Mobile Query Processing Incorporating Multiple Non-collaborative Servers

Say Ying Lim, BBusSys (Monash), MBusSys (Monash)

Thesis submitted in total fulfillment of the requirements for the degree of

Doctor of Philosophy

Monash University

July 2007

Declaration

This thesis is titled:

Mobile Query Processing Incorporating Multiple Non-Collaborative Servers

Written by Say Ying Lim

Thesis directed by Prof. Balasubramaniam Srinivasan and Dr. David Taniar

2007

Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis submitted for the award of any other degree of diploma.

No other person's work has been used without due acknowledgement in the main text of the thesis.

The thesis has not been submitted for the award of any degree or diploma in any other tertiary institution.

_____

Say Ying Lim

Abstract

This thesis studies mobile query processing incorporating multiple non-collaborative servers. The main objective is to investigate how to execute different types of queries taking into account the limitations of the mobile environment. The primary focus is on mobile join and mobile aggregate queries in a wireless environment involving multiple non-collaborative servers. These are the two most common types of queries in traditional database processing, and even more important in a mobile database environment taking into account limitations of a typical mobile environment, including limited computation capabilities, storage resources, and visualization.

Within the mobile environment incorporating multiple non-collaborative servers, join operation is crucial because mobile users always invoke a query that needs to gather information from multiple sources. Aggregate query is also important in a mobile environment due to its nature of summarizing the required information to suit the small display screen of the mobile device, as well as to fit into the limited memory capacity.

Subsequently, this thesis focuses on two major aspects relating to *mobile join query processing* and *mobile aggregate query processing*. Mobile join query processing includes basic mobile join and Top-k join query processing, whereas mobile aggregate query processing deals with Groupby-Join and division query processing.

For mobile join query processing, we studied several versions of processing methods depending on whether the join processing is carried out in the mobile device or in the server. Hence, the proposed algorithms are known as mobile device side processing (MDSP) and server side processing (SSP). We have studied sort-merge and hash join methods, and found that the sort-merge join method appears to out-perform the hash-based join method, since in a client-server architecture, it is

desirable to outsource the heavy process (e.g. sorting) to the server, and let the mobile client perform the lightweight process (e.g. merging). In addition, a number of block-based processing methods have also been investigated in order to minimize memory utilization and data transfer costs.

Another type of join we investigated is mobile Top-k- join queries. Top-k join queries process particular information that is deemed important within the list. In addition, it displays only the top ranked important information and hence, it is highly suitable for small screen devices. In this thesis, we examined two processing methods, namely mobile Top-k nested-loop and mobile Top-k merge. We also outlined the main difference between our Top-k merge and the traditional merging approach.

For mobile aggregate query processing, we studied in particular mobile groupby-join and mobile division, since both commonly utilize aggregate operations. The groupby-join query processing, traditionally, is divided into groupby-before-join and groupby-after-join. Apart from carrying these out in a mobile environment, the main contribution is demonstrating the ability to process the groupby operation before the join operation, even in the groupby-after-join queries. Consequently, we have achieved a much better performance improvement.

Within the division query processing, beside the relational division, we introduced multiple group division query processing, where the divisor exists in multiple groups. For the relational division, the count-based is employed; whilst for the multiple group division, both sort-merge and aggregate approaches are used. The main contribution in this context is the formulation of processing methods for mobile multiple group division queries. For the aggregate approach, we proposed the pruning concept, which manages to reduce the data transfer costs.

In order to evaluate the behaviour of the proposed methods, we have formulated analytical models to simulate the performance of each of the algorithms. Analytical performance evaluation provides the cost models for each proposed algorithm. The results of the performance are compared and analyzed. The performance graphs show the superiority of the proposed algorithms.

Acknowledgments

With deepest thanks I would like to acknowledge the invaluable assistance of the following people:

Professor Bala Srinivasan:

Being my main supervisor, Professor Srinivasan has offered me invaluable support and guidance on how to present my research findings in a meaningful way. He also took precious time from his busy schedule to give me advice, support and encouragement. I wish to express my sincere thanks for all his help. His generosity with his time started from the beginning of my PhD candidature and extended to the very end.

Dr David Taniar:

The next important person whom I sincerely want to thank is Dr Taniar, my co-supervisor. Not only did he provide me with the guidance that I needed throughout my candidature, he has also given me continuous support in every possible ways until the completion of this thesis. Without his guidance, I would not have been able to complete the writing of this thesis smoothly. In addition, he has also shown me the enjoyable aspects of being a student, from Bachelor to Master and then to PhD, a memory that will always be cherished.

Family members:

I also want to take this golden opportunity to thank my parents and sisters who have always been by my side supporting me continuously in every possible way. Not only have they supported me financially, but they have also consistently given me valuable and unswerving encouragement throughout my career as a student, all of which I value.

Monash University:

Thank you to the Faculty of Information Technology for awarding me the scholarship and I am especially grateful to all staff members and fellow students, especially those at Caulfield School of Information Technology for their unconditional help and friendship.

Table of Contents

List of Figures

List of Tables

List of Publications

Lim, S.Y, Taniar, D. and Srinivasan, B., "Mobile Information Processing Involving Multiple Non-Collaborative Sources", *International Journal of Business Data Communications and Networking*, **3**(2):72-93, 2007.

Lim, S.Y., Taniar, D., and Srinivasan, B., "Data Caching in a Mobile Database Environment", *Business Data Communications and Networking: A Research Perspective*, Chapter VIII, pp. 187-210, 2007.

Lim, S.Y., Taniar, D., and Srinivasan, B., "A Taxonomy of Database Operations on Mobile Devices", *Mobile Multimedia: A Communication Engineering Perspective*, Chapter 10, pp. 197-215, 2006.

Lim, S.Y., Taniar, D., and Srinivasan, B., "On-Mobile Aggregate Query Processing incorporating Multiple Non-Collaborative Servers", *Proceedings of the 3rd International Conference on Advances in Mobile Multimedia (MoMM2005)"*, books@ocg.at Band 195, Austrian Computer Society (OCG), pp. 21-30, Kuala Lumpur, 2005.

Lim, S.Y., Taniar, D., and Srinivasan, B., "User Interface Design for Decision Guide Websites", *Proceedings of the 7th International Conference on Information Integration and Web-based Applications and Services (iiWAS'2005)*, books@ocg.at Band 196, Austrian Computer Society, pp. 619-628, Kuala Lumpur, 2005.

Lim, S.Y., Taniar, D., and Srinivasan, B., "On-Mobile Query Processing incorporating Multiple Non-Collaborative Servers", *Ingénierie des Systèmes d'Information*, Special Issue on Mobility in the Information Systems and Databases, **10**(5):9-35, 2005.

Lim, S.Y., Taniar, D., and Srinivasan, B., "Mobile Information Processing incorporating Location-Based Services", *Proceedings of the IEEE 3rd International Conference on Industrial Informatics, INDIN 2005*, Perth, Australia, IEEE Computer Society Press, pp. 1-6, 2005.

Chapter 1

Introduction

1.1 Overview

This thesis investigates query processing techniques in a mobile environment incorporating multiple non-collaborative servers. This chapter introduces an overview of the nature of this thesis as well as its aims and objectives.

Mobile technology has been increasingly in demand and is widely used to allow people to be connected wirelessly without having to worry about the distance barrier (Bose et al., 2005; Chrysanthis and Pitoura, 2000). It can be seen as a new resource for accomplishing various everyday activities that are carried out on the move. The direction of the mobile technology industry is beginning to evolve as more mobile users have emerged. This new technology enables users to access information anytime, anywhere (Acharya, Kumar and Yang, 2007; Heuer and Lubinski, 1996; Kottkamp and Zukunft, 1998).

People have a tremendous capacity for utilizing mobile devices in innovative ways for various purposes. Mobile devices are capable of processing and retrieving data from multiple remote databases (Lo et al., 2004; Malladi and Davis, 2003). This allows mobile users who wish to collect data from different remote databases by sending queries to the servers and then be able to process the multiple information gathered from these sources locally on the mobile devices (Mamoulis et al., 2003;

Ozakar, Morvan and Hameurlain, 2005). By processing the data locally, mobile users would have more control of what they actually want as the final results of the query. They can therefore choose to query information from different servers and join them to be processed locally according to their requirements. Also, being able to obtain specific information over several different sites would help to optimise results to mobile users' queries. This is because different sites may give different insights into a particular issue and if all these different insights are joined together, the returned result would be more complete.

Example 1.1: An international tourist, while traveling within a foreign country, does not know the whereabouts of the available vegetarian restaurants. He looks for restaurants recommended by both the Tourist Office and Vegetarian Community. First, using his wireless PDA, he would download information broadcast by the Tourist Office. Then, he would download the information provided by the second organization mentioned above. Once he has obtained the two lists from the two information providers that may not correspond to each other, he may perform an operation on his mobile device that joins the contents from the two related providers. This illustrates the importance of being able to collate, in a mobile device, information obtained from various non-collaborative sources.

In order to derive and assemble data from several servers in a mobile device, a new research domain known as *Mobile Query Processing incorporating Multiple Non-collaborative Servers* has arisen, and this topic is the central focus of this thesis.

The subsequent sub-sections cover some basic concepts relating to the meaning of non-collaborative servers, and deal with the emerging issues and complexities that are currently faced in the mobile environment.

1.1.1 Multiple Non-collaborative Servers

The word ‘collaborative’ usually relates to the traditional distributed databases where the aim is to integrate the data of a particular enterprise and to provide centralized and controlled access to that data (Özsu and Valduriez, 1997). The main motivating factor behind the notion of distributed databases is the realization of an important concept of database technology, which is data integration and not centralization.

Basically, it is wise to fragment the centralized server. There are two different types of fragmentation that are normally found in distributed databases which include the horizontal fragmentation approach or vertical fragmentation approach (Taniar et al., 2008; Vlach, 2000). The difference can be seen in Figure 1.1 which demonstrates how a centralized database uses the horizontal and vertical fragmentation approach respectively. Each type of fragmentation that is being used has its own advantages and disadvantages according to the needs of the organization.

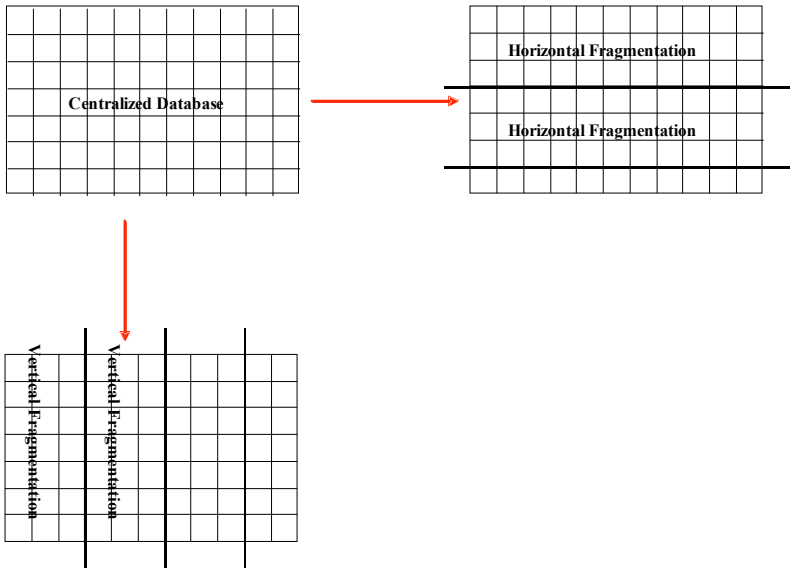


Figure 1.1: Horizontal vs. Vertical Fragmentation

Example 1.2: The Village cinemas fragment the central data that keeps all movies of the Village cinemas into several servers on each Village branch. A user currently in the Glen Waverley Village cinema may want to obtain data from the Glen

Waverley server, the City server and the Crown server. This poses the question of how to integrate the data: that is, whether to transfer data from Glen Waverley to integrate with City, or whether to call City data to be processed. The issue arises of which path is the best for processing the query.

Figure 1.2 shows two possible paths that can be used to integrate data from the three different cinema branches. There are many other possible paths that can be taken to integrate the data from the three different sources. Although distributed databases deal with integration data from multiple sources, the sources here are within the same enterprise (Elmasri and Navathe, 2007; Silberschatz, Korth and Sudarsan, 2006). However, our focus is on the data servers that are independent, and hence non-collaborative.

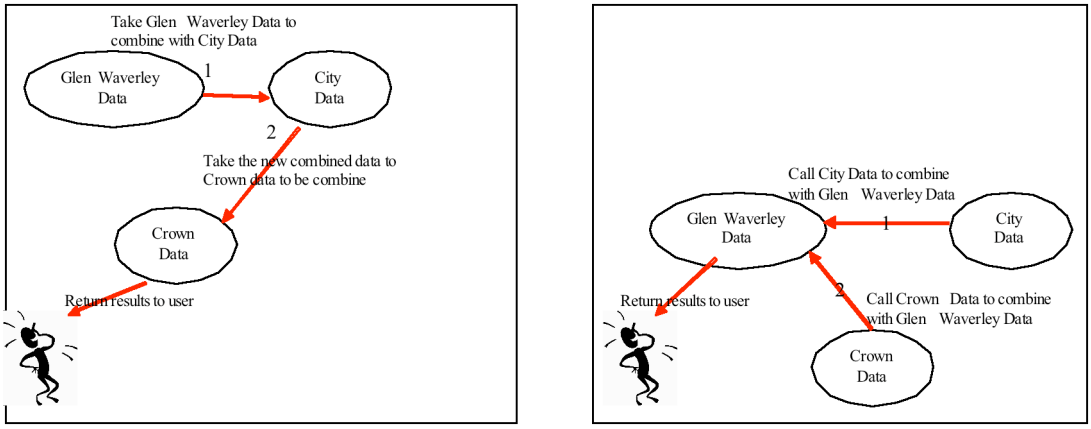


Figure 1.2: Query processing in a distributed database

The technology of distributed databases may not be appropriate for use in the mobile environment which involves not only the nomadic clients who move around, but also non-collaborative servers which are, basically, servers that are maintained by different organizations (Lo et al., 2004). Therefore, non-collaborative or independent servers would refer to servers that do not know each other and are not related to one another. There are basically just individual server providers which disseminate data to the users and they do not communicate with one another.

Since servers may be just independent service providers, often these servers are specialized within the domain. Hence, the information they are providing, such as information on restaurants disseminated by a server, normally just focuses on the restaurants information with perhaps additional limited supporting information which might include how to get there (information about transport). However, this is only supporting information since it does not exactly show the route that the user should take from his present location. Therefore, there is still a need to obtain full information from multiple servers, in this case, the servers that deal with restaurants and transportation separately.

Example 1.3: A mobile user may want to know the timetable for the transportation services to a particular event. Transportation timetables, as well as the events, are normally stored in different servers, such as a transport server and an event server. The transport server would deal with transportation data, while an event server would deal with currently scheduled events. Therefore, in order to know the transportation timetable for a particular event, the user has to gather data from the two different servers which are normally independent of each other.

In addition, not all service providers are supported by the use of a mediator (Lo et al, 2004). Therefore, information obtained from other independent non-related service providers needs to be processed individually. It should not be assumed that every service provider is linked through a mediator. Hence, in our research we focus on non-collaborative service providers and we refer to them as independent servers. Thus, it is vital to gather information from independent servers because it is often not enough to obtain data from just a single server.

1.1.2 Issues and Complexity of Local Mobile Database Operations

Due to the dynamic nature of this mobile environment, mobile devices face several limitations (Marsit et al., 2005, Pissinou, Makki and Campbell, 1999). These include limited processing capacity as well as storage capacity. Moreover, limited bandwidth is an issue because the wireless bandwidth is smaller compared with the fixed networks. This leads to poor connection and frequent disconnection. Another major issue would be the small screen which limits the visual display. Therefore, it is

important to comprehensively study how database operations may be carried out locally on mobile devices.

A wireless environment would consist of PDAs (Personal Digital Assistants), wireless network connections, and changing user environment (e.g. car, street, building site). This gives rise to some issues and complexity of the mobile operations. Also, the limited screen space is another constraint. If the results of the join are too large, then it is too cumbersome to be displayed on the small mobile device screen. The visual access is thus limited by the small screen of the mobile devices.

Processors may also be overloaded with time consuming joins, especially those that involve thousands of records from many different servers, and completion time can be expected to be longer.

Another issue to take into account is that, by having a complex join that involves a large amount of data, the consequences would be an increase in communication cost from this server. One must keep in mind that when using mobile devices, the aim is to minimize the communication cost which is the cost of shipping query and results from the database site to the requested site.

The above limitations - small displays, low bandwidth, low processing power as well as low battery life and operating memory - dramatically limit the query processing power. As a result, it is extremely important to study comprehensive database operations that are performed on mobile devices taking into account all the issues and complexities.

In the research project, the focus is mainly on information gathering from multiples sources and therefore we would not be focusing on the inherent characteristics of the mobile environment. We would assume that the mobile environment limitations such as unreliability of wireless communication, asymmetric communication etc. will be handled by the network providers.

1.2 Objectives of this Thesis

In this thesis, we will investigate the techniques for mobile query processing involving non-collaborative servers. We will consider different types of query requirements, as well as inherent constraints found in mobile devices and their environment. The main objective is to investigate how performance improvement of mobile query processing can be achieved by devising new algorithms for processing queries in the mobile environment.

In order to achieve the above stated objective, this thesis addresses the following issues:

i. To investigate techniques for performing join operations on the mobile devices locally and on the servers involving non-collaborative servers using several traditional join query processing methods.

ii. To extend the mobile join operation to include block-based processing for the purpose of minimizing memory utilization.

iii. To design algorithms that cope with the issue of a mobile device's small display screen, low battery life, lengthy processing time as well as high transfer cost when large data sets are involved. In order to achieve this, an innovative technique that can provide important Top-k query results is investigated.

iv. To design algorithms to deal with aggregation from various non-collaborative servers. Since the aggregation also involves multiple data sources, the query requirements need to combine aggregation and join operations which introduce the need to investigate GroupBy-join query processing.

v. To devise techniques that deal with division operation, as well as formulating an innovative division operation involving multiple groups of divisors.

vi. To formulate analytical models for the proposed mobile query processing, and to perform analysis. This is complemented by a simulation model in order to validate the accuracy of the analytical model.

This research addresses information gathering through query processing by mobile users that span across several non-collaborative servers. And in the real mobile environment, not all servers are able to accept direct queries. There may be occasions when the server may perform data broadcast. And if this is the case, in order to retrieve and process the data that are obtained individually from different types of servers together, there is a need for advanced techniques that are able to process the data that obtained from multiple sources together. There is no doubt that query processing incorporating multiple non-collaborative servers plays a crucial role, without which the performance and mobile users' satisfaction will not be very positive since very often users prefer and indeed demand data from several servers.

1.3 Scope of Research

It is widely recognized that traditional join queries are often unable to handle the gathering of data based on queries from multiple non-collaborative servers. Thus, this research will address several issues revolving around modifying the traditional queries algorithms as well as proposing new algorithms that can improve query processing from multiple non-collaborative servers. Moreover, the research will take into account other mobile factors such as limited screen display, limited memory capacity that cause performance bottleneck. Figure 1.3 defines the scope of this research which is mainly divided into three major areas: basic mobile query processing, advanced mobile query processing, and performance evaluation using analytical and simulation models. A query taxonomy will also be formulated to define the scope of mobile query processing.

Basic query processing includes formulating algorithms for processing join locally on the mobile device using data obtained from multiple servers regardless of whether the data are being broadcasted or are from direct querying. The plan is to develop the best join processing technique that is able to reduce the transfer cost. The mobile environment comprises several mobile issues that can be the environment itself or the device itself. Therefore, it is crucial to address this issue by investigating query processing techniques that cope with the nature of the mobile environment such

as processing Top-k results, and in particular to deal with small display screen of the mobile device itself.

We also include algorithms that use aggregate functions to further enhance processing and also achieve user queries that involve the GroupBy-Join operation. Division operations, which have not attracted many researchers, are developed and expanded. Due to the various methods of information gathering, in this thesis we focus on processing queries for information gathering in the mobile environment that involves multiple non-collaborative servers.

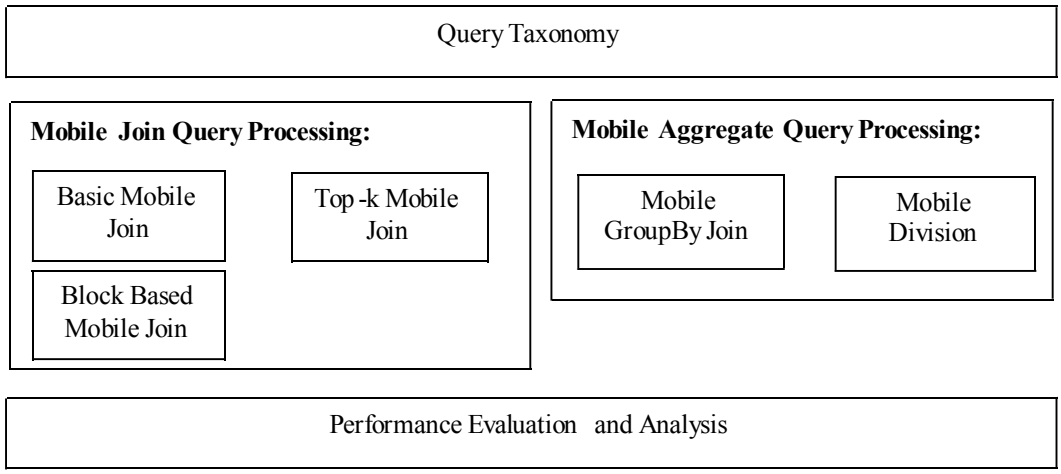


Figure 1.3: Scope of Research

1.4 Research Methodology

The main focus of this thesis is to improve the efficiency of query processing in the mobile environment. This research is carried out in a phased-based manner:

Taxonomy of Query Processing: This step helps one to fully understand and classify various types of database queries operation for the mobile environment. When formulating the taxonomy, it is desirable to provide an understanding of the possible database operations that can be performed on the mobile devices by the mobile users as well as providing a direction for query processing and optimization. The main aim of such a framework is to be able to identify and investigate the possible operations that can be carried out in mobile databases.

Problem Definition: Query processing is one of the well-studied areas in the field of database. In addition, mobile users often need to gather information from multiple sources that are distributed because there are times when information obtained from a single source may not be sufficient. To achieve efficiency in mobile query processing, it is important to identify the drawbacks of the state-of-the-art traditional join and mobile issues, and discover the causes of the limitation encountered when using these methods. Perhaps, such drawbacks will open up new opportunities for further improving the performance of mobile query processing incorporating multiple non-collaborative servers.

Design of Algorithms: Design several algorithms to enhance the performance of mobile query processing incorporating multiple non-collaborative servers. Apart from performance improvement, we also design a number of algorithms in order to consider mobile issues such as low battery life, lengthy processing time and transfer cost, and small display screen.

Performance Analysis: To demonstrate the efficiency of the proposed methods, all proposed methods and some of the state-of-the-art prior methods are compared in order to demonstrate their performance and verify their effectiveness when applied to various real world data sets. Additionally, several experiments have been carried out in the performance evaluation study by providing a thorough and complete analysis of each of the proposed methods. Then, in order to validate the efficiency and scalability of the proposed methods, several experiments were conducted allowing us to compare the performance of our proposed methods with corresponding existing methods.

