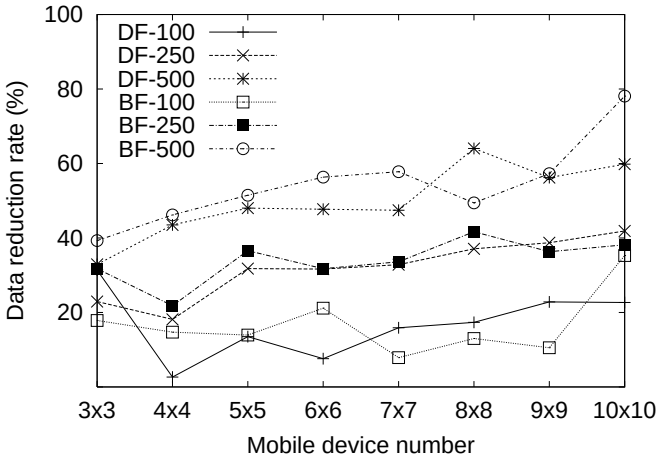
Through the pre-tests in the static setting, we found that the use of estimated versus exact selections led only to very slight differences, especially for uniform datasets. It is also shown that dynamic filtering of tuples yields better DRRs. Thus, we decided to use only under-estimation of dominating regions when selecting filtering tuples in the simulation, and dynamically update them between mobile



(a) DRR vs. global cardinality

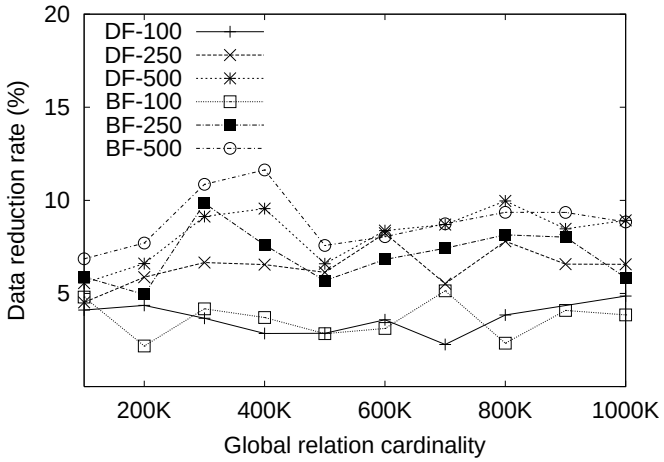


(b) DRR vs. attribute dimensionality

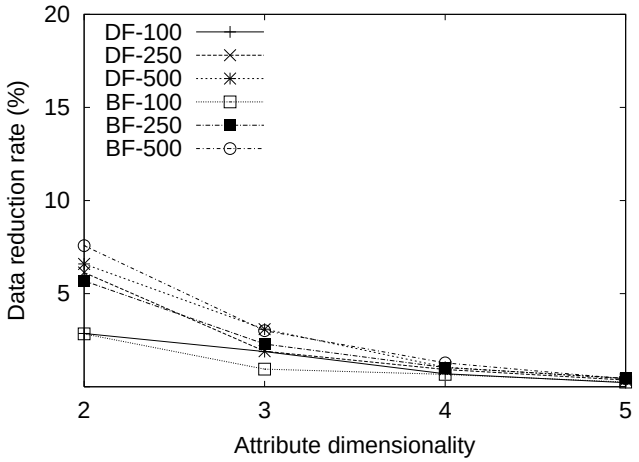


(c) DRR vs. mobile network scale

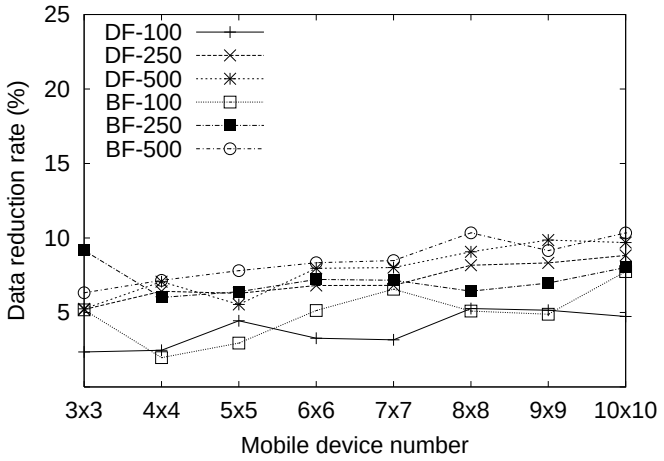
Figure 5.8: DRR on independent datasets in MANET simulation



(a) DRR vs. global cardinality



(b) DRR vs. non-spatial dimensionality



(c) DRR vs. mobile network scale

Figure 5.9: DRR on anti-correlated datasets in MANET simulation

devices, if possible. The same series of datasets are used in the simulation as in the pre-tests. The *DRR* results are shown in Figures 5.8 and 5.9, where DF (BF) is for the depth-first (breadth-first) query forwarding strategy and the integers are distances of interest in queries.

For both distributions, *DRR*s are lower compared to those in the static setting. The is attributed to the MANET setting, where not all devices always participate in the query processing, thus decreasing the data reduction. The mobile characteristic also makes *DRR* changes more untidy as the global cardinality increases. This is because it is not fixed which part(s) of the global relation that do not participate in the query processing. The *DRR* change in terms of attribute dimensionality is still pronounced, which indicates that dimensionality still plays an important role to the query processing performance in MANETs.

