Scaling Continuous Query Services for Future Computing Platforms and Applications

A Thesis
Presented to
The Academic Faculty

by

Buğra Gedik

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August, 2006

Scaling Continuous Query Services for Future Computing Platforms and Applications

Approved by:

Date Approved: 25 May 2006

To my family,

Yusuf, Meral, and Tuğba Gedik.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the generous help and encouragement of many individuals. I would like to express my gratitude to everyone who has contributed to the process leading to my dissertation. Here I would like to mention a few of these people.

First and foremost, I would like to thank my advisor Prof. Ling Liu for her tremendous help and guidance in every phase and aspect of my Ph.D. study. The flexibility and freedom she has provided me in deciding and evolving my research focus has been an invaluable asset in my quest to acquire, generate, and disseminate new knowledge. I'm indebted for the countless hours she has spent on perfecting my work and the immense amount of advice she has given me on research, career, and life related matters. She has set the perfect example of being an inspiring researcher and teacher for me. Her example has been very influential in shaping this work and will undoubtedly continue to light my way in my future career.

I would like to give my special thanks to Prof. Calton Pu for being an inspirational figure for me and for his always apposite and timely advice. I would like to also thank every member of our DiSL research group for providing a friendly and dynamic working environment and for engaging in insightful research discussions with me, that have helped in continuously improving and polishing my work.

I would like to thank my manager Dr. Philip S. Yu and my mentor Dr. Kun-Lung Wu at IBM T.J. Watson Research Labs for letting me work on topics related to my thesis during my three summer internships with them. Their impact on my research perspective has definitely helped me in raising the quality of my work; and making their acquaintance have opened several new doors in my research career.

I would like to thank my committee members Prof. Sol M. Shatz, Prof. Leo Mark, Prof. Kishore Ramachandran, and Prof. Christian S. Jensen for being very supportive of my research and for their constructive critiques that have greatly contributed to my thesis.

I would like to thank my dear friends Selçuk Aktürk, Şeyhmus İnci, Tarık Arıcı, and Gültekin Kuyzu for being the colors of my life in Atlanta, to my long time friend Onur Özyer for always keeping in-touch with me, and to Tansel Özyer for his brotherly advice.

I would like to dedicate this work to my father, mother, and sister for their everlasting love and gratuitous support for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The ever increasing rate of digital information available from on-line sources drives the need for building information monitoring applications to assist users in tracking relevant changes in these sources and accessing information that is of interest to them in a timely manner. This growth in the amount of digital information available is partly due to the emergence of pervasive networks and ubiquitous computing platforms. These advances not only make it easy to access information from anywhere at anytime, but also facilitate the acquisition and generation of new information. For instance, the advances in mobile and sensor systems create a new set of information sources, capturing the changes in the physical environment, as well as the changes in the activity and location context of mobile users.

Continuous queries (CQs) are standing queries that are continuously evaluated over dynamic sources to track information changes that meet user specified thresholds and notify users of new results in near real-time. CQ systems can be considered as powerful middleware for supporting information monitoring applications. A significant challenge in building CQ systems is scalability, caused by the large number of users and queries, and by the large and growing number of information sources with high update rates. Further stressing the importance of the scalability problem in providing CQ services is the unique challenges posed by the future computing platforms, such as P2P, mobile, and sensor-based computing platforms. Highly distributed and decentralized characteristics of these environments, as well as the constrained nature of network, computing, and power resources in these environments, call for new techniques to scale CQ services in supporting information monitoring applications, different than the traditional techniques applied in client/server-based systems where most of the CQ processing is performed at a centralized location.

This thesis uses CQs to shepherd through and address the challenges involved in supporting information monitoring applications in future computing platforms. The focus is

on P2P web monitoring in Internet systems, location monitoring in mobile systems, and environmental monitoring in sensor systems. Although different computing platforms require different software architectures for building scalable CQ services, there is a common design philosophy that this thesis advocates for making CQ services scalable and efficient. This can be summarized as "*move computation close to the places where the data is produced.*" A common challenge in scaling CQ systems is the resource-intensive nature of query evaluation, which involves continuously checking updates in a large number of data sources and evaluating trigger conditions of a large number of queries over these updates, consuming both cpu and network bandwidth resources. If some part of the query evaluation can be pushed close to the sources where the data is produced, the resulting early filtering of updates will save both bandwidth and cpu resources. This is the main intuition behind this common design philosophy.

In summary, this thesis develops system-level architectures and techniques to address scalability problems in distributed information monitoring services through the use of CQ systems and concepts. Briefly, the following contributions are made in the areas of P2P, mobile, and sensor CQ systems:

- We introduce PeerCQ, a scalable peer-to-peer architecture for continuous query-based information monitoring on the Internet. Within the PeerCQ framework, we also introduce a dynamic passive replication scheme to support reliable information monitoring.
- We introduce MobiEyes, a scalable distributed architecture for continuous query-based location monitoring in mobile systems. Furthermore, we introduce the MAI indexing technique for server side query processing support for location monitoring CQs.
- We introduce a selective sampling-based energy-efficient data collection framework for sensor CQ systems. In addition, we introduce resource-aware query evaluation techniques for CQ systems dealing with sensor streams.

This thesis shows that distributed CQ architectures that are designed to take advantage of the opportunities provided by ubiquitous computing platforms and pervasive networks, while at the same time recognizing and resolving the challenges posed by these platforms, lead to building scalable and effective CQ systems to better support the demanding information monitoring applications of the future.

CHAPTER I

INTRODUCTION

The proliferation of ubiquitous computing devices and pervasive networks has fueled the growth in the amount of digital information that is available at anytime and accessible from anywhere. On the bright side, the increasing number of on-line and dynamic information sources enable dissemination of fresh information and rapid response to changes. On the down side, high aggregate update rate of these large and growing number of information sources, and large number of user subscriptions for change detection and notification create challenges for building scalable *information monitoring* applications.

Continuous Queries (CQs) have traditionally been used to express information monitoring interests of users over on-line and dynamic data sources [124, 78]. *CQ Services*, that accept user subscriptions expressed in the form of CQs and continuously execute CQs to provide streams of results to users, are employed to support information monitoring applications [78, 80]. Although scalable CQ middleware has been an important topic of research in the literature [78, 31, 27, 6, 10], new designs and architectures are needed, that can cope with the challenges posed and exploit the opportunities provided by the emerging ubiquitous computing platforms. These platforms often exhibit highly distributed, heterogeneous, and decentralized nature, different than the traditional client/server-based systems assumed by the existing work in this area.

Before discussing the challenges and issues involved in scaling CQ services for future computing platforms and applications, we first provide some background information on CQs and information monitoring services they support, give motivating application examples, and describe the properties of the computing platforms on which the CQ services are expected to run in a pervasive computing world.

1.1 CQs and Information Monitoring

Continual Queries are primitives used to express information monitoring requests. They are standing queries that monitor information updates and return results whenever the updates reach certain user specified thresholds. There are three main components of a CQ: *query*, *trigger*, and *stop* condition. Whenever the trigger condition becomes true, the query part is executed and the part of the query result that is different from the result of the previous execution is returned. The stop condition specifies the termination of a CQ.

Compared to traditional queries, CQs have a number of distinguishing characteristics:

- As opposed to traditional queries that are evaluated over a single snapshot of the data sources and provide one-time results, CQs are evaluated continuously over changing data sources and provide a *stream* of results.

- CQs provide users with a push-based model for accessing information, as opposed to pull-based (request/response) model of traditional queries. User subscriptions are expressed in the form of CQs and interested information updates are *pushed* back to the users by the CQ service in the form of result notifications.

- Evaluating CQs is resource intensive. The continuous nature of the process requires adapting to dynamics of the system during runtime. Traditional queries are usually optimized before execution.

Due to their continuous execution model and push-based result delivery, CQ services are well suited for supporting information monitoring applications. Given the resource intensive nature of CQ evaluation, scaling CQ services is an important consideration in building large-scale information monitoring applications.

1.2 Example Applications

The types of information monitoring applications that can benefit from CQ services are diverse. Here we would like to give three examples from three different domains, namely: i) Web Monitoring in Internet Systems, ii) Location Monitoring in Mobile Systems, and iii) Environmental Monitoring in Sensor Systems.

1.2.1 Web Monitoring in Internet Systems

Web monitoring refers to monitoring and tracking various types of changes to static and dynamic web pages. An example web information monitoring request could be as follows: "Notify me with the IBM stock price (www.quote.com/quotes.aspx?symbols=NYSE:IBM) when the Nasdaq index (www.quote.com/quotes.aspx?symbols=NASDAQ) increases by 5%, during the next 3 months". In CQ terms, the trigger condition of this information monitoring request is "Nasdaq index increases by 5%", the query component is "retreive the IBM stock price", and the stop condition is "3 months pass after query submission".

Centralized software architectures for providing web-based CQ services has been a topic of study in the past [80]. Recent commercial interest in similar functionality, such as Google Alerts [50], and the ever increasing number of web pages [125] attest to the need for architechting scalable and distributed CQ services for web monitoring.

1.2.2 Location Monitoring in Mobile Systems

Location monitoring refers to monitoring and tracking various types of changes to positions of mobile objects. An example location monitoring request could be as follows: "Give me the locations and names of the gas stations offering gasoline for less than $1.2 per gallon within 10 miles, during next half an hour" posted by a car driver. In CQ terms, the trigger condition of this information monitoring request is "there is a change in the set of gas stations within 10 miles of the user's location", the query component is "retrieve locations and names of the gas stations within 10 miles of the user's location", and the stop condition is "half an hour passes after query submission". Note that in this example the query point (user's location) is mobile, whereas the queried objects (gas stations) are still. The example scenarios also include cases in which both the query point and the queried objects are mobile, such as CQs used to monitor nearby taxi services (e.g. Google Ride Finder [51]).

The proliferation of mobile devices is creating an increasing interest in location-based services [111]. The additional complexity brought by the continuously changing nature of the queries in addition to the queried objects, makes location monitoring CQs particularly challenging to evaluate efficiently. The computational capabilities of today's mobile devices create opportunities to scale CQ services through distributed query processing architectures, for supporting large-scale location monitoring applications.

1.2.3 Environmental Monitoring in Sensor Systems

Environmental monitoring refers to monitoring and tracking various types of changes to readings of sensors that are used to capture environmental phenomena, such as temperature, humidity, solar radiation, etc. An example environmental monitoring request could be as follows: "Report me the locations and values of the sensor nodes reporting at least 10% above the average temperature value when the average temperature raises by 5° Celcius, during the next 10 days". In CQ terms, the trigger condition of this information monitoring request is "average temperature raises by 5° Celcius", the query component is "retrieve the locations and values of the sensor nodes reporting at least 10% above the average temperature", and the stop condition is "10 days pass after query submission".

The emergence of small, low cost, and low power sensor devices is enabling seamless integration of the physical world with pervasive networks [41]. CQ services running on networks formed by large number of sensor nodes are highly beneficial for environmental monitoring applications. However, unique features of sensor networks such as their low power budget, as well as the unique features of the data collected from sensor networks such as the high and unpredictable aggregate rates of sensor streams, create challenges in efficient collection of data from sensor networks, and scalable query processing over the data collected from sensor networks. Thus, new techniques and architectures are needed to scale sensor-based CQ services for supporting environmental monitoring applications.

1.3 Challenges and Issues

Scalability, in terms of number of users, queries, and data sources, is a common obstacle in building continuous query services in future computing environments. Large number of long standing continuous queries and data sources on which these CQs are defined put a heavy burden on processing resources (due to cost of frequent query re-evaluation), as well as on network resources (due to frequent access of data items from remote sources to track their updates). Besides these generic problems, there are also challenges specific to different computing platforms on which CQ services are to be built. Diversity of the application domains for continuous queries is an indication of the diversity of the computing platforms involved in building scalable CQ service architectures to support information monitoring applications. The unique features of these different computing platforms necessitate designs that are tailored to the challenges posed and opportunities provided by these platforms.

Traditional distributed computing systems that are based on the client/server model and wired-networks have long served as a strong foundation for the development of distributed information systems. Although client-server based distributed computing paradigms continue to hold their strong place in today's computing systems, some new paradigms have also emerged in recent years. These include large-scale decentralized computing systems exemplified by peer-to-peer overlays and grids, and wireless computing systems exemplified by mobile networks and services. Interestingly, recent developments in sensor networks have brought together decentralized operation and wireless communications in one domain. Following these trends, this thesis aims at developing system-level architectures and techniques to support CQ services in peer-to-peer, mobile, and sensor platforms.

We now discuss some of the limitations and challenges faced in P2P systems, mobile systems, and sensor systems, as well as challenges faced in scaling CQ services in supporting information monitoring applications in these computing platforms.

1.3.1 Peer-to-Peer Systems

Peer-to-peer (P2P) systems are massively distributed computing systems in which nodes communicate with one another to distribute tasks, exchange information, and collaboratively perform a function in a decentralized manner. P2P computing has a number of attractive properties. First, P2P systems do not rely on centralized control, but have decentralized operation. This avoids central-point of failure and provides better fault-tolerance. Second, P2P systems exhibit self-organizing and self-adapting behavior. Thus, they do not incur administrative costs associated with setup, reconfiguration, and maintenance. Third but not the least, P2P systems scale with the number of clients, since the nodes of the system (called peers) act both as servers and clients.

1.3.1.1 General Challenges

P2P computing possesses many advantages, but it also brings about many challenges [86]. In particular, decentralized control makes it harder to make optimal choices with regard to system configuration. This is due to the lack of global knowledge at any node and the difficulty of resource look-up. The large number of nodes that are both clients and servers result in high peer failure rate, which makes providing quality of service and achieving high levels of reliability harder than they are in traditional distributed systems.

1.3.1.2 CQ Specific Challenges

In P2P computing, partitioning of long standing continuous queries to processing nodes (peers) entails new techniques for coordinated use of network and processing resources. Balancing peer loads and maximizing system utilization are two major challenges in the presence of node heterogeneity, decentralized operation, and non-uniform user interest distributions. Moreover, replication techniques are needed for ensuring reliable execution of long standing monitoring queries on a heterogeneous network of peer nodes, with high peer failure and turn-over rate. Such replication techniques should establish uninterrupted execution of CQs, in the existence of peer failures and frequent peer departures.

1.3.2 Mobile Systems

Mobile systems are characterized by lightweight clients that are battery powered and can communicate wirelessly, which effectively make them mobile and ubiquitous. Mobile systems provide users with the ability to stay on-line while on-the-go and access information services pervasively. These information services are usually tailored toward the location and/or activity context of the mobile users; much to their convenience. While providing opportunities to offer value-added services to users, mobile systems also create many challenges in designing, building, and deploying such services, mainly due to the unique properties of mobile devices and networks [108].

1.3.2.1 General Challenges

Mobile devices are resource constrained, in terms of processing power as well as memory and disk capacity. They have low power budget and limited network bandwidth (which may be asymmetric as in the case of cellular networks, in which upload bandwidth is usually lower than download bandwidth). Wireless communication is unreliable and may not always be available, resulting in intermittent connection. Due to the small size of mobile devices, user interfaces are also restricted. These challenges set mobile systems apart from traditional distributed systems, and they should be taken into account when designing software architectures for providing information services in mobile systems.

1.3.2.2 CQ Specific Challenges

When nodes are mobile, the highly dynamic nature of position information and limited wireless bandwidth available to mobile nodes introduce new challenges to location monitoring applications, due to the spatio-temporal nature of mobile continuous query services. On one hand, such continuous location services demand scalable server side processing to handle the large amount of highly dynamic location queries. On the other hand, the limited wireless bandwidth available to mobile devices and the location-dependent characteristics of queries call for distributed and bandwidth aware architecture design that is effective for partitioning continuous query processing tasks and utilizing processing capabilities of mobile clients. Such distributed architectures, that push some of the query processing to the mobile client side, should also respect the resource-limited nature of mobile clients.

1.3.3 Sensor Systems

Sensor nodes are tiny, low cost, and low power devices that can perform computation, wireless communication, and environmental sensing. Networks formed by sensor nodes enable seamless integration of the physical world with pervasive networks [59]. The decentralized and unattended nature of operation make sensor networks an attractive tool for extracting and gathering data by sensing real-world phenomena. As a result, environmental monitoring applications are expected to benefit enormously from sensor network services.

1.3.3.1 General Challenges

The large number of networked sensors brings a number of unique system design challenges, different from those posed by traditional distributed systems. First, a major limitation of sensor devices is their limited battery life. Different from mobile devices, sensor nodes are not easily recharged due to their tiny form factor and often harsh physical deployment environments. Second, unreliable wireless communication, asymmetric connectivity between the nodes, and ad-hoc network infrastructure make building reliable distributed protocols within sensor networks a significant challenge. Third, sensor devices have scarce resources, in terms of cpu, memory, and wireless bandwidth. These resources are orders of magnitude smaller in capacity compared to most mobile devices. Thus, resource-aware low-power designs are not only desirable but a key design consideration in building in-network services for supporting sensor network applications.

1.3.3.2 CQ Specific Challenges

When the nodes are low cost and low power sensor nodes capable of environmental sensing and producing continuous streams of data, we need to address two additional challenges in order to make continuous query services scalable. First, we need energy-efficient data collection mechanisms to extract data streams from power constrained sensor nodes, in order to achieve longer network lifetimes. Second, we need scalable techniques for energy-efficient processing of data streams within the sensor network and for resource-aware evaluation of continuous queries over sensor streams at query processing centers. High aggregate rate of sensor streams calls for a paradigm shift in how CQs are processed, moving away from the traditional "store and then process" model of database management systems to "on-the-fly processing" model of emerging data stream management systems, such as StreamBase [117].

1.4 Contribution of the Thesis

This thesis aims at developing system-level architectures and techniques to support continuous query services for future computing platforms and applications, focusing on information monitoring applications in mobile, peer-to-peer, and sensor network computing domains.

To our knowledge, this is the first dissertation work that utilizes CQs to shepherd through and address the challenges involved in supporting information monitoring applications in future computing platforms. Before we present the individual contributions of this thesis in each of these areas, namely contributions with regard to P2P CQ services, mobile CQ services, and sensor CQ services, we first discuss the common design philosophy this thesis have in solving problems related with scalability of CQ services in these areas.

1.4.1 Common Design Philosophy

Although different computing platforms require different software architectures for building CQ services, there is a common design philosophy that this thesis advocates for making CQ services scalable and efficient. This same design philosophy is applied in developing solutions to provide scalable CQ services in the three application domains we have discussed so far, each associated with a different computing platform. These are: P2P web monitoring in Internet systems, location monitoring in mobile systems, and environmental monitoring in networked sensor systems. This common philosophy can be briefly stated as follows: "*Move computation close to the places where the data is produced.*"

One of the challenges in CQ systems is the resource-intensive nature of query evaluation, a major part of which is continuously checking updates in a large number of data sources and evaluating trigger conditions of a large number of queries over these updates, consuming both cpu and network bandwidth resources. If we can move some part of the query evaluation close to the sources where the data is produced, the resulting early filtering of updates will save both bandwidth and cpu resources.

In the context of P2P information monitoring on the Internet, executing CQs at peers that are closer to the data sources will save bandwidth resources, as well as cpu resources by the way of grouping similar CQs whose triggers are defined on the same data sources. In the context of location monitoring in mobile systems, moving part of the query evaluation into the mobile nodes, in order to filter some of the position updates reported wirelessly to the central server for processing, will reduce both the wireless network bandwidth consumed and the cpu resources used at the server side. And finally, in the context of environmental monitoring in sensor systems, moving certain filtering functionality into the individual sensor nodes will reduce the amount of wireless communication required to extract data from the network, and thus will improve the network lifetime.

We now list the individual contributions of this thesis in these areas:

1.4.2 P2P CQ Services

In regard to P2P CQ services, this thesis addresses the challenges involved in executing large number of continuous queries on a network of unreliable peers. A decentralized architecture called *PeerCQ* and effective service partitioning algorithms associated with it are developed for distributing continuous queries to peers in a way that improves system utilization, while maintaining good load balance of peers and providing high reliability. The main technical highlights of this development are twofold, the *service partitioning* mechanism and the *dynamic replication* mechanism.

1.4.2.1 Service Partitioning

Service partitioning is defined as the assignment of CQs to peers for execution. We develop an effective mechanisms for service partitioning at the P2P protocol layer. Concrete contributions include the donation-based peer-aware query assignment mechanism for handling peer heterogeneity, request similarity based CQ-aware assignment mechanism for handling skewness in information monitoring requests, and the relaxed matching technique that provides a good balance between system utilization and load balance in the presence of peer joins, departures, and failures. We present experimental results to show that due to non-uniform nature of information monitoring requests, effective grouping of CQs will benefit both load balance and system utilization, but maximizing the grouping of similar CQs and thus the overall system utilization will not result in good load balance.

1.4.2.2 Dynamic Replication

We develop an effective dynamic passive replication scheme to ensure reliable processing of long-running information monitoring requests in a decentralized environment of inherently unreliable peers. We derive analytical models to formalize the fault-tolerance properties of this replication scheme. We present experimental results to demonstrated that, when equipped with this dynamic replication scheme, PeerCQ is able to achieve high reliability for moderate values of the replication factor.

1.4.3 Mobile CQ Services

In regard to mobile CQ services, this thesis develops a distributed continuous location query processing framework that promotes effective partitioning of continuous location query processing between the server and the mobile clients, by taking advantage of the increasingly available computational power available in mobile clients. In addition, we develop a centralized continuous location query processing framework that employs motion adaptive spatio-temporal indexes to evaluate continuous location queries in an IO and time efficient manner, on the server side. These two solutions are complementary in the sense that the former is more suitable for resource-wise more capable mobile clients, whereas the latter one better suits mobile clients with scarce resources that can not contribute to query processing.

1.4.3.1 Distributed Location Monitoring

We develop a distributed location monitoring architecture through the design of *MobiEyes*. The main idea behind MobiEyes' distributed architecture is to promote a careful partitioning of a near real-time location monitoring task into an optimal coordination of server-side processing and client-side processing. Such a partitioning allows evaluating moving location queries with a high degree of precision using a small number of location updates, thus providing highly scalable and more cost-effective location monitoring services. A set of optimization techniques are used to limit the amount of computation to be handled by the mobile objects and enhance the overall performance and system utilization of MobiEyes. We present experimental results to show that MobiEyes can lead to significant savings in terms of server load and messaging cost when compared to solutions relying solely on central processing of location information.

1.4.3.2 Motion-adaptive Indexing

We develop a motion adaptive indexing scheme for efficient evaluation of moving continuous queries over mobile object locations, in a centralized server. It uses the concept of motion-sensitive bounding boxes to model moving objects and moving queries. These bounding boxes automatically adapt their sizes to the dynamic motion behaviors of individual objects. Instead of indexing frequently changing object positions, less frequently changing object and query motion-sensitive bounding boxes are indexed. This helps decrease the number of updates to the indexes. More importantly, predictive query results are employed to optimistically pre-calculate query results, and thus decrease the number of searches on the indexes. We present experiments to show that the proposed motion adaptive indexing scheme is IO and time efficient for the evaluation of moving continuous queries.

1.4.4 Sensor CQ Services

In regard to sensor CQ services, this thesis develops energy-efficient techniques for continuous data collection in sensor networks. Such data streams extracted from sensor networks are key to building environmental monitoring applications. To further support these applications, we develop techniques for evaluating CQs over sensor streams in a resource-aware fashion, adapting to the rates as well as other key characteristics of the sensor streams.

1.4.4.1 Sensor Network Data Collection

We develop an adaptive model-based prediction framework that uses the *selective sampling* concept to enable energy-efficient data collection in sensor networks. It requires smaller number of nodes to sample and report their values, thus significantly lowering the power consumption. Values of non-sampler nodes are predicted using the locally derived probabilistic models. We present experimental results to show that this framework can achieve high accuracy with low power consumption, by exploiting the strong spatial and temporal correlations existent among the sensor readings.

1.4.4.2 Data Stream Processing

We develop rate-aware adaptive query evaluation techniques for scalable processing of long running window-based continuous queries over sensor streams. We present results to show that adaptation to stream statistics, such as stream rates and time-correlations among different streams, can improve result accuracy during rate bursts when the available computational resources are not sufficient to process every update in a timely manner, especially for large window sizes or costly query conditions. The main focus is on multi-way stream joins and resource constrained scenarios.

1.5 Organization of the Thesis

The rest of this thesis is organized as a series of chapters, each one dedicated to a specific topic within the context of P2P, mobile, and sensor CQ systems. In each of these chapters, background information and system models are given before the core technical content is described. The specific contributions are given in the introduction part of each chapter, whereas the related work in the literature is reported at the end of each chapter. Concretely, this thesis is composed of the following chapters.

Chapter 2 presents the PeerCQ system, which is a peer-to-peer architecture for continuous query-based information monitoring on the Internet. Two major components of the PeerCQ system, namely the service partitioning mechanism and the dynamic replication mechanism, are discussed and several experimental results are presented to study the scalability and the effectiveness of the PeerCQ system.

Chapter 3 presents the MobiEyes system, which is a distributed architecture for continuous query-based location monitoring in mobile systems. Mechanisms for partitioning of CQ processing between the server side and the mobile client side, as well as mechanisms for mobile client side optimizations, are discussed. Experimental results are presented to study the scalability and the effectiveness of the MobiEyes system.

Chapter 4 presents MAI, a motion-adaptive indexing scheme for continuous query-based centralized location monitoring in mobile systems. Two major components of the MAI scheme, namely the motion-sensitive bounding boxes and the predictive query results, are discussed and several experimental results are presented to study the scalability and the effectiveness of the MAI approach.

Chapter 5 presents an energy-efficient data collection framework for sensor CQ systems. There main components of this framework are discussed in order – i) sensing-driven cluster construction, ii) correlation-based sampler selection and model derivation, and iii) selective data collection and model-based prediction. Experimental results are presented to study the scalability and the effectiveness of this data collection framework.

Chapter 6 presents resource-aware query evaluation techniques for CQ systems dealing with sensor streams. An algorithm, called GrubJoin, is presented for performing load shedding with the aim of maximizing the output rate of multi-way join queries in CPU limited scenarios. Experimental results are presented to study the scalability and the effectiveness of GrubJoin.

Chapter 7 discusses some open issues and concludes the thesis.

CHAPTER II

PEERCQ - P2P INFORMATION MONITORING IN INTERNET SYSTEMS USING CQS

We present PeerCQ, a decentralized architecture for Internet scale information monitoring using a network of heterogeneous peer nodes. PeerCQ uses Continual Queries (CQs) as its primitives to express information-monitoring requests. The PeerCQ development has three unique characteristics. First, we develop a systematic and serverless approach to large scale information monitoring, aiming at providing a fully distributed, highly scalable and self-configurable architecture for scalable and reliable processing of large number of CQs over a network of loosely coupled, heterogeneous, and possibly unreliable nodes (peers). Second, we introduce an effective service partitioning scheme at the P2P protocol layer to distribute the processing of CQs over a peer-to-peer information monitoring overlay network, while maintaining a good balance between system utilization and load balance in the presence of peer joins, departures and failures. A unique feature of our service partitioning scheme is its ability to incorporate strategies for handling hot spot monitoring requests and peer heterogeneity into the load balancing scheme in PeerCQ. Third but not the least, we develop a dynamic passive replication scheme to enable reliable processing of long-running information monitoring requests in an environment of inherently unreliable peers, including an analytical model to discuss its fault tolerance properties. We report a set of experiments demonstrating the feasibility and the effectiveness of the PeerCQ approach.

2.1 Introduction

Peer-to-peer (P2P) systems are massively distributed computing systems in which peers (nodes) communicate directly with one another to distribute tasks, exchange information or share resources. There are currently several P2P systems in operation, and many more are under development. Gnutella [47] and Kazaa [71] are among the most prominent first generation peer-to-peer file sharing systems operational today. These systems are often referred to as unstructured P2P networks and they share two unique characteristics. First, the topology of the overlay network and the placement of the files within the network are largely unconstrained. Second, they use a decentralized file lookup scheme. Requests for files are flooded with a certain scope. There is no guarantee to find an existing file within a bounded number of hops. The random topology combined with flooding-based routing is clearly not scalable since the load on each peer grows linearly with the total number of queries in the network, which in turn grows with the size of the system.

Chord [116], Pastry [103], Tapestry [142], CAN [101] are examples of the second generation of peer-to-peer systems. Their routing and location schemes are structured based on distributed hash tables. In contrast to the first generation P2P systems such as Gnutella and Kazaa, these systems provide guaranteed content location (persistence and availability) through a tighter control of the data placement and the topology construction within a P2P network. Queries on existing objects are guaranteed to be answered in a bounded number of network hops. Their P2P routing and location schemes are also considered more scalable. These systems differ from one another in terms of their concrete P2P protocol design, including the distributed hash algorithms, the lookup costs, the level of support for network locality, and the size and dependency of routing table with respect to the size of the P2P overlay network.

Surprisingly, many existing P2P protocols [35, 116, 101, 103, 47] do not distinguish peer heterogeneity in terms of computing and communication capacity. As a result, these protocols distribute tasks and place data to peers assuming all peers participate and contribute equally to the system. Work done in analyzing characteristics of Gnutella in [107] shows that peers participating in these systems are heterogeneous with respect to many characteristics, such as connection speeds, CPU, shared disk space, and peers' willingness to participate. These evidences show that P2P applications should respect the peer heterogeneity and user (application) characteristics in order to be more robust [107].

In this chapter we describe PeerCQ [45], a peer-to-peer information monitoring system, which utilizes a large set of heterogeneous peers to form a peer-to-peer information monitoring network. Many application systems today have the need for tracking changes in multiple information sources on the web and notifying users of changes if some condition over the information sources is met. A typical example in business world is to monitor availability and price information of specific products, such as "monitor the price of 5 mega pixel digital cameras during the next two months and notify me when one with price less than $500 becomes available", "monitor the IBM stock price and notify me when it increases by 5%". In a large scale information monitoring system [78], many users may issue the same information monitoring request such as tracking IBM stock price changes during a given period of time. We call such phenomenon the *hot spot* queries (monitoring requests). Optimizations for hot spot queries can significantly reduce the amount of duplicate processing and enhance the overall system utilization.

In general, offering information-monitoring service using a client/server architecture imposes two challenging requirements on the server side. First, the server should have the ability to handle tens of thousands or millions of distributed triggers firing over hundreds or thousands of Web sites. Second, the server should be scalable as the number of triggers to be evaluated and the number of Web sites to be monitored increase. It is widely recognized that the client/server approach to large-scale information monitoring is expensive to scale, and expensive to maintain. The server side forms a single point of failure.

Compared to information monitoring in client-server systems, peer-to-peer information monitoring has a number of obvious advantages. First, there is no additional administrative management cost as the system grows. Second, there is no hardware or connection cost, since the peers are the user machines and the connections to the data sources are the user connections. Third, there is no upgrade cost due to scaling since resources grow with clients. The only cost for PeerCQ information monitoring from the perspective of a service provider is the cost of developing the PeerCQ application and making it work effectively in practice.

PeerCQ uses continual queries as its primitives to express information monitoring requests. Continual Queries (CQs) [78] are standing queries that monitor information updates and return results whenever the updates reach certain specified thresholds. There are three main components of a CQ: query, trigger, and stop condition. Whenever the trigger condition becomes true, the query part is executed and the part of the query result that is different from the result of the previous execution is returned. The stop condition specifies the termination of a CQ.

PeerCQ poses several technical challenges in providing information monitoring services using a P2P computing paradigm. The first challenge is the need of a smart service-partitioning mechanism. The main issue regarding service partitioning is to achieve a good balance between improving the overall system utilization and maintaining the load balance among peers of the system. By balanced load we mean there are no peers that are overloaded. By system utilization, we mean that when taken as a whole the system does not incur large amount of duplicated computations or consume unnecessary resources such as the network bandwidth between the peers and the data sources. Several factors can affect the load balancing decision, including the computing capacity and the desired resource contribution of the peers, the willingness of peers to participate, and the characteristics of the continual queries. The second technical challenge is the reliability of CQ processing in the presence of peer departures and failures. The study reported in [107] shows that large scale peer-to-peer systems are confronted with high peer turnover rate. In PeerCQ, failure handling mechanisms are needed to detect failures and ensure correct CQ execution.

2.2 System Overview

Peers in the PeerCQ system are user machines on the Internet that execute information monitoring applications. Peers act both as clients and servers in terms of their roles in serving information monitoring requests. An information-monitoring job, expressed as a continual query (CQ), can be posted from any peer in the system. There is no scheduling node in the system. No peers have any global knowledge about other peers in the system.

There are three main mechanisms that make up the PeerCQ system. The first mechanism is the overlay network membership. Peer membership allows peers to communicate directly with one another to distribute tasks or exchange information. A new node can join the PeerCQ system by contacting an existing peer (an entry node) in the PeerCQ network.

There are several bootstrapping methods to determine an entry node. We may assume that a PeerCQ service has an associated DNS domain name. It takes care of resolving the mapping of PeerCQ's domain name to the IP address of one or more PeerCQ bootstrapping nodes. A bootstrapping node maintains a short list of PeerCQ nodes that are currently alive in the system. To join PeerCQ, a new node looks up the PeerCQ domain name in DNS to obtain a bootstrapping node's IP address. The bootstrapping node randomly chooses several entry nodes from the short list of nodes and supplies their IP addresses. Upon contacting to an entry node of PeerCQ, the new node is integrated into the system through the PeerCQ protocol's initialization procedures.

The second mechanism is the PeerCQ protocol, including the service partitioning and the routing query based lookup algorithm. In PeerCQ every peer participates in the process of evaluating CQs, and any peer can post a new CQ of its own interest. When a new CQ is posted by a peer, this peer first determines which peer will process this CQ with the objective of utilizing system resources and balancing the load on peers. Upon a peer's entrance into the system, a set of CQs that needs to be re-distributed to this new peer is determined by taking into account the same objectives. Similarly, when a peer departs from the system, the set of CQs of which it was responsible is re-assigned to the rest of peers, while maintaining the same objectives − maximize the system utilization and balance the load of peers.

The third mechanism is the processing of information monitoring requests in the form of continual queries (CQs). Each information monitoring request is assigned to an identifier. Based on an identifier matching criteria, CQs are executed at their assigned peers and cleanly migrated to other peers in the presence of failure or peer entrance and departure.

Figure 1 shows a sketch of the PeerCQ system architecture from a user's point of view. Each peer in the P2P network is equipped with the PeerCQ middleware, a two-layer software system. The lower layer is the PeerCQ protocol layer responsible for peer-to-peer communication. The upper layer is the information monitoring subsystem responsible for CQ subscription, trigger evaluation, and change notification. Any domain-specific information monitoring requirements can be incorporated at this layer.

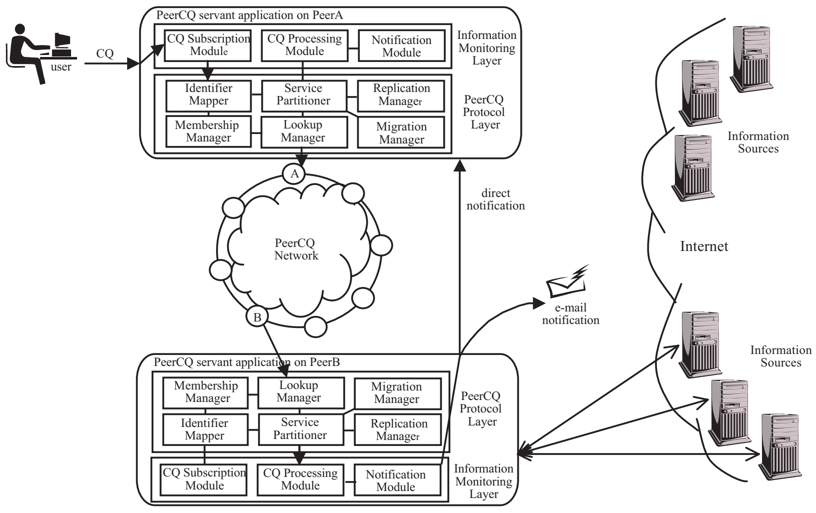


Figure 1: PeerCQ architecture

A user composes his or her information monitoring request in terms of a CQ and posts it to the PeerCQ system via an entry peer, say `Peer A`. Based on the PeerCQ's service partition scheme (see Section 2.3.2), `Peer A` is not responsible for this CQ. Thus it triggers the PeerCQ's P2P lookup function. The PeerCQ system determines which peer will be responsible for processing this CQ using the PeerCQ service partitioning scheme. Assume that `Peer B` was chosen to execute this CQ. `Peer B` is referred to as the *executor* peer of this CQ. After the CQ is assigned to `Peer B`, it starts its execution there. During this execution, when an interested information update is detected, the query is fired, and the peer that posts this CQ is notified with the newly updated information. The notification could be realized by e-mail or by directly sending it to `Peer A` if it is online at the time of notification. Note that, even if a peer is not participating in the system at a given time, its previously posted CQs are in execution at other peers.

2.3 The PeerCQ P2P Protocol

The PeerCQ protocol specifies three important types of peer coordination: (1) how to find peers that are best to serve the given information monitoring requests in terms of load balance and overall system utilization, (2) how new nodes join the system, and (3) how PeerCQ manages failures or departures of existing nodes.

2.3.1 Overview

Similar to most of the distributed hash table (DHT) based P2P protocols [69, 35, 116, 103, 142, 93], PeerCQ provides a fast and distributed computation of a hash function, mapping information monitoring requests (in form of continual queries) to nodes responsible for them. PeerCQ protocol design differs from other DHT based protocols, such as Chord [116], Pastry [103], Tapestry [142], and CAN [101], in a number of ways. First, PeerCQ provides two efficient mapping functions as the basic building blocks for distributing information monitoring requests (CQs) to peers with heterogeneous capabilities. The mapping of peers to identifiers takes into account of peer heterogeneity and load dynamics at peers to incorporate peer awareness into the service partitioning scheme. The mapping of CQs to identifiers incorporates CQ grouping optimization [81] for hot spot CQs found frequently in large scale information monitoring applications, striving for efficient processing of large number of information monitoring requests and minimizing the cost for duplicate processing of hot spot CQs. Second, PeerCQ introduces relaxed matching algorithms on top of the strict matching based on numerical distance between CQ identifiers and peer identifiers, when distributing CQs to peers, aiming at achieving good load balance and good system utilization.

In PeerCQ, an information monitoring request (subscription) is described in terms of a continual query (CQ). Formally, a CQ is defined as a quadruplet, denoted by cq : (cq_id, $trigger$, $query$, $stop\_cond$) [78]. cq_id is the unique identifier of the CQ, which is an m-bit unsigned value. $trigger$ defines the target data source to be monitored (mon_src), the data items to be tracked for changes (mon_item), and the condition that specifies the update threshold (amount of changes) of interest (mon_cond). $query$ part specifies what information should be delivered when the mon_cond is satisfied. $stop\_cond$ specifies the termination condition for the CQ. For notational convenience, in the rest of this chapter a CQ is referenced as a tuple of seven attributes, namely cq : (cq_id, mon_src, mon_item, mon_cond, $query$, $notification$, $stop\_cond$). Consider the example monitoring request "monitor Nasdaq index and tell me IBM stock price when Nasdaq index value increases by 5% in the next three months". One way to express this request is to use the following continual query: $\langle$ *cq_id*, *mon_src*: http://www.quote.com, *mon_item*: Nasdaq index (/quotes.aspx?symbols=NASDAQ), *mon_cond*: increase by 5%, *query*: IBM stock price (/quotes.aspx?symbols=NYSE:IBM), *notification*: my email, *stop_cond*: next 3 months $\rangle$.

The PeerCQ system provides a distributed service partitioning and lookup service that allows applications to register, lookup, and remove an information monitoring subscription using an m-bit CQ identifier as a handle. It maps each CQ subscription to a unique, effectively random m-bit CQ identifier. To enable efficient processing of multiple CQs with similar trigger conditions, the CQ-to-identifier mapping also takes into account the similarity of CQs such that CQs with the similar trigger conditions can be assigned to same peers (see Section 2.3.2 for details). This property of the PeerCQ is referred to as *CQ-awareness*.

Similarly, each peer in PeerCQ corresponds to a set of m-bit identifiers, depending on the amount of resources donated by each peer. A peer that donates more resources is assigned to more identifiers. We refer to this property as *Peer-awareness*. It addresses the service partitioning problem by taking into account of peer heterogeneity and by distributing CQs over peers such that the load of each peer is commensurate to the peer capacities (in terms of cpu, memory, disk, and network bandwidth). Formally, let P denote the set of all peers in the system. A peer p is described as a tuple of two attributes, denoted by $p : (\{peer\_ids\}, (peer\_props))$. $peer\_ids$ is a set of m-bit identifiers. No peers share any identifiers, i.e. $\forall p, p' \in P, p.peer\_ids \cap p'.peer\_ids = \emptyset$. The identifier length m must be large enough to make the probability of two nodes or two CQs hashing to the same identifier negligible. *peer_props* is a composite attribute which is composed of several peer properties, including IP address of the peer, peer resources such as connection type, CPU power and memory, and so on. The concrete resource donation model may be defined by PeerCQ applications (see Section 2.3.2 for further details).

Identifiers are ordered in an m-bit identifier circle modulo 2^m. The 2^m identifiers are organized in an increasing order in the clockwise direction. To guide the explanation of the PeerCQ protocol, we define a number of notations:

– The distance between two identifiers i, j, denoted as $Dist(i, j)$, is the shortest distance between them on the identifier circle, defined by $Dist(i, j) = min(|i - j|, 2^m - |i - j|)$. By this definition $Dist(i,j) = Dist(j,i)$ holds.

$-$ Let $path(i,j)$ denote the set of all identifiers on the clockwise path from identifier i to identifier j on the identifier circle. An identifier k is said to be *in-between* identifiers i and j, denoted as $k \in path(i,j)$, if $k \neq i$, $k \neq j$, and it can be reached before j going in the clockwise path starting at i.

$-$ A peer p' with its peer identifier j is said to be an *immediate right neighbor* to a peer p with its peer identifier i, denoted by $(p',j) = IRN(p,i)$, if there are no other peers having identifiers in the clockwise path from i to j on the identifier circle. Formally the following condition holds: $i \in p.peer\_ids \wedge j \in p'.peer\_ids \wedge \nexists p'' \in P$ s.t. $\exists k \in p''.peer\_ids$ s.t. $k \in path(i,j)$. The peer p with its peer identifier i can also be referred to as the *immediate left neighbor*(ILN) of peer p' with its identifier j. The definition is symmetric.

$-$ A *neighbor list* of a peer p_0 associated with one of its identifiers i_0, denoted by $NeighborList(p_0, i_0)$, is formally defined as follows: $NeighborList(p_0,\ i_0) = [(p_{-r},\ i_{-r}),\ \ldots,\ (p_{-1},\ i_{-1}),\ (p_0,\ i_0),\ (p_1,\ i_1),\ \ldots,\ (p_r,\ i_r)]$, s.t. $\bigwedge_{k=1}^{r}((p_k,\ i_k) = IRN(p_{k-1},\ i_{k-1})) \wedge \bigwedge_{k=1}^{r}((p_{-k},\ i_{-k}) = ILN(p_{-k+1},\ i_{-k+1}))$. The size of the neighbor list is $2r+1$ and we call r the neighbor list parameter. Informally, the neighbor list of a peer p_0 with identifier i_0 consists of three components: (1) the first r number of identifiers on the clockwise path starting from i_0, (2) the first r number of identifiers on the counter clockwise path starting from i_0, and (3) identifier i_0 of p_0 itself. For each identifier, the neighbor list also stores the peer who owns this identifier.

Table 1: Basic API of the PeerCQ system

Function	Description
p.join(out: *status*)	a node p adds itself to the PeerCQ system
p.leave(out: *status*)	a node p departs the PeerCQ system
p.post(in: *cq*, out: *cq_id*)	a node p posts a *cq* to the system for execution
p.terminate(in: *cq_id*)	a node p posts a termination request for a CQ with id. *cq_id*.

The basic application interface (API) provided by the PeerCQ system consists of four basic functions, shown in Table 1. The first two API calls are functions for nodes to join or leave the PeerCQ system. p.join($status$) function adds a node p to the PeerCQ system and returns status information regarding the result of this operation. p.leave($status$) function departs a node p from the system and returns a status value indicating whether an error has occurred or not. Given a peer p, when p.post(cq, cq_id) is called, PeerCQ finds the destination peer that should be responsible for executing the cq posted by peer p and ships cq to that destination peer for execution. We call p the *initiator* peer of the given CQ and the destination peer the *executor* peer of the CQ. p.post(cq, cq_id) returns the cq_id as a result of the operation. The function p.terminate(cq_id) is used by the issuer peer of a CQ to terminate the processing of its CQ specified by the identifier cq_id.

2.3.2 Capability-Sensitive Service Partitioning

The PeerCQ protocol extends the existing routed-query based P2P protocols, such as Chord [116] or Pastry [103], to include a capability-sensitive service partitioning scheme. Service partitioning can be described as the assignment of CQs to peers. By capability-sensitive, we mean that the PeerCQ service partitioning scheme extends a randomized partition algorithm, commonly used in most of the current DHT-based protocols, with both *peer-awareness* and *CQ-awareness* capability. As demonstrated in [116, 103, 142, 93], randomized partitioning schemes are easy to implement in decentralized systems. However they perform poorly in terms of load balancing in heterogeneous peer-to-peer environments.

PeerCQ capability-sensitive service partitioning manages the assignment of CQs to appropriate peers in three stages: (1) mapping peers to identifiers to address peer-awareness; (2) mapping CQs to identifiers to address CQ-awareness, and (3) matching CQs to peers in a two-phase matching algorithm. The main objective for the PeerCQs capability-aware service partitioning is two folds. First, we want to balance the load of peers in the system while improving the overall system utilization. Second, we want to optimize the hot spot CQs such that the system as a whole does not incur large amount of redundant computations or consume unnecessary resources such as the network bandwidth between the peers and the data sources.

We implement *peer-awareness* based on peer donation and dynamic mapping of peers to identifiers. Each peer donates a self-specified portion of its resources to the system and can dynamically adjust the amount of donations through on-demand or periodical revision of donations. The scheduling decisions are based on the amount of donated resources. We implement *CQ-awareness* by distributing CQs having similar triggers to the same peers. Two CQs, cq and cq', are considered *similar* if they are interested in monitoring updates on the same item from the same source, i.e. $cq.mon\_src = cq'.mon\_src \wedge cq.mon\_item = cq'.mon\_item$. These CQs share the same monitoring source (a stock quote web site) and the same monitoring item (IBM stock price). By CQ-awareness, we mean that CQs with similar triggers will be assigned to and processed by the same peers.

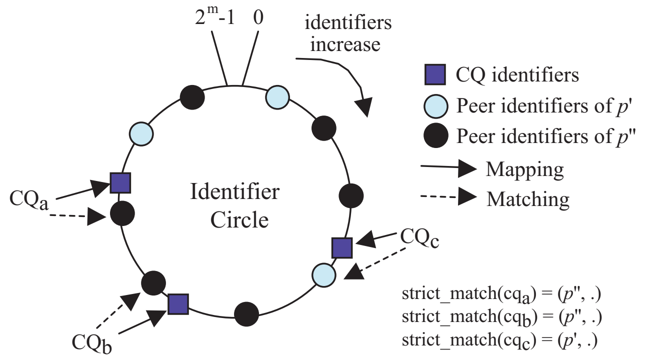

2^m-1
0
identifiers increase
CQ identifiers
Peer identifiers of p'
Peer identifiers of p''
Mapping
Matching
Identifier Circle
CQ_a
CQ_b
CQ_c
strict_match(cq_a) = (p'', .)
strict_match(cq_b) = (p'', .)
strict_match(cq_c) = (p', .)

Figure 2: Example peer to peer id. set mapping

2.3.2.1 Mapping peers to identifiers

In PeerCQ a peer is mapped to a set of m-bit identifiers, called the peer's identifier set ($peer\_ids$). m is a system parameter and it should be large enough to ensure that no two nodes share an identifier or this probability is negligible. To balance the load of peers with heterogeneous resource donations when distributing CQs to peers, the peers that donate more resources are assigned more peer identifiers, so that the probability that more CQs will be matched to those peers is higher. Figure 2 shows an example of mapping of two peers, say p' and p'', to their peer identifiers. Based on the amount of donations, peer p' has 3 peer identifiers, whereas peer p'' has 6. The example shows that p'' is assigned more CQs than p' using the strict matching defined in Section 2.3.2.3.

The number of identifiers to which a peer is mapped is calculated based on a peer donation scheme. We introduce the concept of *ED* (effective donation) for each peer in the PeerCQ network. *ED* of a peer is a measure of its donated resources effectively perceived by the PeerCQ system. For each peer, an effective donation value is first calculated and later used to determine the number of identifiers onto which this peer is going to be mapped. The calculation of ED is given in Appendix A. The mapping of a peer to peer identifiers needs to be as uniform as possible. This can be achieved by using the base hashing functions like MD5 or SHA1 (or any well-known message digest function).

The following algorithm explains how the peer identifier set is formed given the effective donation of a peer:

```
GeneratePeerIDs($p$, $ED$)
(1) $p.peer\_ids \leftarrow \emptyset$
(2) for $i = 1$ to DONATION_TO_IDENT($ED$)
(3)     $d \leftarrow$ CONCAT($p.peer\_props.IP$, $counter$)
(4)     add SHA1($d$, $m$) into $p.peer\_ids$
(5)     increment $counter$
```

The function *donation_to_ident* is responsible of mapping the effective donation value of a peer to the number of m-bit identifiers in the 2^m identifier space, which forms the peer's peer identifier set. $SHA1$ is a message digest function. The first parameter of this digest function is the input message that will be digested. The input message is formed by concatenating the IP address of the peer, and a counter which is initialized to a one second real time clock at the initialization time and incremented each time a peer identifier is generated. This concatenation forms a unique input message for each peer identifier. The second parameter m is the length of the output in bits.

In summary, the peer-to-identifier mapping algorithm maps a peer to a set of randomly chosen m-bit identifiers on the m-bit identifier circle. The number of identifiers each peer will be mapped to is determined in terms of the effective donation (ED) of the peer, initially computed upon the entry of the peer to the PeerCQ overlay network. In order to handle run-time fluctuations of workload at each peer node, a peer can revise its effective donation through periodic updates or by an on-demand revision upon sudden load surge experienced at the peer node.

2.3.2.2 Mapping CQs to identifiers

This mapping function maps a CQ to an m bit identifier in the identifier space with modulo 2^m. An important design goal for this mapping function is to implement CQ-awareness by mapping CQs with similar triggers (same monitoring sources and same monitoring items) to the same peers as much as possible, in order to produce the CQ to peer matching that achieves higher overall utilization of the system.

A CQ identifier is composed of two parts. The first part is expected to be identical for similar CQs and the second part is expected to be uniformly random to ensure the uniqueness of the CQ identifiers. This mechanism allows similar CQs to be mapped into a contiguous region on the m-bit identifier circle. The length of a CQ identifier is m. The length of the first part of an m-bit CQ identifier is a, which is a system parameter called *grouping factor*. Given m and a, the method that maps CQs to the CQ identifiers uses two message digest functions. A sketch of the method is described as follows:

```
CalculateCQID($p$, $cq$)
(1) $d \leftarrow$ CONCAT($cq.mon\_src$, $cq.mon\_item$)
(2) $part1 \leftarrow$ SHA1($d$, $a$)
(3) $d \leftarrow$ CONCAT($p.peer\_props.IP$, $counter$)
(4) $part2 \leftarrow$ SHA1($d$, $m - a$)
(5) $cq.cq\_id \leftarrow$ CONCAT($part1$, $part2$)
(6) increment $counter$
```

The first digest function generates the same output for similar CQs, and the second digest function generates a globally unique output for each CQ posted by a peer. Although there is a small probability of generating the same identifier for two different CQs, the collisions can be detected by querying the network with the generated identifier before posting the CQ. Similar argument is valid for peer identifiers. The first parameter of the first digest function is the concatenation of the data source and the item of interest being monitored. The second parameter is the length of the output in bits. The second digest function generates a random number of length $m - a$ for each CQ. The first parameter of the second digest function is the concatenation of the IP address of the peer posting this CQ, and a counter which is initialized to a one-second real time clock at the initialization time and incremented each time a CQ identifier is generated. The second parameter is the length of the output in terms of bits. The CQ-to-identifier mapping returns an m-bit CQ identifier *cq_id* by concatenating the outputs of these two digest functions.

According to the parameter a (grouping factor) of the first digest function, the identifier circle is divided into 2^a contiguous regions. The CQ-to-identifier mapping implements the idea of assigning similar CQs to the same peers by mapping them to a point inside a contiguous region on the identifier circle. As the number of CQs is expected to be larger than the number of peers, the number of CQs mapped inside one of these regions is larger than the number of peers mapped. Introducing smaller regions (i.e., the grouping factor a is larger) increases the probability that two similar CQs are matched to the same peer. This by no means implies that the peers within a contiguous region are assigned only to CQs that are similar for two reasons. First, if the grouping factor a is not large enough, then two non-similar CQs might be mapped into the same contiguous region by the hashing function used (SHA1 in our case). Second, peers might have more than one identifier possibly belonging to different contiguous regions. Given the non-uniform nature of the monitoring requests, there is a trade-off between reducing redundancy in CQ evaluation and balancing load. By setting larger values for the grouping factor a, two extreme situations may occur. On one hand, there may be regions on the identifier circle, in which peers are responsible for too many CQs, and on the other hand, there are some other regions in which peers may be assigned too few CQs and are starving. Thus, the grouping factor a should be chosen carefully to optimize the processing of similar CQs while keeping a good balance of the peer loads. We refer to the grouping provided by the CQ-to-identifier mapping as the *level-one grouping*. A fine tuning of the level-one grouping will be described later in the relaxed matching discussion.

2.3.2.3 Assignment of CQs to Peers: The Two Phase Matching Algorithm

In PeerCQ, the assignments of CQs to peers are based on a matching algorithm defined between CQs and peers, derived from a relationship between CQ identifiers and peer identifiers. The matching algorithm consists of two phases, namely the strict phase and the relaxed phase.

In the *Strict Matching* phase, a simple matching criterion, similar to the one defined in Consistent Hashing [69], is used. A distinct feature of the PeerCQ strict matching algorithm is the two identifier mappings that are carefully-designed to achieve some level of peer-awareness and CQ-awareness, namely the mapping of CQs to CQ identifiers that enables the assignment of CQs having similar triggers to same peers, and the mapping of peers with heterogeneous resource donations to a varying set of peer identifiers. In the *Relaxed*

Matching phase, an extension to strict matching is applied to relax the matching criteria to include application semantics in order to achieve the desired level of peer-awareness and CQ-awareness.

Strict Matching

The idea of strict matching is to assign a CQ to a peer such that the chosen peer has a peer identifier that is numerically closest to the CQ identifier among all peer identifiers on the identifier circle. Formally, strict matching can be defined as follows: The function $strict\_match(cq)$ returns a peer p with identifier j, denoted by a pair (p, j), iff the following condition holds:

$$strict\_match(cq) = (p, j), \text{ where}$$
$$j \in p.peer\_ids \ \wedge \ \forall p' \in P, \forall k \in p'.peer\_ids, Dist(j, cq.cq\_id) \leq Dist(k, cq.cq\_id)$$

Peer p is called the *owner* of the cq. This matching is strict in the sense that it follows the absolute numerical closeness between CQ identifier and peer identifier to determine the destination peer. In other words, suppose two adjacent peer identifiers (together with their peers) (p, i) and (q, j) on the identifier circle such that the CQ identifier lies between i and j. If $Dist(i, cq\_id) \leq Dist(j, cq\_id)$, then the peer p associated with identifier i is the owner peer of this CQ.

Relaxed Matching

The goal of Relaxed Matching is to fine tune the performance of PeerCQ service partitioning by incorporating additional characteristics of the information monitoring applications. Concretely, in the Relaxed Matching phase, the assignments of CQs to peers are revised to take into account factors such as the network proximity of peers to remote data sources, whether the information to be monitored is in the peer's cache, and how peers are currently loaded. By taking into account the network proximity between the peer responsible of executing a CQ and the remote data source being monitored by this CQ, the utilization of the network resources is improved. By considering the current load of peers and whether the information to be monitored is already in the cache, one can further improve the system utilization.

We calculate these three measures for each match made between a CQ and a peer at the strict matching phase. Let p denote a peer and cq denote the CQ assigned to p.

– **Cache affinity factor** is a measure of the availability of a CQ that is being executed at a peer p, which monitors the same data source and the same data item as cq. It is defined as follows:

$$CAF(p.peer\_props.cache, cq.mon\_item) = \begin{cases} 1 & \text{if } cq.mon\_item \text{ is in } p.peer\_props.cache \\ 0 & \text{otherwise} \end{cases}$$

– **Peer load factor** is a measure of a peer p's willingness to accept an additional CQ to execute, considering its current load. The PLF factor provides opportunity for reassigning a CQ to a less loaded peer whenever the executor peer of this CQ needs to be determined. It is defined as follows:

$$PLF(p.peer\_props.load) = \begin{cases} 1 & \text{if } p.peer\_props.load \leq thresh * \text{max\_load} \\ 1 - \frac{p.peer\_props.load}{\text{MAX\_LOAD}} & \text{if } p.peer\_props.load > thresh * \text{max\_load} \end{cases}$$

– **Data source distance factor** is a measure of the network proximity of the peer p to the data source of the CQ specified by identifier cq. SDF is defined as follows:

$$SDF(cq.mon\_src, p.peer\_props.IP) = \frac{1}{ping\_time(cq.mon\_src, p.peer\_props.IP)}$$

Let $UtilityF(p, cq)$ denote the utility function of relaxed matching, which returns a utility value for assigning cq to peer p, calculated based on the three measures given above:

$$UtilityF(p, cq) = PLF(p.peer\_props.load) * \\ (CAF(p.peer\_props.cache, cq.mon\_item) + w * SDF(p.peer\_props.IP, cq.mon\_src))$$

Note that the peer load factor PLF is multiplied with the sum of cache affinity factor CAF and the data source distance factor SDF. This gives more importance to the peer load factor. For instance a peer which has a cache ready for the CQ, and is also very close to the data source will not be selected to execute the CQ if it is heavily loaded. w is used as a constant to adjust the importance of data source distance factor with respect to cache affinity factor. For instance a newly entered peer, which does not have a cache ready for the given CQ but is much closer to the data source being monitored by the CQ, can be assigned to execute the CQ depending on the importance of SDF relative to CAF as adjusted by the w value.

Although the PLF (peer load factor) used in the relaxed matching helps assigning a CQ to a less loaded peer, an increase in the load of a peer after CQs are assigned is not considered by the PLF. However, the PeerCQ design provides easy mechanisms to address such situations. The dynamic changes in the load of a peer can be handled by adjusting the number of peer identifiers. For instance, an increase in the load due to other processes executed by the peer can be handled by decreasing the number of peer identifiers possessed by the peer. This will offload the CQs associated with the dropped peer identifiers to other less loaded peers (as determined by the PLF component of relaxed matching).

Formally, relaxed matching can be defined as follows: The function $relaxed\_match(cq)$ returns a peer p with identifier i, denoted by a pair (p,i) if and only if the following holds:

$$\begin{aligned} relaxed\_match(cq) = (p,i),\ \text{where} \\ (p',j) = strict\_match(cq)\ \wedge\ (p,i) \in NeighborList(p',j) \\ \wedge\ \forall (p'',k) \in NeighborList(p',j), UtilityF(p,cq) \geq UtilityF(p'',cq) \end{aligned}$$

The idea behind the relaxed matching can be summarized as follows: The peer that is matched to a given CQ according to the strict matching, i.e. the owner of the CQ, has the opportunity to query its neighbors to see whether there exists a peer that is better suited to process the CQ in terms of load awareness, cache awareness, and network proximity of the data sources being monitored. In case such a neighbor exists, the owner peer will assign this CQ to one of its neighbors for execution. We call the neighbor node chosen according to the relaxed matching the *executor* of the CQ.

It is interesting to note that the cache-awareness property of the relaxed matching provides additional level of CQ awareness by favoring the selection of a peer as a CQ's executor if the peer has a cache ready for the CQ (which means that one or more similar CQs are already executing at that peer). We refer to the cache-awareness based grouping as *level-two grouping*, which can be seen as an enhancement to the mapping of similar CQs to the same peer in the strict matching phase.

An extreme case of relaxed matching is called the *random relaxed matching*. Random relaxed matching is similar to relaxed matching except that, instead of using a value function to find the best peer to execute a CQ, it makes a random decision among the neighbors of the CQ owner. In the rest of this chapter we call the original relaxed matching *optimized relaxed matching*. Unless otherwise specified, the term relaxed matching and optimized relax matching are used interchangeably.

2.3.3 PeerCQ Service Lookup

The PeerCQ service lookup implements the two-phase matching described in the previous section. Given a CQ, the lookup operation is able to locate its *owner* and *executor* using only $O(\log N)$ messages in a fully decentralized P2P environment, where N is the number of peers. Similar to several existing design of the DHT lookup services [103, 142, 93, 116], the lookup operation in PeerCQ is performed by routing the lookup queries towards their destination peers using routing information maintained at each peer. The routing information consists of a *routing table* and a *neighbor list* for each identifier possessed by a peer. The routing table is used to locate a peer that is more likely to answer the lookup query, where a neighbor list is used to locate the owner peer and the executor peer of the CQ.

Two basic API functions are provided to find peers that are most appropriate to execute a CQ based on the matching algorithms described in the previous section:

$p.lookup(i)$: The *lookup* function takes an m-bit identifier i as its input parameter, and returns a peer -identifier pair (p, j) satisfying the matching criteria used in strict matching, i.e. $j \in p.peer\_ids \;\wedge\; \forall p' \in P, \forall k \in p'.peer\_ids, Dist(j, cq.cq\_id) \leq Dist(k, cq.cq\_id)$.

$p.get\_neighbors(i)$: This function takes an identifier from the peer identifier set of p as a parameter. It returns the neighbor list of $2r + 1$ peers associated with the identifier i of the peer p, i.e., $NeighborList(p, i)$.

The $p.lookup(i)$ function implements a routed query based lookup algorithm. Lookup is performed by recursively forwarding a lookup query containing a CQ identifier to a peer whose peer identifier is closer to the CQ identifier in terms of the strict matching, until it reaches the owner peer of this CQ. A naive way of answering a lookup query is to iterate on the identifier circle using only the neighbor list until the matching is satisfied. The routing tables are used simply to speed up this process. Initialization and maintenance of the routing tables and the neighbor lists do not require any global knowledge. The number of messages used by our lookup operation is logarithmic with respect to the number of peers in the system. More importantly, neither the mappings introduced by PeerCQ nor the implementation of PeerCQ's relaxed matching increases the asymptotic complexity of the number of messages required by the lookup service to carry out CQ to peer matching.

2.3.4 Peer Joins and Departures

In a dynamic P2P network peers can join or depart at any time and peer nodes may fail without notice. A key challenge in implementing these operations is how to preserve the ability of the system to locate every CQ in the network and the ability of the system to balance the load when distributing or re-distributing CQs. To achieve these objectives, PeerCQ needs to preserve the following two principles: (1) Each peer identifier's routing table and neighbor list are correctly maintained. (2) The strict matching and the relaxed matching are preserved for every CQ.

The maintenance of the routing information for DHT based P2P systems in the presence of peer joins and departures is studied in the context of several P2P systems [103, 142, 93, 116]. As a result, in the following subsection we focus on the mechanisms used in PeerCQ to ensure the second principle. Then we extend the discussion to the maintenance of relaxed matching. We defer the discussion on how PeerCQ handles node failures to the next section.