1.5 Contributions

The specific contributions of this thesis are listed below, and the relationship between the contributions and the research scope is shown in Figure 1.4.

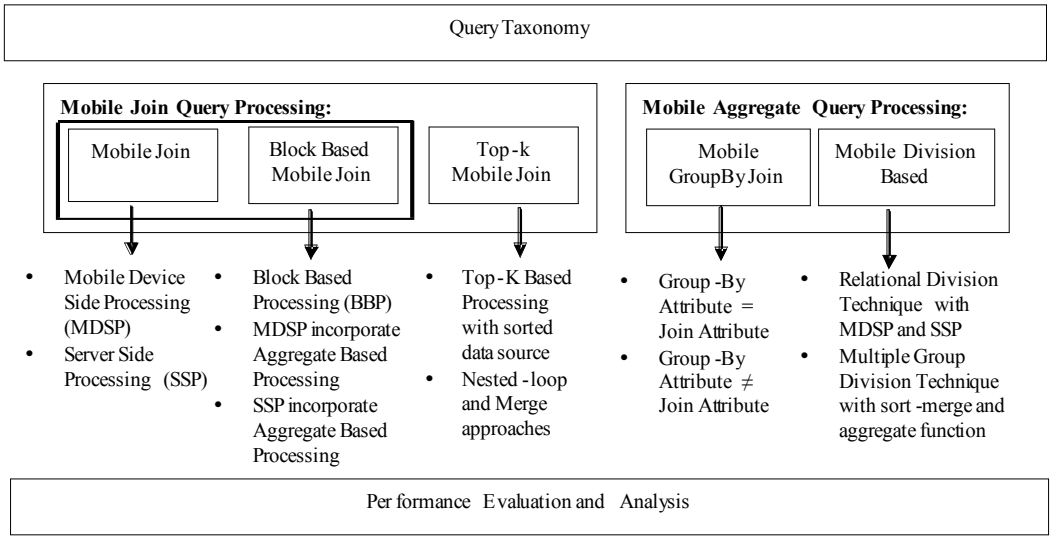


Figure 1.4: Thesis Contributions

- **Query Taxonomy**

  In addition to the well known traditional join query, other query types, such as Top-k join, groupby-join, and division, have been identified and expand considerably the complexity of mobile environment requirements involving multiple servers that are union compatible but are non-collaborative.

- **Mobile Join Processing**

  Two basic mobile join processing models that are introduced are join processing that occurs locally on the mobile device and join processing that requires the processing to be done on the server side. These two different processing sides are a useful combination for high performance depending on the amount of data sets.

- **Block-based Mobile Join Processing**

  In addition, block based processing is incorporated on top of the proposed join algorithms. This is introduced due to the aim of minimizing memory utilization and reducing processing cost.

- **Top-k Join Result Processing**

  Addressing the major mobile issues of small display screen, a collection of Top-k result based algorithms is developed. Due to the nature of different possible ways of obtaining data from servers, the proposed algorithms will take into account a server with query and non-query capability when downloading data from multiple servers. Various local processing methods including nested loop and sort merge are also studied in conjunction with the different ways of obtaining data from the servers.

- **Groupby-Join Based Processing**

  An extended type of operation is proposed which uses the aggregate function in addition to the proposed mobile device side and server side processing. This proposed technique is an extension type of operation and can complement well the basic join processing that is introduced. A selection of Groupby-Join has also been introduced to optimize query processing within the mobile environment in a more efficient and cost effective way in different circumstances.

- **Division Based Processing**

  Universal quantification is an important query in both traditional databases and in a mobile environment incorporating multiple non-collaborative servers. Aggregation is extended to be used in a division query. Additionally, the traditional division query focusing on one group of divisor is extended to cover multiple group divisors.

In summary, the main contribution of this research project is the development of techniques that can accommodate information processing involving multiple sources. The expected outcome of this research project is a set of proposed techniques

for processing the abovementioned queries, which are typically found in a mobile environment. In meeting user needs that may require gathering information from various different non-collaborative sources, this project will bring benefits to mobile device users; moreover, it will address current limitations such as small display screen, low battery life, low network bandwidth etc.

1.6 Thesis Organization

This thesis is organized into 8 chapters, and the inter-relationship between the chapters is depicted in the Figure 1.5.

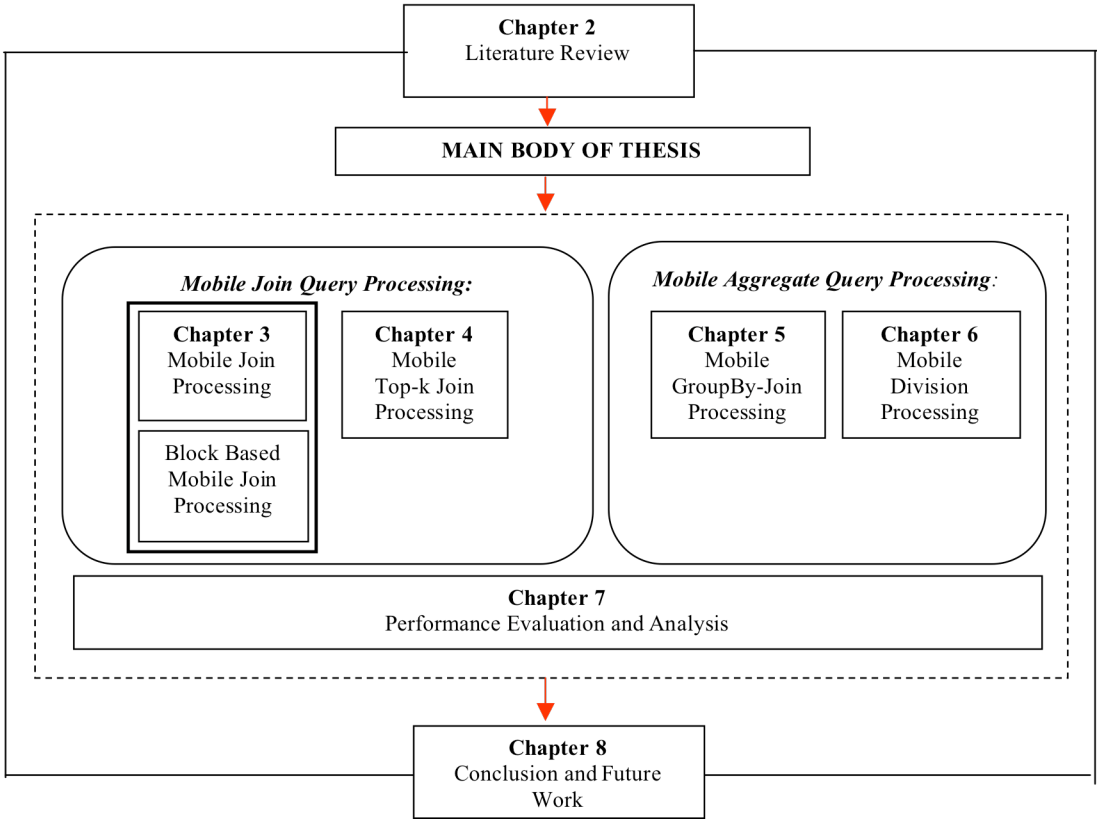


Figure 1.5: Thesis Organization

Chapter 2 describes the literature review on existing related work in connection especially with the field of mobile technology and mobile queries. The major aim of the literature review is to acquire an idea of what has been done by other

researchers and gain an insight into the contribution of this thesis in the context of other existing work. It outlines the achievements of the conventional methods and other researchers' work in the same domain and analyses the pros and cons of these methods. More importantly, it highlights the problems which remain outstanding.

The main body of this thesis, which addresses the problems pointed out in Chapter 2, is divided into four major parts*:* (*i*) *Mobile Join Based Processing,* (*ii*) *Top-k Join Processing,* (*iii*) *Mobile GroupBy-Join Processing,* and (*iv*) *Division Based Processing.* Each of these parts is explained in separate chapters.

Chapter 3 presents join query processing techniques that are divided into local processing which is on the mobile device side and remote processing which is on the server side. This chapter also performs a comparative analysis of the different versions of the proposed techniques based on the transfer cost of each technique in order to evaluate its cost effectiveness compared with the others.

Chapter 4 introduces techniques to overcome the issue of having a small display screen on the mobile device. Therefore, the results of interest are only the Top-k results which are join information from multiple sources and display only the most important information. These algorithms are divided into two stages where the first stage concentrates on obtaining the data from the servers, and the latter stage performs the local matching by adopting either the nested loop or the sort merge approach.

Chapter 5 is an extension of the previous chapter which introduces the concept of aggregation function. Additionally, join queries involving groupby functions are also addressed. In processing groupby-join queries, we focus on addressing the group by prior to join as it will minimize the overall processing time.

Chapter 6 focuses on universal quantification, in particular, division based query processing. Apart from addressing the relational division query, where the divisor forms a group, we extend it to incorporate multiple group divisors.

Chapter 7 covers the performance evaluation by formulating various cost models for each proposed technique to evaluate and analyse the efficiency of the

proposed algorithms. We develop analytical models and carry out experimentation analysis of the results.

Chapter 8 presents the conclusion of this thesis. It provides the summary of the results achieved, outlines its contributions, and provides an insight into future issues that are worthy of further investigation.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents an extensive review covering related work in the area of mobile query processing. It not only provides an overall summary of mobile technology, but also examines what other researchers have been exploring in the general area of mobile query processing and traditional database processing. The main aim of this chapter is therefore to provide a broad understanding of the work that has preceded and is related to this thesis.

This chapter is organized as follows. Section 2.2 gives the background of mobile technology together with a framework for mobile query processing architecture covering server, on-air and client strategies. Section 2.3 provides a mobile query taxonomy incorporating multiple non-collaborative servers. Next, Section 2.4 presents the join query processing including traditional join and Top-k join query processing, followed by a list of the current outstanding problems. Section 2.5 looks further into other types of aggregate query processing including Groupby-Join and division. A list of existing problems is also provided. Section 2.6 recaps with a list or research questions and problem definitions which will be addressed in this thesis. Finally, Section 2.7 gives the conclusion. The entire structure of Chapter 2 can be seen in Figure 2.1.

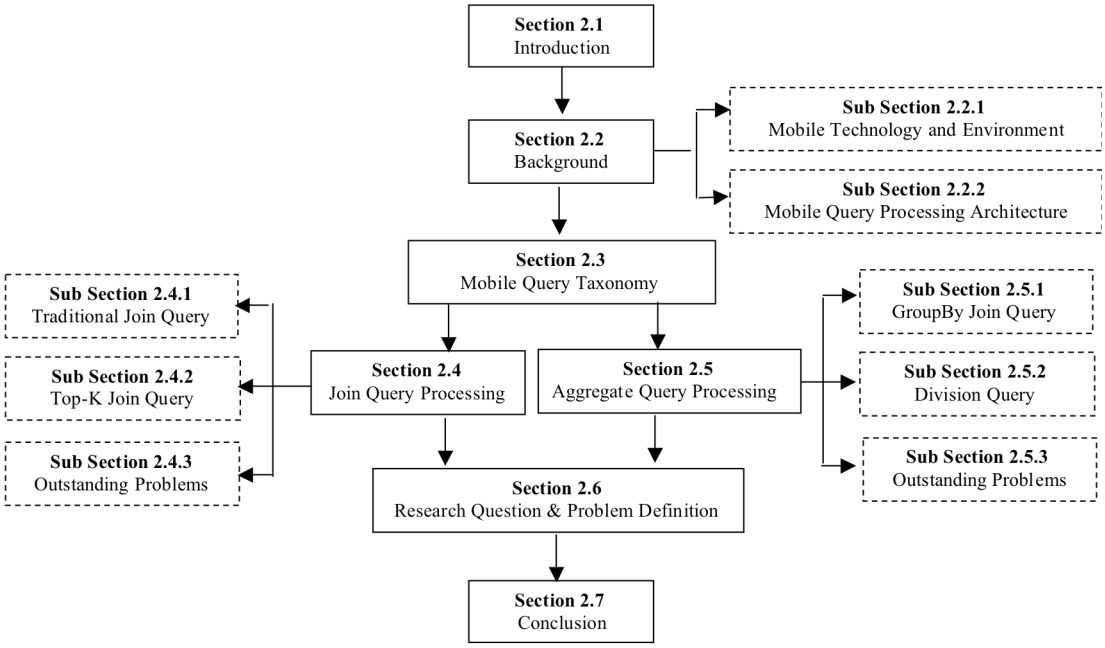


Figure 2.1: Structure of Chapter 2

2.2 Background: A Mobile Database Environment

As a preliminary to the thesis, we provide a discussion of the general background of a mobile database environment including basic information about a mobile environment. The subsequent sub-section is an investigation of existing work on the framework of mobile query architecture.

2.2.1 Mobile Technology and Environment

Generally, mobile devices are defined as electronic equipment which operate without cables for the purposes of communication, data processing and exchange, which can be carried by their users and can receive, send or transmit information anywhere, anytime due to their mobility and portability (Bose et al., 2005). In particular, mobile devices include mobile phones, Personal Digital Assistants (PDAs), laptops that can be connected to a network, and PDA-mobile phones that add mobile phone functionalities to a PDA (Waluyo, Goh, Taniar and Srinivasan, 2005).

Mobile users, with their mobile devices and servers that store data, are involved in a typical mobile environment (Wolfson et al., 2006). Each of these mobile

users communicates with a single or multiple servers that may be independent of one another. However, communication between mobile users and servers is required in order to carry out any transactions and information retrieval. Basically, the servers are normally static, whereas mobile users can move from one place to another and are therefore dynamic. Nevertheless, mobile users have to be within a specific region so as to be able to receive signals in order to connect to the servers (Waluyo, Srinivasan and Taniar, 2004b).

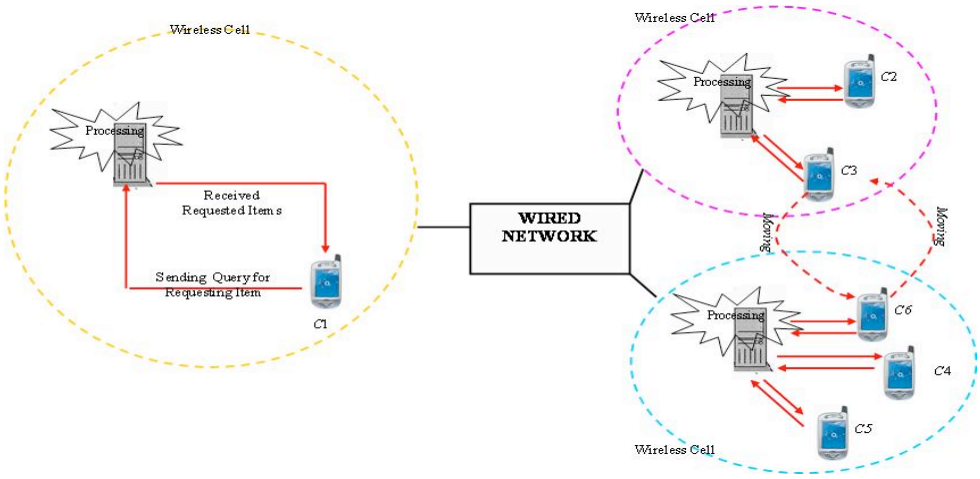


Figure 2.2: A Mobile Database Environment

Figure 2.2 illustrates a scenario of a mobile database environment which involves mobile users moving from one location to another location. Whenever mobile users are within a specific region or cell, they can access information provided by the servers within that region. And the mobile users will obtain the desired data and downloaded into their mobile device. When they move to a different location, the information that they access may have been changed due to the change of the region. So in a new region, mobile users can download other data. Assuming is *C*3 currently in Location *B* and is accessing *S*2. Once he receives the desired data, he moves to Location *C* and now accesses *S*3 which provides different data. Having these two individual data sources, the mobile device can manipulate the data locally.

Example 2.1: A property investor, while driving his car, downloads a list of nearby apartments for sale from a real-estate agent. As he moves, he downloads the

requested information again from the same real-estate agent. Because his position has changed since he made the first inquiry, the two lists of apartments for sale would be different due to the relative location when this investor was requesting information. Based on these two lists, the investor would probably like to perform an operation on his mobile device to show only those apartments that exist in the latest list, and not in the first list. This kind of list operation is often known as a 'difference' or 'minus' or 'exclude' operation, and this is incurred due to information which is location-dependent and is very much relevant in a mobile environment.

Hence, in a typical mobile environment, it is common to encounter situations where a mobile user is currently obtaining data from a current active region, but the information acquired is still incomplete. This leads to the need for further processing involving other data that is obtained from different servers.

2.2.2 A Framework of Mobile Query Architecture

Even though some of these existing works can solve some of the issues that arise from this project, there are still several limitations which are presented in the following sections. Recent related work done by others in the field of mobile database queries include query processing via (*a*) *server strategy*, (*b*) *on-air strategy* and (*c*) *client strategy* (Acharya, Kumar and Yang, 2007; Bose et al., 2005; Chan, Si and Leong, 1998; Chang and Yang, 2002; Waluyo, Srinivasan and Taniar, 2005; Wolfson et al., 2006). Figure 2.3 depicts the architecture of the available strategies of query processing in a mobile environment.

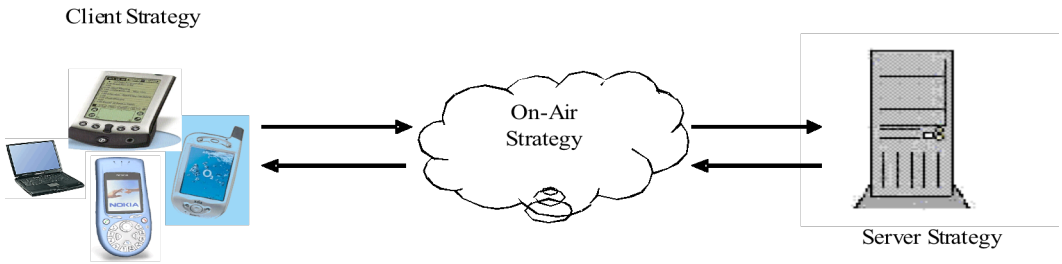


Figure 2.3: A Mobile Query Architecture

In general, the *server strategy* refers to mobile users sending a query to the server for processing and then the results being returned to the user (Waluyo, Srinivasan, and Taniar, 2004b; Wolfson et al., 2006). Issues, such as location dependency, should be taken into account since different locations will be accessing different servers, and subsequently it impacts on the processing by the server and the return of the results based on the new location of the mobile user (Pissinou, Makki and Campbell, 1999). Our approach differs from this strategy in the sense that we focus on how to process the already downloaded data in a mobile device and manipulate the data locally to return satisfactory results, taking into account the limitations of mobile devices.

The *on-air strategy* which is also known as the *broadcasting strategy* is basically where the server broadcasts data to the air, and mobile users tune into a channel to download the necessary data (Waluyo, Srinivasan and Taniar, 2003; 2004c). This broadcasting technique broadcasts a set of database items to the air to a large number of mobile users over a single channel or multiple channels (Datta et al., 1999; Huang and Chen, 2004; Waluyo, Srinivasan, and Taniar, 2004a). This strategy strongly addresses the problem of channel distortion and fault transmission. With the set of data on the air, mobile users can tune into one or more channels to get the data. This, subsequently, improves query performance. This also differs from our approach in the sense that our focus is not on the way that the mobile users download the data in terms of whether it is downloaded from data on the air or downloaded from data in the server, but rather on how we process the downloaded data locally in mobile devices.

The *client strategy* is where the mobile user downloads multiple lists of data from the server and processes them locally on their mobile device (Lo et al., 2004; Ozakar, Morvan, and Hameurlein, 2005). This strategy deals with processing locally in the mobile devices itself, such as when data are downloaded from remote databases and need to be processed to return a join result. Downloading both independent relations entirely may not be a good method due to the limitations of mobile devices which have limited memory space to hold large volume of data and small display screens which limit the visualization (Lo et al., 2004). Thus, efficient space

management of output contents has to be taken into account. In addition, this strategy also relates to maintaining cached data in the local storage, since efficient cache management is critical in mobile query processing (Cao, 2003; Elmagarmid et al., 2003; Xu et al., 2003; Zheng, Xu and Lee, 2002). This approach is similar to our work in terms of processing data that are downloaded from remote databases locally and readily for further processing.

Each of the above query processing strategies has been studied in relation to the mobile environment and will be discussed in more detail in the following sections.

(*a*) *Server Strategy*

In general, the *server strategy* refers to mobile users sending a query to the server for processing and then the results being returned to the mobile user (Lee and Chen, 2001; Waluyo, Srinivasan, and Taniar, 2004b). The problem with this strategy is promulgated by any disconnection that occurs to the mobile user especially during the transmission of the query; the sudden disconnection may result in loss of information. There are several related issues that have been investigated in the server strategy related work. However in this thesis, we discuss only two main issues involving data placements and data scheduling. By determining how data will be placed on the server disk, this will affect the query processing in terms of having frequently accessed data being able to be retrieved more quickly thereby improving query response time.

Also, deciding which data should be given priority is important in relation to the server strategy. This aspect regarding query scheduling, which concerns issues of finding how the query is scheduled, is important as is the issue of finding how to process the query in an efficient manner. Possible ways to process the query being issued by the mobile user can occur either on the server side or client side depending on the processing side which will incur a lower transfer cost as well as less memory consumption.

Issues, such as location dependency, are taken into account since different locations will be accessing different servers, and subsequently this affects the processing by the server and the query results being returned based on the new

location of the mobile user (Kottkamp and Zukunft, 1998). Various techniques have been developed to update the new location movement of the mobile user. The time function method provides an estimation of the location of mobile users at different times (Cao, Wang and Li, 2003). The limitation of this technique is that it avoids excessive location updates due to no explicit update being necessary. By indexing the locations, we can predict the future movement of the mobile users (Chen, Wu and Yu, 2003; Hung and Leu, 2003).

Our approach differs from the existing strategies in the sense that we focus on how to process the already downloaded data on a mobile device and manipulate the data locally, taking into account the limitations of mobile devices. We are also dealing with multiple non-collaborative servers, whereas others (Cao, Wang and Li, 2003; Chen, Wu and Yu, 2003; Hung and Leu, 2003; Kottkamp and Zukunft, 1998) deal with optimization of query processing in a single server.

(*b*) *On-Air Strategy*

The *on-air strategy* which is also known as the *broadcasting strategy* is basically where the server broadcasts data to the air and mobile users tune into a channel to download the necessary data (Peng and Chen, 2003; Waluyo, Srinivasan, and Taniar, 2004a). This broadcasting technique broadcasts a set of database items to the air to a large number of mobile users over a single channel or multiple channels and are related to activities that take place on-air (Chang and Chiu, 2002; Huang, Chen and Peng, 2003). This strategy strongly addresses the problem of channel distortion and fault transmission. With the set of data on the air, mobile users can tune into one or more channels to get the desired data items. However, unexpected situations may arise such as when a mobile user may not have enough memory to cache the desired data items, or s/he may experience a lengthy tune-in time for the desired data. Several issues in the existing related work for on-air strategy that have been investigated include data organization, data selection and data indexing. All these take into account optimizing the capacity, response time as well as bandwidth usage.

The question of the organization of data items is related to matching the order of the broadcast data with the order of data required by the query (Huang and Chen,

2004). The basic idea is to be able to allocate related data items in such a way that a mobile user does not have to wait for a substantial amount of time for the desired data items. This helps to reduce waiting and downloading time. Related work addresses this issue by examining the query access patterns and semantics of the queries that are being issued by the user (Lee, Leong and Si, 2002). However, a complex cost model has to be used in order to decide the best organization of data. And this raises another important issue that needs to be addressed, especially when a large number of database items are to be broadcast.