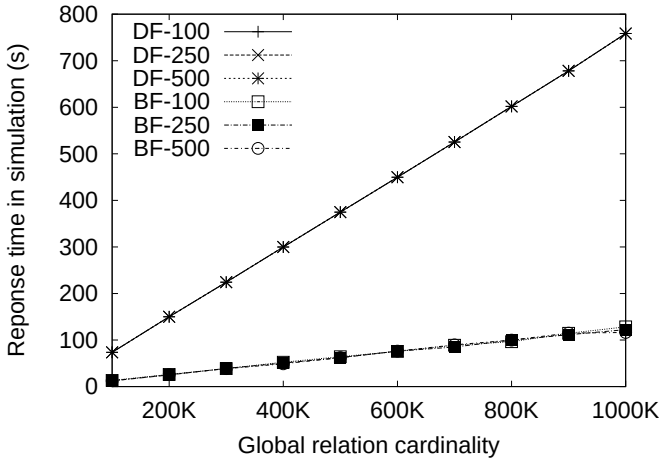
5.6.3 Response Time

For the BF strategy, the response time is defined as the elapsed time from the moment that a query is issued at a mobile device M_{org} to the moment that 80% of all other devices in the network have sent back results, since it is not ensured that all devices are always reachable and available in MANETs. In our initial experiments, it was found that a query forwarded in BF manner more often than not fell into an infinite wait for replies from all mobile peers in the simulated MANET. This is due to the instable wireless connections in the MANET, which totally isolate some peers such that they are not accessible from any other peers including the query originator. This happens in a real MANET environment. To overcome this infinite wait problem, we stipulate that a query forwarded in BF ends when it has got replies from 80% of all peers. The percentage was a careful decision based on empirical tests. For the DF strategy, the response time is defined a little different in

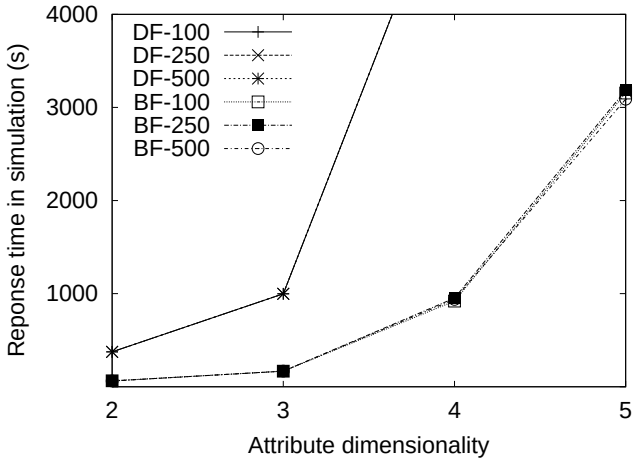
that a query ends when the query originator M_{org} receives the result and finds that all its neighbors have processed the query. As a matter of fact, DF strategy also faces the similar difficulty caused by instable network connections, as those peers inaccessible in BF are also unreachable in DF. Nevertheless, DF strategy does not fall into an infinite wait as it can backtrack and finally end a query as soon as it finds no neighbor to contact.

The simulation results are shown in Figures 5.10 and 5.11, covering independent and anti-correlated datasets, respectively. The response time consists of both the wireless communication overhead and the estimated local processing time on involved devices. In the experiments in Chapter 4, we learn that the local skyline processing cost (algorithm hsSkyline on the hybrid storage) increases almost linearly with respect to the local relation cardinality of each given dimensionality. We add the local filtering mechanism to algorithm hySkyline as presented in Figure 5.4, and run it on the pocket PC used in Chapter 4. We use two batches of 50K local relations: one with 2 to 5 independent attributes and the other with 2 to 5 anti-correlated attributes. We run 5 queries on each relation, each query issued with a query position randomly determined within the spatial extent, a distance of interest randomly picked from {100, 250, 500}, and a filtering tuple randomly chosen from the global relation. Only minor extra costs are observed in the consequent results, compared to those counterparts gained in Chapter 4. Based on these findings, we use the results gained in this way as a baseline to linearly estimate the local skyline query processing times on devices in the simulation. Note that the accuracy of the estimation of the local cost is not a significant concern, as in our simulation the response time of a query is dominated by the communication overhead via the MANET.

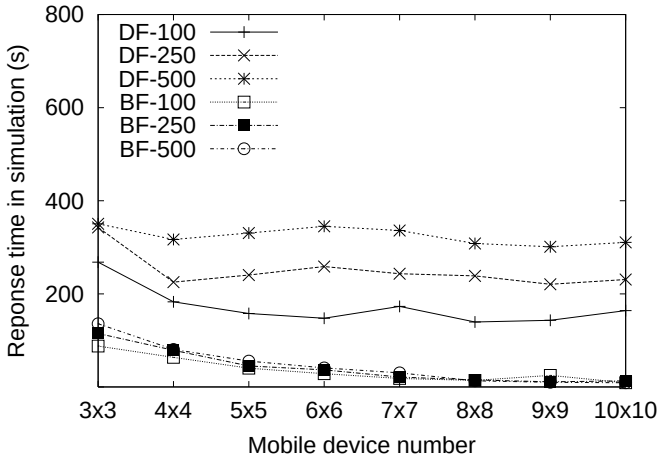
From the figures, we see that BF exhibits shorter response time than does DF.