2.3.4.1 Maintaining the Two-Phase Matching of CQs to Peers

It is important to maintain the two-phase matching criteria in order to preserve the ability of the system to sustain the installed CQs, while balancing the peer load in the presence of peer joins, departures, and failures.

Joins, Departures with Strict Matching

Assuming that after a new peer p joins the PeerCQ network, its routing table and neighbor list information are initialized, the subset of CQs that need to transfer their ownership to this newly joined peer p can be calculated as follows: For each identifier $i \in p.peer\_ids$, a set of CQs owned by p's immediate left and right neighbors before p joins the system, is migrated to p if they meet the strict matching criteria. The departure of a peer p requires a similar but reverse action to be taken. Again for each identifier $i \in p.peer\_ids$, p distributes all CQs it owns to immediate left and right neighbors associated with i according to strict matching.

Joins, Departures with Relaxed Matching

For CQs migrated to a new peer p, p becomes the owner of these CQs. By applying the relaxed matching, the executor peer can be located from p's neighbor list. Concretely, each peer keeps two possibly intersecting sets of CQs, namely *Owned CQs* and *Executed CQs*. Owned CQs set is formed by the CQs that are assigned to a peer according to strict matching and the executed CQs set is formed by the CQs that are assigned to a peer according to relaxed matching. CQs in the executed CQs set of a peer are executed by that peer, whereas the CQs in the owned CQs set are kept by the owner peer for control purposes.

A peer p upon entering the system first initializes its owned CQs set as described in the strict matching case. Then it determines where to execute these CQs based on relaxed matching. If peers different than the previous executors are chosen to execute these CQs, then they are migrated from the previous executors to the new executors. Peers whose neighbor lists are affected due to the entrance of the peer p into the system also re-evaluate the relaxed matching phase for their owned CQs, since p's entrance might have caused the violation of relaxed matching for some peers.

The departure process follows a reverse path. A departing peer p distributes its owned CQs to its immediate neighbors in terms of strict matching. Then the neighbors determine which peers to execute these CQs according to the relaxed matching. The departing peer p also returns CQs in its executed CQs set to their owners, and these owner peers find the new executor peers of these CQs according to the relaxed matching.

Concurrent Joins & Departures

Concurrent joins and departures of peers introduces additional challenges both in initializing routing information of newly joined peers, updating routing information of existing peers, and redistributing CQs. More concretely, the problem is how to guarantee concurrent updates of neighbor lists correctly and efficiently as the PeerCQ network evolves. In order to provide consistency in the presence of concurrent joins and departures, in the first prototype of PeerCQ, we enable only one join or one departure operation at a time within a neighbor list. This is achieved by a distributed synchronization algorithm executed within neighbor list boundaries, which serializes the modifications to the neighbor list of each peer identifier. We use a mutual exclusion algorithm [102] to ensure the correctness instead of a weaker solution based on periodic polls to detect and correct inconsistencies as it is done in Chord [116].

2.3.5 Handling Node Failures with Dynamic Replication

It is known that failures are unavoidable in a dynamic peer-to-peer network where peer nodes correspond to user machines. A failure in PeerCQ is a disconnection of a peer from the PeerCQ network without notifying the system. This can happen due to a network problem, computer crash or improper program termination. Byzantine failures that include malicious program behavior are not considered. We assume a fail-stop model where timeouts can be used for detecting failures. In PeerCQ, failures are detected through periodic pollings between peers in a neighbor list.

Failures threaten the system reliability in two aspects. First, a failure may result in incorrect routing information. Second, a failure of a peer will cause CQ losses if no additional mechanisms are employed. The former problem is solved using routing maintenance mechanisms similar to those in handling peer departure. The only difference is that the detection of a failure triggers the maintenance of the routing information instead of a volunteer disconnection notification. However the latter problem requires a more involved solution.

There are two important considerations in PeerCQ regarding providing fault-tolerant reliable service. One is to *provide CQ durability* and the other is to *provide uninterrupted CQ processing*. CQ durability refers to the ability of PeerCQ to maintain the property that no

CQs executed at a peer will get lost when it departs or fails unexpectedly. Uninterrupted CQ processing refers to the ability of PeerCQ to pick up those CQs dropped due to the departure or failure of an existing peer and continue their processing. Whenever CQ durability is violated, the uninterrupted CQ processing will be violated too. Furthermore, when a peer p fails, if there are existing peers that hold additional replicas of the CQs that p runs as their executor, but do not have sufficient information on the execution state of those CQs, then the execution of these CQs will be interrupted, possibly resulting in some inconsistent behavior. The proper resumption of the CQ execution upon the failure of its executor peer requires the replicas to hold both the CQs and their runtime state information.

2.3.5.1 PeerCQ Replication Scheme

In order to ensure smooth CQ execution and to prevent failures interrupting CQ processing and threatening CQ durability, we need to replicate each CQ. We describe PeerCQ replication formally as follows: A CQ, denoted as cq, is replicated at the peers contained in the following set:

$$\begin{aligned} \mathit{ReplicationList}(cq) = [(p_{-\lfloor rf/2 \rfloor}, i_{-\lfloor rf/2 \rfloor}), \ldots, (p_{-1}, i_{-1}), (p_0, i_0), (p_1, i_1), \ldots, (p_{\lceil rf/2 \rceil}, i_{\lceil rf/2 \rceil})], \\ \text{where} \bigwedge_{k=1}^{\lceil rf/2 \rceil} p_{i_k} = IRN(p_{k-1}, i_{k-1}) \ \wedge \bigwedge_{k=1}^{\lfloor rf/2 \rfloor} p_{i_{-k}} = ILN(p_{-k+1}, i_{-k+1}) \\ \wedge \ (p_0, i_0) = \mathit{strict\_match}(cq) \end{aligned}$$

This set is called the *replication list*, and is denoted as $\mathit{ReplicationList}(cq)$. Size of the replication list is $rf + 1$, where rf is called the *replication factor*. Replication list size should be smaller than or equal to the neighbor list size to maintain the property that replication is a localized operation, i.e. $\mathit{ReplicationList}(cq) \subset \mathit{NeighborList}(p, i)$, where $(p, i) = \mathit{strict\_match}(cq)$.

In addition to replicating a CQ, some execution states of the CQ needs to be replicated together with the CQ and be updated when the CQ is executed and its execution state changes, in order to enable correct continuation of the CQ execution after a failure. Recall Section 2.2, the executor peer of a CQ needs to maintain three execution states about this CQ: (1) the evaluation state of the trigger, which contains monitoring source, monitoring item and trigger condition evaluation result; (2) the query result returned since last CQ evaluation, and (3) the notification state of the CQ. In PeerCQ, changes on the states associated with each CQ are propagated to replicas of the CQ in two steps with either an eager mode or a deferred mode: (1) Whenever an executor peer of a CQ updates the related states of a CQ, it notifies the CQ owner immediately to ensure that such updates be propagated to all replicas of the CQ. (2) Upon receiving update notification from the executor peer of a CQ, the owner peer of the CQ may choose to send update notifications to all other peers holding replicas of the CQ immediately (eager mode) or to propagate the update to rest of the replicas using a deferred strategy that considers different tradeoffs between performance and reliability. The propagation of state update to the owner ensures that at least two peers hold the update state of each CQ and the probability of both executor peer and owner peer fail together is relatively low.

Since CQs should be available for processing at anytime once they are installed in the system, PeerCQ requires a strong and dynamic replication mechanism. By strong and dynamic replication we mean that at any time each CQ should have certain number of replicas available in the system, and this property should be maintained dynamically as the peers enter and exit the system. As a result, our replication consists of two phases. In Phase one, a CQ is replicated at a certain number of peers. This phase happens immediately after a CQ is installed into the system. In Phase two, the number of replicas existing in the system is kept constant, and all replicas are kept consistent. The second phase is called the *replica management* phase and lasts until the CQ's termination condition is met or the CQ is explicitly removed from the system.

One important decision for PeerCQ replication scheme is where to replicate CQs. In order to preserve the correctness of the lookup mechanism, and preserve good load-balance we select the peers to host the replicas of a CQ from the peers in the neighbor list of the owner peer of this CQ. Moreover, choosing these peers from the neighbor list localizes the replication process (no search is required for locating replica holders), which is an advantage in a fully-decentralized system. Furthermore, peers that are neighbors on the identifier circle are not necessarily close to each other geographically, thus the probability of collective failures is low.

2.3.5.2 Fault Tolerance

Given the description of the PeerCQ replication scheme, we define two different kinds of events that result in loosing CQs. One is the case where the existing peers that are present in the system are not able to hold (either for replication or for execution) any more CQs due to their heavy load. There is nothing to be done for this if the system is balanced in terms of peer loads. Because this indicates insufficient number of peers present in the system. The other case is when all replica holders of a CQ (or CQs) fail in a short time interval, not letting the dynamic replica management algorithm to finish its execution. We call this time interval the *recovery time*, denoted by Δt_r. We call the event of having all peers contained in a replication list fail within the interval Δt_r, a *deadly failure*. We first analyze the cases where we have deadly failures and then give an approximation for the probability of having a deadly failure due to a peer's departure. We assume that peers depart by failing with probability pf and the time each peer stays in the network, called the *service time*, is exponentially distributed with mean st.

Let us denote the CQs owned by a peer p that satisfies $strict\_match(cq) = (p, i)$ as $O_{p,i}$. Let $RL_{p,i}(t)$ be the set of peers in the replication list of CQs in $O_{p,i}$ at time t, where replication list is a subset of the neighbor list and has size $rf+1$. rf is named as *replication factor*. Assume that the peer p fails right after time t_a. Then $RL_{p,i}(t_a)$ consists of the peers that are assumed to be holding replicas of CQs in $O_{p,i}$ at time t_a. Let us denote the time of the latest peer failure in $RL_{p,i}(t_a)$ as t_l and the length of the shortest time interval which covers the failure of peers in $RL_{p,i}(t_a)$ as Δt where $\Delta t = t_l - t_a$. If Δt is not large enough, i.e. $\Delta t < \Delta t_r$, then p's failure at time t_a together with the failures of other peers in $RL_{p,i}(t_a)$ will cause a deadly failure. This will result in loosing some or all CQs in $O_{p,i}$.

Let $Pr_{df}(p)$ denote the probability of a peer p's departure to result in a deadly failure. Then we define: $Pr_{df}(p,i) = Pr\{$All peers in $RL_{p,i}(t)$ has failed within a time interval $<$ Δt_r, where p failed at time $t\}$. Then we have:

$$Pr_{df}(p) = 1 - \prod_{i \in p.peer\_ids} (1 - Pr_{df}(p,i))$$

If we assume $\bigcap_{i \in p.peer\_ids} RL_{p,i}(t) = p$, then $\forall_{i,j \in p.peer\_ids}\ Pr_{df}(p,i) = Pr_{df}(p,j)$. Then we have:

$$Pr_{df}(p) = 1 - (1 - Pr_{df}(p,i))^{p.ident\_count} \tag{1}$$

Let t_0 denote a time instance at which all peers in $RL_{p,i}(t)$ was alive. Furthermore let us denote the amount of time each peer in $RL_{p,i}(t)$ stayed in the network since t_0 as random variables $A_1, \ldots, A_{rf+1}$. Due to the memorylessness property of the exponential distribution, $A_1, \ldots, A_{rf+1}$ are still exponentially distributed with $\lambda = 1/st$. Then, we have $Pr_{df}(p,i) = pf^{rf+1} * Pr\{MAX(A_1, \ldots, A_{rf}) < \Delta t_r\}$, which leads to:

$$\begin{aligned} Pr_{df}(p,i) &= pf^{rf+1} * \prod_{i=1}^{rf} Pr\{A_i < \Delta t_r\} \\ Pr_{df}(p,i) &= pf^{rf+1} * \prod_{i=1}^{rf} (1 - e^{-\Delta t_r / st}) \end{aligned} \tag{2}$$

Equations 1 and 2 are combined to give the following equation:

$$Pr_{df}(p) = 1 - \Big(1 - pf^{rf+1} * \prod_{i=1}^{rf} (1 - e^{-\Delta t_r / st})\Big)^{p.ident\_count}$$

In a setup where $rf = 4$, $pf = 0.1$, $\Delta t_r = 30$secs and $st = 60$mins, $p.identifier\_count = 5$, $Pr_{df}(p)$ turns out to be $\simeq 2.37 * 10^{-13}$. We further investigate the fault tolerance capability of PeerCQ in Section 2.4.4. Note that the greater the replication factor rf is, the lower the probability of loosing CQs. However having a greater replication factor increases the cost of managing the replicas. As described earlier, the job of dealing with replica management of a CQ is the responsibility of the CQ's owner.

2.4 Simulation-based Experiments and Results

To evaluate the effectiveness of PeerCQ's service partitioning scheme with respect to system utilization and load balancing, we have designed a series of experiments. Here we first describe our experimental setup.

We built a simulator that assigns CQs to peers using the service partitioning and lookup algorithms described in the previous sections. The system parameters to be set in the simulator include: m, length of identifiers in bits; a, grouping factor; r, neighbor list parameter;

N, number of peers; K, number of CQs. With this simulator, we conduct our experiments under different stabilization states of the system as well as under unstable states. The system states were modeled with different numbers of peers, different workloads of CQs, and different configurations of some system parameters. The measurements were taken on these snapshots. In all experiments reported in this chapter, the length of the identifiers (m) is set to 128.

We model each peer with its resources, the amount of donation, the reliability factor, and its IP address. The resource distribution is taken as normal distribution. The donations of peers are set to be a half of their resources. We model CQs with the data sources, the data items of interest, and the update thresholds being monitored. There are $D = 5 * 10^3$ data sources and 10 data items on each data source. The distribution of the user interests on the data sources is selected to model the hot spots that arise in real-world situations due to the popularity of some triggers. Both normal distribution for modeling the user interests on the data sources and a zipf distribution (Section 2.4.2.4) are considered in the experiments.

2.4.1 Effect of Grouping Factor

An important factor that may affect the effectiveness of the service partitioning scheme is the grouping factor. Recall Section 2.3.2.2, the grouping factor a is introduced at the protocol level to promote the idea of grouping similar CQs to optimize the processing of similar information monitoring requests. The grouping factor a is designed to tune the probability of assigning similar CQs to the same peer. The larger the a value is, the higher the probability that two similar CQs will be mapped to the same peer, and thus the fewer number of CQ groups per peer. However, increasing a has limitations as discussed in Section 2.3.2.2.

This experiment considers a 10,000 node network ($N = 10^4$), and the total number of CQs in the network is 100 times of N, i.e., $K = 10^6$. Figure 3 shows the effects of increasing a on grouping when a is $0, 8$, and 10. Figures 4 shows the effects of increasing a on grouping when a is $12, 14$, and 16. The values on the x-axis are the number of CQ groups that the

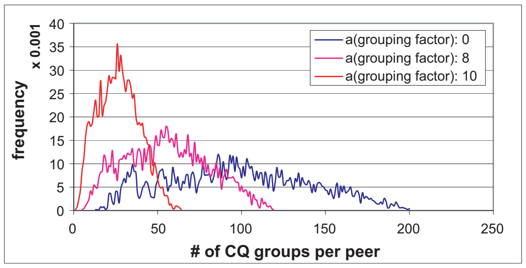


Figure 3: Effect of grouping for $a = 0, a = 8$, and $a = 10$

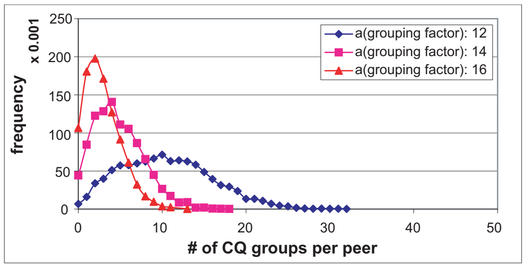


Figure 4: Effect of grouping for $a = 12, a = 14$, and $a = 16$

peers have and the corresponding values on the y-axis are the frequencies of peers having x number of CQ groups. From these two figures one can observe that when a is set to 0, there is nearly no grouping since the average CQ group size is close to one and the number of CQ groups is large (one CQ per group, and the number of CQs a peer is responsible for may reach up to 200). The number of groups decreases as a is set to a larger value. The average size of the CQ groups also increases as a is set to a larger value. Consider the simulation results from Figure 3 and Figure 4, when $a = 8$ the largest number of groups a peer may have is decreased to 120 or so. When $a = 10$ the largest number of groups a peer may have is dropped to less than 70. When the grouping factor a is set to be 16, the largest number of groups a peer has is less than 20. Small number of CQ groups implies larger sizes of the CQ groups.

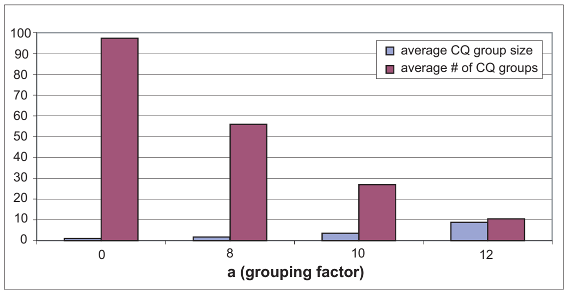


Figure 5: Influence of a on avg. CQ group sizes and avg. # of CQ groups

Figure 5 compares the average group size and the average number of groups per peer. The values on the x-axis of Figure 5 are the grouping factors, where the two series represent average CQ group size (average number of CQs per CQ group) and average number of CQ groups per peer respectively. When $a = 0$, there is nearly no grouping since the average CQ group size is close to one and the number of CQ groups is large (one CQ per group). As the grouping factor increases, the average size of the CQ groups also increases, while the number of CQ groups decreases.

These observations have an important implication. Assignment of CQs to peers that try to achieve better grouping (setting the grouping factor a to be higher) will *decrease* the number of CQ groups processed by a peer, while *increasing* the number of CQs contained in each CQ group (CQ group size). As a result, the average load of peers will be decreased and the overall system utilization will be better. However, increasing the grouping factor too much causes a lot of peers getting no CQs!

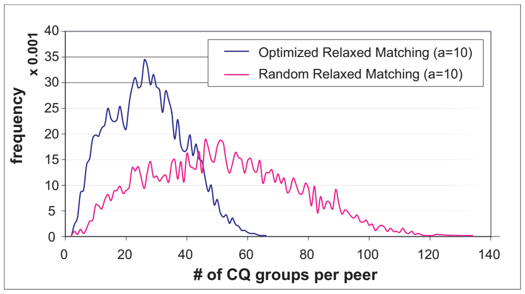


Figure 6: Optimized relaxed matching compared to strict matching

To provide an in-depth understanding of the effect of grouping factor, we compare the *optimized relaxed matching* algorithm with the *random relaxed matching* algorithm under a given grouping factor. Figure 6 compares the two matching algorithms when $a = 10$. It is clear that the optimized relaxed matching is more effective in its ability to group CQs, which is due to its cache-awareness. We can say that random relaxed matching has only *level-one grouping* which is the grouping provided by the grouping factor, where the optimized relaxed matching algorithm also has *level-two grouping* supported through its cache-awareness.

2.4.2 Effectiveness with respect to Load Balancing and System Utilization

This section presents a set of experiments to evaluate the effectiveness of the PeerCQ service partitioning scheme with respect to load balance and system utilization. By better system utilization, we mean that the system can achieve higher throughput and lower overall consumption of resources in terms of processing power and network bandwidth. By load balancing, we mean that no peer in the system is overloaded due to joins or departure of other peers or due to the increase of requests to monitoring data sources that are hot spots at times.

2.4.2.1 Load Balance v.s. System Utilization

It is interesting to point out that by incorporating the grouping optimization into PeerCQ, we observe that the goal of balancing the load over peers may not always be consistent with the goal of maximizing the overall system utilization in PeerCQ.

To illustrate this observation, consider a simple example: Assume we have two peers, p and p', which are identical in terms of their capacities. Assume that there are seven CQs that need to be distributed to these two peers. One CQ of type a denoted as cqa_1 and six CQs of type b denoted as $cqb_1,\ldots,cqb_6$. Furthermore, assume that CQs of the same type are similar and thus can be grouped together. The scenario that shows a better *overall* utilization of system resources is the case where p is assigned only one CQ which is cqa_1 and p' is assigned six CQs, namely $cqb_1,\ldots,cqb_6$. By using the full power of CQ grouping, one can minimize the repeated computation and duplicated consumption of network resources. However, the optimal system utilization may not necessarily imply a good balance of loads on peers. Even though the most expensive computation of the CQ processing is the continued testing of CQ triggers, our experience with the continual query systems show that the cost of grouping, querying, and notification is not negligible [78]. Therefore, it is likely that the scenario where the system is best utilized may not be the same as the scenario where the load of the system is best balanced among peers.

2.4.2.2 CQ Load of a Peer

We define the CQ load of a peer to be the number of CQs processed by the peer divided by the number of identifiers it has. Due to the support of grouping in the CQ processing, the CQ load of a peer should not be used as a measure to compare peer loads. To illustrate this observation, consider a case where a peer, say p, is assigned 10 CQs and another peer, say p', which has twice the number of peer identifiers that p has, is assigned 20 CQs. Note that their CQ loads are equal. Furthermore, assume that the 10 CQs assigned to p are partitioned into five CQ groups of sizes 4, 2, 2, 1, 1; and the 20 CQs assigned to p' are partitioned into two CQ groups of sizes 12, 8. In this case it is quite possible that the peer p' is loaded less than peer p although their CQ loads are equal. This due to the fact that, the number of CQ groups in p' (which is 2) is smaller than the number of CQ groups in p (which is 5), although their CQ loads are equal.

2.4.2.3 Computing Peer Load

In order to analyze the load on peers we first formalize the notion of load on a peer. In PeerCQ, the cost associated with the P2P protocol level processing is considered to be proportional to peer capacities, since the protocol level processing is proportional to the number of identifiers a peer has. Based on this understanding, we consider the continued monitoring of remote data sources and data items of interest to be the dominating factor in computing the peer load. We formalize the load on a peer p as follows[1]:

Let G_p represent the set of groups that peer p has, denoted by a vector $\langle g_1, \ldots, g_n \rangle$, where n is the number of CQ groups that peer p has. We refer to n as the size of G_p denoted by $size(G_p)$. Each element g_i represents a group in p, which can be identified by the data source being monitored and the data items of interest. The size of a group g_i, which is the number of CQs it contains, is denoted by $size(g_i)$. Let $cost(g_i)$ be the cost of processing all CQs in a group g_i, $monCost(g_i)$ be the cost of monitoring a data item, and $gCost(size(g_i))$ be the cost of grouping for group g_i, which is dependent on the number of CQs in g_i. Then the cost of processing all CQs in a peer, denoted as $cost(G_p)$, can be calculated as follows:

[1]For the purpose of simulation, it is assumed that the frequency of changes is the same for all data items.

$cost(G_p) = \sum_{i=1}^{size(G_p)} cost(g_i) = \sum_{i=1}^{size(G_p)} (monCost(g_i) + gCost(size(g_i))$.

In our experiments, we assume that the cost of grouping increases linearly with group size. In particular if processing one CQ costs 1 unit, then processing k similar CQs costs $1 + x * k$ units, where $x * k$ corresponds to the cost of grouping k CQs. x is taken as 0.25 in our simulations. This setting was based on the grouping effect and cost study we have done on WebCQ [81].

Given that the cost of detecting changes on the data items of interest from remote data sources is the dominating factor in the overall cost of processing a CQ, we assume that the cost of monitoring is the same for all data items independent of the monitoring conditions defined by CQs, and is equal to *monCost*, then the cost of processing all CQs on a peer p can be reduced to: $cost(G_p) = size(G_p) * monCost + \sum_{i=1}^{size(G_p)} gCost(size(g_i))$.

In order to calculate the load on a peer, the cost is normalized via dividing it by the effective donation. Because, the notion of load on a peer in our system is relative to the effective donation of the peer. Let ED_p be the effective donation of peer p. We calculate the load on a peer as: $load(p) = cost(G_p)/ED_p$.

The load values of peers are used as both a measure of system utilization and a measure of load balance in our experiments. First, the *mean peer load*, which is the average of peer load values, is used as a measure of system utilization. The smaller the mean load is, the better the system utilization is. However, the system utilization is also influenced by the amount of network bandwidth consumed, which is captured by the *average network cost* defined below. Second, the *variation in peer loads* is used as a measure of load balance. To compare different scenarios, the load variance is normalized by dividing it by the mean load. This measure is called the *balance in peer loads.* Small values of balance in peer loads imply a better load balance.

PeerCQ service partitioning makes use of network proximity between peers and data sources when assigning CQs to peers. It aims at decreasing the network cost of transferring data items from the data sources to the peers of the system. For simulation purpose, we assign a cost to each (peer, data source) pair in the range [10,1000]. We model such a cost by the ping times between peers and data sources. Then we calculate the sum of these costs for each CQ group at each peer and divide it by the total number of peers to get an average. Let P denote the network consisting of N peers, and *net_cost* be the function that assigns costs to (peer, data source) pairs, then the resulting value named as *average network cost* and denoted by *avgNetCost*, is equal to: $avgNetCost = \frac{1}{N} \sum_{p \in P} \sum_{i=1}^{size(G_p)} net\_cost(p, g_i.mon\_src)$

2.4.2.4 Experimental Results

All experiments in this section were conducted over a network consisting of N peers and K CQs, where $N = 10^4$ and $K = 10^6$. To evaluate the effectiveness of the optimized relaxed matching algorithm, we compare it with the random relaxed matching algorithm using the set of parameters discussed earlier, including the grouping factor a, the mean peer load, the variance in peer loads, the balance in peer loads, the average network cost, and the variance in CQ loads of peers.

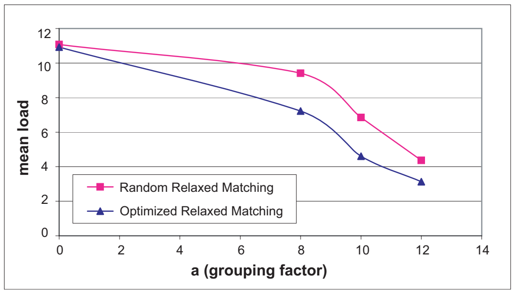


Figure 7: Effect of a and relaxed matching on mean peer load

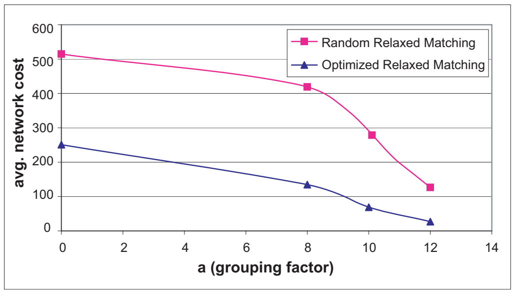


Figure 8: Effect of a and relaxed matching on average network cost

Figure 7 shows the effect of the grouping factor a on the effectiveness of relaxed matching with respect to mean load. Similarly, Figure 8 shows the effect of the grouping factor a on the effectiveness of relaxed matching with respect to network cost. From Figures 7 and 8, we observe a number of interesting facts:

First, as the grouping factor increases, both the mean peer load and the average network cost decreases. Increasing the grouping factor helps in decreasing the mean peer load, since it reduces the redundant computation by enabling more group processing. Optimized relaxed matching provides more effective reduction in the mean peer load due to its level-two grouping. Level-two grouping works better as the grouping factor a increases (i.e., the level-one grouping increases).

Second, increasing the grouping factor also helps in decreasing the average network cost, since the cost of fetching data items of interest from remote data sources incurs only once per CQ group, and served for all CQs within the group. It is also clear that optimized relaxed matching provides more effective reduction in their average network cost, due to its level-two grouping (which results in better grouping) and its data source awareness, which incorporates the network cost of accessing the data items of a CQ that are being monitored into the service partitioning decision.

Third but not least, the decrease in the mean peer load and in the average network cost is desirable, since it is an implication of better system utilization. However if the grouping factor increases too much, then the goal of load balancing over the peers of the system will suffer.

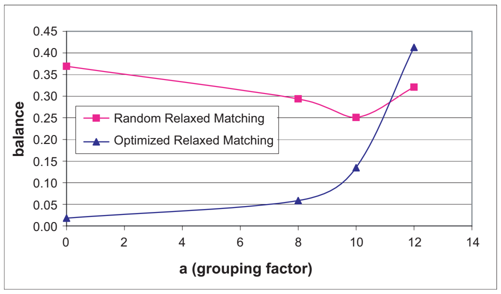


Figure 9: Effect of a and relaxed matching on load balance

Figure 9 shows the effect of increasing the grouping factor a on load balance of both the optimized relaxed matching algorithm, and the random relaxed matching algorithm. As expected, the optimized relaxed matching provides better load balance, since optimized relaxed matching explicitly considers peer loads in its value function for determining the peer that is appropriate for executing a CQ. In the case of $a = 0$ it provides the best load balance. However, as the grouping increases, peers having identifiers belonging to some hot spotted regions of the identifier space are matched much more CQs than others (due to the non-uniform nature of information monitoring interests and the mechanisms used to match CQs to peers). Consequently, the load balance gets worse as the grouping increases. For our experiment setup, the load balance degrades quickly when a is 8 or higher.

It is interesting to note that random relaxed matching shows an improvement in load balance for smaller values of the grouping factor and start switching to a degradation trend when a is set to 10 or higher. This is mainly due to the fact that random relaxed matching only relies on randomized algorithms to achieve load balance in the system. Thus the load balance obtained in the case of $a = 0$ is inferior when compared to optimized relaxed matching. This means that there are overloaded and under-loaded peers in the system. Grouping helps decreasing the loads of over-loaded peers by enabling group processing. This effect decreases the gap between overloaded peers and under-loaded peers, resulting in a better balance to some extent.

Finally, it is important to note that, when we increase a too much, the optimized relaxed matching loses its advantage in terms of load balancing over the random relaxed matching. Intuitively this happens due to the fact that in optimized random relaxed matching there are two levels of grouping, whereas in random relaxed matching there is only one level of grouping. More concretely, in overloaded regions of the identifier space, there is nothing to balance. In under-loaded regions, when a increases, the optimized relaxed matching maps more CQs to fewer peers due to the second-level grouping, causing even more unbalance since several peers get no CQs at all from the under loaded region.

In summary, to provide a reasonable balance between overall system utilization and load balance, it is advisable to choose a value for a, which is equal to or smaller than the value where the randomized relaxed matching changes its load balance trend to degradation, but is greater than half of this value. This results in the range [6, 10] in our setup. In this range, higher values are better for favoring overall system utilization, whereas lower values

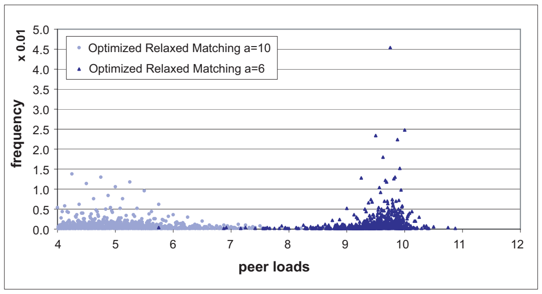


Figure 10: Effect of a and relaxed matching on load distribution

are better for favoring load balance. Figure 10 shows this trade-off. The values on the x-axis are the peer load values, and the corresponding values on the y-axis are the frequencies of peers having x amount of load. By looking at the points it is easy to see that the balance is better when $a = 6$ and load values are lower when $a = 10$.

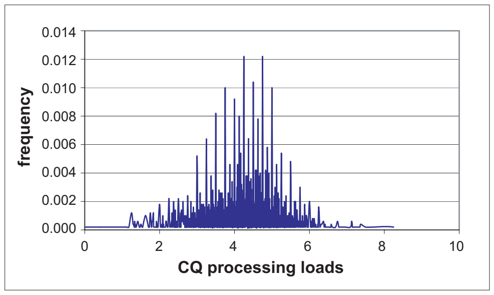


Figure 11: Load distributions, for normally distributed CQ interest

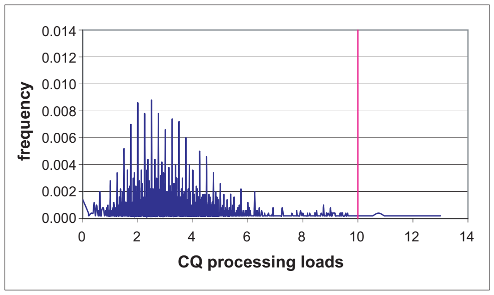


Figure 12: Load distributions, for zipf distributed CQ interest

Figure 11 shows the distribution of the CQ processing loads over peers. Figure 12 plots the same graph except that the information monitoring interests of CQs that are used to generate the graph follow a zipf distribution which is more skewed than the normal distribution used in Figure 11. The vertical line in Figure 12 which crosses the x-axis at 10 marks the maximum acceptable load, thus the region on the right of the vertical line represent overloaded peers. Comparing these two figures show that more skewed distributions in information monitoring interests reduce the balance in CQ processing loads.

2.4.3 Effect of Relaxed Matching Criteria

The relaxed matching criteria, which is characterized by the utility function used in selecting CQ executors, has influence on several performance measures. In this section we examine the effect of each individual component of the utility function on some of these measures. The experiment is set up over a network of 10^4 nodes with 10^6 CQs. and the grouping factor a is set to be 8.

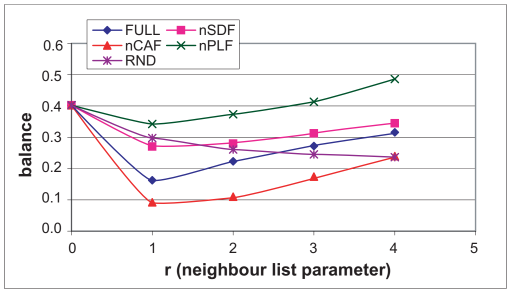


Figure 13: Balance in loads for different utility functions

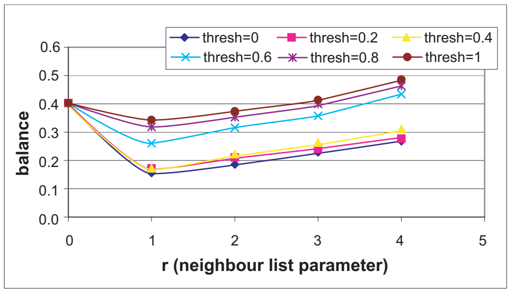


Figure 14: Balance in loads for different threshold values in PLF

Figure 13 shows the effect of individual utility function components on the balance in CQ processing loads as a function of neighbour list size, r. The line labeled as FULL corresponds to the unmodified utility function. Lines labeled as nX correspond to utility functions in which the component X is taken out ($X \in \{PLF, CAF, SDF\}$). The line labeled as RND corresponds to a special utility function which produces uniformly random values in the range [0,1] resulting in randomized relaxed matching. The first observation from Figure 13 is that in all cases except RND, the balance shows an initial improvement with increasing r which is replaced by a degradation for larger values of r. For RND the balance continuously but slowly improves with r. The degradation in balance is due to excessive grouping. When r is large, there is more opportunity for grouping and excessive grouping leads to less balanced CQ processing loads. Figure 13 clearly shows that PLF is the most important factor in achieving a good load balance. Since PLF is the most influential factor in achieving good load balance, a lower *thresh* value used in PLF factor increases its impact thus slows down the r related degradation in the balance. This is shown in Figure 14. Figure 13 also shows that CAF is responsible for the degradation of balance with increasing r values. However CAF is an important factor for decreasing the mean CQ processing load of a peer by providing grouping of similar CQs. Although RND provides a better load balance than FULL for $r \geq 3$, the mean CQ processing load of a peer is not decreasing with increasing r when RND is used as opposed to the case where FULL is used. The latter effect is shown in Figure 15.

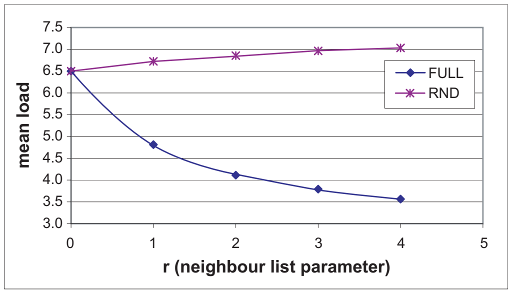


Figure 15: Mean CQ processing load for different utility functions

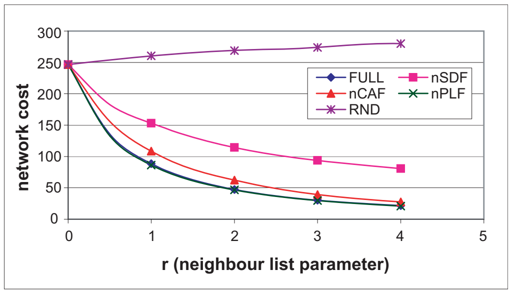


Figure 16: Network cost as a function of r for different utility functions

Figure 16 shows the effect of individual utility function components on the network cost due to CQ executions. The increasing r values provide increased opportunity to minimize this cost due to larger number of peers available for selecting an executor peer with respect to a CQ. Since SDF is explicitly designed to decrease the network cost, its removal from the utility function causes increase in the network cost. Figure 16 shows that CAF also helps decreasing the network cost. This is because it provides grouping which avoids redundant fetching of the data items.

2.4.4 CQ Availability under Peer Failure

One situation that is crucial for the PeerCQ system is the case where peers are continuously leaving the system without any peers entering; or the peer entrance rate is too low when compared to the peer departure rate, so that the number of peers present in the system decreases rapidly. Although we do not expect this kind of trend to continue for a long period, it can happen temporarily. In order to observe the worst case, we have setup our simulation so that the system starts with $2 * 10^4$ peers and 10^6 CQs and each peer departs the system by failing after certain amount of time. The time each peer stays in the system is taken as exponentially distributed with mean equal to 30 mins, i.e. $st = 30$ mins. It is clear that in such a scenario the system will die loosing all CQs, since all peers will depart eventually. However, we want to observe the behavior with different rf values under a worst case scenario to see how gracefully the system degrades for different replication factors.

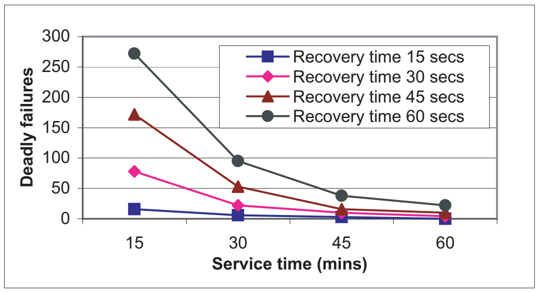


Figure 17: Deadly failures $rf = 2$

The graphs in Figures 17, 18 and 19 plot the total number of deadly failures that have occurred during the whole simulation for different mean service times (st), recovery times (Δt_r), and replication factors (rf). These graphs show that the number of deadly failures is smaller when the replication factor is larger, the recovery time is smaller and the mean service time is longer. Note that our simulation represents a worse scenario, where every

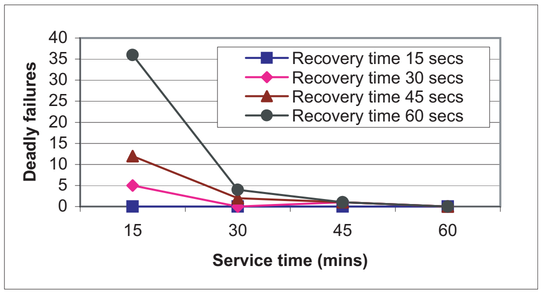


Figure 18: Deadly failures $rf = 3$

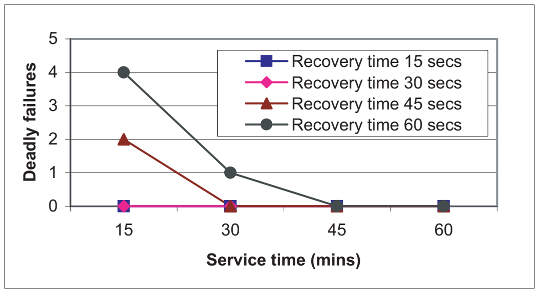


Figure 19: Deadly failures $rf = 4$

peer leaves the system by a failure and no peer enters into the system. However, a replication factor of 4 presents a very small number of or even no deadly failures.

These experiments show that the dynamic replication provided by PeerCQ is able to achieve high reliability with moderate values for the replication factor. Although stronger reliability guarantees can be achieved through increasing the replication factor further, it has the side-effect of increasing the cost of replication. Here we provide a sample experimental result pertaining to cost of replication, in order to give a general idea of the trade-offs involved in deciding an appropriate value for the replication factor.

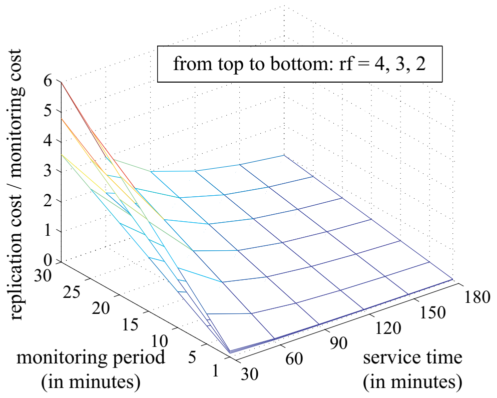


Figure 20: Cost of replication relative to monitoring cost

Figure 20 plots the network cost of replication relative to the network cost of monitoring as a function of mean peer service time and CQ monitoring period. Mean peer service time is the average time a peer stays in the network before it fails or departs. Monitoring period is the time between two successive pollings of the data sources for monitoring changes on data items. It is observed from the figure that the cost of replication is small compared to the monitoring cost when the mean service time values are high or when the monitoring period values are low. It is also observed that smaller replication list sizes (i.e. smaller rfs) help in decreasing the relative cost of replication. For applications with tighter latency requirements the monitoring period should be small and the replication is less likely to be an issue in terms of network cost. On the other hand, for applications specifying larger monitoring periods, the cost of replication can be adjusted by changing the rf values and adjusting the desired level of reliability.

2.5 Related Work

WebCQ [80] is a system for large-scale web information monitoring and delivery. It makes heavy use of the structure present in hypertext and the concept of continual queries. It is a server-based system, which monitors and tracks various types of changes to static and dynamic web pages. It includes a proxy cache service in order to reduce communication with the original information servers. PeerCQ is similar to WebCQ in terms of functionality but differs significantly in terms of the system architecture, the cost of administration, and the technical algorithms used to scheduling CQs. PeerCQ presents a peer-to-peer architecture for large scale information monitoring, which is more scalable and less expensive to maintain due to the total decentralization and the self-configuring capability.

Scribe [104] is an P2P application that is related to event monitoring and notification. It presents a publish/subscribe based P2P event notification infrastructure. Scribe uses Pastry [103] as its underlying peer-to-peer protocol and builds application level multicast trees to notify subscribers from events published in their subscribed topic. Pastry's location algorithm is used to find rendezvous points for managing the group communication needed for a topic. It uses topic identifiers to map topics to peers of the system. However, Scribe is a topic based event notification system, where PeerCQ is a generic information monitoring and event notification system. In PeerCQ notifications are generated based on the monitoring done on the web using the supplied CQs that encapsulate the interested information update requests. In Scribe notifications are generated from publish events of the topic subscribers.

Several P2P protocols have been proposed to date, among which the most representative ones are CAN [101], Chord [116], Tapestry [142], and Pastry [103]. Similar to these existing DHT based systems, the PeerCQ P2P protocol described in this chapter is developed by extending the Plaxton routing proposal in [93]. The unique features of PeerCQ are its ability to incorporate peer-awareness and CQ-awareness into the service partition scheme and its ability to achieve a good balance between load balance and overall system utilization. The peer awareness in PeerCQ is supported by virtual peer identifiers. Although our solution was developed independently [45], the concept of virtual servers in [100] also promote the use of varying number of peer identifies for the purpose of load balancing. However, load balancing in PeerCQ has a unique characteristics. Its use of virtual peer identifies is combined with CQ grouping and relaxed matching techniques, making the PeerCQ service partitioning scheme unique and more scalable in handling hot spot monitoring requests. Finally, our dynamic passive replication scheme share some similarity to the replication techniques used in CFS [37] with a number of major differences. First, in PeerCQ there is a need for updating the state of the replicas as the CQs execute and change state. Thus the replication scheme needs to maintain strong consistency among replicas. Second, PeerCQ provides dynamic relocation of CQ executors as better peers for executing them join the system to maintain the relaxing matching dynamically.

CHAPTER III

MOBIEYES - DISTRIBUTED LOCATION MONITORING IN MOBILE SYSTEMS USING CQS

With the growing popularity and availability of mobile communications, our ability to stay connected while on the move is becoming a reality instead of science fiction just a decade ago. An important research challenge for modern location-based services is the scalable processing of location monitoring requests on a large collection of mobile objects. The centralized architecture, though studied extensively in literature, would create intolerable performance problems as the number of mobile objects grows significantly. This chapter presents a distributed architecture and a suite of optimization techniques for scalable processing of continuously moving location queries. Moving location queries can be viewed as standing location tracking requests that continuously monitor the locations of mobile objects of interest and return a subset of mobile objects when certain conditions are met. We describe the design of MobiEyes, a distributed real time location monitoring system in a mobile environment. The main idea behind the MobiEyes' distributed architecture is to promote a careful partition of a real time location monitoring task into an optimal coordination of server-side processing and client-side processing. Such a partition allows evaluating moving location queries with a high degree of precision using a small number of location updates, thus providing highly scalable location monitoring services. A set of optimization techniques are used to limit the amount of computation to be handled by the mobile objects and enhance the overall performance and system utilization of MobiEyes. Important metrics to validate the proposed architecture and optimizations include messaging cost, server load, and amount of computation at individual mobile objects. We evaluate the scalability of the MobiEyes location monitoring approach using a simulation model based on a mobile setup. Our experimental results show that MobiEyes can lead to significant savings in terms of server load and messaging cost, compared to solutions relying on central processing only.

3.1 Introduction

With the growing availability of mobile communications, and the rapid drop in prices for basic mobility enabling equipments like GPS devices [127], smart cell phones, and handhelds, we are entering a world where people, computers, vehicles, and other mobile objects are interconnected, and traditional wired networks are being replaced by their wireless counterparts, which facilitates our ability to stay connected while on the move.

There are two representative types of emerging location-based services: location-aware content delivery and location-sensitive resource management. The former uses location data to tailor the information delivered to the mobile users in order to increase the quality of service and the degree of personalization. Examples include delivering accurate driving directions, instant coupons to customers nearby or approaching a store, or answering location-based queries for nearest resource information like local restaurants, hospitals, gas stations, or police cars within 5 miles upon a car accident. The latter uses location data combined with route schedules and resource management plans to direct service personnel or transportation systems, optimize personnel utilization, handle emergency requests, and reschedule in response to external conditions like traffic and weather. Examples include systems for fleet management, mobile workforce management, and transportation management. Scalable location query processing is an enabling technology for all these applications [130, 129].

An important research challenge for location information management and future mobile computing applications is a scalable architecture that is capable of handling large and rapidly growing number of mobile objects and processing complex queries over mobile object positions. Significant research efforts have been dedicated to techniques for efficient processing of spatial and temporal continuous queries on mobile objects in a centralized location monitoring system [119, 74, 94, 25, 105, 92, 112, 72, 1]. However, as several researchers have pointed out [19, 57, 129], the centralized location monitoring architecture can create intolerable performance problems as the number of mobile objects grows rapidly. We observe two inherent assumptions that limit the scalability of many existing centralized approaches. First, most of the existing location management systems assume that mobile objects are only responsible for reporting their location information periodically, and the server (or a hierarchy of servers) is fully responsible for detecting interesting location changes, determining which mobile objects should be included in which moving queries at each instance of time or at a given time interval, and return those location updates that match certain thresholds to the users. For mobile applications that need to handle large number of mobile objects, these centralized approaches can suffer from dramatic performance degradation in terms of server load and network bandwidth. The second drawback of the existing centralized architectures is the assumption that all mobile objects in a given universe of discourse and their location updates are relevant for answering moving queries posted to the system. In reality, only a small subset of mobile objects is relevant to a given moving location query at any given instance of time. Thus, the amount of processing on the location updates of those mobile objects that do not contribute to the answers of the moving location queries is unnecessary and wasted, and it may cause significant performance degradation when the number of mobile objects is large and growing dynamically.

Keeping these problems in mind, we design and develop MobiEyes, a distributed real-time location monitoring service for moving location queries over a large and growing number of mobile objects. A moving location query can be viewed as a standing location tracking request that continuously monitors the locations of mobile objects of interest and return a subset of mobile objects that satisfy certain conditions. In this chapter we focus on the distributed architecture and a suite of optimization techniques for scalable processing of moving location queries on mobile objects.

A promising approach to tackle the scalability problem of centralized location monitoring architectures is to ship certain amount of the moving query processing down to a set of "nearby" mobile objects and to have the server mainly act as a mediator between these mobile objects. *The distribution of some moving query processing effort from the server to a selective set of mobile objects can be seen as a mechanism for moving computation close to the places where the location data of interest is produced.* Such techniques are especially beneficial when there are a large number of continuously mobile objects, generating immense number of position updates, but only a small subset of the location updates are of interest to each location query. By utilizing the computational capabilities available at the mobile objects, we can reduce the load on the server, filter irrelevant location updates, and increase the overall utilization and the scalability of the system. This computation partitioning approach taken by MobiEyes for evaluating moving queries is further motivated by the rapid and continued upsurge of computational capabilities in mobile devices, ranging from GPS-based navigational systems in cars to hand-held devices and smart cell phones.

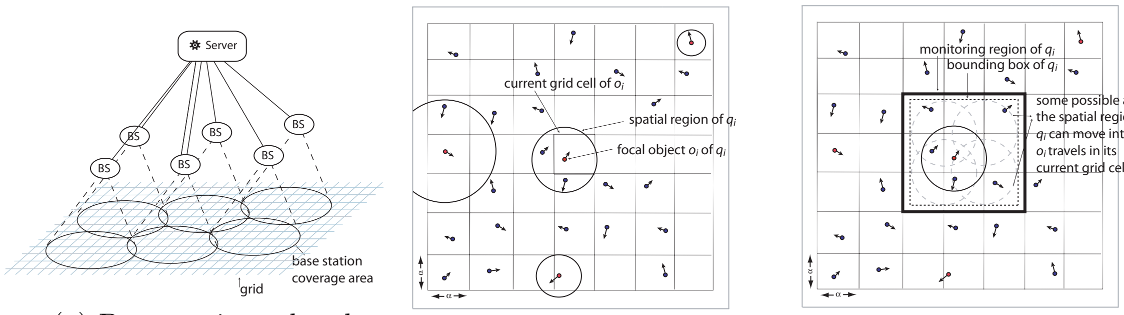
In order to control the amount of computations to be handled by the mobile objects that are nearby the spatial regions of active location queries, and to enhance the overall performance and system utilization of MobiEyes, we develop a set of optimization techniques, such as *Moving Query Grouping*, *Lazy Query Propagation*, and *Safe Query Periods*. We use Query Grouping to constrict the amount of computation to be performed by the mobile objects and to minimize the number of messages sent on the wireless medium in situations where there are large groups of moving queries with identical focal objects. We use Lazy Query Propagation to allow tradeoffs between query precision and network bandwidth cost as well as energy consumption on the mobile objects. We use Safe Periods to decrease the query processing load on mobile objects due to periodic reevaluation of moving queries. Important metrics to validate the proposed architecture and optimizations include messaging cost, server load, and amount of computation at individual mobile objects. We present an analytical model for estimating the messaging cost of our solution, which guides us to find the optimal settings of certain system-wide parameters. We evaluate the scalability of the MobiEyes distributed location monitoring approach using a simulation model based on a mobile setup. The experimental results show that the MobiEyes approach can lead to significant savings in terms of server load and messaging cost, when compared to solutions relying on fully centralized processing of location information at the server(s).

3.2 System Model

The MobiEyes system model includes the set of underlying assumptions used in the design of MobiEyes, the mobile object model, and the moving query model. Before we get into a formal description of the system model, we first informally describe the concept of moving queries on mobile objects.

A moving location query on mobile objects (MQ for short) is a *spatial continuous moving query over locations of mobile objects*, and we also call a moving location query a moving query (MQ) for reference convenience. A MQ defines a spatial region bound to a specific mobile object and a filter which is a Boolean predicate on object properties. The result of a MQ consists of objects that are inside the area covered by the query's spatial region and satisfy the query filter. MQs are continuous queries [78] in the sense that the results of queries continuously change as time progresses. We refer to the object to which a MQ is bounded, the *focal object* of that query. The set of objects that are subject to be included in a query's result are called *target objects* of the MQ. Note that the spatial region of a MQ also moves as the focal object of the MQ moves.

There are many examples of moving queries on mobile objects in real life. For instance, the query MQ_1: "Give me the number of friendly units within 5 miles radius around me during next 2 hours" can be submitted by a soldier equipped with mobile devices marching in the field, or a moving tank in a military setting. The query MQ_2: "Give me the positions of those customers who are looking for taxi and are within 5 miles (of my location, at an interval of every minute) during the next 20 minutes" can be posted by a taxi driver moving on the road. The focal object of MQ_1 is the solider marching in the field or a moving tank. The focal object of MQ_2 is the taxi driver on the road. Different specializations of MQs can result in interesting and useful classes of queries on mobile objects. One such specialization is the case where the target objects are static objects in the query region, which leads to *moving queries on static objects*. An example of such a query is MQ_3: "Give me the locations and names of the gas stations offering gasoline for less than \$1.2 per gallon within 10 miles, during next half an hour" posted by a driver of a moving car, where the focal object of the query is the car on the move and the target objects are buildings within 10 miles with respect to the location of the car on the move. Another interesting specialization is the case where the queries are posed with static focal objects or without focal objects. In this case, a MQ becomes a *static spatial continuous query on mobile objects*. An example query is MQ_4: "Give me the list of AAA vehicles that are currently on service call in


Server
BS
base station coverage area
grid
current grid cell of o_i
spatial region of q_i
focal object o_i of q_i
α
monitoring region of q_i
bounding box of q_i
some possible areas the spatial region of q_i can move into as o_i travels in its current grid cell

(a) Base station related concepts (b) Mobile object concepts (c) MQ related concepts

Figure 21: Illustration of concepts

downtown Atlanta (or 5 miles from my office location), during the next hour".

3.2.1 System Assumptions

The design of the MobiEyes system is based on the following assumptions. All these assumptions are widely agreed upon by many or have been seen as common practice in most existing mobile systems in the context of monitoring and tracking of mobile objects: (1) *Mobile objects are able to locate their positions and are able to determine their velocity vector*, e.g. using GPS [127]. (2) *Mobile objects have synchronized clocks*, e.g. using GPS or NTP [85]. (3) *Mobile objects have computational capabilities to carry out computational tasks.*

In addition, we assume that the geographical area of interest is covered by several base stations, which are connected to the service provider's server farm (simply called the server in the rest of the chapter). We assume that all location service requests are served through a three-tier architecture, that consists of mobile objects, base stations, and the server. Broadcast is used to establish connections from the server to the mobile objects through the base stations. The mobile objects can only communicate with the base station if they are located in the coverage area of the base station. Base stations can communicate with the server through wired networking. Figure 21 (a) shows this architecture.

3.2.2 The Mobile Object Model

Let O be the set of mobile objects. Formally we can describe a mobile object $o \in O$ by a quadruple: $\langle oid, pos, \overline{vel}, \{props\} \rangle$. oid is the unique object identifier. pos is the current position of the object o. $\overline{vel} = (velx, vely)$ is the current velocity vector of the object, where $velx$ is its velocity in the x-dimension and $vely$ in the y-dimension. $\{props\}$ is a set of properties about the mobile object o, including spatial, temporal, and object-specific properties (even application specific attributes registered on the mobile unit by the user).

The basic notations used in the subsequent sections of the chapter are formally defined below:

Rectangle shaped region and circle shaped region : A rectangle shaped region is defined by $Rect(lx, ly, w, h) = \{(x, y) : x \in [lx, lx + w] \ \wedge \ y \in [ly, ly + h]\}$, where lx and ly are the x-coordinate and the y-coordinate of the lower left corner of the rectangle, w is the width and h is the height of the rectangle. A circle shaped region is defined by $Circle(cx, cy, r) = \{(x, y) : (x - cx)^2 + (y - cx)^2 \leq r^2\}$, where cx is the x-coordinate and cy is the y-coordinate of the circle's center, and r is the radius of the circle.

Universe of Discourse (UoD): We refer to the geographical area of interest as the universe of discourse, which is defined by $U = Rect(X, Y, W, H)$, where X is the x-coordinate and Y is the y-coordinate of the lower left corner of the rectangle shaped region corresponding to the universe of discourse. W is the width and H is the height of the universe of discourse.

Grid and Grid cells: In MobiEyes, we map the universe of discourse $U = Rect(X, Y, W, H)$ onto a grid G of cells. Each grid cell is an $\alpha \times \alpha$ square area, and α is a system parameter that defines the cell size of the grid G. Formally, a grid corresponding to the universe of discourse U can be defined as $G(U, \alpha) = \{A_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N, A_{i,j} = Rect(X+i*\alpha, Y+j*\alpha, \alpha, \alpha), M = \lceil H/\alpha \rceil, N = \lceil W/\alpha \rceil\}$. $A_{i,j}$ is an $\alpha \times \alpha$ square area representing the grid cell that is located on the ith row and jth column of the grid G.

Position to Grid Cell Mapping: Let $pos = (x, y)$ be the position of a mobile object in the universe of discourse $U = Rect(X, Y, W, H)$. Let $A_{i,j}$ denote a cell in the grid $G(U, \alpha)$. $Pmap(pos)$ is a position to grid cell mapping, defined as $Pmap(pos) = A_{\lceil \frac{pos.x - X}{\alpha} \rceil, \lceil \frac{pos.y - Y}{\alpha} \rceil}$.

Current Grid Cell of an Object: Current grid cell of a mobile object is the grid cell which contains the current position of the mobile object. If $o \in O$ is an object whose current position, denoted as $o.pos$, is in the Universe of Discourse U, then the current grid cell of the object is formally defined by $curr\_cell(o) = Pmap(o.pos)$.

Base Stations: Let $U = Rect(X, Y, W, H)$ be the universe of discourse and B be the set of base stations overlapping with U. Assume that each base station $b \in B$ is defined by a circle region $Circle(bsx, bsy, bsr)$ with (bsx, bsy) being the center of the circle and bsr being the radius of the circle. We say that the set B of base stations covers the universe of discourse U, i.e. $\bigcup_{b \in B} b \supseteq U$.

Grid Cell to Base Station Mapping: Let $Bmap : \mathbb{N} \times \mathbb{N} \rightarrow 2^B$ define a mapping, which maps a grid cell index to a non-empty set of base stations. We define $Bmap(i, j) = \{b : b \in B \ \wedge \ b \cap A_{i,j} \neq \emptyset\}$. $Bmap(i, j)$ is the set of base stations that cover the grid cell $A_{i,j}$.

See Figure 21(a) and Figure 21(b) for example illustrations.

3.2.3 Moving Query Model

Let Q be the set of moving queries. Formally we can describe a moving query $q \in Q$ by a quadruple: $\langle qid, oid, region, filter \rangle$. qid is the unique query identifier. oid is the object identifier of the focal object of the query. $region$ defines the shape of the spatial query region bound to the focal object of the query. $region$ can be described by a closed shape description such as a rectangle, or a circle, or any other closed shape description which has a computationally cheap point containment check. This closed shape description also specifies a binding point, through which it is bound to the focal object of the query. Without loss of generality we use a circle, with its center serving as the binding point to represent the shape of the region of a moving query in the rest of the chapter. $filter$ is a Boolean predicate defined over the properties $\{props\}$ of the target objects of a moving query q. Example properties include characteristics of the target objects, or specific spatial regions. A moving query "give me the list of AAA vehicles on highway 85 north and within 20 miles of my current location" can be posed by a driver of a car on the road. This query has used the filter "AAA vehicles on highway 85 north" to define the target objects of interest. For presentation convenience, in the rest of the chapter we consider the result of a MQ as the set of object identifiers of the mobile objects that locate within the area covered by the spatial region of the query and satisfy the filter condition.

Formal definition of basic notations regarding MQs is given below (see Figure 21(c) for illustrations):

Bounding Box of a Moving Query: Let $q \in Q$ be a query with focal object $fo \in O$ and spatial region *region*, let rc denote the current grid cell of fo, i.e. $rc = \mathit{curr\_cell}(fo)$. Let lx and ly denote the x-coordinate and the y-coordinate of the lower left corner point of the current grid cell rc. The *Bounding Box* of a query q is a rectangle shaped region, which covers all possible areas that the spatial region of the query q may move into when the focal object fo of the query travels within its current grid cell. For circle shaped spatial query region with radius r, the bounding box can be formally defined as $\mathit{bound\_box}(q) = \mathit{Rect}(rc.lx - r, rc.ly - r, \alpha + 2r, \alpha + 2r)$.

Monitoring Region of a Moving Query: The grid region defined by the union of all grid cells that intersect with the bounding box of a query forms the monitoring region of the query. It is formally defined as, $\mathit{mon\_region}(q) = \bigcup_{(i,j)\in S} A_{i,j}$, where $S = \{(i,j) : A_{i,j} \cap \mathit{bound\_box}(q) \neq \emptyset\}$. The monitoring region of a moving query covers all the objects that are subject to be included in the result of the moving query when the focal object stays in its current grid cell.

Nearby Queries of an Object: Given a mobile object o, we refer to all MQs whose monitoring regions intersect with the current grid cell of the mobile object o the *nearby queries* of the object o, i.e. $\mathit{nearby\_queries}(o) = \{q : \mathit{mon\_region}(q) \cap \mathit{curr\_cell}(o) \neq \emptyset \wedge q \in Q\}$. Every mobile object is either a target object of or is of potential interest to its nearby MQs.

MobiEyes handles the dynamics of mobile object side query installation at the granularity of grid cells, which entails that the bounding boxes should be extended to cover an integral number of grid cells. Such extended bounding boxes are called monitoring regions. This is the main intuition behind defining monitoring regions.

We make two important observations. First, at any given time we have a set O of mobile objects and a set Q of moving queries in the MobiEyes system. When the size of the set O is relatively large compared to the size of Q, it is more likely that there are mobile objects in O, which do not have any nearby moving queries in Q. The larger the size of set O, the higher the number of mobile objects whose current grid cells do not intersect with the monitoring region of any MQs currently installed in the system. Second, each moving query in MobiEyes is associated with a stop condition specifying when the moving query will be terminated. This warrants that every moving query will be terminated at some point in time. We can view a moving query MQ as a continual query [78] of the form $\langle query, trigger, stop\rangle$. The *query* component is defined in terms of the *focal object*, the query *region*, and the *filter*. The *trigger* condition is the location update reporting interval of the focal object, and the *stop* condition is the termination time of the MQ. We say a MQ is active if its stop condition is not yet met. In the rest of the chapter we model a MQ in terms of its *query* component.

3.3 Distributed Architecture

Motivated by the fact that mobile users are typically interested in other mobile objects nearby regardless of the total network size, the main idea underlying the MobiEyes' distributed architecture is to have each mobile object determine by itself whether or not it should be included in the result of a location query nearby, without requiring global knowledge regarding the moving location queries and the object positions of interest. An immediate advantage of the MobiEyes approach is the significant saving in terms of server load and communication bandwidth. Concretely, in MobiEyes, mobile objects that are not focal objects of any moving location queries do not need to report their position or velocity changes to the location server if they do not have any nearby queries. The focal objects only report to the location server when their velocity vectors change above certain threshold or when they move out of their current grid cells.