Also, by looking at recent access, the contents and organization of data broadcast can be determined (Waluyo, Srinivasan, and Taniar, 2004c). However, this technique appears not to be taking into account the handling of a request that has been around for quite some time. It only reduces the probability of occurrence, but if it occurs, then it is not taken into consideration. Also, whenever there is too many data in the broadcast cycle, a decision has to be made regarding prioritization.

One of the core issues is determining the priorities of the data items to be broadcast. This refers to which data items should be broadcast in the next period (Huang and Chen, 2004). Several scheduling algorithms have been proposed which include '*First Come First Served*' which sequences the data items according to their requested time and this demonstrates that any access request would receive a response after waiting a substantial amount of time (Chang and Chiu, 2002). Although there is no endless waiting, this algorithm has the disadvantage of having low average performance because it takes into account only the requested time and ignores the difference in access frequency of various data items. Another proposed algorithm includes *'Most Request First*' which prioritizes those data items with most requests (Datta et al., 1999). However, the shortcoming of this algorithm is that those data items that have few requests would always be lined up behind the data items that have more frequent requests, and therefore those less frequent request items would have an endless waiting period. An improved algorithm from the same related work would be '*Long Wait First*' which gives priority to the data items that have the longest waiting time (Acharya, Kumar and Yang, 2007). This algorithm considers both the number of requests as well as the waiting time so as to reduce the occurrence of endless waiting.

A selection mechanism is designed to reduce the broadcast cycle length which can reduce the query response time. During each broadcast cycle, the data items can be qualified as either hot or cold data items. Hot data items are data items that are accessed by most mobile users; conversely, cold data items are those that are less in demand. It is often important to replace the cold data items with the new hot data items which are believed to be more in demand. Based on several existing works regarding the selection of data items, several replacement algorithms that deal with replacing cold data items with hot data items have been investigated. The proposed algorithms namely 'Mean', 'Window' and 'Exponentially Weighted Moving Average' maintain a score for each data item in order to estimate the access probability (Lee, Leong and Si, 2002). The scores are obtained by measuring the cumulative access frequencies.

Another issue that has been investigated in relation to the on-air strategy would be data indexing. Data indexing is believed to lower the tuning time by providing information for the mobile user to tune into the broadcast channel at an appropriate time when the desired data items arrive (Lee, Leong and Si, 2002). The use of indexing helps mobile users to search desired data items by determining when the data arrives, thereby reducing query processing time which benefits mobile users, as well as utilizing power more efficiently (Cao, 2002a). Along with the broadcast data, some form of directory, which is known as the index, is attached. The information consists of the exact time of the data to be broadcast. And thus, while waiting for the desired data items to arrive, the mobile users can switch to "doze" mode and switch back to "active" mode when the desired data items arrived. Indicating the time that the indexed data record will be broadcast is one of the current approaches in the existing related work (Chen, Wu and Yu, 2003).

However, there are several trade-offs in this current approach. If the index is broadcast sparsely, the client might miss the index records in his first attempt and would have to keep tuning until the desired index record or the real data record are obtained. This increases the tuning time tremendously. On the other hand, broadcasting an index too frequently will increase the size of the broadcast data and thus leads to increase duration of the broadcast cycle which eventually leads to higher

database access time. However, if we eliminate the index completely, although it will yield minimal access time, the mobile user would have to listen to every single broadcast data item until the desired one is obtained. Therefore, when designing the index directory, several concerns including finding the optimal balance between tuning time and response time must be taken into account, as both will be greatly affected due to the occupancy of the index in the broadcast cycle.

One limitation of existing broadcasting mechanisms is their inability to efficiently recover from faults induced by unreliable wireless transmission which forces mobile users to wait for the next broadcast cycle if the required data item or index records is damaged during transmission. Other limitations of this existing on-air strategy involves their main concern in using a single broadcast channel and trying to limit the broadcast data by selecting the data items that are more in demand.

This on-air strategy differs from our approach because we focus on taking data broadcast into consideration when gathering information from multiple non-collaborative servers, and not on how the mobile users download the data in terms of whether it is downloaded from data on the air or downloaded from data in the server. Rather, we concentrate on the way that the downloaded data is processed locally in mobile devices if broadcast data are involved.

(*c*) *Client Strategy*

The *client strategy* relates to data caching in mobile databases which allows mobile users to obtain as high a computing speed as the server by involving a much smaller volume of data items. It also maintains cached data in the local storage since efficient cache management is critical in mobile query processing (Cao, 2003; Elmagarmid et al, 2003; Xu et al, 2003; Zheng, Xu and Lee, 2002). It deals with the question of how mobile users maintain and manipulate the data in its local cache in an efficient manner. The advantages are amplified since each mobile user is likely to initiate queries frequently within a short time span. On most occasions, with the inherent characteristics of a mobile environment that suffers from narrow bandwidth and frequent disconnection, caching the frequently accessed data in the local cache will aid in enhancing the performance, especially data availability. Most related work on

client strategies, discusses in particular, issues relating to caching replacement, granularity and coherency or invalidation (Cao, 2003; Chuang and Hsu, 2004; Elmagarmid et al., 2003; Hu and Lee, 1998).

Due to limited memory capacity, cache replacement needs investigation. The replacement policy discards old cache data items that are no longer relevant or are out of date and replaces them with the newly obtained data items (Cheng, Xu, and Lee, 2002). The issue that needs to be addressed with respect to this policy is to determine which data items are no longer needed and need to be replaced. This has to be addressed carefully because if a bad replacement policy is being used, then it may result in waste of energy as well as memory space since the mobile user may not be able to use the cached data but will still have to send a query to obtain the desired data items. Thus, the effectiveness of a caching replacement will affect the performance of the queries: if an effective cache replacement is used, the processing of a query will be much better, and this enhanced performance will also allow a greater number of cold queries to be served, especially during a disconnection situation. Most of the replacement policies that were investigated involve utilizing access probability as the primary factor when determining which data items are to be replaced.

For caching in respect to location dependency, the distance between the mobile user's current location and that of the cache data needs to be considered and often it is associated with semantic caching. Semantic caching stores semantic descriptions and associated answers for the previously issued queries. Due to the rapid movement of mobile users as well as the location parameter, by having semantic descriptions of the previous queries, the performance of future queries will, supposedly, be enhanced (Lee, Leong, and Si, 2002).

Cache granularity relates to determining the physical form of cached data items. It appears to be one of the key issues in cache management systems. There are three different level of caching granularities in object-oriented databases which include (*a*) attribute caching, (*b*) object caching and (*c*) hybrid caching (Chan, Si, and Leong, 1998). Attribute caching refers to frequently accessed attributes that are stored in the client's local storage. As for object caching, instead of the attribute itself being cached, the object is cached. However, attribute caching creates undesirable

overheads due to the large number of independent cache attributes. Thus, hybrid caching appears to be a better approach, since it takes advantage of both granularities.

Cache coherence, also known as invalidation strategy, involves cache invalidation and update schemes to invalidate and update out-dated or non-valid cached items (Chan, Si and Leong, 1998; Cao, 2003). After a certain period, a cached data may no longer be valid and therefore mobile users should obtain a newer cache before retrieving the data (Tan, 2001). There are several techniques that have been proposed to overcome this issue. These include (*a*) stateful server, (*b*) stateless server, (Barbara and Imielinski, 1994) and (*c*) leases file aching mechanism (Lee, Leong and Si, 2002). A stateful server refers to the server having an obligation to its clients; that is, it is responsible for notifying the users about changes, if there are any. In contrast, a stateless server refers to the server not being aware of its clients. Therefore, the server broadcasts a report which contains the updated item either asynchronously or synchronously. The leases files mechanism, also known as the lazy invalidation approach, assigns each mobile user the responsibility for invalidating its cached items. The main concern of this strategy lies is in determining the refresh time for the cached data.

Currently, the client strategy focuses mainly on traditional queries and is not applicable to a wireless communication environment, especially when this involves more complex queries such as location dependent queries that can be either a series of continuous queries or on-demand queries. Thus, the main drawback of the client strategy relates to the new nomadic queries. Our proposal differs from the client caching strategy, since our focus is on assembling information from various data sources. Therefore, our research will develop strategies to obtain and process data efficiently when gathering data from various multiple non-collaborative sources.

2.3 Taxonomy of Mobile Query Processing Incorporating Multiple Non-collaborative Servers

One of the reasons for presenting the taxonomy of mobile query processing is to be able to provide a framework for the different types of database operations to be used in various types of queries. It also indicates the need for database operations as a result of the limitation that is inherent in a mobile environment. We group database operations in mobile devices into two main classifications: (*a*) *Mobile Join Query Processing* and (*b) Mobile Aggregate Query Processing.*

The mobile join query processing is further divided into the following types:

- Join Query Processing
- Top-K Join Query Processing

Join Query Processing is basically the process of combining data from one relation with another relation (Elmasri and Navathe, 2007). In the context of a mobile environment, joins are used to bring together information that is stored in two or more different, independent servers or remote databases (Lo et al, 2004). It joins multiple data from different servers into a single output to be displayed by the mobile device.

Example 2.2: A Japanese tourist while traveling to Malaysia wants to know the available vegetarian restaurants in Malaysia. He looks for restaurants recommended by both the Malaysian Tourist Office and Malaysian Vegetarian Community. First, using his wireless PDA, he needs to download information broadcast from the Malaysian Tourist Office. Then, he downloads the information provided by the second organization mentioned above. Once he has obtained the two lists from the two information providers, he would like to join the contents from the two relations that may not be collaborative to each other.

An important issue to be investigated is that the mobile device has limited visualization capability. This recommends the use of *Top-k query processing*, where we display only the top *k* query results, such as the top-10 ranked of the query results

(Xin et al, 2006). In other words, we are not interested in displaying or even processing the complete query results. With the small screen limitation, it is almost impossible to display everything that has been obtained from multiple sources on one single screen. And often mobile users are not interested in everything, but rather, highly important information only. And this is generally known as Top-k queries.

Example 2.3: A movie enthusiast is interested to know which movies are worth watching which are listed as the Top 10 movies based on reviews from movies servers in two different countries, namely United States and Australia. The mobile user is interested in knowing which movies will be in the Top 10 list on average based on the data obtained from the two different movies servers of the United States and Australia.

The above examples illustrate an on-mobile join case. It shows the importance of assembling information obtained from various independent sources in a mobile device. It shows how a join operation is needed to be performed on a mobile device as the mobile user downloads information from two different sources which are not collaborative, and wants to assemble information through a join operation on his mobile device as well as taking into account the limitation of the small display screen of the mobile device.

The mobile aggregate query processing is further divided into types of queries as follows:

- Groupby-Join Query Processing
- Division Query Processing

Group-By Join Query Processing is common in conjunction with the aggregate function, which involves multiple collections of data from different places (Taniar et al., 2004). However, this is also relevant to the mobile environment when a query processing involves gathering information from multiple non-collaborative servers to be joined and produces a single table which will become an input to the groupby operation. Thus, this type of query involves join and aggregate functions.

Example 2.4: A movie enthusiast is interested in obtaining the number of movies that have a particular actor. Assuming there are two servers that the user would need to query. One server has a list of all movies (e.g. Movie table) and the other has a list of all artists (e.g. Actor table). Assuming that these two servers are independent, the user therefore, would need to group by each artist from the Movie table so that a count for all the movies by each artist can be obtained. And this information needs to be joined together with the Actor table so that additional information for that particular artist can be obtained.

Division Query Processing is a powerful concept for querying databases, which enables a database system to evaluate complex "for-all" predicates (Date, 1995). This is also known as the universal quantification query and has been neglected in past research even though the power of universal quantification and relational division enables the analysis of many-to-many relationships and set valued attributes (Graefe and Cole, 1995). The same goes for queries within the mobile environment. This type of division query processing appears to be useful and appropriate in certain circumstances and therefore is vital to explore its potential use in the mobile environment.

Example 2.5: The human resource department wants to recruit new staff for a certain new job vacancy. In the meantime, the HR will have a list of criteria that need to be met by the candidates applying for job. Therefore, there will be two non-collaborative servers, where one server contains all the details about the applicants, and another server contains only the skills criteria for that advertised job. The HR needs to find which potential candidates possess all the skills listed, so that they can be short-listed for the vacancy. Any applicants who do not meet one or more of the skills criteria will not qualify as a match. In another words, the company is looking for only those applicants who possess all the required skills.

The above examples illustrate the mobile aggregate case. It shows the need for displaying only certain information via the mobile device by using the aggregate function. The aggregate operation serves to summarize data from multiple sources to take into account the limitation of memory and processing power as well as other limitations related to the nature of the mobile environment.

In summary, mobile join and mobile aggregate queries are among the two most commonly used types of queries in the database environment which can be applied to the mobile environment to address directly specific issues in the mobile environment such as low bandwidth, and in mobile devices such as small display screen and low battery life.

2.4 Mobile Join Query Processing

This section is divided according to the various types of mobile join query processing including the traditional join and Top-k join results query processing. We discuss existing techniques for traditional join and Top-k including the related outstanding problems that we will address in this thesis.

2.4.1 Traditional Join Query Processing

Due to the nature of relational databases where information is split into multiple tables, it appears that there arises the need for information to be assembled together through gathering the data from multiple tables (Elmasri and Navathe, 2007). This is done via the join operations. Thus, join operation is one of the most common operations in relational databases. Join operation is also considered one of the most expensive operations in relational database processing as it is a binary operation taking two tables for processing. Very often the tables contain lots of records to be joined together and therefore sensible solutions and algorithms are needed for overcoming the complexity of join operations.

Generally, a join operation is performed so as to link two tables based on the nominated attribute – one from each table. The link is created because of the equality of the values from the two designated attributes. Because of this equality element, these types of join queries are called *Equi-Join* queries (Date, 1995). Figure 2.4 illustrates a join query between two tables: table *R* and table *S* based on attribute *attr*2 of table *R* and attribute *attr*1 of table *S*. The results are the matched records from the two tables.

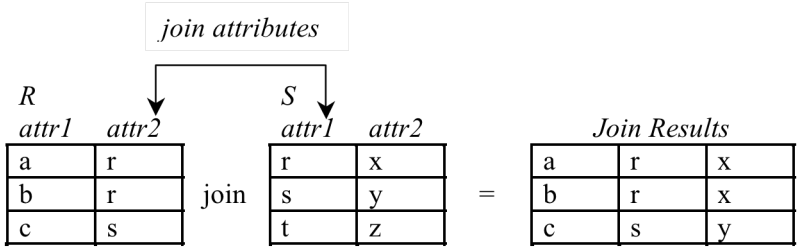


Figure 2.4: Join operation

A typical equi-join query would be a join between two tables through a link from the primary key (*PK*) of one table with the foreign key (*FK*) of the other table. This link between two tables is often necessary to assemble information, since the desired information is often split during the design as a result of the normalization process.

An example of a *PK-FK* equi-join is "to retrieve each student together with his/her enrolment details". This query joins table Enrolment and table Student. The attribute that links both tables Student and Enrolment, is attribute Sid that exists in both tables. The SQL of this query is shown below.

```
Select *
From STUDENT S, ENROLMENT E
Where S.Sid = E.Sid;
```

Joining can also be performed on two unrelated tables (Date, 1995). The join operation is often required by users, even though there is no relationship between these two tables. The join must, however, be also based on a designated attribute of both tables. In order words, the relationship between the two tables is not established through the design, but through the join query.

An example of a non PK-FK equi-join query is "to retrieve pairs of lecturers and courses from the same department". The query joins table Lecturer and table Course, and the join attribute is attribute *Ldept* of Lecturer and *Cdept* of Course. In our database schema, both tables Lecturer and Course are not directly related, apart from a non-direct relationship through table Enrolment. However, in this query, we explicitly join the two tables. The SQL statement for the above query is as follows.

```
Select *
From LECTURER L, COURSE C
Where L.Ldept = C.Cdept;
```

An example of two PKs equi-join is "to retrieve lecturers who are also students". This can be obtained by joining tables Student and Lecturer on their PKs. Assume the value of person ID (e.g. student ID and lecturer ID) is unique in the university, and therefore a lecturer who already has an ID when enrolling in a program (e.g. graduate program) keeps that ID. The SQL statement for this query is as follows:

```
Select *
From LECTURER L, STUDENT S
Where L.Lid = S.Sid;
```

From the query point of view, there is no difference among these three equi-join queries, as all of these queries join two tables and the join predicates are clearly defined. However, when processing these equi-join queries, the difference is escalated due to the presence of an index for every PK.

In order to study mobile join algorithms, firstly it is necessary to understand the traditional join algorithms that have been widely used. There have been many join algorithms proposed in the literature which include the following (Elmasri and Navathe, 2007; Ramakrishnan and Gehrke, 2000):

(*a*) Nested-loop join algorithm,

(*b*) Sort-merge join algorithm, and

(*c*) Hash-based join algorithms

Each of the above algorithms will be studied in more detail. The sample data in Figure 2.5 will be used for illustrative purposes. For simplicity, the two tables are named table *R* and table *S*, each of which has two attributes: one is an alphabetical attribute and the other is a numerical attribute. The join attribute is the numerical attribute. Table *R* has 15 records, whereas table *S* has 9 records.

Table R

Adam	8
Bobby	22
Cindy	16
Dave	23
Edward	11
Frankie	25
Greg	3
Harry	17
Irene	14
Jo	2
Keith	6
Linda	20
Mona	1
Nano	5
Ogie	19

Table S

Archaeology	8
Business	15
Computing	2
Dance	12
Education	7
Finance	21
Grammar	10
Health	11
IT	18

Figure 2.5: Sample data

(*a*) *Nested Loop Join Algorithm*

Nested-loop join is the simplest form of join algorithm, where for each record of the first table, it goes through all records of the second table (Elmasri and Navathe, 2007, Taniar et al, 2008). This is repeated for all records of the first table. It is called a nested-loop because it consists of two levels of loops: *inner loop* (looping for the second table) and *outer loop* (looping for the first table).

Using the sample data in Figure 2.5, firstly, take the first record of table *R* (record **A**dam) and then go through record by record of table *S* to see whether there is a match based on the join attribute which in this case is the numerical attribute. In this case, it is lucky that the match is found in the first record of table *S* (record Archaeology). In other words, the match of Adam from table R matches the value of Archaeology from table *S* with the join attribute value of 8.

Assuming that the join attribute is based on unique attributes, once a match has been found in table *S*, there is no need to continue searching for other matches for Adam. The same process is repeated for the second record of table *R* (record Bobby). In this case, the worst scenario happens, since none of the records in table *S* matches Bobby. This means record Bobby containing join attribute of 22 does not appear anywhere at all in table *S*. This process continues until all records from table *R* have

been processed. The end results using the sample data would finally produce three matches as in Figure 2.6. The nested-loop join algorithm can be summarized in Figure 2.7.

Join Results

Adam	8	Archaeology
Edward	11	Health
Jo	2	Computing

Figure 2.6: Results of Nested Loop

Algorithm 2.1: Nested-Loop Join

```
1.  For each record of Table R
2.     Read record from table S
3.     For each record of table S
4.        Read record from table S
5.        Compare the join attributes
6.        If matched Then
7.           Store records in Query Results
8.        End If
9.     End For
10. End For
```

Figure 2.7: Nested-Loop Join Algorithm

In terms of efficiency, note that in general every record in table *S* has to be visited (read) as many times as there are records in table *R*. Assuming that there are *N* records from table *R*, and *M* records from table *S*, the efficiency of the nested-loop algorithm is known to be $O(N*M)$. Although the algorithm is very simple, it is definitely not efficient, because of repeated I/O scans to one of the tables (Elmasri and Navathe, 2007; Ramakrishnan and Gehrke, 2000; Taniar et al., 2008).

(*b*) *Sort-Merge Join Algorithm*

Sort-Merge join is based on sorting and merging operations (Elmasri and Navathe, 2007). The first step of joining is to sort the two tables based on the joining attribute in an ascending order which in this example is the numerical attribute. The second step is to merge the two sorted tables. If the value of the first joining attribute is

smaller than the other, it skips to the next value of the first joining attribute. It skips to the next value of the second joining attribute, if it is the opposite. When the two values match, the two corresponding records are merged, and placed into the query result. This process continues until one of the tables runs out of records.

Using the sample data from Figure 2.5 where the numerical attributes are not sorted, it will now be sorted as in Figure 2.8. The sorting is based on the join attributes which are the numerical attributes from these two tables.

Table R

Mona	1
Jo	2
Greg	3
Nano	5
Keith	6
Adam	8
Edward	11
Irene	14
Cindy	16
Harry	17
Ogie	19
Linda	20
Bobby	22
Dave	23
Frankie	25

Table S

Computing	2
Education	7
Archaeology	8
Grammar	10
Health	11
Dance	12
Business	15
IT	18
Finance	21

Figure 2.8: Sorted tables

Join Results

Jo	2	Computing
Adam	8	Archaeology
Edward	11	Health

Figure 2.9: Results of Sort Merge

Once the two tables have been sorted based on the join attributes, the merging process starts. The first step is to take the first record of table *R* (record Mona) and

compare it with the first record of table *S* (record Computing) to see whether the two numerical attribute values are the same. In this case, we have the value Mona from table *R* is 1 and the value Computing from Table *S* is 2. We observe that it is not the same. Therefore, the record that has a smaller value has to move on. In this case, we need to move on to the second record of table *R* (record Jo) because value of 1 is smaller than 2. Then compare it again with the same record of table *S* (record Computing). Since they are matched, put them into the result. If the join attribute values are unique, when a match is found, then the next process is to take the next records from both tables and ignore the previous records that have been matched which have been sent to the join results. This means that, after records Jo and Computing are matched, we take record Greg from table *R* and record Education from table *S*. The process is then repeated until one of the tables has exhausted all the records. Figure 2.9 shows that the final results of the join using the sort merge join algorithm.

Sort-merge join algorithm is also quite simple. The main task is basically sorting which is done first to the two tables. The merging itself is quite simple. Figure 2.10 summarizes the sort-merge join algorithm, where the join attributes are unique.

Algorithm 2.2: Sort-Merge Join

```
1.  Sort records of table R based on the Join Attribute
2.  Sort records of table S based on the Join Attribute
3.  Let i=1 and j=1
4.  Repeat
5.    Read record R(i)
6.    Read record S(j)
7.    If join attribute R(i) < join attribute S(j) Then
8.      i++
9.    Else If join attribute R(i) > join attributeS(j) Then
10.     j++
11.   Else
12.     Put record R(i) and S(j) into the Query Result
13.     i++; j++
14.   End If
15.   If either R(i) or S(j) is EOF Then
16.     Break
17.   End If
18. End Repeat
```

Figure 2.10: Sort-Merge Join algorithm

If the sort-merge join algorithm containing join attributes that are not unique then it needs to be revised slightly. In this case, there is a small nested-loop among duplicate values of the matched join attributes. For example, if there is another record after Jo which has the same number as Jo (e.g. Josh/2), and also there is another record after Computing which has the same number as Computing (e.g. CompEng/2), then there will be a small nested loop among these records to produce 4 records, namely Jo/2/Computing, Jo/2/CompEng, Josh/2/Computing, and Josh/2/CompEng.