(a) RT vs. global cardinality

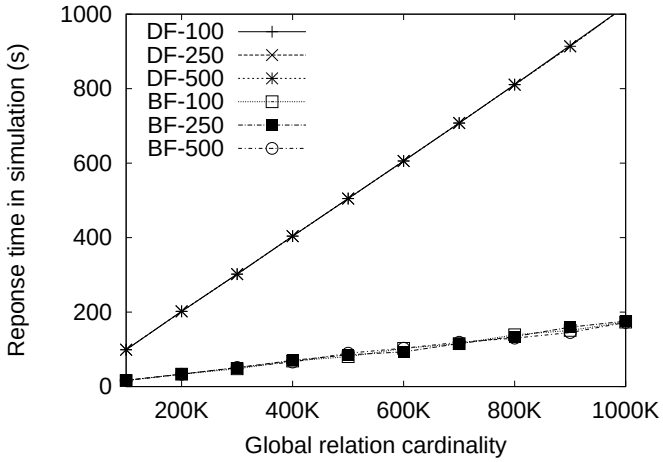


(b) RT vs. attribute dimensionality

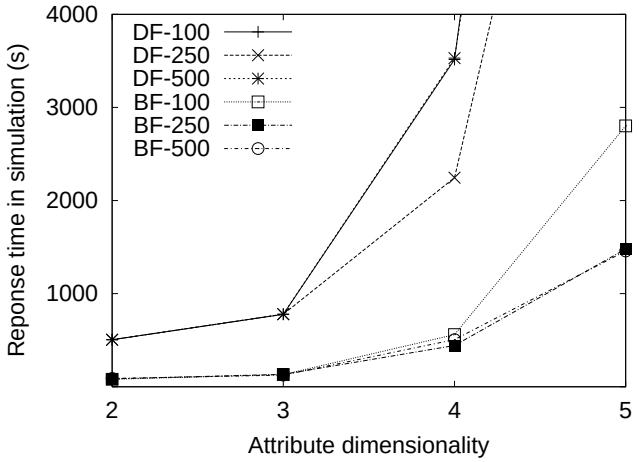


(c) RT vs. mobile scale

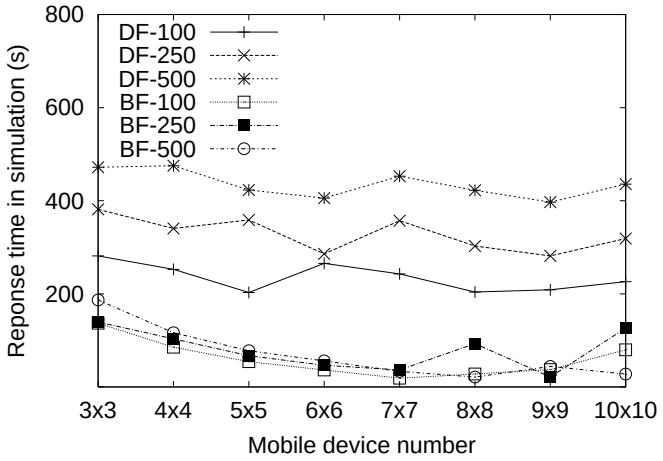
Figure 5.10: Response time on independent datasets in MANET simulation



(a) RT vs. global cardinality



(b) RT vs. non-spatial dimensionality



(c) RT vs. mobile network scale

Figure 5.11: Response time on anti-correlated datasets in MANET simulation

The most important reason for the difference is that the BF query forwarding strategy enables parallel query processing among the mobile devices, while the DF strategy only allows each query to be processed serially along all devices involved. Another reason that leads to a marginal difference is that we only count 80% results back in computing the response time for BF, whereas DF needs to wait longer before a query stops.

DF deteriorates much more quickly than does BF when the dimensionality increases, as shown in Figures 5.10(b) and 5.11(b). Local skyline processing over multi-dimensional datasets is time-consuming on resource-constrained mobile devices. BF offsets that effect through parallelism; on the contrary, DF is only hurt by that effect because of serialization.

BF improves as the number of mobile devices increases, as shown in Figures 5.10(c) and 5.11(c). This is because more devices increases the degree of parallelism of BF. The distance constraints make a more obvious difference to DF than to BF. This is also attributed to DF's serialization, which is more sensitive than parallelism to distance constraints, as larger search ranges usually involves more devices and data.

5.6.4 Query Message Count

In the simulation we found that the cardinality, the dimensionality, and the distribution have little impact on the message count. Therefore, we only show in Figure 5.12 how the message count varies as the number of mobile devices increases. Although BF shows better performance than does DF in terms of response time, this gain is not for free. Parallelism generates and forwards more messages in the wireless network, which in turn consumes more wireless communication bandwidth. As a result of this effect, the improvement of response time slows down in

our simulation (shown in Figures 5.6.3 and 5.6.3) as the number of mobile devices increases.

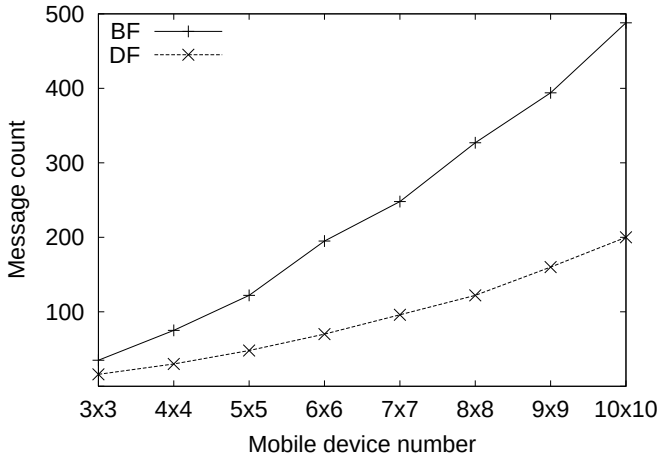
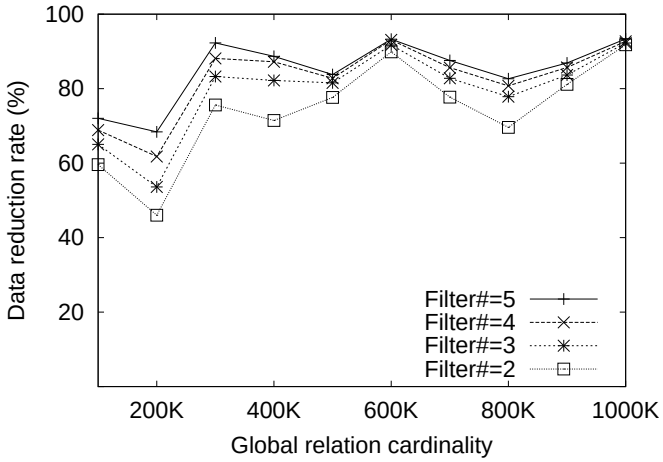