A main challenge in the design of the MobiEyes distributed architecture is two fold. First, we need to develop algorithms to carefully partition the processing cost of MQs into the mediation processing at the server side and the local processing at the mobile object side. Second, we need methods to identify the most appropriate subset of mobile objects that can contribute to the answer of a given MQ in a given period of time.

We first introduce *the concept of monitoring region* for each moving query to model the part of the grid area to which the spatial query region is confined when the focal object of the query changes its position within its current grid cell. Second, we design the *concrete data structures* and the *distributed coordination mechanisms* that are location query aware for managing the communication and collaboration between the server and the chosen subset of mobile objects, focusing on reducing the server load as well as the messaging cost and thus the network bandwidth and power consumption at mobile objects. This architecture design is especially effective when the number of mobile objects in the geographical region considered is large and the number of MQs is relatively small and skewed in terms of query distribution over the mobile objects considered. We also develop a set of *optimizations* for efficient processing of MQs. We employ the lazy query propagation technique to reduce the messaging cost of the communication between the mobile objects and the server. We introduce the MQ grouping techniques to allow MobiEyes to handle hot spot queries efficiently. We utilize the concept of safe period to allow further reduction on the amount of local query processing at the mobile object side. We use the dead-reckoning technique, popular in many centralized proposals for processing spatial queries on mobile objects, to predict the position changes of mobile objects of interest.

In the subsequent sections, we first describe the main data structures used in MobiEyes and then provide a detailed discussion on server side processing and mobile object side processing. We defer the discussion on the set of optimization techniques to Section 3.4.

3.3.1 Data Structures

Server Side Data Structures

The server side stores four types of data structures: the focal object table FOT, the server side moving query table SQT, the reverse query index matrix RQI, and the static grid cell to base station mapping $Bmap$.

Focal Object Table, $FOT = (\underline{oid}, pos, \overline{vel}, tm)$, is used to store information about mobile objects that are the focal objects of MQs. The table is indexed on the oid attribute, which is the unique object identifier. tm is the time at which the position, pos, and the velocity vector, $\overline{vel}$, of the focal object with identifier oid were recorded on the mobile object side. When the focal object reports to the server its position and velocity change, it also includes this timestamp in the report.

Server Side Moving Query Table, SQT = ($\underline{qid}$, oid, $region$, $curr\_cell$, mon_region, $filter$, $\{result\}$), is used to store information about all spatial queries hosted by the system. The table is indexed on the qid attribute, which represents the query identifier. oid is the identifier of the focal object of the query. $region$ is the query's spatial region. $curr\_cell$ is the grid cell in which the focal object of the query locates. mon_region is the monitoring region of the query. $\{result\}$ is the set of object identifiers representing the set of target objects of the query. These objects are located within the query's spatial region and satisfy the query filter.

Reverse Query Index, RQI, is an $M \times N$ matrix whose cells are a set of query identifiers. M and N denote the number of rows and the number of columns of the Grid corresponding to the Universe of Discourse of a MobiEyes system. $RQI(i, j)$ stores the identifiers of the queries whose monitoring regions intersect with the grid cell $A_{i,j}$. $RQI(i, j)$ represents the nearby queries of an object whose current grid cell is $A_{i,j}$, i.e. $\forall o \in O, nearby\_queries(o) = RQI(i, j)$, where $curr\_cell(o) = A_{i,j}$. Formally it is defined as follows: $RQI(i, j) = \{qid : \exists e \in SQT \text{ s.t. } e.qid = qid \ \wedge \ e.mon\_region \cap A_{i,j} \neq \emptyset\}$.

Mobile Object Side Data Structures

Each mobile object o stores a local query table LQT and a Boolean variable $hasMQ$.

Local Query Table, LQT = (qid, pos, $\overline{vel}$, tm, $region$, mon_region, $isTarget$) is used to store information about moving queries whose monitoring regions intersect with the current grid cell in which the mobile object o currently locates in. qid is the unique query identifier assigned at the time when the query is installed at the server. pos is the last known position, and $\overline{vel}$ is the last known velocity vector of the focal object of the query. tm is the time at which the position and the velocity vector of the focal object was recorded (by the focal object of the query itself, not by the object on which LQT resides). $isTarget$ is a Boolean variable describing whether the object was found to be inside the query's spatial region at the last evaluation of this query by the mobile object o.

The Boolean variable $hasMQ$ provides a flag showing whether the mobile object o storing the LQT is a focal object of some query or not.

3.3.2 Server Side Processing

The location server side processing consists of two main tasks. First, it handles moving location query installation requests from end-users of the location service application, including mobile users. Second, it performs the mediation of location query processing at the server side and mobile object side, including receiving and responding to (i) the significant changes in velocity vector information of the mobile objects that are focal objects of some location queries and (ii) the grid cell change events resulting from movement of focal or non-focal objects out of their current grid cells. We leave the details of the second task to Section 3.4.

To enable efficient processing at mobile object side, we introduce the *monitoring region of a moving location query* to identify all mobile objects that may get included in the query's result when the focal object of the query moves within its current cell. The main idea is to have those mobile objects that reside in a moving query's monitoring region to be aware of the query and to be responsible for calculating if they should be included in the query result. Thus, the mobile objects that are not in the neighborhood of a moving query do not need to be aware of the existence of the moving query, and the query result can be efficiently maintained by the objects in the query's monitoring region in a differential manner.

Installing MQs at the Location Server: Installation of a moving location query to the MobiEyes system consists of two phases. First, the MQ is installed at the server side and the server state is updated to reflect this. Second, the query is registered at the set of mobile objects that are located inside the monitoring region of this MQ.

When the server receives a new MQ in the form $(oid, region, filter)$, it performs the following installation actions. (1) It first checks whether the focal object with identifier *oid* is already contained in the FOT table. (2) If the focal object of the query already exists, it means that either someone else has installed the same query earlier or there exist multiple queries with different filters but the same focal object. Since the FOT table already contains velocity and position information regarding the focal object of this query, the installation simply creates a new entry for this new MQ and adds this entry to the sever-side query table SQT and then modifies the RQI entry that corresponds to the current grid cell of the

```
(A) SERVER: RECEIVED QUERY($oid$, $region$, $filter$)
(1)  Let $e = (oid, *, *, *)$ in $FOT$
(2)  if $e \neq \emptyset$
(3)      $pos \leftarrow e.pos$
(4)  else
(5)      Locate the position, $pos$, and velocity vector, $\overline{vel}$, of the mobile object with identifier $oid$
         together with the time, $tm$, this information was recorded. In case the query is received
         from the object with identifier $oid$ itself, then this information is also assumed to be received
         with the query. Otherwise request this information from the object.
(6)      Insert the entry $(oid, pos, \overline{vel}, tm)$ into $FOT$
(7)  Assign a unique identifier, $qid$, to the query
(8)  $curr\_cell \leftarrow Pmap(pos)$
(9)  Set $mon\_region$ using $curr\_cell$ and $region$
(10) Insert the entry ($qid$, $oid$, $region$, $curr\_cell$, $mon\_region$, $filter$, $\emptyset$) into $SQT$
(11) foreach $A_{i,j} \subset mon\_region$
(12)     $RQI(i,j) \leftarrow RQI(i,j) \cup \{qid\}$
(13) SEND($oid$, [Query installed])
(14) $B' \leftarrow \bigcup_{A_{i,j} \in mon\_region} Bmap(i,j)$
(15) foreach $b \in B'$
(16)     BROADCAST($b$, [Install query ($qid$, $pos$, $\overline{vel}$, $tm$, $region$, $filter$)])

(B) MOBILE OBJECT: RECEIVED MESSAGE [QUERY INSTALLED]()
(1)  $hasMQ \leftarrow true$

(C) MOBILE OBJECT: RECEIVED MESSAGE [INSTALL QUERY]($qid$, $pos$, $\overline{vel}$, $tm$, $region$, $filter$)
(1)  Use $Pmap(pos)$ and $region$ to calculate the $mon\_region$
(2)  if current position is in $mon\_region$ and $filter(this) = true$
(3)      Insert entry $(qid, pos, \overline{vel}, tm, region, mon\_region, false)$ into $LQT$
```

Figure 22: Algorithm: New moving query posted to server

focal object to include this new MQ in the reverse query index (detailed in step (4)). At this point the query is installed on the server side. (3) However, if the focal object of the query is not present in the *FOT* table, then the server-side installation manager needs to contact the focal object of this new query and request the position and velocity information. Then the server can directly insert the entry $(oid, pos, \overline{vel}, tm)$ into FOT, where tm is the timestamp when the object with identifier oid has recorded its pos and $\overline{vel}$ information. (4) The server then assigns a unique identifier qid to the query and calculates the current grid cell ($curr\_cell$) of the focal object and the monitoring region (mon_region) of the query. A new moving query entry (qid, oid, $region$, $curr\_cell$, mon_region, $filter$) will be created and added into the SQT table. The server also updates the RQI index by adding this query with identifier qid to $RQI(i,j)$ if $A_{i,j} \cap mon\_region(qid) \neq \emptyset$. At this point the query is installed on the server side.

There is a small complication involved in the installation of new MQs (see action (3)). If the focal object of the new MQ is not present in the FOT table, and the MQ is not posted by the focal object itself, then the server needs to first locate the focal object of the query in order to obtain its current position and velocity vector information. A simple solution is to use an additional table stored on the location server side, which stores base station level [3, 21, 13, 89], or higher level location information regarding mobile objects. For instance, in case we store base station level location information regarding mobile objects, given an object identifier it is possible to locate the base station which covers its current location. During query installation, this will enable us to communicate with the focal object to receive its position and velocity vector information. The granularity of the information maintained about object positions characterizes the tradeoff between the cost of locating an object (in which base station's coverage area it resides in) and the cost of maintaining this information.

After installing queries on the server side, the server needs to complete the installation by triggering query installation on the mobile object side. This job is done by performing two tasks. First, the server sends an installation notification to the focal object with identifier oid, which upon receiving the notification sets its $hasMQ$ variable to true. This makes sure that the mobile object knows that it is now a focal object and is supposed to report velocity vector changes to the server. The second task is for the server to forward this query to all objects that reside in the query's monitoring region, so that they can install the query and monitor their position changes to determine if they become the target objects of this query. To perform this task, the server uses the mapping $Bmap$ to determine the minimal set of base stations that covers the monitoring region. Then the query is sent to all objects that are covered by the base stations in this set through broadcast messages. The detailed procedures for location query installation are given in Figure 22.

3.3.3 Mobile Object Side Processing

In MobiEyes, a mobile user can issue a location query or choose to enter a sleep mode at anytime. Whenever mobile objects are awake and active, we assume that they are willing

```
(A) MOBILE OBJECT: PERIODICAL QUERY PROCESSING()
(1)  Let $pos$ be the current position of the object and $ctm$ be the current time
(2)  foreach $e$ in $LQT$
(3)      $fpos \leftarrow e.pos + (ctm - e.tm) * e.\overline{vel}$ {predict the position of the focal object}
(4)      if $(pos - fpos) \in e.region$
(5)          if $e.isTarget = false$
(6)              SEND($server$, [Query result change $(e.qid, oid, true)$])
(7)              $e.isTarget \leftarrow true$
(8)      else if $e.isTarget = true$
(9)          SEND($server$, [Query result change $(e.qid, oid, false)$])
(10)         $e.isTarget \leftarrow false$

(B) SERVER: RECEIVED MESSAGE [QUERY RESULT CHANGE]($qid$, $oid$, $isTarget$)
(1)  Let $e = (qid, *, *, *, *, *, *, *)$ in $SQT$
(2)  if $isTarget = true$
(3)      $e.result \leftarrow e.result \cup \{oid\}$
(4)  else
(5)      $e.result \leftarrow e.result \setminus \{oid\}$
```

Figure 23: Algorithm: Mobile object query processing logic

to participate in the corporative processing of nearby location queries. The task of making sure that the mobile objects in the monitoring region of a moving location query are aware of this MQ, is accomplished through server broadcasts, which are triggered by (i) server side installations of new moving location queries or (ii) significant velocity vector changes of the focal objects or (iii) current grid cell changes of focal or non-focal objects.

Installing MQs at the Mobile Object Side: The mobile object side processing for location query installation is performed as follows: Upon receiving a broadcast message, a mobile object examines each MQ in the broadcast message using its local state and determines whether this MQ is nearby and whether it should be registered locally. The decision is primarily based on whether the mobile object itself is within the monitoring region of the MQ and whether the query's filter is also satisfied by the mobile object. If the answer to both of these questions is yes, the mobile object registers the query into its local query table LQT. Otherwise the object discards the MQ. The details of these procedures are given in Figure 22 (C).

Query Processing at the Mobile Objects: A mobile object periodically processes all location queries registered in its LQT table. For each locally registered MQ, the mobile object predicts the position of the focal object of the MQ using the velocity, time, and position information available in the LQT entry of the MQ (line 3 under (A) in Figure 23). Then the object compares its current position and the predicted position of the focal object of this MQ through a simple containment check based on the spatial region of this MQ, and determines whether itself is covered by the query's spatial region or not. If the result is not different than the last result computed in the previous time step, no reporting to the server is performed. Otherwise, we have one of two situations: the mobile object has just moved into the spatial region of the MQ or it has just moved out of the spatial region of this MQ. In both cases, the change is relayed to the location server. The server, in turn, differentially updates the query result to keep the answer to this MQ up to date. Figure 23 describes mobile object side query processing in detail.

3.4 Optimizations: Efficient and Reliable Processing of MQs

3.4.1 Efficient Processing of MQs

We have presented the basic algorithms for distributed processing of moving location queries. In this section we describe four optimization mechanisms used in MobiEyes to efficiently handle system dynamics with the aim of minimizing the server load and the amount of local processing at the mobile object side, and reducing the communication cost between the mobile objects and the location server.

3.4.1.1 Handling Mobility of Queries and Objects

Once a moving location query (MQ) is installed in the MobiEyes system, the focal object of the MQ needs to report to the server only when there is a significant change to its location information. We consider two types of changes to be significant: (1) when a focal object moves out of its current grid cell, or (2) when the focal object of a MQ changes its velocity vector beyond a pre-defined threshold. On the other hand, a non-focal object may [1] need to report to the server only when it changes its current grid cell. We describe the mechanisms for handling velocity vector changes first and then discuss the mechanisms for handling objects that change their current grid cells.

[1]Depends on whether Eager Query Processing (EQP) or Lazy Query Processing (LQP) is used

Handling Velocity Vector Changes with Dead Reckoning

The velocity vector of a mobile object will almost always change at each time step in a real world setup, although the change might be insignificant compared to the previous time step. One way to control the amount of location updates from mobile objects to the location server is to notify the server of the new velocity vector information of the focal object of a MQ and have the server to relay such update to the objects located in the monitoring region of the MQ, only if the change in the velocity vector is significant. In MobiEyes, we use a variation of *dead reckoning* to decide what constitutes a (significant) velocity vector change.

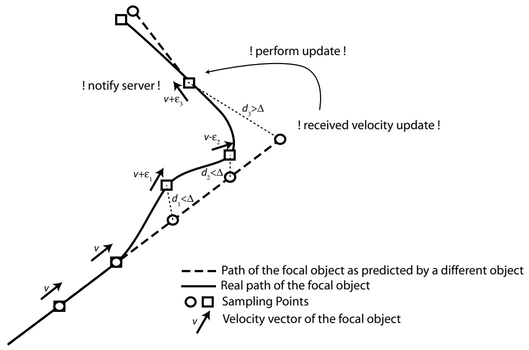

! perform update !
! notify server !
$v+\varepsilon_3$
$d_3>\Delta$
! received velocity update !
$v-\varepsilon_2$
$v+\varepsilon_1$
$d_2<\Delta$
$d_1<\Delta$
v
v
Path of the focal object as predicted by a different object
Real path of the focal object
Sampling Points
v
Velocity vector of the focal object

Figure 24: Dead reckoning in MQ evaluation

Concretely, at each time step the focal object of a query samples its current position and calculates the difference between its current position and the position predicted using the last velocity vector information it reported to the location server. In case this difference is larger than a threshold, say Δ, the new velocity vector information is relayed to the location server [2]. Figure 24 provides an illustration. The path of a focal object is depicted with a solid line, where its path predicted by objects in its monitoring region (based on its last velocity information relayed to the objects) is depicted with a dashed line. At each time step, the focal object first samples its position, which is depicted by small squares in the figure. Then it calculates the position that other objects predict it to be at, which is depicted with small circles in the figure. In case the distance between these two positions is larger than Δ, the focal object notifies the server with its new velocity vector and the

[2]We do not consider inaccuracies due to motion modeling. See [131] for a discussion of motion update policies and tradeoffs.

```
(A) MOBILE OBJECT: PERIODIC VELOCITY CHANGE PROCESSING()
(1) if $hasMQ = true$
(2)     Let $\overline{pvel}$ be the last velocity and $ppos$ be the last position information relayed to the server
        where $ptm$ is the time this information was relayed
(3)     Record the new position, $pos$, new velocity vector, $\overline{vel}$, and the current time, $tm$
(4)     $opos \leftarrow ppos + (tm - ptm) * \overline{pvel}$ {perform dead-reckoning}
(5)     if $|opos - pos| > \Delta$
(6)         SEND($server$, [New velocity data $(oid, pos, \overline{vel}, tm)$])

(B) SERVER: RECEIVED MESSAGE [NEW VELOCITY DATA]($oid$, $pos$, $\overline{vel}$, $tm$)
(1) Let $e = (oid, *, *, *)$ in $FOT$
(2) $e \leftarrow (oid, pos, \overline{vel}, tm)$ {update the entry $e$}
(3) foreach $e = (*, oid, *, *, *, *, *, *)$ in $SQT$
(4)     $B' \leftarrow \bigcup_{A_{i,j} \subset e.mon\_region} Bmap(i, j)$
(5)     foreach $b \in B'$
(6)         BROADCAST($b$, [Query velocity change $(e.qid, pos, \overline{vel}, tm)$])

(C) MOBILE OBJECT: RECEIVED MESSAGE[QUERY VELOCITY CHANGE]($qid$, $pos$, $\overline{vel}$, $tm$)
(1) Let $e = (qid, *, *, *, *, *, *)$ in $LQT$
(2) if $e \neq \emptyset$
(3)     $e.pos \leftarrow pos$; $e.\overline{vel} \leftarrow \overline{vel}$; $e.tm \leftarrow tm$
```

Figure 25: Algorithm: Mobile object changed velocity vector

server relays the new velocity information of the focal object to all objects in its monitoring region through broadcast (see the fifth time step in Figure 24).

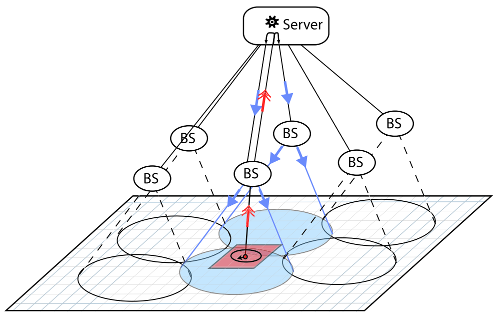


Figure 26: Conveying velocity vector changes

Concretely, when the focal object of a MQ reports a significant velocity vector change, it sends its new velocity vector, its position and the timestamp at which this information was recorded, to the server. The server first updates the FOT table with the information received from the focal object. Then for each query associated with the focal object, the server communicates the newly received information to objects located in the monitoring region of the query by using an optimized broadcast schedule, which generates the minimum number of broadcasts using the grid cell to base station mapping *Bmap*. An illustration of this process is given in Figure 26, where a focal object together with its monitoring region is shown. The focal object sends its new velocity information to the server (shown with double headed arrows in the figure), which in turn broadcasts this information using two base stations that cover the monitoring region of the query (shown with single headed arrows in Figure 26). The algorithm given in Figure 25 describes velocity vector change handling in detail.

One way to optimize the broadcast frequency of propagating velocity updates to mobile objects of interest is to let the server *batch several velocity vector updates from mobile objects and broadcast them during agreed upon time intervals* so that the mobile objects can activate their radio only during scheduled intervals, thus leading to considerable saving in terms of power consumption.

Handling Objects that Change Grid Cells: Eager or Lazy Query Propagation

Based on the grid structure and the monitoring region of MQs in MobiEyes, as long as all mobile object move within their current cells, the list of nearby MQs responsible by the mobile objects will remain the same. However, when a mobile object changes its current grid cell, such location update may cause a change to the set of moving location queries this object is responsible for monitoring. In case the object that has changed its current grid cell is a focal object, the location update may also cause a change to the set of mobile objects responsible for monitoring the MQs bounded to this focal object.

With *Eager Query Propagation* (*EQP*), when an object changes its current grid cell, it immediately notifies the server of this change by sending its object identifier, its previous grid cell and its new current grid cell to the server. The object also removes those queries whose monitoring regions no longer cover its new current grid cell from its local query table *LQT*. Upon receipt of the notification, if the object is a non-focal object, the server only needs to find what new queries should be installed on this object and then perform the query installation on this mobile object.

The server uses the reverse query index *RQI* together with the previous and the new current grid cell of the object to determine the set of new queries that has to be installed

```
(A) MOBILE OBJECT: CHANGED CURRENT GRID CELL()
(1) Let $pos$ be the new position of the mobile object
(2) Remove all entries $e$ in $LQT$ satisfying $pos \not\subset e.mon\_region$
(3) Let $A_{i_p,j_p}$ be the previous and $A_{i_c,j_c}$ be the current grid cell in which $pos$ lies
(4) SEND($server$, [Object changed grid cell $(oid, (i_p, j_p), (i_c, j_c))$])

(B) SERVER: RECEIVED MESSAGE [OBJECT CHANGED GRID CELL]($oid$, $(i_p, j_p)$, $(i_c, j_c)$)
(1)  $Q_{diff} \leftarrow RQI(i_c, j_c) \setminus RQI(i_p, j_p)$
(2)  foreach $qid \in Q_{diff}$
(3)      Let $e = (qid, *, *, *, *, *, *, *)$ in $SQT$
(4)      SEND($oid$, [Install query $(e.qid, e.pos, e.\overline{vel}, e.tm, e.region, e.filter)$])
(5)  Let $e = (oid, *, *, *)$ in $FOT$
(6)  foreach $e_s = (*, oid, *, *, *, *, *, *)$ in $SQT$
(7)      $e_s.curr\_cell \leftarrow A_{i_c,j_c}$
(8)      $old\_mon\_region \leftarrow e_s.mon\_region$
(9)      Set $e_s.mon\_region$ using $e_s.region$ and $e_s.curr\_cell$
(10)     foreach $A_{i,j} \subset (old\_mon\_region \setminus e_s.mon\_region)$
(11)         $RQI(i, j) \leftarrow RQI(i, j) \setminus \{e_s.qid\}$
(12)     foreach $A_{i,j} \subset (e_s.mon\_region \setminus old\_mon\_region)$
(13)         $RQI(i, j) \leftarrow RQI(i, j) \cup \{e_s.qid\}$
(14)     $combined\_area \leftarrow e_s.mon\_region \cup old\_mon\_region$
(15)     $B' \leftarrow \bigcup_{A_{i,j} \subset combined\_area} Bmap(i, j)$
(16)     foreach $b \in B'$
(17)         BROADCAST($b$, [Update query $(e_s.qid, e.pos, e.\overline{vel}, e.tm, e_s.region, e_s.filter)$])

(C) MOBILE OBJECT: RECEIVED MESSAGE [UPDATE.QUERY]($qid$, $pos$, $\overline{vel}$, $tm$, $region$, $filter$)
(1) Use $pos$ and $region$ to calculate the $mon\_region$
(2) Let $e = (qid, *, *, *, *, *, *)$ in $LQT$
(3) if current position is in $mon\_region$
(4)     if $e \neq \emptyset$
(5)         $e.mon\_region \leftarrow mon\_region$
(6)     else if $filter(this) = true$
(7)         Insert $(qid, pos, \overline{vel}, tm, region, mon\_region, false)$ into $LQT$
(8) else
(9)     Remove entry $e$ (if exists) from $LQT$
```

Figure 27: Algorithm: Mobile object changed its current grid cell

on this mobile object. Then the server sends the set of new queries to the mobile object for installation. The focal object table FOT and the server query table SQT are used to create the required installation information for the queries to be installed on the object. However, if the object that changes its current grid cell is also a focal object of some query, then for each query with this object as its focal object, the server performs the following operations: (1) It updates the query's SQT table entry by resetting the current grid cell and the monitoring region to their new values. (2) It also updates the RQI index to reflect the change. (3) Then the server computes the union of the query's previous monitoring region and its new monitoring region, and (4) sends a broadcast message to all objects that reside in this combined area. This message includes information about the new state of the query. Upon receipt of this message from the server, a mobile object performs the following operations for installing or removing a moving location query. It checks whether its current grid cell is covered by the query's monitoring region. If not, the object removes the query from its *LQT* table (if the entry already exists), since the object's position is no longer covered by the query's monitoring region. Otherwise, it installs the query if the query is not already installed and the query filter is satisfied, by adding a new query entry in the *LQT* table. In case that the query is already installed in *LQT*, it updates the monitoring region of the query's entry in *LQT*. The detailed procedure is given by Figure 27.

When using an eager query propagation scheme, we require each mobile object (focal or non-focal) that changes its current grid cell to report this change to its location server immediately. The only reason for a non-focal object to communicate with the server is to immediately obtain the list of new location queries that it needs to register in response to the change of its current grid cell. One way to reduce the amount of communication between mobile objects and the location server is to use a lazy query propagation scheme. This allows us to eliminate the need for non-focal objects to contact the server to obtain the list of new MQs. Concretely, instead of obtaining the new location queries from the server and installing them immediately on the object upon a change of grid cell, the mobile object can wait until the location server broadcasts the next velocity vector changes regarding the focal objects of the MQs to the area in which the object locates. In this case the velocity vector change notifications are expanded to include the spatial region and the filter of the moving location queries, so that the object can install the new queries upon receiving the broadcast message on the velocity vector changes of the focal objects of the MQs. Using lazy propagation, a mobile object upon changing its current grid cell will be unaware of the new set of location queries nearby until the focal objects of these location queries change their velocity vectors or move out of their current grid cells. Obviously lazy propagation works well when the gird cell size α is large and the focal objects change their velocity vectors frequently. The lazy query propagation may not prevail over the eager query propagation, when: (1) most of the focal objects do not change their velocity vectors to cause inaccurate predictions beyond the specified threshold frequently, (2) the grid cell size α is too small, and (3) the non-focal objects change their current grid cells at a much faster rate than the focal objects. In such situations, non-focal objects may end up not being included in some of nearby moving location queries. We experimentally evaluate the *Lazy Query Propagation* (LQP) approach and study its performance advantages as well as its impact on query result accuracy in Section 3.5.

3.4.1.2 Location Query Grouping

In MobiEyes, a mobile user can pose many different queries and a query can be posed multiple times by different users. Thus, many moving location queries may share the same focal object. Effective optimizations can be applied to handle multiple location queries bound to the same mobile object. These optimizations help decreasing both the computational load on the mobile objects and the messaging cost of the MobiEyes approach, in situations where the query distribution over focal objects is skewed. We define a set of moving location queries as *groupable MQs* if they are bounded to the same focal object. We refer to those groupable MQs that have the same monitoring region as *MQs with matching monitoring regions*, and refer to the groupable MQs that have different monitoring regions as *MQs with non-matching monitoring regions* (See Figure 28). Based on these different sharing patterns, different grouping techniques can be applied to groupable MQs.

Grouping MQs with Matching Monitoring Regions

MQs with matching monitoring regions can be grouped most efficiently to reduce the communication and processing costs of such queries. We illustrate this with an example.

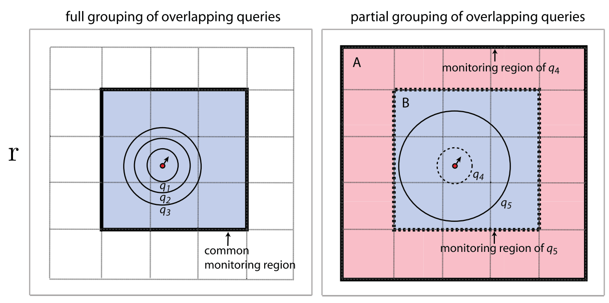

full grouping of overlapping queries
partial grouping of overlapping queries
r
q_1
q_2
q_3
common monitoring region
A
monitoring region of q_4
B
q_4
q_5
monitoring region of q_5

Figure 28: Minimizing duplicate processing by grouping moving location queries

Consider three MQs: $q_1 = (qid_1,\ oid_i,\ r_1,\ filter_1)$, $q_2 = (qid_2,\ oid_i,\ r_2,\ filter_2)$, and $q_3 = (qid_3,\ oid_i,\ r_3,\ filter_3)$ that share the same monitoring region. Note that these queries share their focal object, which is the object with identifier oid_i. Instead of shipping three separate queries to the mobile objects, the server can combine these queries into a single query as follows: $q_3 = (qid_3,\ oid_i,\ (r_1, r_2, r_3),\ (filter_1, filter_2, filter_3))$. To facilitate the local processing of groupable MQs, we introduce the concept of *query bitmap*, which is a bitmap containing one bit for each location query in a query group, each bit can be set to 1 or 0 indicating whether the corresponding query should include the mobile object in its result or not. When a mobile object is processing a set of groupable MQs with matching monitoring regions, it first checks if its current position is inside the spatial region of a location query with a larger radius. Only when its current position is inside the spatial region of a location query with a larger radius, it needs to consider the location queries with smaller radiuses. When a mobile object reports to the server whether it is included in the results of queries that form the grouped query or not, it will attach the query bitmap to the notification report. With the query bitmap the location server can easily determine whether the reporting mobile object should be included in the query results of which groupable MQs.

Grouping MQs with Non-Matching Monitoring Regions

For groupable MQs with non-matching monitoring regions, we propose to perform the location query grouping at the mobile object side only. We illustrate this claim with an example. Consider an object o_j inside region B in Figure 28. Since there is no global server side grouping performed for queries q_4 and q_5, o_j has both of them installed in its LQT table. o_j can save some processing by linking these two queries inside its LQT table. This way it only needs to consider the query with smaller radius only if it finds out that its current position is inside the spatial region of the one with the larger radius.

3.4.1.3 Reducing Local Processing Cost with Safe Period Optimization

In MobiEyes, each mobile object that resides in the monitoring region of a query needs to evaluate the queries registered in its local query table LQT periodically. For each query the candidate object needs to determine if it should be included in the answer of the query. The interval for such periodic evaluation can be set either by the server or by the mobile object itself. A safe-period optimization can be applied to reduce the computation load on the mobile object side, which computes a safe period for each object in the monitoring region of a query, if an upper bound ($maxVel$) exists on the maximum velocities of the mobile objects.

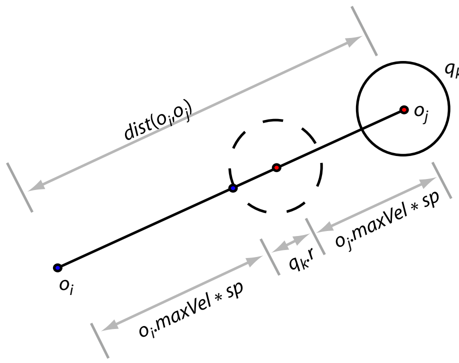


Figure 29: Safe period optimization

The safe periods for queries are calculated by an object o as follows: For each query q in its LQT table, the object o calculates a worst case lower bound on the amount of time that has to pass for it to locate inside the area covered by the query q's spatial region. We call this time, the *safe period* (sp) of the object o with respect to the query q, denoted as $sp(o, q)$. The safe period can be formally defined as follows. Let o_i be the object that has the query q_k with focal object o_j in its LQT table, and let $dist(o_i, o_j)$ denote the distance between these two objects, and let $q_k.region$ denote the circle shaped region with radius r. In the worst case, the two objects approach to each other with their maximum velocities in the direction of the shortest path between them, as shown in Figure 29. Then $sp(o_i, q_k) = \frac{dist(o_i,o_j)-r}{o_i.maxVel+o_j.maxVel}$.

Once the safe period sp of a mobile object is calculated for a query, it is safe for the object to start the periodic evaluation of this query after the safe period has passed. In order to integrate this optimization with the base algorithm, we include a *processing time* (ptm) field into the LQT table, which is initialized to 0. When a query in LQT is to be processed, ptm is checked first. In case ptm is ahead of the current time ctm, the query is skipped. Otherwise, it is processed as usual. After processing of the query, if the object is found to be outside the area covered by the query's spatial region, the safe period *sp* is calculated for the query and processing time *ptm* of the query is set to current time plus the safe period, *ctm*+*sp*. When the query evaluation period is short, or the object speeds are low or the cell size α of the grid is large, this optimization can be very effective.

3.4.2 Reliable Processing of MQs

Another important issue in distributed processing of moving queries is the level of reliability guarantee that the system can provide. In MobiEyes, the reliability of the system is defined by the reliability of the mobile objects and the reliability of the server (including the reliability of the communication between mobile objects and the server) with respect to distributed processing of moving queries.

3.4.2.1 Reliability of the Mobile Objects

In MobiEyes, each mobile object whose current grid cell intersects with the monitoring region of a moving query will maintain an entry associated with that query in its local MQ table (*LQT*) (recall Section 3.3.1). The *LQT* table is the most important state information maintained at the mobile object. It keeps all the MQs that this mobile object needs to process. In the event of failure, such as the computational unit on a mobile object crashes, the *LQT* state is either lost (when the previous state information was not stored persistently and made available) or less current (when the recovery can only restart the system at the previous checkpoint of the state). One way to recover this state is to obtain the new state through re-initialization with the server upon restart. However several problems (such as stale *LQT* state, incorrect query results) may occur when the mobile object continues to move in the presence of crashes on its computing unit or the focal objects of some MQs in its *LQT* table continue to move during the failure period. The degree of damages caused by such problems depends upon whether the mobile object experiencing the failure is a non-focal or focal object.

Failure at a Non-focal Mobile Object

When failure happens at a mobile object that is a non-focal object, we may have the mobile object incorrectly included in some of the query results for an arbitrarily long time. This is primarily caused by the following two possible errors due to the failure at the non-focal object: First, the non-focal object may continue to move in the presence of crashes on its computing unit. Such location changes will not be reported to the server due to the failure at the non-focal object. Second, the focal objects of some MQs in the *LQT* table of this non-focal object may continue to move during the failure period and report their locations to the server whenever a significant change occurs. The location updates of these focal objects will be disseminated to this non-focal object by the server through broadcast, but this broadcast will not be received and processed at the non-focal object due to failure. In both cases, the location changes of the non-focal object or the location changes of the focal objects in its *LQT* table, during the period of failure, may cause following events to happen: (1) The spatial regions of some MQs in the *LQT* table of the non-focal object no longer contain its position; and (2) The current grid cell of the non-focal object no longer intersects with the monitoring regions of some MQs listed in its *LQT* table before the crash.

In an ordinary situation, when the non-focal object detects that the spatial region of a MQ in its *LQT* table no longer contains its position, the non-focal object will be removed from the result set of this MQ and this change will be reported to the server through a query result change update. Similarly, when the non-focal object detects that the monitoring region of a MQ in its *LQT* table does not intersect with its current grid cell anymore, it will remove the MQ from its *LQT* table. Furthermore, the reverse query index maintained at the server (which corresponds to the entries in *LQT* table) will no longer list this MQ in the moving query list corresponding to the current grid cell of the non-focal object.

However, due to the failure happened at the non-focal object, its *LQT* table can only be recovered to the new state through re-initialization with the server upon the restart of the computing unit, based on the reverse query index of the non-focal object's current grid cell. The new *LQT* table may not contain the MQs whose results, before the crash, included the non-focal object (due to events (1) and (2) described above). As a consequence, the results of these MQs at the server side will falsely include this non-focal object, for two reasons: (*i*) The non-focal object did not send the query result changes to the server due to failure;

and (*ii*) It could not send these changes to the server upon the restart, as its new *LQT* table does not include these MQs anymore.

In short, it is the mobile object's responsibility to report to the server when it changes its state with regard to being included in or excluded from a query's result. When the mobile object experiences failure such as crashes of the computing unit, it loses not only its *LQT* table but also the local processing and reporting capability. Since the recovered *LQT* table through re-initialization with the server upon restart may not include some of the MQs whose query results should have been updated during the failure period to exclude the mobile object from their query result sets, the failure leads to incorrect query results maintained at the server side for an arbitrary long time.

Failure at a Focal Mobile Object

When failure happens at a mobile object that is a focal object, in addition to the problems stated above, a new problem arises: We may have stale moving query entries, corresponding to the focal object experiencing failure, residing in the *LQT* tables of other mobile objects. This is because, location changes of the focal object that has experienced failure will be lost during the failure period. In an ordinary situation, such location changes may result in MQs associated with the focal object to be removed from the *LQT* tables of other objects when the current cells of these objects do not intersect with the new monitoring region of the MQs associated with the focal object.

However, due to the failure happened at the focal object, such monitoring region changes are lost at the focal object and in turn *LQT* state updates are not performed at the mobile objects whose *LQT* tables contain MQs associated with the focal object. As a result, the mobile objects that were residing in the monitoring region just before the focal object crashed, but are not residing in the monitoring region upon the restart of the focal object, may still have an entry in their *LQT* tables corresponding to the failed focal object, which should have been removed in an ordinary situation. Furthermore, these entries may contain stale information about focal objects and may result in sending wrong query result updates. In the worst case, these entries may stay in the *LQT* tables for an arbitrarily long time.

The above-mentioned problems can be aggravated when the computational unit on a mobile object fails and *stops* for an arbitrarily long time. When this happens, we need efficient mechanisms such that queries corresponding to such failed and stopped focal objects and query result entries corresponding to such failed and stopped focal or non-focal objects can be detected in time and removed from the system.

In MobiEyes, we introduce a simple and yet effective mechanism, called *forced updates*, to solve the problems described above.

Error Handling Through Forced Updates

MobiEyes provides two kinds of forced updates to ensure the reliability of the system against failures:

Forced Result Updates: In the basic model of MobiEyes without reliability guarantee, each mobile object sends query result updates to the server only when it is included into or excluded from the result of a MQ in its LQT table. With forced result updates, we additionally require that each mobile object reports its state on whether it is included in or excluded from the result of a MQ in its LQT table to the server periodically (every T_r time unit), regardless of whether or not a change has occurred in the inclusion status of the object with respect to the results of queries in LQT.

Forced Velocity Vector Updates: Similarly, in the basic model of MobiEyes, each focal object sends location updates to the server whenever its velocity vector has changed significantly. With forced velocity vector updates, we additionally require that each focal object reports its location update to the server periodically (every T_r time unit) even if no significant change occurred in the velocity vector.

With the forced result updates, a query result entry is considered invalid if it is not updated during the last $c * T_r$ time units. Similarly, with forced velocity vector updates, a moving query entry in a LQT table is considered invalid if the velocity vector field of the entry is not updated during the last $c * T_r$ time units. The time interval parameter T_r adjusts the tradeoff between performance and accuracy. With smaller values of T_r errors are resolved faster (increased accuracy) but more messages need to be exchanged between mobile objects and the server. Accuracy is obtained with the price of performance. On the other hand, with larger values of T_r we require less messages to be exchanged between mobile objects and the server (increased performance), but errors are resolved slower (decreased accuracy). The parameter c is used to control the tolerance to delays and errors in the communication.

3.4.2.2 Reliability of the Server

The distributed approach taken by MobiEyes significantly decreases the load on the server side, making a server crash less probable. However, a crash on the server side is a serious issue for the system. Handling a server crash requires more than recovering the server state, because the state maintained persistently on the server side may not reflect the most recent updates at the server before the crash occurred. Thus the server state upon recovery may contain stale information. Since the algorithms for updating the server state during the normal operation of the system are mostly incremental, an effective way to handle a server failure is to use a failover solution through replicated servers [109]. In the absence of failover servers, a straight forward approach to handle a server crash is to re-initialize the whole system.

3.5 Experiments

In this section we describe three sets of simulation based experiments, which are used to evaluate our solution. The first set of experiments illustrates the scalability of the MobiEyes approach with respect to server load. The second set of experiments focuses on the messaging cost and studies the effects of several parameters on the messaging cost. We also give an analytical estimate of the messaging cost and compare it with the results from simulations. The third set of experiments investigates the amount of computation a mobile object has to perform, by measuring on average the number of queries a mobile object needs to process during each local evaluation period.

3.5.1 Simulation Setup

We list the set of parameters used in the simulation in Table 2. In all of the experiments presented in the rest of the chapter, the parameters take their default values if not specified otherwise. The area of interest considered in the simulation is a square shaped region of 300

Table 2: Simulation parameters

Parameter	Description	Value range	Default value
ts	Time step	30 seconds	
α	Grid cell side length	0.5-16 miles	5 miles
no	Number of objects	1,000-100,000	10,000
nmq	Number of moving queries	100-1,000	1,000
nmo	Number of objects changing velocity vector per time step	100-10,000	1,000
area	Area of consideration	300 x 300 square miles	
alen	Base station side length	5-80 miles	10 miles
qradius	Query radius	{3, 2, 1, 4, 5} miles	
qselect	Query selectivity	0.75	
mospeed	Max. object speed	{100, 50, 150, 200, 250} miles/hour	

x 300 square miles. The number of objects we consider ranges from 1,000 to 100,000 where the number of queries range from 100 to 1,000. These numbers can be scaled up without effecting the conclusions we draw from our experiments, as long as the object density is kept constant. The ratio of these parameters to one another closely follows the values used in [94].

We randomly select focal objects of the queries using a uniform distribution. The spatial region of a query is taken as a circular region whose radius is a random variable following a normal distribution. For a given query, the mean of the query radius is selected from the list {3, 2, 1, 4, 5}(miles) following a zipf distribution with parameter 0.8 and the std. deviation of the query radius is taken as 1/5th of its mean. The selectivity of the queries is taken as 0.75. This means that 75% of the objects satisfy the filter of a given query.

We model the movement of the objects as follows. We assign a maximum velocity to each object from the list {100, 50, 150, 200, 250}(miles/hour), using a zipf distribution with parameter 0.8. The default object speeds are set high in our experimental setup in order to stress test the algorithms. We also experiment with lower object speeds (that are more favorable against MobiEyes) by scaling the values in the default object speed list by a velocity factor smaller than 1. The simulation has a time step parameter of 30 seconds. In every time step we pick a number of objects at random and set their normalized velocity vectors to a random direction, while setting their velocity to a random value between zero and their maximum velocity. All other objects are assumed to continue their motion with their unchanged velocity vectors. The number of objects that change velocity vectors during each time step is a parameter whose value ranges from 100 to 10,000.

3.5.2 Server Load

In this section we compare our MobiEyes distributed query processing approach with two popular central query processing approaches, with regard to server load. The two centralized approaches we consider are indexing objects and indexing queries. Both are based on a central server on which the object locations are explicitly manipulated by the server logic as they arrive, for the purpose of answering queries. We can either assume that the objects are reporting their positions periodically or we can assume that periodically object locations are extracted from velocity vector and time information associated with mobile objects, on the server side. We first describe these two approaches and later compare them with the distributed MobiEyes distributed approach with regard to server load.

Indexing Objects: The first centralized approach to processing spatial continuous queries on mobile objects is by indexing objects. In this approach a spatial index is built over object locations. We use an R*-tree [16] for this purpose. As new object positions are received, the spatial index (the R*-tree) on object locations is updated with the new information. Periodically all queries are evaluated against the object index and the new results of the queries are determined. This is a straightforward approach and it is costly due to the frequent updates required on the spatial index over object locations. A better way to evaluate MQs is to use an index on queries instead of objects, as the number of queries is expected to be smaller than the number of objects.

Indexing Queries: The second centralized approach to processing spatial continuous queries on mobile objects is by indexing queries. In this approach a spatial index, again an R*-tree indeed, is built on moving queries. As the new positions of the focal objects of the queries are received, the spatial index is updated. This approach has the advantage of being able to perform differential evaluation of query results. When a new object position is received, it is run through the query index to determine to which queries this object actually contributes. Then the object is added to the results of these queries, and is removed from the results of other queries that have included it as a target object before.

We have implemented both the *object index* and the *query index* approaches for centralized processing of MQs. As a measure of server load, we took the time spent by the simulation for executing the server side logic per time step. Figure 30 and Figure 32 depict the results obtained. Note that the y-axises, which represent the sever load, are in log-scale. The x-axis represents the number of queries considered in Figure 30, and the different settings of α parameter in Figure 32.

It is observed from Figure 30 that the MobiEyes approach provides up to two orders of magnitude improvement on server load. In contrast, the object index approach has an almost constant cost, which slightly increases with the number of queries. This is due to the fact that the main cost of this approach is to update the spatial index when object positions change. Although the query index approach clearly outperforms the object index approach for small number of queries, its performance worsens as the number of queries increase. This is due to the fact that the main cost of this approach is to update the spatial index when focal objects of the queries change their positions. Our distributed approach also shows an increase in server load as the number of queries increase, but it preserves the relative gain against the query index.

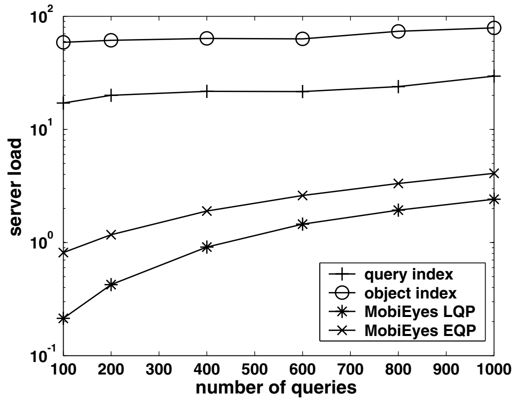


Figure 30: Impact of distributed query processing on server load

Figure 30 also shows the improvement in server load using lazy query propagation (LQP) compared to the default eager query propagation (EQP). However as described in Section 3.4, lazy query propagation may have some inaccuracy associated with it. Figure 31

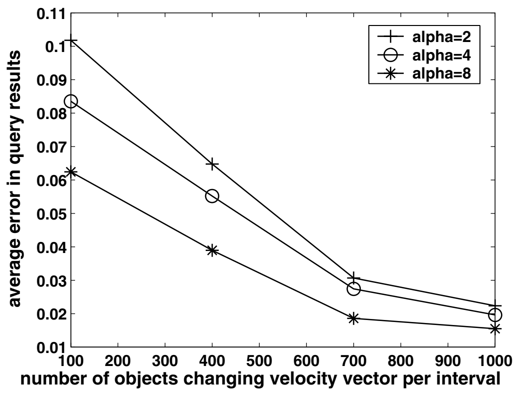


Figure 31: Error associated with lazy query propagation

studies this inaccuracy and the parameters that influence it. For a given query, we define the *error* in the query result at a given time, as the number of missing object identifiers in the result (compared to the correct result) divided by the size of the correct query result. Figure 31 plots the average error in the query results when lazy query propagation is used as a function of number of objects changing velocity vectors per time step for different values of α. Frequent velocity vector changes are expected to increase the accuracy of the query results. This is observed from Figure 31 as it shows that the error in query results decreases with increasing number of objects changing velocity vectors per time step. Frequent grid cell crossings are expected to decrease the accuracy of the query results. This is observed from Figure 31 as it shows that the error in query results increases with decreasing α.

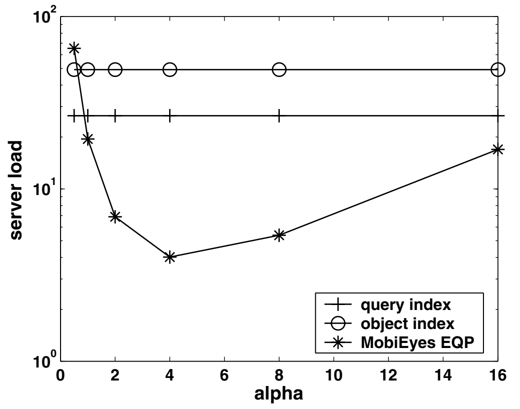


Figure 32: Effect of α on server load

Figure 32 shows that the performance of the MobiEyes approach in terms of server load worsens for too small and too large values of the α parameter. However it still outperforms the object index and query index approaches. For small values of α, the frequent grid cell changes increase the server load. On the other hand, for large values of α, the large monitoring areas increase the server's job of mediating between focal objects and the objects that are lying in the monitoring regions of the focal objects' queries. Several factors may affect the selection of an appropriate α value. We further investigate the problem of selecting a good value for α, through the use of an analytical model, in the next section.

3.5.3 Messaging Cost

In this section we discuss the effects of several parameters on the messaging cost of our solution. In most of the experiments presented in this section, we report the total number of messages sent on the wireless medium per second. The number of messages reported includes two types of messages. The first type of messages are the ones that are sent from a mobile object to the server (uplink messages), and the second type of messages are the ones broadcasted by a base station to a certain area or sent to a mobile object as a one-to-one message from the server (downlink messages). We evaluate and compare our results using two different scenarios. In the first scenario each object reports its position directly to the server at each time step, if its position has changed. We name this as the *naïve* approach. In the second scenario each object reports its velocity vector at each time step, if the velocity vector has changed (significantly) since the last time. We name this as the *central velocity* approach. This is the minimum amount of information required for a centralized approach to evaluate queries unless there is an assumption about object trajectories. Both of the scenarios assume a central processing scheme.

One crucial concern is defining an optimal value for the parameter α, which is the length of a grid cell. The graph in Figure 33 plots the number of messages per second as a function of α for different number of queries. As seen from the figure, both too small and too large values of α have a negative effect on the messaging cost. For smaller values of α this is because objects change their current grid cell quite frequently. For larger values of α this is mainly because the monitoring regions of the queries become larger. As a result, more

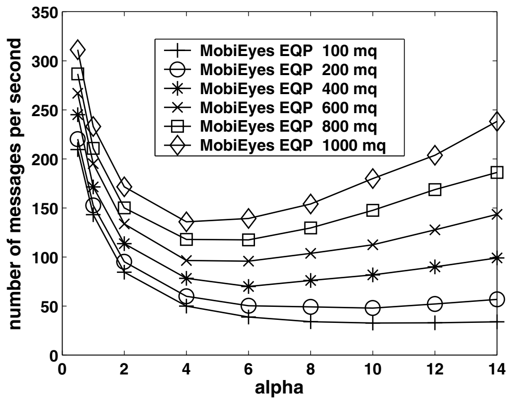


Figure 33: Effect of α on messaging cost

broadcasts are needed to notify objects in a larger area, of the changes related to focal objects of the queries they are subject to be considered against. Figure 33 shows that values in the range [4,6] are ideal for α with respect to the number of queries ranging from 100 to 1000. The optimal value of the α parameter can be derived analytically using a simple model (see Appendix B).

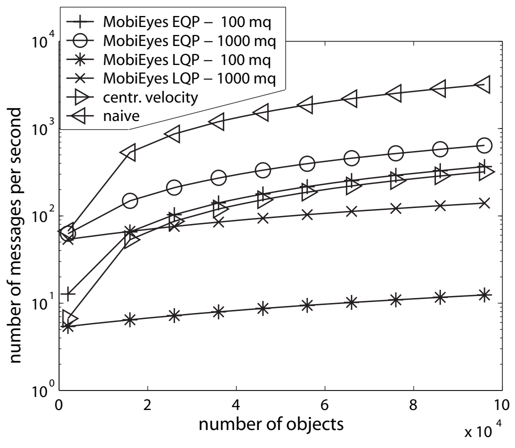


Figure 34: Effect of number of objects on messaging cost

Figure 34 studies the effect of number of objects on the messaging cost. It plots the number of messages per second (in logarithmic scale) as a function of number of objects for different numbers of queries. While the number of objects is altered, the ratio of the number of objects changing their velocity vectors per time step to the total number of objects is kept constant and equal to its default value as obtained from Table 2. It is observed that, when the number of queries is large and the number of objects is small, all approaches come close to one another, except the central velocity approach which provides lower messaging cost. When both the number of queries and the number of objects are small, again all approaches come close to one another, this time except the naÿe approach which incurs higher messaging cost. However, all approaches other than MobiEyes with LQP, have a high messaging cost when the ratio of the number of objects to the number of queries is high. On the other hand, MobiEyes with EQP shows similar scalability with the central velocity approach. MobiEyes with LQP scales better than all other approaches with increasing number of objects is because, it does not require non-focal objects to report their positions to the server.

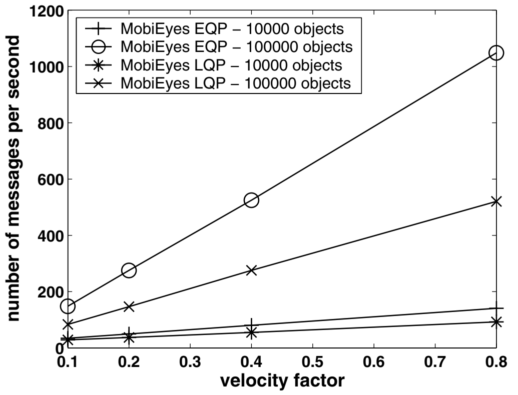


Figure 35: Effect of objects speeds on messaging cost

Figure 35 shows the messaging cost as a function of velocity factor for different numbers of objects. Note that the default object speeds are set high in our experimental setup in order to stress test the algorithms. In this experiment, velocity factor is used to scale down the object speeds. Figure 35 shows that the relatively high messaging cost of MobiEyes with EQP is mainly due to high object speeds, where the impact of object speeds on messaging cost is similar for MobiEyes with LQP although on a smaller scale. The main reason for MobiEyes with EQP to perform better with lower speeds is the decreased number of cell crossings for non-focal objects, which results in less communication with the server. We also observe from Figure 35 that the improvement in messaging cost with decreasing object speeds is more prominent for large number of objects.

Figure 36 studies the effect of number of objects changing velocity vector per time step

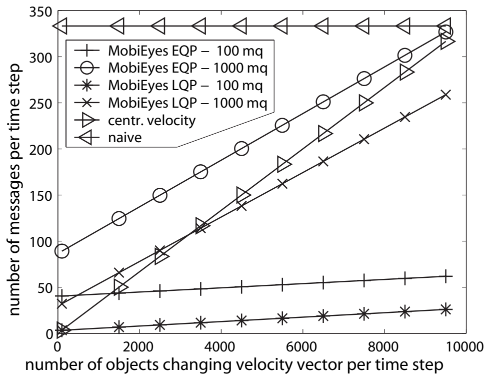


Figure 36: Effect of number of objects changing velocity vector on messaging cost

on the messaging cost. It plots the number of messages per second as a function of the number of objects changing velocity vector per time step for different numbers of queries. An important observation from Figure 36 is that the messaging cost of MobiEyes with LQP scales well when compared to the central velocity approach, since the gap between the two increases as the number of objects changing velocity vector per time step increases. Note that the central velocity approach degrades to the naïve approach when all objects change their velocity vectors at each time step. MobiEyes with EQP performs better than the central velocity approach only for small number of queries and large number of objects changing velocity vector per time step.

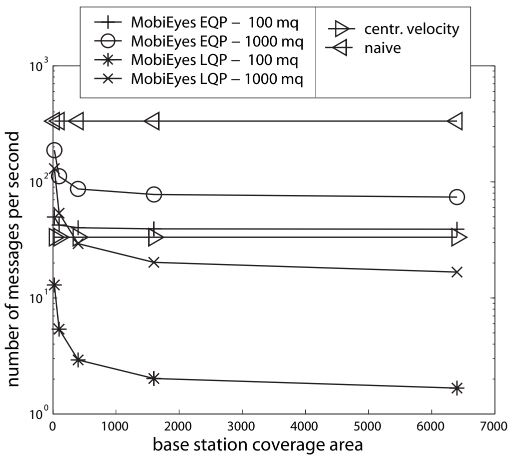


Figure 37: Effect of base station coverage area on messaging cost

Figure 37 studies the effect of base station coverage area on the messaging cost. It plots the number of messages per second as a function of the base station coverage area for different numbers of queries. It is observed from Figure 37 that increasing the base station coverage decreases the messaging cost up to some point after which the effect disappears. The reason for this is that, after the coverage areas of the base stations reach to a certain size, the monitoring regions associated with queries always lie in only one base station's coverage area. Although increasing base station size decreases the total number of messages sent on the wireless medium, it will increase the average number of messages received by a mobile object due to the size difference between monitoring regions and base station coverage areas. In a hypothetical case where the universe of disclosure is covered by a single base station, any server broadcast will be received by any mobile object. In such environments, indexing on the air [63] can be used as an effective mechanism to deal with this problem. We do not consider extreme scenarios like satellite broadcast and focus on cellular networks.

In MobiEyes, a large base station coverage area gives less effective results only when the total number of mobile objects in the region of coverage is large. Because, there will be large number of focal objects in the region of coverage and many of the broadcasted updates originating from these focal objects will be discarded by a large number of mobile objects due to the mismatch between the monitoring region sizes and the base station coverage area sizes (the latter being much larger). However, large base station coverage areas make very poor use of the frequency spectrum and are not suitable for hot spot regions. Such large coverage areas are more common in sparsely populated regions where the total number of mobile nodes is relatively small. This nature of base station coverage area sizes in cellular networks is very favorable for the MobiEyes system. For recent 3G cellular network technologies such as CDMA200 1xEV-DO [98], base station coverage areas in urban settlements are small, in general not larger than 3 miles radius.

We have evaluated the scalability of the MobiEyes in terms of the total number of messages exchanged in the system and the reduction of the server load of MobiEyes approach. Now we study the per object power consumption due to communication between mobile objects and the server. We measure the average communication related to power consumption using a simple radio model where the transmission path consists of transmitter electronics and transmit amplifier where the receiver path consists of receiver electronics. Considering a GSM/GPRS device [66], we take the power consumption of transmitter and receiver electronics as 150mW and 120mW respectively and we assume a 300mW transmit amplifier with 30% efficiency [66]. We consider 14kbps uplink and 28kbps downlink bandwidth (typical for current GPRS technology). Note that sending data is more power consuming than receiving data. [3]

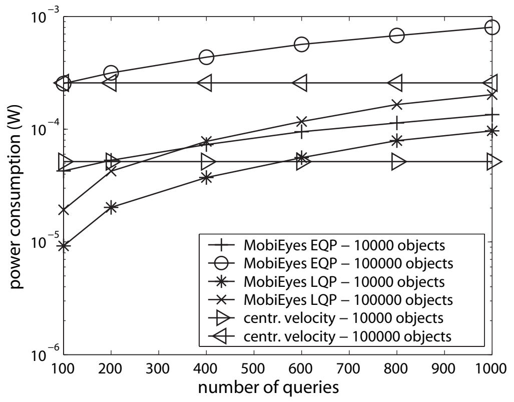


Figure 38: Effect of # of queries on object power consumption due to communication

We simulated the MobiEyes approach using message sizes instead of message counts for messages exchanged and compared its power consumption due to communication with the naive and central velocity approaches. The graph in Figure 38 plots the per object power consumption due to communication (on logarithmic y-axis) as a function of number of queries for different numbers of mobile objects. Since the naive approach requires every object to send its new position to the server, its per object power consumption is the worst, thus it is not included in the comparison. In MobiEyes, however, a non-focal object does not send its position or velocity vector to the server, but it receives query updates from the server. Since the cost of receiving data in terms of energy consumption is lower than transmitting, MobiEyes is expected to be effective in terms of per object power consumption. It is observed from Figure 38 that, except for the case when the number of objects is small and the number of queries is large, the central velocity approach is outperformed by MobiEyes with LQP. MobiEyes with LQP performs especially well when the number of

[3]In this setting transmitting costs $\sim 80\mu jules/bit$ and receiving costs $\sim 5\mu jules/bit$

objects is large. On the other hand, MobiEyes with EQP performs worse than the central velocity approach, with the exception of cases where the number of queries is small. An important factor that increases the per object power consumption in MobiEyes is the fact that an object also receives updates regarding queries that are irrelevant. This is mainly due to the difference between the size of a broadcast area and the monitoring region of a query.

3.5.4 Amount of Computation on Mobile Object Side

In this section we study the amount of computation placed on the mobile object side by the MobiEyes approach to processing of MQs. One measure of this is the number of queries a mobile object has to evaluate at each time step, which is the size of the LQT table (Recall Section 3.3.1).

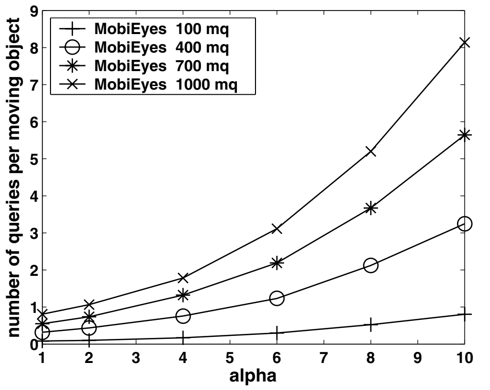


Figure 39: Effect of α on the avg. # of queries evaluated per step on a mobile object

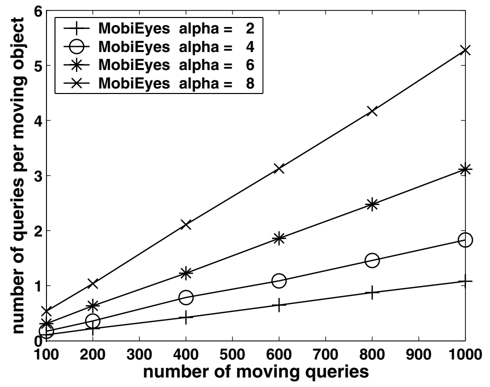


Figure 40: Effect of the total # of queries on the # of queries evaluated on a mobile object

Figure 39 and Figure 40 study the effect of α and the effect of the total number of queries on the average number of queries a mobile object has to evaluate at each time step (average LQT table size). The graph in Figure 39 plots the average LQT table size as a function of α for different number of queries. The graph in Figure 40 plots the same measure, but this time as a function of number of queries for different values of α. The first observation from these two figures is that the size of the LQT table does not exceeds 10 for the simulation setup. The second observation is that the average size of the LQT table increases exponentially with α where it increases linearly with the number of queries.

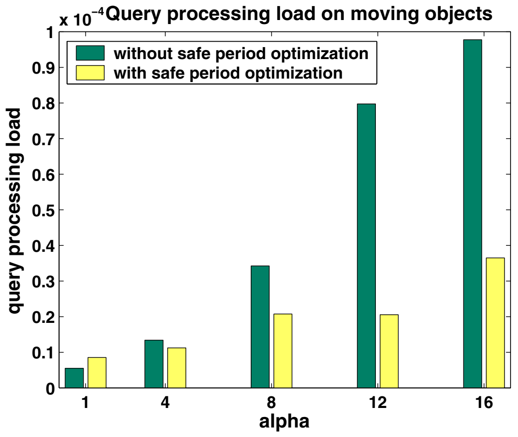


Figure 41: Effect of the safe period opt. on the query processing load of a mobile object

Figure 41 studies the effect of the safe period optimization on the average query processing load of a mobile object. The x-axis of the graph in Figure 41 represents the α parameter, and the y-axis represents the average query processing load of a mobile object. As a measure of query processing load, we took the average time spent by a mobile object for processing its LQT table in the simulation. Figure 41 shows that for large values of α, the safe period optimization is very effective. This is because, as α gets larger, monitoring regions get larger, which increases the average distance between the focal object of a query and the objects in its monitoring region. This results in non-zero safe periods and decreases the cost of processing the LQT table. On the other hand, for very small values of α, like $\alpha = 1$ in Figure 41, the safe period optimization incurs a small overhead. This is because the safe period is almost always less than the query evaluation period for very small α values and as a result the extra processing done for safe period calculations does not pay off.