In terms of its efficiency, the sort-merge join algorithm is better than the nested-loop join. Given N and M be the number of records in tables R and S, respectively, the sorting is assumed to use $O((NlogN) + (MlogM))$ for the two tables. As the merging has linear complexity based on $O(N+M)$, the overall complexity of the sort-merge join algorithm is hence $(O(NlogN) + (MlogM))+ O(N+M)$, which is considerably much better than the nested-loop of $O(N*M)$, especially if N and M are very large (Elmasri and Navathe, 2007; Ramakrishnan and Gehrke, 2000; Taniar et al., 2008).

(*c*) *Hash-Based Join Algorithm*

A number of hash-based join algorithms such as hybrid-hash, Grace hash join, have been proposed in the literature (Elmasri and Navathe, 2007; Silberschatz, Korth and Sudarshan, 2006; Taniar et al., 2008). A hash-based join is basically made up of two processes: *hashing* and *probing*. A hash table is created by hashing all records of the first table using a particular hash function. Records from the second table are also hashed using the hash function and probed. When any match is found, the two records are concatenated and placed in the query result.

A decision must be made about which table is to be hashed and which table is to be probed. Since a hash table has to be created, it would be better to choose the smaller table for hashing, and the larger table for probing. Using the sample data in Figure 2.5, the smaller table would be table *S* and thus it is chosen to be hashed.

Figure 2.11 illustrates how table *S* is hashed into a hashed table (Taniar et al., 2008). In this example, assume that the hash function used in the hashing process is done by summing the first and the second digit of the hashed attribute, which in this

case is the join attribute. The hashing is done record by record. First, record Archaelogy/8 is hashed to the hash table index entry 8, and then record Business/15 is hashed to hash index 6 by taking the 15 and summing up its first and second digits respectively (1+5) to the equivalent of 6.

In any hashing process, collision must be taken care of. For example, record Computing/2 is hashed to index entry 2 in the hash table, and record Health/11 is hashed to the same hash table index because the attribute 11 is actually (1+1=2). In this example, the collision is handled by creating additional entries on the same index entry. The sample in Figure 2.11 shows several collisions, as well as several empty index spots.

Upon completion of the hashing process, the next step is the probing stage that takes the other table, which in this case is table *R,* and hashes record by record using the same hashing function. If it is hashed to a non-empty index entry, then each record on that entry must be examined to see whether a match can be found.

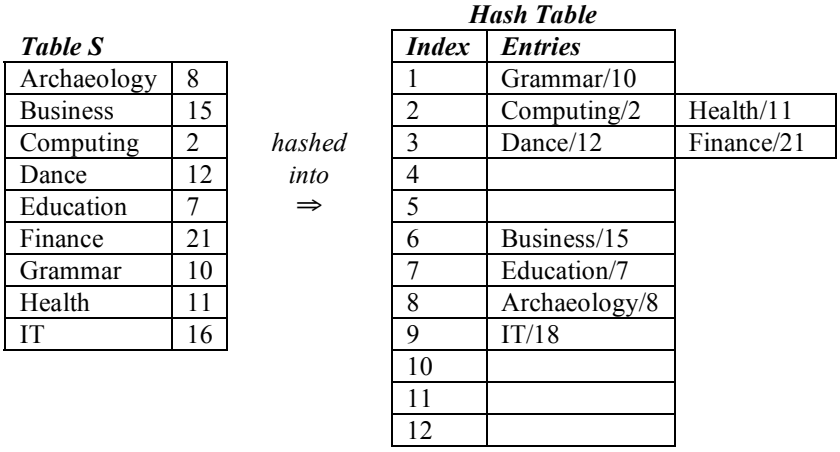


Figure 2.11: Hashing Table *S*

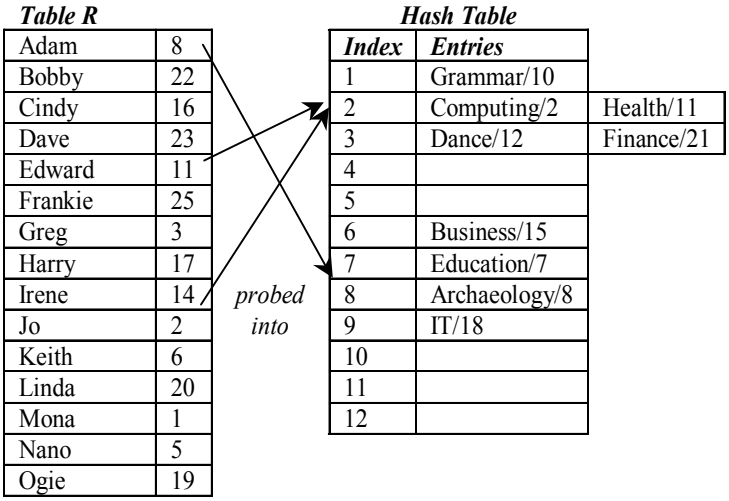


Figure 2.12: Probing Table *R*

Figure 2.12 shows the probing stage where record Adam/8 is hashed and probed into index entry 8, and the match is found with Archaeology/8 and it is a qualifying match since both the join attributes are matched. Another example is record Edward/11, which is hashed and probed into index entry 2. Since there are more than one record on that index entry that is computing/2 and health/11, a traversal of each of the records must be made and then examined to ascertain whether it is matched with Edward/11. In this case, Edward/11 found a match with Health/11. It is the same case with Jo/2, which found a match with Computing/2. The final result using the hashed based algorithm is depicted in Figure 2.13.

Join Results

Adam	8	Archaeology
Edward	11	Health
Jo	2	Computing

Figure 2.13: Results of Hash-Based Join

Other records in table *R* were hashed into either an empty index entry or hashed into a non-empty index entry but did not find a match. An example of the former is for record Bobby/22, which is hashed into index entry 4 which is empty. An

example of the latter is Cindy/16 hashed into index entry 7 but which did not find any qualifying match with the record in that index entry. Although it contains an entry in the same index, the join attribute does not match because Cindy has a join attribute of 16 but the existing value in index 7 is Education that holds the join attribute of 7.

Figure 2.14 summarizes an algorithm for hash-based join (assume table *S* is used for hashing, whereas table *R* for probing). Note from the hashing and probing process mentioned above, that each record from the above two tables are scanned only once. Consequently, there is no repeat scan as in the nested-loop and sort-merge join algorithms. Therefore, hash-based join, which is based on $O(M) + O(N)$, is considered much more efficient than both the nested-loop and sort-merge joins.

Algorithm 2.3: Hashed-Based Join

```
1.  Let H is a hash function
2.  For each record in table S
3.    Read a record from table S
3.    Hash record on join attribute using H into hash table
5.  End For
6.  For each record in table R
7.    Read a record from table R
8.    Hash record on join attribute using H into hash table
9.    Probe into the Hash table
10.   If an index entry is found Then
11.     Compare record on index entry with record of table S
12.     If matched Then
13.       Put the pair into the Query Result
14.     End If
15.   End If
16. End For
```

Figure 2.14: Hashed-Based join algorithm

Complexity Analysis

After examining all various existing join algorithms, the complexity of join algorithms is normally dependent on the number of times that a disk scan needs to be performed. Disk scan is considered the most expensive operation on computer systems, as it requires disk arm movement which is very slow in comparison with CPU operations. Therefore, the minimization of disk scan is the ultimate objective not only in join algorithms, but also in any query processing algorithms.

The complexity of the three join algorithms discussed above can be summarized as follows (Elmasri and Navathe, 2007; Taniar et al., 2008):

- Nested-loop join algorithm = $O(N*M)$
- Sort-merge join algorithm = $(O(NlogN+MlogM))+O(N+M)$
- Hashed-based join algorithm = $O(N+M)$

Note that N and M are table sizes. The hashed-based join algorithm is widely accepted as the most efficient join algorithm, as proven by the complexity listed above. To illustrate how well a hashed-based join algorithm operates, Figure 2.15 shows a comparison of the complexity of the three algorithms in various data sizes N and M which depict the number of records from two different tables R and S correspondingly.

From Figure 2.15, we note that the complexity of the nested loop grows exponentially. When N and M are large, $O(N*M)$ becomes very large. On the other hand, sort merge is not that bad compared with the nested loop. Nevertheless, the hash-based is still the best of all the three traditional join query algorithms. Therefore, based on this simplified comparison, it is clear and obvious that the hash-based join algorithm should be the ultimate choice.

N	*M*	*Nested Loop*	*Sort Merge*	*Hash*
10	10	100	40	20
100	100	10,000	600	200
1,000	1,000	1,000,000	8,000	2,000
10,000	10,000	100,000,000	100,000	20,000
100,000	100,000	10,000,000,000	1,200,000	200,000
1,000,000	1,000,000	1,000,000,000,000	14,000,000	2,000,000

Figure 2.15: Complexity comparison of the various join algorithms

2.4.2 Top-k Join Query Processing

Top-k queries deal with processing and displaying only *k* number of query results which are highly ranked (Anguili and Pizzuti, 2005; Chaudhuri and Gravano, 1999). Consequently, there is no need to display or even process the entire query results. Many existing works on Top-k queries are particularly relevant to domains such as web databases, P2P network, sensor network, data stream and mobile environment (Anguili and Pizzuti, 2004; Cao and Wang, 2004; Chaudhuri and Gravano, 2004; Deng et al, 2006; He et al., 2004; Hwang and Chang, 2005). Each of these will be further explored as follows.

Mobile issues such as low bandwidth and the low battery life of the mobile device is always a concern for mobile users. Very often it is frustrating to mobile users when the query capabilities do not meet the response requirements due to bandwidth limitation or the battery runs out before the results are obtained; this is always a major problem.

Top-k has made possible the earliest output of objects that are highly relevant, and this is especially useful in mobile environments because the available bandwidth can be used efficiently. Existing works such as the proposed algorithm of SR-Combine can meet the requirements of dealing with tough response times in a mobile environment (Metwally, Agrawal, and Abbadi, 2005; Michel, Triantafillou and Weikum, 2005). The proposed technique delivers some first results early which leads to less waiting time; this eventually impacts on both the access cost and processing power. Most importantly, this approach is self-adapting for retrieval and focuses on real time requirements; hence, it is able to adapt to various environment regardless of whether it is supported by a higher bandwidth or a faster network.

The P2P environment has been gaining more attention lately because of the increasing number of users interested in content sharing. This has made it even more crucial to restrict the number of results to minimize traffic among peers because more and more people will be flooding the network. Thus, by using the Top-k queries within the P2P network, only the most relevant results are returned. This is because

users are generally concerned only with results that are relevant to their query and want only the best ones to be returned.

Several Top-k query algorithms in P2P network have been introduced by researchers (He, Shu and Wang, 2004; Liu, Feng, and He, 2006; Matsunam, Terada, and Nishio, 2005). One of these researches has decentralized Top-k query evaluation by using local ranking, merging and optimized routing to minimize results being returned to the users. This approach can be successfully used to exploit the computing resources in the network because it requires ranking and merging of results.

Other existing works relate to the popular approach of evaluating Top-k query in a hierarchical way. All this needs is to have each peer perform a local Top-k query. Different peers obtain different Top-k results, and these are merged hierarchically from bottom to top. The merging approach is similar to the decentralized Top-k which efficiently handles traffic network (Akbarinia, Pacitti and Valduriez, 2006). In addition, this technique maintains a histogram for each different peer according to the top k results returned by the peers which are used to estimate a numerical score so that a query needs to be forwarded to only the most promising neighboring peers. This, consequently, improves search efficiency because the histogram is able to facilitate peer selection.

Another existing approach for Top-k query in P2P focuses on the fact that users need only those results that are important by reducing the reply traffic through controlling the number of query replies. The original straightforward Top-k method is one whereby each peer simply replies to a query; as a result of reducing query replies, the increasing network traffic is also relieved (Matsunami, Terada, and Nishio, 2005).

In addition, a few challenges have been identified when processing Top-k in a web application (Lee and Tsai, 2001; Xie et al., 2006) which include the possibility of the relation attributes not being available other than through external web-accessible form interfaces. Subsequently, this produces the need to query repeatedly for a potentially large set of data sets. However, work has been done to overcome the above challenges by proposing a technique that will process Top-k queries in a setting where the attributes for which users specify target values are handled by external, autonomous sources with a variety of access interfaces.

When relating to traditional database systems, the most straightforward way over relational database existing work is related to Top-k queries involves techniques of optimizing the queries using a score function whereby the scoring is done through a traditional SQL Order By clause. These include the introduction of a new rank-join algorithm that has been done by other researchers that combines join with the ranking algorithm (Ilyas, Aref, Elmagarmid, 2003) This work is closely related to the other work of supporting Top-k selection queries where the aim was to apply a scoring function on multiple attributes of the same relation to select attributes ranked on their combined score. Basically, the new rank-join algorithm makes use of the individual orders of data to produce join results ordered by a user-specific scoring function. The idea of this proposed technique is to rank the join results progressively during the join operation. With the implementation of individual ordering, the evaluation of Top-k queries can be improved by eliminating the need to sort the join results on the combined score.

Ranking aggregate data from multiple tables is also closely related to queries in a relational database where the Top-k query combines different rankings of the same set of objects from different tables, and returns the *k* items that consist of the highest total score based on some aggregate functions. One of the related works investigated a new algorithm that is designed to minimize the number of data sets that are to be accessed, and decrease processing cost as well as the memory usage when processing the Top-k queries. This algorithm, known as LARA, processed the Top-k queries by sequentially accessing sources of ranked data scores (Xin et al., 2006).

Data stream also appears to be an increasingly popular application but, due to the large size of the data that keeps flowing down, often it becomes a problem for users that may not have the full capability to handle the large, continuous flow of data. Thus, with the vast sizes and fast flow of data, some stream types should be processed so that data is not deleted forever. For instance, defining top k involving data streams is based on the frequent elements receiving 1% or more of the total hits which might constitute the top 100 elements. One of the proposed algorithms which is known as a space-saving algorithm, uses the main idea of maintaining partial information of interest. This algorithm is able to efficiently guarantee strict error

bounds for approximate counts of elements using limited space which can address the Top-k memory requirements.

Besides using Top-k queries in contexts such as web, relational database, mobile database, it also plays a critical role in the distributed sensor network where many decision making related applications may find it useful for purposes such as identifying interesting objects, network monitoring and etc. For instance, a related work of Top-k queries in the distributed sensor network with the objective of find the *k* highest ranked results for the user is known as the threshold join algorithm (Ilyas, Aref, Elmagarmid, 2004). This algorithm is designed for queries where the user has an ongoing interest in having the *k* most relevant results. For instance, a biologist analyzing a forest environment might be interested in changes and conditions over a long period of time, not on just a one-off occasion.

The proposed threshold join algorithm concentrates on minimizing it against the expensive communication medium. Thus, the algorithm uses a non-uniform threshold on the queried attribute in order to minimize the number of attributes that have to be transferred to the querying node. In addition, this algorithm is deployed for network aggregation to minimize the utilization of a network.

In summary, although much research has been undertaken in the area of Top-k queries, none explores Top-k queries in the context of multiple non-collaborative servers in a mobile environment. Therefore in this thesis, we will explore the existing works and integrate them into processing Top-k queries that will involve multiple non-collaborative servers, and processing such Top-k queries efficiently by overcoming, as much as possible, the challenging factors and limitations of the mobile environment. The major similarity among all the existing techniques is that they aim to satisfy user perception which is the relative importance of attributes and obtaining a set of results that satisfy the condition – that is, the most important or most relevant results.

2.4.3 Outstanding Problems of Join Query Processing in Mobile Environment

Based on the investigation, there are several limitations and issues which can be addressed so that proposals made in the existing works on traditional query processing can be improved and adapted, especially for the mobile environment. Thus, in this sub-section, we examine the existing problems arising from previous researchers' work in the following domains:

- Join Query Processing
- Top-k Query Processing

(*a*) *Problems of the Join Query Processing*

Upon exploring the existing domain of various traditional join queries that have been widely and commonly used in the database environment, one of the most obvious minor issues would be the improvement of these traditional join techniques so as to be able to apply them in mobile query processing involving multiple non-collaborative servers (Lo et al, 2004). These traditional join techniques include the popular use of the sort merge join and the hash join (Elmasri and Navathe, 2007).

Besides the aforementioned minor issue, a more major problem that has been identified and needs to be addressed is the limitation of one of the current traditional join techniques, namely the hash-based join technique. Our comparative analysis, that was given in the previous section that dealt with traditional join processing, indicates that the hash-based join is apparently the most efficient among all the other join techniques. It shows that hash-based join is better than sort merge join.

However, in a mobile environment, this may not appear to be the case. This is because the capacity of a mobile device is usually small and limited and hence a hash-based join in a mobile device might not always appear to be the best solution. Rather, a sort merge join may be more appropriate in certain circumstances in mobile a environment. This is because, in a typical mobile environment, the server-mobile architecture is like a client-server architecture; therefore, it might be desirable for

heavy processing such as sorting to be dedicated to the servers to be processed. With the already sorted results being returned to the mobile device, the local processing will be lighter.

As a result, in this thesis, we will investigate the use of a more appropriate join technique such as sort merge join, taking into account the limitation of the mobile devices. We will look at exploring this further in terms of the applicability of the sort merge join technique in a mobile environment incorporating multiple non-collaborative servers.

(*b*) *Problems of the Top-k Query Processing*

Top-k Queries have been explored for quite some time and have been popular in non-mobile environments, such as web databases, Peer to Peer (P2P) network, sensor network, data stream and relational databases (Kimelfel and Sagiv, 2006; Yiu et al., 2007). The application of Top-k was evident in most of the existing work that we investigated.

Since most researches are based only on non-mobile environments, the minor problem we are addressing would be the application of Top-k queries in a mobile environment involving multiple non-collaborative servers. This is extremely important and is highly motivated since very often mobile users who send a query for data in the mobile environment are interested in obtaining only the data that is greatly important to them and often they are not interested in obtaining full results. This is mainly due to the fact that the mobile device has a limited memory and a short battery life span; moreover, only a limited bandwidth is available in the mobile environment.

There is also a major, more serious problem that should be addressed. In the past, Top-k has often been associated with selection and/or aggregate functions and therefore has not received much attention in relation to the mobile environment and all its constraints. Furthermore, the existing work often treated Top-k as post-processing that is done after the full join results have been obtained. However, according to the traditional query optimization, we have been made aware of and warned that a join operation should always be delayed as much as possible. Therefore, the main challenge in addressing this major problem is how to incorporate

Top-k processing into the join by having the Top-k to be processed in advance before obtaining the full result of the join.

Thus, this thesis examines Top-k join queries in the mobile environment, especially in formulating efficient methods to process such queries. In addition, we also consider that some servers may already have sorted/ranked data. We will therefore address not only how Top-k queries are carried out in a mobile environment, but also investigate its efficiency when the data sources have already been individually ranked.

2.5 Aggregate Query Processing

This section is divided into various types of aggregate query processing which include Groupby-Join query processing and division query processing.

2.5.1 Group-By Join Query Processing

Groupby-Join queries are often used for strategic decision making due to the nature of group-by queries where raw information is grouped according to the designated groups and within each group aggregate functions are normally carried out. As the source information for these queries is commonly drawn from various tables, joining tables – together with grouping, become necessary. These types of queries are often known as “GroupBy-Join” queries (Taniar and Rahayu, 2006).

The data obtained from the different tables are joined to produce a single result that becomes an input to the group-by operation which basically involves join and group by. For simplicity of description and without loss of generality, we consider queries that involve only one aggregate function and a single join.

In most findings from the existing works, it appears that Groupby-Join query processing is found in the context of parallel query processing. Parallel query processing processes a query using a parallel machine; that is, a machine with multiple processors. There are many different kinds of parallel machines but the key idea is to use multiple processors to process a query so that the query elapsed time can

be faster. Scientifically, in strategic decision making, parallelization of GroupBy-Join queries is unavoidable in order to speed up query processing time.

There are two options of executing the queries as follows (Taniar et al, 2008):

- Group-by Before Join (group-by first and then join)
- Group-by After Join (join first and then group-by)

To illustrate these two types of GroupBy queries, we use the following tables from a Suppliers-Parts-Projects database (Date, 1995):

```
SUPPLIER (S#, Sname, Status, City)
PARTS (P#, Pname, Colour, Weight, Price, City)
PROJECT (J#, Jname, City, Budget)
SHIPMENT (S#, P#, J#, Qty)
```

(a) *GroupBy-Before-Join Query*

"*GroupBy-Before-Join*" query is where the join attribute is also one of the group-by attributes. Basically the group-by has to be performed first before the join. For example, the query to "retrieve project numbers, names, and total quantity of shipments for each project having the total shipments quantity of more than 1000" is shown by the following SQL:

```
Select PROJECT.JID, PROJECT.Jname, SUM(Qty)
From PROJECT, SHIPMENT
Where PROJECT.JID = SHIPMENT.JID
Group By PROJECT.JID, PROJECT.Jname
Having SUM(Qty)>1000
```

In the above query, basically we want to join the Project and Shipment tables together using `JID` as the join attribute which also appears to be the group-by attribute. When this happens, it is expected that the group-by operation be carried out first, and then the join operation. In processing this query, all `Project` records are grouped based on the `JID` attribute. The result is then joined with table `Shipment`.

The following descibes how group-by before join is processed by assuming there are three processors, each of which has some portion of table Project and table Shipment as in Figure 2.16

Processor 1		Processor 2		Processor 3	
Project.JID	**Shipment.JID**	**Project.JID**	**Shipment.JID**	**Project.JID**	**Shipment.JID**
P1	P2	P2	P1	P3	P1
P4	P2	P5	P2	P6	P3
P7	P3	P8	P4	P9	P3
	P4		P4		P3
	P5		P5		P4

Figure 2.16: Initial data placement in three processors

Step 1 is to group table Shipment based on the group by attribute which is Project.JID in each processor. The result of the grouping to obtain the count for each JID on the table Shipment will produce Figure 2.17. Table Project will contain no change at all. In other words, the Shipment table carries out a local group by and performs an aggregate function count.

Step 2 is to distribute the groups to processors according to the range allocated to each processor. This means that each processor can hold a certain range of JID. For instance, Processor 1 is allocated to accept only JID ranges from *P*1 to *P*3, followed by Processor 2 that accepts *P*4 to *P*6, and Processor 3 accepts *P*7 to *P*9. The distribution phase is done on both Project and Shipment tables. The temporary results of the distribution are depicted in Figure 2.18. The arrows show the movement of records. For clarity of presentation, not all arrows are displayed in the figure. Note that after distribution, in this example, in Processor 3, table Shipment does not have any records.

After distributing the Project ID to the right allocation of processors, we proceed to Step 3 where the joining will be done from each processor as in Figure 2.19. Note that Processor 3 does not produce any join result, simply because it does not have any records from one of the two tables. In the results, each pair shows a project id from table Project and the project id together with the count from table

Shipment. Hence, for example, in Processor 1, (*P*1,*P*1=2) indicates that *P*1 from table Project matches with *P*1 from table Shipment whose count equals 2.