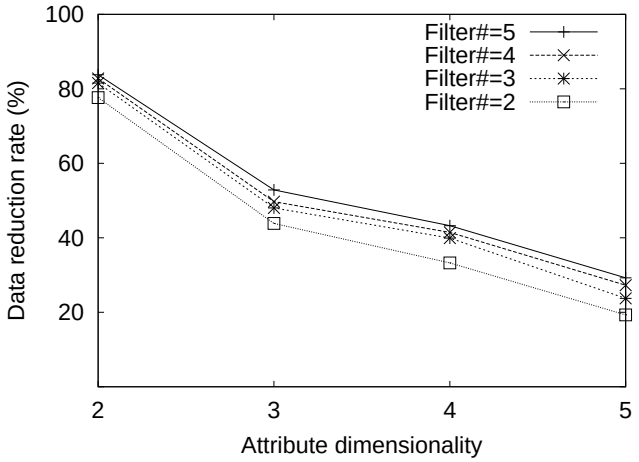
Figure 5.12: Query message count

5.6.5 Data Reduction Efficiency with Multiple Filtering Tuples

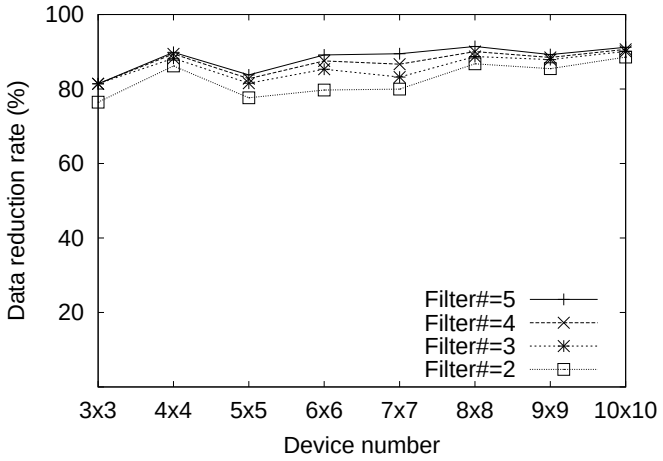
So far in the experiments we have used only one filtering tuple, though we allow the single filtering tuple to be changed dynamically. It is possible to generalize the filtering idea and use more than only one filtering tuple. This poses questions of how many (and which) points should be used as filters to ensure sufficient data reduction rate. To obtain an intuition of the effect of multiple filtering tuples, we conduct experiments with multiple filtering tuples in a static setting used in Section 5.6.2. In the experiments, we vary the number of filtering tuples from 2 to 5. In each single experiment, the number of filtering tuples is fixed. Given the set of local skyline points, k ($2 \leq k \leq 5$) points with the largest exact VDR values are selected as multiple filtering tuples. During the query forwarding, they are dynamically changed to ensure that those skyline points with the highest expected