3.6 Related Work

Real-time evaluation of *static* spatial queries on mobile objects, at a centralized location, is a well studied topic. Most of the work done so far has focused on efficient indexing structures [94, 105, 118] for efficient evaluation of static continual range queries at a central location, or indexing schemes and algorithms for handling mobile object positions [25, 74, 119, 72, 1, 17, 120]. Very few have considered the benefits of a careful tradeoff between computation and communication. To our knowledge, only the SQM system introduced in [25, 24] has proposed a distributed solution for evaluation of static spatial queries on mobile objects, that makes use of the computational capabilities present at the mobile objects.

Several existing research efforts on mobile object databases [131, 105, 132, 72] model mobile object trajectories as piecewise linear functions of time, and process these less frequently changing functions instead of more frequently changing object positions. This is commonly viewed as an important strategy for efficiently processing queries on mobile object positions. The design of the MobiEyes system also uses such a linear model to ease analytical derivations and engineering issues of the system. Concretely, in MobiEyes the use of velocities for predicting the positions of objects that are of interest to a mobile object, on the mobile object side, is more similar to the use of dead reckoning in on-line games and PDES [43] systems for building distributed virtual environments. There are very few attempts to use non-linear motion modeling in mobile object databases [2]. A discussion on whether linear modeling assumptions can easily carry out in practice and its possible implications can be found in [131].

Safe period optimization described in Section 3.4 is inspired by the safe region optimization introduced in [94]. However, there are some major differences: A safe region is calculated for an object considering all queries, so that the object is guaranteed to stay outside of all query regions as long as it resides in the safe region. Also safe regions are introduced for static range query evaluation at a centralized server. In contrast, a safe period is calculated for an object considering a single query, so that the object is guaranteed to reside outside the query region during the safe period. Compared to safe region, the safe period optimization is a more focused local optimization technique, which is computed at each mobile object that belong to a restricted subset, namely the mobile objects that reside within the monitoring region of a MQ.

The work presented in [61] deals with the problem of monitoring changes in sensor readings and detecting mobile phenomena in sensor networks. Since the sensed phenomena may move in the environment, the set of nodes that detect this phenomena also change or "move". The queries employed in [61] are similar to queries used in our work, in the sense that the nodes in the result set may change continuously. However, we must emphasize that in MobiEyes, queries (as well as target objects in the query results) can have the property of being mobile. This is closely related with the concept of focal objects, which is missing in [61].

Recently there has been works that explicitly support efficient evaluation of moving queries over moving objects. The most notable are SINA [87] and MAI (see Chapter 4).

SINA [87] is a spatial query evaluation engine that makes use of the "incremental query processing" concept. The periodic re-evaluations are achieved through a three phase process that refreshes the previous query results based on the current position changes using positive and negative updates, as opposed to re-computing all of the possibly invalidated results from scratch. The three phases of the query re-evaluation are hashing, invalidation, and joining. The hashing phase uses a grid for indexing purposes, which is a server side data structure. This is very different than the use of grid and grid cells in our system. MobiEyes utilizes grid cells to facilitate the partitioning of moving query evaluation into a distributed coordination of server side processing and mobile object side processing. The concept of monitoring region relies on grid cells to define the set of nodes that should register a query as a nearby query and process it locally. The dynamics of the system, such as updating the set of nearby queries registered at mobile nodes, are handled with the help of the grid.

MAI is a motion adaptive indexing scheme for time and IO efficient evaluation of moving queries over moving objects. MAI uses the concept of motion-sensitive bounding boxes to model moving objects and moving queries. These bounding boxes automatically adapt their sizes to the dynamic motion behaviors of individual objects. Instead of indexing frequently changing object positions, MAI indexes less frequently changing object and query motion-sensitive bounding boxes, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. MAI also uses predictive query results to optimistically pre-calculate query results. Motion-sensitive bounding boxes are used to incrementally update the predictive query results.

Both SINA and MAI are centralized processing based approaches, and require to receive position updates (raw positions in SINA and velocity vector changes in MAI) from all mobile nodes. This means that communication cost does not change significantly with the specific type of centralized processing engine employed. Second, since MobiEyes distributes the job of processing queries to mobile nodes of the system, the scalability in terms of server load will be drastically better than any centralized processing based solution.

CHAPTER IV

MAI - CENTRALIZED LOCATION MONITORING IN MOBILE SYSTEMS USING CQS

This chapter describes a *motion adaptive* indexing scheme for efficient evaluation of moving continual queries (MCQs) over moving objects. It uses the concept of *motion-sensitive bounding boxes* (MSBs) to model moving objects and moving queries. These bounding boxes automatically adapt their sizes to the dynamic motion behaviors of individual objects. Instead of indexing frequently changing object positions, we index less frequently changing object and query MSBs, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. This helps decrease the number of updates to the indexes. More importantly, we use *predictive query results* to optimistically pre-calculate query results, decreasing the number of searches on the indexes. Motion-sensitive bounding boxes are used to incrementally update the predictive query results. Furthermore, we introduce the concepts of *guaranteed safe radius* and *optimistic safe radius* to extend our motion adaptive indexing scheme to evaluating moving continual *k-nearest neighbor* (kNN) queries. Our experiments show that the proposed motion adaptive indexing scheme is efficient for the evaluation of both moving continual range queries and moving continual kNN queries.

4.1 Introduction

With the continued advances in mobile computing and positioning technologies, such as GPS [127], location management has become an active area of research. Several research efforts have been made to address the problem of indexing moving objects or moving object trajectories to support efficient evaluation of continual spatial queries. Our focus in this chapter is on *moving continual queries over moving objects* (MCQs for short). There are two major types of MCQs – *moving continual range queries* and *moving continual k-Nearest*

Neighbor queries.

Efficient evaluation of MCQs is an important issue in both mobile systems and moving object tracking systems. Research on evaluating range queries over moving object positions has so far focused on static continual range queries [94, 67, 25]. A static continual range query specifies a spatial range together with a time interval and tracks the set of objects that locate within this spatial region over the given time period. The result of the query changes as the objects being queried move over time. Although similar, a moving continual range query exhibits some fundamental differences when compared to a static continual range query. A moving continual range query has an associated moving object, called the *focal object* of the query; the spatial region of the query moves continuously as the query's focal object moves. Moving continual queries introduce a new challenge in indexing, mainly due to the highly dynamic nature of both queries and objects.

MCQs have different applications, such as environmental awareness, object tracking and monitoring, location-based services, virtual environments and computer games, to name a few. Here is an example of a moving continual query MCQ_1: "Give me the positions of those customers who are looking for taxi and are within 5 miles (of my location at each instant of time or at an interval of every minute) during the next 20 minutes," posted by a taxi driver on the road. The focal object of MCQ_1 is the taxi on the road. Another example is MCQ_2: "Give me the number of friendly units within 5 miles radius around me during the next 2 hours," posted by a soldier equipped with mobile devices marching in the field, or a moving tank in a military setting. The focal object of MCQ_2 is the soldier marching in the field or the moving tank.

Different specializations of MCQs can result in interesting classes of MCQs. One is called *moving continual queries over static objects*, where the target objects are stationary objects in the query region. An example of such a query is MCQ_3: "Give me the locations and names of the gas stations offering gasoline for less than $1.2 per gallon within 10 miles, during the next half an hour," posted by a driver of a moving car, where the focal object of the query is the car on the move and the target objects are the gas stations within 10 miles with respect to the location of the car. Another interesting specialization is the so called *static continual queries over moving objects*, where the queries are posed with static focal objects or without focal objects. An example query is MCQ_4: "Give me the list of AAA vehicles that are currently on service call in downtown Atlanta (or 5 miles from my office location), during the next hour." Note that these specializations of MCQs are computationally easier to evaluate. Our focus in this chapter is the evaluation of MCQs in their most general form, such as MCQ_1 and MCQ_2.

Due to frequent updates to the index structures, traditional indexing approaches built on moving object positions generally do not work well for MCQs [94, 67]. In order to tackle this problem, several researchers have introduced alternative approaches based on the idea of indexing the parameters of the motion functions of the moving objects [72, 105, 118, 2]. They effectively alleviate the problem of frequent updates to the indexes, as the indexes need to be updated only when the parameters change. These approaches are mostly based on R-tree-like structures and produce time parameterized minimum bounding rectangles that enlarge continuously [105, 118, 94]. As a consequence of enlarged bounding rectangles, the search performance can deteriorate over time and the index structures may need to be reconstructed periodically [94, 105]. As far as update costs are concerned, approaches based on time parameterized rectangles [105, 118] can provide excellent performance. However, they are not *sufficient* for processing MCQs. This is because they do not support incremental re-evaluation of queries and the **continual** nature of these queries dictates that the same queries must be re-evaluated at frequent intervals. Thus, there is a need for new methods that can evaluate these MCQs incrementally.

In this chapter, we describe a *motion-adaptive indexing* (MAI) scheme for efficient processing of moving continual queries over moving objects. It uses the concept of *motion-sensitive bounding boxes* (MSBs) to model both moving objects and moving queries. Instead of indexing frequently changing object positions, we index less frequently changing object and query MSBs, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. This helps decrease the number of updates performed on the indexes. However, the main use of MSBs is to facilitate incremental processing of MCQs. We provide two techniques to reduce the costs of query re-evaluation and search on the MSB indexes. First, we optimistically pre-calculate query results and incrementally maintain such *predictive query results* under the presence of object motion changes. MSBs are used to control the amount of pre-computation to be performed for calculating the predictive query results and to decide when the results need to be updated. Second, we support *motion adaptive* indexing. We automatically adapt the sizes of MSBs to the changing moving behaviors of the corresponding individual objects. By adapting to moving object behavior at the granularity of individual objects, the moving queries can be evaluated faster by performing fewer IOs. Furthermore, we extend the MAI approach to the evaluation of moving continual *k-nearest neighbor* queries, by introducing the concepts of *guaranteed safe radius* and *optimistic safe radius* that are used to leverage the moving continual range queries for answering kNN queries.

Another interesting contribution of this chapter is the development of an analytical model for estimating the cost of moving query evaluation, and the use of analytical models to guide the setting and the adaptation of several system parameters for our proposed indexing scheme. The proposed motion-adaptive indexing scheme is independent of the underlying spatial index structures by design. In the experiments reported in this chapter, we use both R*-trees and statically partitioned grids for measuring the performance of our indexing scheme. Our experimental results show that the motion adaptive indexing scheme is efficient for the evaluation of both moving continual *range* queries and moving continual *k-nearest neighbor* queries. We report a series of experimental performance results for different workloads including scenarios based on skewed object and query distribution, and demonstrate the effectiveness of our motion adaptive indexing scheme through comparisons with other alternative indexing approaches.

4.2 The System Model

The basic elements of our system model are a set of moving or stationary objects and a set of moving or static continual (range or kNN) queries. A fundamental challenge we address in this chapter is to study what kind of indexing scheme can efficiently answer the moving queries. Fast evaluation is critical for processing moving queries, as it not only improves the freshness of the query results by enabling more frequent re-evaluation, but also increases the scalability of the system by enabling timely evaluation of a large number of moving queries over a large number of moving objects.

4.2.1 Basic Concepts and Problem Statement

We denote the set of moving or stationary objects as O, where $O = O_m \cup O_s$ and $O_m \cap O_s = \emptyset$. O_m denotes the set of moving objects and O_s denotes the set of stationary objects. We denote the set of moving or static queries as Q, where $Q = Q_m \cup Q_s$ and $Q_m \cap Q_s = \emptyset$. Q_m denotes the set of moving continual range queries and Q_s denotes the set of static continual range queries. Since we focus on moving continual queries in this chapter, from now on we use moving queries and moving continual queries interchangeably.

Moving Objects. We describe a moving object $o_m \in O_m$ by a quadruple: $\langle i_o, \vec{p}, \vec{v}, a_p \rangle$. Here, i_o is the unique object identifier, $\vec{p} = (p_x, p_y)$ is the current position of the moving object where p_x is its position in the x-dimension and p_y is its position in the y-dimension, $\vec{v} = (v_x, v_y)$ is the current velocity vector of the object, and a_p is a set of properties about the object. A stationary object can be modeled as a special case of moving object where the velocity vector is set to zero, $\forall o_s \in O_s, o_s.\vec{v} = (0, 0)$.

Moving Queries. We describe a moving query $q_m \in Q_m$ by a quadruple: $\langle i_q, i_o, r, f \rangle$. Here, i_q is the unique query identifier, i_o is the object identifier of the focal object of the query, r defines the shape of the spatial query region bound to the focal object of the query, and f is a Boolean predicate, called *filter*, defined over the properties (a_p) of the target objects of the query. Note that, r can be described by a closed shape description such as a rectangle or a circle. This closed shape description also specifies a binding point, through which it is bound to the focal object of the query. In the rest of the chapter we assume that a moving continual query specifies a circle as its range with its center serving as the binding point and we use r to denote the radius of the circle. A static spatial continual range query can be described as a special case where the query either has no focal object or the focal object is a stationary object. Namely, $\forall q_s \in Q_s, q_s.i_o = \mathit{null} \vee q_s.i_o \in O_s$. We assume that a static continual range query specifies a rectangle or a circle as its range.

Before we give an overview of our approach, we first review three basic types of indexing techniques for evaluating moving range queries over moving objects and discuss their advantages and inherent weaknesses.

Object-only Indexing (*OI*). In the object-only indexing approach, a spatial index is built on the object positions. Each time a new object position is received, the object index is updated. At each query evaluation phase, all queries are evaluated against the object index. An inherent drawback of the basic object-only indexing approach is the re-evaluation of all queries against the object index regardless of whether or not the object position changes are of interest to the query. Object-only indexing is open to optimizations that can decrease the number or cost of the updates on the object index (see velocity constrained indexing in [94] and TPR-trees in [105]).

Query-only Indexing (*QI*). In the query-only indexing approach, a spatial index is built on the spatial regions of the queries. Each time a new query position (the position of the query's focal object) is received, the query index is updated. At each query evaluation phase, each object position is evaluated against the query index and the queries that contain the object's position are determined. Note that this has to be done for every object as opposed to doing it only for objects that have moved since the last query evaluation phase. This is due to the fact that underlying queries are potentially moving. This significantly decreases the effectiveness of query-only indexing approach, although in the context of static continual range queries it has been shown that a query index may improve performance significantly [94, 133, 134].

Object and Query Indexing (*OQI*). In the object and query indexing approach, two spatial indexes are built, one for the object positions and another for the spatial regions of the queries. Each time an object position is received, the object index is updated. Similarly, each time a new query position (the position of a query's focal object) is received, the query index is updated. At each query evaluation phase, each *new* object position is evaluated against the query index and the queries that contain the object's position are determined. Then the query results are updated differentially. Similarly at each query evaluation phase, each *new* query position is evaluated against the object index and the new result of the query is determined. The OQI approach evaluates object positions against the query index only for those objects that have changed their positions since the last query evaluation phase, as opposed to all the objects required by the query-only indexing approach. The OQI approach also evaluates queries against the object index only for those queries that have moved since the last query evaluation phase, as opposed to all the queries required by the object-only indexing approach. Although the OQI approach incurs a higher cost due to the maintenance of an additional index structure, it is open to a wider range of optimizations to reduce the cost and it does not have certain restrictions of the object-only indexing or query-only indexing approach.

4.2.2 Overview of the Proposed Solution

Cognizant of the pros and cons of the above three basic indexing schemes, we propose a motion-adaptive indexing scheme for efficient processing of moving queries over moving objects. We use the concept of *motion-sensitive bounding boxes* to model the dynamic behavior of both moving objects and moving queries. Such bounding boxes are not updated unless the position of a moving object or the spatial region of a moving query exceeds the borders of its bounding box. Instead of indexing frequently changing object positions or spatial regions of moving queries, we index less frequently changing motion sensitive bounding boxes. This significantly decreases the number of update operations performed on the indexes. Our indexing scheme maintains both an index of object-based motion sensitive bounding boxes (denoted as $Index_o^{msb}$) and an index of query-based motion sensitive bounding boxes (denoted as $Index_q^{msb}$).

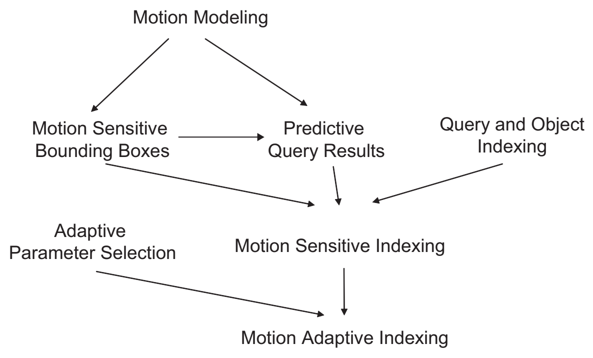

Motion Modeling
Motion Sensitive Bounding Boxes
Predictive Query Results
Query and Object Indexing
Adaptive Parameter Selection
Motion Sensitive Indexing
Motion Adaptive Indexing

Figure 42: Roadmap of methods applied for moving query evaluation

More importantly, to address the problem of increased search cost due to frequent evaluation of queries, we employ two optimization techniques: (i) *predictive query results* and (ii) *motion adaptive indexing*. Query results are optimistically precomputed in the presence of object motion changes, with the amount of pre-computation to be performed controlled by the motion sensitive bounding boxes. The sizes of the motion sensitive bounding boxes are dynamically adapted to the changing motion behaviors at the granularity of individual objects, allowing moving queries to be evaluated faster by performing fewer IOs. Figure 42 gives a roadmap of methods applied for MCQ evaluation.

In the rest of this section we describe the motion modeling and motion update generation, which provides the foundation for *motion sensitive bounding boxes* and *predictive query results*.

Motion Modeling. Modeling motions of the moving objects for predicting their positions is a commonly used method in moving object indexing [130, 72]. In reality, a moving object moves and changes its velocity vector continuously. Motion modeling uses approximation for prediction. Concretely, instead of reporting their position updates each time they move, moving objects report their velocity vector and position updates only when their velocity vectors change and this change is significant enough (This technique is known as dead reckoning [43]). In order to evaluate moving queries in between the last update reporting and the next update reporting, the positions of the moving objects are predicted using a simple linear function of time. Given that the last received velocity vector of an object is $\vec{v}$, its position is $\vec{p}$ and the time its velocity update was recorded is t, the future position of the object at time $t + \Delta t$ can be predicted as $\vec{p} + \Delta t * \vec{v}$. We use a linear motion function in this chapter, since it is the commonly used model in moving object databases [131]. We refer readers to [2] for a study of non-linear motion modeling for moving object indexing.

Prediction-based motion modeling decreases the amount of information sent to the query processing engine by reducing the frequency of position reporting from each moving object. Furthermore, it allows the system to optimistically precompute future query results. We below briefly describe how the moving objects generate and send their motion updates to the server where the query evaluation is performed.

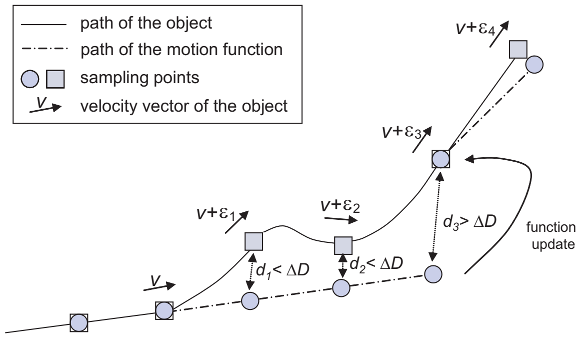


Figure 43: Motion update generation

Motion Update Generation. In order for the moving objects to decide when to report their velocity vector and position updates, they need to periodically compute if their velocity vectors have changed significantly. Concretely, at each time step a moving object samples its current position and calculates the difference between its current position and the position predicted by the dead reckoning algorithm based on the last motion update it reported to the server. In case this difference is larger than a specified threshold, say ΔD, the new motion function parameters are relayed to the server. Figure 43 provides an illustration. The path of a moving object is depicted with a solid line, where its path predicted by the server is depicted with a dashed line. The small squares on the solid line represents the current positions sampled by the moving object at each time step and the small circles on the dashed line represent the positions that the server predicts the object to be at in each of the corresponding time steps.

4.3 Efficient Evaluation of Moving Continual Range Queries

In this section, we describe the motion adaptive indexing scheme for efficient processing of moving range queries over moving objects. We first describe the concept of motion-sensitive bounding boxes, and then discuss the mechanisms used for computing predictive query results, and outline the motion adaptive approach for determining the sizes of motion sensitive bounding boxes. In addition, we provide an overview of the algorithms used for creating and maintaining the motion adaptive indexes, an analytical model for IO estimation, and the concrete mechanism that adaptively determines the bounding box sizes based on the dynamically changing motion behaviors of moving objects and moving queries.

4.3.1 Motion Sensitive Bounding Boxes

Motion sensitive bounding boxes (MSBs) can be defined for both moving queries and moving objects. Given a moving object o_m, its associated MSB is calculated by extending the position of the object along each dimension by $\alpha(o_m)$ times the velocity of the object in that direction. Given a moving query q_m, the MSB of the moving query is calculated by extending the minimum bounding box of the query along each dimension by $\beta(q_m)$ times the velocity of the focal object of the query in that direction (See Figure 44 for illustrations).

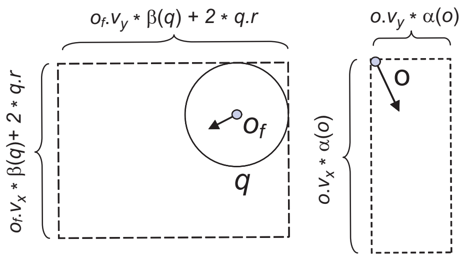

$o_f.v_y * \beta(q) + 2 * q.r$
$o.v_y * \alpha(o)$
$o_f.v_x * \beta(q) + 2 * q.r$
o_f
q
$o.v_x * \alpha(o)$
o

Figure 44: Motion sensitive bounding boxes, MSBs

Let $Rect(l, m)$ denote a rectangle with l and m as any two end points of the rectangle that are on the same diagonal. Let $sign(\vec{x})$ denote a function over a vector $\vec{x}$, which replaces each entry in $\vec{x}$ with 1 if it is greater than or equal to 0, with -1 otherwise. Then we define the MSB for a moving object o and the MSB for a moving query q with focal object o_f as follows:

$$\begin{aligned}
\forall o \in O_m, MSB(o) &= Rect(o.pos, o.pos + \alpha(o) * o.vel) \\
\forall q \in Q_m, MSB(q) &= Rect(o_f.pos - q.radius * w_s, o_f.pos + \\
&\quad \beta(q) * q.vel + q.radius * w_s), \\
&\quad \text{where } w_s \text{ denotes the sign function } sign(q.vel)
\end{aligned}$$

For each moving query, its MSB is calculated and used in place of the query's spatial region in the query-based MSB index, that is $Index_q^{msb}$. Similarly, for each moving object, its MSB is calculated and used in place of the object's position in the object-based MSB index, that is $Index_o^{msb}$.

An important feature of indexing motion sensitive boxes of moving objects and moving queries is the fact that an MSB is not updated unless the query's spatial region or the object's position exceeds the borders of its motion sensitive bounding box. When this happens, we need to invalidate the MSB. As a result, a new MSB is calculated and the $Index_q^{msb}$ or the $Index_o^{msb}$ is updated. This approach reduces the number of update operations performed on the spatial indexes and thus decreases the overall cost of updating the spatial indexes ($Index_o^{msb}$ and $Index_q^{msb}$). It is also crucial to note that, using MSBs does not introduce any inaccuracy in the query results, because *we store the motion function of the object or the query together with its MSB inside the spatial index.*

Although maintaining MSB indexes increase the cost of searching the index due to higher overlap of spatial objects being indexed, for appropriate values of the α and β parameters, the overall gain in the search cost due to the use of MSBs is significant, thanks to the incremental processing capabilities MSBs provide in conjunction with predictive query results. Concretely, when a query has not invalidated its MSB and has not changed its velocity vector, then the predictive results of the query are valid with regard to the objects for which no MSB invalidations or velocity vector changes has taken place. In case some of the objects had MSB invalidations or velocity vector changes, queries are not completely re-evaluated. A query is completely re-evaluated only when it has invalidated its MSB or it has changed its velocity vector. We will discuss the details of query evaluation in greater depth in Section 4.3.3. In summary, the incremental processing of queries helps minimize the overall search cost and compensates for the small increase in the per operation index search cost due to the use of MSBs. Table 3 summarizes the impact of using MSBs on the query evaluation in terms of update and search cost.

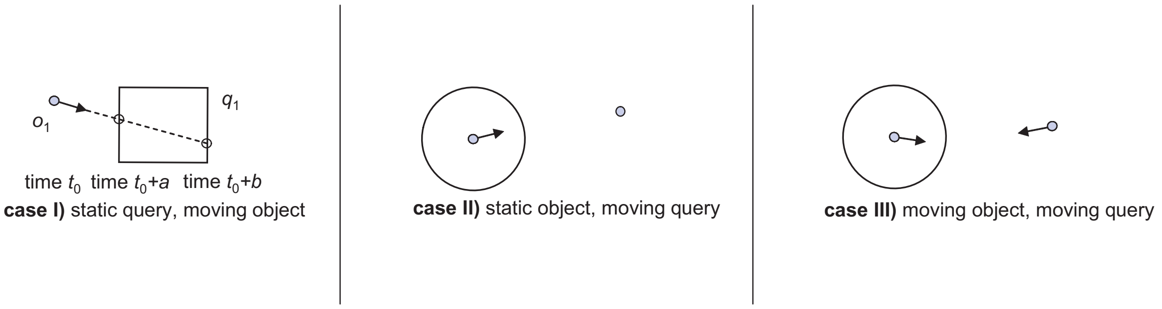
Table 3: Impact of MSBs and predictive query results on query evaluation cost

technique	Overall Update Cost		Overall Search Cost	
MSBs	↓ (due to less frequent updates to the indexes)		↑ (due to increased overlap in indexes)	
Predictive Query Results (using MSBs)	–		↓ (due to incremental query evaluation)	
Together	+	↓	+	↓

Furthermore, MSBs provide the following three advantages: (1) As opposed to approaches that alter the implementation of traditional spatial indexes for decreasing the update cost (like TPR-tree [105] or VCI index [94]), motion sensitive bounding boxes require almost no significant change to the underlying spatial index implementation. (2) They form a basis for deciding for which objects to pre-calculate query results with respect to a query (see Section 4.3.3). (3) By performing size adaptation at the granularity of individual objects, they lead to significant reductions in IO cost (see Section 4.3.4). In order to fully utilize the advantages made possible by MSBs in terms of query evaluation cost, we need mechanisms for dynamically determining the most appropriate values of the α and β parameters based on the motion behavior of moving objects and moving queries.

4.3.2 Predictive Query Results on a Per Object Basis

It is well known that one way of reducing IO and improving efficiency of evaluating moving queries is to pre-calculate future results of the continual queries. This approach has been successfully used in the context of continual moving kNN queries over *static* objects [120]. Most of existing approaches to pre-calculating query results associate a time interval to each query that specifies the valid time for the pre-calculated results. One problem with per query based prediction in the context of moving queries over moving objects is the fact that a change on the motion function of any of the moving objects may cause the invalidation of some of the pre-calculated results. This motivates us to introduce *predictive query results* where the prediction is conducted on per-object basis.



o_1
q_1
time t_0 time t_0+a time t_0+b
case I) static query, moving object
case II) static object, moving query
case III) moving object, moving query

Figure 45: Calculating the valid prediction time intervals

Given a query, its predictive query result differs from a regular query result in the sense that each object in the predictive query result has an associated time interval indicating the time period in which the object is *expected* to be included in the query result. We denote the predictive query result of query $q \in Q$ by $PQR(q)$. Each entry in a predictive query result takes the form $\langle o, [t_s, t_e] \rangle$. We call the entry associated with object $o \in O$ in $PQR(q)$ the *predictive query result entry* of object o with regard to query q, and the interval $[t_s, t_e]$ associated with object o the *valid prediction time interval* of the predictive query result entry.

Calculating the valid prediction time intervals is done as follows. Given a static continual range query and a moving object with its motion function, it is straightforward to calculate the intersection points of the query's spatial region and the ray formed by the moving object's trajectory (See case I in Figure 45). Similarly, to calculate the intersection point of a moving query and a moving or non-moving object (assuming that we only consider moving queries with circle shaped spatial regions), we need to solve a quadratic function of time. Formally, let $q \in Q$ be a query with focal object $o_f \in O_m$, and $o \in O$ be an object, and let $Dist(a, b)$ denote the Euclidean distance between the two points a and b. We can calculate the time interval in which the object o is expected to be in the result set of query q by solving the formula: $Dist(o_f.\vec{p} + t * o_f.\vec{v}, o.\vec{p} + t * o.\vec{v}) \leq q_m.r$. Figure 45 illustrates three different cases that arise in the calculation of the prediction time interval for each per-object based predictive query result entry.

The predictive query results are pre-calculated on per object basis and the result entries are correct unless the motion function of the focal object of a query or the motion function of the moving object associated with the query result entry have changed within the valid prediction time interval. As a result, there are two key questions to answer in order to effectively use the predictive query results in evaluating MCQs:

Prediction – *For each moving query, should we perform prediction on all moving objects? If not, how to determine for which objects we should do prediction?*
Obviously we should not perform prediction for objects that are far away from the spatial region of the query within a period of time, as the predicted results are less likely to hold until those objects reach to the proximity of the query.

Invalidation – *When and how to update the predictive results?*
This can be referred to as the invalidation policy for per-object based prediction. The

predictive query results may be invalid and thus need to be updated when the motion function of a moving query or the motion function of a moving object changes. In addition, the predictive results may require to be refreshed when the objects in the predictive query results have moved away from the proximity of the query or when the objects that did not participate in the prediction have entered the proximity of the query.

4.3.3 Determining Predictive Query Results Using MSBs

MSBs are used to effectively determine for which objects we should perform result prediction with respect to a query (answering the first question listed in Section 4.3.2). Concretely, for a given query, objects whose MSBs intersect with the query's MSB are considered as potential candidates of the query's predictive result. Figure 46 gives an illustration of how predictive query results integrate with motion sensitive bounding boxes. Consider the moving query q_1 with its query MSB and four moving objects o_1, o_2, o_3 and o_4 as shown in Figure 46. In the figure, o_1 is the focal object of query q_1 and the other three moving objects o_2, o_3 and o_4 are associated with their object MSBs. At time t_0 only objects o_2 and o_3 are subject to query q_1's PQR, as their MSBs intersect with the query's MSB. However the valid prediction time interval of object o_3 with regard to query q_1 is empty because there is no such time interval during which o_3 is expected to be inside the query result of q_1. Thus object o_3 should not be included in the PQR of query q_1. At some later time t_1, object o_2 and query q_1 remain inside their MSBs. However objects o_3 and o_4 have changed their MSBs. As a result, objects o_2 and o_4 become potential candidates of query q_1's PQR at time t_1. Since o_2 has not changed its MSB, it remains included in q_1's PQR. By applying the valid prediction time interval test on o_4, we obtain a non-empty time interval with respect to q_1, during which o_4 is expected to be included in the query result. Thus o_4 is added into the PQR of q_1.

In order to achieve an IO efficient solution, the MSB sizes should be adjusted such that the PQRs are calculated for a sufficiently large set of objects to take advantage of pre-computation. However, result prediction should not be performed for objects that are far away from a query and thus are likely to invalidate their PQRs before becoming of interest

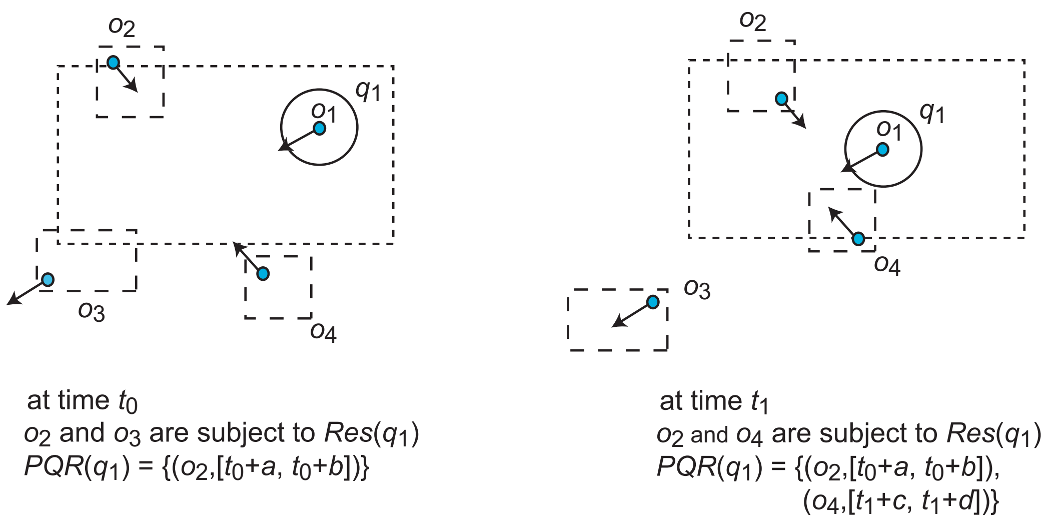

o2
o1
q1
o3
o4
at time t0
o2 and o3 are subject to Res(q1)
PQR(q1) = {(o2,[t0+a, t0+b])}
o2
o1
q1
o4
o3
at time t1
o2 and o4 are subject to Res(q1)
PQR(q1) = {(o2,[t0+a, t0+b]),
(o4,[t1+c, t1+d])}

Figure 46: An illustration of how PQRs integrate with MSBs

to the query. We use α and β parameters to adjust the MSB sizes on per object/query basis to optimize this trade-off. The details are given in Section 4.3.5.

4.3.4 Motion Adaptive Indexing

We have described the main ideas and mechanisms used in our motion-adaptive indexing scheme. In this subsection, we describe motion-adaptive indexing as a query evaluation technique that integrates the ideas and mechanisms presented so far for efficient processing of moving queries over moving objects.

4.3.4.1 Processing Moving Queries: An Overview

The evaluation of moving queries is performed through query evaluation phases executed periodically with regular time intervals of P_s (*scan period*) seconds. We build two spatial MSB indexes, $Index_o^{msb}$ for the objects and $Index_q^{msb}$ for the queries. $Index_o^{msb}$ stores MSBs of the objects accompanied by the associated motion functions as data. Static objects have point MSBs. Similarly, $Index_q^{msb}$ stores the MSBs of the queries accompanied by the associated motion functions of the focal objects of the queries and their radii as data. Static queries have MSBs equal to their minimum bounding rectangles and they do not have associated motion functions.

We create and maintain two tables, a moving object table and a moving query table.

They store information regarding the moving objects and moving queries. The static queries and static objects are included in the spatial MSB indexes but not in the two tables. The periodic evaluation is performed by scanning these tables at each query evaluation phase and performing updates and searches on the spatial indexes as needed in order to incrementally maintain the query results as objects and the spatial regions of the queries move. Detailed descriptions of the two tables are given below:

Moving Object Table (MOT): A MOT entry is a tuple $(i_o, i_q, \vec{p}, \vec{v}, t, B_{msb}, P_{cm}, V_{ch})$ and stores information regarding a moving object. Here, i_o is the moving object identifier, i_q is the query identifier of the moving query whose focal object's identifier is i_o, i_q is $null$ if no such moving query exists, $\vec{p}$ is the last received position, $\vec{v}$ is the last received velocity vector of the moving object, t is the timestamp of the motion updates ($\vec{p}$ and $\vec{v}$) received from the moving object, B_{msb} is the MSB of the moving object, P_{cm} is an estimate on the period of constant motion of the object and V_{ch} is a Boolean variable indicating whether the object has changed its motion function since the last query evaluation phase.

Moving Query Table (MQT): A MQT entry is a tuple $(i_q, \vec{p}, \vec{v}, r, t, B_{msb}, P_{cm}, V_{ch})$ and stores information regarding a moving query. Here, i_q is the moving query identifier, $\vec{p}$ and $\vec{v}$ are the last received position and the last received velocity vector of the query's focal object respectively, t is the timestamp of the motion updates ($\vec{p}$ and $\vec{v}$) received from the focal object, r is the radius of the moving query's spatial region, B_{msb} is the MSB of the moving query, P_{cm} is an estimate on the period of constant motion of the object and V_{ch} is a Boolean variable indicating whether or not the focal object has changed its motion function since the last query evaluation phase. Note that the information in MQT is to some extent redundant with respect to MOT. However the redundant information is required during the moving query table scan. Without redundancy we will need to look them up from the moving object table, which can be costly.

The MOT and MQT table entries are updated whenever new motion updates are received from the moving objects. The P_{cm} entries are updated using a simple weighted running average. The details are given in Figure 47. Assuming that moving objects decide whether or not they should send new motion updates at every P_{mu} seconds (called the

```
MOTION UPDATE RECEIVED($u = \langle i_o, \vec{p}, \vec{v}, t \rangle$)
(1) $e_o = \langle i_o, i_q, \vec{p}, \vec{v}, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MOT$, where $e_o.i_o = u.i_o$
(2) $e_o.P_{cm} \leftarrow \gamma * (u.t - e_o.t) + (1 - \gamma) * e_o.P_{cm}$
(3) $e_o.\vec{p} \leftarrow u.\vec{p}$;   $e_o.\vec{v} \leftarrow u.\vec{v}$
(4) $e_o.t \leftarrow u.t$;   $e_o.V_{ch} \leftarrow$ true
(5) if $e_o.i_q \neq$ null
(6)     $e_q = \langle i_q, \vec{p}, \vec{v}, r, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MQT$, where $e_q.i_q = e_o.i_q$
(7)     $e_q.P_{cm} \leftarrow \gamma * (u.t - e_q.t) + (1 - \gamma) * e_q.P_{cm}$
(8)     $e_q.\vec{p} \leftarrow u.\vec{p}$;   $e_q.\vec{v} \leftarrow u.\vec{v}$
(9)     $e_q.t \leftarrow u.t$;   $e_q.V_{ch} \leftarrow$ true
```

Figure 47: Motion update processing

motion update period), one of our aims is to perform a single query evaluation phase in less than P_{mu} seconds in order not to miss any motion updates, i.e., having $P_s \leq P_{mu}$. If under the available resources, a given implementation of MAI is unable to perform the query evaluation with $P_s \leq P_{mu}$ satisfied, then the query evaluation period P_s has to be increased, i.e., query evaluation has to be performed less frequently. Since the effects of motion updates are reflected to the query results during the next query evaluation step, false positives and false negatives arise in-between query re-evaluations more frequently for larger P_s values. However, this problem is not specific to MAI. In general, when the available resources are not sufficient to handle all queries and position updates in real-time, false positives and negatives will temporarily arise in the query results. When we have $P_s \leq P_{mu}$, then it is at least guaranteed that no motion updates are missed.

Although the moving object and query tables increase the storage requirements of the proposed solution, for most cases the server already contains tables corresponding to all objects and all queries. The object table may contain detailed information about various object attributes and the query table may contain attributes of the queries. In the worst case, where all of the objects and all of the queries are moving, we can expect the size of the database to double due to the inclusion of MOT and MQT. However, we feel that such an increase is acceptable when the improvement in performance is considered.

Figure 48 gives an overall sketch of the query evaluation process. At each query evaluation phase, we need to perform *query table scan* and *object table scan*. The scan algorithms presented in the next subsection describe how these two tasks are performed.

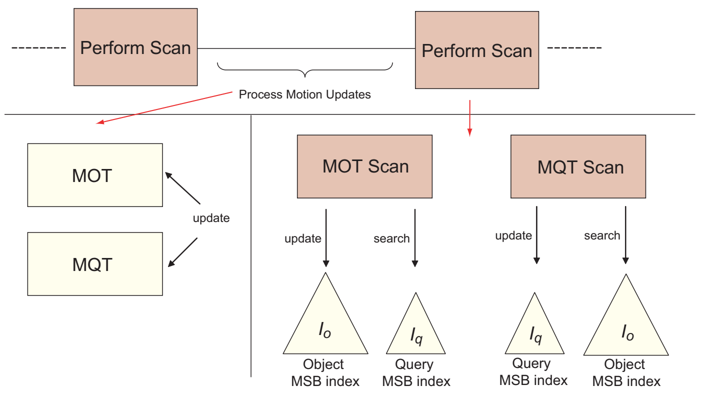


Figure 48: Query evaluation: General view

4.3.4.2 The Scan Algorithms

At each query evaluation phase, two scans are performed. The first scan is on the moving object table, MOT, and the second scan is on the moving query table, MQT. The aim of the MOT scan is to update the $Index_o^{msb}$ and to incrementally update some of the query results by performing searches on the $Index_q^{msb}$. The aim of the MQT scan is to update the $Index_q^{msb}$ and to recalculate some of the query results by performing searches on the $Index_o^{msb}$.

MOT Scan – During the MOT scan, when processing an entry we first check whether the associated object of the entry has invalidated its MSB (using $\vec{p}, \vec{v}, t$, and B_{msb}) or changed its motion function since the last query evaluation period (based on V_{ch}). If none of these has happened, we proceed to the next entry *without performing any operation* on the spatial MSB indexes. Otherwise we first update the $Index_o^{msb}$. In case there is an MSB invalidation, a new MSB is calculated for the object and the $Index_o^{msb}$ is updated. The α value used for calculating the new MSB is selected adaptively, using $|\vec{v}|$ and P_{cm} (See Section 4.3.5.1 for further details). If there has been a motion function change, the data associated with the entry of the object's MSB in the $Index_o^{msb}$ is also updated. Once the $Index_o^{msb}$ is updated, two searches are performed on the $Index_q^{msb}$. First, using the old MSB of the object, the $Index_q^{msb}$ is searched and all the queries whose MSBs intersect with the old MSB of the object are retrieved. The object is then removed from the results of those queries (if it is already in). Then a second search is performed with the newly

```
PERIODIC MOT SCAN()
(1) foreach $e = \langle i_o, i_q, \vec{p}, \vec{v}, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MOT$
(2)     $t_c \leftarrow$ current time
(3)     {Calculate the new object position}
(4)     $e.\vec{p} \leftarrow e.\vec{p} + (t_c - e.t) * e.\vec{v}$; $e.t \leftarrow t_c$
(5)     {$B_{inv}$ is true iff there is MSB invalidation}
(6)     $B_{inv} \leftarrow e.\vec{p} \notin e.B_{msb}$
(7)     {If no MSB invalidation and
(8)             no velocity vector change}
(9)     if $\neg B_{inv} \wedge \neg e.V_{ch}$
(10)        continue{ Nothing to be done}
(11)    $B_{old} \leftarrow e.B_{msb}$
(12)    if $e.V_{ch}$
(13)        $e.V_{ch} \leftarrow$ false
(14)        if $\neg B_{inv}$
(15)            {Update the data associated with $e.i_o$
(16)                    in $Index_o^{msb}$}
(17)            $Index_o^{msb}.updateData(i_o, \langle e.\vec{p}, e.\vec{v}, e.t \rangle)$
(18)    if $B_{inv}$
(19)        $\alpha \leftarrow \alpha\beta Table.lookup(|e.\vec{v}|, e.P_{cm})$
(20)        $e.B_{msb} \leftarrow Rect(e.\vec{p}, e.\vec{p} + \alpha * e.\vec{v})$
(21)        {Update the entry associated with $e.i_o$
(22)                in $Index_o^{msb}$}
(23)        $Index_o^{msb}.update(i_o, e.B_{msb}, \langle e.\vec{p}, e.\vec{v}, e.t \rangle)$
(24)    {Search $Index_q^{msb}$ using the old MSB}
(25)    $Q_o \leftarrow Index_q^{msb}.search(B_{old})$
(26)    {Search (with predictive results) $Index_q^{msb}$
(27)            using the new MSB}
(28)    $Q_n \leftarrow Index_q^{msb}.search(e.B_{msb}, e.\vec{p}, e.\vec{v}, e.t)$
(29)    foreach $s = \langle i_q, t_i = [t_{is}, t_{ie}] \rangle \in Q_n$
(30)        add $\langle e.i_o, t_i \rangle$ into $PQR(s.i_q)$
(31)        remove $s.i_q$ from $Q_o$
(32)    foreach $i_q \in Q_o$
(33)        remove $e.i_o$ from $PQR(s.i_q)$
```

Figure 49: Moving object table scan

```
PERIODIC MQT SCAN()
(1) foreach $e = \langle i_q, \vec{p}, \vec{v}, r, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MQT$
(2)     $t_c \leftarrow$ current time
(3)     $e.\vec{p} \leftarrow e.\vec{p} + (t_c - e.t) * e.\vec{v}$; $e.t \leftarrow t_c$
(4)     {Calculate the new query MBR}
(5)     $B_{qp} \leftarrow Rect(e.\vec{p} - (e.r, e.r), e.\vec{p} + (e.r, e.r))$
(6)     {$B_{inv}$ is true iff there is MSB invalidation}
(7)     $B_{inv} \leftarrow B_{qp} \notin e.B_{msb}$
(8)     {If no MSB invalidation and
(9)             no velocity vector change}
(10)    if $\neg B_{inv} \wedge \neg e.V_{ch}$
(11)        continue{ Nothing to be done}
(12)    if $e.V_{ch}$
(13)        $e.V_{ch} \leftarrow$ false
(14)        if $\neg B_{inv}$
(15)            {Update the data associated with $e.i_q$
(16)                    in $Index_q^{msb}$}
(17)            $Index_q^{msb}.updateData(i_q, \langle e.\vec{p}, e.\vec{v}, e.r, e.t \rangle)$
(18)    if $B_{inv}$
(19)        $\beta \leftarrow \alpha\beta Table.lookup(|e.\vec{v}|, e.P_{cm})$
(20)        $\vec{p_f} \leftarrow e.\vec{p} + \beta * e.\vec{v}$
(21)        $e.B_{msb} \leftarrow Rect(e.\vec{p} - sign(e.\vec{v}) * e.r, \vec{p_f} +$
            $sign(e.\vec{v}) * e.r)$
(22)        {Update the entry associated with $e.i_q$
(23)                in $Index_q^{msb}$}
(24)        $Index_q^{msb}.update(i_q, e.B_{msb}, \langle e.\vec{p}, e.\vec{v}, e.r, e.t \rangle)$
(25)    $PQR(e.i_q) \leftarrow \emptyset$
(26)    {Search (with predictive results) $Index_o^{msb}$
(27)            using the query MSB}
(28)    $O_n \leftarrow Index_o^{msb}.query(e.B_{msb}, e.\vec{p}, e.\vec{v}, e.r, e.t)$
(29)    foreach $s = \langle i_o, t_i = [t_{is}, t_{ie}] \rangle \in O_n$
(30)        add $\langle s.i_o, t_i \rangle$ into $PQR(e.i_q)$
```

Figure 50: Moving query table scan

calculated MSB of the object and all queries whose MSBs intersect with the new MSB of the object are retrieved. For all those queries, result prediction is performed against the object. Lastly, the query result entries obtained from the prediction with non-empty time intervals are added into their associated query results.

MQT Scan – During the MQT scan, when processing a query entry we first check whether the associated query of the entry has invalidated its MSB (using $\vec{p}, \vec{v}, r, t$, and B_{msb}) or its focal object has changed its motion function since the last query evaluation phase (based on V_{ch}). If none of these has happened, we proceed to the next entry *without performing any operation on the spatial indexes.* Otherwise we first update the $Index_q^{msb}$. In case there is an MSB invalidation, a new MSB is calculated for the query and the $Index_q^{msb}$ is updated. The β value used for calculating the new MSB is selected adaptively, using $|\vec{v}|$ and P_{cm} (See Section 4.3.5.1 for details). If there has been a motion function change, the data associated with the entry of the query's MSB in the $Index_q^{msb}$ is also updated. Once the $Index_q^{msb}$ is updated, a single search is performed on the $Index_o^{msb}$ with the newly calculated MSB of the query. All objects whose MSBs intersect with the new query MSB are retrieved. For all those objects, result prediction is performed against the query. The predictive query result entries with non-empty time intervals are added into the query result and all old query results are removed.

Note that after the MOT scan all results are correct for the queries whose MSBs are not invalidated and their focal objects have not changed their motion function. For queries that have invalidated their MSBs or whose focal objects have changed their motion functions, the query results are recalculated during the MQT scan. Therefore, all of the query results are up-to-date after the MQT scan, given that MOT scan is performed first. The order of the scans can be reversed with some minor modifications.

In-between query re-evaluations, false positives and negatives may arise in the query results. False positives may only arise for objects and queries whose motion functions have changed since the last query evaluation step. This is because, when no motion updates take place, PQRs are accurate and can predict the departure of objects from the query regions correctly. On the other hand, false negatives may take place when some of the objects enter into MSBs of some queries in-between query re-evaluations. This happens more frequently when P_s is large. Since we encourage to perform query re-evaluations as frequently as possible, large P_s values are unlikely.

4.3.5 Setting α and β Values

The α and β parameters used for calculating MSBs can be set based on the motion behavior of the objects, in order to achieve more efficient query evaluation. There are two important characteristics of object motions: (a) *the speed of the object* and (b) *the period of constant motion of the object* (i.e., the length of the time period it takes for the motion function to change). For instance, for a query whose focal object changes its motion function frequently, it may not be a good idea to perform too much prediction, thus β value for this query's MSB should be kept smaller. However, for an object with high speed, a small α value may not be appropriate, as it may cause frequent MSB invalidations. As a result, it is important to design a motion-adaptive method that can set the values of α and β parameters adaptively. A common approach to runtime parameter setting is to develop an analytical model and use it to guide the runtime selection of the best parameter settings. We develop an analytical model for estimating the IO cost of performing query evaluation. This model, given in Appendix C, is used as the guide to build an off-line computed $\alpha\beta Table$, giving the best α and β values for different value pairs of speed and period of constant motion of a moving object.

4.3.5.1 $\alpha\beta Table$ and Adaptive Parameter Selection

The cost function developed in this section has a global minimum that optimizes the IO cost of the query evaluation. We build an off-line computed $\alpha\beta Table$, which gives the optimal α and β values for different value pairs of object speed ($\vec{v}$) and period of constant motion (P_{cm}), calculated using the cost function we have developed. We implement the $\alpha\beta Table$ as a 2D matrix, whose rows correspond to different object speeds and columns correspond to different periods of constant motion and the entries are optimal (α, β) pairs. Recall that, as discussed in Section 4.3.4, when we calculate the MSBs of moving objects and moving queries, we already have the estimates on periods of constant motion and speeds of all moving objects including the focal objects of the moving queries. We can decide the best α and β values to use during MSB calculation by performing a single lookup from the off-line computed $\alpha\beta Table$.

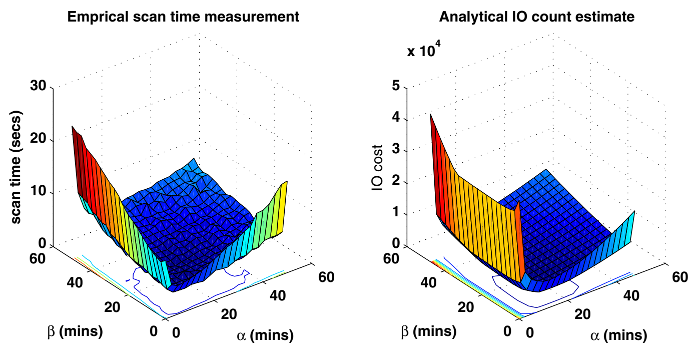


Figure 51: Analytical node IO estimate and experimental query evaluation time

The graph on the left in Figure 51 plots the average time it takes to perform one complete query evaluation phase (labeled as *total query evaluation time*) as a function of α and β. These values are from the actual implementation of motion adaptive indexing. The graph on the right in Figure 51 plots the analytical node IO count estimate of performing one query evaluation phase as a function of α and β. Two important observations can be obtained by comparing these graphs. First, it shows that the IO cost is dominant on the time it takes to perform query evaluation, as the node IO count graph highly determines the shape of the query evaluation time graph. Second, the optimal values of α and β calculated using the analytical cost function indeed results in faster query evaluation.

	velocity								
period of constant motion	10	20	30	40	50	60	70	80	90
0.01	(.98,.75)	(.98,.13)	(.78,.08)	(.63,.07)	(.53,.05)	(.47,.05)	(.42,.05)	(.37,.03)	(.33,.03)
0.05	(.98,.75)	(.98,.18)	(.98,.13)	(.90,.12)	(.73,.10)	(.62,.10)	(.53,.08)	(.47,.08)	(.42,.08)
0.10	(.98,.75)	(.98,.23)	(.98,.18)	(.97,.17)	(.75,.15)	(.62,.13)	(.52,.13)	(.43,.12)	(.38,.12)
0.15	(.98,.75)	(.98,.28)	(.98,.23)	(.93,.22)	(.72,.18)	(.58,.17)	(.50,.17)	(.43,.17)	(.40,.17)
0.20	(.98,.75)	(.98,.33)	(.98,.28)	(.88,.25)	(.70,.22)	(.58,.22)	(.52,.22)	(.47,.22)	(.43,.22)
0.25	(.98,.75)	(.98,.38)	(.98,.32)	(.87,.28)	(.72,.27)	(.62,.27)	(.55,.27)	(.47,.27)	(.40,.27)
0.30	(.98,.75)	(.98,.42)	(.98,.35)	(.88,.32)	(.73,.32)	(.65,.32)	(.53,.32)	(.43,.32)	(.43,.22)
0.35	(.98,.75)	(.98,.45)	(.98,.37)	(.92,.37)	(.77,.37)	(.62,.37)	(.50,.37)	(.50,.23)	(.43,.22)
0.40	(.98,.75)	(.98,.48)	(.98,.42)	(.93,.42)	(.77,.42)	(.58,.42)	(.55,.30)	(.50,.23)	(.45,.20)
0.45	(.98,.75)	(.98,.52)	(.98,.47)	(.98,.47)	(.73,.47)	(.65,.33)	(.55,.30)	(.50,.23)	(.47,.18)
0.50	(.98,.78)	(.98,.55)	(.98,.52)	(.97,.52)	(.70,.52)	(.65,.33)	(.55,.30)	(.52,.22)	(.52,.17)

Figure 52: A sampled subset of the $\alpha\beta Table$ from the experiment of Figure 56

In Figure 52, we give a sampled subset of the $\alpha\beta Table$ that is used in the experiment reported in Figure 56 of Section 4.5. The actual table covers a larger range and has a higher resolution. Each entry in the table is in the form (α, β). We make two observations from Figure 52. First, with increasing object speeds the optimal α and β values decrease. This is because, for high speeds the α and β parameters should be kept small, in order to avoid large MSBs which will cause high overlap and increase the cost of spatial index operations. Second, with decreasing period of constant motion, the optimal α and β values decrease. This is because, large MSBs are undesirable when the predictability is poor (period of constant motion is small), since they will result in a larger number of invalidated PQRs and thus increased IO cost. We will provide performance results on the improvement provided by the adaptive parameter selection in Section 4.5.3.

4.4 Evaluating Moving kNN Queries with Motion Adaptive Indexing

Moving continual *k-nearest neighbor* (kNN) queries over moving objects can be evaluated using the main mechanisms employed for moving range query evaluation. A moving kNN query is defined similar to a moving range query, except that instead of a range, the parameter k is specified for retrieving the k nearest neighbors of the focal object of the query.

A unique feature of our motion adaptive indexing scheme is its ability to efficiently process both continual moving *range* queries and continual moving kNN queries. Note that, for a mobile database system which has to manage both range MCQs and kNN MCQs, solutions that are exclusive to kNN queries will introduce extra overhead, since the indexes and data structures are not shared with the range query evaluation component, further exacerbating the problem of high index maintenance cost in moving object databases. In contrast, our solution uses a common framework to support both range and kNN queries, so that workloads that are mixtures of kNN queries and range queries are efficiently handled. In order to extend the motion adaptive indexing developed for evaluating moving range queries to the evaluation of moving kNN queries, we introduce the concept of *safe radius* and two mechanisms $-$ *guaranteed safe radius* and *optimistic safe radius.* To evaluate kNN queries with the use of safe radii, we need to make the following three changes:

a. During the MQT table scan, when a query invalidates its MSB or changes its motion function, we calculate a *safe radius* which is guaranteed to contain at least k moving objects until the next time the safe radius is calculated (β is an upper bound for this time). Then the kNN query is installed as a standard MCQ with its range equal to the safe radius.

b. Instead of storing time intervals in query result entries, we store the distance of the objects from the focal object of the query as a function of time.

c. At the end of each query evaluation phase, results are sorted based on their distances to their associated focal objects by using the distance functions stored within the query result entries. The top k result entries are then marked as the current results.

The important step here is to calculate a safe radius, that will make sure that at least k objects will be contained within the safe radius during the next t time units. We propose two different approaches to tackle this problem: the guaranteed safe radius (GSR) and the optimistic safe radius (OSR).

The guaranteed safe radius approach retrieves the current k nearest neighbors, and for each object in the list calculates the maximum possible value the distance between the object and the focal object of the query can take *at the end* of the next t time units. This can be calculated using the focal object's motion function and the *upper bounds on the maximum speeds* of these k nearest neighbor objects. The maximum of these k calculated distances will give the safe radius. However, there are two problems. First, it requires us to know the upper bounds on the speeds of moving objects. Second, the calculated safe radius may become unnecessarily large, negatively affecting the performance.

The optimistic safe radius approach retrieves the current k nearest neighbors, and for each object in the list calculates the maximum value of the distance between the object and the focal object of the query can take *throughout* the next t time units, assuming that the objects will not change their motion functions during this time. For each of the k objects, this calculation can be done using the *current motion function* of the object and the motion function of the query's focal object. The maximum of these k calculated distances will give the safe radius. This approach guarantees that k objects will be contained within the safe radius during the next t time units under the assumption that the initial set of

k nearest neighbors do not change their motion functions during this period. When using this approach, if the number of objects in the result of a kNN query turns out to be smaller than k, we fall back to the traditional spatial index kNN search plan for that query until the next time a new safe radius is calculated.

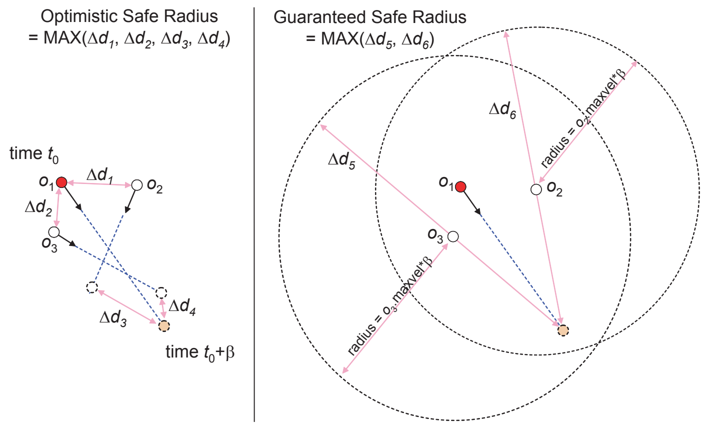


Figure 53: Optimistic and guaranteed safe radius calculation for 2NN queries

Figure 53 illustrates how safe radii are calculated with an example 2NN query, where the focal object is o_1 and the two nearest neighbors at time t_0 are objects o_2 and o_3. The safe radius is calculated to be valid during the next β time units. We will provide the performance comparison of *guaranteed safe radius* (GSR) and *optimistic safe radius*(OSR) in Section 4.5.

4.5 Experimental Results

This section describes five sets of experiments, which are used to evaluate our solution. The first set of experiments compares the performance of motion adaptive indexing against various existing approaches. The second set of experiments illustrates the advantages of adaptive parameter selection over fixed parameter setting on the sizes of bounding boxes. The third set of experiments studies the effect of skewed data and query distribution on query evaluation performance. The fourth set of experiments analyzes the scalability of the proposed approach with respect to queries with varying sizes of spatial regions, varying percentages of moving queries, and varying number of objects. Finally the fifth set of experiments present the effectiveness of the motion adaptive approach to evaluating moving

Table 4: System parameters

Parameter	Default value / Range
area of the region of interest	500000 sq. miles
number of objects	50000 / [50K,200K]
percentage of moving objects	50
number of queries	5000 / [2.5K,20K]
percentage of moving queries	50 / [0,100]
moving query range distribution	{5, 4, 3, 2, 1} miles with Zipf param 0.6
static query side range distribution	{8, 7, 5, 4, 2} miles with Zipf param 0.6
period of constant motion	mean 5 minutes, geometrically distributed
moving object speed	between 0-150 miles/hour uniformly random
scan period	30 seconds
motion update period	30 seconds

continual kNN queries over moving objects.

4.5.1 System Parameters and Setup

In the experiments presented in the rest of the chapter, the parameters take their default values listed in Table 4, when not specified otherwise. Based on the default values, 50% of the objects are moving and the remaining 50% are static. Similarly, 50% of the queries are moving and the remaining 50% are static. Different percentages of moving queries are studied in Section 4.5.6. Moving queries are assigned with range values from the list $\{5, 4, 3, 2, 1\}$(in miles) using a Zipf distribution with parameter 0.6. Static queries are assigned with side range values from the list $\{8, 7, 5, 4, 2\}$ (in miles) using a Zipf distribution with parameter 0.6.

The default object density is taken in accordance with previous work [94, 105]. Objects and queries are randomly distributed in the area of interest, except in Section 4.5.5 where we consider skewed distributions. Objects that belong to different classes with strictly varying movement behaviors are considered in Section 4.5.3. The paths followed by the objects are random, i.e., each time a motion function update occurs, a random direction and a random speed are chosen. The object speeds are selected from the range $(0, 150]$ (in miles/hour) uniformly at random. Table 4 gives details of other important system parameters. We vary the values of many system parameters to study their effects on the performance.

For R*-trees a 101 node LRU buffer is used with 4KB page size. Branching factor of the internal tree nodes is 100 and the fill factor is 0.5. Relative merits of our techniques

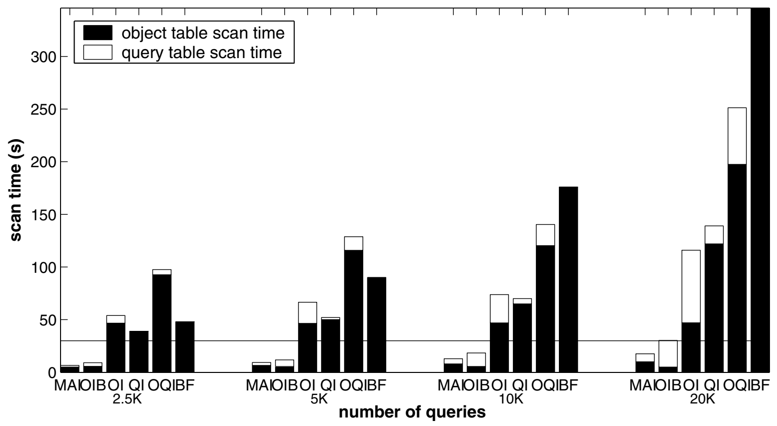


Figure 54: Query evaluation time

shown in the rest of the section are also valid under scenarios with large buffer sizes (which effectively makes it a main memory algorithm), however we do not report those results. All experiments are performed using R*-trees, except that in Section 4.5.5 a static grid based spatial index implementation is used for comparison purposes.