Processor 1		Processor 2		Processor 3	
Project.JID	**Shipment.JID**	**Project.JID**	**Shipment.JID**	**Project.JID**	**Shipment.JID**
P1	P2 = 2	P2	P1 = 1	P3	P1 = 1
P4	P3 = 1	P5	P2 = 1	P6	P3 = 3
P7	P4 = 1	P8	P4 = 2	P9	P4 = 1
	P5 = 1		P5 = 1		

Figure 2.17: Group by and Aggregate count of Table Shipment

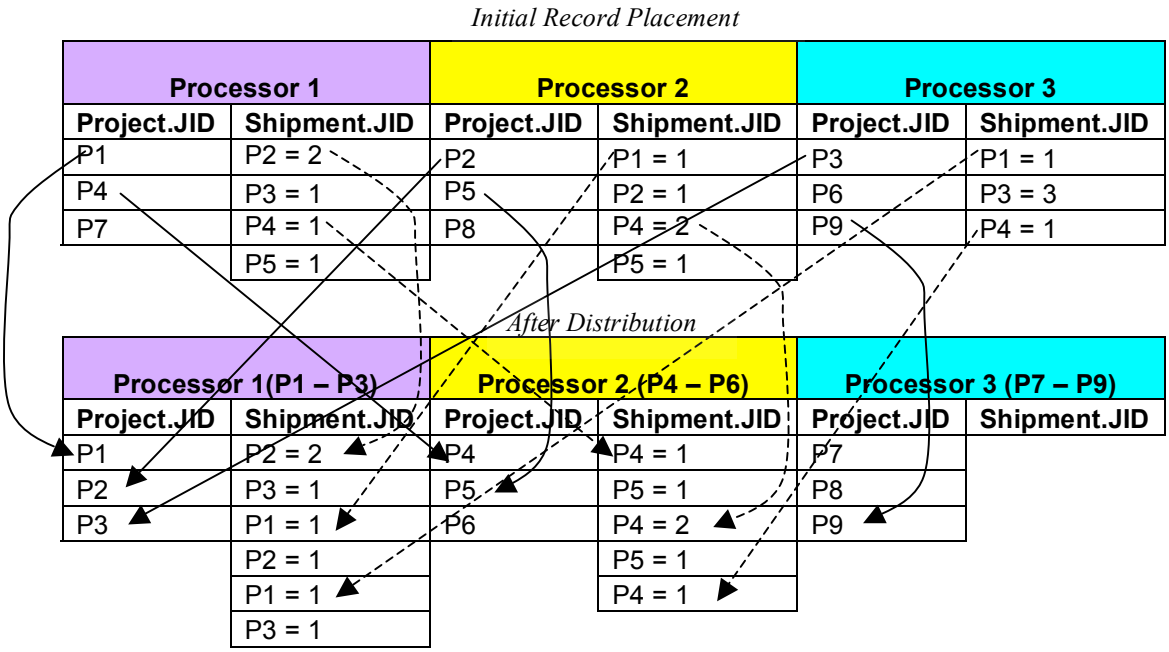


Figure 2.18: Redistribution of records from both Project and Shipment

Processor 1	Processor 2	Processor 3
P1, P1 = 1	P4, P4 = 1	
P1, P1 = 1	P4, P4 = 2	
P2, P2 = 2	P4, P4 = 1	
P2, P2 = 1	P5, P5 = 1	
P3, P3 = 1	P5, P5 = 1	
P3, P3 = 3		

Figure 2.19: Local join results in each processor

After performing the join in each processor, we will need to regroup them by adding up the count for each similar project ID so the results will be as in Figure 2.20. In other words, the final step is to perform a local group-by in each processor again. The final query results are the union of all local results from each processor. By looking at Figure 2.20, we can see that the query "to retrieve number of shipments of each project" produces 5 projects, together with the shipment count. Processor 1 produces three matches, Processor 2 two matches, and none for Processor 3.

Processor 1	Processor 2	Processor 3
P1, P1 = 2	P4, P4 = 4	
P2, P2 = 3	P5, P5 = 2	
P3, P3 = 4		

Figure 2.20: Final result of the query

In summary, Groupby-Before-Join queries are executed in three phases. In the first phase, the *local grouping* phase, each processor performs its group-by operation and calculates its local aggregate values on records of table *R*. In the second phase, the *distribution* phase, the results of local aggregates from each processor, together with records of table *S*, are distributed to all processors according to a partitioning function. Lastly, the third phase, known as the *final join and grouping* phase, two operations are carried out - in particular, the final aggregate or grouping of *R*, and joining it with *S*. The final grouping can be carried out by merging all temporary results obtained in each processor.

Communication costs incurred with this scheme are reduced because data from each processor are aggregated/grouped before being distributed. However, because data from table S in each processor are all distributed during the second phase, this may cause problems because it can be inefficient, particularly if *S* is very large.

(*b*) *GroupBy-After-Join Query*

"*GroupBy-After-Join*" query is when the join has to be done first and only then is it followed by the groupby (Taniar et al, 2004). For example: "group the part shipment by their city locations and select the cities with average quantity of shipment between 500 and 1000". The query written in SQL is as follows.

```
Select PARTS.City, AVG(Qty)
From PARTS, SHIPMENT
Where PARTS.PID = SHIPMENT.PID
Group By PARTS.City
Having AVG(Qty)>500 AND AVG(Qty)<1000
```

The main difference between the groupby before join and groupby after join lies in the join attributes and group-by attributes. In groupby after join, the join attribute is totally different from the group-by attribute. This difference is an especially critical factor in processing GroupBy-Join queries, as there are decisions to be made about which operation should be performed first: the group-by or the join operation. When the join attribute and the group-by attribute are different, there will be no choice but to invoke the join operation first, and then the group-by operation.

In the existing works, the authors argue that it is impossible to do a groupby first when the join attribute is different from the groupby attribute. In such a circumstance, it is compulsory to perform the join first. Consequently, the cost of processing may appear to be higher since it is well known that join processing is often expensive and should be delayed whenever possible. But in this situation, when the attributes of the join and groupby differ, there is no other option.

Using the above SQL, the following will demonstrate the processing of groupby after join. Figure 2.21 shows the records for the Parts and Shipment tables in each separate Processor 1 to Processor 3.

Based on the data in Figure 2.21 above, the first step is to distribute all records from both the Parts and Shipment tables based on the partitioning range function whereby we partition each processor to be allocated to certain range of acceptance like how we did previously in the GroupBy-Before-Join. For instance, Processor 1 accepts only parts *a*1 to *a*3, followed by Processor 2 which accepts parts from *a*4 to

*a*6, and lastly Processor 3 accepts parts from *a*7 to *a*9. Figure 2.22 shows the data placement after the distribution phase. Note in this example that, after distribution, Processor 3 does not have any Shipment records.

Processor 1		Processor 2		Processor 3	
Parts	**Shipment**	**Parts**	**Shipment**	**Parts**	**Shipment**
a1.Syd	a2	a2.Adel	a1	a3.Adel	a1
a4.Melb	a2	a5.Adel	a2	a6.Melb	a3
a7.Syd	a3	a8.Melb	a4	a9.Syd	a3
	a4		a4		a3
	a5		a5		a4

Figure 2.21: Initial data placement in three processors

Initial Placement

Processor 1		Processor 2		Processor 3	
Parts	**Shipment**	**Parts**	**Shipment**	**Parts**	**Shipment**
a1.Syd	a2	a2.Adel	a1	a3.Adel	a1
a4.Melb	a2	a5.Adel	a2	a6.Melb	a3
a7.Syd	a3	a8.Melb	a4	a9.Syd	a3
	a4		a4		a3
	a5		a5		a4

After Distribution

Processor 1 (a1 to a3)		Processor 2 (a4 to a6)		Processor 3 (a7 to a9)	
Part	**Shipment**	**Part**	**Shipment**	**Part**	**Shipment**
a1.Syd	a2	a4.Melb	a4	a7.Syd	
a2.Adel	a2	a5.Adel	a5	a8.Melb	
a3.Adel	a3	a6.Melb	a4	a9.Syd	
	a1		a4		
	a2		a5		
	a1		a4		
	a3				
	a3				
	a3				

Figure 2.22: Data placement after distribution

Next in step 2, based on the table after the distribution phase, we perform a local join for the Part and the Shipment based on the same Part ID for each processor.

Therefore, the result of the join for each processor would be as follows in Figure 2.23. Because Processor 3 does not have any Shipment records; consequently, the local join result is nil in that processor. Each match in the local join results in each processor is shown in pairs of parts. For example, the first match in processor 1 is (*a*1.Syd, *a*1) indicating part *a*1 (located in the city of Sydney) in table Part matches with part a1 in table Shipment. The name of the city where the part is located is needed because later the query needs to produce the query results which are city (or location) based.

After obtaining the join between the Part and Shipment table, the groupby using the groupby attribute which is City in this case will be done for each processor separately. This will return the count for each city in each processor as in Figure 2.24. In this example, all Sydney parts in the processor will be grouped together and a new count is produced, which in this case is 2, as there are 2 Sydney parts. Whereas, there are 7 Adelaide parts in Processor 1.

Next, in Step 3, using the results obtained from Figure 2.24 above, we need to redistribute the result of the groupby for each processor according to the partitioning range function such as Processor 1 accepts only those cities whose names begin with letters *A* to *G*, Processor 2 accepts only those cities starting with letters from *H* to *M* and lastly, Processor 3 accepts cities beginning with letters *N* to *Z*. The result of this redistribution is shown in Figure 2.25.

The final step is to perform the groupby function in each processor to obtain the final output as in Figure 2.26. Because City ‘Adel’ in Processor 1 occurs twice, we will need to get a final count for that particular city so we need to sum them to get the count as ‘9’ in City ‘Adel’. Basically, the last step is to get the final, total count for each city in each processor.

Processor 1	Processor 2	Processor 3
a1.Syd, a1	a4.Melb, a4	
a1.Syd, a1	a4.Melb, a4	
a2.Adel, a2	a4.Melb, a4	
a2.Adel, a2	a4.Melb, a4	
a2.Adel, a2	a5.Adel, a5	
a3.Adel, a3	a5.Adel, a5	
a3.Adel, a3		
a3.Adel, a3		
a3.Adel, a3		

Figure 2.23: Local join results in each processor

Processor 1 Count (City)	Processor 2 Count (City)	Processor 3 Count (City)
Syd, 2	Melb, 4	
Adel, 7	Adel, 2	

Figure 2.24: Local group by in each processor

Processor 1 (A to G)	Processor 2 (H to M)	Processor 3 (N to Z)
Adel, 7	Melb, 4	Syd, 2
Adel, 2		

Figure 2.25: After data redistribution based on the City (Location)

Processor 1	Processor 2	Processor 3
Adel, 9	Melb, 4	Syd, 2

Figure 2.26: Final query results

In summary, the Groupby-After-Join queries are executed in three phases. In the first phase, the *distribution* phase, each processor distributes the data to other processors according to a certain distribution function. In the second phase, the *local join* phase, each processor carries out a local join processing. Also in this phase, group by is carried out in each processor. In the third phase, the *redistribution* phase, the results of the groupby are redistributed among all processors according to some groupby distribution function. And finally, the last phase, the *groupby* phase, is where each processor regroups the groupby data which have been redistributed during the third phase. The final query results are the amalgamation of all results obtained from each processor.

2.5.2 Division Query Processing

Division query processing is often known as the universal quantification and is not supported directly in most traditional database systems (Graefe and Cole, 1995). Despite this concept being a powerful means of querying sets and databases that adds significant power to the query processing and inference capabilities, the universal quantification algorithms have been omitted and have not been explored for large databases. Basically, it is a query request to find an attribute in one relation that satisfies all of a given list of criteria from another relation. Figure 2.27 illustrates a division query between two tables: table *R* and table *S* based on attribute *attr*2 of table *R* and attribute *attr*1 of table *S* whereby there exists a record in the relation matching tables *R* and *S*.

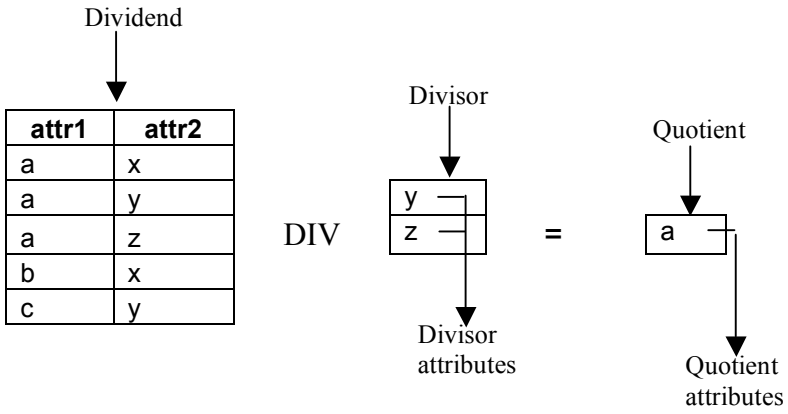


Figure 2.27: Relational division operation

An example of the division query would be a scenario that requires one to "list students that have done *ALL* subjects". The word 'ALL' here refers to the universal quantification. And in the query example, the projection of transcript is the **dividend** and the projection of subjects is the **divisor**. The division result which will return the matching students is called the **quotient** (Graefe and Cole, 1995). In relational algebra, this query is as follows:

$$\pi_{\text{Student-Id, subjectID}}(\text{Transcript}) \div \pi_{\text{subject-ID}}(\text{Subject})$$

Figure 2.28 shows the table for the above relational algebra.

Student	Subjects
Jack	DB
Jill	DP
Jill	Intro DB
Jill	Intro Graphic
Jill	Reading DB
Joe	DP
Joe	Intro Comp
Joe	Intro DB

÷

Subjects
DP
Intro DB
Reading DB

Figure 2.28: Sample data for the division query

Thus, in order to process queries such as the division queries, several methods have been proposed including (*i*) *Naïve Sort-Based Algorithm*, and (*ii*) *Division by Aggregation.*

(*a*) *Naïve Sort-Based Algorithm*

The first algorithm which is the naïve approach directly implements the calculus predicate. Step 1 of this approach is to sort the dividend by first sorting the quotient attributes followed by the divisor attributes. This means that the Student-ID is first sorted and then within each Student-ID, the relevant subject-ID is sorted. Figure 2.29 shows the temporary results of the first step after sorting.

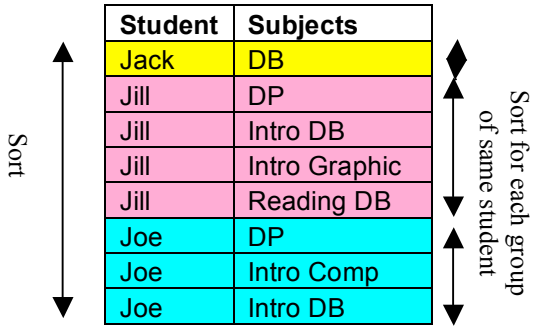


Figure 2.29: Sort the Dividend table

Step 2 would be sorting all the attributes of the divisor which results in the following Figure 2.30.

Subjects
DP
Intro DB
Reading DB

Figure 2.30: Sort the Divisor table

The last step which is Step 3 is to scan the sorted relations and merge the dividend and the division using merge join and nested loop. For instance there are 3 groups in the dividend which is student 'Jack', 'Jill', and 'Joe'. Each of these students has taken different subjects. Therefore, 'Jack' is first merged with the divisor to check whether the subjects he has taken match all the subjects in the divisor. If it does not, then the next dividend group which is 'Jill' will be executed and will merge with the divisor and re-matched for qualifying results. Since 'Jill' has taken all subjects that existed in the divisor, it will be returned as a qualifying match that is the quotient. And the next group is 'Joe' which will be merged with the divisor again for comparison. The final results using the naïve sort-based algorithm are shown in Figure 2.31.

Thus, the dividend is read only once throughout the process that demonstrates the merge join approach, whereas the divisor is read once for each group in the dividend that demonstrates the nested loop approach (Graefe and Cole, 1995).

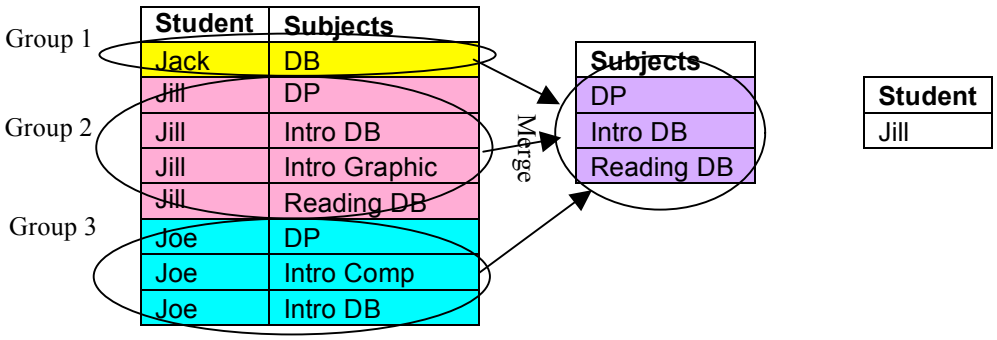


Figure 2.31: Merging process

(*b*) *Division by Aggregation Algorithm*

The second algorithm known as division by aggregation is an alternative algorithm for division queries. Since the naïve division algorithm requires sorting both input and repeated scans of the divisor, it appears to be rather inefficient and slow especially in a situation which involves a large input.

By using the same example as that above which is "to list students who have done *ALL* subjects", the following will demonstrate the method of solving the above queries with the alternative algorithm of using aggregations in play.

The first step is to count the number of subjects for each student in the dividend using the aggregate function and then to do the same thing for the divisor which means counting the total number of subjects in the divisor. This produces the results shown in Figure 2.32

Student	Count(Subjects)
Jack	1
Jill	4
Joe	3

Count(Subjects)
3

Figure 2.32: Count the Divisor attribute

By having the count for each student in the dividend and the count of the divisor, this eliminates those students who do not have enough number of subjects as in the divisor.

From Figure 2.32 above, it shows that 'Jack' has only 1 count subject and the number of subjects in the divisor is 3; therefore, it is eliminated because it will never be possible that he has taken all subjects. Thus, the process is being carried out only to those counts in the dividend that are equal to, or higher than, the count in the divisor. Consequently, this reduces unnecessary processing and will only execute those that are possibly a qualifying match. And for the remaining potential matches, the rest of the process would then be the same which is to check for the subjects between the dividend and divisor.

2.5.3 Outstanding Problems of Aggregate Query Processing in a Mobile Environment

Literature reviews have been carried out of existing work on other aggregate query processing techniques such as Groupby-Join and division query, and several limitations and issues that are currently, or have been previously, faced can certainly be improved in several ways. Thus, in this sub-section, we examine the existing problems from previous researches in Groupby-Join query processing and division query processing.

(a) ***Problems of the GroupBy Query Processing***

After analyzing the existing work on groupby query processing, we concluded that one of the minor problems that arises concerns the application of the Groupby-Join processing to a mobile environment that incorporates multiple non-collaborative servers.

Besides the abovementioned minor problem arising from the traditional groupby approach, there are several major problems as well, a key point of which is the following; if the join attribute and groupby attribute are different, there is no choice but to perform the join processing first. However, it may not be the case in certain circumstances and in this thesis, we will demonstrate that it is possible to do a

groupby function first and delay the join processing so as to produce better performance. This is because, based on the rule of the relational query optimization, it is always more efficient if the join process is performed last.

The next major problem is the question of how to apply the groupby/aggregate technique in the join even though the query does not have a groupby intention. In other words, this means looking at how to make use of the groupby technique in a normal equi-join. To date, we have not yet encountered an existing work that addresses this issue. Thus, our research in this thesis will look partly at improving the join processing technique in the mobile environment that is proposed in Chapter 3 to incorporate the groupby function. This means that, although the query is not a Groupby-Join query which means the query itself does not have a groupby, when it comes to processing such a query, a groupby technique can be used to improve performance.

As a result, our aim in this thesis is to attempt to solve the aforementioned minor and major problems that have arisen from the existing work. The minor problem will be addressed by extending the traditional method of groupby so that it can be used in a mobile environment that incorporates multiple non-collaborative servers; in order to address the major problems, we investigate ways of improving the performance of existing groupby techniques and extending their applicability to normal equi-join.

(*b*) *Problems of the Division Query Processing*

Similarly to the other traditional query processing techniques, the minor problem for the division query processing that arises from the existing work and that needs to be tackled, is the issue of how to apply division processing to mobile query processing involving multiple non-collaborative servers. This problem arises because to date all work in this area has been based on database tables that are related to one another.

The next problem is a more serious one and should be tackled. The examples given by other researchers show that the existing division algorithms do not consider situations where the divisor consists of several groupings.

For instance, our query is "to list those students who have done ALL subjects in a particular course". It is not possible to process this using traditional division algorithms. This can be demonstrated as follows in Figure 2.33.

Thus, our aim in this thesis is to attempt to solve the two problems listed: firstly, how to apply existing techniques to the mobile environment so that processing can be done within multiple non-collaborative servers. Secondly, the other problem to be specifically tackled in this thesis is the issue of traditional division being able to deal with only a single group divisor and not multiple group divisors.

Student	Subjects		Subjects	Course
Adam	English		English	Arts
Betty	Computing		History	Arts
Betty	History		English	Arts
Betty	Multimedia		Foreign Language	Arts
Carrie	English		Computing	IT
Carrie	Foreign Language		Multimedia	IT

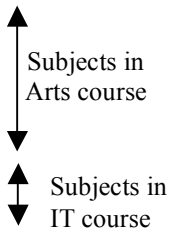


Figure 2.33: Multiple Groups Division

2.6 Research Questions and Problem Definitions

The research questions to be addressed in this thesis include the following. These questions form the framework of the research work presented in this thesis. These questions are also clustered into a number of topics, each of which is discussed in the subsequent chapters. These topics include: (*i*) mobile join, (*ii*) mobile Top-k, (*iii*) mobile groupby-join, and (*iv*) mobile division.

- *Mobile Join Query Processing*
  - How can join query processing techniques be applied to a mobile environment, where multiple non-collaborative servers provide the mobile client with the requested data?
  - Since mobile capacity is limited, how can block-based processing be used in a mobile environment?

- In a mobile environment incorporating multiple non-collaborative servers, it is likely that the processing may be done in a mobile client or in a server. How can the right balance be achieved between the two processing mechanisms in order to produce optimum efficiency, in terms of cost effectiveness and processing performance?

o Mobile Top-k Join Query Processing

- How can Top-k join query processing be processed in a mobile setting taking into account that different servers may have maintained the ranking of the required data?
- How can Top-k join algorithms be proven to be correct, especially during the process where the desired Top-k results seem to be obtained?
- What is the underlying difference between traditional merging and Top-k join merging processing?

o *Mobile Groupby-Join Query Processing*

- How can aggregate grouping be used in a join query to enhance mobile join query processing performance? The focus is still on the join processing, because the query is a join processing. However, the main question is how mobile join query processing may be enhanced through the grouping of records.
- In a multiple non-collaborative server setting, apart from joining different information sources, it is sometimes necessary to perform aggregate functions. In the traditional sense, the order of processing, that is, whether join or aggregate is carried out first, is very much determined by the type of the groupby-join query and whether the join attribute is the same as the groupby attribute. Therefore, the question is how groupby-join query processing can be used in a mobile environment.

- In the traditional sense of processing groupby-join queries, if the join attribute is different from the groupby attribute, the join operation must be carried out first. The main challenge is to investigate whether this phenomenon can be reversed, so that the groupby operation can still be performed before the join, in order to increase the performance.

- Mobile Division Query Processing
  - How can traditional relational division be applied to a mobile processing, where the mobile processing focuses mainly on count-based and sort-based data?
  - In the case of multiple group division query processing which is formulated in this thesis, how can multiple group division query be efficiently processed using sort-merge and aggregate-based processing?
  - How can pruning in multiple group division query processing be performed in order to minimize communication costs as well as processing times?