(a) DRR_m vs. global cardinality

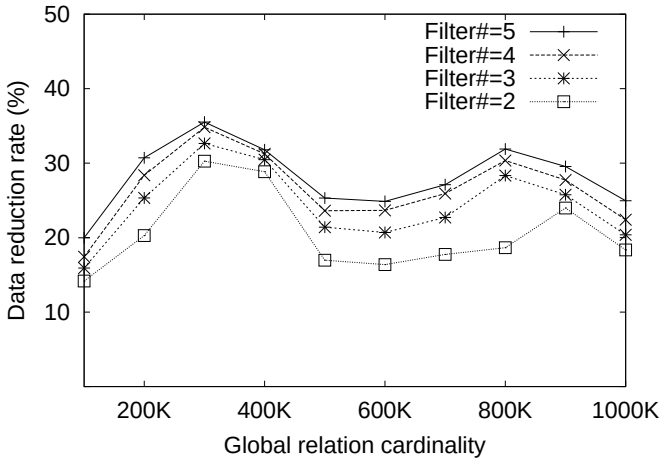


(b) DRR_m vs. attribute dimensionality

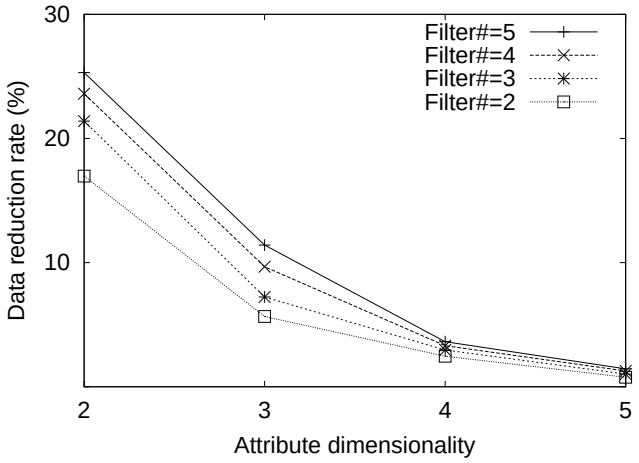


(c) DRR_m vs. device number

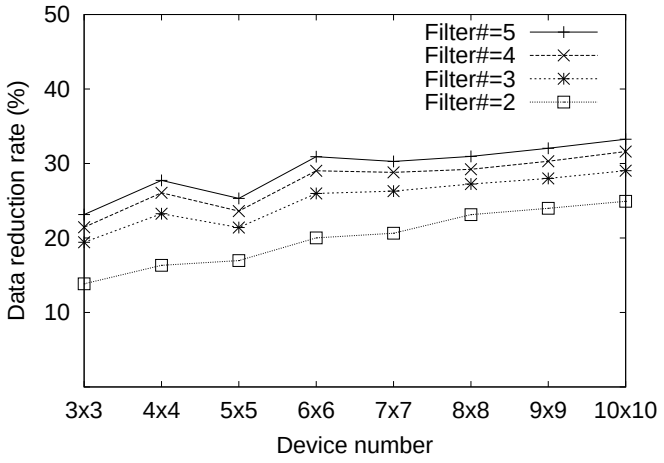
Figure 5.13: DRR_m on independent datasets



(a) DRR_m vs. global cardinality



(b) DRR_m vs. non-spatial dimensionality



(c) DRR_m vs. device number

Figure 5.14: DRR_m on anti-correlated datasets

filtering capacity are always selected.

The results of data reduction rates with multiple filtering tuples (DRR_m) are shown in Figure 5.13 and Figure 5.14, respectively on independent and anti-correlated datasets. Referring to those results, we can see that more filtering tuples produce higher data reduction rates. This is not surprising, because more filtering tuples are likely to identify more unqualified skyline points on each intermediate device. Furthermore, this improvement attributed to multiple filtering tuples is more apparent on anti-correlated datasets. Our estimation of *VDR* values is based on the assumption of an independent data distribution, which causes a gap between the estimation and the real datasets. Nevertheless, when multiple filtering tuples are used, this gap is reduced by the collective filtering effect, and therefore the data reduction rate is increased markedly. Besides, the filtering effect still suffers from the data dimensionality increase. The same reason behind Figures 5.6(b) also applies here.

By comparing those results to their counterparts with single filtering tuple, shown in Figures 5.6 and 5.7, we see that the performance gain from 2 filtering tuples to 5 ones is slighter than that from a single filtering tuple to 2 ones. This indicates that using only 2 filtering tuples is favorable, as it improves the data reduction rate remarkably but incurs slight additional computation cost in selecting the top-2 filtering tuples. This also implies that it does not help much if we change the number of filtering tuples (from 2 to more) on the fly in the query forwarding.

We here employ a simple greedy strategy to select k ($2 \leq k \leq 5$) filtering tuples, which have the largest *VDR* values. It is worth noting that the problem of finding from a set of s skyline points the optimal subset(s) of k ($1 < k < s$) filtering points, which requires the largest collective filtering capability by those k points, is similar to a recent work on selecting k most representative skyline points [55] and is also

of NP-hard computation complexity. According to the experimental results, our simple greedy strategy is efficient in terms of data reduction.

5.7 Summary

In this chapter, we have addressed distributed skyline querying in a wireless mobile ad-hoc network (MANET). The queries have spatial constraints and involve multiple mobile devices in a MANET. Each mobile device contains some portion of the data of the geographical space within which they move. Any mobile device may issue skyline queries against other peers through the MANET.

To reduce the wireless communication cost, we have proposed a distributed query processing strategy that takes advantage of the skyline dominance relationship to filter out non-qualifying intermediate tuples. When the local skyline is computed on the query originator device, the skyline point with the maximum capability to dominate other points is selected based on a probability model we propose. That filtering point is then sent out together with the query request to other peers. On any other device, the filtering point is used to remove unqualified answers in the local skyline otherwise to be sent back to the query originator, and thus reducing the amount of data transmitted. During the query forwarding via multiple hops in the MANET, the filtering point is dynamically changed on involved devices so that its dominating capability is maximized. On each mobile device involved, the local query processing is made faster by specific measures. Extensive experimental studies demonstrate the efficiency of the strategy proposed, in terms of both communication cost savings and response time.

CHAPTER 6

Conclusions and Future Work

In the database community, skyline queries have gained considerable attention in the past few years because of their suitability in retrieving data according to multiple criteria. However, little work on skyline queries has been done in dynamic computing environments including moving objects databases and mobile ad-hoc networks (MANETs). Due to the constant advances in electronics miniaturization, positioning systems and wireless communication, such dynamic environments are becoming increasingly popular. This important trend, plus the significance of skyline queries, renders it meaningful or even pressing to address skyline queries in dynamic computing environments.

This thesis work is motivated by above observations. In this thesis, along three different but correlated aspects, we explore skyline queries in dynamic computing environments. First, by assuming a client/server architecture, we consider continuous skyline queries for moving objects stored on a powerful server. The continuously changing distances between points of interest and the moving query point pose the

most significant challenge in this problem. Second, we address skyline queries on mobile lightweight devices, which are the most usual computing platforms in a mobile environment. Resource limitations in both storage space and computing capability necessitate careful adjustments and configurations, in order to speed up skyline query processing on such devices. Third, we go further and set skyline queries in a MANET, which is formed by multiple mobile lightweight devices via wireless peer-to-peer communication. Our goal in this problem is to find an efficient distributed skyline query processing strategy in a MANET.