We compare the performance of motion adaptive indexing against various existing approaches, in terms of query evaluation time and node IO counts. The approaches used for comparison are: *Brute Force* (*BF*), *Object-only Indexing* (*OI*), *Query-only Indexing* (*QI*), *Object and Query Indexing* (*OQI*), *Motion Adaptive Indexing* (*MAI*), and *Object Indexing with MSBs* (*OIB*). The Brute Force calculation is performed by scanning through the objects. During the scan, all queries are considered against each object in order to calculate the results. The *OI* approach uses an object index which is updated for all objects that have moved since the last query evaluation phase [1] and searched for all queries in order to evaluate the query results. The *QI* approach uses a query index which is updated for all queries that have moved since the last query evaluation step and searched for all object positions in order to update the query results incrementally. *OQI* is a stripped down version of *MAI* without *MSB*s and *PQR*s. *OIB*s is similar to pure object-only indexing, except that the motion sensitive boxes are used instead of object positions in the spatial index (without the *PQR*s).

[1]Although update-efficient object indexes exist [105, 118], we show that their use does not change our conclusions for large or moderate number of queries, in which case search cost is the dominant factor.

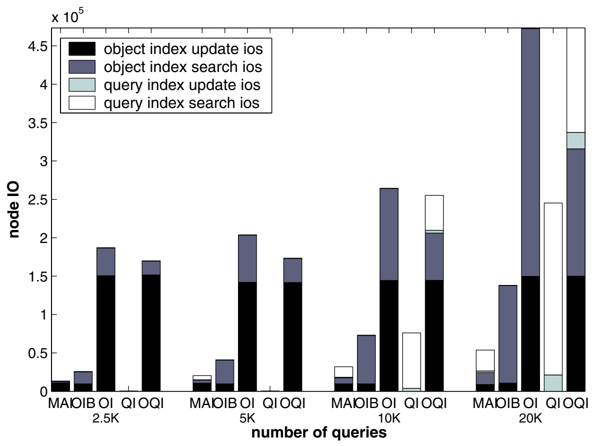


Figure 55: Query evaluation node IO

4.5.2 Performance Comparison

Figure 54 plots the total query evaluation time for fixed number of objects (50K) with varying number of queries (2.5K to 20K). The horizontal line in the figure represents the scan period. We consider a query evaluation scheme as acceptable when the total query evaluation time is less than the scan period. Note that the scan period, P_s, is set to be equal to the motion update period P_{mu} in this set of experiments. Figure 55 plots the query evaluation node IO count for the same setup. The node IO is divided into four different components. These are: (a) node IO due to object index update, (b) node IO due to object index search, (c) node IO due to query index update and (d) node IO due to query index search. Each component is depicted with a different color in Figure 55. Several observations can be obtained from Figure 54 and Figure 55.

First, the approaches with an object index that is updated for all moving objects, do not perform well when the number of queries is small. This is clear from the poor performances of *OI* and *OQI* for 2.5K queries, as shown in Figure 54. The reason is straightforward. The cost of updating the object index dominates when the number of queries is small. This can also be observed by the object index update component of the *OI* in Figure 55. However, there are also significant costs for searching the object index for the *OI* approach. These costs dominate the total IO cost when the number of queries is large (see the case of 20K queries in Figure 55). This points out an important fact, *although it is possible to* *reduce the cost of updating the object index (for instance by using a TPR-tree based object index [105, 118]), MAI still performs significantly better than such an object index based approach.*

Second, the approaches with a query index that is searched for a large number of objects, do not perform well for a large number of queries. This is clear from the poor performances of QI and OQI for 20K queries, as shown in Figure 54. This is due to the fact that, the cost of searching the query index dominates when the number of queries is large. This can also be observed by the query index search component of the QI in Figure 55. Note that, for a small number of queries, the node IO count for QI appears as 0, because the query index fits into the LRU buffer.

Third, the brute force approach performs relatively good compared to OQI and slightly better compared to OI, when the number of queries is small (2.5K), as shown in Figure 54. Obviously BF does not scale with the increasing number of queries, since the computational complexity of the brute force approach is $O(N_o * N_q)$, where N_o is the total number of objects and N_q is the total number of queries. Although OQI seems to be a consistent loser when compared to other indexing approaches, it is interesting to note that the motion adaptive indexing is built on top of it and performs better than all other approaches.

Finally, it is worth noting that only MAI manages to provide good enough performance to satisfy $P_s \leq P_{mu}$ under all conditions. MAI provides around 75-80% savings in query evaluation time under all cases when compared to the best competing approach except OIB. However, OIB performs reasonably well, but fails to scale well with increasing number of queries when compared to the proposed MAI approach.

4.5.3 Effect of Adaptive Parameter Selection

In order to illustrate the advantage of adaptive parameter selection, we compare motion adaptive indexing against itself with static parameter selection. For the purpose of this experiment, we introduce three different classes of moving objects with strictly different movement behaviors. The first class of moving objects change their motion functions frequently (avg. period of constant motion 1 minute) and move slow (max. speed 20 miles/hour). The second class of moving objects possess the default properties described in Section 4.5.1. The third class of moving objects seldom change their motion functions (avg. period of constant motion 30 mins) and move fast (max. speed 300 miles/hour). In order to observe the gain from adaptive parameter selection, we set the α and β parameters to the optimal values obtained for moving objects of the second class for the non-adaptive case.

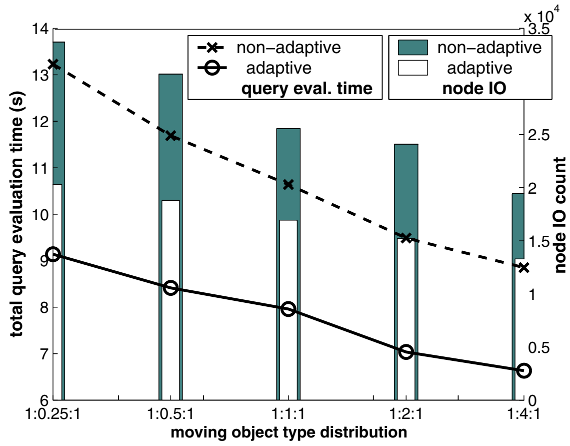


Figure 56: Performance gain due to adaptive parameter selection

Figure 56 plots the time and IO cost of query evaluation for MAI and static parameter setting version of MAI. The x-axis represents the object class distributions. Hence, 1:1:1 represents the case where the number of objects belonging to different classes are the same. Along the x-axis we change the number of objects belonging to the second class. 1:0.25:1 represents the case where the number of objects belonging to the first class and the number objects belonging to the third class are both 4 times the number of objects belonging to the second class. Dually, 1:4:1 represents the case where the second class cardinality is 4 times those of the other two classes. Total query evaluation times are depicted as lines in the figure and their corresponding values are on the left y-axis. The node IO counts are depicted as an embedded bar chart and their corresponding values are on the right y-axis. There are two important observations from Figure 56.

First, we notice that the adaptive parameter selection has a clear performance advantage. This is clearly observed from Figure 56, which shows significant improvement provided by motion adaptive indexing over static parameter setting in both query evaluation time and node IO count.

Second, it is important to note that the objects belonging to the first class or the third class cannot be ignored even if their numbers are small. Even for 1:4:1 distribution, where the second class of objects is dominant, we see a significant improvement with MAI. Note that objects belonging to the first and the third class are expensive to handle. The first class of objects are expensive, as they cause frequent motion updates which in turn causes more processing during MOT and MQT scans. The third class of objects are also expensive, as they cause frequent MSB invalidation which instigates more processing during MOT and MQT scans. The fact that both query evaluation time and node IO count are declining along the x-axis shows that it is obviously more expensive to handle the first and the third class of objects.

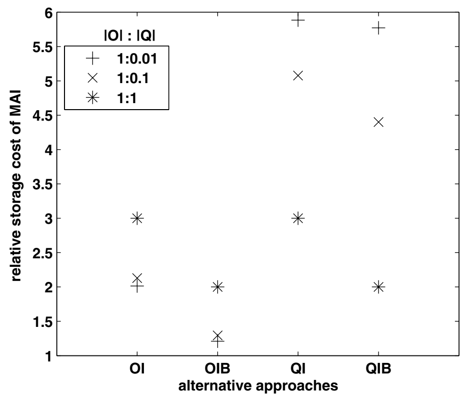


Figure 57: Storage cost of MAI relative to other alternatives

4.5.4 Storage Cost

Since MAI uses both an object index and a query index, its storage requirements are expected to be larger than the storage requirements of the other alternatives considered in this section. However, given that the processing resources are the limiting factor for handling continuous queries in the mobile object monitoring context, this increase in the storage cost is acceptable considering the savings in IO cost and query evaluation time provided by MAI. In Figure 57, we report the storage cost of the MAI approach, relative to other alternatives, for three different settings for the $|O|$:$|Q|$ ratio, that are 1:0.01, 1:0.1, and 1:1. We observe from the figure that, relative to OI and OIB, MAI has a storage cost of around 2 times and 1.25 times for the case of $|O|$:$|Q|$ = 0:0.01 and around 2.15 times and 1.35 times for the case of $|O|$:$|Q|$ = 0:0.1, respectively. For the extreme case of $|O|$:$|Q|$=1:1, where the number of queries is equal to the number of objects, we see that MAI has a storage cost of around 3 times and 2 times relative to OI and OIB, respectively. In general, the number of queries is expected to be smaller than the number of objects, thus it is fair to say that MAI has a storage cost that is around 2 times of a simple object index based approach. The figure also shows results relative to the QI and QIB approaches. It is observed that MAI incurs up to 5 times more storage cost compared to QI, the worst case scenario happening when the number of queries is the smallest, that is $|O|$:$|Q|$ = 0:0.01. However, given the poor performance of QI compared to both OI and MAI, the savings it provides in terms of storage cost are not of much value.

4.5.5 Effect of Data and Query Skewness

Our experiments up to now have assumed uniform object and query distribution. In this section we conduct experiments with skewed data and query distributions. We model skewness using two parameters, *number of hot spots* (N_h) and *scatter deviation* (d). We pick N_h different positions within the area of interest randomly, which correspond to hot spot regions. When assigning an initial position to an object, we first pick a random hot spot position from the N_h different hot spots and then place the object around the hot spot position using a normally distributed distance function on both x and y dimensions with zero mean and d standard deviation. Scatter deviation d is set to 25 miles in all experiments and the number of hot spots is varied to experiment with different skewness conditions. Queries also follows the same distribution with objects. Figure 58 shows the object and query distribution for $N_h = 5$ and $N_h = 30$.

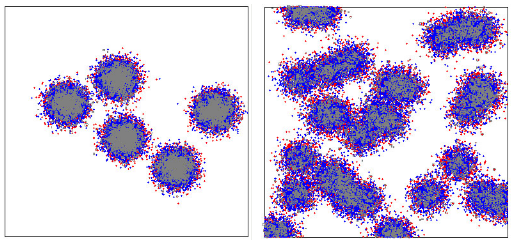


Figure 58: Query and object distribution for $N_h = 5$ and $N_h = 30$

We also experiment with different spatial indexing mechanisms. We have implemented a static grid based spatial index, backed up by a B$^+$-tree with z-ordering [44]. The optimal cell size of the grid is determined based on the workload. The motivation for using a static grid is that, with frequently updated data it may be more profitable to use a statically partitioned spatial index that can be easily updated. Actually, previous work done for static range queries over moving objects [67] has shown that using a static grid outperforms most other well known spatial index structures for in-memory databases. With this experiment we also investigate whether a similar situation exists in secondary storage based indexing in the context of MCQs.

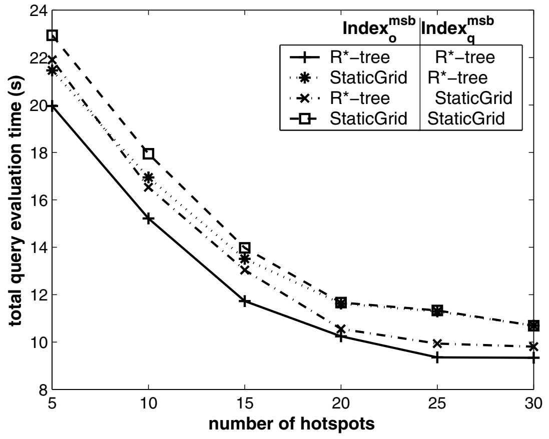


Figure 59: Effect of data and query skewness on performance

Figure 59 plots the total query evaluation time as a function of number of hot spots for different spatial index structures used for $Index_o^{msb}$ and $Index_q^{msb}$. Note that the smaller the number of hot spots, the more skewed the distribution is. Figure 59 shows that decreasing the number of hot spots quadratically increases the query evaluation time. But even for $N_h = 5$, the query evaluation time does not exceed the query evaluation period. Figure 59 also shows that R*-tree performs the best under all conditions.

4.5.6 Scalability Study

In this section we study the scalability of the proposed solution with respect to the varying size of query ranges, the varying percentage of moving queries over the total number of spatial queries, and the varying total number of objects. We first measure the impact of the query range and the moving query percentage on the query evaluation performance. We use the *range factor* (r_f) to experiment with different workloads in terms of different query ranges. The query radius and query side length parameters given in Section 4.5.1 are multiplied by the range factor r_f in order to alter the size of query regions. Note that multiplying the range factor by two in fact increases the area of the query range by four.

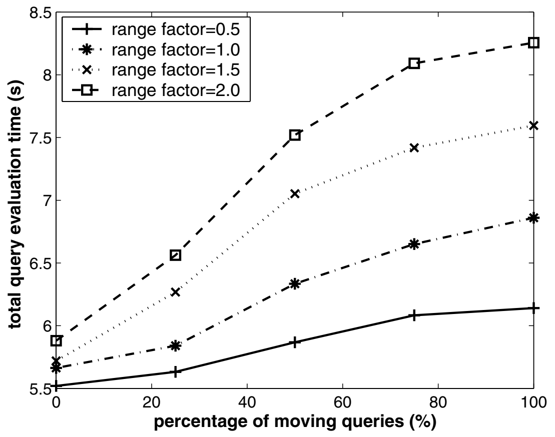


Figure 60: Effect of query range and moving query percentage on performance

Figure 60 plots the total query evaluation time as a function of moving query percentage for different range factors. As shown in Figure 60, the scalability in terms of moving query percentage is extremely good. The slope of the query evaluation time function shows good reduction with increasing percentage of moving objects. Increasing the range factor shows roughly linear increase (with a multiplier that increases with increasing moving query percentage, ≈0.25 to ≈0.5 for 0% to 100%) on the query evaluation time.

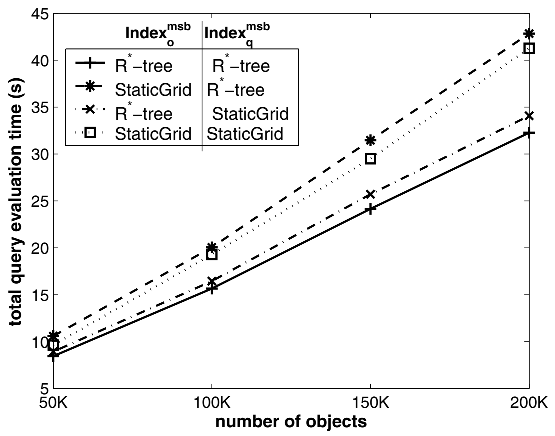


Figure 61: Effect of number of objects on performance

In Figure 61 we study the effect of the number of objects on the query evaluation performance. Figure 61 plots the total query evaluation time as a function of number of objects for different spatial index structures used for $Index_o^{msb}$ and $Index_q^{msb}$. The number of queries is set to its default value of 5K. From Figure 61 we observe a linear increase in the query evaluation time with the increasing number of objects. The query evaluation time for 200K objects is around 4 times the query evaluation time for 50K objects for the R*-tree implementation of $Index_o^{msb}$ and $Index_q^{msb}$, which shows better scalability with increasing number of objects than the static grid implementation.

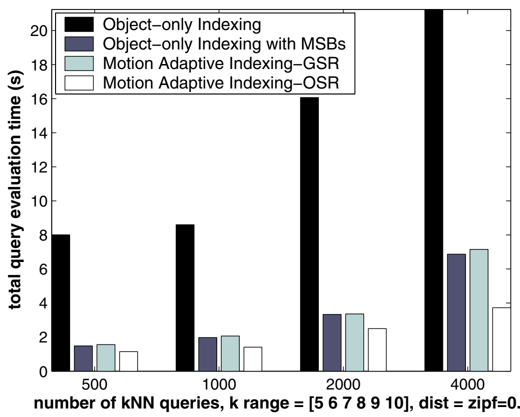


Figure 62: Total query evaluation time for moving continual kNN queries

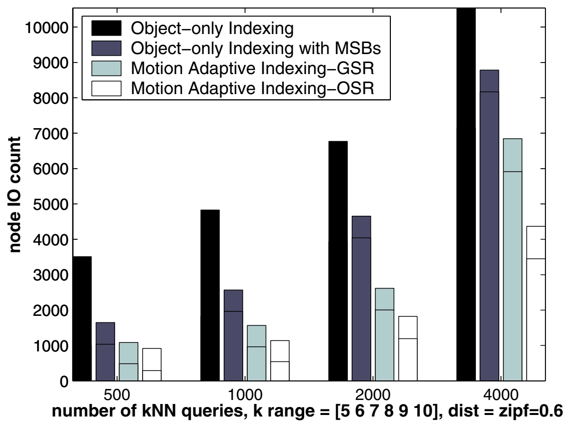


Figure 63: Node IO count for moving continual kNN queries

4.5.7 Performance Comparison for Continual kNN Queries

We compare the performance of MCQ based moving continual kNN query evaluation against the object-only indexing approach. In object-only indexing approach, the object index is updated and the kNN queries are evaluated against the updated object index during each query evaluation phase. In this experiment 10K objects are used with the same object density (N_o/A) specified in Section 4.5.1), where 50% of the objects are moving with the default motion parameters from Section 4.5.1. All queries are moving continual kNN queries and the number of queries ranges from 0.5K to 4K. The k values of the kNN queries are selected from the list $\{5, 6, 7, 8, 9, 10\}$ using a Zipf distribution with parameter 0.6. Figure 62 plots the total query evaluation time and Figure 63 plots the node IO count for different number of objects with different approaches. The node IO count is divided into two components. The lower part shows the node IO due to index searches, where the upper part shows the node IO due to index updates.

Evaluating moving continual kNN queries with motion adaptive indexing shows significant improvement over object-only indexing approach. Between the two variations of safe radius, OSR (optimistic safe radius based approach) performs better than GSR (guaranteed safe radius based approach). Object-only indexing with MSBs (OIB) slightly outperforms GSR. However, OSR provides 20-40% improvement in total query evaluation time over OIB.

An interesting statistic is the average result accuracy of the range MCQs used to answer kNN queries, for GSR and OSR techniques. Concretely, the ratio of the k value specified in the query to the average number of results in the MCQ used to answer the query is an important measure to assess the effectiveness of using range queries as a filtering step in answering kNN queries.

In Figure 64, the range MCQ result accuracy for kNN queries is plotted as a function of k for optimistic and guaranteed safe radius techniques. The k values used in the figure are in the range $[5, 10]$. For $k = 5$ and with OSR, one-fifth of the results of a range MCQ constitute the result of the associated kNN query. This means that the size of the result set of the range MCQ is 25 for a 5NN query, on the average. Importantly, the accuracy of the range MCQs increase with increasing k. For instance, for $k = 10$ and with OSR, one-quarter of the results of a range MCQ constitute the result of the associated kNN query. This increasing trend in accuracy is very useful, since the cost of query evaluation

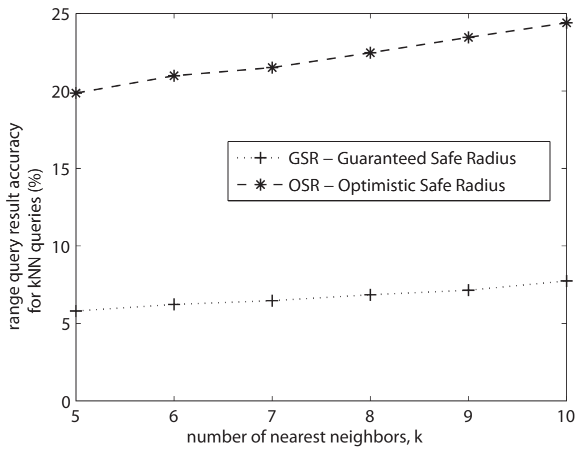


Figure 64: Range MCQ result accuracy for kNN queries

increases with increasing k and it is important that the range MCQs provide good filtering for such costly kNN queries. Figure 64 also shows that GSR performs poorly compared to OSR, having a very low accuracy value of 6% to 8% where k ranges from 5 to 10.

4.6 Related Work

Research on moving object indexing can be broadly divided into two categories, based on (1) the current positions of the moving objects and (2) the trajectories of the moving objects. Our work belongs to the first category. An essential study dealing with the problem of indexing and querying moving object trajectories can be found in [92]. Continual queries are used as a useful tool for monitoring frequently changing information [124, 78]. In the spatial databases domain, continual queries are employed for continuously querying moving object positions. Most of the work on continual queries over moving object positions is either on static continual queries over moving objects [94, 67, 72, 25, 113, 133, 134] or on moving continual queries over static objects [120, 114]. None of the these works has addressed the problem of moving *continual* queries over moving objects.

In [94], velocity constrained indexing and query indexing (Q-index) has been proposed for efficient evaluation of static continual range queries. The same problem is studied in [67], however the focus is on in-memory structures and algorithms. In [105], TPR-tree, an R-tree based indexing structure, is proposed for indexing the motion parameters of moving objects by using time parameterized rectangles and answering queries using this index. TPR*-tree, an extension of TPR-tree optimized for queries that look into the future (predictive), is described in [118]. Note that even though TPR-related indexes [105, 118] support moving queries, these moving queries are predefined regions in the spatio-temporal domain. They are not the moving continual queries, such as MCQ_1 and MCQ_2 discussed in this chapter. Recently, newer indexing schemes that improve upon the performance of TPR-trees have been introduced, such as STRIPES [91] and the B$^+$-tree based indexing technique of [65]. Nevertheless, the focus of these works is on developing search and update efficient indexing structures for managing moving object locations and they do not have special mechanisms to support continual queries, whereas our focus is on developing a logical indexing scheme that leverages already existing indexing structures to support efficient processing of MCQs through incremental evaluation. Advanced indexing structures can be integrated into our MAI approach by replacing the R*-tree based object and query indexes we employ.

In [17], efficient query evaluation techniques for nearest neighbor ($k = 1$) and reverse nearest neighbor queries are developed for moving queries over moving objects. CNN [120] gives an algorithm for pre-calculating k-nearest neighbors with a line segment representing the continuous motion of the query; however the target objects are assumed to be static. In [141], object-only indexing and query-only indexing based techniques are proposed to evaluate moving continuous kNN queries over moving objects. However, the solution is exclusive to kNN queries. In contrary, our approach supports range and kNN queries within the same framework and uses object and query indexing at the same time to optimize the performance for a large range of parameters that include cases where object-only indexing falls short, as well as cases where query-only indexing is ineffective.

The concept of moving continual queries is to some extent similar to Dynamic Queries (DQ) [74]. A dynamic query is defined as a temporally ordered set of snapshot queries in [74]. This is a low level definition as opposed to our definition of moving continual queries which is more declarative and is defined from the users' perspective. The work done in [74] indexes the trajectories of the moving objects and describes how to efficiently evaluate dynamic queries that represent predictable or non-predictable movement of an observer. They also describe how new trajectories can be added when a dynamic query

Table 5: Comparison of motion-adaptive index with existing approaches

	Query Types		
	Moving Query Static Object	Static Query Moving Object	Moving Query Moving Object
MAI	●	●	●
SINA [87]	●	●	●
TPR [105]	○ [2]	●	○[2]
DQ [74]	●	●	●
CNN [120]	●		
Q-index [94]		●	

	System Properties				
	Incremental Evaluation	Predictive Query Results	Index Independence	Motion Modeling	Motion Adaptation
MAI	●	●	●	●	●
SINA [87]	●		●		
TPR [105]				●	
DQ [74]	●			●	
CNN [120]		○ [3]		●	
Q-index [94]	●		●		

is actively running. Their assumptions are in line with their motivating scenario, which is to support rendering of objects in virtual tour-like applications. Our work focuses on real-time evaluation of moving queries in real-world settings, where the trajectories of the moving objects are unpredictable and the queries can potentially be associated with moving objects inside the system. An important feature of our approach is its motion adaptiveness, allowing the query evaluation to be optimized according to the dynamic motion behavior of the objects. Our experiments have shown that such motion adaptive capability offers significant performance gain for evaluating moving queries over moving objects.

The most relevant work to ours, in terms of its support for various types of continual spatial queries discussed in Section 4.1 and its ability to perform incremental evaluation, is the SINA [87] (and its kNN extension SEA-KNN [137]) algorithm that has been developed concurrently and independently with our work. SINA employs hash-based indexing techniques for both objects and queries and generates positive and negative updates (incrementally) through a three-step process consisting of hashing, invalidation and joining. However, there is an inherent difference between our approach and SINA. Specifically, motion modeling

[2]TPR tree only supports moving queries with predefined paths
[3]CNN has per result time intervals, not per object

(described in Section 4.2.2) is integrated into our approach, which enables predictive query results and helps increase the system scalability by reducing the number of location updates received from the moving objects. It has been shown in [33] that the use of linear functions for motion modeling, reduces the amount of updates to one third in comparison to constant functions, for realistic thresholds. However, SINA works on raw location updates in the form of (x, y) coordinate pairs and is not designed to take advantage of motion modeling. On the other hand, motion modeling may introduce additional processing requirements on the moving objects. Fortunately, dead reckoning algorithms for linear motion modeling are simple and can be implemented easily with cheap hardware or software. Besides these, the SINA approach is not motion adaptive like our MAI approach, i.e., it does not optimize the system based on the movement characteristics of the individual objects. In summary, SINA and MAI are different in their assumptions and requirements with respect to the supports required by the mobile objects, as well as in terms of the specific techniques they employ for the purpose of query evaluation. However, both are intended to solve the same high level problem of evaluating moving continuous queries over moving objects.

In [113], a two-level architecture is proposed, where there exist location preprocessors between the moving objects and the database. The location updates are propagated to the database only when the objects cross boundaries of their hash buckets, which are fixed. The database is aware of only the hash buckets and does not know exact positions of objects within the buckets. Some queries have to be propagated to location preprocessors that have the exact information. Going further in this direction, in [25] and [62] distributed architectures that push the location filtering to mobile units were described. Chapter 3 studies distributed location monitoring in detail.

Table 5 summarizes the comparison of our MAI approach with some of the existing approaches. Our approach is the most universal in handling various types of continual queries and has many desirable system properties, such as incremental evaluation of queries and motion adaptation.

CHAPTER V

ENERGY-EFFICIENT DATA COLLECTION FOR SENSOR CQ SYSTEMS

Data collection is a fundamental functionality through which sensor networks form an enabling technology for environmental monitoring applications. Although in-network techniques, such as event detection, aggregation, and query processing, have been studied in the literature as different ways of data collection, one of the most prominent and comprehensive ways of data collection in sensor networks is to periodically extract raw sensor readings. This way of data collection enables complex analysis of data, which may not be possible with in-network aggregation or query processing. However, this flexibility in data analysis comes at the cost of power consumption. In this chapter, we introduce *selective sampling* for energy-efficient periodic data collection in sensor networks. The main idea behind selective sampling is to use a dynamically changing subset of nodes as samplers such that the sensor readings of sampler nodes are directly collected, whereas the values of non-sampler nodes are predicted through the use of probabilistic models that are locally and periodically constructed in an in-network manner. Selective sampling can be effectively used to increase the network lifetime while keeping quality of the collected data high, in scenarios where either the spatial density of the network deployment is superfluous relative to the required spatial resolution for data analysis or certain amount of data quality can be traded off in order to decrease the overall power consumption of the network. Our selective sampling approach consists of three main mechanisms. First, *sensing-driven cluster construction* is used to create clusters within the network such that nodes with close sensor readings are assigned to the same clusters. Second, *correlation-based sampler selection and model derivation* is used to determine the sampler nodes and to calculate the parameters of probabilistic models that capture the spatial and temporal correlations among sensor readings. Last, *selective data collection and model-based prediction* is used to minimize the number of messages used to extract data from the network. A unique feature of our selective sampling mechanisms is the use of localized schemes, as opposed to the protocols requiring global information, to select and dynamically refine the subset of sensor nodes serving as samplers and the model-based value prediction for non-sampler nodes. Such runtime adaptations create a data collection schedule which is self-optimizing in response to changes in energy levels of nodes and environmental dynamics.We present analytical and simulation based results and study the effectiveness of selective sampling under different system settings.

5.1 Introduction

Advances in wireless network technologies, low-power processor and chip design, and micro electromechanical systems have facilitated the proliferation of small, low cost, low power sensor devices that enable seamless integration of the physical world with pervasive networks [41]. The prominent features of such sensor devices are their ability to perform computation, wireless communication, and environmental sensing. On the bright side, the continued price drop in low power sensor devices and their decentralized and unattended nature of operation make sensor networks an attractive tool for extracting and gathering data by sensing real-world phenomena from the physical environment. Environmental monitoring applications are expected to benefit enormously from these developments, as evidenced by recent sensor network deployments supporting such applications [84, 15].

On the downside, the large and growing number of networked sensors and their unattended deployment present a number of unique system design challenges, different from those posed by existing computer networks: (1) *Sensors are power-constrained.* A major limitation of sensor devices is their limited battery life. Wireless communication is a major source of energy consumption, where sensing can also play an important role [39] depending on the particular type of sensing performed (ex. solar radiation sensors [121]). On the other hand, computation is relatively less energy consuming. Motes [60] developed at UC Berkeley and manufactured by Crossbow Inc. [36] are good examples of this type of sensor nodes. (2) *Sensor networks must deal with high system dynamics.* Sensor devices and sensor networks experience a wide range of dynamics, including spatial and temporal change trends in the sensed values that contribute to environmental dynamics, changes in user demands that contribute to task dynamics as to what is being sensed and what is considered interesting changes [42], and changes in the energy levels of the sensor nodes, their location or connectivity that contribute to network dynamics. One of the main objectives in configuring networks of sensors for large scale data collection is to achieve longer lifetimes for sensor network deployments by keeping energy consumption at minimum, while maintaining sufficiently high quality and resolution of the collected data to enable meaningful analysis. These configurations should be periodically re-adjusted to adapt to the various changes resulting from high system dynamics.

5.1.1 Data Collection in Sensor Networks

We can broadly divide data collection, a major functionality supported by sensor networks, into two categories. In *event based* data collection, the sensors are responsible for detecting and reporting (to a base node) events, such as spotting moving targets [75]. Event based data collection is less demanding in terms of the amount of wireless communication, since local filtering is performed at the sensor nodes, and only events are propagated to the base node. In certain applications, the sensors may need to collaborate in order to detect events. Detecting complex events may necessitate non-trivial distributed algorithms [77] that require involvement of multiple sensor nodes. An inherent downside of this kind of data collection is the impossibility of performing in-depth analysis on the raw sensor readings, since they are not extracted from the network.

In *periodic data collection*, periodic updates are sent to the base node from the sensor network, based on the most recent information sensed from the environment. We further classify this approach into two. In *query based* data collection, long standing queries (also called continuous queries [79]) are used to express user or application specific information interests and these queries are installed "inside" the network. Most of the schemes following this approach [82, 83] support aggregate queries, such as minimum, average, and maximum. These types of queries result in periodically generating an aggregate of the recent samples of all nodes. Although aggregation lends itself to simple distributed implementations that enable complete in-network processing of queries, it falls short in supporting holistic aggregates [82] over sensor samples, such as quantiles. Similar to the case of event based data collection, the raw data is not extracted from the network and complex data analysis that requires integration of samples from various nodes at various times, cannot be performed with in-network aggregation.

The most comprehensive way of data collection is to extract raw samples from the network through periodic reporting of each sampled value from every sensor node. This scheme enables arbitrary data analysis at a sensor stream processing center once the data is collected. Such increased flexibility in data analysis comes at the cost of high energy consumption due to excessive communication and consequently decreases the network lifetime. One way of tackling this problem is to use distributed data compression to reduce the total size of the data transmitted on the wireless channel. However, such approaches may require to gather samples belonging to different time intervals before performing compression on them [7]. This may introduce delays, undesirable for real-time applications. As shown in [7], compression techniques that typically trade-off accuracy and delay can cut down the communication cost, thus reduce the energy consumption rate and increase the network lifetime. In this chapter, we develop an alternative approach based on *selective sampling*. The main idea behind selective sampling is to use a carefully selected dynamically changing subset of nodes to sample and to predict the values of the rest of the nodes using probabilistic models. Such models are constructed by exploiting both spatial and temporal correlations existent in sample readings of sensor nodes. There are two major scenarios that can highly benefit from this approach. First, in many sensor network applications, node density of the deployment is selected to result in a spatial resolution higher than the required, mainly because of the short lifespan of the sensor nodes [106], or due to the lack of knowledge about the nature of the phenomenon of interest. As a result, selective sampling can effectively reduce the number of nodes used to sample data, decrease the energy consumption rate of the network, and thus can increase the overall network lifetime. Second and more importantly, there is an inherent trade-off between the accuracy of the collected data and the network lifetime. If the application at hand can tolerate certain levels of error, then selective sampling can be effectively used to save energy by decreasing the quality of received data within acceptable bounds. Such tradeoff is especially useful when the energy left in the network is low and the energy consumption rate is high. A key challenge is to design effective mechanisms that can increase the lifetime of the network while keeping the accuracy of the collected data at satisfactory levels.

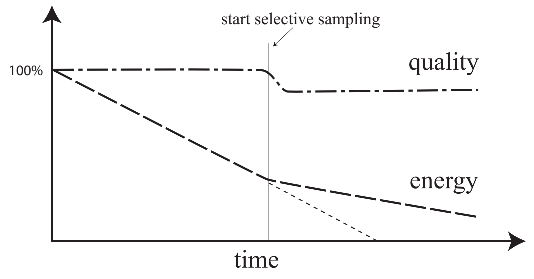


Figure 65: Illustration of energy-quality trade-off

Figure 65 illustrates this trade off graphically. Initially, perfect accuracy is sustained with high rate of energy consumption. Later, selective sampling is used to decrease the rate of energy consumption, while introducing some reduction in data quality, to obtain an increased network lifetime. Note that, it is also possible to use different degrees of selective sampling, depending on the desired energy/quality trade-off.

Another key challenge in designing an energy efficient selective sampling architecture is to empower the system with the ability to respond to high network dynamics. Concretely, the selective sampling approach should support large number of unattended autonomous nodes and should equip the energy-efficient data collection algorithms with self-configuring and self-optimizing capabilities by enabling run-time adaptation to re-select the subset of nodes to sample and to re-construct the correlation-based probabilistic models to enhance the quality of value prediction of non-sampler nodes.

5.1.2 Contributions and Scope of the Chapter

With the above challenges in mind, we identify a number of concrete design principles in designing an effective selective sampling architecture that can respond to changes in energy levels at nodes and network dynamics. First, we need to organize the network into coordination groups such that good probabilistic models can be locally constructed to closely capture the spatial correlations of sensor readings amongst the nodes within each group. Second, we need to utilize the constructed models to find and select the sampler nodes whose sensor readings can provide high accuracy for the prediction to be performed for the non-sampler nodes. Third, but not the least, we need to perform periodic reassignments in order to balance power consumption of the nodes and adapt to possibly changing correlations between sensor readings.

Our selective sampling architecture consists of a three-phase framework and a set of localized algorithms for generating and executing energy-aware data collection schedules. First, we develop *Sensing-driven Cluster Construction* algorithm to group together the nodes such that the ones that are close to each other in terms of their sensor readings (thus the name *sensing-driven*) as well as network hops are put into the same clusters. This is aimed at building a network organization that facilitates local coordination for performing selective sampling and is designed to improve the prediction quality via its sensing-driven nature. Second, we develop *Correlation-based Sampler Selection and Model Derivation* algorithms to partition the nodes within each cluster into a set of subclusters to assist the selection of a set of sampler nodes and to construct one probabilistic model for each subcluster. We address the issues of high prediction accuracy and balanced power consumption by enabling periodic re-configuration of node clusters and periodic re-selection of sampler nodes and re-construction of correlation-based probabilistic models. This allows our selective sampling approach to adapt to possibly changing correlations between sensor readings and balance power consumption of nodes in response to environment and task dynamics. In the third phase, we generate and execute the data collection schedule to collect data from the sensor network in an energy-efficient manner by developing the *Selective Data Collection and Model-based Prediction* algorithms, aiming at keeping the wireless communication at minimum. This enables us to strike a good balance between network lifetime and data quality, and to adjust this balance as needed.

Table 6: Notations for network architecture

Notation	Meaning
N	Total number of nodes in the network
p_i	ith node in the network
$nbr(p_i)$	Neighbors of node p_i in the connectivity graph
$e_i(t)$	Energy left at node p_i at time t
h_i	Cluster head node of the cluster that node p_i belongs to
H	Set of cluster head nodes in the network
C_i	Set of nodes in the cluster with head node p_i
G_i	Set of subclusters in cluster C_i, where $G_i(j)$ is the set of nodes in the jth subcluster in G_i
K_i	Number of subclusters in G_i, also denoted as $\|G_i\|$
S_i	Data collection schedule for cluster C_i, where $S_i[p_j]$ is the status (sampler/non-sampler) of node p_j in S_i

5.2 System Model and Overview

We describe the system model and introduce the basic concepts through an overview of the selective sampling architecture and a brief discussion on the set of algorithms employed. For reference convenience, we list the set of basic notations used in the chapter in Tables 6, 7, 8, and 70. Each table lists the set of notations introduced in its associated section.

5.2.1 Network Architecture

We design our selective sampling based data collection system using a three-layer network architecture. The first and basic layer is the wireless network formed by N sensor nodes and a *data collection tree* constructed on top of the network. We denote a node in the network by p_i, where $i \in \{1, \ldots, N\}$. Each node is assumed to be able to communicate only with its neighbors, that is, the nodes within its communication range. The set of neighbor nodes of node p_i is denoted by $nbr(p_i)$. The neighbor relationship is assumed to be symmetric. The nodes that can communicate with each other form a *connectivity graph*. Figure 66 depicts a segment from a network of hundred sensor nodes. The edges of the connectivity graph are shown with light blue lines (light gray in grayscale). Sensor nodes use a data collection tree for the purpose of propagating their sensed values to a base node. The base node is also the root of the data collection tree. This tree is formed in response to a data collection request, which starts the data collection process. In Figure 66, base node is the shaded one labeled as "56". Every node in the data collection tree, except the root, has a parent node and every non-leaf node has a set of children nodes. The edges of the data collection tree are shown in red color (dark gray in grayscale) in Figure 66. The data collection tree can be easily build in a distributed manner, for instance, by circulating a tree formation message originated from the base node and making use of a min-hop parent selection policy [7], or similar algorithms used for in-network aggregation [83, 82].

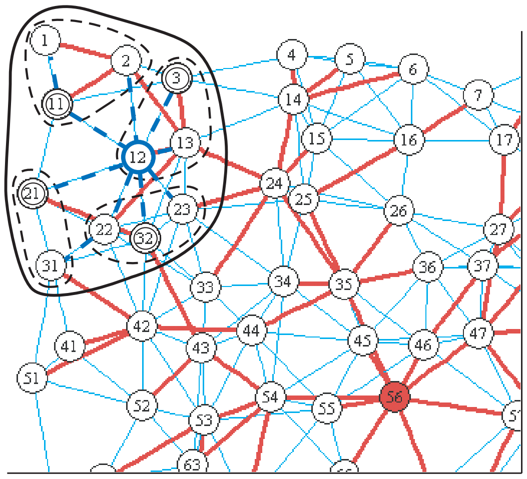


Figure 66: System architecture

The second layer of the architecture consists of node clusters, which partition the sensor network into disjoint regions. Each node in the network belongs to a cluster and each cluster elects a node within the cluster to be the cluster head, and creates a *cluster-connection tree* with the cluster head as its root node to establish the communication between nodes and their cluster head (see Section 3.2 for further detail). We associate each node p_i with a cluster head indicator h_i, $i \in \{1, \ldots, N\}$, to denote the cluster head node of the cluster that node p_i belongs to. The set of cluster head nodes are denoted by H, and is defined formally as $H = \{p_i | \ h_i = p_i\}$. Note that $h_i = p_i$ implies that p_i is a cluster head node (of cluster i). A cluster with p_i as its head node is denoted by C_i and is defined as the set of nodes that belong to it, including its cluster head node p_i. Formally, $C_i = \{p_j | \ h_j = p_i\}$. Given a node p_j has p_i as its cluster head ($h_j = p_i$), we say p_j is in C_i ($p_j \in C_i$). A cluster is illustrated on the upper left corner of Figure 66 with a closed line covering the nodes that belong to the cluster. The cluster head node is drawn in bold and is labeled as "12". An example cluster-connection tree is shown in the figure, where its edges are drawn in dark blue (using dashed lines).

The third layer of our architecture is built on top of the node clusters in the network, by further partitioning each node cluster into a set of *subclusters*. Each node in the network belongs to a subcluster. The set of subclusters in C_i is denoted by G_i, where the number of subclusters in C_i is denoted by K_i where $K_i = |G_i|$. A subcluster within G_i is denoted by $G_i(j), j \in \{1, \ldots, K_i\}$, and is defined as the set of nodes that belong to the jth subcluster in G_i. Given a node cluster C_i, only the head node p_i of this cluster knows all its subclusters ($G_i(j), j \in \{1, \ldots, K_i\}$). Thus the subcluster information is local to the cluster head node p_i and is transparent to other nodes within the cluster C_i. In Figure 66, we show four subclusters for the node cluster with node "12" as its cluster head and these subclusters are circled with closed dashed lines. A key feature of our selective sampling approach is that not all the nodes in the network need to sample and send the sampled values (sensor readings) to the base node via the data collection tree. One of the design ideas is to partition the node cluster in such a way that we can elect a few nodes within each subcluster as the sampling nodes and create a probabilistic model to predict the values of other nodes within this subcluster. From now on, we refer to the nodes that do sampling as *sampler* nodes. In Figure 66, we show sampler nodes with double circled lines (i.e., nodes labeled "3", "11", "21", and "32"). For each cluster C_i, there exists a data collection schedule S_i, which defines the nodes that are samplers in this node cluster. We use the Boolean predicate denoted by $S_i[p_j]$ as an indicator that defines whether node $p_j \in C_i$ is a sampler or not. We use the $[]$ notation whenever the indexing is by nodes.

5.2.2 Selective Sampling Overview

We give an overview of the three main mechanisms that form the crux of our selective sampling approach to data collection. A detailed description of each mechanism is provided in the subsequent sections.

The first mechanism is to construct clusters within the network. This is achieved by the *sensing-driven cluster construction* algorithm, that is executed periodically at every τ_c seconds, in order to perform cluster refinement by incorporating changes in the energy level distribution and the sensing behavior changes of the nodes. We call τ_c the *clustering period.* The node clustering algorithm performs two main tasks – cluster head selection and cluster formation. The cluster head selection component is responsible for defining the guidelines on how to choose certain number of nodes in the network to serve as cluster heads. An important design criterion for cluster head selection is to make sure that on the long run the job of being a cluster head is evenly distributed among all the nodes in the network to avoid burning out the battery life of certain sensor nodes too earlier. The cluster formation component is in charge of constructing clusters according to two metrics. First, nodes that are similar to each other in terms of their sampled values (sensor readings) in the past should be clustered into one group. Second, nodes that are clustered together should be close to each other within certain network hops. The first metric is based on value similarity of sensor readings, which is a distinguishing feature compared to naive minimum-hop based cluster formation where a node joins the cluster that has the closest cluster head node in terms of network hops.

The second mechanism is to create the subclusters for each of the node clusters. The goal of further dividing the node clusters into subclusters is to facilitate the selection of the nodes to serve as samplers and the generation of the probabilistic models for value predication of non-sampler nodes. This is achieved by the *correlation-based sampler selection and model derivation* algorithm that is executed periodically at every τ_u seconds. τ_u is called the *schedule update period.* Concretely, given a node cluster, the cluster head node carries out the sampler selection and model derivation task locally in three steps. In the first step, the cluster head node uses historical data from nodes in its cluster to capture the spatial and temporal correlations in sensor readings and calculate the subclusters so that the nodes whose sample values are highly correlated are put into the same subclusters. In the second step, these subclusters are used to select a set of sampler nodes such that there is at least one sampler node selected from each subcluster. This selection of samplers forms the sampling schedule for the cluster. We introduce a system-wide parameter $\sigma \in (0, 1]$ to define the average fraction of nodes that should be used as samplers. σ is called the *sampling fraction.* Once the sampler nodes are determined, only these nodes collect sample readings and the values of the non-sampler nodes will be predicted at the processing center (or the base node) using a probabilistic model that is constructed for each subcluster. Thus, the third step here is to construct and report a probabilistic model for each subcluster within the network based on the historical readings of all nodes in the subcluster. We introduce a system-supplied parameter β, which defines the average size of the subclusters. β is called the *subcluster granularity* and its setting influences the size and number of the subclusters used in the network.

The third mechanism, is to collect the sampled values from the network and to perform the prediction after the samples are received. This is achieved by the *selective data collection and model-based prediction* algorithm. The selective data collection component works in two steps: (1) Each sampler node samples its reading every τ_d seconds, called the *desired sampling period.* τ_d sets the temporal resolution of the data collection. (2) To empower our selective sampling architecture with self-adaptation, we also need to periodically sample sensor readings from all nodes in the network. Concretely, at every τ_f seconds (τ_f is a multiple of τ_d) all nodes perform sampling. These samples are collected (through the use of cluster-connection trees) and used by the cluster head nodes, aiming at incorporating newly established correlations among sensor readings and network dynamics into decision making process of correlation-based sampler selection and model derivation. τ_f is a system-supplied parameter, called the *forced sampling period.* The model-based predication component is responsible for estimating values of non-sampler nodes within each subcluster using readings of the sampler nodes and the parameters of the probabilistic model constructed for that subcluster.

5.3 Sensing-driven Cluster Construction

The goal of sensing-driven cluster construction is to form a network organization that can facilitate selective sampling through localized algorithms, while achieving the global objectives of energy-awareness and high quality data collection. In particular, clusters help perform operations such as sampler selection and model derivation in a localized manner. By emphasizing on sensing-driven clustering, it also helps to derive better prediction models

Table 7: Notations for sensing-driven clustering

Notation	Meaning
s_i	Head selection probability
r_i	Round counter of node p_i used for clustering
TTL	Max. number of hops a cluster formation message can travel
μ_i	Mean of the sensor readings node p_i has sampled
T_i	Smallest hop distances from cluster heads in proximity of p_i, as known to p_i during cluster formation
V_i	Means of readings from cluster heads in proximity of p_i, as known to p_i during cluster formation
Z_i	Attraction scores for cluster heads in proximity of p_i, where $Z_i[p_j]$ is the attraction score for node $p_j \in H$
f_c	Cluster count factor
α	Data importance factor
τ_c	Clustering period

to increase the prediction quality. The sensing-driven clustering algorithm, executed periodically at every τ_c seconds, performs two main tasks – cluster head selection and cluster formation.

5.3.1 Cluster Head Selection

During the cluster head selection phase, nodes decide whether they should take the role of a cluster head or not. Concretely, every node is initialized not to be a cluster head and does not have an associated cluster in the beginning of a cluster head selection phase. A node p_i first calculates a value called *head selection probability*, denoted by s_i. This probability is calculated based on two factors. The first one is a system wide parameter called *cluster count factor*, denoted by f_c. It is a value in the range (0,1] and defines the average fraction of nodes that will be selected as cluster heads. In other words, $f_c * N$ number of nodes will be selected as cluster heads on the average, with an average cluster size of $1/f_c$. The factors that can affect the decision on the number of clusters and thus the setting of f_c include the size and density of the network. The second factor involved in the setting of s_i is the *relative energy level* of the node. We denote the amount of energy available at node p_i at time t as $e_i(t)$.

The relative energy level is calculated by comparing the energy available at node p_i with the average energy available at the nodes within its one hop neighborhood. The value of the head selection probability is then calculated by multiplying the cluster count factor with the relative energy level. Formally, $s_i = f_c * \frac{e_i(t)*(|nbr(p_i)|+1)}{e_i(t)+\sum_{p_j \in nbr(p_i)} e_j(t)}$. This enables us to favor nodes with higher energy levels for cluster head selection. Once s_i is calculated, node p_i is chosen as a cluster head with probability s_i. If selected as a cluster head, p_i initializes a number of states before starting to circulate cluster formation messages to begin the cluster formation process (described in the next subsection). Concretely, p_i sets h_i to p_i indicating that it now belongs to the cluster with head p_i (itself) and also increments its *round counter*, denoted by r_i to note that a new cluster has been selected for the new clustering round. If p_i is not selected as a cluster head, it waits for some time to receive cluster formation messages from other nodes. If no such message is received, it repeats the whole process starting from the s_i calculation. Considering most realistic scenarios governing energy values available at nodes and practical settings of f_c (< 0.2), this process results in selecting approximately $f_c * N$ number of nodes as cluster heads. The pseudo code is given in the ClustInit procedure of Figure 67.

5.3.2 Cluster Formation

The cluster formation phase starts right after the cluster head selection phase. It organizes the network of sensors into node clusters in two major steps: *message circulation* and *cluster engagement*.

5.3.2.1 Message Circulation

This step involves the circulation of cluster formation messages within the network. These messages are originated at cluster head nodes. Once a node p_i is chosen to be a cluster head, it prepares a message m to be circulated within a bounded number of hops, and structures the message m as follows. It sets $m.org$ to its node identifier p_i. This field represents the originator of the cluster formation message. It sets $m.ttl$ to TTL, where TTL is a system-wide parameter that defines the maximum number of hops this message can travel within the network. This field indicates the number of remaining hops the message can travel. It sets $m.rnd$ to its round counter r_i. It sets $m.src$ to p_i, indicating the sender of the message. Finally, it sets $m.dmu$ to μ_i. Here, μ_i denotes the mean of the sensor readings node p_i has sampled during the time period preceding this round (r_i) of cluster formation. The message

```
ClustInit($p_i$)
(1)  $h_i \leftarrow nil$
(2)  while  $h_i = nil$
(3)      $t \leftarrow$ Current time
(4)      $s_i \leftarrow f_c * \frac{e_i(t)*(|nbr(p_i)|+1)}{e_i(t)+\sum_{p_j \in nbr(p_i)} e_j(t)}$
(5)      if  $rand(0,1) < s_i$
(6)          $h_i \leftarrow p_i$
(7)          $r_i \leftarrow r_i + 1$  /* $r_i \leftarrow 0$ during system init */
(8)          $m.org \leftarrow p_i, m.ttl \leftarrow TTL$
(9)          $m.rnd \leftarrow r_i, msg.src \leftarrow p_i$
(10)         $m.dmu \leftarrow \mu_i$
(11)         foreach  $p_j \in nbr(p_i)$
(12)             SendMsg($p_j, m$)
(13)     else if  a cluster formation message received
(14)         return

ReceiveMsg($p_i$, $m$)
(1)  if  $m.rnd > r_i$
(2)      $r_i \leftarrow m.rnd$
(3)      $T_i \leftarrow \emptyset$, $V_i \leftarrow \emptyset$
(4)  else if  $m.rnd < r_i$
(5)      return
(6)  $a \leftarrow 1 + TTL - m.ttl$
(7)  if  $T_i[m.org] \neq \emptyset$  and  $T_i[m.org] < a$
(8)      return
(9)  $T_i[m.org] \leftarrow a$, $V_i[m.org] \leftarrow m.dmu$
(10) $m.ttl \leftarrow m.ttl - 1, m.src \leftarrow p_i$
(11) foreach  $p_j \in nbr(p_i) \backslash msg.src$
(12)     SendMsg($p_j, m$)

PickClusters($p_i$)
(1)  $y \leftarrow 1/\mathcal{N}(\mu_i |\, \mu_i, \sigma_i)$
(2)  foreach  $j, T_i[p_j] \neq \emptyset$
(3)      $a \leftarrow 1 - T_i[p_j]/TTL$  /* hop distance factor */
(4)      $b \leftarrow \mathcal{N}(V_i[p_j] |\, \mu_i, \sigma_i) * y$  /* data distance factor */
(5)      $Z_i[p_j] \leftarrow a + \alpha * b$
(6)  $h_i \leftarrow argmax_{p_j}(Z_i[p_j])$
```

Figure 67: Sensing-driven cluster construction

m is then sent to all neighbors of node p_i.

Upon reception of a message m at a node p_i, we first compare the *rnd* field of the message to p_i's current round counter r_i. If $m.rnd$ is smaller than r_i, we discard the message, since it is likely a delayed message in some earlier clustering round and should be disregarded. If $m.rnd$ is larger than r_i, then this is the first cluster formation message p_i has received for the new round. As a result, we increment r_i to indicate that node p_i is now part of the current round. Moreover, we initialize two data structures, denoted by T_i and V_i. Both are initially empty. $T_i[p_j]$ stores the shortest known hop count from a cluster head node p_j to node p_i, if a cluster formation message is received from p_j. $V_i[p_j]$ stores *dmu* field of the cluster formation messages that originated from node p_j and reached node p_i. Once the processing of the *rnd* field of the message is over, we calculate the number of hops this message traveled, by investigating the *ttl* field, which yields the value $1 + TTL - m.ttl$. If $T_i[m.org]$ is not empty (meaning this is not the first message we received in this round that originated from node $m.org$) and $T[m.org]$ is smaller than or equal to the number of hops the current message has traveled, we discard the message. Otherwise, we set $T[m.org]$ to $1 + TTL - m.ttl$ and $V_i[m.org]$ to $m.dmu$. Once T_i and V_i are updated with the new information, we modify and forward the message to all neighbors of p_i, except the node specified by *src* field of the message. The modification on the message involves decrementing the *ttl* field and setting the *src* field to p_i. The pseudo code for the message circulation phase is given within the ClustInit and ReceiveMsg procedures in Figure 67.

5.3.2.2 *Cluster Engagement*

This step involves making a decision about which cluster to join, once hop distance and mean sample value information are collected. Concretely, a node p_i that is not a cluster head, performs the following procedure to determine its cluster. For each cluster head node from which it has received a cluster formation message in this round (i.e. $\{p_j | \, T_i[p_j] \neq \emptyset\}$), it calculates an *attraction score*, denoted by $Z_i[p_j]$, for the cluster head p_j. Then it joins the cluster head with the highest attraction score, i.e., it sets h_i to $argmax_{p_j}(Z_i[p_j])$. The calculation of the attraction score $Z_i[p_j]$ involves two factors. The first factor is called the *hop distance factor* and is calculated as $1 - T_i[p_j]/TTL$. It takes its minimum value 0 when p_i is TTL hops away from p_j and its maximum value $1 - 1/TTL$ when p_i is one hop away from p_j. The second factor is called the *data distance factor* and it is calculated as $\mathcal{N}(V_i[p_j] \| \mu_i, \varsigma_i^2)/\mathcal{N}(\mu_i | \mu_i, \varsigma_i^2)$. Here, $\mathcal{N}$ represents the Normal distribution and ς_i^2 is a locally estimated variance of the sampled values at node p_i. The data distance factor measures the similarity between the mean of the sensor readings at node p_i and the mean readings at its cluster head node p_j. It takes its maximum value of 1 when $V_i[p_j]$ is equal to μ_i. Its value decreases as the difference between $V_i[p_j]$ and μ_i increases, and approaches to 0 when the difference approaches to infinity. This is a generic way to calculate the data distance factor and does not require detailed knowledge about the data being collected. However, if such knowledge is available, a domain-specific data distance function can be applied. For instance, if a domain expert can set a system wide parameter Δ to be the maximum acceptable bound of the difference between the mean sample value of a node and the mean sample value of its head node, then we can specify a distance function $f(d) = d/\Delta$, where d is set to $|V_i[p_j] - \mu_i|$. In this case, the data distance factor can be calculated as $max(0, 1 - f(d))$. With this definition, the distance factor will take its maximum value of 1 when d is 0, and its value will linearly decrease to 0 as d reaches Δ.

We compute the attraction score as a weighted sum of the hop distance factor and the data distance factor, where the latter is multiplied by a factor called *data importance factor*, denoted by α. α takes a value in the range $[0, \infty)$. A value of 0 means only hop distance is used for the purpose of clustering. Larger values result in a clustering that is more dependent on the distances between the mean sample values of the nodes. The pseudo code for cluster engagement step is given by the PickClusters procedure in Figure 67.

5.3.3 Cluster-connection Tree Formation

Each node cluster in the network not only elects a cluster head node but also forms a cluster connection tree during the cluster construction. Such cluster connection trees are used to accomplish the communication of nodes with their cluster heads. Concretely, the cluster-connection trees are formed as follows: When a node p_i receives a cluster formation message originated at a cluster head node p_j, p_i notes down the node from which it has received this cluster formation message as the candidate for becoming the parent node of p_i in the cluster connection tree anchored at p_j. If there are several distinct cluster head nodes that circulate a cluster formation message to node p_i, then for each one of such cluster heads, say p_j, p_i stores only one forwarder node, say p_u, which is the one that has forwarded the cluster formation message of p_j with the highest TTL value. Now this forwarder node p_u becomes the parent pointer of p_i for cluster C_j. After nodes have decided on which of the node clusters to join at the end of the cluster engagement step, each node sends a confirmation message to its corresponding cluster head using the parent pointer stored for that cluster. If a node p_i is in cluster C_j **or** has forwarded a confirmation message destined to the cluster head node p_j, then it keeps its parent pointer associated with C_j and at the same time it also records the nodes from which it has received a confirmation message destined to the cluster head node p_j as its child pointers in the cluster connection tree rooted at p_j. Any parent pointers that are not used to forward a confirmation message can be dropped to minimize the state kept for maintaining cluster-connection trees. All nodes that keep their parent pointers for cluster C_j are part of the cluster-connection tree of C_j. It is important to note that a node that participates in the cluster-connection tree of a node cluster C_j may **not** necessarily be a member of C_j, i.e. it is possible that $h_i \neq p_j$. This property ensures that with the cluster-connection tree of a node cluster C_j, its cluster head p_j can reach all nodes in C_j and vice versa.

5.3.4 Effect of α on Clustering

We use an example scenario to illustrate the effect of α on clustering. Figure 68 (a) shows an arrangement of 100 sensor nodes and the connectivity graph of the sensor network formed by these nodes. In the background, it also shows a colored image that represents the environmental values that are sensed by the nodes. The color on the image changes from tones of red (light gray in grayscale) to tones of blue (dark gray in grayscale) moving diagonally from the upper right corner to the lower left corner, representing a decrease in the sensed values. Figures 68 (b), (c), and (d) show three different clusterings of the network for different values of α (0, 10, 20 respectively), using $f_c = 0.1$ and $TTL = 5$. Each cluster is shown with a different color. Nodes within the same cluster are labeled with a unique cluster identifier. The cluster head nodes are marked with a circle. It is clearly observed that, with increased α, the clusters tend to align diagonally, resulting in a clustering where nodes sampling similar values are assigned to the same clusters. However, this effect is limited by the value of TTL, since a node cannot belong to a cluster whose cluster head is more that TTL hops away. We provide a quantitative study on the effect of α on the quality of the clustering in Section 5.6.

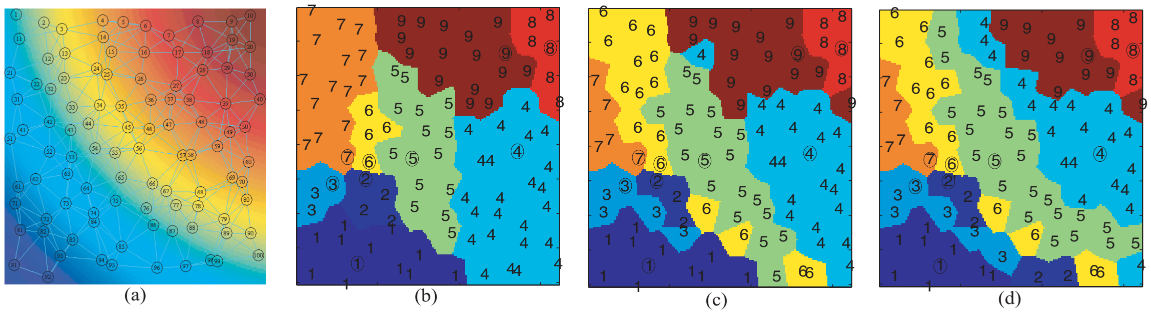


Figure 68: Illustration of sensing-driven clustering

From Figures 68(c) and (d), one can observe that combining hop distance factor and sensor reading similarity captured by data distance factor, some of the resulting clusters may appear disconnected. As discussed in the previous section, by creating a cluster-connection tree for each node cluster, we guarantee that a cluster head node can reach all nodes in its cluster. When the number of connected subcomponents within a cluster is large, the overhead for a cluster head to communicate with nodes within its cluster will increase. However, the number of connected subcomponents of a sensing-driven cluster can not be large in practice due to three major reasons: First, since there is a fixed TTL value used in cluster creation, the nodes that belong to the same cluster can not be more than a specified number of hops away, thus it is not possible that two nodes from different parts of the network are put within the same cluster just because the values they are sensing are very similar. Second, the decision to join a cluster is not only data dependent. Instead, it is a combination (adjusted by α) of hop distance factor and data distance factor that defines a node's affinity to join a cluster. As a result, unless TTL and α values are both set to impractically large values, there won't be many connected components belonging to the same cluster. Finally, since the sensor readings are expected to be spatially correlated, it is unlikely to have contiguous regions with highly heterogeneous sensor readings (which would have resulted in clusters with many connected subcomponents).

5.3.5 Setting of Clustering Period τ_c

The setting of clustering period τ_c involves two considerations. First, the cluster head nodes have additional responsibilities when compared to other nodes, due to sampler selection and model derivation process (see more detail in the next section), which causes them to consume energy at higher rates. Therefore, large τ_c values may result in imbalanced power levels and decrease network connectivity in the long run. Consequently, the value of τ_c parameter should be small enough to enable selection of alternate nodes as cluster heads. However, its value is expected to be much larger than the desired sampling and forced sampling periods, τ_d and τ_f. Second, time dependent changes in sensor readings may render the current clustering obsolete with respect to data distance factor. For instance, in environmental monitoring applications, different times of a day may result in different node clusters. Thus, clustering period should be adjusted accordingly to enable continued refinement of the clustering structure in response to different sensing patterns resulting from environmental changes.

5.4 Correlation-based Sampler Selection and Model Derivation

The goal of sampler selection and model derivation is three folds. First, it needs to further group nodes within each node cluster into a set of subclusters such that the sensor readings of the nodes within each subcluster are highly correlated (thus prediction is more effective). Second, it needs to derive and report (to sampler nodes) a sampling schedule that defines the sampler nodes. And third, it needs to derive and report (to the base node) parameters of the probabilistic models associated with each subcluster so that prediction can be performed. Correlation-based sampler selection and model derivation is performed by each cluster head node through a three-step process, namely *subclustering*, *sampler selection*, and *model and*

Table 8: Notations for correlation-based sampler selection and model derivation

Notation	Meaning
D_i	Forced samples collected at node $p_i \in H$, where $D_i[p_j]$ is the series of consecutive forced samples from node $p_j \in C_i$
$\mathcal{C}_i$	Correlation matrix at node $p_i \in H$, where $\mathcal{C}_i[p_u, p_v]$ is the correlation between the series $D_i[p_u]$ and $D_i[p_v]$
$\mathcal{D}_i$	Subclustering distance matrix at node $p_i \in H$, where $\mathcal{D}_i[p_u, p_v]$ is the subclustering distance between p_u and p_v
β	Subcluster granularity
σ	Sampling fraction
τ_u	Schedule update period

schedule reporting. We now describe these three steps in detail.