2.7 Conclusion

Since the existing related works are mainly concerned with using one strategy at a time to perform database queries on a single server, it is fundamentally important to have the option of using multiple different strategies together to perform information processing from multiple servers. Therefore, this research is significant, since it addresses information gathering through query processing by mobile users that span several non-collaborative servers. And in the real mobile environment, not all servers are able to accept direct queries. There may be occasions when the servers do data broadcast. And if this is the case, in order to retrieve and process the data that are obtained individually from different types of servers together, there is the need for advanced techniques that are able to process the data obtained from multiple sources together.

Based on our investigation of existing work, we intend to concentrate on using different strategies, such as via server or on air to download data, and devise a means of performing join queries locally on mobile devices taking into account the limitations of mobile devices. However, our approach focuses on using a combination of various possible join queries that are to be carried out locally to solve the major issues such as the limited memory and limited screen space of mobile devices. Although many traditional join query processing algorithms exist, they have their limitations since most of them take into account only the tables that are obtained from related servers. Therefore, when it comes to querying from non-collaborative servers, these techniques are not efficient.

The difference between our proposed research and the existing related work is that our work focuses mainly on being able to adapt to the mobile environment using techniques that combine various possible strategies that exist in the mobile query processing environment. We will also address some of the major issues of mobile devices and propose mobile join algorithms that are to be used in conjunction with multiple non-collaborative servers.

Chapter 3

Mobile Join Query Processing

3.1 Introduction

This chapter presents a collection of proposed techniques dealing with mobile queries to uniquely process mobile queries efficiently. The main objective is to minimize the data exchange and transfer cost between mobile devices and remote servers which is the sending query and the returning results of the query. We divide the proposed techniques into two main groups, namely mobile device side processing and server side processing. These two basic techniques are the foundation for more advanced techniques known as block-based processing.

Figure 3.1 illustrates the structure of this chapter. Section 3.2 describes the basic processing techniques which consist of mobile device side processing (MDSP) and server side processing (SSP) and is further extended into incorporating block-based processing (BBP) (refer to Section 3.3). At the end of each section, walkthrough examples will be demonstrated. Lastly, section 3.4 gives the conclusion and sums up the contribution.

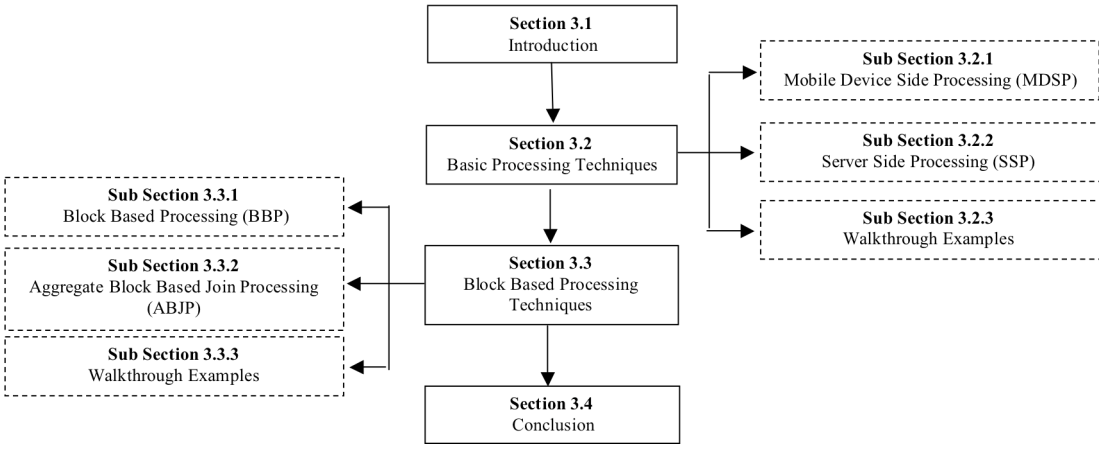


Figure 3.1: Structure of Chapter 3

3.2 Basic Processing Techniques

The proposed techniques consist of two basic techniques, namely: (*i*) mobile device side processing (MDSP), and (*ii*) server side processing (SSP). A walkthrough example of the proposed techniques will be presented at the end of this section.

3.2.1 Mobile Device Side Processing (MDSP)

In general, mobile device side processing deals with acquiring and then transferring information from the servers to the mobile device to be processed locally on the mobile device. Hence, the mobile device will process locally when determining the matching items. Location-dependent processing on mobile devices of information gathered from multiple non-collaborative sources commonly involves traditional set operations, such as intersection, difference, union, etc (Elmasri and Navathe, 2007). Therefore, the lists of information obtained from these sources are generally *similar* in terms of the semantics (such as two lists of restaurants), but may be slightly or vastly different, in terms of the contents (e.g. the two restaurant lists contain collections of restaurants in two different places or of two different kinds).

Depending on how the data is initially obtained, we divide *MDSP* into three different versions – we call them *MDSP*1, *MDSP*2, and *MDSP*3 for short.

(*a*) *MDSP*1

*MDSP*1 downloads data from both servers and then performs local processing on the mobile device. Therefore, *MDSP*1 downloads all attributes including primary and non-primary key attributes directly from respective servers (e.g. Location *A* and *B*) to the mobile device.

The mobile device will carry out the local processing which joins the downloaded data from the two servers. This version of local processing on the mobile device appears to be the simplest way available since it gets everything from both places into the mobile device directly. Also, it obviously consumes most main memory.

*MDSP*1 includes the following steps:

Step 1: Download every attribute including primary and non primary key from the server in location *A* to mobile device. The mobile device might invoke the following SQL query: `Select * from TblInLocA order by PK1;`

Step 2: Download each attribute including primary and non-primary key from server in location *B* to mobile device. Similar SQL may be used to request data from location *B*, such as: `Select * from TblInLocB order by PK2;`

Step 3: Qualification matches determined for the data obtained from steps 1 and 2 are processed on the mobile device itself. As the mobile device may not have a database engine due to various reasons, such as limited memory and processing capabilities, local processing may have to be programmed manually. However, this can be easily achieved using traditional merging techniques, since the two lists have been pre-sorted when obtained from the two respective servers.

The pseudocode of *MDSP*1 algorithm is shown in Figure 3.2.

Algorithm 3.1: MDSP1

```
// Step 1:
1. Send Query-1 to Server-1:
   Select * Into R
   From Table_R
   Order By PK;
// Step 2:
2. Send Query-2 to Server-2
   Select * Into S
   From Table_S
   Order By PK
// Step 3: On-mobile merging
3. i=1; j=1;
4. Let Query result Qr = {}
5. Do
6.    If i is EOF(R) or j is EOF(S) Then
7.       Exit Loop
8.    End If
9.    If Ri.PK = Sj.PK Then
10.      Qr = Ri = Sj;
11.      i++; j++;
12.   Else
13.      If Ri.PK > Sj.PK Then
14.         j++
15.      Else
16.        i++
17.      End If
18.   End If
19.End Do
```

Figure 3.2: MDSP1 Algorithm

(*b*) *MDSP*2

*MDSP*2 downloads everything from one server, but only the primary key from the other. This version is very similar to the previous one, except that only the primary key (PK) is downloaded from one server. After the matching process has been completed locally in the mobile device, in many cases, there is no need to download the full details of the matching records, since the details have been downloaded from the one server. However, in other cases, the matching records may optionally obtain the full details from the second server, as the second server initially gave the PK only.

*MDSP*2 includes the following steps:

Step 1: Download all attributes including primary and non primary keys from the server in location *A* to the mobile device. The query to be used in requesting this information is identical to that of MDSP1.

Step 2: Download only the primary key from the server in location *B* to the mobile device. The query to be used in requesting a list of PK from the server in location *B* is as follows: `Select PK2 from TblInLocB order by 1;`

Algorithm 3.2: MDSP2

```
// Step 1:
1. Send Query-1 to Server-1:
   Select * Into R
   From Table_R
   Order BY PK;
// Step 2:
2. Send Query-2(PK) to Server-2:
   Select PK Into S
   From Table_S;
// Step 3: On-mobile merging
3. i=1; j=1;
4. Let Query result Qr = {}
5. Do
6.    If i is EOF(R) or j is EOF(S) Then
7.       Exit Loop
8.    End If
9.    If Ri.PK = Sj.PK Then
      // Step 3a
10.      If s.other_attributes needed for Qr Then
11.         Send Query-3 to Server-2:
            Select * into STemp
            From Table_S
            Where Ri.PK  = Sj.PK;
12.      End If
13.      Qr = Ri = STemp
14.      i++; j++;
15.   End If
16.   If Ri.PK > Sj.PK Then
17.      j++
18.   Else
19.      i++
20.   End If
21.End Do
```

Figure 3.3: MDSP2 Algorithm

Step 3: Similar process as for *MDSP*1 is conducted here on the mobile device. Once the matches have been produced, there is an optional step to perform depending on whether there is additional information for the matched records from the server in location *B*.

Step 3a: Optionally retrieve non-key attributes from the server in location *B* to the mobile device, using the following SQL: `Select * from TblInLocB where PK2 in (N…)`, where *N* is a list of matched PK produced in step 3 above.

The pseudocode of the *MDSP*2 algorithm is shown in the following Figure 3.3.

(*c*) *MDSP*3

*MDSP*3 is where only the primary keys from both servers are downloaded in the first instance, and then local processing is performed based on these primary keys which avoids unnecessary downloading of other attributes that may not be of interest to the users. There is a significant difference between *MDSP*3 and the previous two versions. Because only PK is obtained from both servers, after the local processing, the matching PK needs to get the full details from the respective servers. Therefore, there is a trade-off between a full download in the beginning (i.e. *MDSP*1 and 2) and more communication round trips (i.e. *MDSP*3).

*MDSP*3 includes the following processing steps.

Step 1: Download only the primary key from the server in location *A* to the mobile device using the following SQL command: `Select PK1 from TblInLocA order by PK1;`

Step 2: Download only the primary key from the server in location *B* to the mobile device using similar SQL as for step 1 above, but accessing the appropriate table in location *B*.

Step 3: Local processing is now carried out based on the two lists of primary keys. Once the matches have been produced, the following further steps are followed..

Step 4: We then need to download only non-key attributes from one server only. In this case, the matching PKs must be listed in the query to request non-key attributes from a server. Supposed the server is in location *A*, the SQL to donwload the non-key attributes of the matched PK is as follows: `Select * from TblInLocB where PK2 in (N…)`, where *N* is a list of matched PK produced in step 3 above. This process is similar to step 3 in *MDSP*2.

Algorithm 3.3: MDSP3

```
// Step 1:
1. Send Query-1 (PK) to Server-1:
   Select PK Into R
   From Table_R;
// Step 2:
2. Send Query-2 (PK) to Server-2:
   Select PK Into S
   From Table_S;
// Step 3: On-mobile merging
3. i=1; j=1;
4. Let Query result Qr = {}
5. Do
6.   If i is EOF(R) or j is EOF(S) Then
7.      Exit Loop
8.   End If
9.   If Ri.PK = Sj.PK Then
        // Step 4:
10.     If R.other_attributes needed for Qr Then
11.        Send Query-3 to Server-1:
           Select * Into Rtemp
           From Table_R
           Where Ri.PK=Sj.PK;
12.     Else
        // Step 4a:
13.        If S.other_attributes needed for Qr Then
14.        Send Query-4 to Server-2:
           Select * Into Stemp
           From Table_S
           Where Ri.PK=Sj.PK;
15.     End If
16.     Qr = Ri + Stemp + Rtemp;
17.     i++; j++;
18.  End If
19.End Do
```

Figure 3.4: MDSP3 Algorithm

Optionally, when both servers have unique information about each record, primarily due to its location, the matched PKs need to get details of the records (e.g. non-key attributes) from the other server as well. Therefore,

Step 4a: Optionally obtain non-key attributes from the other server using the similar SQL as in step 4 above, but accessing the respective table in the other server.

The pseudocode of MDSP-3 algorithm is shown in the following Figure 3.4.

3.2.2 Server Side Processing (SSP)

When a large amount of data is to be downloaded from both servers, there is the possibility that they may not be supported by the limited memory and processing capability of mobile devices (Myers, 2003; Paulson, 2003). This section investigates alternative techniques that can help minimize the amount of data transfer, taking into account the mobile device limitations. This alternative moves the processing from local processing to server processing. Hence, this technique requires only one list from one of the servers to be downloaded and then the mobile device acts as a mediator to send the data to the other server to be processed. In this section, we focus on two possible versions of performing server side processing, which we name *SSP*1 and *SSP*2:

(*a*) *SSP*1

*SSP*1 downloads everything from one server, sends a query to the other server using this data, and then performs the whole matching process in the second server. *SSP*1 would have every required column consisting of primary and non-primary keys from the table in the server in location *A*.

Compared with mobile-device side processing, server side processing is different in the sense that with the one full list downloaded into the mobile device, a query is then sent to the other server of location *B* to be processed in determining the qualified match.

*SSP*1 follows the steps described below:

Step 1: Download every attribute including primary and non primary keys from the server in location *A* to the mobile device. In downloading the data, the mobile device would send the following SQL query to the server in location *A*: `Select * from TblInLocA order by PK1;`

Step 2: Send the primary key that is obtained from step 1 above to the server in location *B* to process the matching in order to determine the qualified match. The SQL to be sent to the server in location *B* would be: `Select * from TblInLocB where PK2 in (N…)`, where *N* is a list of PK obtained from step 1 above.

Step 3: The server which processes the query from step 2 above will return all attributes of the matched records. These are returned to the mobile device as the final results. In other words, qualification determination is the process performed in the second server using the primary keys obtained initially from the first server.

The pseudocode of SSP1 algorithm is shown in Figure 3.5. Since the processing is carried out purely in the server, the server processing is not shown in the algorithm.

Algorithm 3.4: SSP1

```
1. Let query result Qr = {}
2. Send Query-1 to Server-1:
   Select * Into R
   From Table_R
   Order By PK;
3. Send Query-2 to Server-2:
   Select * Into S
   From Table_S
   Where Table_S.PK IN {R.PK.item1,R.PK.item2...R.PK.itemn};
4. Qr = S
```

Figure 3.5: SSP1 Algorithm

(*b*) *SSP*2

*SSP*2 downloads the primary keys only from one server and then sends a query to the other server using these primary keys, and finally does the match processing in the second server. After processing on the server, the results including all non primary

key attributes, together with the primary keys that are qualified, are sent back altogether to the mobile device. This version eliminates the overhead cost of downloading unwanted non primary key data, since in most highly selective operations, most attributes are not needed to contribute to the end results. But on the other hand, more round trips are needed.

In more detail, *SSP*2 requires the following steps:

Step 1: Download only the primary key from the server in Location *A* to the mobile device using the following SQL command: `Select PK1 from TblInLocA order by 1`.

Step 2: Send the primary key downloaded from step 1 to the server in Location *B* to find the matching qualified data. In this case, the SQL query would be identical to the second step of *SSP*1 where the SQL includes a list of primary keys from step 1 to be shipped to the server in location *B*.

Step 3: The process in the second server, which processes the query sent by the mobile device as stated in the second step, would be identical to the corresponding step in *SSP*1.

Step 4: Optionally however, if the non-primary key attributes in the server in location *A* have additional information about the query results which are not in location *B*, the mobile device has to send a query to the first server to obtain these details. This option is different from *SSP*1, because the first step of *SSP*1 itself already downloads all the details. This is not the case with *SSP*2 as the first step only obtains the PK, and consequently, if the mobile device needs any other details, then step 4 becomes necessary.

The pseudocode of *SSP*2 algorithm is shown in Figure 3.6. When comparing *SSP*1 wth *SSP*2, *SSP*2 should appear to be more efficient in terms of additional download cost as well as memory usage since it downloads only one full list from one server and the other server is just the primary key values.

```
Algorithm 3.5: SSP-2
1. Let query result Qr = {}
2. Send Query-1 to Server-1:
   Select PK Into R
   From Table_R;
3. Send Query-2 to Server-2:
   Select * Into S
   From Table_S
   Where Table_S.PK IN {R.PK.item1,R.PK.item2...R.PK.itemn};
4. If R.other_attributes needed for Qr Then
5.    Send Query-3 to Server-1:
      Select * Into Rtemp
      From Table_R
      Where Ri.PK = S.PK;
6. End If
7. Qr = S + Rtemp
```

Figure 3.6: SSP2 Algorithm

3.2.3 Walkthrough Examples

In order to clarify how each of the proposed MDSP and SSP techniques work, we will give exemplary illustrations. In this section, we provide a comprehensive, step-by-step walkthrough to give a general understanding of how the records are obtained and processed before being output on the mobile device. The following Figure 3.7 will be used as sample data throughout this section.

(a) Mobile Device Side Processing Walkthrough Examples

In order to illustrate a complete walkthrough example of the MDSP technique, we will assume that a query has been issued to different servers which are responsible for storing information about movies that are currently showing and would like to know which of the movies exited in both the different servers. We will illustrate it using two different scenarios. With the first scenario we demonstrate the sort merge processing technique, and with the second scenario we demonstrate the use of hash-based processing.

Movie_Title	Number
The Lion King	1
Harry Potter	2
Lord of the Ring	3
Friends	5
Batman	6
James Bond	8
Shark Tale	11
Ghost Train	14
Quill	16
Madagascar	17
Dogs and Cats	19
…..	…..
…..	…..
…..	…..

(a) Table 1 in Server *R*

Movie_Title	Number
Batman	2
The Lion King	7
Lord of the Ring	8
Kung Fu	10
Dogs and Cats	11
Quill	12
House Of Wax	15
Amitiville	18
Zorro	21
American Pie	4
Intolerable Cruelty	23
…..	…..
…..	…..
…..	…..

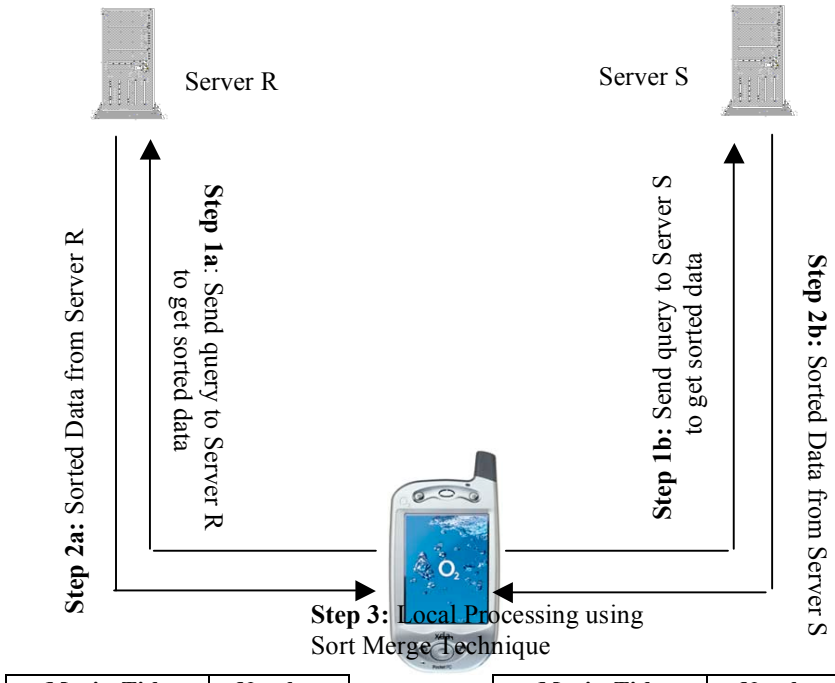
(b) Table 2 in Server *S*

Figure 3.7: Sample Data (Unsorted)

Scenario 1: Sort-Merge Processing

With the first scenario, we use the sort-merge processing and thus, a query to first obtain sorted data from both servers respectively is needed before a merge can be obtained.

The first step is to send a query to both servers to get sorted data into the mobile device. And from the mobile device, local processing is being done through the sort merge technique and the results are obtained. The whole process of using the traditional sort merge in the proposed MDSP technique is summarized in Figure 3.8.



Movie_Title	Number
Batman	6
Dogs and Cats	19
Friends	5
Ghost Train	14
Harry Potter	2
James Bond	8
Lord of the Ring	3
Madagascar	17
Quill	16
Shark Tale	11
The Lion King	1

Qualified match between the PK values are process in the mobile device locally

Movie_Title	Number
American Pie	4
Amitiville	18
Batman	2
Dogs and Cats	11
House Of Wax	15
Intolerable Cruelty	23
Kung Fu	10
Lord of the Ring	8
Quill	12
The Lion King	7
Zorro	21

Step 4: Return the process qualify match to a new set of join results table

Join Results:

Movie_Title	Number from R	Number from S
Batman	6	2
Dogs and Cats	19	11
Lord of the Ring	3	8
Quill	16	12
The Lion King	1	7

Figure 3.8: Traditional sort merge in MDSP

Scenario 2: Hash-based Processing

With the second scenario we use hash-based processing; therefore, unlike the first scenario, our query will eliminate the `Order By` clause when we query the servers.

The first step is to send a query to both servers to get individual list data into the mobile device that is in their original unsorted list. And from the mobile device, local processing is being done through the hash-based technique in Step 2 followed by Step 3 which is hashing and probing stage.

However, we take into account the memory limitation because the memory of the mobile device is small and therefore everything might not fit into the main memory, thereby causing memory overflow. This memory overflow will cause some hash data to be sent to the hard disk and be returned to the main memory when needed which is taken care of in Step 3 of the process.

Next step is that, with the two lists of data that have been downloaded from Server *R* and Server *S* in the mobile device, we will now perform one of the traditional hash-based techniques to obtain a qualifying match. The first step in the hashing technique is to take a list of data from the server which contains the fewer data sets. However in this case, we have the same number of data sets and therefore assume that we take the list of data from Server *S* to be hashed.

Figure 3.9 shows a list of data from Server *S* that has been downloaded into the mobile device ready to be hashed. The hash results will be divided between the main memory and the hard disk due to the limited memory capacity of the mobile device. Therefore, we will assume the main memory can hold a hash index of only 1-6 at a time and any index after 6 would flow to the hard disk. This introduces the concept of memory overflow.

Original table of Server S

Movie_Title	Number
Batman	2
The Lion King	7
Lord of the Ring	8
Kung Fu	10
Dogs and Cats	11
Quill	12
House Of Wax	15
Amitiville	18
Zorro	21
American Pie	4
Intolerable Cruelty	23
…..	…..
…..	…..
…..	…..