6.1 Conclusions

6.1.1 Continuous Skyline Queries for Moving Objects

The literature on skyline queries has so far dealt mainly with static query points and static points of interest. This static setting is being shaken by the growing popularity of moving objects. Motivated by this, we consider continuous skyline queries for moving objects. In our problem, a continuous query is issued by a continuously moving query point, and the data points of interest can be either static or moving too. The changing distance between the query point to any data point is considered as a unique dimension in the skyline computation.

Due to the movement of query point (and points of interest if applicable), the skyline result taking the changing distance into account changes continuously. Re-computing the skyline result from scratch every time it changes is not attractive, as this method does not capture the problem characteristics to ensure solution efficiency and feasibility.

Based on a thorough analysis that exploits the spatiotemporal coherence of the problem, we propose our solution that maintains continuous skyline result in

an incremental way. An invariant part of the skyline result is identified and utilized to derive a search bound for further processing and maintenance. Then the preconditions for potential skyline changes are strictly discovered, based on the popular linear movement model. After the detailed analysis, we propose a kinetic-based data structure [12] and relevant processing algorithms for continuous skyline queries.

Our solution for continuous skyline queries is applicable to both static and moving points of interest. For the latter case, we also address how to accommodate moving plan updates of points of interest. We also present the space and time costs of the proposed solution. Results of an extensive experiment demonstrate that the proposed solution is robust and efficient.

6.1.2 Skyline Queries on Mobile Lightweight Devices

In a mobile computing environment, devices are frequently lightweight as they have limited resources including storage space and computing capabilities. Though processing skyline queries locally on such devices is attractive when users are moving around, this remains a challenge because of the resource constraints. Squeezing existing skyline algorithms for normal computers into such devices is unlikely to achieve good performance.

On the contrary, we propose for such devices proper measures that contribute to speed up the on-device skyline query processing. Base on an analysis on existing data storage schemes for resource-limited devices, a hybrid storage scheme is proposed to store data points of interest on devices in a space-efficient way. The hybrid storage scheme deals with spatial coordinates and non-spatial attributes differently. Different point location coordinates are directly stored in their float values, whereas non-spatial attributes sharing float type duplicates are stored in

a scheme modified from the ID based storage [76]. With attribute value domains correctly sorted, subsequent skyline processing only needs to access the integer IDs. One attribute is selected and also sorted to further reduce value comparison in skyline computation. Based on the hybrid storage scheme, we propose an adaption of the existing skyline algorithm SFS [29] for the resource-limited devices. Compared to straightforward using of existing algorithms, our method is more efficient as it carries out less value comparisons, and it compares integer IDs instead of raw float values.

We conduct experiments on a real HP pocket PC device. The experimental results show that our hybrid storage proposal not only saves on-device storage cost but also speeds up on-device skyline query processing.

6.1.3 Skyline Queries Against Mobile Lightweight Devices in MANETs

The previous problem addresses skyline queries on individual mobile lightweight devices. Due to the wireless peer-to-peer networking technologies nowadays, such devices are able to constitute self-organizing, wireless mobile ad hoc networks (MANETs) that allow seamless, low-cost, and easily deployed communications [11, 20]. Within a MANET, such mobile devices can exchange data and even collaborate query processing. This makes it possible for one device to issue distributed skyline queries, and each query retrieves data from multiple mobile peers instead of accessing local data only.

To efficiently process such distributed skyline queries in a MANET, we need to cut both the local skyline query processing time and the data transmission time among mobile devices. For the former, the findings on the previous problem are utilized to guide on-device configurations. For the latter, we propose a filtering

based query processing strategy that is able to identify some unqualified data points and prevent them from being transmitted among mobile devices.

When the local skyline is computed on the device originating the query, the skyline point with the maximum capability to dominate other points is selected based on a probability model we propose. That point, termed *filtering point*, is then attached to the query request sent out to other peers. On any other device, the filtering point is used to filter out unqualified answers in the local skyline otherwise to be sent back to the query originator. During the query forwarding via multiple hops in the MANET, the filtering tuple is dynamically changed on involved devices so as to maximize the dominating capability it holds.

We conduct extensive simulation experiments using a MANET simulator JiST-SWANS [1]. The results show that our proposal reduces the amount of data to transmit, especially for independent datasets. The data reduction consequently shortens the response time of distributed skyline queries in the simulation.

6.1.4 Discussion

Throughout this thesis, we have focused on full space skyline computation in dynamic environments. Nevertheless, our solutions also support user queries that are concerned with subspace skylines, as long as proper adaptations are taken.

For the first problem, subspace skyline can be supported in this way. For each subspace including both dynamic distance and static attributes, its corresponding static partial subspace skyline could be pre-computed and used as the SK_{ns} we have in the full skyline case. Whereas, our spatiotemporal analysis and algorithms still work as the changing distance is the same for both full space skyline and any subspace skyline. To support arbitrary subspace skyline, extra computation time is needed to build up the whole skyline cube [98] for all static attributes before the

system is ready for continuous skyline queries. To store the whole skyline cube, the storage requirement will be considerably high. Efficient data structures are therefore expected. An existing proposal [93] may fit to this requirement.

For the second and third problems, the hybrid storage of the local relations may become less helpful as the topological order of all tuples is unlikely to hold on an arbitrary subspace. To compute a subspace skyline, we could resort to BNL algorithm. The benefit still available is that comparisons are only to be conducted between integer IDs instead of raw values.

For the third problem, the filtering strategy will still work as well. The selection of filtering tuple can be adapted easily. A full space dominating region will be projected onto the corresponding subspace of which the skyline is wanted, and VDR values will be calculated in the subspace correspondingly.