5.4.1 Subclustering

This step is used to create subclusters that form a basis for selecting sampler nodes of the network, deriving correlations among nodes within subclusters, constructing probabilistic models accordingly, and performing value predication of non-sampler nodes. Higher correlations among the nodes within each subcluster typically lead to higher quality sampler selection and higher accuracy of model-based value prediction of non-sampler nodes. Thus, given a cluster, the first issue involved in developing an effective subclustering algorithm is to obtain samples from nodes within this cluster. The second issue is to compute correlations between every pair of nodes within the cluster and define a correlation distance metric that can be used as the distance function for subclustering.

5.4.1.1 Forced Sampling

Recall from Section 5.2.2, we introduce the concept of forced sampling period. By periodically collecting sample readings from all nodes in the network (forced sampling), the cluster head nodes can refine the subclustering structure by invoking a new run of the subclustering process, which utilizes the forced samples collected during the most recent clustering period to generate a new set of subclusters, each associated with a newly derived correlation-based probabilistic model. We denote the forced samples collected at cluster head node p_i by D_i. $D_i[p_j]$ denotes the series of consecutive forced samples from node p_j, where p_j is in the node cluster with p_i as the head (i.e. $p_j \in C_i$).

5.4.1.2 *Correlation Matrix and Distance Metric*

During subclustering, a cluster head node p_i takes the following concrete actions. It first creates a correlation matrix $\mathcal{C}_i$ such that for any two nodes in the cluster C_i, say p_u and p_v, $\mathcal{C}_i[p_u, p_v]$ is equal to the correlation between the series $D_i[p_u]$ and $D_i[p_v]$, formally $\frac{(D_i[p_u]-\mathrm{E}[D_i[p_u]])*(D_i[p_v]-\mathrm{E}[D_i[p_v]])^T}{L*\sqrt{\mathrm{Var}(D_i[p_u])}*\sqrt{\mathrm{Var}(D_i[p_v])}}$, where L is the length of the series and ^{T} represents matrix transpose. This is a textbook definition [26] of correlation between two series, expressed using the notations introduced within the context of this work. Correlation values are always in the range $[-1, 1]$, -1 and 1 representing strongest negative and positive correlation. A value of 0 implies two series are not correlated. As a result, the absolute correlation can be used as a metric to define how good two nodes are, in terms of predicting one's sample from another's. For each node cluster, we first compute its correlation matrix using forced samples. Then we calculate the correlation distance metric between nodes, denoted by $\mathcal{D}_i$. $\mathcal{D}_i[p_u, p_v]$ is defined as $1-|\mathcal{C}_i[p_u, p_v]|$. Once we get the distance metric, we use agglomerative clustering [55] to subcluster the nodes within cluster C_i into K_i number of subclusters, where $G_i(j)$ denotes the set of nodes in the jth subcluster. We use a system-wide parameter called *subcluster granularity*, denoted by β, to define average subcluster size. Thus, K_i is calculated by $\lceil |C_i|/\beta \rceil$. We'll discuss the effects of β on performance later in this section. The pseudo code for the subclustering step is given within the SubclusterAndDerive procedure in Figure 69.

5.4.2 Sampler Selection

This step is performed to create or update a data collection schedule S_i for each cluster C_i, in order to select the subset of nodes that are best qualified to serve as samplers throughout the next schedule update period τ_u. After a cluster head node p_i forms the subclusters, it initializes the data collection schedule S_i to zero for all nodes within its cluster, i.e. $S_i[p_j] = 0, \forall p_j \in C_i$. Then for each subcluster $G_i(j)$, it determines the number of sampler nodes to be selected from that subcluster based on the size of the subcluster $G_i(j)$ and the sampling fraction parameter σ defined in Section 5.2. At least one node should be selected as a sampler node from each subcluster. Thus we can calculate the number of

```
DeriveSchedule($p_i \in H$)
(1)  Periodically, every $\tau_u$ seconds
(2)      $D_i$: data collected since last schedule derivation, $D_i[p_j](k)$ is the $k$th forced sample
         from node $p_j$ collected at node $p_i$
(3)      $(S_i, \mathcal{C}_i, G_i) \leftarrow$ SubclusterAndDerive($p_i$, $D_i$)
(4)      for  $j = 1$  to  $|G_i|$
(5)          $\mathcal{X}_{i,j} : \mathcal{X}_{i,j}[p_u] = \mathrm{E}[D_i[p_u]]; p_u \in G_i(j)$
(6)          $\mathcal{Y}_{i,j} : \mathcal{Y}_{i,j}[p_u, p_v] = \mathcal{C}_i[p_u, p_v] * \sqrt{\mathrm{Var}(D_i[p_u])} * \sqrt{\mathrm{Var}(D_i[p_v])}; p_u, p_v \in G_i(j)$
(7)          SendMsg($base, \mathcal{X}_{i,j}, \mathcal{Y}_{i,j}$)
(8)      foreach  $p_j \in C_i$
(9)          $D_i[p_j] \leftarrow \emptyset$
(10)         SendMsg($p_j, S_i[p_j]$)

SubclusterAndDerive($p_i \in H$)
(1)  $\forall p_u, p_v \in C_i,\ \mathcal{C}_i[p_u, p_v] \leftarrow$ Correlation between $D_i[p_u], D_i[p_v]$
(2)  $\forall p_u, p_v \in C_i,\ \mathcal{D}_i[p_u, p_v] \leftarrow 1 - |\mathcal{C}_i[p_u, p_v]|$
(3)  $K_i \leftarrow \lceil |C_i| / \beta \rceil$     /* number of subclusters */
(4)  Cluster the nodes in $C_i$, using $\mathcal{D}_i$ as distance metric, into $K_i$ subclusters
(5)  $G_i(j)$ : nodes in the $j$th subcluster within $C_i$, $j \in \{1, \ldots, K_i\}$
(6)  $t \leftarrow$ Current time
(7)  $\forall p_u \in C_i, S_i[p_u] \leftarrow 0$
(8)  foreach  $j \in \{1, \ldots, K_i\}$
(9)      $a \leftarrow \lceil \sigma * |G_i(j)| \rceil$
(10)     foreach  $p_u \in G_i(j)$, in decreasing order of $e_u(t)$
(11)         $S_i[p_u] \leftarrow 1$
(12)         if  $a = |\{p_v|\ S_i[p_u] = 1\}|$  then  break
(13) return $(S_i, \mathcal{C}_i, G_i)$
```

Figure 69: Correlation-based sampler selection and model derivation

sampler nodes for a given subcluster $G_i(j)$ by $\lceil \sigma * |G_i(j)| \rceil$. Based on the above formula, we can calculate the actual fraction of nodes selected as the sampler nodes of the network at any given instance of time. This actual fraction may deviate from the system-supplied sampling fraction parameter σ. We refer to the actual fraction of sampler nodes as the *effective* σ to distinguish it from the system-supplied σ. The *effective* σ can be estimated as $f_c * \lceil 1/(f_c * \beta) \rceil * \lceil \beta * \sigma \rceil$. The pseudo code for the derivation step is given within the SubclusterAndDerive procedure in Figure 69.

5.4.3 Model and Schedule Reporting

This step is performed by a cluster head node in two steps, after generating the data collection schedule for each node cluster. First, the cluster head informs the nodes about their status as samplers or non-samplers. Then the cluster head sends the summary information to the base node, which will be used to derive the parameters of probabilistic models used in predicting the values of non-sampler nodes. To implement the first step, a cluster head node p_i notifies each node p_j within its cluster about p_j's new status with regard to being a sampler node or not by sending $S_i[p_j]$ to p_j. To realize the second step, for each subcluster $G_i(j)$, p_i calculates a data mean vector for nodes within the subcluster, denoted by $\mathcal{X}_{i,j}$, as follows: $\mathcal{X}_{i,j}[p_u] = \mathrm{E}[D_i[p_u]], p_u \in G_i(j)$. p_i also calculates a data covariance matrix for nodes within the subcluster, denoted by $\mathcal{Y}_{i,j}$ and defined as follows: $\mathcal{Y}_{i,j}[p_u, p_v] = \mathcal{C}_i[p_u, p_v] * \sqrt{\mathrm{Var}(D_i[p_u])} * \sqrt{\mathrm{Var}(D_i[p_v])}, p_u, p_v \in G_i(j)$. For each subcluster $G_i(j)$, p_i sends $\mathcal{X}_{i,j}$, $\mathcal{Y}_{i,j}$ and the identifiers of the nodes within the subcluster, to the base node. This information will later be used for deriving the parameters of a Multi-Variate Normal (MVN) model for each subcluster (see Section 5.5). The pseudo code is given within the DeriveSchedule procedure of Figure 69.

5.4.4 Effects of β on Performance

The setting of the system supplied parameter β (subcluster granularity) may have effects on the overall performance of a selective sampling based data collection system, especially in terms of sampler selection quality, value predication quality, messaging cost, and energy consumption. Intuitively, large values of β may decrease the prediction quality, because it will result in large subclusters with potentially low overall correlation between its members. On the other hand, too small values may also decrease the prediction quality, since the opportunity to exploit the spatial correlations fully will be missed with very small β. Regarding the messaging cost of sending sampling summarization and model derivation information to the base node, one extreme case is where each cluster has one subcluster (very large β). In this case, the covariance matrix may become very large and sending it to the base station may increase the messaging cost and have a negative effect on the energy-efficiency. In contrast, smaller β values will result in a lower messaging cost, since covariance values of node pairs belonging to different subclusters will not be reported. Although the the second dimension favors a small β value, decreasing beta will increase the deviation of effective σ from the system specified σ, introducing another dimension. For instance, having $\beta = 2$ will result in a minimum effective σ of around 0.5, even if σ is specified much smaller. This is because each subcluster must have at least one sampler node. Consequently, the energy saving expected when σ is set to a certain value is dependent on the setting of β. In summary, small β values can make it impossible to practice high energy saving/low prediction quality scenarios. We investigate these issues quantitatively in Section 5.6.

5.4.5 Setting of Schedule Update Period τ_u

The schedule update period τ_u is a system supplied parameter and it defines the time interval for re-computing the subclusters of a node cluster in the network. Several factors may affect the setting of τ_u. First, the nodes that are samplers consume more energy compared to non-samplers, since they perform sensing and report their sensed values. Consequently, the value of τ_u parameter should be small enough to enable selection of alternate nodes as samplers through the use of energy-aware schedule derivation process, in order to balance power levels of the nodes. Moreover, such alternate node selections help in evenly distributing the error of prediction among all the nodes. As a result, τ_u is provisioned to be smaller compared to τ_c, so that we can provide fine-grained sampler re-selection without much overhead. Second, the correlations among sensor readings of different nodes may change

Notation	Meaning
$\mathcal{X}_{i,j}$	Data mean vector for nodes in $G_i(j)$, where $\mathcal{X}_{i,j}[p_u]$ is the mean of the forced samples from node $p_u \in G_i(j)$.
$\mathcal{Y}_{i,j}$	Data covariance matrix for nodes in $G_i(j)$, where $\mathcal{Y}_{i,j}[p_u, p_v]$ is the covariance between the series $D_i[p_u]$ and $D_i[p_v]$
$U^{+}_{i,j}$	Set of nodes belonging to $G_i(j)$ that are samplers
$U^{-}_{i,j}$	Set of nodes belonging to $G_i(j)$ that are not samplers
$W^{+}_{i,j}$	Set of last reported sensor readings of nodes in $U^{+}_{i,j}$
$W^{-}_{i,j}$	Set of predicted sensor readings of nodes in $U^{-}_{i,j}$
τ_d	Desired sampling period
τ_f	Forced sampling period

Figure 70: Notations for selective data collection and model-based prediction

with time and deteriorate the prediction quality. As a result, the schedule update period should be adjusted accordingly, based on dynamics of the specific application at hand.

5.5 Selective Data Collection and Model-based Prediction

Our selective sampling approach achieves energy efficiency of data collection services by collecting sample readings from only a subset of nodes (sampler nodes) that are carefully selected and dynamically changing (after every schedule update period). The values of non-sampler nodes are predicted using probabilistic models whose parameters are derived from the recent samples of nodes that are spatially and temporally correlated. The energy saving is a result of smaller number of messages used to extract and collect data from the network, which is a direct benefit of smaller number of sensing operations performed. Although all nodes have to sample after every forced sampling period (recall that these samples are used for predicting the parameters of MVN models for the subclusters), these forced samples do not propagate up to the base node, and are collected locally at cluster head nodes. Instead, only a summary of the model parameters are submitted to the base node after each correlation-based model derivation step.

In effect, one sample value from every node is calculated at the base node (or at the sensor stream processing center). However, a sample value comes from either a *direct* sample or a *predicted* sample. Direct samples are the ones that originate from actual sensor readings. If a node p_i is a sampler, i.e. $S_j[p_i] = 1$ where $h_i = p_j$, it periodically reports its sensor reading to the base node using the data collection tree, i.e. after every desired sampling period τ_d, except when forced sampling and desired sampling periods coincide (recall that τ_f is a multiple of τ_d). In the latter case, the sensor reading is sent to the cluster head node h_i using the cluster-connection tree, and is forwarded to base node from there. If a node p_i is a non-sampler node, i.e. $S_j[p_i] = 0$ where $h_i = p_j$, then it only samples after every forced sampling period, and its sensor readings are sent to the cluster head node h_i using the cluster-connection tree and are *not* forwarded to the base node. A short pseudo code describing this is given by the SensData procedure in Figure 71.

5.5.1 Calculating predicted sample values

The problem of predicting the sample values of non-sampler nodes can be described as follows. Given a set of sample values belonging to same sampling step from sampler nodes within a subcluster $G_i(j)$, how can we predict the set of sample values belonging to non-sampler nodes within $G_i(j)$, given the mean vector $\mathcal{X}_{i,j}$ and covariance matrix $\mathcal{Y}_{i,j}$ for the subcluster. We denote the set of sampler nodes from subcluster $G_i(j)$ by $U^+_{i,j}$, defined as $\{p_u | \; p_u \in G_i(j), S_i[p_u] = 1\}$. Similarly, we denote the set of non-sampler nodes by $U^-_{i,j}$, defined as $\{p_u | \; p_u \in G_i(j), S_i[p_u] = 0\}$. Let $W^+_{i,j}$ be the set of sample values from the same sampling step, received from the sampler nodes $U^+_{i,j}$. Using a MVN model to capture the spatial and temporal correlations within a subcluster, we utilize the following theorem that can be found in texts on statistical inference [26], to predict the values of the non-sampler nodes:

Theorem 1: *Let X be a MVN distributed random variable with mean μ and covariance matrix Σ. Let μ be partitioned as $\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$ and Σ partitioned as $\begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$. According to this, X is also partitioned as X_1 and X_2. Then the distribution of X_1 given $X_2 = A$ is also MVN with mean $\mu^* = \mu_1 + \Sigma_{12} * \Sigma_{22}^{-1} * (A - \mu_2)$ and covariance matrix $\Sigma^* = \Sigma_{11} - \Sigma_{12} * \Sigma_{22}^{-1} * \Sigma_{21}$.*

In accordance with the theorem, we construct μ_1 and μ_2 such that they contain the mean values in $\mathcal{X}_{i,j}$ that belong to nodes in $U^-_{i,j}$ and $U^+_{i,j}$, respectively. A similar procedure is performed to construct $\Sigma_{11}, \Sigma_{12}, \Sigma_{21}$, and Σ_{22} from $\mathcal{Y}_{i,j}$. Σ_{11} contains a subset of $\mathcal{X}_{i,j}$ which describes the covariance among the nodes in $U^-_{i,j}$, and Σ_{22} among the nodes in $U^+_{i,j}$. Σ_{12}

```
SensData($p_i$)
(1)  if  $S_j[p_i] = 0$, where $h_i = p_j$
(2)      Periodically, every $\tau_f$ seconds
(3)          $d_i \leftarrow$ Sense()
(4)          SendMsg($h_i, d_i$)
(5)  else
(6)      Periodically, every $\tau_d$ seconds
(7)          $d_i \leftarrow$ Sense()
(8)          $t \leftarrow$ Current time
(9)          if  $mod(t, \tau_f) = 0$
(10)             SendMsg($h_i, d_i$)
(11)         else
(12)             SendMsg($base, d_i$)
```

```
PredictData($i$, $j$, $U^+$, $U^-$, $W^+$)
$U^+ = \{p_{u_1^+}, \ldots, p_{u_k^+}\}$ : set of nodes from $j$th subcluster in $C_i$ whose data values are received
$U^- = \{p_{u_1^-}, \ldots, p_{u_l^-}\}$ : set of nodes from $j$th subcluster in $C_i$ whose data values are missing
$W^+ : W^+(a), a \in \{1, \ldots, k\}$ is the value reported by node $p_{u_a^+}$
(1)  $\mathcal{X}_{i,j}$: mean vector for $j$th subcluster in $C_i$
(2)  $\mathcal{Y}_{i,j}$: covariance matrix for $j$th subcluster in $C_i$
(3)  for  $a = 1$  to  $l$
(4)      $\mu_1(a) \leftarrow \mathcal{X}_{i,j}[p_{u_a^-}]$
(5)      for $b = 1$ to $l$, $\Sigma_{11}(a,b) \leftarrow \mathcal{Y}_{i,j}[p_{u_a^-}, p_{u_b^-}]$
(6)      for $b = 1$ to $k$, $\Sigma_{12}(a,b) \leftarrow \mathcal{Y}_{i,j}[p_{u_a^-}, p_{u_b^+}]$
(7)  for $a = 1$ to $k$
(8)      $\mu_2(a) \leftarrow \mathcal{X}_{i,j}[p_{u_a^+}]$
(9)      for $b = 1$ to $k$, $\Sigma_{22}(a,b) \leftarrow \mathcal{Y}_{i,j}[p_{u_a^+}, p_{u_b^+}]$
(10) $\mu^* = \mu_1 + \Sigma_{12} * \Sigma_{22}^{-1} * (W^+ - \mu_2)$
(11) $\Sigma^* = \Sigma_{11} - \Sigma_{12} * \Sigma_{22}^{-1} * \Sigma_{12}^T$
(12) Use $\mathcal{N}(\mu^*, \Sigma^*)$ to predict values of nodes in $U^-$
```

Figure 71: Selective data collection and model-based prediction

contains a subset of $\mathcal{X}_{i,j}$ which describes the covariance between the nodes in $U^-_{i,j}$ and $U^+_{i,j}$, and Σ_{21} is its transpose. Then the theorem can be directly applied to predict the values of non-sampler nodes $U^-_{i,j}$, denoted by $W^-_{i,j}$. $W^-_{i,j}$ can be set to $\mu^* = \mu_1 + \Sigma_{12} * \Sigma_{22}^{-1} * (W^+_{i,j} - \mu_2)$, which is the maximum likelihood estimate, or $\mathcal{N}(\mu^*, \Sigma^*)$ can be used to predict the values with desired confidence intervals. We use the former in the rest of the chapter. The details of prediction step is given by the PredictData procedure in Figure 71.

5.5.2 Prediction Models

The detailed algorithm governing the prediction step can consider alternative inference methods and/or statistical models with their associated parameter specifications, in addition to the prediction method described in this section and the Multi-Variate Normal model used with data mean vector and data covariance matrix as its parameters. Our data collection framework is flexible enough to accommodate such alternative prediction methodologies. For instance, we can keep the MVN model and change the inference method to Bayesian inference. This can provide significant improvement in prediction quality if prior distributions of the samples are available or can be constructed from historical data. This flexibility allows us to understand how different statistical inference methods may impact the quality of the model-based prediction. We can go one step further and change the statistical model used, as long as the model parameters can be easily derived locally at the cluster heads and are reasonably compact in size.

5.5.3 Setting of Forced and Desired Sampling Periods τ_f and τ_d

The setting of forced sampling period τ_f involves three considerations. First, increased number of forced samples (thus smaller τ_f) may be desirable, since it can improve the ability to capture correlations in sensor readings better. Second, large number of forced samples can cause the memory constraint on sensor nodes to be a limiting factor, since the cluster head nodes are used to collect forced samples. Pertaining to this, a lower bound on τ_f can be computed based on the number of nodes in a cluster and the schedule update period τ_u. For instance, if we want the forced samples to occupy an average memory size of M units where each sensor reading occupy R units, then we should set τ_f to a value larger than $\frac{\tau_u * R}{f_c * M}$. Third, less frequent forced sampling results in smaller set of forced samples, which is more favorable in terms of messaging cost and overall energy consumption. In summary, the value of τ_f should be set taking into account the memory constraint and the desired trade-off between prediction quality and network lifetime. The setting of desired sampling period τ_d defines the temporal resolution of the collected data and is application specific.

5.6 *Performance Study*

We present analytical and simulation based experimental results to study the effectiveness of our selective sampling approach. We divided the experiments into two sets. The first set of experiments compares different variations of selective sampling and studies the impact of various parameters on performance, with regard to messaging cost. These results are based on analytical derivations given in Appendix D. The second set of experiments study the effect of various system parameters on the quality of collected data as well as the quality/lifetime trade-off. These experiments are based on simulations using real-world data.

5.6.1 Messaging Cost

For the purpose of comparison, we introduce two variations of selective sampling − *central* approach and *local* approach. The central approach presents one extreme of the spectrum, in which both the model prediction and the value prediction of non-sampling nodes are carried out at the base node or processing center outside the network. This means that all forced samples are forwarded to the base node to compute the correlations in a centralized location. In the local approach, value prediction is performed at the cluster heads instead of the base nodes or the processing center outside the network, and predicted values are reported to the base node. Although the local approach results in a large messaging cost and works against the idea of selective sampling, it can be used to serve as a base case for comparison. The selective sampling solution falls in between these two extremes. We call it the *hybrid* approach due to the fact that the spatial and temporal correlations are captured and summarized locally within the network, whereas the value prediction is performed centrally at the base node. We calculate the total number of messages spent for data collection using different approaches, namely hybrid, central, local, and *non-selective*. We compare the results for different values of system parameters. The non-selective case refers to *naïve* periodic data collection with no support for selective sampling.

In general, the gap between local and non-selective approaches, with the local approach being more expensive in terms of messaging cost, indicates the overhead of cluster construction, sampler selection and model derivation, and selective data collection steps when the savings due to selective sampling are removed (local approach). On the other hand, the gap between central and hybrid approaches, with the hybrid being less expensive thus better, indicates the savings obtained by only reporting the summary of correlations among sensor nodes within each of the subclusters (hybrid approach), instead of forwarding all forced samples to the base node (central approach). The default parameters used in this set of experiments are as follows. The total time is set to $T = 1000000$ units. The total number of nodes in the network is set to 600 unless specified otherwise. f_c is selected to result in an average cluster size of 30 nodes. Desired and forced sampling periods are set to $\tau_d = 1$ and $\tau_f = 10$ time units. Clustering period is set to $\tau_c = 5000$ time units and the schedule update period is set to $\tau_u = 1000$ time units. Sampling fraction σ is set to 0.25 and β is set to 10.

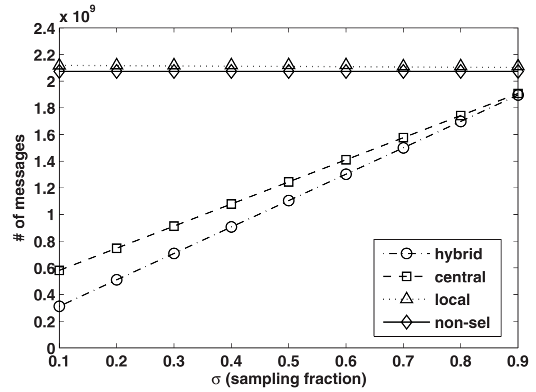


Figure 72: Total # of messages as a function of the sampling fraction

Figure 72 plots the total number of messages as a function of the sampling fraction σ. We make several observations from the figure. First, as expected, central and hybrid approaches provide significant improvement over local and non-selective approaches. This improvement decreases as σ increases, since increasing values of σ imply that larger number of nodes are becoming samplers. Second, the overhead of schedule and model derivation step can be observed by comparing non-selective and local approaches. Note that the gap between the two is very small and implies that this step incurs very small messaging overhead. Third, the improvement provided by the hybrid approach can be observed by comparing hybrid and central approaches. We see an improvement ranging from 50% to 12% to around 0%, while σ increases from 0.1 to 0.5 to 0.9. This shows that the hybrid approach is superior to the central approach and is effective in terms of messaging cost especially when σ is small.

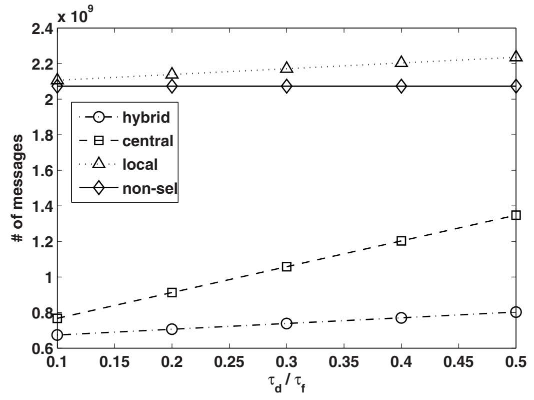


Figure 73: Total # of messages as a function of desired to forced sampling ratio

Figure 73 plots the total number of messages as a function of the desired sampling period to forced sampling period ratio (τ_d/τ_f). In this experiment τ_d is fixed at 1 and τ_f is altered. We make two observations. First, there is an increasing overhead in the total number of messages with increasing τ_d/τ_f, as it is observed from the gap between local and non-selective approaches. This is mainly due to the increasing number of forced samples, which results in higher number of values from sampler nodes to first visit the cluster head node and then reach the base node, causing an overhead compared to forwarding values directly to the base node. Second, we observe that the hybrid approach prevails over other alternatives and provides an improvement over central approach, ranging from 10% to 42% while τ_d/τ_f ranges from 0.1 to 0.5. This is because the forced samples are only propagated up to the cluster head node with the hybrid approach.

Figure 74 plots the total number of messages as a function of the number of nodes.

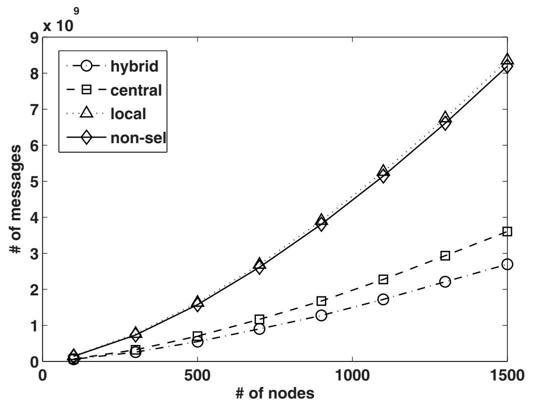


Figure 74: Total number of messages as a function of the number of nodes

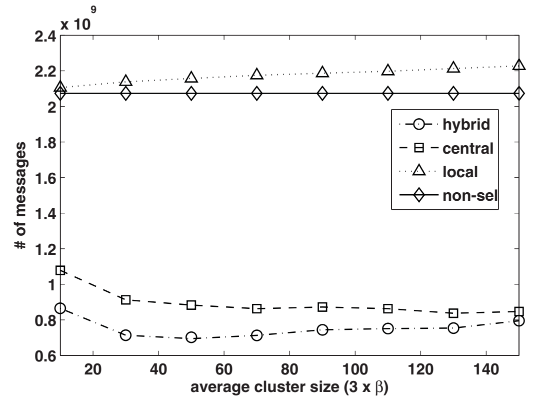


Figure 75: Total number of messages as a function of the average cluster size

The main observation from the figure is that, central and hybrid approaches scale better with increasing number of nodes, where the hybrid approach keeps its relative advantage over central approach (around %25 in this case) for different network sizes. Figure 75 plots the total number of messages as a function of the average cluster size (i.e. $1/f_c$). β is also increased as the average cluster size is increased, so that the average number of subclusters per cluster is kept constant (around 3). From the gap between local and non-selective approaches, we can see a clear overhead that increases with cluster size. On the other hand, this increase does not cause an overall increase in the messaging cost of the hybrid approach until the average cluster size increases well over its default value of 30. It is observed from the figure that the best value for the average cluster size is 50 for this scenario, where smaller and larger values increase the messaging cost. It is also interesting to note that in the extreme case, where there is a single cluster in the network, central and hybrid approaches should converge. This can be observed from the right end of the x-axis in the figure.

5.6.2 Data Collection Quality

We study the data collection quality of our selective sampling approach through a set of simulation based experiments using real data. In particular, we study the effect of α on the quality of clustering, the effect of α, β and subclustering methodology on the prediction error, the trade-off between network lifetime (energy saving) and prediction error, and the load balance in selective sampling schedule derivation. For the purpose of experiments presented in this section, 1000 sensor nodes are placed in a square grid with a side length of 1 unit and the connectivity graph of the sensor network is constructed assuming that two nodes that are at most 0.075 units away from each other are neighbors. Settings of other relevant system parameters are as follows. TTL is set to 5. The sampling fraction σ is set to 0.5. β is set to 5 and f_c is set to 0.02 resulting in an average cluster size of 50. The data set used for the simulations is derived from the GPCP One-Degree Daily Precipitation Data Set (1DD Data Set) [46]. It provides daily, global 1x1-degree gridded fields of precipitation measurements for the 3-year period starting from January 1997. This data is mapped to our unit square and a sensor reading of a node at time step i is derived as the average of the five readings from the ith day of the 1DD data set whose grid locations are closest to the location of the sensor node (since the dataset has high spatial resolution).

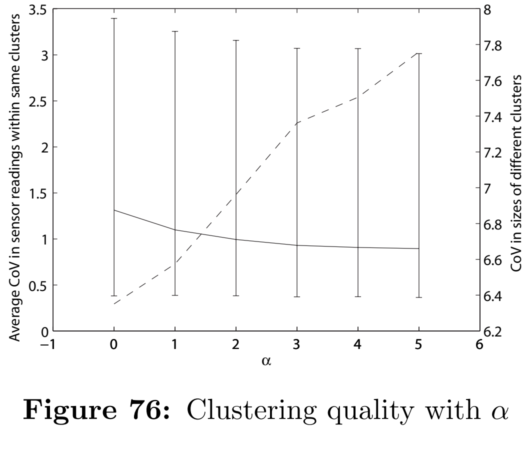


Figure 76: Clustering quality with α

5.6.2.1 *Effect of α on the Quality of Clustering*

Figure 76 plots the average coefficient of variance (CoV) of sensor readings within same clusters (with a solid line using the left y-axis), for different α values. For each clustering, we calculate the mean, maximum and minimum of the CoV values of the clusters, where CoV of a cluster is calculated over the mean data values of sensor nodes within the cluster. Averages from several clusterings are plotted as an error bar graph in Figure 76, where the two ends of the error bars correspond to average minimum and average maximum CoV values. Smaller CoV values in sensor readings imply a better clustering, since our aim is to gather together sensor nodes whose readings are similar. We observe that increasing α from 0 to 4 decreases the CoV around %50, where further increase in α do not provide improvement for this experimental setup. To show the interplay between the shape of the clusters and sensing-driven clustering, Figure 76 also plots the CoV in the sizes of clusters (with a dashed line using the right y-axis). With hop based clustering (i.e., $\alpha = 0$), the cluster sizes are expected to be more evenly distributed when compared to sensing-driven clustering. Consequently, the CoV in the sizes of clusters increases with increasing α, implying that the shape of clusters are being influenced by the similarity of sensor readings. These results are in line with our visual inspection based results shown in Figure 68 in Section 5.3.4.

Table 9: Error for different α values

α	Mean Absolute Deviation (Relative)	%90 Confidence Interval
0	0.3909 (0.1840)	[0.0325, 2.5260]
1	0.3732 (0.1757)	[0.0301, 2.0284]
2	0.3688 (0.1736)	[0.0296, 1.9040]
3	0.3644 (0.1715)	[0.0290, 1.7796]
4	0.3600 (0.1695)	[0.0284, 1.6552]

5.6.2.2 *Effect of α on the Prediction Error*

In order to observe the impact of data-centric clustering on prediction quality, we study the effect of increasing data importance factor α on the prediction error. The second column of Table 9 lists the mean absolute deviation (MAD) of the error in predicted sample values for different α values listed in the first column. The value of MAD relative to the mean of the data values (2.1240) is also given within parenthesis in the first column. Although we observe a small improvement around 1% in the relative MAD when α is increased from 0 to 4, the improvement is much more prominent when we examine the higher end of the 90% confidence interval of absolute deviation, given in the third column of Table 9. The improvement is around 0.87, which corresponds to an improvement of 25% relative to the data mean.

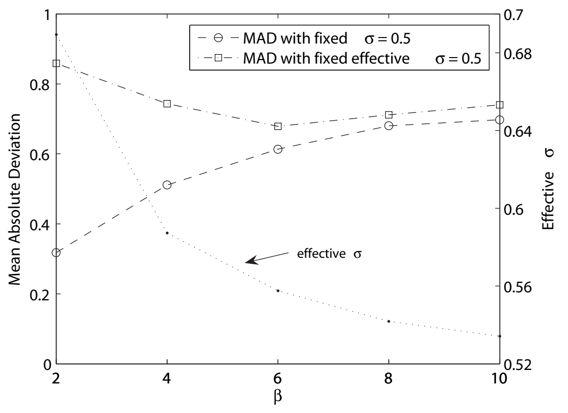


Figure 77: Effect of β on prediction error

5.6.2.3 Effect of β on the Prediction Error

As mentioned in Section 5.4.4, decreasing subcluster granularity parameter β is expected to increase effective σ. Higher effective σ implies larger number of sampler nodes and thus improves the error in prediction. Figure 77 illustrates this inference concretely, where the mean absolute derivation (MAD) of the error in predicted sample values and effective σ are plotted as a function of β. MAD is plotted with a dashed line and is read from the left y-axis, whereas effective σ is plotted with a dotted line and is read from the right y-axis. We see that decreasing β from 10 to 2 decreases MAD around 50% (from 0.44 to 0.22). However, this is mainly due to the fact that the average number of sampler nodes is increased by 26% (0.54 to 0.68). To understand the impact of β better and to decouple it from the number of sampler nodes, we fix effective σ to 0.5. Figure 77 plots MAD as a function of β for fixed effective σ, using a dash-dot line. It is observed that both small and large β values result in higher MAD whereas moderate values for β achieve smaller MAD.

This is very intuitive, since small sized models (small β) are unable to fully exploit the available correlations between node samples, whereas large sized models (large β) become ineffective due to decreased amount of correlation among the samples of large and diverse node groups.

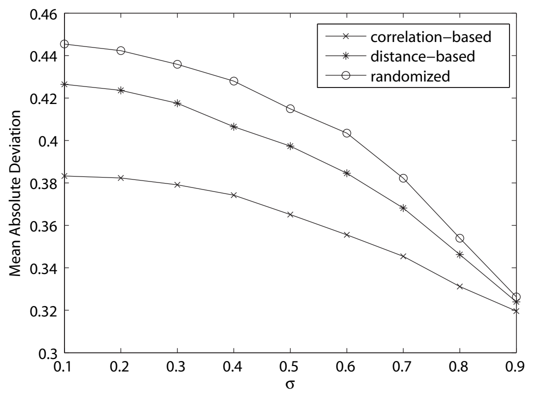


Figure 78: MAD with different subclusterings

5.6.2.4 Effect of Subclustering on the Prediction Error

This experiment intends to show how different subclustering methods can affect the prediction error. We consider three different methods: correlation-based subclustering (as described in Section 5.4), distance-based subclustering in which location closeness is used as the metric for deciding on subclusters, and randomized subclustering which uses purely random assignment to form subclusters. Figure 78 plots MAD as a function of σ for these three different methods of subclustering. The results listed in Figure 78 are averages of large number of subclusterings. We observe that randomized and distance-based subclustering perform up to 15% and 10% worse respectively, when compared correlation-based subclustering, in terms of mean absolute deviation of the error in value predication. The differences between these three methods in terms of MAD is largest when σ is smallest and disappears as σ approaches 1. This is quite intuitive, since smaller σ values imply that the prediction is performed with smaller number of sampler node values.

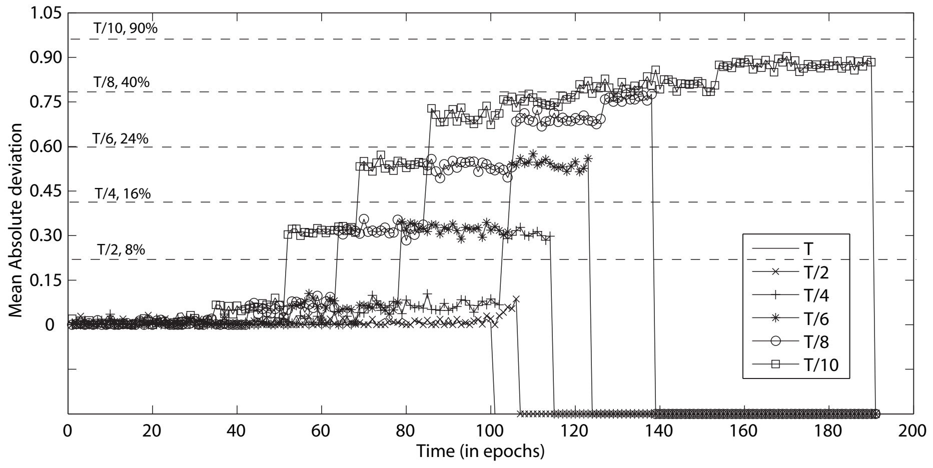


Figure 79: Prediction error vs. lifetime trade-off

5.6.2.5 Prediction Error/Lifetime Trade-off

We study the trade-off between prediction error and network lifetime by simulating selective sampling with dynamic σ adjustment for different σ reduction rates. We assume that the main source of energy consumption in the network is wireless messaging and sensing. We set up a scenario such that, without selective sampling the average lifetime of the network is $T = 100$ units. This means that, the network enables us to collect data with 100% accuracy for 100 time units and then dies out. For comparison, we use selective sampling and experiment with dynamically decreasing σ as time progresses, in order to gradually decrease the average energy consumption, while introducing an increasing amount of error in the collected data. Figure 79 plots the mean absolute deviation (MAD) as a function of time for different σ reduction rates. In the figure $T/x, x \in \{1, 2, 4, 6, 8, 10\}$ denotes different reduction rates, where σ is decreased by 0.1 every T/x time units. σ is not dropped below 0.1. A negative MAD value in the figure implies that the network has exceeded its lifetime. Although it is obvious that the longest lifetime is achieved with the highest reduction rate (easily read from the figure), most of the time it is more meaningful to think of lifetime as bounded by the prediction error. In other words, we define the *ϵ-bounded network lifetime* as the longest period during which the MAD is always below a user defined threshold ϵ. Different thresholds are plotted as horizontal dashed lines in the figure, crossing the y-axis. In order to find the σ reduction rate with the highest ϵ-bounded network lifetime,

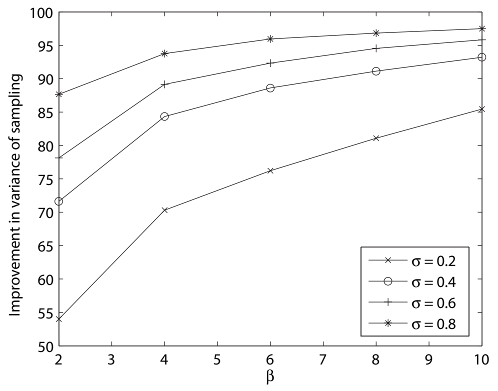


Figure 80: Load balance in schedule derivation

we have to find the error line that has the largest x-axis coordinate (lifetime) such that its corresponding y-axis coordinate (MAD) is below ϵ and above zero. Following this, the approach with the highest ϵ-bounded lifetime is indicated over each ϵ line together with the improvement in lifetime. We observe that higher reduction rates do not always result in a longer ϵ-bounded network lifetime. For instance, $T/4$ provides the best improvement (around 16%) when ϵ is around 0.4, whereas $T/8$ provides the best improvement (around 40%) when ϵ is around 0.8.

5.6.2.6 Load Balance in Sampler Selection

Although saving battery life (energy) and increasing average lifetime of the network through the use of selective sampling is desirable, it is also important to make sure that the task of being a sampler node is equally distributed among the nodes. To illustrate the effectiveness of our sampler selection mechanism in achieving the goal of load balance, we compare the variation in the amount of time nodes have served as a sampler between our sampler selection scheme and a scenario where the sampler nodes are selected randomly. The improvement in the variance (i.e., the percentage of decrease in variance when using our approach compared to randomized approach) is plotted as a function of β for different σ values in Figure 80. For all settings, we observe an improvement above 50% provided by our sampler selection scheme.

5.7 Discussions

Setting of Selective Sampling Parameters – There are a number of system parameters involved in our selective sampling approach to data collection in sensor networks. Most notable are: α, τ_d, τ_c, τ_u, τ_f, and β. We have described various trade-offs involved in setting these parameters. We now give a general and somewhat intuitive guideline for a base configuration of these parameters. Among these parameters, τ_d is the one that is most straightforward to set. τ_d is the desired sampling period and defines the temporal resolution of the collected data. A default value of 1 seconds can provide more than enough temporal resolution for most environmental monitoring applications. When domain-specific data distance functions are used during the clustering phase, a basic guide for setting α is to set it to 1. This results in giving equal importance to data distance and hop distance factors. The clustering period τ_c and schedule update period τ_u should be set in terms of the desired sampling period τ_d. Other than the cases where the phenomenon of interest is highly dynamic, it is not necessary to perform clustering and schedule update frequently. However, re-clustering and re-assigning sampling schedules help achieve better load balancing due to alternated sampler nodes and cluster heads. As a result, one balanced setting for these parameters is $\tau_c = 1$ hour and $\tau_u = 15$ minutes. From our experimental results, we conclude that these values result in very little overhead. The forced sampling period τ_f defines the number of the sample readings used for calculating the probabilistic model parameters. For the suggested setting of τ_u, having $\tau_f = 0.1 * \tau_u = 1.5$ minutes results in having 90 samples out of 900 readings per schedule update period. This is statistically good enough for calculating correlations. Finally, β is the subcluster granularity parameter and based on our experimental results, we suggest a reasonable setting of $\beta \in [5, 7]$. Note that both small and large values for beta will degrade the prediction quality.

Messaging Cost and Energy Consumption – One of the most energy consuming operations in sensor networks is the sending and receiving of messages, although the energy cost of keeping the radio in the active state also presents non-negligible cost [88]. It is important to note that our selective sampling approach to data collection operates in a completely periodic manner. By scheduling the selective sampling based data collection periodically, we ensure that during most of the time there is no messaging activity in the system. As a result, taking the number of messages exchanged during data collection as the major indicator of energy consumption is a sound assumption. Considering the fact that there exist generic protocols for exploiting such timing semantics found in most of the data collection applications [32], we can easily incorporate an exisiting energy efficient radio management protocol in order to save energy by reducing or avoiding the need for keeping the radio active during periods of inactivity.

Reliable Maintenance of Network Structures – There are three important network structures to be maintained in our selective sampling approach to data collection. These are the cluster head nodes, the cluster connection trees, and the data collection tree. Note that the cluster connection trees and the data collection tree are solely used for communicating with the cluster heads and the base node, respectively. This means that these trees need not be maintained in case there is a routing mechanism already existent in the network, such as geographical routing (ex. GPSR [70]) or other well-known ad-hoc routing mechanisms such as AODV, DSR, and DSDV (see [22]). The use of these trees in selective sampling is completely different than their use in aggregation schemes in which in-network processing is performed at each non-leaf node of the tree. In selective sampling, most of the in-network processing takes place in the cluster head nodes. As a result, there are two important issues that requires special attention for ensuring reliability. First, the failure of cluster head nodes should be detected and in case of failures new cluster head nodes should be elected. This can be achieved by applying classical primary/backup techniques from distributed computing. Second, it is important that the model parameters sent from the cluster head nodes are successfully transmitted to the base node. This can be achieved by employing end-to-end acknowledgment schemes between the cluster heads and the base node. Note that the loss of a message which includes a sensor reading destined to the base node is not a serious problem, since the value of such a node can be predicted at the base node using the probabilistic models and the readings of other nodes, albeit with some error. In comparison, the loss of a message in an aggregation scheme results in neglecting the values of all the nodes under a subtree.

5.8 *Related Work*

Energy efficiency plays a fundamental role in localized and distributed algorithms for wireless sensor networks in the context of various different services and protocols, such as broadcasting [64], message routing [110], medium access control [140], time synchronization [40], and location determination [96]. Data collection is another important service provided by sensor networks, especially for environmental monitoring applications. We review the literature related to sensor data collection in three categories: sensor data collection systems, node clustering in ad-hoc networks, and probabilistic inference in sensor networks.

5.8.1 Sensor Data Collection Systems

In Section 5.1 we have discussed the distinction between our selective sampling approach and other data collection approaches, such as those based on event detection [5], in-network aggregation [82], and distributed compression [95]. In summary, selective sampling is designed for energy efficient periodic collection of raw sensor readings from the network for the purpose of performing detailed data analysis that can not be done using in-network executed queries or locally detected events. The energy saving is a result of trading-off data accuracy, which is achieved by using a dynamically changing subset of nodes as samplers. This is in some ways similar to previously proposed energy saving sensor network topology formation algorithms, such as PEAS [139], where only a subset of nodes are made active, while preserving the network connectivity. Selective sampling uses a similar logic, but in a different context and for a different purpose: only a subset of nodes are used to actively sample, while the quality of the collected data is kept high.

There are a number of recent works [56, 39, 73] that has considered the trade-off between energy consumption and data collection quality. In [56] algorithms are proposed to minimize the sensor node energy consumption in the process of answering a set of user supplied queries with specified error thresholds. The queries are answered using uncertainty intervals cached at the server. These cached intervals are updated using an optimized schedule of server-initiated and sensor-initiated updates. Our selective sampling approach is not bound to queries and collects data periodically, so that both on-line and archival applications can make use of the collected data.

BBQ [39] is a model-driven data acquisition framework for sensor networks, that uses global optimizations for generating energy efficient data collection schedules. It is designed for multi-sensor systems, where nodes have multiple sensors with different energy consumption specifications. The correlations between readings from different sensors within a node are exploited to build statistical models, which enables prediction of the sensor readings that are energy-wise expensive to sample, from other sensor readings that are energy-wise cheap to sample, for instance temperature readings from voltage readings. Our selective sampling approach uses statistical models in a similar manner, but instead of modeling intra-node correlations among readings of different-type sensors, we model inter-node correlations among the readings of same-type sensors from different nodes. Moreover, unlike [39], our framework is not query bound.

Snapshot Queries [73] is perhaps the most relevant work to ours. In [73], each sensor node is either represented by one of its neighbors or it is a representative node. Although this division is similar to sampler and non-sampler nodes of our selective sampling approach, there is a fundamental difference. The neighboring relationship imposed on representative nodes imply that the number of representatives is highly dependent on the connectivity graph of the network. For instance, as the connectivity graph gets sparse, the number of representative nodes may grow relative to the total network size. This restriction does not apply to the number of sampler nodes in selective sampling, since the selection process is supported by a clustering algorithm and is not limited to one-hop neighborhoods. In [73], representative nodes predict the values of their dependent neighbors for the purpose of query evaluation. This can cut down the energy consumption dramatically for aggregate queries, since a single value will be produced as an aggregate from the value of the representative node and the predicted values of the dependent neighbors. However this local prediction will not support such savings when queries have holistic aggregates [82] or require collection of readings from all nodes. Thus, selective sampling employs a hybrid approach where prediction is performed outside the network. Moreover, the model based prediction performed in our selective sampling approach uses correlation based schedule derivation to subcluster nodes into groups based on how good these nodes are in predicting each other's value. Any node within the same cluster can be put into the same subcluster, independent of the neighboring relationship between them. As opposed to this, snapshot queries does not use a model and instead employs binary linear regression for each representative-dependent node pair.

5.8.2 Node Clustering in Ad-hoc Networks

With respect to ad-hoc networks, most of the previous work in distributed node clustering have focused on constructing one-hop clusters [76, 12, 14, 20, 29]. In a one-hop cluster, each node is at most one-hop away from its cluster head. A few exceptions to this line of work are [30], [4], and [99]. In [4], a heuristic-based distributed algorithm is introduced for building clusters in which each node is at most d hops away from its cluster head. d is a system parameter and the algorithm tends to create clusterings in which clusters have approximately the same size. In [30], several distributed clustering algorithms are proposed for constructing k-hop clusters, where each node is at most k-hops away from the cluster head. In [99], a connectivity-based distributed node clustering algorithm is proposed, where nodes that are "highly connected" in the connectivity graph are put into the same cluster. All these algorithms, though distributed, do not attempt to cluster the network based on sensing structure. Hence the clusters discovered are not necessarily "good" clusters from a prediction stand-point. In contrast, our clustering algorithm is unique in being sensing-driven, i.e., the criteria for clustering is data-centric, and results in clusters that improve prediction quality.

5.8.3 Inference in Sensor Networks

Our selective sampling approach to energy efficient data collection in sensor networks uses probabilistic models, whose parameters are locally inferred at the cluster head nodes and are later used at the base node to predict the values of non-sampler sensor nodes. Several recent works have also proposed to use probabilistic inference techniques to learn unknown variables within sensor networks [90, 52, 135, 23]. In [52], regression models are employed to fit a weighted combination of basis functions to the sensor field, so that a small set of regression parameters can be used to approximate the readings from the sensor nodes. In [23], probabilistic models representing the correlations between the sensor readings at various locations are used to perform distributed calibration. In [135], a distributed fusion scheme is described to infer a vector of hidden parameters that linearly relate to each sensor's reading with a Gaussian error. Finally, in [90] a generic architecture is presented to perform distributed inference in sensor networks. The solution employs message passing on distributed junction-trees, and can be applied to a variety of inference problems, such as sensor field modeling, sensor fusion, and optimal control.

CHAPTER VI

RESOURCE-AWARE JOIN EVALUATION FOR SENSOR CQ SYSTEMS

Tuple dropping, though commonly used for load shedding in most data stream operations, is generally inadequate for multi-way, windowed stream joins. The output rate can be unnecessarily degraded because it does not exploit time correlations that are likely to exist among interrelated streams. In this chapter, we introduce *GrubJoin*: an adaptive, multi-way, windowed stream join that effectively performs time correlation-aware CPU load shedding. GrubJoin maximizes the output rate by achieving near-optimal *window harvesting*, which picks only the most profitable segments of individual windows for the join and ignores the less valuable ones. Due mainly to the combinatorial explosion of possible multi-way join sequences involving segments of individual join windows, GrubJoin faces a set of unique challenges, such as determining the optimal window harvesting configuration and learning the time correlations among the streams. To tackle these challenges, we formalize window harvesting as an optimization problem, develop greedy heuristics to determine near-optimal window harvesting configurations and use approximation techniques to capture the time correlations among the streams. Experimental results show that GrubJoin is vastly superior to tuple dropping when time correlations exist and is equally effective when time correlations are nonexistent.

6.1 Introduction

In today's highly networked and digital world, businesses often rely on time-critical tasks that require analyzing data from on-line sources and generating responses in real-time. In many industries, the on-line data to be analyzed comes in the form of data streams, i.e., as time-ordered series of events or readings. Examples include stock tickers in financial services, link statistics in networking, sensor readings in environmental monitoring and emergency response, and surveillance data in Homeland Security. In these examples, rapidly increasing rates of data streams and stringent response time requirements of applications force a paradigm shift in how the data are processed, moving away from the traditional "store and then process" model of database management systems (DBMSs) to "on-the-fly processing" model of emerging data stream management systems (DSMSs). This shift has recently created a strong interest in research on DSMS-related topics, in both academia [6, 10, 27] and industry [117].

In DSMSs, CPU load shedding is needed when the available processing resources are not sufficient to handle the processing demands of the continuous queries installed in the system, under the current rates of the input streams. Without load shedding, the mismatch between the available resources and the demands will result in delays that violate the response time requirements of the queries. It will also cause an unbounded growth in system queues that overload memory capacity and further bog down the system. As a solution to these problems, CPU load shedding can be broadly defined as a mechanism to reduce the amount of processing performed for evaluating stream queries, in an effort to match the service rate of a DSMS to its input rate, at the cost of producing a degraded output. Depending on the stream operators, the output degradation may take different forms, such as a lower resolution in a multimedia encoder operator or a subset result in a join operator.

Joins are key operators in DSMSs and costlier to evaluate when compared with others, such as selections and projections. They are used by many applications to correlate events from various streams. For example, let us look at two stream join applications here.

Example 1 - Finding similar news items from different news sources: Assuming that news items from CNN, Reuters, and BBC are represented by weighted keywords (join attribute) in their respective streams, we can perform a windowed inner product join to find similar news items from different sources.

Example 2 [53] - *Tracking objects using multiple video (sensor) sources*: Assuming that scenes (readings) from video (sensor) sources are represented by multi-attribute tuples of numerical values (join attribute), we can perform a distance-based similarity join to detect objects that appear in all of the video (sensor) sources.

Hence, it is important to study load shedding techniques in the context of stream joins, particularly in the face of bursty and unpredictable stream rates. In this chapter, we focus on CPU load shedding for multi-way (usually more than two), windowed stream joins. Join operations are performed on the tuples stored within user-defined, time-based join windows, which constitute one of the most common join types in the DSMS research [9, 49, 68].

So far, the predominantly used approach to CPU load shedding in stream joins has been tuple dropping [8, 123]. This can be seen as a *stream throttling* approach, where the rates of the input streams are sufficiently reduced via the use of tuple dropping, in order to sustain a stable system. However, tuple dropping generally is ineffective in shedding CPU load for multi-way, windowed stream joins. The output rate of a multi-way join can be unnecessarily degraded because tuple dropping does not recognize, hence fails to exploit, time correlations that are likely to exist among interrelated streams. Time correlations exist because causal events manifest themselves in these interrelated streams at different, but correlated, times.

The time-correlation assumption indicates that, for pairs of matching tuples from two streams, there exists a non-flat match probability distribution which is a function of the time difference between the timestamps of the tuples. For instance, in Example 1 above, it is more likely that a news item from one source will match with a temporally close news item from another source. In this case the streams are almost aligned and the probability that a tuple from one stream will match with a tuple from another stream decreases as the difference between their timestamps increases. However, the streams can also be unaligned, either due to delays in the delivery path, such as network and processing delays, or due to the inherent effect of time of event generation. As an illustration to the unaligned case, in Example 2 above, similar tuples appearing in different video streams or similar readings found in different sensor streams will have a *lag* between their timestamps, due to the time it takes for an object to pass through all cameras or all sensors.

In this chapter, we present *GrubJoin*[1]: an adaptive, multi-way, windowed stream join that effectively performs time correlation-aware CPU load shedding. While shedding load,

[1]As an intransitive verb, grub means "to search laboriously by digging". It relates to the way that the most profitable segments of individual join windows are picked and processed with window harvesting in order to maximize the join output.

GrubJoin maximizes the output rate by achieving near-optimal *window harvesting* within an *operator throttling* framework. In contrast to stream throttling, operator throttling performs load shedding within the stream operator, i.e., regulating the amount of work performed by the join. This requires altering the processing logic of the multi-way join by parameterizing it with a *throttle fraction*. The parameterized join incurs only a throttle fraction of the processing cost required to perform the full join operation. As a side effect, the quality or the quantity of the output produced may be decreased when load shedding is performed.

While shedding CPU load, window harvesting maximizes the output rate by picking only the most profitable segments of individual join windows for the join operations while ignoring the less valuable ones, similar to farmers harvesting fruits, such as strawberries, by picking only the ripest while leaving the less ripe untouched. For efficient implementation, GrubJoin divides each join window into multiple, small-sized segments of basic windows. Due mainly to the combinatorial explosion of possible, multi-way join sequences involving segments of different join windows, GrubJoin faces a set of challenges in performing window harvesting. These challenges are unique for a multi-way, windowed stream join and they do not exist for a two-way, windowed stream join. In particular, there are three major challenges:

− First, mechanisms are needed to configure window harvesting parameters so that the throttle fraction imposed by operator throttling is respected. We should also be able to assess the optimality of these mechanisms in terms of output rate, with respect to the best achievable for a given throttle fraction and known time correlations between the streams.

− Second, in order to be able to react and adapt to the possibly changing stream rates in a timely manner, the reconfiguration of window harvesting parameters must be a lightweight operation, so that the processing cost of reconfiguration does not consume the processing resources used to perform the join.

− And third, we should develop low cost mechanisms for learning the time correlations among the streams, in case they are not known or are changing and should be adapted.

We tackle the first challenge by developing a cost model and formulating window harvesting as an optimization problem. We handle the latter two challenges by developing GrubJoin - a multi-way stream join algorithm that employs *i*) greedy heuristics for making near-optimal window harvesting decisions, and *ii*) approximation techniques to capture time correlations among the streams.

To the best of our knowledge, this is the first work on time correlation-aware CPU load shedding for multi-way, windowed stream joins that are adaptive to the input stream rates. However, we are not the first to recognize and take advantage of the time correlation effect in join processing. In the context of two-way stream joins with limited memory, the *age-based* load shedding framework of [115] pointed out the importance of time correlation effect and exploited it to make tuple replacement decisions. Furthermore, in the context of traditional joins, the database literature includes join operators, such as Drag-Join [58], that capitalized on the time of data creation effect in data warehouses, which is very similar to the time correlation effect in stream joins.

6.1.1 Summary of Contributions

In summary, this chapter makes three major contributions:

1) We introduce window harvesting as an in-operator load shedding technique for multi-way, windowed stream joins, that can adjust the amount of shedding performed based on the throttle fraction defined by our operator throttling framework. We formalize window harvesting configuration as an optimization problem and show how it can be utilized to exploit the time correlations among the streams to maximize the output rate of the join.

2) We develop the GrubJoin algorithm, that can adapt to the changes in the input stream rates, the current system load, and the time correlations among the streams. It performs near-optimal window harvesting and has very low overhead, thanks to the heuristic methods it employs for performing reconfiguration of harvesting parameters, and the approximation techniques it uses to learn the time correlations among the streams.

3) We report results from our experimental studies and show that GrubJoin performs vastly superior to tuple dropping in terms of output rate when time correlations exist among the

streams, and is equally effective as tuple dropping in the absence of correlations.

6.2 *Preliminaries*

Before going into the details of operator throttling and window harvesting, in this section we present our window-based stream join model, introduce some notations, and describe the basics of multi-way, windowed stream join processing.

We denote the ith input stream by S_i, where $i \in [1..m]$ and $m \geq 2$ denotes the number of input streams of the join operator, i.e., we have an m-way join. Each stream is a sequence of tuples ordered by an increasing timestamp. We denote a tuple by t and its timestamp by $T(t)$. Current time is denoted by T. We assume that tuples are assigned timestamps upon their entrance to the DSMS. We do not enforce any particular schema type for the input streams. Schemas of the streams can include attributes that are single-valued, set-valued, user-defined, or binary. The only requirement is to have timestamps and an appropriate join condition defined over the input streams. We denote the current rate, in terms of tuples per second, of an input stream S_i as λ_i.

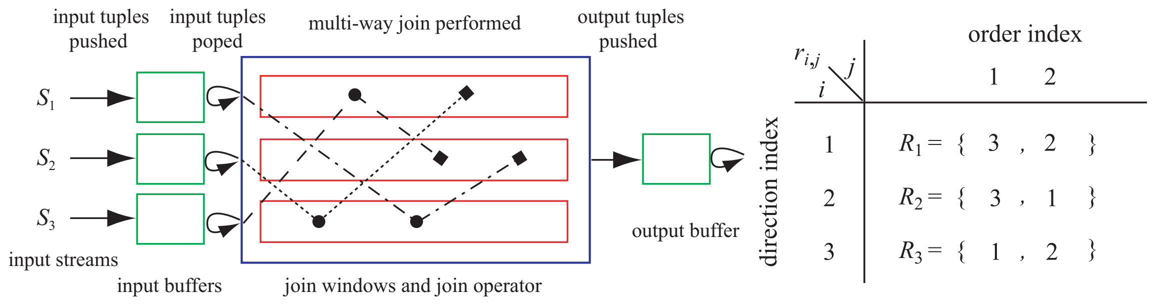


Figure 81: Multi-way, windowed stream join processing, join directions, and join orders

An m-way stream join operator has m join windows, as shown in the 3-way join example of Figure 81. The join window for stream S_i is denoted by W_i, and has a user-defined size, in terms of seconds, denoted by w_i. A tuple t from S_i is kept in W_i only if $T \geq T(t) \geq T - w_i$. The join operator has buffers (queues) attached to its inputs and output. The input stream tuples are pushed into their respective input buffers either directly from their source or from output of other operators. The join operator processes tuples by fetching them from its input buffers, processes the join, and pushes the resulting tuples into the output buffer.

The GrubJoin algorithm we develop in this chapter can be seen as a descendant of MJoin [128]. MJoins have been shown to be very effective for performing fine-grained adaptation and are very suitable for streaming scenarios, where the rates of the streams are bursty and may soar during peak times. In an MJoin, there are m different *join directions*, one for each stream, and for each join direction there is an associated *join order.* The ith direction of the join describes how a tuple t from S_i is processed by the join algorithm, after it is fetched from the input buffer. The join order for direction i, denoted by $R_i = \{r_{i,1}, r_{i,2}, \ldots, r_{i,m-1}\}$, defines an ordered set of window indexes that will be used during the processing of $t \in S_i$. In particular, tuple t will first be matched against the tuples in window W_l, where $l = r_{i,1}$. Here, $r_{i,j}$ is the jth join window index in R_i. If there is a match, then the index of the next window to be used for further matching is given by $r_{i,2}$, and so on. For any direction, the join order consists of $m-1$ distinct window indices, i.e., R_i is a permutation of $\{1, \ldots, m\} - \{i\}$. Although there are $(m-1)!$ possible choices of orderings for each join direction, this number can be smaller depending on the join graph of the particular join at hand. We will talk more about join order selection in Section 6.5. Figure 81 illustrates join directions and orders for an example 3-way join. Once the join order for each direction is decided, the processing is carried out in an NLJ (nested-loop join) fashion. Since we do not focus on any particular type of join condition, NLJ is a natural choice. Section 6.7 will discuss about applying indexed processing to our approach, for special types of joins.

6.3 Operator Throttling

Operator throttling is a load shedding framework for stream operators. It regulates the amount of load shedding to be performed by calculating and maintaining a throttle fraction, and relies on an in-operator load shedding technique to reduce the CPU cost of executing the operator in accordance with the throttle fraction. We denote the throttle fraction by z. It has a value in the range $(0, 1]$. Concretely, $z = \phi$ means that the in-operator load shedding technique should adjust the processing logic of the operator such that the CPU cost of executing it is reduced to ϕ times the original. As expected, this will have side-effects on the quality or quantity of the output from the operator. In the case of stream joins, applying in-operator load shedding will result in a reduced output rate. Note that the concept of operator throttling is general and applies to operators other than joins. For instance, an aggregation operator can use the throttle fraction to adjust its aggregate re-evaluation interval to shed load [122], or a data compression operator can decrease its compression ratio based on the throttle fraction [97].

6.3.1 Setting of the Throttle Fraction

The correct setting of the throttle fraction depends on the performance of the join operator under current system load and the incoming stream rates. We capture this as follows.