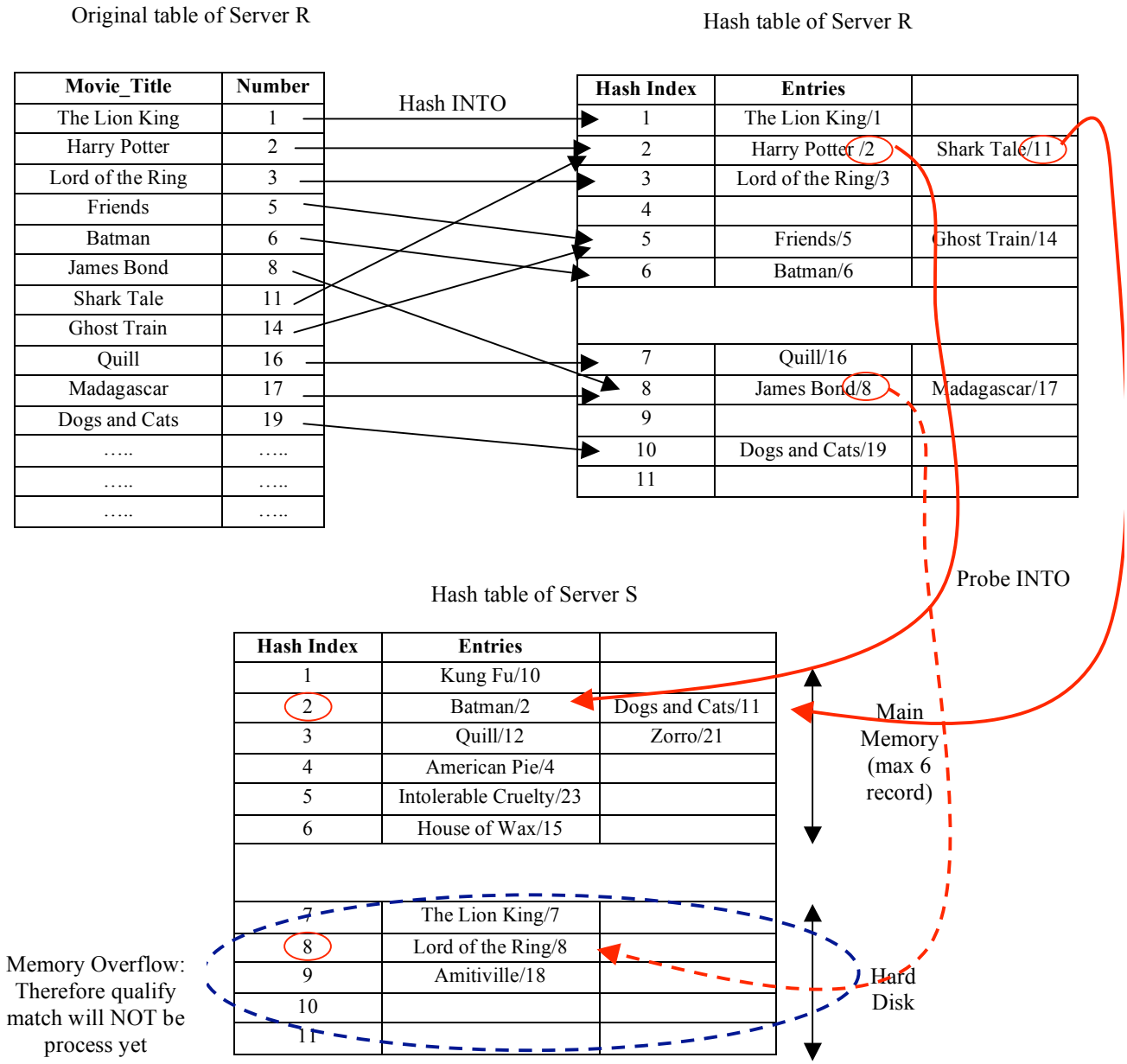
Hash table of Server S

Main memory size of 6 records hash entries

Hash Index	Entries	
1	Kung Fu/10	
2	Batman/2	Dogs and Cats/11
3	Quill/12	Zorro/21
4	American Pie/4	
5	Intolerable Cruelty/23	
6	House of Wax/15	
7	The Lion King/7	
8	Lord of the Ring/8	
9	Amitiville/18	
10		
11		

Figure 3.9: Data from Server *S* ready to be Hash

After performing the hashing process, we proceed to the probing stage of the hash-based technique whereby we perform a hash of the other table which is from Server *R* and probe it against the hash table of Server *S*. In this scenario, in order to demonstrate using hash-based processing in a mobile environment, we will assume that the join attribute is the number attribute instead of the movie_title. This is illustrated in Figure 3.10.



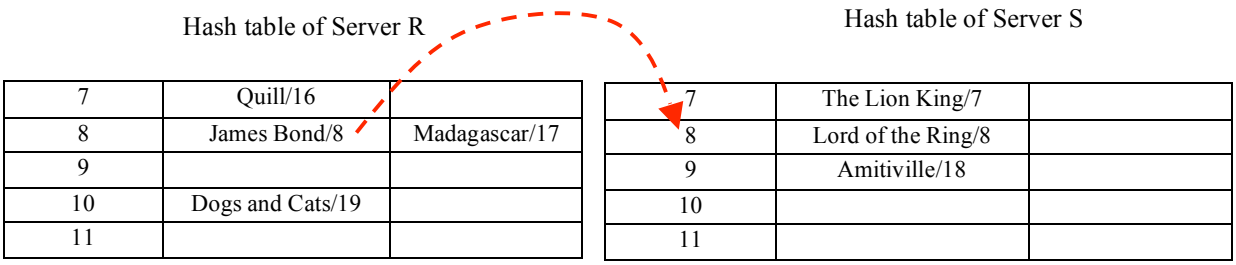
Join Results (Main Memory):

Movie_Title (R)	Number from R/S	Movie_Title (S)
Harry Potter	2	Batman
Shark Tale	11	Dogs and Cats

Figure 3.10: Obtaining Results of the Hashing Approach

The last step addresses the memory overflow issue. This means we will now process for a qualify match that is in the hard disk. Thus, the original index number of 1-6 that was previously in the main memory will be sent to the hard disk when the data sets from the hard disk are uploaded into the main memory.

In order to do that, recalling that our block size that we determined in the main memory is from index numbers 1-6, we will now take the next 6 index numbers of 7-12 from the hard disk onto the main memory for processing as in Figure 3.11. Therefore, the data as depicted in Figure 3.10, is now in the main memory ready to be processed. The qualify match in this case is James Bond/8/Lord of the Ring which we will send to combine with the rest of the other qualify matches in the join results table as in Figure 3.12. This process will be repeated until all blocks have been processed.



7	Quill/16	
8	James Bond/8	Madagascar/17
9		
10	Dogs and Cats/19	
11		

7	The Lion King/7	
8	Lord of the Ring/8	
9	Amitiville/18	
10		
11		

Figure 3.11: Data for the next set of index to be Hash

Movie_Title (R)	Number from R/S	Movie_Title (S)
Harry Potter	2	Batman
Shark Tale	11	Dogs and Cats
James Bond	8	Lord of the Ring

Figure 3.12: Results of the Hashing Process

(b) Server Side Processing Walkthrough Example

We use SSP2 to demonstrate how server side processing works. As in SSP2, we download only the primary keys from one server and send it to the other server to be processed. Therefore, we first obtain only the primary key value from Server *R*.

After that, we send the list of primary keys that we obtained from Server *R* to Server *S* so that it can process and determine which primary key values from Server *R* existed in Server *S*. The aim is to obtain the results of the matching items together with the non-key attributes obtained from Server *S*. Figure 3.13 illustrates the processing for SSP that shows the final results of the qualified match which the server has processed and that are to be sent back to the mobile device.

Optionally, if the results for the non-key attributes obtained from Server *S* are insufficient, or if there are additional non-key attributes corresponding to the qualified primary key that the user may need, he/she can then send an additional query to retrieve the additional non-key attributes from Server *R* based on the specified qualified primary key match.

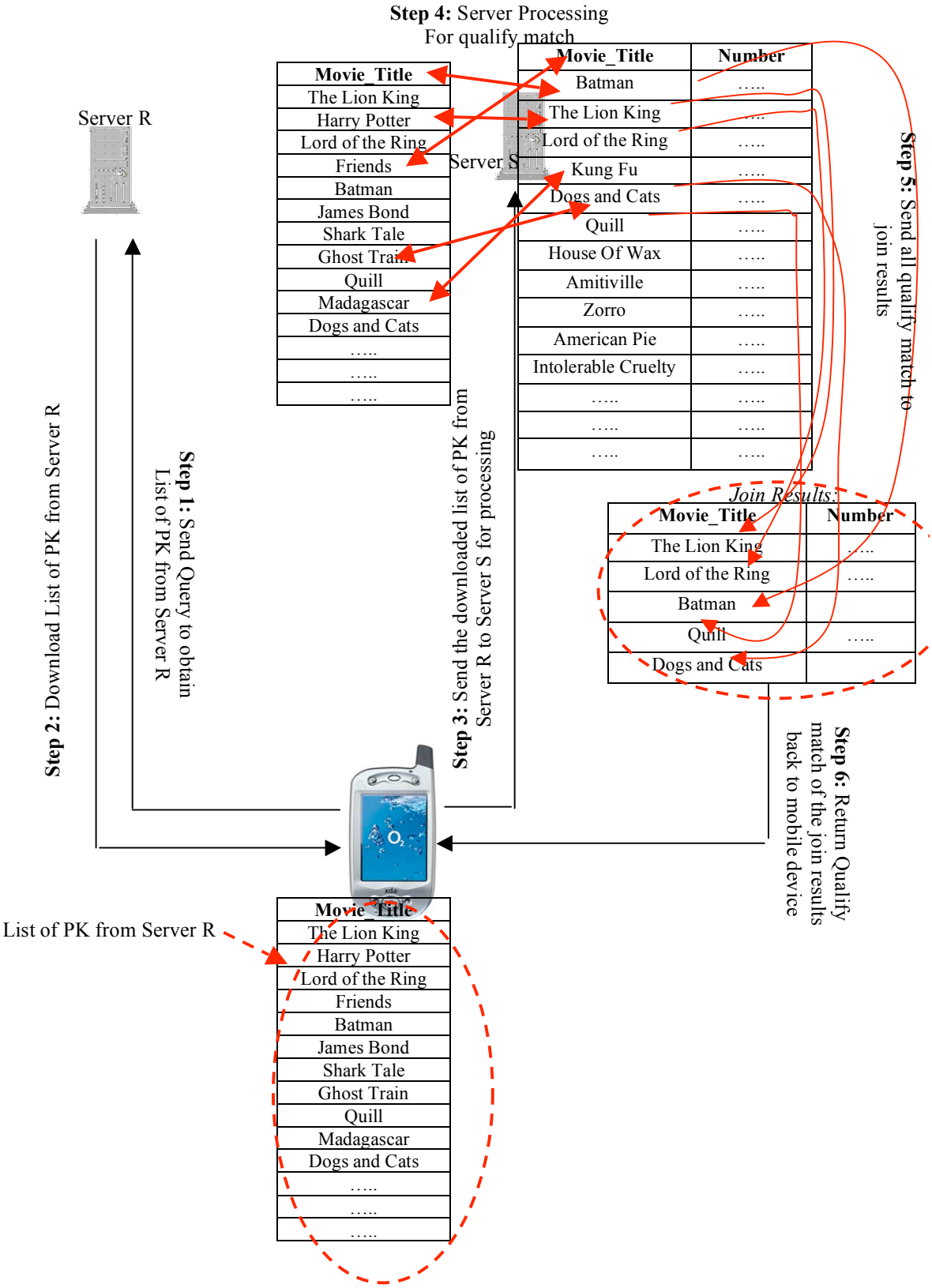


Figure 3.13: Processing using SSP based

3.3 Block-based Processing Techniques

The proposed techniques that are built on top of the previously discussed two basic techniques in this section are: (*i*) Block-based Processing (BBP), and (*ii*) Aggregate Block-based Join Processing (ABJP). A walkthrough example of each of the proposed techniques will be presented at the end of this section.

3.3.1 Block-Based Processing (BBP)

Considering that mobile devices have limited memory capacity, in many circumstances, it is essential that the data not be transferred between server and the mobile device as a whole, but rather it should be broken down and transferred in a block manner, fetching one block at a time. Both *MDSP* and *SSP* as discussed previously still require the download of at least one full list from the server, and consequently it may cause insufficient memory space for storing the full list on the mobile devices itself. Hence, it is extremely important to provide another approach considering that the mobile device has a small capacity as well as limited processing capability.

The following techniques incorporate processing the list of data in blocks regardless of whether the processing is done locally on the mobile device or whether it is done on the server. Incrementing processing in blocks is one of the main features which will maximize the efficiency of the processing. A mobile user might not be interested in looking at all of the results obtained from the two servers, but rather would be content with just the few top results obtained. If this is the case, then it is worthless to further process the subsequent results since the mobile user would have been satisfied and truncated the call.

Thus, if processing does not incorporate the block-based feature, then all results will be processed and given to the user. But with the block-based feature, the process can be stopped immediately at any checkpoint. Therefore, the benefit of implementing block-based processing is that it will reduce local mobile device memory usage and avoid unnecessary or worthless transfer bytes.

In this section, we would like to focus on two different versions of performing a block-based processing.

(*a*) *MDSP-Block*

MDSP-Block is a mobile-side processing technique that incorporates block-based processing. First, we need to provide a fixed number of records per block to be downloaded from one server from a location which is then to be compared with the list from another server. It is called 'Static' because, regardless of whether the records in a block have been matched against and kept, we do not eliminate the block until all the records in that block have been compared and processed. We would keep the whole block event although that particular block may contain records that have been qualified and sent to the mobile device.

We can demonstrate the above technique with the following steps:

Step 1: Download one block consisting of N^{th} records in ascending order from table in Location *A*. The mobile device would send the following SQL query to the server: `Select * from TblInLocA where i <= recnum < j order by PK1`. The records to be fetched from the server are denoted from record number *i* to *j*.

Step 2: The second step is identical to the first one, except that it now fetches records from the table in location *B*.

Step 3: The mobile device will then simply perform a comparison between the two blocks of records and keep the qualified match.

Step 4: To continue processing another block, we look at the latest record of each block and keep the block in which the last record is the larger, and discard the other block.

Step 5: The block that has been discarded in Step 4 above needs to be replaced by a new block.

Step 6: Repeat the above processing until all blocks have been downloaded and processed. Note that the record numbers *i* and *j* for each block are incremented respectively.

The pseudocode of MDSP-Block algorithm is shown in Figure 3.14.

Algorithm 3.6: MDSP Block

```
1. Let Query Result Qr = {}
2. Set R_Flag = S_Flag = 0
3. I = j= 0
4. Do

      // Download a new block
5.    If R_Flag = 0 Then
6.       Send Query-1 to Server-1:
         Select * Into R
         From Table_R
         Order By PK
         // If the SQL Above returns NULL value
7.       If R = NULL Then
8.          Exit Loop
9.       End If

         // Set the flag to identify a new block just
         // been downloaded
10.      Set R_Flag = 1
11.   End If

      // Download a new block
12.   If S_Flag = 0 Then
13.      Send Query-2 to Server-2:
         Select * Into S
         From Table_S
         Order By PK
         // If the SQL Above returns NULL value
14.      If S = NULL Then
15.         Exit Loop
16.      End If

         // Set the flag to identify a new block just
         // been downloaded
17.      Set S_Flag = 1
18.   End If

      //Comparing one record at a time
19.   If Ri.PK = Sj.PK Then
20.      Qr = Ri=Sj
21.      i++; j++
22.   Else
23.      If Ri.PK <Sj.PK Then
24.         i++
25.      Else
26.         If Ri.PK >Sj.PK Then
27.            j++
28.         End If
```

```
29.       End If
30.    End If

       // Set the flag to show new R block needs to
       // be downloaded
31.    If end_block (Ri) Then
32.       Set R_Flag = 0
33.    Else
34.       If end_block (Sj) Then
             // Set the flag to show new S block needs to
             // be downloaded
35.          Set S_Flag = 0
36.       End If
37.     End if
38. End Do
```

Figure 3.14: MDSP Block-based Algorithm

(*b*) *SSP-Block*

In *SSP-Block*, we touch on exploiting a block-based processing on the server side. Basically, it is almost similar to the original *SSP* techniques that have been discussed in the previous section, where only one server will be involved in the processing. The only difference is that in *SSP-Block* we do not download the whole list at once but rather we divide it into blocks and download it block-by-block.

We can demonstrate the above technique with the following steps:

Step 1: Count the records for each table and determine which table has the smaller number of records to be then downloaded.

Step 2: Download one block in a similar way as the *MDSP-Block*.

Step 3: Send the one block that has been downloaded from the chosen server to the other server to be processed.

Step 4: Return the qualified matches, if any, from the server to the mobile device.

Step 5: Repeat the processing from Steps 2 to 4 until all blocks from the table from the chosen server have been downloaded and processed.

The pseudocode of SSP-Block algorithm is shown in Figure 3.15.

```
Algorithm 3.7: SSP Block
1. Let query result Qr = {}
2. Do
3.    Send Query-1 to Server-1:
      Select * Into R
      From Table_R
      Order By PK
4.    Send Query-2 to Server-2:
      Select * Into S
      From Table_S
      Where Table_S.PK IN {R.PK.item1,R.PK.item2...R.PK.itemn};
5.    If R.other_attributes needed for Qr Then
6.       Send Query-3 to Server-1:
         Select * Into Rtemp
         From Table_R
         Where Ri.PK = S.PK;
7.    End If
8.    Qr = S + Rtemp
9. End Do Until all blocks from Server-1 Processed
```

Figure 3.15: SSP Block Algorithm

3.3.2 Aggregate Block-based Join Processing

In this section, the proposed techniques actually build upon the two existing basic techniques that were discussed previously, namely: (*i*) Mobile device side processing (MDSP), and (*ii*) Server side processing (SSP). We refer to these types of queries as "*Aggregate Block-Based Join*" queries. Basically, the aggregate block join processing is to show the use of extending the block processing with the aggregate function. In comparison with previously discussed block-based approach, this proposed algorithm would be an extension technique where our main aim is to use the aggregate function to download the count of a particular group rather than the actual block that is within the block size that contains all the records.

Recalling the block-based approach, we defined the block size prior to processing so that there is always a limit to the number of records that can be downloaded at one time in order to prevent over congestion. However, the only difference between the block approach in the previous section and this section is that, in this proposed aggregate-based block join processing, the records in a block will have something in common that we are particularly interested in that binds them

together. Figures 3.16a and 3.16b show a big picture of the difference between the previous block-based processing with the aggregate-based block join processing respectively.

Therefore in this section, our proposed techniques are extended versions - incorporating aggregation operation into the basic techniques and hence, it is known as Aggregate-based Block Join Processing (ABJP). Like the mobile join described in the previous section, we also use an MDSP and an SSP-based method for the aggregate query processing.

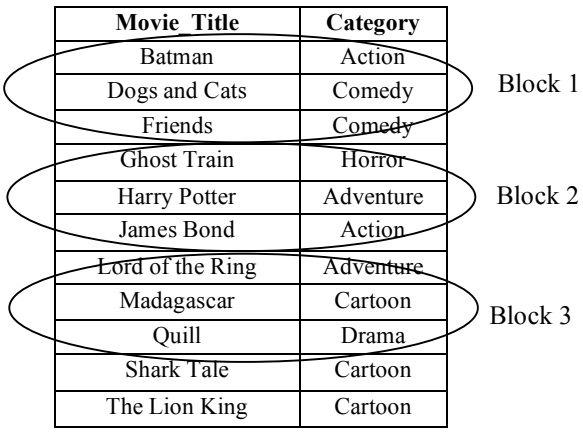


Figure 3.16a: Block-based Processing

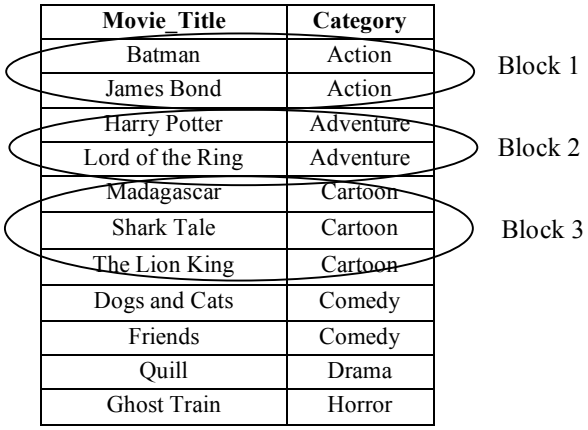


Figure 3.16b: Aggregate Block Join

(a) Aggregate Block Join Processing (ABJP) – MDSP version

Generally, mobile device side processing with aggregate operation deals with acquiring information from the servers to the mobile device to be processed locally on the mobile device and then executing the aggregation operation. Hence, the mobile device will process locally when determining the matching items first before carrying out the post-processing operation, which in this case is aggregation.

In this version of aggregate query processing on the mobile device side, everything is downloaded from both servers to be processed locally on the device itself. Then only the aggregation operation takes place once the intersection between the two different lists from the remote servers is matched locally from the mobile devices.

Aggregate Join Query Processing using MDSP approach has steps as follows:

Step 1: Obtain the count for the category attribute you are interested in to be grouped from Server *A* to the mobile device. In obtaining the count for the category attribute from Server *A*, the mobile device might invoke the following SQL query: `Select category, count(*) from TblInLocA group by category.`

Step 2: Get the count for the category attribute you are interested in to be group from Server *B* to the mobile device. Similar SQL may be used to get the count for the category attribute from Server *B*, such as: `Select category, count(*) from TblInLocB group by category.`

Step 3: Eliminate category attributes that contain zero count because we are interested only in items that exist in both servers which is the intersection between the two different lists.

Step 4: Download all the other attributes or selected attributes for each category attribute that does not contain zero count in a sorted manner one category at a time from both Servers *A* and *B* respectively to the mobile device. The SQL query that can be used in obtaining the other attributes from the respective servers in Location *A* and Location *B* are as follows:

```
Select title, category
From TblInLocA
Where category = 'catname'
Order by title;

Select title, category
From TblInLocB
Where category = 'catname'
Order by title;
```

Step 5: Perform matching locally on the mobile device to obtain the full list of the intersection

Step 6: Repeat Step 4 and Step 5 until all category attributes have been satisfied.

The pseudocode of the MDSP-Aggregate algorithm is shown in Figure 3.17.

Algorithm 3.8: MDSP Aggregate

```
1. Send Query-1 to Server-1:
   Select Category, Count(*) into R
   From Table_R
   Group By Category;
2. Send Query-2 to Server-2:
   Select Category, Count(*) into S
   From Table_S
   Group By Category;
3. i=1; j=1;
4. Let Query Result Qr = {}
5. Do
6.   If i is EOF(R) or j is EOF(S) Then
7.     Exit Loop
8.   End If
9.   If Ri.Category = Sj.Category Then
10.    Select PK, Category INTO R'
       From Table_R
       Where Category = Ri.Category
       Order By PK;
11.    Select PK, Category INTO S'
       From Table_S
       Where Category = Sj.Category
       Order By PK;
12.  End If
13.End Do
14.m=1; n=1
15.Do
16.  If m is EOF(R') or n is EOF(S') Then
17.    Exit Loop
18.  End If
19.  If R'm.PK = S'n.PK Then
20.    Qr = R'm = S'n
22.    m++; n++
22.  Else
23.    IF R'm.PK > S'n.PK Then
24.      n++
25.    Else
26.      m++
27.    End If
28.  END IF
29.  i++; j++
30.  IF Ri.Category < Sj.Category Then
31.    i++
32.  ELSE
33.    j++
35.  END IF
36.END DO
```

Figure 3.17: MDSP Aggregate Algorithm

(b) Aggregate Block Join Processing (ABJP) – SSP version

In this section, we would like to focus on the server side processing that is extended with the aggregation operation. It is possible that if a large amount of data is to be downloaded from both servers, it may not be supported by the limited memory and processing capability of mobile devices.

Hence, we now investigate the alternative techniques that can help minimize the amount of data transfer - taking into account the mobile device limitations. This alternative moves the processing in determining the qualified data from locally processing on the mobile device to processing on the server side. This technique requires only one list from one of the servers to be downloaded and then the mobile device acts as a mediator to send the primary key to the other server to be processed in order to get the qualified match before performing the post-processing operation such as aggregation.