6.2 Directions for Future Work

This thesis studies skyline queries in dynamic environments. There are several directions for future work that extends the research presented in this thesis.

- First, the Euclidean distance we have used in all three problems can be replaced by network distance, as moving objects may be constrained in spatial networks such as a road transportation network. In the first problem, the Euclidean distance is used as a unique dimension in the skyline computation. When the network distance is used instead, the preconditions of skyline changes must be reexamined, and result maintenance must be modified accordingly. The key difference is the time parameterized distance functions between query point and points of interest, which are reduced to be segmentally linear [28] in a road network. In the rest two problems, the spatial

constraint for a skyline query will be specified in network distance, which accordingly requires necessary and appropriate modifications in the relevant solutions.

- Second, we have assumed the linear movement model for moving objects in Chapter 3. Though this currently is the most popular model used in research on moving objects, it is also of interest to address continuous skyline queries for moving objects that are abstracted in other models including uncertainty model [83]. Distance computation is still the crucial part when other models are assumed. Accuracy might be traded for query processing efficiency, if the model to use does not support accurate and fast distance computation.

- Third, we use a simple greedy strategy to choose from s local skyline points k $(1 < k < s)$ filtering points in Chapter 5. To incorporate such an NP-hard selection of multiple filtering tuples into lightweight mobile devices, specific heuristics are needed to improve the local computation efficiency. Otherwise, considerable extra local selection costs can harm the overall performance of the distributed filtering based skyline query processing, if the selected multiple filtering points do not additionally identify and reduce enough unqualified skyline points on intermediate devices. The similar unexpected outcome can also happen in a wired distributed environment. As the subsequent work beyond this thesis, we currently are investigating into those issues.

BIBLIOGRAPHY

[1] JiST/SWANS. http://jist.ece.cornell.edu.

[2] Merriam-webster online dictionary. http://www.m-w.com.

[3] Superwaba. http://www.superwaba.com.

[4] International standardization organizatin (ISO). *Integrated Circuit(s) Cards with Contacts – Part 7: Interindustry Commands for Structured Card Query Language (SCQL)*, ISO/IEC 7816-7, 1999.

[5] Fusing ad hoc and P2P. *Pictures of the Future (Siemens Magazine for Research and Innovation)*, Spring:38–19, 2005.

[6] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '00)*, pages 175–186, 2000.

[7] A. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a memory resident dbms. In *Proceedings of the 30th IEEE Computer Society International Conference (COMPCON '85)*, pages 54–57, 1985.

[8] G. Antoshenkov, D. B. Lomet, and J. Murray. Order preserving compression. In S. Y. W. Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE '96)*, pages 655–663. IEEE Computer Society, 1996.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 1–16, 2002.

[10] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT '04)*, pages 256–273, 2004.

[11] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, editors. *Mobile Ad Hoc Networking*. Wiley-IEEE Press, New Jersey, 2004.

[12] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 747–756, 1997.

[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*, pages 322–331, 1990.

[14] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proceedings of the 6th International Database Engineering and Applications Symposium (IDEAS '02)*, pages 44–53, 2002.

[15] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 179–187, 1990.

[16] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM (JACM)*, 25(4):536–543, 1978.

[17] G. Bernard, J. Ben-Othman, L. Bouganim, G. Canals, S. Chabridon, B. Defude, J. Ferrié, S. Gançarski, R. Guerraoui, P. Molli, P. Pucheral, C. Roncancio, P. Serrano-Alvarado, and P. Valduriez. Mobile databases: a selection of open issues and research directions. *SIGMOD Record*, 33(2):78–83, 2004.

[18] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. Picodmbs: Scaling down database techniques for the smartcard. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 11–20, 2000.

[19] S. Börzönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*, pages 421–430, 2001.

[20] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the 4th annual international conference on Mobile computing and networking (MOBICOM '98)*, pages 85–97, 1998.

[21] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters (IPL)*, 33(2):63–65, 1989.

[22] Budiarto, S. Nishio, and M. Tsukamoto. Data management issues in mobile and peer-to-peer environments. *Data & Knowledge Engineering (DKE)*, 41(2-3):183–204, 2002.

[23] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 203–214, 2005.

[24] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, pages 503–514, 2006.

[25] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT '06)*, pages 478–495, 2006.

[26] P. Charas. Peer-to-peer mobile network architecture. In *Proceedings of the 1st International Conference on Peer-to-Peer Computing (P2P '01)*, pages 55–61, 2001.

[27] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 64, 2006.

[28] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 865–876, 2005.

[29] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of the 19th International Conference on Data Engineering (ICDE '03)*, pages 717–719, 2003.

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

[31] L. D. Fife and L. Gruenwald. Research issues for data communication in mobile ad-hoc network database systems. *SIGMOD Record*, 32(2):42–47, 2003.

[32] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT '04)*, pages 67–87, 2004.

[33] P. Godfrey. Skyline cardinality for relational processing. In *Proceedings of the 3rd International Symposium on Foundations of Information and Knowledge Systems (FoIKS '04)*, pages 78–97, 2004.

[34] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 229–240, 2005.

[35] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE '98)*, pages 370–379, 1998.

[36] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pages 47–57, 1984.

[37] D. Hearn and M. P. Baker. *Computer Graphics C Version.* Prentice-Hall International, Inc., New Jersey, 1997.

[38] G. Hjaltason and H. Samet. Distance browsing in spatial database. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[39] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 479–490, 2005.

[40] X. Huang and C. S. Jensen. In-route skyline querying for location-based services. In *Proceedings of the 5th International Workshop on Web and Wireless Geographical Information Systems (W2GIS '05)*, pages 120–135, 2004.