Let us denote the adaptation interval by Δ. This means that the throttle fraction z is adjusted every Δ seconds. Let us denote the tuple consumption rate of the join operator for S_i, measured for the last adaptation interval, by α_i. In other words, α_i is the tuple pop rate of the join operator for the input buffer attached to S_i, during the last Δ seconds. On the other hand, let λ'_i be the tuple push rate for the same buffer during the last adaptation interval. Using α_i's and λ'_i's we capture the performance of the join operator under current system load and incoming stream rates, denoted by β, as:

$$\beta \;=\; \sum_{i=1}^{m} \alpha_i / \sum_{i=1}^{m} \lambda'_i$$

The β value is used to adjust the throttle fraction as follows. We start with a z value of 1, optimistically assuming that we will be able to fully execute the operator without any overload. At each adaptation step (Δ seconds), we update z from its old value z^{old} based on the formula:

$$z \;=\; \begin{cases} \beta \cdot z^{old} & \beta < 1 \\ min(1, \gamma \cdot z^{old}) & \text{otherwise} \end{cases}$$

If β is smaller than 1, z is updated by multiplying its old value with β, with the aim of adjusting the amount of shedding performed by the in-operator load shedder to match the tuple consumption rate of the operator to tuple production rate of the streams. Otherwise

($\beta \geq 1$), the join is able to process all the incoming tuples with the current setting of z, in a timely manner. In this latter case, z is set to minimum of 1 and $\gamma \cdot z^{old}$, where γ is called the *boost factor*. This is aimed at increasing the throttle fraction, assuming that additional processing resources are available. If not, the throttle fraction will be readjusted during the next adaptation step. Note that, higher values of the boost factor result in being more aggressive at increasing the throttle fraction.

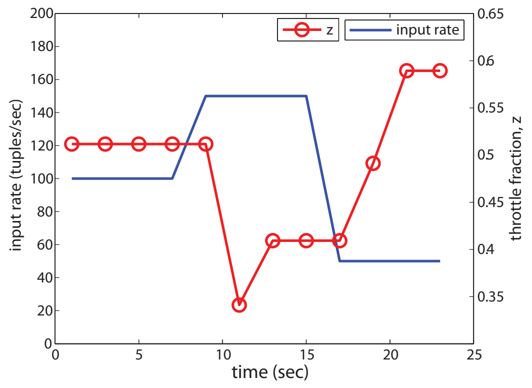


Figure 82: z adaptation example

Figure 82 shows an example of throttle fraction adaptation from our implementation of GrubJoin using operator throttling. In this example Δ is set to 4 seconds and γ is set to 1.2. Other experimental parameters are not of interest for this example. The input stream rates are shown as a function of time using the left y-axis, and the throttle fraction z is shown as a function of time using the right y-axis. Looking at the figure, besides the observation that the z value adapts to the changing rates by following an inversely proportional trend, we also see that the reaction in the throttle fraction follows the rate change events with a delay due to the length of the adaptation interval. Although in this example Δ is sufficiently small to adapt to the bursty nature of the streams, in general its setting is closely related with the length of the bursts. Moreover, the time it takes for the in-operator load shedder to perform reconfiguration in accordance with the throttle fraction is an important limitation in how frequent the adaptation can be performed, thus how small Δ can be. We discuss more about this in Section 6.4.2.

6.3.2 Buffer Capacity vs. Tuple Dropping

As opposed to stream throttling, operator throttling does not necessarily drop tuples from the incoming streams. The decision of how the load shedding will be performed is left to the in-operator load shedder, which may choose to retain all unexpired tuples within its join windows. However, depending on the size of the input buffers, operator throttling framework may still result in dropping tuples outside the join operator, albeit only during times of mismatch between the last set value of the throttle fraction and its ideal value. As an example, consider the starting time of the join, at which point we have $z = 1$. If the stream rates are higher than the operator can handle with z set to 1, then the gap between the incoming tuple rate and the tuple consumption rate of the operator will result in growing number of tuples within buffers. This trend will continue until the next adaptation step, at which time throttle fraction will be adjusted to stabilize the system. However, if during this interval the buffers fill up, then some tuples will be dropped. The buffer size can be increased to prevent tuple dropping, at the cost of introducing delay. If buffer sizes are small, then tuple dropping will be observed only during times of transition, during which throttle fraction is higher than what it should ideally be.

6.4 Window Harvesting

Window harvesting is an in-operator load shedding technique we develop for multi-way, windowed stream joins. The basic idea behind window harvesting is to use only the most profitable segments of the join windows for processing, in an effort to reduce the CPU demand of the operator, as dictated by the throttle fraction. By making use of the time correlations among the streams in deciding which segments of the join windows are most valuable for output tuple generation, window harvesting aims at maximizing the output rate of the join. In the rest of this section, we first describe the fundamentals of window harvesting and then formulate window harvesting as an optimization problem.

6.4.1 Fundamentals

Window harvesting involves organizing join windows into a set of *basic windows* and, for each join direction, selecting the most valuable segments of the windows to use for performing the join.

6.4.1.1 Basic Windows

Each join window W_i is divided into basic windows of size b seconds. Basic windows are treated as integral units, thus there is always one extra basic window in each join window to handle tuple expiration. In other words, W_i consists of $1 + n_i$ basic windows, where $n_i = \lceil w_i/b \rceil$. The first basic window is partially full, and the last basic window contains some expired tuples (tuples whose timestamps are out of the join window's time range, i.e., $T(t) < T - w_i$). Every b seconds the first basic window fills completely and the last basic window expires totally. Thus, the last basic window is emptied and it is moved in front of the basic window list as the new first basic window.

At any time, the unexpired tuples in W_i can be organized into n_i *logical* basic windows, where the jth logical basic window ($j \in [1..n_i]$), denoted by $B_{i,j}$, corresponds to the ending ϑ portion of the jth basic window plus the beginning $1-\vartheta$ portion of the $(j+1)$th basic window. We have $\vartheta = \delta/b$, where δ is the time elapsed since the last basic window expiration took place. It is important to note that, a logical basic window always stores tuples belonging to a fixed time interval *relative* to the current time. This small distinction between logical and real basic windows become handy when selecting the most profitable segments of the join windows to process. Note that, accessing the tuples in a logical basic window does not require a search operation. A basic window is organized as a timestamp-ordered, doubly-linked list. As a result, for accessing tuples in $B_{i,j}$ we can perform iteration over the $(j+1)$th basic window and backward iteration over the jth basic window. Concepts related with basic windows are illustrated in Figure 83.

There are two major advantages of using basic windows. First, basic windows make expired tuple management more efficient [48]. This is because the expired tuples are removed from the join windows in batches, i.e., one basic window at a time. Second, without basic windows, accessing tuples in a logical basic window will require a search operation to locate a tuple within the logical basic window's time range. In general, small basic windows are more advantageous in better capturing and exploiting the time correlations. On the other hand, too small basic window sizes will cause overhead in join processing as well as in window harvesting configuration. This trade-off is studied in Section 6.6.

6.4.1.2 Configuration Parameters

There are two sets of configuration parameters for window harvesting, which together determine the segments of the windows that will be used for join processing. These are:

- *Harvest fractions*; $z_{i,j}, i \in [1..m], j \in [1..m-1]$: For the ith direction of the join, the fraction of the jth window in the join order (i.e., join window W_l, where $l = r_{i,j}$) that will be used for join processing is determined by the harvest fraction parameter $z_{i,j} \in (0, 1]$. There are $m \cdot (m-1)$ different harvest fractions. The settings of these fractions are strongly tied with the throttle fraction and the time correlations among the streams. The details will be presented in Section 6.4.2.

- *Window rankings*; $s^v_{i,j}, i \in [1..m], j \in [1..m-1], v \in [1..n_{r_{i,j}}]$: For the ith direction of the join, we define an ordering over the logical basic windows of the jth window in the join order (i.e., join window W_l, where $l = r_{i,j}$), such that $s^v_{i,j}$ gives the index of the logical basic window that has rank v in this ordering. $B_{l,s^1_{i,j}}$ is the first logical basic window in this order, i.e., the one with rank 1. The ordering defined by $s^v_{i,j}$ values is strongly influenced by the time correlations among the streams (see Section 6.4.2 for details).

In summary, the most profitable segments of the join window W_l, where $l = r_{i,j}$, that will be processed during the execution of the ith direction of the join is selected as follows. We first pick $B_{l,s^1_{i,j}}$, then $B_{l,s^2_{i,j}}$, and so on, until the total fraction of W_l processed reaches $z_{i,j}$. Other segments of window W_l that are not picked are ignored and not used during the execution of the join.

Figure 83 shows an example of window harvesting for a 3-way join, for the join direction R_1. In the example, we have $n_i = 5$ for $i \in [1..3]$. This means that we have 5 logical basic windows within each join window and as a result 6 basic windows per join window in

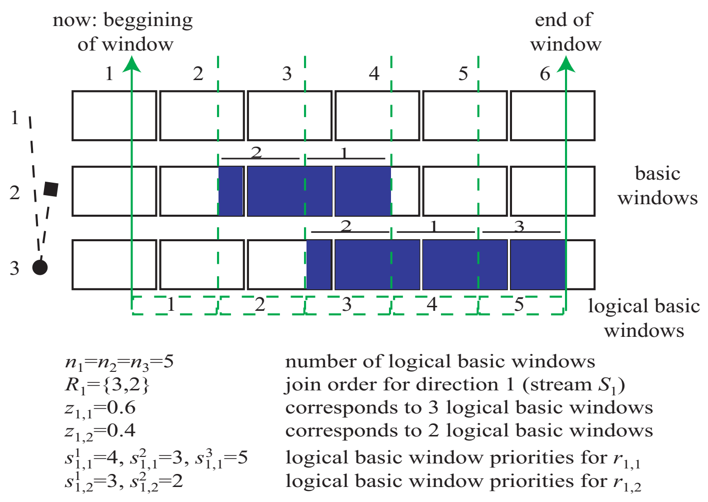

now: beggining of window
end of window
1
2
3
4
5
6
1
2
3
2
1
2
1
3
basic windows
1
2
3
4
5
logical basic windows
n_1=n_2=n_3=5 number of logical basic windows
R_1={3,2} join order for direction 1 (stream S_1)
$z_{1,1}$=0.6 corresponds to 3 logical basic windows
$z_{1,2}$=0.4 corresponds to 2 logical basic windows
$s^1_{1,1}$=4, $s^2_{1,1}$=3, $s^3_{1,1}$=5 logical basic window priorities for $r_{1,1}$
$s^1_{1,2}$=3, $s^2_{1,2}$=2 logical basic window priorities for $r_{1,2}$

Figure 83: Example of window harvesting

practice. The join order for direction 1 is given as $R_1 = \{3, 2\}$. This means W_3 is the first window in the join order of R_1 (i.e., $r_{1,1} = 3$) and W_2 is the second (i.e., $r_{1,2} = 2$). We have $z_{1,1} = 0.6$. This means that $n_{r_{1,1}} \cdot z_{1,1} = 5 \cdot 0.6 = 3$ logical basic windows from $W_{r_{1,1}} = W_3$ are to be processed. Noting that we have $s^1_{1,1} = 4$, $s^2_{1,1} = 3$, and $s^3_{1,1} = 5$, the logical basic windows within W_3 that are going to be processed are selected as $3, 4$, and 5. They are marked in the figure with horizontal lines, with their associated rankings written on top. The corresponding portions of the basic windows are also shaded in the figure. Note that there is a small shift between the logical basic windows and the actual basic windows (recall ϑ from Section 6.4.1.1). Along the similar lines, the logical basic windows 2 and 3 from W_2 are also marked in the figure, noting that $r_{1,2} = 2$, $z_{1,2} = 0.4$ corresponds to 2 logical basic windows, and we have $s^1_{1,2} = 3$, $s^2_{1,2} = 2$.

In the rest of this section, we describe the setting of window harvesting configuration parameters.

6.4.2 Configuration of Window Harvesting

Configuration of window harvesting involves setting the window ranking parameters and the harvest fraction parameters. This configuration is performed at the adaptation step, every Δ seconds.

6.4.2.1 Setting of Window Rankings

We set window ranking parameters $s^v_{i,j}$'s in two steps. First step is called *score assignment*. Concretely, for the ith direction of the join and the jth window in the join order R_i, that is W_l where $l = r_{i,j}$, we assign a *score* to each logical basic window within W_l. We denote the score of the kth logical basic window, which is $B_{l,k}$, by $p^k_{i,j}$. We define $p^k_{i,j}$ as the probability that an output tuple $(\ldots, t^{(i)}, \ldots, t^{(l)}, \ldots)$ has:

$$b \cdot (k-1) \leq T(t^{(i)}) - T(t^{(l)}) \leq b \cdot k$$

Here, $t^{(i)}$ denotes a tuple from S_i. This way, a logical basic window in W_l is scored based on the likelihood of having an output tuple whose encompassed tuples from S_i and S_l have an offset between their timestamps such that this offset is within the time range of the logical basic window.

The score values are calculated using the time correlations among the streams. For now, we will assume that the time correlations are given in the form of probability density functions (*pdf*s) denoted by $f_{i,j}$, where $i, j \in [1..m]$. Let us define $A_{i,j}$ as a random variable representing the difference $T(t^{(i)}) - T(t^{(j)})$ in the timestamps of tuples $t^{(i)}$ and $t^{(j)}$ encompassed in an output tuple of the join. Then $f_{i,j} : [-w_i, w_j] \rightarrow [0, \infty)$ is the probability density function for the random variable $A_{i,j}$. With this definition, we have $p^k_{i,j} = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. In practice, we develop a lightweight method for approximating a subset of these pdfs and calculating $p^k_{i,j}$'s from this subset efficiently. The details are given in Section 6.5 as part of the GrubJoin.

The second step of the setting of window ranking parameters is called *score ordering*. In this step, we sort the scores $\{p^k_{i,j} : k \in [1..n_{r_{i,j}}]\}$ in descending order and set $s^v_{i,j}$ to k, where v is the rank of $p^k_{i,j}$ in the sorted set of scores. If the time correlations among the streams change, then a new set of scores and thus a new assignment for the window rankings is needed. This is again handled by the reconfiguration performed at every adaptation step.

6.4.2.2 Setting of Harvest Fractions

Harvest fractions are set by taking into account the throttle fraction and the time correlations among the streams. First, we have to make sure that the CPU cost of performing the join agrees with the throttle fraction z. This means that the cost should be at most equal to z times the cost of performing the full join. Let $C(\{z_{i,j}\})$ denote the cost of performing the join for the given setting of the harvest fractions, and $C(\mathbf{1})$ denote the cost of performing the full join. We say that a particular setting of harvest fractions is feasible if and only if $z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\})$.

Second, among the feasible set of settings of the harvest fractions, we should prefer the one that results in the maximum output rate. Let $O(\{z_{i,j}\})$ denote the output rate of the join operator for the given setting of the harvest fractions. Then our objective is to maximize $O(\{z_{i,j}\})$. In short, we have an optimization problem:

Optimal Window Harvesting Problem:

$$\begin{array}{ll} \underset{\{z_{i,j}\}}{argmax} & O(\{z_{i,j}\}) \\ \mathbf{s.t.} & z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\}) \end{array}$$

The formulations for functions C and O are given in Appendix E. Our formulations are similar to previous work [68, 8], with the exception that we integrate time correlations among the streams into the processing cost and output rate computations.

6.4.3 Bruteforce Solution

One way to solve the optimal window harvesting problem is to enumerate all possible harvest fraction settings assuming that the harvest fractions are set to result in selecting an integral number logical basic windows, i.e., $\forall_{\substack{i \in [1..m] \\ j \in [1..m-1]}}, z_{i,j} \cdot n_{r_{i,j}} \in \mathbb{N}$. Although straightforward to implement, this bruteforce approach results in considering $\prod_{i=1}^{m} n_i^{m-1}$ possible configurations. If we have $\forall i \in [1..m], n_i = n$, then we can simplify this as $\mathcal{O}(n^{m^2})$. As we will show in the experimental section, this is computationally very expensive due to the long time required to solve the optimization problem with enumeration, and makes it almost impossible to perform frequent adaptation. In the next section we will discuss an efficient heuristic that can find near-optimal solutions quickly, with much smaller computational complexity.

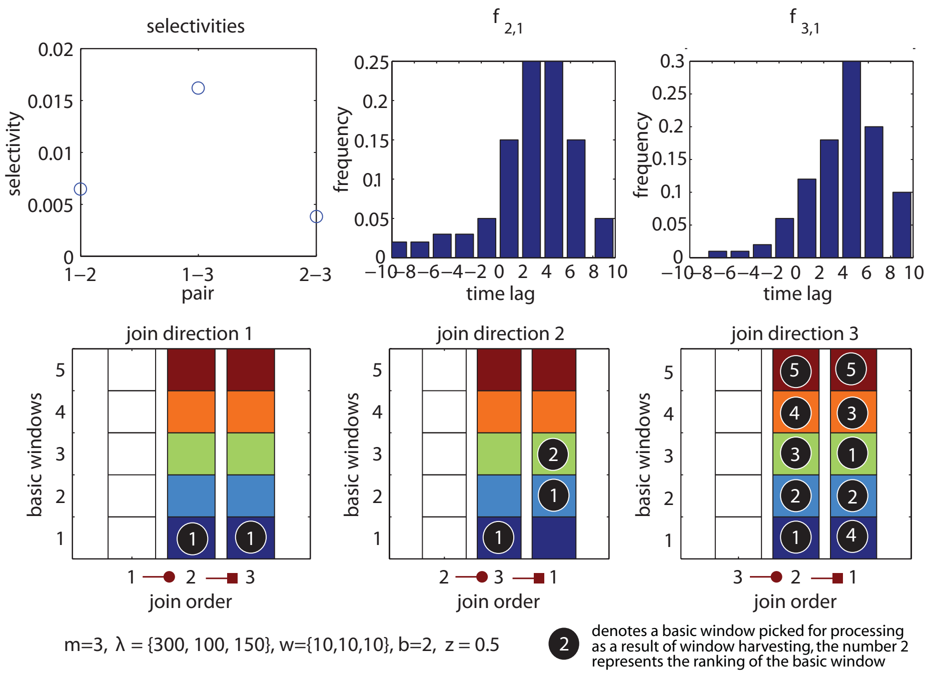


Figure 84: Optimal window harvesting example

6.4.3.1 Example of Optimal Configuration

Figure 84 shows an example scenario illustrating the setting of window harvesting parameters optimally. In this scenario we have a 3-way join with $\lambda_1 = 300$, $\lambda_2 = 100$, $\lambda_3 = 150$, $w_1 = w_2 = w_3 = 10$, $b = 2$, and $z = 0.5$. The topmost graph on the left in Figure 84 shows the selectivities, whereas the two graphs next to it show the time correlation pdfs, $f_{2,1}$ and $f_{3,1}$. By looking at $f_{2,1}$, we can see that there is a time lag between the streams S_1 and S_2, since most of the matching tuples from these two streams have a timestamp difference of about 4 seconds, S_2 tuple being lagged. Moreover, the probability that two tuples from S_1 and S_2 match decreases as the difference in their timestamps deviates from the 4 second time lag. By looking at $f_{3,1}$, we can say that the streams S_1 and S_3 are also unaligned, with S_3 lagging behind by around 5 seconds. In other words, most of the S_3 tuples match with S_1 tuples that are around 5 seconds older. By comparing $f_{2,1}$ and $f_{3,1}$, we can also deduce that S_3 is slightly lagging behind S_2, by around 1 seconds. As a result, our intuition tells us that the third join direction is more valuable than the others, since the tuples from other streams that are expected to match with an S_3 tuple are already within the join windows when an S_3 tuple is fetched. In this example, the join orders are configured as follows: $R_1 = \{2, 3\}, R_2 = \{3, 1\}, R_3 = \{2, 1\}$. This decision is based on the low selectivity first heuristic [128], as it will be discussed in the next section. The resulting window harvesting configuration, obtained by solving the optimal window harvesting problem by using the bruteforce approach, is shown in the lower row of Figure 84. The logical basic windows selected for processing are marked with dark circles and the selections are shown for each join direction. We observe that in the resulting configuration we have $z_{3,1} = z_{3,2} = 1$, since all the logical basic windows are selected for processing in R_3. This is inline with our intuition that the third direction of the join is more valuable than the others.[2]

6.5 GrubJoin

GrubJoin is a multi-way, windowed stream join operator with built-in window-harvesting. It uses two main methods to make window harvesting work in practice. First, it employs a heuristic method to set the harvest fractions, and second it uses approximation techniques to learn the time correlations among the streams and to set the logical basic window scores based on that. In this section we describe the details of these two methods.

6.5.1 Heuristic Setting of Harvest Fractions

The heuristic method we use for setting the harvest fractions is greedy in nature. It starts by setting $z_{i,j} = 0, \forall i, j$. At each greedy step it considers a set of settings for the harvest fractions, called the *candidate set*, and picks the one with the highest *evaluation metric* as the new setting of the harvest fractions. Any setting in the candidate set must be a forward step in increasing the $z_{i,j}$ values, i.e., we must have $\forall i, j, z_{i,j} \geq z^{old}_{i,j}$, where $\{z^{old}_{i,j}\}$ is the setting of the harvest fractions that was picked at the end of the previous greedy step. The process terminates once a step with an empty candidate set is reached. We introduce three different evaluation metrics for deciding on the best configuration within the candidate set. In what follows, we first describe the candidate set generation and then introduce the three alternative evaluation metrics.

[2]See a demo at http://disl.cc.gatech.edu/SensorCQ/optimizer.html

6.5.1.1 Candidate Set Generation

The candidate set is generated as follows. For the ith direction of the join and the jth window within the join order R_i, we add a new setting into the candidate set by increasing $z_{i,j}$ by $d_{i,j}$. In the rest of the chapter we take $d_{i,j}$ as $1/n_{r_{i,j}}$. This corresponds to increasing the number of logical basic windows selected for processing by one. This results in $m \cdot (m-1)$ different settings, which is also the maximum size of the candidate set. The candidate set is then *filtered* to remove the settings which are infeasible, i.e., do not satisfy the processing constraint of the optimal window harvesting problem dictated by the throttle fraction z. Once a setting in which $z_{u,v}$ is incremented is found to be infeasible, then the harvest fraction $z_{u,v}$ is *frozen* and no further settings in which $z_{u,v}$ is incremented are considered in the future steps of the algorithm.

There is one small complication to the above described way of generating candidate sets. Concretely, when we have $\forall j, z_{i,j} = 0$ for the ith join direction at the start of a greedy step, then it makes no sense to create a candidate setting in which only one harvest fraction is non-zero for the ith join direction. This is because no join output can be produced from a join direction if there is one or more windows in the join order for which the harvest fraction is set to zero. As a result, we say that a join direction i is not *initialized* if and only if there is a j such that $z_{i,j} = 0$. If at the start of a greedy step, we have a join direction that is not initialized, say ith direction, then instead of creating $m - 1$ candidate settings for the ith direction, we generate only one setting in which all the harvest fractions for the ith direction are incremented, i.e., $\forall j, z_{i,j} = d_{i,j}$.

Computational Complexity: In the worst case, the greedy algorithm will have $(m-1) \cdot \sum_{i=1}^{m} n_i$ steps, since at the end of each step at least one harvest fraction is incremented for a selected join direction and window within that direction. Taking into account that the candidate set can have a maximum size of $m \cdot (m-1)$ for each step, the total number of settings considered during the execution of the greedy heuristic is bounded by $m \cdot (m-1)^2 \cdot \sum_{i=1}^{m} n_i$. If we have $\forall i \in [1..m], n_i = n$, then we can simplify this as $\mathcal{O}(n \cdot m^4)$. This is much better than the $\mathcal{O}(n^{m^2})$ complexity of the exhaustive algorithm, and as we will show in the next section it has satisfactory running time performance.

```
GreedyPick($z$)
(1)  $cO \leftarrow cC \leftarrow 0$ {current cost and output}
(2)  $\forall_{1 \le i \le m}, I_i \leftarrow$ false {initialization indicators}
(3)  $\forall_{1 \le j \le m-1}^{1 \le i \le m}, F_{i,j} \leftarrow$ false {frozen fraction indicators}
(4)  $\forall_{1 \le j \le m-1}^{1 \le i \le m}, z_{i,j} \leftarrow 0$ {fraction parameters}
(5)  while true
(6)      $bS \leftarrow 0$ {best score for this step}
(7)      $u \leftarrow v \leftarrow -1$ {direction and window indices}
(8)      for $i \leftarrow 1$ to $m$ {for each direction}
(9)          if $I_i =$ true {if already initialized}
(10)             for $j \leftarrow 1$ to $m-1$ {for each window in join order}
(11)                 if $z_{i,j} = 1$ or $F_{i,j} =$ true {$z_{i,j}$ is maxed or frozen}
(12)                     continue{move to next setting}
(13)                 $z' \leftarrow z_{i,j}$ {store old value}
(14)                 $z_{i,j} \leftarrow \text{Min}(1, z_{i,j} + d_{i,j})$ {increment}
(15)                 $S \leftarrow \text{Eval}(z, \{z_{i,j}\}, cO, cC)$
(16)                 $z_{i,j} \leftarrow z'$ {reset to old value}
(17)                 if $S > bS$ {update best solution}
(18)                     $bS \leftarrow S; u \leftarrow i; v \leftarrow j$
(19)                 else if $S < 0$ {infeasible setting}
(20)                     $F_{i,j} \leftarrow$ true {froze $z_{i,j}$}
(21)         else {if not initialized}
(22)             $\forall_{1 \le j \le m-1}, z_{i,j} \leftarrow d_{i,j}$ {increment all}
(23)             $S \leftarrow \text{Eval}(z, \{z_{i,j}\}, cO, cC)$
(24)             $\forall_{1 \le j \le m-1}, z_{i,j} \leftarrow 0$ {reset all}
(25)             if $S > bS$ {update best solution}
(26)                 $bS \leftarrow S; u \leftarrow i$
(27)     if $u = -1$ {no feasible configurations found}
(28)         break{further increment not possible}
(29)     if $I_u =$ false {if not initialized}
(30)         $I_u \leftarrow true$ {update initialization indicator}
(31)         $\forall_{1 \le j \le m-1}, z_{u,j} \leftarrow d_{i,j}$ {increment all}
(32)     else $z_{u,v} = z_{u,v} + d_{i,j}$ {increment}
(33)     $cC = C(\{z_{i,j}\})$ {update current cost}
(34)     $cO = O(\{z_{i,j}\})$ {update current output}
(35) return $\{z_{i,j}\}$ {Final result}
Eval($z, \{z_{i,j}\}, cO, cC$)
(1)  $S \leftarrow -1$ {metric score of the solution}
(2)  if $C(\{z_{i,j}\}) > r \cdot C(\mathbf{1})$ {if not feasible}
(3)      return $S$ {return negative metric score}
(4)  switch($heuristic\_type$)
(5)      case $BestOutput$:
(6)          $S \leftarrow O(\{z_{i,j}\})$; break
(7)      case $BestOutputPerCost$:
(8)          $S \leftarrow \frac{O(\{z_{i,j}\})}{C(\{z_{i,j}\})}$; break
(9)      case $BestDeltaOutputPerDeltaCost$:
(10)         $S \leftarrow \frac{O(\{z_{i,j}\})-cO}{C(\{z_{i,j}\})-cC}$; break
(11) return $S$ {return the metric score}
```

Figure 85: Greedy heuristic for setting the harvest fractions

6.5.1.2 Evaluation Metrics

The evaluation metric used for picking the best setting among the candidate settings significantly impacts the optimality of the heuristic. We introduce three alternative evaluation metrics and experimentally compare their optimality in the next section. These evaluation metrics are:

- *Best Output*: The best output metric picks the candidate setting that results in the highest join output $O(\{z_{i,j}\})$.

- *Best Output Per Cost*: The best output per cost metric picks the candidate setting that results in the highest join output to join cost ratio $O(\{z_{i,j}\})/C(\{z_{i,j}\})$.

- *Best Delta Output Per Delta Cost*: Let $\{z^{old}_{i,j}\}$ denote the setting of the harvest fractions from the last step. Then the best delta output per delta cost metric picks the setting that results in the highest additional output to additional cost ratio $\frac{O(\{z_{i,j}\})-O(\{z^{old}_{i,j}\})}{C(\{z_{i,j}\})-C(\{z^{old}_{i,j}\})}$.

Figure 85 gives the pseudo code for the heuristic setting of the harvest fractions. In the pseudo code the candidate sets are not explicitly maintained. Instead, they are iterated over on-the-fly and the candidate setting that results in the best evaluation metric is used as the new setting of the harvest fractions.

6.5.1.3 Illustration of the Greedy Heuristic

Figure 86 depicts an example illustrating the inner workings of the greedy heuristic for a 3-way join. The example starts with a setting in which $z_{i,j} = 0.2, \forall i, j$ and shows the following greedy steps of the heuristic. The harvest fraction settings are shown as 3-by-2 matrices in the figure. Similarly, 3-by-2 matrices are used (on the right side of the figure) to show the frozen harvest fractions. Initially none of the harvest fractions are frozen. In the first step a candidate set with six settings is created. In each setting one of the six harvest fractions is incremented by 0.1. As shown in the figure, out of these six settings the last two are found to be infeasible, and are marked with a cross. These two settings are the ones in which $z_{3,1}$ and $z_{3,2}$ were incremented, and thus these two harvest fractions are frozen at their last values. Among the remaining four settings, the one in which $z_{2,1}$ is increased is found to give the highest evaluation metric score. This setting is marked with

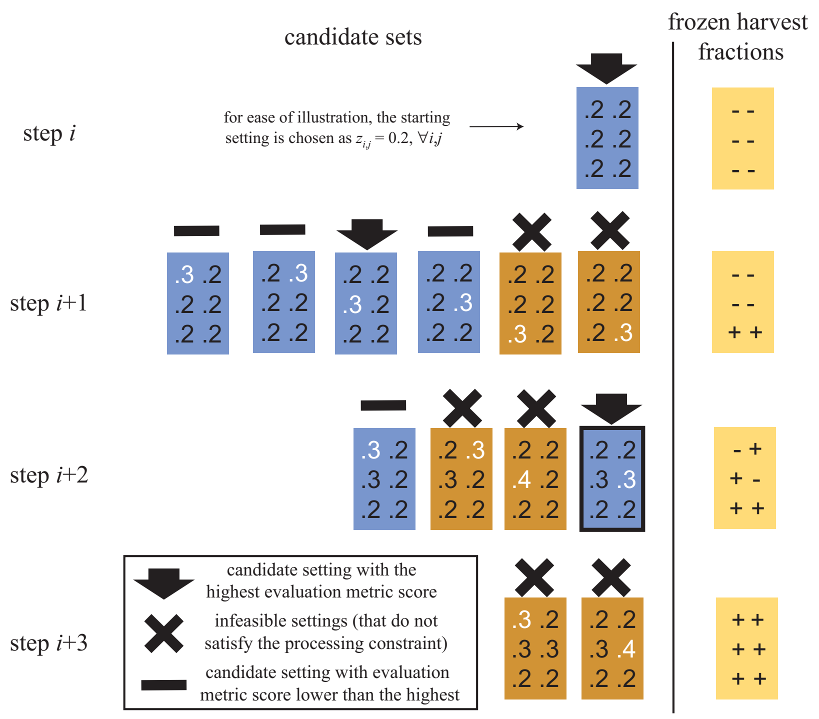

candidate sets
frozen harvest fractions
step i
for ease of illustration, the starting setting is chosen as $z_{i,j} = 0.2, \forall i,j$
step i+1
step i+2
step i+3
candidate setting with the highest evaluation metric score
infeasible settings (that do not satisfy the processing constraint)
candidate setting with evaluation metric score lower than the highest

Figure 86: Illustration of the greedy heuristic

an arrow in the figure, and forms the base setting for the next greedy step. The remaining three settings, marked with a line in the figure, are simply discarded. In the second step only four new settings are created, since two of the harvest fractions were frozen. As shown in the figure, among these four new settings two are found to be infeasible and thus two more harvest fractions are frozen. The setting marked with the arrow is found to have the best evaluation metric score and forms the basis setting for the next step. However, both of the two settings created for the next step are found to be infeasible and thus the last setting from the second step is determined as the final setting. It is marked with a frame in the figure.

6.5.2 Learning Time Correlations

The time correlations among the streams can be learned by monitoring the output of the join operator. Recall that the time correlations are captured by the pdfs $f_{i,j}$, where $i, j \in [1..m]$. $f_{i,j}$ is defined as the pdf of the difference $T(t^{(i)}) - T(t^{(j)})$ in the timestamps of the tuples $t^{(i)} \in S_i$ and $t^{(j)} \in S_j$ encompassed in an output tuple of the join. We can approximate $f_{i,j}$

by building a histogram on the difference $T(t^{(i)}) - T(t^{(j)})$ by analyzing the output tuples produced by the join algorithm.

This straightforward method of approximating the time correlations has two important shortcomings. First and foremost, since window harvesting uses only certain portions of the join windows, changing time correlations cannot be captured. Second, for each output tuple of the join we have to update $\mathcal{O}(m^2)$ number of histograms to approximate all pdfs, which hinders the performance. We tackle the first problem by using *window shredding*, and the second one through the use of sampling and *per stream histograms*. We now describe these two techniques.

6.5.2.1 Window Shredding

For a randomly sampled subset of incoming tuples, we do not perform the join using window harvesting, but instead we use *window shredding*. We denote our *sampling parameter* by ω. On the average, for only ω fraction of the tuples we perform window shredding. ω is usually small (< 0.1). Window shredding is performed by executing the join fully, except that the first window in the join order of a join direction is processed only partially based on the throttle fraction z. The tuples to be used from such windows are selected so that they are roughly evenly distributed within the window's time range. This way, we get rid of the bias introduced in the output due to window harvesting, and can safely use the output generated from window shredding for building histograms to capture the time correlations. Moreover, since window shredding only processes z fraction of the first windows in the join orders, it respects the processing constraint of the optimal window harvesting problem.

6.5.2.2 Per Stream Histograms

Although the histograms used for approximating the time correlation pdfs are updated only for the output tuples generated from window shredding, the need for maintaining $m \cdot (m-1)$ histograms is still excessive and unnecessary. We propose to maintain only m histograms, one for each stream. The histogram associated with W_i is denoted by $\mathcal{L}_i$ and it is an approximation to the pdf $f_{i,1}$, i.e., the probability distribution for the random variable $A_{i,1}$ (introduced in Section 6.4.2.1).

Maintaining only m histograms that are updated only for the output tuples generated from window shredding introduces very little overhead, but necessitates developing a new method to calculate logical basic window scores ($p^k_{i,j}$'s) from these m histograms. Recall that we had $p^k_{i,j} = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. Since we do not maintain histograms for all pdfs ($f_{i,j}$'s), this formulation should be updated. We now describe the new method we use for calculating logical basic window scores.

We start with introducing some notations. We will assume that the histograms are equi-width histograms, although extension to other types are possible. $\mathcal{L}_i$ has a valid time range of $[-w_i, w_1]$, which is the input domain of $f_{i,1}$. Let $\mathcal{L}_i(I)$ denote the frequency for the time range I, and $\mathcal{L}_i[k]$ denote the frequency for the kth bucket in $\mathcal{L}_i$. Let $\mathcal{L}_i[k^*]$ and $\mathcal{L}_i[k_*]$ denote the higher and lower points of the kth bucket's time range, respectively. Finally, let $|\mathcal{L}_i|$ denote the number of buckets in $\mathcal{L}_i$.

From the definition of $p^k_{i,j}$, we have:

$$p^k_{i,j} = P\{A_{i,l} \in b \cdot [k-1, k]\}, \text{ where } r_{i,j} = l$$

For the case of $i = 1$, nothing that $A_{i,j} = -A_{j,i}$, we have:

$$\begin{aligned} p^k_{1,j} &= P\{A_{l,1} \in b \cdot [-k, -k+1]\} \\ &= \int_{x=-b\cdot k}^{-b\cdot(k-1)} f_{l,1}(x)\, dx \end{aligned}$$

We can approximate this using $\mathcal{L}_l$, as follows:

$$p^k_{1,j} \approx \mathcal{L}_l(b \cdot [-k, -k+1]) \tag{3}$$

For the case of $i \neq 1$, we will use the trick $A_{i,l} = A_{i,1} - A_{l,1}$:

$$\begin{aligned} p^k_{i,j} &= P\{(A_{i,1} - A_{l,1}) \in b \cdot [k-1, k]\} \\ &= P\{A_{i,1} \in b \cdot [k-1, k] + A_{l,1}\} \end{aligned}$$

Making the simplifying assumption that $A_{l,1}$ and $A_{i,1}$ are independent, we get:

$$\begin{aligned} p^k_{i,j} &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot P\{A_{i,1} \in b \cdot [k-1, k] + x\}\, dx \\ &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot \int_{y=b\cdot(k-1)+x}^{b\cdot k+x} f_{i,1}(y)\, dy\, dx \end{aligned}$$

We can approximate this using $\mathcal{L}_l$ and $\mathcal{L}_i$, as follows:

$$p^k_{i,j} \approx \sum_{v=1}^{|\mathcal{L}_l|} \left(\mathcal{L}_l[v] \cdot \mathcal{L}_i(b \cdot [k-1,k] + \frac{\mathcal{L}_l[v^*] + \mathcal{L}_l[v_*]}{2}) \right) \tag{4}$$

Equations (3) and (4) are used together to calculate the logical basic window scores by only using the m histograms we maintain. In summary, we only need to capture the pdfs $f_{i,1}, \forall i \in [1..m]$ to calculate $p^k_{i,j}$ values. This is achieved by maintaining $\mathcal{L}_i$ for approximating $f_{i,1}$. $\mathcal{L}_i$'s are updated only for output tuples generated from window shredding. Moreover, window shredding is performed only for a sampled subset of input tuples defined by the sampling parameter ω. The logical basic window scores are calculated from $\mathcal{L}_i$'s during the adaptation step (every Δ seconds). This whole process results in very little overhead during majority of the time frame of the join execution. Most of the computations are performed during the adaptation step.

6.5.3 Join Orders and Selectivities

The GrubJoin algorithm uses the MJoin [128] approach for setting the join orders $R_i, \forall i \in [1..m]$. This setting is based on the low selectivity first heuristic. Concretely, let U_i be the sorted set $\{\sigma_{i,j} : 1 \leq j \neq i \leq m\}$, in ascending order. Then we set $r_{i,j}$ to k, where $\sigma_{i,k}$ is the jth item in the set U_i. This technique assumes that all possible join orderings are possible, as it is in a star shaped join graph. In practice, the possible join orders should be pruned based on the join graph and then the heuristic should be applied.

Although the low selectivity first heuristic has been shown to be effective, there is no guarantee of optimality. In this work, we choose to exclude join order selection from our optimal window harvesting configuration problem, and treat it as an independent issue. We require that the join orders are set before the window harvesting parameters are to be determined. This helps cutting down the search space of the problem significantly. Using a well established heuristic for order selection and solving the window harvesting configuration problem separately is an effective technique that makes it possible to execute adaptation step much faster. This enables more frequent adaptation.

6.6 *Experimental Results*

The GrubJoin algorithm has been implemented within our operator throttling based load shedding framework and has been successfully demonstrated as part of a large-scale stream processing prototype at IBM T.J. Watson. Here, we report two sets of experimental results to demonstrate the effectiveness of our approach. The first set of experiments evaluate the optimality and the runtime performance of the proposed heuristic algorithms used to set the harvest fractions. The second set of experiments use synthetically generated streams to demonstrate the superiority of window harvesting to tuple dropping, and to show the scalability of our approach with respect to various parameters, such as the number of join streams, the incoming stream rates, and the basic window size. All experiments presented in this chapter are performed on an IBM PC with 512MB main memory and 2.4Ghz Intel Pentium4 processor, using Java with Sun JDK 1.5.

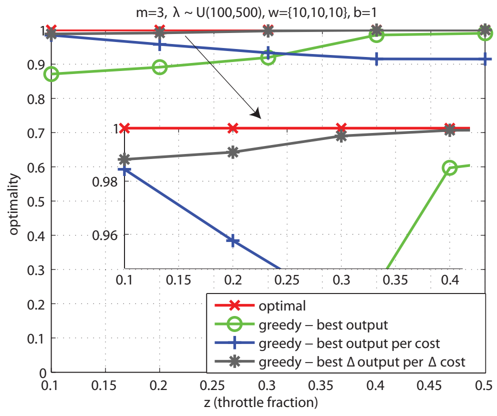


Figure 87: Effect of different evaluation metrics on optimality of greedy heuristic

6.6.1 Setting of Harvest Fractions

An important measure for judging the effectiveness of the three alternative metrics used in the candidate set evaluation phase of the greedy heuristic, is the optimality of the resulting setting of the harvest fractions with respect to the output rate of the join, compared to the best achievable obtained by setting the harvest fractions using the exhaustive search algorithm. The graphs in Figure 87 show optimality as a function of throttle fraction z for the three evaluation metrics, namely *BestOutput* (*BO*), *BestOutputPerCost* (*BOpC*), and

$BestDeltaOutputPerDeltaCost$ ($BDOpDC$). An optimality value of $\phi \in [0, 1]$ means that the setting of the harvest fractions obtained from the heuristic yields a join output rate of ϕ times the best achievable, i.e., $O(\{z_{i,j}\}) = \phi \cdot O(\{z^*_{i,j}\})$ where $\{z^*_{i,j}\}$ is the optimal setting of the harvest fractions obtained from the exhaustive search algorithm and $\{z_{i,j}\}$ is the setting obtained from the heuristic. For this experiment we have $m = 3$, $w_1 = w_2 = w_3 = 10$, and $b = 1$. All results are averages of 500 runs. For each run, a random stream rate is assigned to each of the three streams using a uniform distribution with range $[100, 500]$. Similarly, selectivities are randomly assigned. We observe from Figure 87 that $BOpC$ performs well only for very small z values (< 0.2), whereas BO performs well only for large z values ($z \geq 0.4$). $BDOpDC$ is superior to other two alternatives and performs optimally for $z \geq 0.4$ and within 0.98 of the optimal elsewhere. We conclude that $BDOpDC$ provides a good approximation to the optimal setting of harvest fractions. We next study the advantage of heuristic methods in terms of running time performance, compared to the exhaustive algorithm.

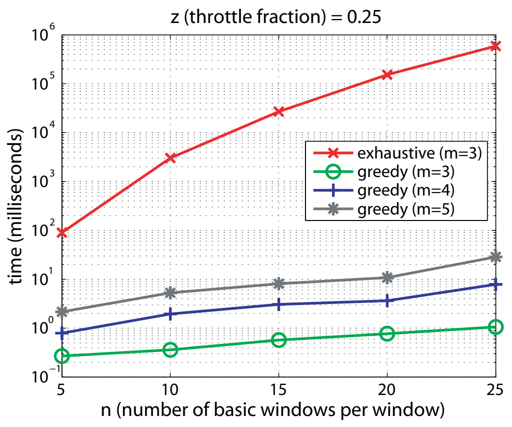


Figure 88: Running time performance w.r.t. m and number of basic windows

The graphs in Figure 88 plot the time taken to set the harvest fractions (in milliseconds) as a function of the number of logical basic windows per join window (n), for exhaustive and greedy approaches. The results are shown for 3-way, 4-way, and 5-way joins with the greedy approach and for only 3-way join with the exhaustive approach. The throttle fraction z is set to 0.25 in this experiment. Note that the y-axis is in logarithmic scale. As expected, the exhaustive approach takes several orders of magnitude more time than the greedy approach.

Moreover, the time taken for the greedy approach increases with increasing n and m, in compliance with its complexity of $\mathcal{O}(n \cdot m^4)$. However, what is important to observe here is the absolute values. For instance, for a 3-way join the exhaustive algorithm takes around 3 seconds for $n = 10$ and around 30 seconds for $n = 20$. Both of these values are simply unacceptable for performing fine grained adaptation. On the other hand, for $n \leq 20$ the greedy approach performs the setting of harvest fractions within 10 milliseconds for $m = 5$ and much faster for $m \leq 4$.

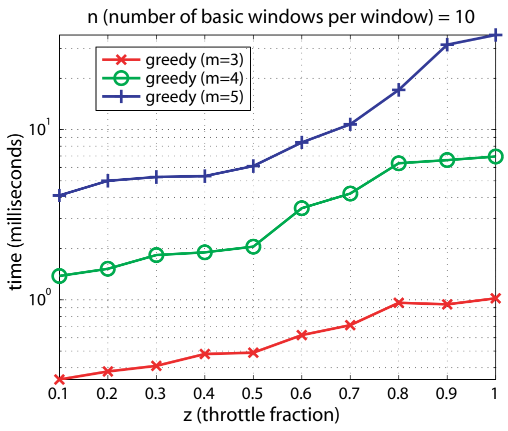


Figure 89: Running time performance with respect to m and throttle fraction

The graphs in Figure 89 plot the time taken to set the harvest fractions as a function of throttle fraction z, for greedy approach with $m = 3$, 4, and 5. Note that z affects the total number of greedy steps, thus the running time. The best case is when we have $z \approx 0$ and the search terminates after the first step. The worst case occurs when we have $z = 1$, resulting in $\approx n \cdot m \cdot (m - 1)$ steps. We can see this effect from Figure 89 by observing that the running time performance worsens as z gets closer to 1. Although the degradation in performance for large z is expected due to increased number of greedy steps, it can be avoided by reversing the working logic of the greedy heuristic.

Concretely, instead of starting from $z_{i,j} = 0, \forall i, j$ and increasing the harvest fractions gradually, we can start from $z_{i,j} = 1, \forall i, j$ and decrease the harvest fractions gradually. We call this version of the greedy algorithm *greedy reverse*. Note that greedy reverse is expected to run fast when z is large, but its performance will degrade when z is small. The solution is to switch between the two algorithms based on the value of z. We call this version of the

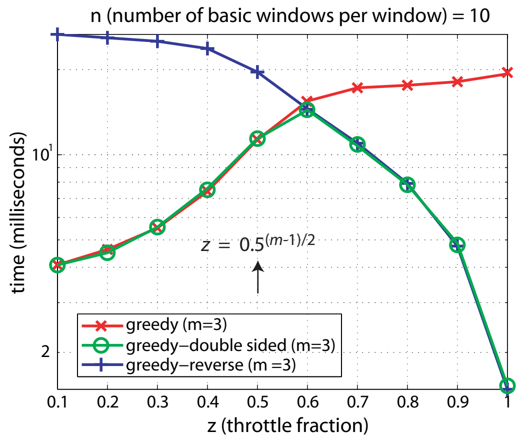


Figure 90: Running time of greedy algorithms with respect to z

algorithm *greedy double-sided*. It uses the original greedy algorithm when $z \le 0.5^{(m-1)/2}$ and greedy reverse otherwise. The graphs in Figure 90 plot the time taken to set the harvest fractions as a function of throttle fraction z, for $m = 3$ with three variations of the greedy algorithm. It is clear from the figure that greedy double-sided makes the switch from greedy to greedy reverse when z goes beyond 0.5 and gets best of the both worlds, i.e., performs good for both small and large values of the throttle fraction.

6.6.2 Results on Join Output Rate

In this section, we report results on the effectiveness of GrubJoin with respect to join output rate, under heavy system load due to high rates of the incoming input streams. We compare GrubJoin with a stream throttling based approach called *RandomDrop*. In the case of RandomDrop, excessive load is shed by placing drop operators in front of input stream buffers, where the parameters of the drop operators are set based on the input stream rates using the static optimization framework of [8]. We report results on 3-way, 4-way, and 5-way joins. When not explicitly stated, the join refers to a 3-way join. The window size is set to $w_i = 20, \forall i$ and b is set to 2, resulting in 10 logical basic windows per join window. The sampling parameter ω is set to 0.1 for all experiments. The results reported in this section are from averages of several runs. Unless stated otherwise, each run is 1 minutes, and the initial 20 seconds are used for warm-up. The default value of the adaptation period Δ is 5 seconds for the GrubJoin algorithm, although we experiment with other Δ values in some of the experiments.

The join type we employ in the experiments reported in this subsection is ϵ-join. A set of tuples are considered to be matching iff their values (assuming single-valued numerical attributes) are within ϵ distance of each other. ϵ is taken as 1 in the experiments. We model stream S_i as a stochastic process $\mathbf{X_i} = \{X_i(\varphi)\}$. $X_i(\varphi)$ is the random variable representing the value of the tuple $t \in S_i$ with timestamp $T(t) = \varphi$. A tuple simply consists of a single numerical attribute with the domain $\mathcal{D} = [0, D]$ and an associated timestamp. We define $X_i(t)$ as follows:

$$X_i(\varphi) = (D/\eta) \cdot (\varphi + \tau_i) + \kappa_i \cdot \mathcal{N}(0,1) \mod D$$

In other words, $\mathbf{X_i}$ is a linearly increasing process (with wrap-around period η) that has a random Gaussian component. There are two important parameters that make this model useful for studying GrubJoin. First, the parameter κ_i, named as *deviation parameter*, enables us to adjust the amount of time correlations among the streams. If we have $\kappa_i = 0, \forall i$, then the values for the time-aligned portions of the streams will be exactly the same, i.e., the streams are identical with possible lags between them based on the setting of τ_i's. If κ_i values are large, then the streams are mostly random, so we do not have any time correlation left. Second, the parameter τ (named as *lag parameter*) enables us to introduce lags between the streams. We can set $\tau_i = 0, \forall i$ to have aligned streams. Alternatively, we can set τ_i to any value within the range $(0, \eta]$ to create non-aligned streams. We set $D = 1000$, $\eta = 50$, and vary the time lag parameters (τ_i's) and the deviation parameters (κ_i's) to generate a rich set of scenarios. Note that GrubJoin is expected to provide additional benefits when the time correlations among the streams are strong and the streams are non-aligned.

6.6.2.1 *Varying λ, Input Rates*

The graphs in Figure 91 show the output rate of the join as a function of the input stream rates, for GrubJoin and RandomDrop. For each approach, we report results for both aligned and non-aligned scenarios. In the aligned case, we have $\tau_i = 0, \forall i$ and in the non-aligned case we have $\tau_1 = 0, \tau_2 = 5$, and $\tau_3 = 15$. The deviation parameters are set as $\kappa_1 = \kappa_2 = 2$

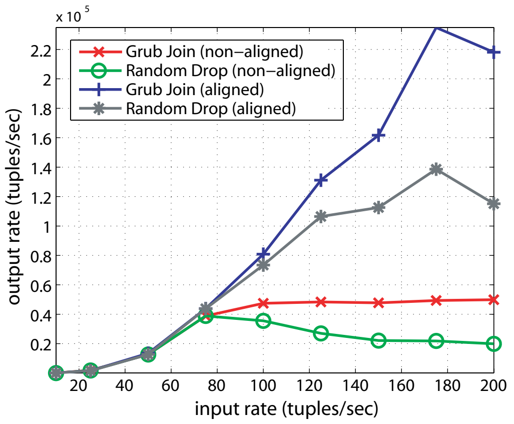


Figure 91: Effect of varying the input rates on the output rate w/wo time-lags

and $\kappa_3 = 50$. As a result, there is strong time correlation between S_1 and S_2, whereas S_3 is more random. We make three major observation from Figure 91. First, we see that GrubJoin and Random Drop perform the same for small values of the input rates, since there is no need for load shedding until the rates reach 100 tuples/seconds. Second, we see that GrubJoin is vastly superior to RandomDrop when the input stream rates are high. Moreover, the improvement in the output rate becomes more prominent for increasing input rates, i.e., when there is a greater need for load shedding. Third, GrubJoin provides up to 65% better output rate for the aligned case and up to 150% improvement for the non-aligned case. This is because the lag-awareness nature of GrubJoin gives it an additional upper hand for sustaining a high output rate when the streams are non-aligned.

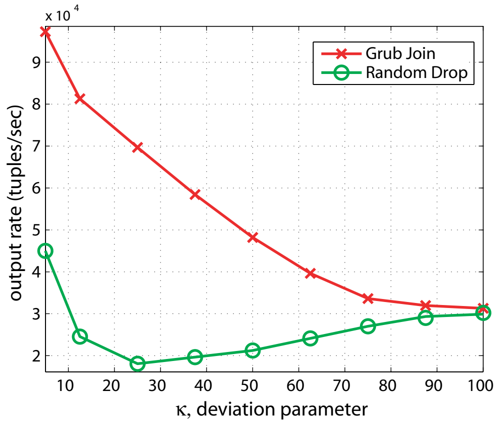


Figure 92: Effect of varying the amount of time correlations on the output rate

6.6.2.2 Varying Time Correlations

The graphs in Figure 92 study the effect of varying the amount of time correlations among the streams on the output rate of the join, with GrubJoin and RandomDrop for the non-aligned case. Recall that the deviation parameter κ is used to alter the amount of time correlations. It can be increased to remove the time correlations. In this experiment κ_3 is altered to study the change in output rate. The other settings are same with the previous experiment, except that the input rates are fixed at 200 tuples/second. We plot the output rate as a function of κ_3 in Figure 92. We observe that the join output rate for GrubJoin and Random Drop are very close when the time correlations are almost totally removed. This is observed by looking at the right end of the x-axis. However, for the majority of the deviation parameter's range, GrubJoin outperforms RandomDrop. The improvement provided by GrubJoin is 250% when $\kappa_3 = 25$, 150% when $\kappa_3 = 50$, and 25% when $\kappa_3 = 75$. It is also worth describing the behavior of RandomDrop in this experiment. Note that as κ gets larger, RandomDrop start to suffer less from its inability to exploit time correlations by using only the usefull segments of the join windows for processing. On the other hand, when κ gest smaller, the selectivity of the join increases as a side effect and in general the output rate increases. These two contrasting factors result in a bimodal graph for RandomDrop.

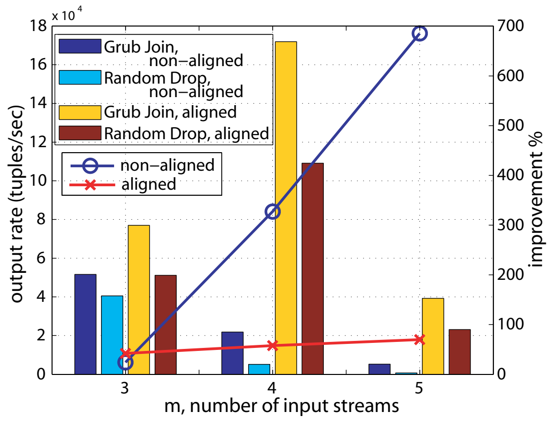


Figure 93: Effect of the # of input streams on the improvement provided by GrubJoin

6.6.2.3 Varying m, # of Input Streams

We study the effect of m (number of input streams) on the improvement provided by GrubJoin, in Figure 93. The m values are listed on the x-axis, whereas the corresponding output rates are shown in bars using the left y-axis. The improvement in the output rate (in terms of percentage) is shown using the right y-axis. Results are shown for both aligned and non-aligned scenarios. The input rates are set to 100 tuples/second for this experiment. We observe that, compared to RandomDrop, GrubJoin provides an improvement in output rate that is linearly increasing with the number of input streams. Moreover, this improvement is more prominent for non-aligned scenarios and reaches up to 700% when we have a 5-way join. This shows the importance of performing intelligent load shedding for multi-way, windowed stream joins. Naturally, joins with more input streams are costlier to evaluate. For such joins, effective load shedding techniques play an even more crucial role in keeping the output rate high.

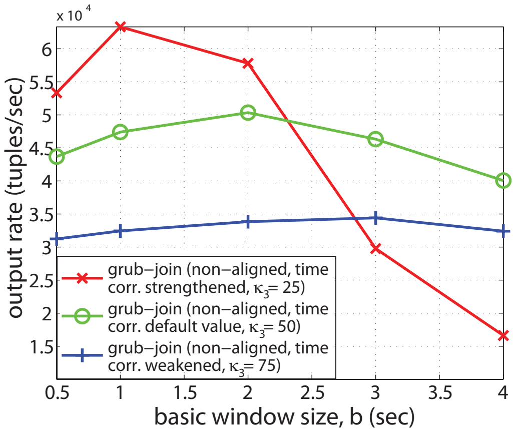


Figure 94: Effect of basic window size on output rate

6.6.2.4 Impact of Basic Window Size

As we have mentioned earlier, small basic windows are preferable when the time correlations are strong, in which cases it is advantageous to precisely locate the profitable sections of the join windows for processing. However, small basic windows increase the total number of basic windows within a join window and thus make configuration of window harvesting costly. In order to study this effect, in Figure 94 we plot the output rate of the join as a function of the basic window size for different levels of time correlations among the streams for a 3-way join. We can see from the figure that decreasing the basic window size improves the output rate only to a certain extent and further decreasing the basic window size hurts the performance. The interesting observation here is that, the basic window value for which the best output rate is achieved varies based on the strength of the time correlations, and this optimal value increases with decreasing time correlations. This is intuitive, since with decreasing time correlations there is not much gain from small basic windows and the overhead starts to dominate. The good news is that the impact of basic window size on the output rate of the join is diminishing when the time correlations are weakening (see the line for $\kappa_3 = 75$, which is flatter than others). As a result, it is still preferable to pick small basic window sizes. However, since the cost of setting the harvest fractions is dependent on the number of basic windows, rather than their size, it is advisable not to exceed 20 basic windows per join window based on our results in Section 6.6.1. The default value of 2 seconds we have used for most of the experiments in this section is a conservative setting resulting in 10 basic windows.

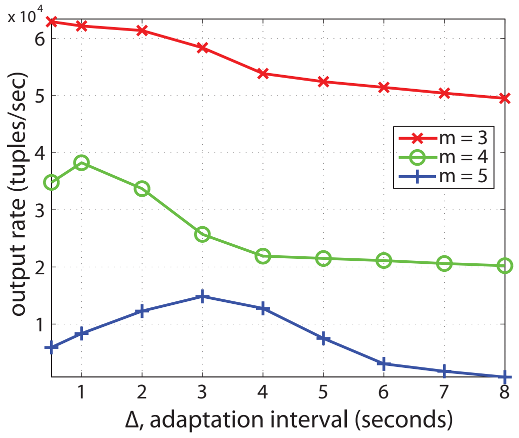


Figure 95: Effect of adaptation period on output rate

6.6.2.5 Overhead of Adaptation

In order to adapt to the changes in the input stream rates, the GrubJoin algorithm re-adjusts the window rankings and harvest fractions every Δ seconds. We now experiment with a scenario where input stream rates change as a function of time. We study the effect of using different Δ values on the output rate of the join. Recall that the default value for Δ was 5 seconds. In this scenario the stream rates start from 100 tuples/second, change to 150tuples/second after 8 seconds, and change to 50tuples/second after another 8 seconds. The graphs in Figure 95 plot the output rate of GrubJoin as a function of Δ, for different m values. Remember that larger values of m increases the running time of the heuristic used for setting the harvest fractions, thus is expected to have a profound effect on how frequent we can perform the adaptation. The Δ range used in this experiment is $[0.5, 8]$.

We observe from Figure 95 that the best output rate is achieved with the smallest Δ value 0.5 for $m = 3$. This is because for $m = 3$, adaptation step is very cheap in terms of computational cost. We see that the best output rate is achieved for $\Delta = 1$ for $m = 4$ and for $\Delta = 3$ for $m = 5$. The $\mathcal{O}(n \cdot m^4)$ complexity of the adaptation step is a major factor for this change in the ideal setting of the adaptation period for larger m. In general, a default value of $\Delta = 5$ seems to be too conservative for stream rates that show frequent fluctuations. In order to get better performance, the adaptation period can be shortened. The exact value of Δ to use depends on the number of input streams, m.

Table 10: Impact of the join condition cost on the performance

cost multiplier	1	2	4	6	8	10
improvement	6%	10%	15%	22%	27%	35%

6.6.2.6 Cost of Join Conditions

One of the motivating scenarios for CPU load shedding is the costly join conditions. We expect that the need for load shedding will become more salient with the increasing cost of the join conditions and thus GrubJoin will result in more profound improvement over tuple dropping schemes. To study the effect of join condition cost on the relative performance of GrubJoin over RandomDrop, we took the highest input stream rate at which the GrubJoin and RandomDrop perform around the same for the non-aligned scenario depicted in Figure 91 and used this rate (which is 75 tuples/sec) with different join condition costs to find out the relative improvement in output rate. We achieve different join condition costs by using a *cost multiplier*. A value of x for the cost multiplier means that the join condition is evaluated x times for each comparison made during join processing to emulate the costlier join condition. The results are presented in Table 10.

As expected, the relative improvement provided by GrubJoin increases with increasing cost multiplier. An increase of 35% is observed for a cost multiplier of 10. It is interesting to note that the increase in stream rates, as it can be observed from Figure 91, has a more pronounced impact compared to the cost of the join condition. This can be attributed to the fact that increasing stream rates not only increases the number of tuples to be processed per time unit, but it also increases the number of tuples stored within time based join windows, further increasing the cost of join processing and the strain on the CPU resources.

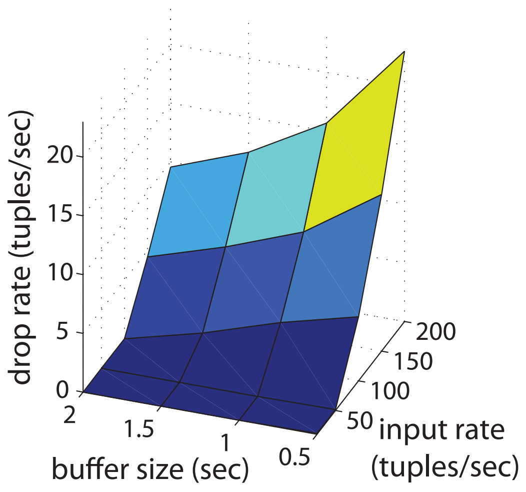


Figure 96: Tuple dropping behavior of operator throttling

6.6.2.7 Tuple Dropping Behavior

In Section 6.3.2, we have mentioned that the operator throttling framework can lead to dropping tuples during times of transition, when the throttle fraction is not yet set to its ideal value. This is especially true when the input buffers are small. In the experiments reported in this section we have used very small buffers with size 10 tuples. However, as stated before, the tuple drops can be avoided by increasing the buffer size, at the cost of introducing delay.

The graph in Figure 96 plots the average tuple drop rates of input buffers as a function of buffer size and input stream rates. The throttle fraction z is set to 1, 20 seconds before the average drop rates are measured. The adaptation interval is set to its default value, i.e., $\Delta = 5$. As seen from the figure, 1 second buffers can cut the drop rate around 30% and 2 seconds buffers around 50% for input stream rates of around 200 tuples/second. However, in the experiments reported in this chapter we chose not to use such large buffers, as they will introduce delays in the output tuples.

6.7 Discussions

Memory Load Shedding: This chapter focuses on CPU load shedding for multi-way, windowed stream joins. However, memory is also an important resource that may become a limiting factor when the join windows can not hold all the unexpired tuples due to limited memory capacity. The only way to handle limited memory scenarios is to develop *tuple admission* policies for join windows. Tuple admission policies decide which tuples should be inserted into join windows and which tuples should be removed from the join windows when there is no more space left to accommodate a newly admitted tuple. A straightforward memory conserving tuple admission policy for GrubJoin is to allow every tuple into join windows and to remove tuples from the logical basic windows that are not selected for processing such that there are no selected logical basic windows with larger indicies within the same join windows. More formally, the tuples within the logical basic windows listed in the following list can be dropped:

$$\Big\{B_{i,j} : \neg\exists u, v, k \text{ s.t. } \Big(r_{u,v} = i \wedge k \in [1..z_{u,v} \cdot n_i] \wedge s^k_{u,v} \geq j\Big)\Big\}$$

Indexed Join Processing: We have so far assumed that the join is performed in a NLJ fashion. However, special types of joins can be accelerated by appropriate index structures. For instance, ϵ-joins can be accelerated through sorted trees and equi-joins can be accelerated through hash tables. As long as the cost of finding matching tuples within a join window is proportional (not necessarily linearly) to the fraction of the window used, our solution can be extended to work with indexed joins by pluging in the appropriate cost model. Note that these indexes are to be built on top of basic windows. Since tuple insertion and deletion costs are significant for indexed joins, it is more advantageous to maintain indexes on individual basic windows, which are much smaller in size compared with the entire join window. However, there is one case where the benefit of load shedding may be less compelling: equi-join. In an equi-join, the time taken to find matching tuples within a join window is constant with hashtables and is independent of the window size. Most of the execution time is spent on generating output tuples. As a result, the design space for intelligent CPU load shedding techniques is not as large.

6.8 Related Work

The related work in the literature on load shedding in stream join operators can be classified along four major dimensions. The first dimension is the metric to be optimized when shedding load. Our work aims at maximizing the output rate of the join, also known as the MAX-subset metric [38]. Although output rate has been the predominantly used metric for join load shedding optimization [8, 38, 115, 136], other metrics have also been introduced in the literature, such as the Archive-metric proposed in [38], and the sampled output rate metric introduced in [115].

The second dimension is the constrained resource that necessitates load shedding. CPU and memory are the two major limiting resources in join processing. Thus, in the context of stream joins, works on memory load shedding [115, 38, 136] and CPU load shedding [8] have received significant interest. In the case of user-defined join windows, the memory is expected to be less of an issue. Our experience shows that for multi-way joins, CPU becomes a limiting factor before the memory does. As a result, our work focuses on CPU load shedding. However, as discussed in Section 6.7, our framework can also be used to save memory.

The third dimension is the stream characteristic that is exploited for optimizing the load shedding process. Stream rates, window sizes, and selectivities among the streams are the commonly used characteristics that are used for load shedding optimization [8, 68]. However, these works do not incorporate tuple semantics into the decision process. In *semantic* load shedding, the load shedding decisions are influenced by the values of the tuples. In frequency-based semantic load shedding, tuples whose values frequently appear in the join windows are considered as more important [38, 136]. However, this approach only works for equi-joins. In time correlation-based semantic load shedding, also called age-based load shedding [115], a tuple's profitability in terms of producing join output depends on the difference between its timestamp and the timestamp of the tuple it is matched against [115]. Our work takes this latter approach.

The fourth dimension is the fundamental technique that is employed for shedding load. In the limited memory scenarios the problem is a caching one [8] and thus tuple admission/replacement is the most commonly used technique for shedding memory load [115, 38, 136]. On the other hand, CPU load shedding can be achieved by dropping tuples from the input streams (i.e., stream throttling) [8]. As we show in this chapter, our window harvesting technique is superior to tuple dropping and prefers to perform the join partially, as dictated by our operator throttling framework.

To the best of our knowledge, this is the first work to address the adaptive CPU load shedding problem for multi-way stream joins. The most relevant work in the literature is the tuple-dropping-based optimization framework of [8], which supports multiple streams but is not adaptive. The age-based load shedding framework of [115] is also relevant, as our work and [115] share the time correlation assumption. However, the memory load shedding techniques used in [115] are not applicable to the CPU load shedding problem, and moreover they are designed for two-way joins. Finally, [28] deals with the CPU load shedding problem in the context of stream joins, however the focus is on the special case in which one of the relations resides on the disk and the other one is streamed in.

CHAPTER VII

CONCLUSION AND FUTURE WORK

With the ever increasing rate of digital information available from on-line sources, fostered by the proliferation of ubiquitous computing devices and pervasive networks, information monitoring has become an important application for end users, enabling them to access relevant changes in their interested information sources in a timely manner. Continuous query services are traditionally used to support information monitoring applications. However, due to the large and growing number of users, queries, and information sources, as well as the high rate of updates in the dynamic information sources, scalability becomes a key challenge in building CQ services to support information monitoring applications. Furthermore, the unique challenges posed and opportunities provided by ubiquitous computing platforms call for new techniques for scaling CQ services, different than the ones used in traditional client/server-based systems with centralized control.

In this thesis we have developed system-level architectures and techniques to support continuous query services for future computing platforms and applications, focusing on information monitoring applications in mobile, peer-to-peer, and sensor network computing domains. More specifically, we have focused on P2P web monitoring in Internet systems, location monitoring in mobile systems, and environmental monitoring in sensor networks and systems. In general, we have showed that ubiquitous computing platforms and pervasive networks, that exhibit highly decentralized and distributed control, involve resource-constrained devices and low-bandwidth networks with high error rates, require new techniques and solutions for addressing scalability problems in building CQ services. However, a common design philosophy employed in this thesis successfully applies to scaling CQ services in future computing platforms and applications. Concretely, moving parts of the query processing load involved in providing CQ services, close to the information sources where the data is produced, often results in putting less stress on network resources and results in less reliance on powerful centralized computing resources, which together enable scaling to large number of users, queries, and information sources, as well as to high update rates, and more frequent query evaluation with little delay in results.

Concretely, this thesis includes the following five major developments, targeted toward scaling P2P, mobile, and sensor-based CQ services in future computing platforms.

P2P Information Monitoring in Internet Systems using CQs

We have developed PeerCQ, a fully decentralized peer-to-peer architecture for Internet-scale distributed information monitoring applications. The main contribution of the PeerCQ system is the smart service partitioning scheme it employs at the P2P protocol layer, with the objective of achieving good load balance and good system utilization. This scheme has three unique properties: First, it introduces a donation based peer-awareness mechanism for handling the peer heterogeneity. Second, it introduces CQ-awareness mechanism for optimizing hot spot CQs. Third, it integrates CQ-awareness and peer-awareness through two phase matching algorithms into the load balancing scheme, while maintaining decentralization and self-configurability. In addition, we have developed a dynamic passive replication scheme to provide increased CQ durability and uninterrupted CQ processing in PeerCQ. We have reported a set of simulation-based experiments, demonstrating the effectiveness of the PeerCQ approach in supporting decentralized information monitoring on the Internet.

Distributed Location Monitoring in Mobile Systems using CQs

We have developed a distributed and scalable solution to processing moving location queries on mobile objects and described the design of MobiEyes, a distributed real-time location monitoring system in a mobile environment. The MobiEyes system makes three main contributions. First, it introduces the concept of moving queries on mobile objects to distinguish moving queries on mobile objects from a well-studied class of static spatial queries on mobile objects. Second, it involves the design of a distributed algorithm for real-time evaluation of continuously moving queries on mobile objects, which utilizes the computational power at mobile objects, leading to significant savings in terms of server load and messaging cost, when compared to solutions relying on central processing of location information at the server. Third, it employs several optimization techniques like lazy query propagation, query grouping and safe periods, to reduce the local processing load on the mobile object side and decrease the messaging cost of the system. We have demonstrated the effectiveness of MobiEyes approach through a set of simulation based experiments.

Centralized Location Monitoring in Mobile Systems using CQs

We have developed MAI, a server-side motion-adaptive indexing scheme for efficient processing of moving queries over moving objects. Our approach has three unique features. First, it uses the concept of motion-sensitive bounding boxes (MSBs) to model the dynamic motion behavior of both moving objects and moving queries, and promotes indexing less frequently changing MSBs together with the motion functions of the objects, instead of indexing frequently changing object positions. This significantly decreases the number of update operations performed on the indexes. Second, it proposes to use motion adaptive indexing in the sense that the sizes of the MSBs can be dynamically adapted to the moving object behavior at the granularity of individual objects. Finally, it advocates the use of predictive query results to reduce the number of search operations to be performed on the spatial indexes. We have reported a series of experimental performance results to demonstrate the effectiveness of our motion adaptive indexing scheme, through comparisons with other alternative indexing mechanisms.

Energy-efficient Data Collection for Sensor CQ Systems

We have developed selective sampling for energy-efficient periodic data collection in sensor networks. In particular, we have showed that selective sampling can be effectively used to increase the network lifetime, while still keeping the quality of the collected data high. We have described three main mechanisms, that together form the crux of our selective sampling approach. First, sensing-driven cluster construction is used to create clusters within the network such that nodes with close sensor readings are assigned to the same clusters. Second, correlation-based sampler selection and model derivation is used to determine the sampler nodes and to calculate the parameters of MVN models that capture the spatial and temporal correlations among sensor readings within subclusters. Last, selective data collection and model-based prediction is used to minimize the number of messages used to collect data from the network, where values of non-sampler nodes are predicted at the base node using the MVN models. We have demonstrated the effectiveness of selective sampling under different system settings through results derived from analytical and simulation based experimental studies.

Resource-aware Join Evaluation for Sensor CQ Systems

We have developed GrubJoin, an adaptive, multi-way, windowed stream join which performs time correlation-aware CPU load shedding. We have introduced the concept of window harvesting as an in-operator load shedding technique for GrubJoin. Window harvesting keeps stream tuples within the join windows until they expire and shed excessive CPU load by processing only the most profitable segments of the join windows while ignoring the less valuable ones. Window harvesting learns and exploits time correlations among the streams to maximize the output rate of the join. We have developed several heuristic and approximation-based techniques to make window harvesting effective in practice for GrubJoin, which has built-in load shedding capability based on window harvesting that is integrated with an operator throttling framework. We have conducted several experimental studies to show that GrubJoin is vastly superior to tuple dropping when time correlations exist among the input streams, and is equally effective in the absence of such correlations.

7.1 Open Issues and Future Research Directions

While this thesis presented a set of techniques and system level-architectures to scale CQ services with the aim of supporting information monitoring applications, it has also draw attention to a number of open issues and left room for future work toward addressing these. In this section, we start with discussing open issues in the context of this thesis and then present an outlook of future directions.

7.1.1 Open Issues in the Context of this Thesis

We first discuss open issues for each of the three application domains and platforms considered in this thesis, and then turn our attention to hybrid platforms and applications.

Web Monitoring in P2P Systems: The PeerCQ system we have developed for web monitoring provides effective CQ partitioning in a DHT-based P2P network, with good load balance, system utilization, and reliability properties. However, nodes can still get overloaded in PeerCQ, especially when the resource availability of the system as a whole is not sufficient to handle the CQ workload under PeerCQ's service partitioning scheme. As a result, development of a QoS-aware admission control mechanism for CQs is an important extension to PeerCQ and is an open research problem.

Security in PeerCQ constitutes another interesting research dimension. The PeerCQ system needs authentication and access control mechanisms for CQ management, as well as mechanisms to protect the system from distributed denial of service attacks (e.g. malicious users injecting large number of bogus CQs into the system). Yet another issue is the anonymity of the users that submit CQs into the system. To our knowledge these problems have not been addressed in the context of P2P CQ systems and are still open.

Location Monitoring in Mobile Systems: The MobiEyes and MAI systems we have developed for location monitoring in mobile systems are very effective in terms of evaluating mobile CQs quickly by incurring low disk IO and CPU processing as well as low wireless communication. Continuous range and kNN queries with potentially moving query regions/points have been our primary focus in this work. However, our work will significantly benefit from extensions aimed at covering a larger and richer set of continuous queries, such as continuous reverse kNN queries [18] and mobile CQs in the context of road-network-based mobile object tracking [34]. Even though exclusive techniques exist in the literature for supporting these extensions, in order to avoid the excessive cost of maintaining separate data structures for answering different types of queries, it is important to devise a unified framework that has the ability to efficiently evaluate a large set of different query types.

Environmental Monitoring in Sensor Networks: The selective sampling-based data collection scheme we have developed for sensor CQ systems provides energy-aware extraction of data streams from the sensor network and is complemented by the resource-aware query evaluation techniques we have developed for server-side processing of data streams. However, there are a number of extensions that can significantly improve the applicability of our work. These include devising sleeping schedules to save energy by reducing the amount of time the radio is kept at listen mode, providing mathematical guarantees to support bounded error in data collection, and probabilistic query processing techniques at the server-side to handle imprecision in sensor readings. Another research direction is providing reliability and error handling features for our selective data collection framework.

Hybrid Platforms: This thesis research is focused around providing techniques for scaling CQ services in three different computing platforms (P2P, mobile, and sensor networks), but there are emerging platforms that share some characteristics with all of the three base platforms considered in this thesis. A recent example is inter-vehicle networks. A modern vehicle can be connected to a traditional mobile network (such as a cell phone network), yet can have additional local communication capabilities to form P2P networks with other close by vehicles. There has been several recent research initiatives targeted toward utilizing these P2P networks for various applications, such as information exchange [138] and coordinated evacuation [54]. From a different perspective, a vehicle can also be seen as a media-rich sensor node, even though a vehicle is not power-constrained like the tiny sensor devices found in sensor networks. Vehicles can encompass a variety of different sensor devices, such as cameras, GPS, and diagnostic sensors. Data management and CQ services for inter-vehicle networks are recently being investigated by researchers [11]. We believe that information monitoring applications and CQ systems designed for inter-vehicular networks can benefit from solutions borrowing some of the techniques and ideas that we have developed for the three base application domains and platforms considered in this thesis.

7.1.2 An Outlook of Future Research Directions

The main motivation of this thesis is the existence of large number of on-line data sources with high update rates, that makes pull-based manual monitoring of such sources difficult and necessitates the development of information monitoring applications that can provide easy to use push-based services, and thus the development of CQ systems that are required to build such information monitoring applications. These information sources are increasingly taking the form of data streams. Examples of data streams include stock tickers in financial services, link statistics in networking and telecommunications, sensor network readings in environmental monitoring and emergency response, and satellite and live experimental data in scientific computing.

The increasing number of stream-based information sources entails developing event-based data stream processing middleware that can support a variety of data management applications, not limited to information monitoring. Even though these systems are expected to be more generic than CQ systems, most of the techniques developed in this thesis are applicable in the context of data stream management systems. Here, we list a number of research directions related with distributed data stream management systems.

Large-scale Stream Processing

Large-scale distributed data stream processing involves executing large number of data processing tasks, each composed of a number of interconnected stream processing elements, on a large and distributed set of processing nodes. Large-scale distributed stream processing poses many research challenges. These include providing self-optimizing behavior with respect to workload dynamics, resource availability, load balance, and network dynamics, as well as providing self-healing properties with respect to node failure and network partitioning. Large number of data processing tasks and stream sources they use, the highly dynamic rates of data streams and thus the changing runtime load incurred by data processing tasks, and the distributed and decentralized nature of the processing nodes, create many challenges in developing scalable and adaptive task partitioning, placement, and scheduling schemes in distributed data stream management systems.

Distributed Stream Delivery

In distributed data stream management systems, a data stream sourced at one node is usually of interest to a number of other nodes that are the consumers of the stream. Supply of data streams to all interested nodes constitutes an interesting research problem, that we call the distributed stream delivery problem. A consumer node interested in receiving a stream may define a filter on the source stream so that only the desired portion of the stream is received and the bandwidth consumption is minimized. However, building an optimal stream delivery configuration may not always be possible, mainly due to the insufficient network bandwidth resources, which is usually a consequence of high stream rates, large number of consumers, or low level of overlap in filter specifications. This tells us that quality of service (QoS) is a critical factor in designing effective stream delivery paths. Developing scalable and effective stream delivery paths and adapting such delivery paths in the presence of workload and network dynamics is a challenging problem.

Persistence Support in Stream Processing

Although the main focus of data stream management systems is on-line processing of streaming data, it is often required to analyze or even combine the streaming data together with the archived data, while at the same time archiving the streaming data for later use. Since it is often much slower to access archived data, several challenges exist in archiving streaming data and integrating the archived data with the on-line processing of streaming data. These include developing summary and index structures for quick processing, update, and access of archived data, as well as utility-based expiration schemes to manage storage space in the presence of unbounded streams.

APPENDIX A

EFFECTIVE DONATION (ED) CALCULATION IN PEERCQ

Here we describe the calculation of effective donation (ED) of a peer. ED is an integer variable in the range $[1, c]$. $ED = 1$ means the effective donation of the peer is minimum and $ED = c$ means it is maximum. ED represents the perceived donation of the peer by the PeerCQ system. There are some constants and variables that are used to calculate ED. We describe them first, and then give the algorithm for ED calculation:

R (resources) is a constant vector, denoted as $[r_1, ..., r_4]$, representing the resource types. Its value is [“cpu”, “hard disk”, “memory”, “network bandwidth”]. r_i, $1 \leq i \leq 4$, is the ith resource type. AR (actual resources) is a vector variable, denoted as $[ar_1, ..., ar_4]$, representing the amount of actual resources the peer machine possesses. ar_i, $1 \leq i \leq 4$, is the amount of type r_i resource possessed by the peer machine. Each element ar_i is an integer, where ar_1 is the speed of the CPU, ar_2 is the capacity of the hard disk, ar_3 is the amount of main memory, and ar_4 is the amount of network bandwidth.

RP (resource power) is a vector variable, denoted as $[rp_1, ..., rp_4]$, representing the amount of power the peer machine possesses for each resource. rp_i is the power of type r_i resource possessed by the peer machine. Each element rp_i, $1 \leq i \leq 4$, of this vector is an integer in the range [1,5]. $rp_i = 1$ means that the peer is very poor in terms of resource type r_i and $rp_i = 5$ means it is very powerful in terms of resource type r_i. RP is calculated from AR with the use of mapping functions.

MF (mapping functions) is a vector of functions, denoted as $[mf_1, ..., mf_4]$, where for $1 \leq i \leq 4$, $mf_i(ar_i) = rp_i$, meaning mf_i takes as a parameter amount of actual resources of type r_i and returns the power of that type of resource.

PD (peer donation) is a vector variable, denoted as $[pd_1, ..., pd_4]$, representing the donation of the peer. pd_i, $1 \leq i \leq 4$, is the percentage of type r_i resource the peer wants to donate to the system. Each element pd_i of this vector is a real number in the range (0,1]. PD can be defined by the administrator of the peer and also have a preset default value. A peer may dynamically change the amount of PD according to the load changes on its machine. One way to implement such load adaptation is to combine workload monitoring and multi-level workload driven PD policy such that the PD value of a peer will be revised accordingly as the workload change on the peer machine exceeds certain thresholds.

DP (donated power) is a vector variable, denoted as $[dp_1, ..., dp_4]$, representing the amount of power the peer donates. dp_i, $1 \leq i \leq 4$, is the donated power of type r_i resource. Each element dp_i of this vector is a real number in the range (0,5] and has similar interpretation to elements of RP (resource power).

RI (resource importance) is a constant vector $[ri_1, ..., ri_4]$, representing the importance of each resource regarding CQ system. The elements ri_i, $1 \leq i \leq 4$, of this vector are positive real numbers and sum up to 1.

rel (reliability) is a variable that denotes the reliability of the peer. It can be calculated at the initialization time by the equation: $rel = MIN(1,\ average\_uptime\_per\_session\ /\ expected\_uptime)$, where *average_uptime_per_session* is the average time the peer participates each time it joins to the system. It can be updated on each exit or incrementally while running. *rel* is set to 1 if it is the first time for this peer to join the system. *expected_uptime* is a time considered as 'reasonable' to participate in PeerCQ system.

Then the ED (effective donation) is calculated as described in Figure 97, given a peer p, its PD (peer donation), AR (actual resources), and *rel* (reliability).

```
CalculateED($p$, $PD$, $AR$, $rel$)
(1) $ED \leftarrow 0$
(2) {$i$ stands for the four types of resources}
(3) for $i = 1$ to $4$
(4)     $RP[i] \leftarrow MF[i](AR[i])$
(5)     $DP[i] \leftarrow PD[i] * RP[i]$
(6)     $ED \leftarrow ED + RI[i] * DP[i]$
(7) $ED \leftarrow \lceil rel * (c/5) * ED \rceil$
(8) return $ED$
```

Figure 97: Effective donation calculation

APPENDIX B

ANALYTICAL MODEL FOR MESSAGING COST ESTIMATION IN MOBIEYES

Here we give an analytical estimate of the messaging cost of MobiEyes, which can also be used to set the optimal value of the cell size parameter α of the grid G corresponding to the universe of discourse (recall Figure 32 and Figure 33 that the effect of α on server load is analogous). The cost estimation is based on the EQP approach and its extension to LQP is straightforward. Let $mcost(T)$ be the average number of messages exchanged (messaging cost) during a given time period of T seconds per one object. Let $avgspd$ be the average moving speed of an object and let $avgr$ be the average query radius. The messaging cost can be divided into two components, namely the cost due to an object changing its current grid cell (denoted as $cell\_change\_cost$) and the cost due to a focal object changing its velocity vector (denoted as vel_change_cost). Let nmq, nmo, ts be defined as shown in Table 2. Then we can estimate $mcost(T)$ as follows:

$$mcost(T) = \frac{T}{\frac{\alpha}{avgspd}} * cell\_change\_cost + \frac{T}{ts} * \frac{nmo}{no} * \frac{nmq}{no} * vel\_change\_cost$$

The expression preceding $cell\_change\_cost$ is a crude estimate of the number of times a given object changes its current grid cell during the time interval T. The expression preceding vel_change_cost is the probability that a given object is a focal object of some query times the number of times a given object changes its velocity vector during the interval T.

The vel_change_cost is composed of a message being sent from a focal object to the server plus the number of broadcasts performed to convey this velocity change to a set of objects that reside in the monitoring region of the given focal object, which can be estimated as: $vel\_change\_cost = 1 + (1 + \frac{mrslen}{alen})^2$. Here $mrslen = \alpha * (1 + 2 * \lceil avgr/\alpha \rceil)$ is the average length of the side of a monitoring region. $(1 + \frac{mrslen}{alen})^2$ is an estimate of the number of broadcast areas required to cover a monitoring region, where $alen$ is the average base station coverage area side length (listed in Table 2).

The *cell_change_cost* is composed of a message being sent from the object to the server plus the cost of sending new queries of interest to the object (*new_queries_cost*) plus the cost of relaying the monitoring region change to a set of other objects (*region_update_cost*) in case the object of interest is a focal object of some query. This can be estimated as:

$$\begin{aligned}
cell\_change\_cost &= 1 + new\_queries\_cost + \frac{nmq}{no} * region\_update\_cost \\
new\_queries\_cost &= nmq * \frac{\alpha * mrslen}{area} \\
region\_update\_cost &= \left(1 + \frac{mrslen}{alen}\right) * \left(1 + \frac{\alpha + mrslen}{alen}\right)
\end{aligned}$$

Let A and A' denote any two adjacent cells, assuming an object moves from cell A to cell A'. In the *new_queries_cost* formula, $\alpha * mrslen$ is the size of the region that covers the possible current grid cells of focal objects whose monitoring regions contain the new cell A' but not the old cell A. Multiplying this value with $nmq/area$ gives an estimate on the number of queries whose monitoring regions intersect the new cell, but not the old cell.

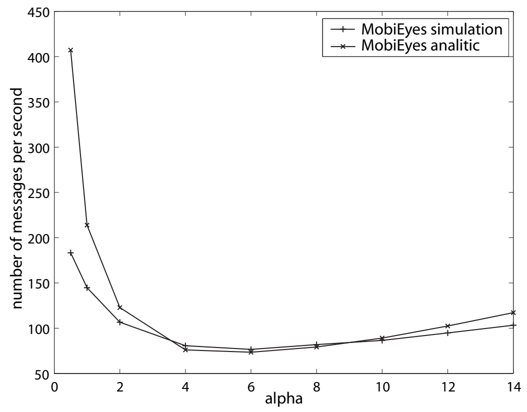


Figure 98: Comparison of analytical messaging cost with simulation results

The *region_update_cost* formula estimates the number of broadcast areas required to cover the region which is the union of old and new monitoring regions of a query. *mrslen* gives the length of one side of this region, where $\alpha + mrslen$ gives the other.

Figure 98 compares the analytical estimate on the number of messages with the results obtained from the simulation for different values of α. The y-axis represents the number of messages exchanged per time step, where x-axis represents different α values. It is clear that the estimate is quite accurate for most of the values of α. When α is small, the crude estimate of the number of times an object changes its current grid cell results in overestimating the messaging cost. This problem alleviates as α increases. The small underestimate when we have larger α values is due to the fact that our analytical estimate does not include the cost of notifications sent from an object to the server when the object is added into or removed from the result of a query.

APPENDIX C

ANALYTICAL MODEL FOR IO COST ESTIMATION IN MAI

We develop an analytical model for estimating the IO cost of performing query evaluation, i.e., the two scans performed at each query evaluation phase. The formulations in this section are derived based on the average values for the speed and the period of constant motion of a moving object. For the purpose of off-line *$\alpha\beta Table$* creation, the associated speed and period of constant motion values are taken from the table cells. Table 11 lists some of the symbols used in this section and their meanings.

Table 11: Symbols and their meanings

P_s	scan period	R_{mq}	average moving query radius
P_{cm}	avg. period of constant motion	L_{sq}	average static query side length
N_o	number of objects	V_a	average moving object speed
N_{mo}	number of moving objects	A	area of the region of interest
N_q	number of queries	α	MSB parameter for objects
N_{mq}	number of moving queries	β	MSB parameter for queries

Let A_{mo} denote the average area of a moving object MSB and A_{mq} denote the average area of a moving query MSB. Denoting the average object speed as V_a, based on the definition of MSBs we have:

$$\begin{aligned} A_{mo} &= (\alpha * V_a/\sqrt{\pi})^2, \text{ and} \\ A_{mq} &= (\beta * V_a/\sqrt{\pi} + 2 * R_{mq})^2 \end{aligned}$$

The derivation of A_{mo} follows from the fact that the side of a moving object MSB has average size of α times the average speed of the object on the side's direction. Averaging over all possible angles for the velocity vector, we have $A_{mo} = (\alpha * V_a)^2 * \frac{1}{2*\pi} \int_0^{2*\pi} |\sin x| * |\cos x| \, dx$ $= (\alpha * V_a/\sqrt{\pi})^2$. The derivation for the moving query MSBs follow a similar formulation, with the exception that the diameter of the query, denoted by $2 * R_{mq}$, is also included in the equation.

Let A_o denote the average size of the object bounding boxes stored in the $Index_o^{msb}$ (static object's are assumed to have a box with zero area) and A_q denote the average size of the query bounding boxes stored in the $Index_q^{msb}$. Then, we have:

$$\begin{aligned} A_o &= A_{mo} * \frac{N_{mo}}{N_o}, \text{ and} \\ A_q &= \frac{N_{mq}}{N_q} * A_{mq} + (1 - \frac{N_{mq}}{N_q}) * L_{sq}^2 \end{aligned}$$

The derivation of A_o follows from the fact that N_{mo}/N_o fraction of the objects (that are moving objects) have an average MSB size of A_{mo} and the rest (stationary objects) have an MSB size of 0. The derivation of A_q follows similarly. Stationary queries, that form N_{mq}/N_q fraction of all queries, have an average MSB size of L_{sq}^2, where L_{sq} is the average side length of a static range query. On the other hand, moving queries, that form N_{mq}/N_q fraction of all queries, have an average MSB size of A_{mq}.

Given this information, the following four quantities can be analytically derived based on well studied R-tree cost models [126]: node IO cost during the processing of (1) an object table entry for updating the $Index_o^{msb}$, C_o^u; (2) an object table entry for searching the $Index_q^{msb}$, C_o^s; (3) a query table entry for updating the $Index_q^{msb}$, C_q^u; (4) a query table entry for searching the $Index_o^{msb}$, C_q^s.

Let N_o^{vc} denote the expected value of the number of distinct objects causing velocity change events during one scan period and N_q^{vc} denote the expected value of the number of distinct queries causing velocity change events during one scan period. If $P_s/P_{cm} < 1$, only some of the moving objects will cause velocity change events. Hence, we have:

$$\begin{aligned} N_o^{vc} &= N_{mo} * \min(1, \frac{P_s}{P_{cm}}), \text{ and} \\ N_q^{vc} &= N_{mq} * \frac{N_o^{vc}}{N_{mo}} \end{aligned}$$

The derivation of N_q^{vc} follows from the fact that a moving query causes a velocity change event only if its focal object causes a velocity change event and that only N_o^{vc}/N_{mo} fraction of the moving objects cause velocity change events.

Let N_o^{bi} denote the expected value of the number of objects causing box invalidations during one scan period and N_q^{bi} denote the expected value of the number of queries causing box invalidations during one scan period. If $P_s/\alpha < 1$, only some of the moving objects will cause box invalidations. Similarly, if $P_s/\beta < 1$, only some of the moving queries will cause box invalidations. Then, we have:

$$\begin{aligned} N_o^{bi} &\approx \min(1, \frac{P_s}{\alpha}) * N_{mo}, \text{ and} \\ N_q^{bi} &\approx \min(1, \frac{P_s}{\beta}) * N_{mq} \end{aligned}$$

Let N_{mot} denote the expected value of the number of entries in the object table that caused velocity change or box invalidation events and N_{mqt} denote the expected value of the number of entries in the query table that caused velocity change or box invalidation events. Assuming that an object causes a velocity change event independent of whether it has caused an MSB invalidation and similarly assuming that a query causes a velocity change event independent of whether it has caused an MSB invalidation, we have:

$$\begin{aligned} N_{mot} &= N_o^{vc} + N_o^{bi} - N_o^{bi} * \frac{N_o^{vc}}{N_{mo}}, \text{ and} \\ N_{mqt} &+ N_q^{vc} + N_q^{bi} - N_q^{bi} * \frac{N_q^{bi}}{N_{mq}} \end{aligned}$$

Finally, the total IO cost for the periodic scan, C_{io}, can then be calculated, considering that for an entry of MOT that requires processing due to velocity change or MSB invalidation, an update on the $Index_o^{msb}$ and two searches on the $Index_q^{msb}$ are needed and for an entry of MQT that requires processing due to velocity change or MSB invalidation, an update on the $Index_q^{msb}$ and a search on the $Index_o^{msb}$ are needed:

$$C_{io} = N_{mot} * (C_o^u + 2 * C_o^s) + N_{mqt} * (C_q^u + C_q^s) \tag{5}$$

APPENDIX D

ANALYTICAL MODEL FOR MESSAGING COST ESTIMATION IN SELECTIVE SAMPLING

We derive analytical formulas to capture the messaging cost of our selective sampling solution. For the purpose of comparison, we introduce two variations of selective sampling. We name the way it is described so far, the *hybrid* approach. The name hybrid comes from the fact that the spatial and temporal correlations are captured and summarized locally within the network, while the prediction is performed outside the network. In the *central* approach both phases are performed at the base node. This means that, all forced samples are forwarded to the base node, so that the correlations can be captured from the data. In the *local* approach, prediction is performed on the cluster heads and predicted values are reported to the base node. Although the local approach results in a large messaging cost and is against the idea of selective sampling, it serves as a base case for comparison.

In the rest of this section, we calculate the total number of messages sent and received (spent) within the network during a time interval of T seconds, denoted by M_t^h, M_t^c, and M_t^l, respectively for hybrid, central, and local approaches. We denote the total number of clusterings and sub-clusterings (schedule derivation step) performed during the time interval of length T as N_{nc} and N_{ns}. We have $N_{nc} = \lfloor T/\tau_c \rfloor$ and $N_{ns} = \lfloor T/\tau_u \rfloor$. Similarly, the total number of forced samplings and virtual samplings are denoted by N_{fs} and N_{vs}. We have $N_{fs} = \lfloor T/\tau_f \rfloor$ and $N_{vs} = \lfloor T/\tau_v \rfloor$.

The total number of messages can be broken into three components, namely messages spent during *i*) clustering, *ii*), sub-clustering, and *iii*) data collection. The messages spent during clustering, denoted by M_{tc}, is same for all approaches and can be defined as $M_{tc} = N_{nc} * M_{cs}$, where M_{cs} denotes the number of messages spent during one clustering step. Since N_{nc} is expected to be much smaller than N_{ns}, N_{fs}, and N_{vs}, we omit the derivation of M_{cs} in the interest of space. Now we describe the derivation of the remaining two components *ii*) and *iii*), for three different scenarios. We use the notations I_i^b and I_i^c to denote the distance of node n_i (in terms of hops) to the base node and its cluster head node h_i, respectively. Each message is assumed to have the size of a basic message, where a basic message includes a node identifier and a sensor reading.

The derivation for the **Hybrid Approach** is as follows:

$$\begin{aligned}
M_{ts}^h &= N_{ns} * \sum_{i=1}^{N} I_i^c + N_{ns} * \sum_{n_i \in H} \left(I_i^b * \sum_{j=1}^{|G_i|} \left(|G_i(j)| * \frac{3 + |G_i(j)|}{4} \right) \right) \\
M_{tm}^h &= N_{fs} * \sum_{n_i \in H} \sum_{n_j \in C_i} (I_j^c + S_i[n_j] * I_i^b) + (N_{vs} - N_{fs}) * \sum_{n_i \in H} \sum_{n_j \in C_i} (S_i[n_j] * I_j^b) \\
M_t^h &= M_{tc} + M_{ts}^h + M_{tm}^h \qquad (6)
\end{aligned}$$

Here M_{ts}^h denotes the sub-clustering component for the hybrid approach. It consists of two sub-components, messages spent for notifying each node after schedules are derived, and the messages spent for reporting the covariance matrix and mean vector to the base node for each sub-cluster. Note that the covariance matrix is symmetric and thus not all the entries are reported. M_{tm}^h denotes the data collection component for the hybrid approach. In summary, it counts the messages from sampler nodes and non-sampler nodes. Messages from non-sampler nodes are forwarded up to the cluster heads after every forced sampling period. On the other hand, messages from sampler nodes are forwarded up to the base node after every virtual sampling period. Finally, the total number of messages for the hybrid approach, denoted by M_t^h, is calculated in Equation 6 as the sum of three components.

The derivation for the **Central Approach** is as follows:

$$\begin{aligned}
M_{ts}^c &= N_{ns} * \sum_{i=1}^{N} I_i^c \\
M_{tm}^c &= N_{fs} * \sum_{i=1}^{N} I_i^b + (N_{vs} - N_{fs}) * \sum_{n_i \in H} \sum_{n_j \in C_i} (S_i[n_j] * I_j^b) \\
M_t^c &= M_{tc} + M_{ts}^c + M_{tm}^c \qquad (7)
\end{aligned}$$

Here M_{ts}^c denotes the sub-clustering component for the central approach. It consists of the messages used for notifying each node after schedules are derived. As opposed to hybrid scenario, it does *not* include the reporting of covariance matrices or mean vectors. M^c_{tm} denotes the data collection component for the central approach. Different from the hybrid scenario, all forced samples are forwarded up to the base node. Finally, the total number of messages for the central approach, denoted by M^c_t, is calculated in Equation 7 as the sum of three components.

The derivation for the **Local Approach** is as follows:

$$\begin{aligned}
M^l_{ts} &= N_{ns} * \sum_{i=1}^{N} I^c_i \\
M^l_{tm} &= N_{fs} * \sum_{n_i \in H} \sum_{n_j \in C_i} (I^c_j + S_i[n_j] * I^b_i) \\
&+ (N_{vs} - N_{fs}) * \sum_{n_i \in H} \sum_{n_j \in C_i} (S_i[n_j] * I^b_j) + N_{vs} * \sum_{n_i \in H} \sum_{n_j \in C_i} (I^b_i * (1 - S_i[n_j])) \\
M^l_t &= M_{tc} + M^l_{ts} + M^l_{tm} \qquad (8)
\end{aligned}$$

Here M^l_{ts} denotes the sub-clustering component for the central approach and is identical to the sub-clustering component of the central approach. M^l_{tm} denotes the data collection component for the local approach. It can be considered as the data collection component of the hybrid approach, plus the number of messages spent for forwarding the predicted samples from the cluster head nodes to the base node. Finally, the total number of messages for the local approach, denoted by M^l_t, is calculated in Equation 8 as the sum of three components.

APPENDIX E

ANALYTIC MODEL FOR COST AND OUTPUT ESTIMATION IN GRUBJOIN

We describe the analytical formulations of the functions $C(\{z_{i,j}\})$ and $O(\{z_{i,j}\})$. For the formulation of C, we will assume that the processing cost of performing the NLJ join is proportional to the number of tuple comparisons made per time unit. We do not include the cost of tuple insertion and removal in the following derivations, although they can be added with little effort.

The total cost C is equal to the sum of the costs of individual join directions, where the cost of performing the ith direction is λ_i times the number of tuple comparisons made for processing a single tuple from S_i. We denote the latter with C_i. Thus, we have:

$$C = \sum_{i=1}^{m} (\lambda_i \cdot C_i)$$

C_i is equal to the sum of the number of tuple comparisons made for processing each window in the join order R_i. The number of tuple comparisons performed for the jth window in the join order, that is $W_{r_{i,j}}$, is equal to the number of times $W_{r_{i,j}}$ is iterated over, denoted by $N_{i,j}$, times the number of tuples used from $W_{r_{i,j}}$. The latter is calculated as $z_{i,j} \cdot S_{i,j}$, where $S_{i,j} = \lambda_{r_{i,j}} \cdot w_{r_{i,j}}$ gives the number of tuples in $W_{r_{i,j}}$. We have:

$$C_i = \sum_{j=1}^{m-1} (z_{i,j} \cdot S_{i,j} \cdot N_{i,j})$$

$N_{i,j}$, which is the number of times $W_{r_{i,j}}$ is iterated over for evaluating the ith direction of the join, is equal to the number of partial join results we get by going through only the first $j-1$ windows in the join order R_i. We have $N_{i,1} = 1$ as a base case. $N_{i,2}$, that is the number of partial join results we get by going through $W_{r_{i,1}}$, is equal to $P_{i,1} \cdot \sigma_{i,r_{i,1}} \cdot S_{i,1}$, where $\sigma_{i,r_{i,1}}$ denotes the selectivity between W_i and $W_{r_{i,1}}$, and as before $S_{i,1}$ is the number of tuples in $W_{r_{i,1}}$. Here, $P_{i,1}$ is a *yield factor* that accounts for the fact that we only use $z_{i,j}$

fraction of $W_{r_{i,j}}$. If the pdfs capturing the time correlations among the streams are flat, then we have $P_{i,j} = z_{i,j}$. We describe how $P_{i,j}$ is generalized to arbitrary time correlations shortly. By noting that for $j \geq 2$ we have $N_{i,j} = N_{i,j-1} \cdot P_{i,j-1} \cdot \sigma_{i,r_{i,j-1}} \cdot S_{i,j-1}$ as our recursion rule, we generalize our formulation as follows:

$$N_{i,j} = \prod_{k=1}^{j-1} \left(P_{i,k} \cdot \sigma_{i,r_{i,k}} \cdot S_{i,k}\right)$$

In the formulation of $P_{i,j}$, for brevity we will assume that $z_{i,j}$ is a multiple of $1/n_{r_{i,j}}$, i.e., an integral number of logical basic windows are selected from $W_{r_{i,j}}$ for processing. Then we have:

$$P_{i,j} = \sum_{k=1}^{z_{i,j} \cdot n_{r_{i,j}}} p_{i,j}^{s_{i,j}^k} / \sum_{k=1}^{n_{r_{i,j}}} p_{i,j}^k$$

To calculate $P_{i,j}$, we use a scaled version of $z_{i,j}$ which is the sum of the scores of the logical basic windows selected from $W_{r_{i,j}}$ divided by the sum of the scores from all logical basic windows in $W_{r_{i,j}}$. Note that $p_{i,j}^k$'s (logical basic window scores) are calculated from the time correlation pdfs as described earlier in Section 6.4.2.1. If $f_{i,j}$ is flat, then we have $p_{i,j}^k = 1/n_{r_{i,j}}, \forall k \in [1..n_{r_{i,j}}]$ and as a consequence $P_{i,j} = z_{i,j}$. Otherwise, we have $P_{i,j} > z_{i,j}$. This means that we are able to obtain $P_{i,j}$ fraction of the total number of matching tuples from $W_{r_{i,j}}$ by iterating over only $z_{i,j} < P_{i,j}$ fraction of $W_{r_{i,j}}$. This is a result of selecting the logical basic windows that are more valuable for producing join output. This is accomplished by utilizing the window rankings during the selection process. Recall that these rankings ($s_{i,j}^v$'s) are calculated from logical basic window scores.

We easily formulate O using $N_{i,j}$'s. Recalling that $N_{i,j}$ is equal to the number of partial join results we get by going through only the first $j-1$ windows in the join order R_i, we conclude that $N_{i,m}$ is the number of output tuples we get by fully executing the ith join direction. Since O is the total output rate of the join, we have:

$$O = \sum_{i=1}^{m} \lambda_i \cdot N_{i,m}$$

REFERENCES

[1] Agarwal, P. K., Arge, L., and Erickson, J., "Indexing moving points," in *ACM Symposium on Principles of Database Systems (PODS)*, 2000.

[2] Aggarwal, C. C. and Agrawal, D., "On nearest neighbor indexing of nonlinear trajectories," in *ACM Symposium on Principles of Database Systems (PODS)*, 2003.

[3] Akyildiz, I. and Wang, W., "A dynamic location management scheme for next generation multi-tier pcs systems," *IEEE Transactions on Wireless Communications*, vol. 1, no. 1, pp. 178–190, 2002.

[4] Amis, A. D., Prakash, R., Huynh, D., and Vuong, T., "Max-Min D-cluster formation in wireless ad hoc networks," in *IEEE Conference on Computer Communications*, 2000.

[5] ans Ramesh Govindan, C. I. and Estrin, D., "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[6] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., and Widom, J., "STREAM: The stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, 2003.

[7] Arici, T., Gedik, B., Altunbasak, Y., and Liu, L., "PINCO: A pipelined in-network compression scheme for data collection in wireless sensor networks," in *IEEE International Conference on Computer Communications and Networks*, 2003.

[8] Ayad, A. M. and Naughton, J. F., "Static optimization of conjunctive queries with sliding windows over infinite streams," in *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[9] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J., "Models and issues in data stream systems," in *ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[10] Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., and Zdonik, S., "Retrospective on Aurora," *International Journal on Very Large Data Bases (VLDB Journal), Special Issue on Data Stream Processing*, 2004.

[11] Balakrishnan, H., Madden, S., Bychkovsky, V., Chen, K., Daher, W., Goraczko, M., Hu, H., Hull, B., Miu, A., and Shih, E., "Cartel: A mobile sensor computing system." http://cartel.csail.mit.edu/abstracts/cartel_oview.html, (Accessed May 12, 2006).

[12] Bandyopadhyay, S. and Coyle, E. J., “An energy efficient hierarchical clustering algorithm for wireless sensor networks,” in *IEEE Conference on Computer Communications*, 2003.

[13] Bar-Noy, A., Kessler, I., and Sidi, M., “Mobile users: To update or not to update?,” *ACM Wireless Networks*, vol. 1, no. 2, pp. 175–185, 1995.

[14] Basagni, S., “Distributed clustering for ad hoc networks,” in *International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, 1999.

[15] Batalin, M., Rahimi, M., Yu, Y., Liu, S., Sukhatme, G., and Kaiser, W., “Call and response: Experiments in sampling the environment,” in *ACM Conference on Embedded Networks Sensor Systems (SenSys)*, 2004.

[16] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B., “The R*-Tree: An efficient and robust access method for points and rectangles,” in *ACM International Conference on Management of Data (SIGMOD)*, 1990.

[17] Benetis, R., Jensen, C. S., Karciauskas, G., and Saltenis, S., “Nearest neighbor and reverse nearest neighbor queries for moving objects,” in *International Database Engineering and Applications Symposium (IDEAS)*, 2002.

[18] Benetis, R., Jensen, C. S., Kareiauskas, G., and Saltenis, S., “Nearest and reverse nearest neighbor queries for moving objects,” *International Journal on Very Large Data Bases (VLDB Journal)*, 2006. DOI 10.1007/s00778-005-0166-4.

[19] Bennett, F., Clarke, D., Evans, J., Hopper, A., Jones, A., and Leask, D., “Piconet: Embedded mobile networking,” *IEEE Personal Communications*, vol. 4, no. 5, pp. 8–15, 1997.

[20] Bettstetter, C. and Krausser, R., “Scenario-based stability anlysis of the distributed mobility-adaptive clustering (DMAC) algorithm,” in *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2001.

[21] Bhattacharya, A. and Das, S. K., “Lezi-update: An information-theoretic approach to track mobile users in PCS networks,” in *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[22] Broch, J., Maltz, D. A., Johnson, D. B., Hu, Y.-C., and Jetcheva, J., “A performance comparison of multi-hop wireless ad hoc network routing protocols,” in *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[23] Byckovskiy, V., Megerian, S., Estrin, D., and Potkonjak, M., “A collaborative approach to in-place sensor calibration,” in *IEEE International Symposium on Information Processing in Sensor Networks (ISPN)*, 2003.

[24] Cai, Y., Hua, K., and Cao, G., “Processing range-monitoring queries on heterogeneous mobile objects,” in *IEEE International Conference on Mobile Data Management (MDM)*, 2004.

[25] Cai, Y. and Hua, K. A., "An adaptive query management technique for efficient real-time monitoring of spatial regions in mobile database systems," in *IEEE International Performance Computing and Communications Conference*, 2002.

[26] Casella, G. and Berger, R. L., *Statistical Inference.* Duxbury Press, June 2001.

[27] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Raman, V., Reiss, F., and Shah, M. A., "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[28] Chandrasekaran, S. and Franklin, M. J., "Remembrance of streams past: Overload-sensitive management of archived streams," in *International Conference on Very Large Data Bases (VLDB)*, 2004.

[29] Chatterjee, M., Das, S., and Turgut, D., "WCA: A weighted clustering algorithm for mobile ad hoc networks," *Journal of Cluster Computing*, vol. 5, April 2002.

[30] Chen, G. and Stojmenovic, I., "Clustering and routing in mobile wireless networks," tech. rep., SITE, University of Ottawa, 1999.

[31] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y., "NiagaraCQ: A scalable continuous query system for Internet databases," in *ACM International Conference on Management of Data (SIGMOD)*, 2000.

[32] Chipara, O., Lu, C., and Roman, G.-C., "Efficient power management based on application timing semantics for wireless sensor networks," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[33] Civilis, A., Jensen, C. S., Nenortaite, J., and Pakalnis, S., "Efficient tracking of moving objects with precision guarantees," in *IEEE International Conference on Mobile and Ubiquitous Systems: Networks and Services, (MobiQuitous)*, 2004.

[34] Civilis, A., Jensen, C. S., and Pakalnis, S., "Techniques for efficient road-network-based tracking of moving objects," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 5, pp. 698–712, 2005.

[35] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W., "Freenet: A distributed anonymous information storage and retrieval system," in *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[36] "Crossbow technology." http://www.xbow.com, (Accessed November 24, 2004).

[37] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I., "Wide-area cooperative storage with CFS," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[38] Das, A., Gehrke, J., and Riedewald, M., "Approximate join processing over data streams," in *ACM International Conference on Management of Data (SIGMOD)*, 2003.

[39] Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., and Hong, W., "Model-driven data acquisition in sensor networks," in *International Conference on Very Large Data Bases (VLDB)*, 2004.

[40] Elson, J. and Estrin, D., "Time synchronization for wireless sensor networks," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[41] Estrin, D., Culler, D., Pister, K., and Sukhatme, G., "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing*, vol. 1, January 2002.

[42] Estrin, D., Govindan, R., Heidemann, J. S., and Kumar, S., "Next century challenges: Scalable coordination in sensor networks," in *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[43] Fujimoto, R. M., *Parallel and Distributed Simulation Systems.* WileyInterscience, 2000.

[44] Gaede, V. and Gunther, O., "Multidimensional access methods," *ACM Computing Surveys*, vol. 30, no. 2, pp. 170–231, 1998.

[45] Gedik, B. and Liu, L., "PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[46] "Global precipitation climatology project." http://www.ncdc.noaa.gov/oa/wmo/wdcamet-ncdc.html, (Accessed November 24, 2004).

[47] Gnutella, "The gnutella home page." http://gnutella.wego.com/, (Accessed December 24, 2002).

[48] Golab, L., Garg, S., and Ozsu, M. T., "On indexing sliding windows over online data streams," in *International Conference on Extending Database Technology (EDBT)*, 2004.

[49] Golab, L. and Ozsu, M. T., "Processing sliding window multi-joins in continuous queries over data streams," in *International Conference on Very Large Data Bases (VLDB)*, 2003.

[50] "Google alerts." http://www.google.com/alerts, (Accessed February 14, 2006).

[51] "Google ridefinder." http://labs.google.com/ridefinder, (Accessed February 14, 2006).

[52] Guestrin, C., Thibaux, R., Bodik, P., Paskin, M. A., and Madden, S., "Distributed regression: An efficient framework for modeling sensor network data," in *IEEE International Symposium on Information Processing in Sensor Networks (ISPN)*, 2004.

[53] Hammad, M. A. and Aref, W. G., "Stream window join: Tracking moving objects in sensor-network databases," in *Scientific and Statistical Database Management, SSDBM*, 2003.

[54] Hamza-Lup, G. L., Hua, K. A., Peng, R., and Ho, A. H., "A maximum flow approach to dynamic handling of multiple incidents in traffic evacuation management," in *IEEE Conference on Intelligent Transportation Systems (ITSC)*, 2005.

[55] Han, J. and Kamber, M., *Data Mining: Concepts and Techniques.* Morgan Kaufmann, August 2000.

[56] Han, Q., Mehrotra, S., and Venkatasubramanian, N., "Energy efficient data collection in distributed sensor environments," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[57] Has, Z. J., "Panel report on ad hoc networks," *Mobile Computing and Communications Review*, vol. 2, no. 1, 1998.

[58] Helmer, S., Westmann, T., and Moerkotte, G., "Diag-Join: An opportunistic join algorithm for 1:N relationships," in *International Conference on Very Large Data Bases (VLDB)*, 1998.

[59] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K., "System architecture directions for network sensors," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[60] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K., "System architecture directions for network sensors," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[61] Huang, Q., Lu, C., and Roman, G. C., "Spatiotemporal multicast in sensor networks," in *ACM Conference on Embedded Networks Sensor Systems (SenSys)*, 2003.

[62] Ilarri, S., Mena, E., and Illarramendi, A., "A system based on mobile agents for tracking objects in a location-dependent query processing environment," in *International Workshop on Database and Expert Systems Applications (DEXA)*, 2001.

[63] Imielinski, T., Viswanathan, S., and Badrinath, B., "Energy efficient indexing on air," in *ACM International Conference on Management of Data (SIGMOD)*, 1994.

[64] I.Stojmenovic, Seddigh, M., and Zunic, J., "Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 13, no. 1, 2002.

[65] Jensen, C. S., Lin, D., and Ooi, B. C., "Query and update efficient B+-tree based indexing of moving objects," in *International Conference on Very Large Data Bases (VLDB)*, 2004.

[66] J.Kucera and Lott, U., "Single chip 1.9 ghz transceiver frontend mmic including Rx/Tx local oscillators and 300 mw power amplifier," *MTT Symposium Digest*, vol. 4, pp. 1405–1408, June 1999.

[67] Kalashnikov, D. V., Prabhakar, S., Hambrusch, S., and Aref, W., "Efficient evaluation of continuous range queries on moving objects," in *International Workshop on Database and Expert Systems Applications (DEXA)*, 2002.

[68] Kang, J., Naughton, J., and Viglas, S., "Evaluating window joins over unbounded streams," in *IEEE International Conference on Data Engineering (ICDE)*, 2003.

[69] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R., "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing (STOC)*, 1997.

[70] Karp, B. and Kung, H. T., "GPSR: greedy perimeter stateless routing for wireless networks," in *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[71] KaZaa, "The kazaa home page." http://www.kazaa.com/, (Accessed January 04, 2006).

[72] Kollios, G., Gunopulos, D., and Tsotras, V. J., "On indexing mobile objects," in *ACM Symposium on Principles of Database Systems (PODS)*, 1999.

[73] Kotidis, Y., "Snapshot queries: Towards data-centric sensor networks," in *IEEE International Conference on Data Engineering (ICDE)*, 2005.

[74] Lazaridis, I., Porkaew, K., and Mehrotra, S., "Dynamic queries over mobile objects," in *International Conference on Extending Database Technology (EDBT)*, 2002.

[75] Li, D., K.Wong, Hu, Y., and Sayeed, A., "Detection, classification, tracking of targets in micro-sensor networks," *IEEE Signal Processing Magazine*, March 2002.

[76] Lin, C. R. and Gerla, M., "Adaptive clustering for mobile wireless networks," *IEEE Journal of Selected Areas in Communications*, vol. 15, no. 7, 1997.

[77] Liu, J., Reich, J., Cheung, P., and Zhao, F., "Distributed group management for track initiation and maintenance in target localization applications," in *Workshop on Information Processing in Sensor Networks*, 2003.

[78] Liu, L., Pu, C., and Tang, W., "Continual queries for internet scale event-driven information delivery," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pp. 610–628, July/August 1999.

[79] Liu, L., Pu, C., and Tang, W., "Continual queries for internet scale event-driven information delivery," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 11, July/August 1999.

[80] Liu, L., Pu, C., and Tang, W., "Detecting and delivering information changes on the web," in *ACM International Conference on Information and Knowledge Management (CIKM)*, November 2000.

[81] Liu, L., Tang, W., Buttler, D., and Pu, C., "Information monitoring on the web: A scalable solution," *Springer World Wide Web*, 2002.

[82] Madden, S., Franklin, M., Hellerstein, J., and Hong, W., "Tag: a tiny aggregation service for ad-hoc sensor networks," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[83] Madden, S., Szewczyk, R., Franklin, M., and Culler, D., "Supporting aggregate queries over ad-hoc wireless sensor networks," in *IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[84] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J., "Wireless sensor networks for habitat monitoring," in *ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[85] Mills, D. L., "Internet time synchronization: The network time protocol," *IEEE Transactions on Communications*, pp. 1482–1493, 1991.

[86] Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Rollins, B. R. S., and Xu, Z., "Peer-to-peer computing," Tech. Rep. HPL-2002-57R1, Hewlett Packard Labs, 2002.

[87] Mokbel, M. F., Xiong, X., and Aref, W. G., "SINA: Scalable incremental processing of continuous queries in spatio-temporal databases," in *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[88] "Moteiv. telos revb datasheet." http://www.moteiv.com/pr/2004-12-09-telosb.php, (Accessed December 30, 2004).

[89] Naor, Z. and Levy, H., "Minimizing the wireless cost of tracking mobile users: An adaptive threshold scheme," in *IEEE Conference on Computer Communications*, 1998.

[90] Paskin, M. A. and Guestrin, C. E., "A robust architecture for distributed inference in sensor networks," in *IEEE International Symposium on Information Processing in Sensor Networks (ISPN)*, 2005.

[91] Patel, J. M., Chen, Y., and Chakka, V. P., "STRIPES: An efficient index for predicted trajectories," in *ACM International Conference on Management of Data (SIGMOD)*, 2004.

[92] Pfoser, D., Jensen, C. S., and Theodoridis, Y., "Novel approaches in query processing for moving object trajectories," in *International Conference on Very Large Data Bases (VLDB)*, 2000.

[93] Plaxton, C. G., Rajaraman, R., and Richa, A. W., "Accessing nearby copies of replicated objects in a distributed environment," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1997.

[94] Prabhakar, S., Xia, Y., Kalashnikov, D. V., Aref, W. G., and Hambrusch, S. E., "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," *IEEE Transactions on Computers*, vol. 51, no. 10, pp. 1124–1140, 2002.

[95] Pradhan, S., Kusuma, J., and Ramchandran, K., "Distributed compression in a dense sensor network," *IEEE Signal Processing Magazine*, March 2002.

[96] Priyantha, N., Balakrishnan, H., Demaine, E., and Teller, S., "Anchor-free distributed localization in sensor networks," tech. rep., MIT Laboratory for Computer Science, 2003.

[97] Pu, C. and Singaravelu, L., "Fine-grain adaptive compression in dynamically variable networks," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[98] QualComm, "Wireless access solutions using 1xEV-DO." http://www.qualcomm.com/technology/1xev-do/webpapers/wp_wirelessaccess.pdf, (Accessed March 08, 2005).

[99] Ramaswamy, L., Gedik, B., and Liu, L., "Connectivity based node clustering in decentralized peer-to-peer networks," in *IEEE International Conference on Peer-to-Peer Computing*, 2003.

[100] Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I., "Load balancing in structured p2p systems," in *International Workshop on Peer-to-Peer Systems*, 2003.

[101] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., "A scalable content-addressable network," in *ACM SIGCOMM Conference*, 2001.

[102] Ricart, G. and Agrawala, A. K., "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM (CACM)*, pp. 9–17, 1981.

[103] Rowstron, A. and Druschel, P., "Pastry: Scalable, decentralized object location and routing for largescale peer-to-peer systems," in *ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[104] Rowstron, A., Kermarrec, A., Castro, M., and Druschel, P., "SCRIBE: The design of a large-scale event notification infrastructure," in *International Workshop on Networked Group Communication*, 2001.

[105] Saltenis, S., Jensen, C. S., Leutenegger, S. T., and Lopez, M. A., "Indexing the positions of continuously moving objects," in *ACM International Conference on Management of Data (SIGMOD)*, 2000.

[106] Sankarasubramaniam, Y., Akan, O. B., and Akylidiz, I. F., "ESRT: Event-to-sink reliable transport in wireless sensor networks," in *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2003.

[107] Saroiu, S., Gummadi, P. K., and Gribble, S. D., "A measurement study of peer-to-peer file sharing systems," Tech. Rep. UW-CSE-01-06-02, University of Washington, 2001.

[108] Satyanarayanan, M., "Fundamental challenges in mobile computing," in *ACM Symposium on Principles of Distributed Computing (PODC)*, November 1996.

[109] Schneider, F. B., "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[110] Schurgers, C. and Srivastava, M. B., "Energy efficient routing in wireless sensor networks," in *Military Communications Conference (MILCOM)*, 2001.

[111] Science, C. and Board, T., *IT Roadmap to a Geospatial Future.* The National Academics Press, November 2003.

[112] Sistla, A. P., Wolfson, O., Chamberlain, S., and Dao, S., "Modeling and querying moving objects," in *IEEE International Conference on Data Engineering (ICDE)*, 1997.

[113] Song, Z. and Roussopoulos, N., "Hashing moving objects," in *IEEE International Conference on Mobile Data Management (MDM)*, 2001.

[114] Song, Z. and Roussopoulos, N., "K-nearest neighbor search for moving query point," in *International Symposium on Spatial and Temporal Databases (SSTD)*, 2001.

[115] Srivastava, U. and Widom, J., "Memory-limited execution of windowed stream joins," in *International Conference on Very Large Data Bases (VLDB)*, 2004.

[116] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H., "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Conference*, 2001.

[117] "Streambase systems." http://www.streambase.com/, (Accessed December 17, 2005).

[118] Tao, Y., Papadias, D., and Sun, J., "The TPR*-Tree: An optimized spatio-temporal access method for predictive queries," in *International Conference on Very Large Data Bases (VLDB)*, 2003.

[119] Tao, Y. and Papadias, D., "Time-parameterized queries in spatio-temporal databases," in *ACM International Conference on Management of Data (SIGMOD)*, 2002.

[120] Tao, Y., Papadias, D., and Shen, Q., "Continuous nearest neighbor search," in *International Conference on Very Large Data Bases (VLDB)*, 2002.

[121] "Taos Inc. ambient light sensor (ALS)." http://www.taosinc.com/images/product/-document/tsl2550-e58.pdf, (Accessed November 24, 2004).

[122] Tatbul, N. and Zdonik, S., "No false positives: Window-aware load shedding for data streams," Tech. Rep. CS-05-06, Brown University, 2005.

[123] Tatbul, N., Cetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M., "Load shedding in a data stream manager," in *International Conference on Very Large Data Bases (VLDB)*, 2003.

[124] Terry, D., Goldberg, D., Nichols, D., and Oki, B., "Continuous queries over append-only databases," in *ACM International Conference on Management of Data (SIGMOD)*, 1992.

[125] "The zooknic internet geography project." http://www.zooknic.com, (Accessed February 14, 2006).

[126] Theodoridis, Y., Stefanakis, E., and Sellis, T., "Efficient cost models for spatial queries using R-trees," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 12, no. 1, pp. 19–32, 2000.

[127] "US Naval Observatory GPS Operations." http://tycho.usno.navy.mil/gps.html, (Accessed April 22, 2003).

[128] Viglas, S. D., Naughton, J. F., and Burger, J., "Maximizing the output rate of multi-way join queries over streaming information sources," in *International Conference on Very Large Data Bases (VLDB)*, 2003.

[129] Wolfson, O., "The opportunities and challenges of location information management," in *Intersections of Geospatial Information and Information Technology Workshop*, 2001.

[130] Wolfson, O., "Moving objects information management: The database challenge," in *Next Generation Information Technologies and Systems (NGITS)*, 2002.

[131] Wolfson, O., Sistla, P., Chamberlain, S., and Yesha, Y., "Updating and querying databases that track mobile units," *Springer Distributed and Parallel Databases*, vol. 7, no. 3, pp. 257–387, 1999.

[132] Wolfson, O., Xu, B., Chamberlain, S., and Jiang, L., "Moving objects databases: Issues and solutions," in *Statistical and Scientific Database Management*, 1998.

[133] Wu, K.-L., Chen, S.-K., and Yu, P. S., "Processing continual range queries over moving objects using VCR-based query indexes," in *IEEE International Conference on Mobile and Ubiquitous Systems: Networks and Services, (MobiQuitous)*, 2004.

[134] Wu, K.-L., Chen, S.-K., and Yu, P. S., "On incremental processing of continual range queries for location-aware services and applications," in *IEEE International Conference on Mobile and Ubiquitous Systems: Networks and Services, (MobiQuitous)*, 2005.

[135] Xiao, L., Boyd, S., and Lall, S., "A scheme for robust distributed sensor fusion based on average consensus," in *IEEE International Symposium on Information Processing in Sensor Networks (ISPN)*, 2005.

[136] Xie, J., Yang, J., and Chen, Y., "On joining and caching stochastic streams," in *ACM International Conference on Management of Data (SIGMOD)*, 2005.

[137] Xiong, X., Mokbel, M. F., and Aref, W. G., "SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *IEEE International Conference on Data Engineering (ICDE)*, 2005.

[138] Xu, B., Ouksel, A., and Wolfson, O., "Opportunistic resource exchange in inter-vehicle ad hoc networks," in *IEEE International Conference on Mobile Data Management (MDM)*, 2004.

[139] Ye, F., Zhong, G., Lu, S., and Zhang, L., "PEAS: A robust energy conserving protocol for long-lived sensor networks," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[140] Ye, W., Heidemann, J., and Estrin, D., "An energy-efficient mac protocol for wireless sensor networks," in *IEEE Conference on Computer Communications*, 2002.

[141] Yu, X., Pu, K. Q., and Koudas, N., "Monitoring k-nearest neighbor queries over moving objects," in *IEEE International Conference on Data Engineering (ICDE)*, 2005.

[142] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D., "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, University of California Berkeley, 2001.

VITA

Buğra Gedik was born in the capital city Ankara and was raised in the northeastern province Trabzon, both in Turkey. He obtained a bachelors degree (B.S.) in Computer Science, from the Computer Engineering and Information Science department of Bilkent University in 2001 (Ankara, Turkey). Subsequently, he joined the Computer Science Ph.D. program at the College of Computing of Georgia Institute of Technology (Atlanta, GA, USA). As a member of the DiSL research group at the College of Computing, he conducted research on various aspects of distributed data intensive systems, including peer-to-peer computing, mobile data management, and sensor network computing. He led three projects in the DiSL research group, namely PeerCQ, MobiEyes, and SensorCQ. His research in these projects has resulted in numerous publications that have appeared in various international conferences and journals on distributed systems and data management. He has also been a collaborator with the IBM T.J. Watson Research Center. He holds or applied for a number of patents on his work at IBM, dealing with data stream processing systems.