[41] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in MANETs. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 66, 2006.

[42] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(12):1645–1658, 2006.

[43] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB '03)*, pages 512–523, 2003.

[44] C. S. Jensen. Geo-enabled, mobile services-a tale of routes, detours, and dead ends. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA '06)*, pages 6–19, 2006.

[45] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 768–779, 2004.

[46] W. Jin, J. Han, and M. Ester. Mining thick skylines over large databases. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD '04)*, pages 255–266, 2004.

[47] J. Jing, A. S. Helal, and A. Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2):117–157, 1999.

[48] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '99)*, pages 261–272, 1999.

[49] V. Koltun and C. H. Papadimitriou. Approximately dominating representatives. In *Proceedings of the 10th International Conference on Database Theory (ICDT '05)*, pages 204–214, 2005.

[50] G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. Segall. When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad hoc networks. In *Proceedings of the 1st International Conference on Peer-to-Peer Computing (P2P '01)*, pages 75–, 2001.

[51] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 275–286, 2002.

[52] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[53] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. DADA: A data cube for dominant relationship analysis. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, pages 659–670, 2006.

[54] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 502–513, 2005.

[55] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: the k most representative skyline operator. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE '07)*, pages 86–95, 2007.

[56] C. Lindemann and O. P. Waldhorst. A distributed search service for peer-to-peer file sharing in mobile applications. In *Proceedings of the 2nd International Conference on Peer-to-Peer Computing (P2P '02)*, pages 73–80, 2002.

[57] E. Lo, K. Y. Yip, K.-I. Lin, and D. W. Cheung. Progressive skylining over web-accessible databases. *Data & Knowledge Engineering (DKE)*, 57(2):122–147, 2006.

[58] H.-X. Lu, Y. Luo, and X. Lin. An optimal divide-conquer algorithm for 2d skyline queries. In *Proceedings of the 7th East European Conference on Advances in Databases and Information Systems (ADBIS '03)*, pages 46–60, 2003.

[59] H. Luo, R. Ramjee, P. Sinha, L. E. Li, and S. Lu. UCAN: A unified cellular and ad-hoc network architecture. In *Proceedings of the 9th annual international conference on Mobile computing and networking (MOBICOM '03)*, pages 353–367, 2003.

[60] J. Macker and M. Corson. *Mobile Ad Hoc Networking*, chapter Mobile Ad Hoc Networks (MANETs): Routing Technology for Dynamic, Wireless Networking. Wiley-IEEE Press, New Jersey, 2004.

[61] D. H. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4):318–324, 1974.

[62] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 623–634, 2004.

[63] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 188–198, 2002.

[64] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[65] S. Pace, G. P. Frost, I. Lachow, D. Frelinger, D. Fossum, D. Wassem, and M. M. Pinto. *The Global Positioning System: Assessing National Policies*. RAND, Santa Monica, California, 1995.

[66] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 467–478, 2003.

[67] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[68] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 253–264, 2005.

[69] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers (TOC)*, 51(10):1124–1140, 2002.

[70] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[71] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[72] J. F. Roddick, M. J. Egenhofer, E. G. Hoel, D. Papadias, and B. Salzberg. Spatial, temporal and spatio-temporal databases - hot issues and directions for phd research. *SIGMOD Record*, 33(2):126–131, 2004.

[73] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*, pages 71–79, 1995.

[74] F. N. Rüdiger Schollmeier, Ingo Gruber. Protocol for peer-to-peer networking in mobile environments. In *Proceedings of the 12th International Conference*

On Computer Communications and Networks (ICCCN '03), pages 121–127, 2003.

[75] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pages 331–342, 2000.

[76] R. Sen and K. Ramamritham. Efficient data management on lightweight computing device. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, 2005.

[77] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 751–762, 2006.

[78] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13rd International Conference on Data Engineering (ICDE '97)*, pages 422–432, 1997.

[79] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proceedings of the 2nd International Conference on Mobile Data Management (MDM '01)*, pages 161–172, 2001.

[80] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD '01)*, pages 79–96, 2001.

[81] R. E. Steuer. *Multiple criteria optimization.* Wiley, New York, 1986.

[82] K. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 301–310, 2001.

[83] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 611–622, 2004.

[84] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*, pages 334–345, 2002.

[85] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(3):377–391, 2006.

[86] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 287–298, 2002.

[87] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 65, 2006.

[88] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[89] Wikipedia. Galileo positioning system. http://en.wikipedia.org/wiki/galileo_positioning_system., 2006.

[90] O. Wolfson and B. Xu. Opportunistic dissemination of spatio-temporal resource information in mobile peer to peer networks. In *Proceedings of the 1st International Workshop on Peer2Peer Data Management, Security and Trust (PDMST '04)*, pages 954–958, 2004.

[91] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM '98)*, pages 111–122, 1998.

[92] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT '06)*, pages 112–130, 2006.

[93] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, pages 491–502, 2006.

[94] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 643–654, 2005.

[95] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM '04)*, pages 317–326, 2004.

[96] B. Xu and O. Wolfson. Data management in mobile peer-to-peer networks. In *Proceedings of the 2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P '04)*, pages 1–15, 2004.

[97] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, 1984.

[98] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 241–252, 2005.

[99] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 443–454, 2003.

[100] Z. Zhang, X. Guo, H. Lu, A. K. H. Tung, and N. Wang. Discovering strong skyline points in high dimensional spaces. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM '05)*, pages 247–248, 2005.

[101] G. K. Zipf. *Human behaviour and the principle of least effort: An Introduction to Human Ecology.* Addison-Wesley, Reading, MA, 1949.