SSP – Aggregate Query Processing includes the following steps:

Step 1: Similarly to MDSP – Aggregate Query Processing, obtain the count for the category attribute you are interested in to be grouped from servers in both Location *A* and Location *B* to the mobile device. In obtaining the count for the category attribute from the server in Location A, the mobile device might invoke the following SQL query: `Select category, count(*) from TblInLocA group by category.`

Step 2: In obtaining the count for the category attribute from the server in Location B, the mobile device will use the same SQL statement as above which is: `Select category, count(*) from TblInLocB group by category.`

Step 3: Now we compare the two count lists with their respective category attribute and choose the lower count from each respective category attribute by comparing the lists obtained from the two servers. Hence, we keep a new list of category attributes after the counts have been compared, and select the one that contains the lowest count. We also eliminate the category that contains zero counts in the other servers as we are interested only in items that exist in both servers.

Step 4(a): The first part of Step 4 is to download the join attribute or any other attributes from either server in Location *A* or Location *B* in a sorted manner depending on whether which of the server contain lesser count for that particular category in step 3. The SQL query that is use in this is as follows: `Select title, category from TblInLocA where category='catname' order by title.`

Algorithm 3.9: SSP Aggregate

```
// Step 1: Get the count from Server A
1.  Send a query to Server A to get the count for each group
    Select category, count(*)
    From TblInLocA
    Group By category;
    Store the group counts into CountA
// Step 2: Get the count from Server B
2.  Send a query to Server B to get the count for each group
    Select category, count(*)
    From TblInLocB
    Group By category;
.   Store the group counts into CountB
// Step 3: Compare the counts
3.  For each match of category between CountA and CountB
4.    Compare the counts
// Step 4: Send a query to the correct server
5.  If count of record CountA is < count of record CountB Then
6.    Send a query to Server A
      Select title, category
      From TblInLocA
      Where category='catname'
.     Order By title
7.  Else
8.    Send a query to Server B
      Select category, count(*)
      From TblInLocB
      Where title in (....) and category = 'catname'
9.  End If
10. End For
//Step 5: Repeat Steps 3 and 4 until all counts processed
```

Figure 3.18: SSP Aggregate Algorithm

Step 4(b): The second part of Step 4 is to send the above query to the other server that contains the larger number of count for that particular category. This step determines the match to get the intersection. The following SQL query can be use to

send the query to the other server to obtain the final count of the intersection for that particular category. `Select category, count(*) from TblInLocB where title in (....) and category = 'catname'.`

Step 5: Repeat Steps 3 and 4 until all category attributes have been satisfied.

The algorithm is shown in Figure 3.18. The five steps are clearly marked in the algorithm.

3.3.3 Walkthrough Examples

In this section, we would like to present walkthrough examples to demonstrate how MDSP and SSP work using blocks processing by using the sample data of the previous section where the data is originally unsorted in the respective servers.

(a) Block-based Processing Walkthrough Example

In order to illustrate BBP, we would initially issue a query using the ORDER BY command, so that the mobile user sees the data that has already been sorted.

Our aim is to obtain the qualified match that is the intersection between Server *R* and Server *S*; that is, the record that exists in both servers. By just looking at the above tables, we can see that our output items for intersection of Server *R* and Server *S* are Batman, Dogs and Cats, Lord of the Ring, Quill, and The Lion King. In the subsequent sub-sections, we demonstrate how block processing obtains the intersection results.

Walkthrough Example for MDSP-Block

Firstly we need to download one block from the sorted records in *S*1. As we assumed the block interval size is 3, therefore our first step is to obtain the first three records from *S*1. Step 2 is similar to Step 1 which is to obtain the first three records from *S*2 as in Figure 3.19a.

In Step 3, we need to compare *S*1 and *S*2 and keep the qualified match which is the record that exists in both *S*1 and *S*2. This is known as the intersection. As seen

above, the movie_title Batman exists in both *S*1 and *S*2 and this is our qualified match so we keep this record. This is illustrated in Figure 3.19b.

In the next step, we look at the latest records of each block and compare them to determine which block contains the larger final record and we retain that particular block. We notice that the last record of the block in *S*1 is Friends and the last record in *S*2 is Batman. So by comparing Friends and Batman, we keep the first block of *S*1 and discard the first block of *S*2 as in Figure 3.19c.

Movie_Title	Description
Batman	…..
Dogs and Cats	…..
Friends	…..
Ghost Train	…..
Harry Potter	…..
James Bond	…..
Lord of the Ring	…..
Madagascar	…..
Quill	…..
Shark Tale	…..
The Lion King	…..

S1

Movie_Title	Description
American Pie	…..
Amitiville	…..
Batman	…..
Dogs and Cats	…..
House Of Wax	…..
Intolerable Cruelty	…..
Kung Fu	…..
Lord of the Ring	…..
Quill	…..
The Lion King	…..
Zorro	…..

S2

Figure 3.19a: Blocks from the two tables (S1 and S2)

Block 1

Movie_Title	Description
Batman	…..
Dogs and Cats	…..
Friends	…..

Compare to get the same match

Block 1

Movie_Title	Description
American Pie	…..
Amitiville	…..
Batman	…..

Figure 3.19b: Comparison to obtain the match

Movie_Title	Description
Batman	…..
Dogs and Cats	…..
Friends	…..

Compare the last records in the block

Movie_Title	Description
American Pie	…..
Amitiville	…..
Batman	…..

Figure 3.19c: Comparison to determine which block to keep and which to discard

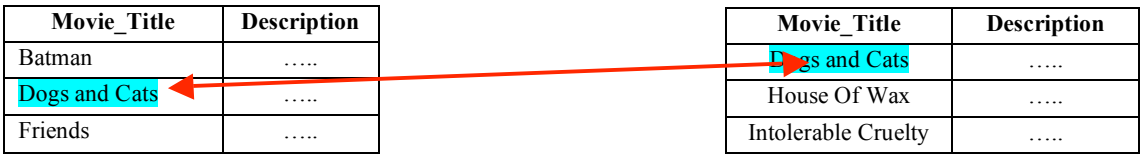


Figure 3.19d: Subsequent comparison to obtain the match

In the next step, we download the second block from the server that contains the table that holds the smaller final record. In this example, it would be *S*2 as we notice that Batman is smaller than Friends. So, the second block of *S*2 is the next three records from the second table. Again, we repeat the processing step to make a comparison in order to determine the qualified match between two blocks as in Figure 3.19d. All processes are repeated until all blocks have been downloaded and processed.

Walkthrough Example for SSP-Block

For *SSP* block-based processing, the steps are quite similar to *MDSP* but instead of downloading blocks from each server, we now need to download only from a server that contains a lower count record. Step 1 is to obtain the count from each server and we choose the table that contains the smaller number of records as our target table to be downloaded. Assuming the count function that returns *S*1 contains 26 records and *S*2 contains 35 records. So in the next step, we choose *S*1 to download our list in blocks and start downloading one block from the server that has the lower number of records which in this case is *S*1.

The difference between this version and other block-based versions is that we do not have to send a query asking for a sorted list. After downloading the first block from *S*1, in the next step we send the block to the server to be processed. Hence, we would send the block that we have just downloaded from *S*1 to *S*2 that contains the whole list as shown in Figure 3.20.

The records that are sent in blocks will be scanned through the whole list in the server and if there is any match, then the server returns the qualified match to the mobile device. This means that each record from the block will be compared with all

the records in the server and, if there is a match, then it will return to the mobile device. This process continues until all blocks have been sent to the server for comparison and processing.

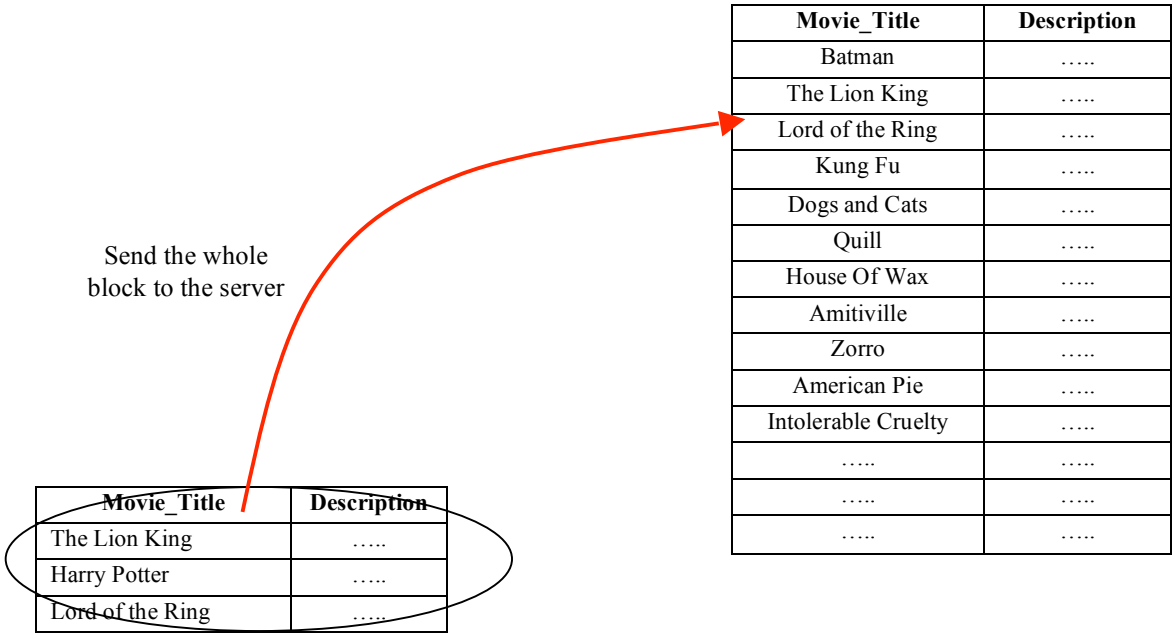


Figure 3.20: Transferring a block

(b) Aggregate Block-based Join Processing Walkthrough Example

This section presents an illustration of each of the proposed techniques to demonstrate how the processes work and differentiate between the different versions of the techniques.

Figure 3.21 shows two tables that have been obtained from two servers in different locations that are union compatible. This means that the servers' tables have actually been obtained from the same source, but because it is location-dependent processing, the contents obtained from different locations would have different contents but an identical structure.

Movie_Title	Category
The Lion King	Cartoon
Harry Potter 2	Adventure
Lord of the Ring	Adventure
Friends	Comedy
Batman	Action
James Bond	Action
Shark Tale	Cartoon
Ghost Train	Horror
Quill	Drama
Madagascar	Cartoon
Dogs and Cats	Adventure
…..	…..
…..	…..

(a) Table 1 in Location *A*

Movie_Title	Category
Batman	Action
The Lion King	Cartoon
Lord of the Ring	Adventure
Kung Fu	Action
Dogs and Cats	Adventure
Bewitched	Comedy
House Of Wax	Horror
Amitiville	Horror
Zorro	Action
American Pie	Comedy
Intolerable Cruelty	Comedy
…..	…..
…..	…..

(b) Table 2 in Location *B*

Figure 3.21: Tables from two locations to be processed on-mobile

Our aim is to obtain the qualified match of the intersection between *S*1 and *S*2 which is the record that exists in both *S*1 and *S*2, and group them in terms of their category with counts of each category. By just looking at the above tables, we can see that our output should be as indicated by Figure 3.22.

Category	Count
Action	1
Adventure	2
Cartoon	1
Comedy	0
Drama	0

Figure 3.22: Expected output intersection results with aggregation operation

Walkthrough Example for MDSP-based Aggregate Join Query Processing

In this section, we would like to present a walkthrough example to demonstrate how MDSP work with the extended feature of including the aggregation operation. Figure 3.23 shows two tables that are obtained from two servers of different locations that are union compatible where the servers contain different records but they are of identical structure.

Movie_Title	Category
The Lion King	Cartoon
Harry Potter 2	Adventure
Lord of the Ring	Adventure
Friends	Comedy
Batman	Action
James Bond	Action
Shark Tale	Cartoon
Ghost Train	Horror
Quill	Drama
Madagascar	Cartoon
Dogs and Cats	Adventure

(a) Table 1 in Server 1

Movie_Title	Category
Batman	Action
The Lion King	Cartoon
Lord of the Ring	Adventure
Kung Fu	Action
Dogs and Cats	Adventure
Bewitched	Comedy
House Of Wax	Horror
Amitiville	Horror
Zorro	Action
American Pie	Comedy
Intolerable Cruelty	Comedy

(b) Table 2 in Server 2

Figure 3.23: Tables from two servers to be processed on-mobile

Our aim is to obtain the qualified match of the intersection between $S1$ and $S2$ which are the records that exist in both and $S1$ and $S2$ and group them in terms of their category with a count of each category. In other words, we would like to join the records based on the join attribute between the two servers. The above tables show that our output should be that indicated by Figure 3.24.

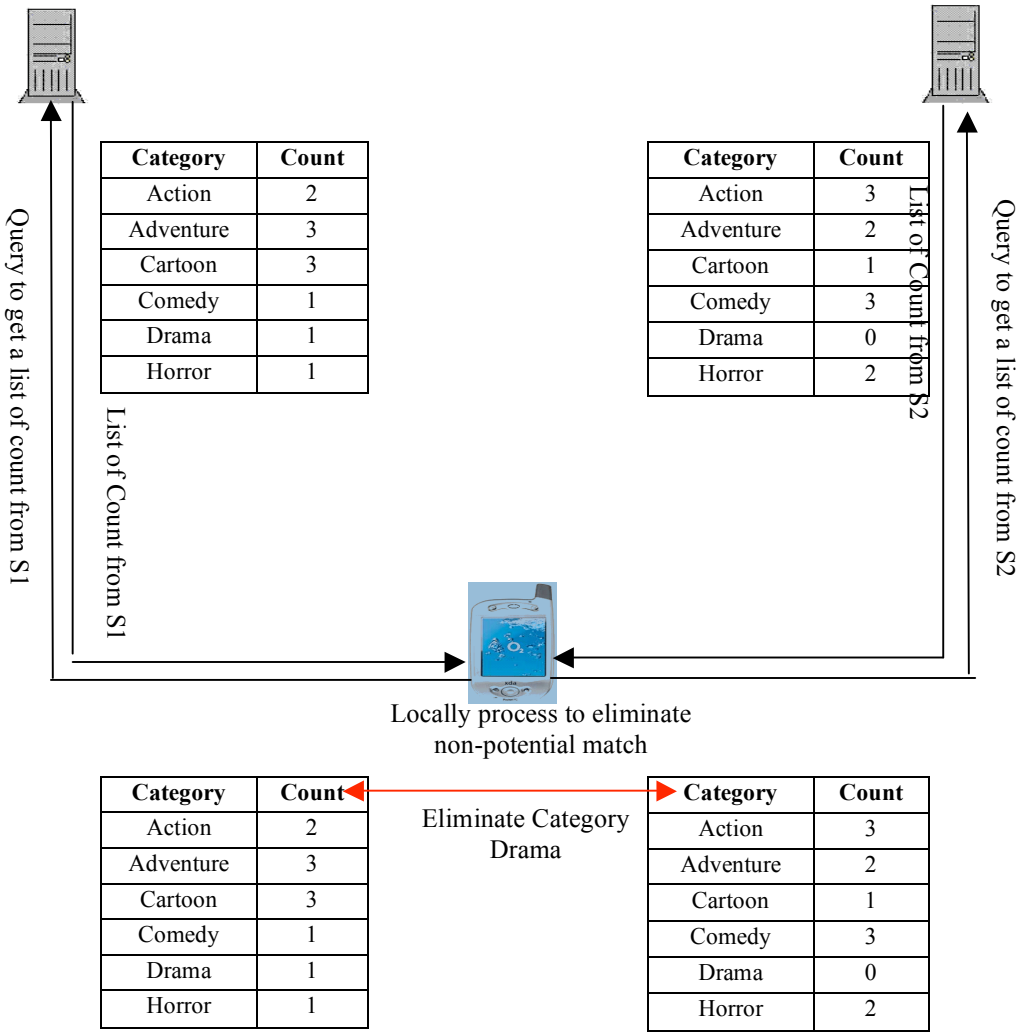
Category	Count
Action	1
Adventure	2
Cartoon	1
Comedy	0
Drama	0

Figure 3.24: Expected output intersection results with aggregation operation

Now we will demonstrate how to obtain the aggregate results by using MDSP and incorporating it with aggregate query processing.

Firstly, we need to count the items based on each category from both respectively. Therefore, our first step is to obtain the count from *S*1. Step 2 is similar to Step 1 which is to obtain the count from *S*2. In Step 3, we first determine the category that contains zero count in the other server and we will eliminate these items since we are interested only in items that exist in both *S1* and *S2* which is the intersection. In this case, we would ignore the drama category in *S1* because no drama category exists in *S2*.

Figure 3.25 shows the output for the count of each server that is downloaded to the mobile device as well as locally processing the two lists of count so that only potential matches are retained on the mobile device.



Category	Count
Action	2
Adventure	3
Cartoon	3
Comedy	1
Drama	1
Horror	1

Category	Count
Action	3
Adventure	2
Cartoon	1
Comedy	3
Drama	0
Horror	2

Category	Count
Action	2
Adventure	3
Cartoon	3
Comedy	1
Drama	1
Horror	1

Category	Count
Action	3
Adventure	2
Cartoon	1
Comedy	3
Drama	0
Horror	2

Figure 3.25: Downloading the Count Result from the Servers

Next, with the two lists of counts excluding the category drama, we would go through category by category and download the primary key from each server for the movies that falls within that category. Thus, Step 4 is to download the join key which is the movie title in this example from *S*1 for category 'Action' to the mobile device followed by Step 5 which is to download the movie title from *S*2 to the mobile device. The local process on the mobile device will then check whether or not the two lists of primary keys from *S*1 and *S*2 match. If matched, then they will be kept as one of the join results stored temporarily that will be needed to obtain a final count for each category. This can be seen in Figure 3.26.

Qualify Match Process Locally

Batman	Action
James Bond	Action

Batman	Action
Kung Fu	Action
Zorro	Action

Temporary Join Intersection Result:

Movie_Title	Movie_Category
Batman	Action

Figure 3.26: Temporary results to be combined later

Next, we move to downloading the primary key for the next category, 'Adventure' which is a repetition of Step 4 and Step 5 again. And the same process matching will be executed. And again, the qualify match will then be stored temporarily together with the previously obtained qualified match in the previous category as shown in Figure 3.27.

Harry Potter 2	Adventure
Lord of the Ring	Adventure
Dogs and Cats	Adventure

Lord of the Ring	Adventure
Dogs and Cats	Adventure

Temporary Join Intersection Result:

Movie_Title	Movie_Category
Batman	Action
Lord of the Ring	Adventure
Dogs and Cats	Adventure

Figure 3.27: Temporary results that have been combined with previous temporary record

Now that we are done with the second category 'Adventure', it will be followed by the third category and so on and so forth until all categories have been processed and matched. By the end of processing all the categories and obtaining the

qualify match, the final output in the temporary join intersection result would be as in Figure 3.28. Therefore, there will be three different categories with each category containing a different number of values.

Movie_Title	Movie_Category
Batman	Action
Lord of the Ring	Adventure
Dogs and Cats	Adventure
The Lion King	Cartoon

Figure 3.28: Final Results obtained from the temporary records

The final result that the mobile user would be interested in when making an aggregate query is a count number for each group. This basically would return a final output as in Figure 3.29 which is based on the grouping of each category.

Category	Count
Action	1
Adventure	2
Cartoon	1

Figure 3.29: Final Results to be displayed

Walkthrough Example for SSP-based Aggregate Join Query Processing

In this section, we would like to present a walkthrough example to demonstrate how SSP works with the extended feature of including the aggregation operation.

Firstly, we need to count the items based on each category from both servers in Location *A* and Location *B* respectively. Therefore, our first step is to obtain the count from *S*1. Step 2 is similar to Step 1 which obtains the count from *S*2. Figure 3.30 is the output for the count of each server in its respective location.

Category	Count
Action	2
Adventure	3
Cartoon	3
Comedy	1
Drama	1
Horror	1

(a) Count for Table 1 in Location *A*

Category	Count
Action	3
Adventure	2
Cartoon	1
Comedy	3
Drama	0
Horror	2

(b) Count for Table 2 in Location *B*

Figure 3.30: Count Result for tables in two different locations

In Step 3, we first determine the category that contains zero count in the other server and we will eliminate it since we are interested only in items that exist in both *S1* and *S2* which is the intersection. In this case, we ignore the drama category in *S1* because no drama category exists in *S2*. Next, we compare the two lists obtained separately from S1 and S2 and for each category, we compare the number of counts and always choose the server to download the other attributes that contains the lower count number. For example, for category Action, we compare the *S1* and *S2* counts and we discover that *S1* contains only 2 counts for Action whereas *S2* contains 3 counts for Action. So in this case, we would choose to download the join attributes from *S1* which is our Step 4.

Now, after determining that the Action category in S1 contains a fewer count, the first part of Step 4 is to download the join key, which is the movie title in this example, from S1 for category Action to the mobile device. The next part of Step 4 concerns the results. We now send the movie titles to *S2* to be matched to see whether or not both Batman and James Bond exist in *S2*. If they exist, then it is a match. Otherwise, we ignore it and go to the next category. In this example, we found that Batman in *S1* also exists in *S2* so it is a qualified match. So the intersection for category Action is Batman and the count is one since there is only one similarity. The process illustrated in Figure 3.31.

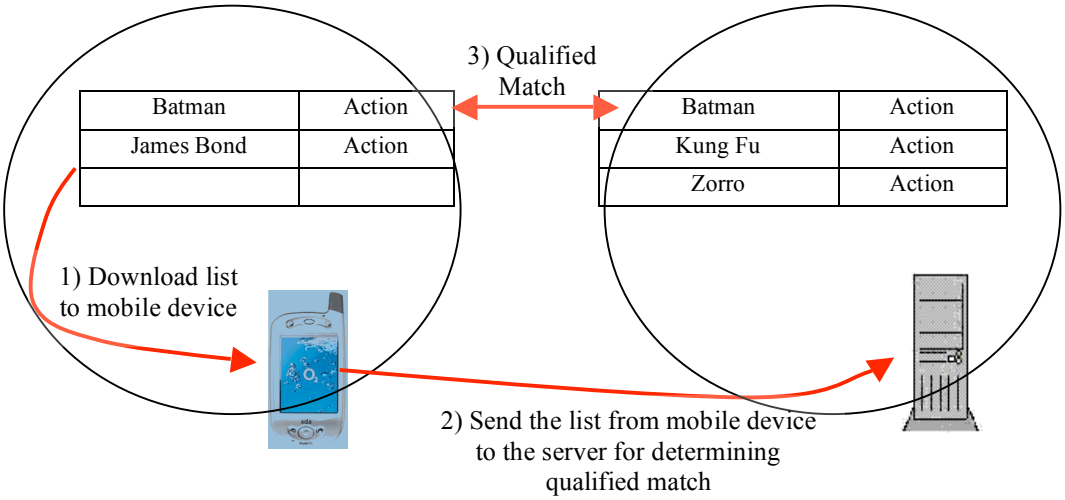


Figure 3.31: Processing in SSP for the 1st Block

Now that we are done with the category Action, we go to the next category which is Adventure. Again we repeat Step 3 to determine which server contains a smaller count to be chosen to download the join attribute. We can see that *S2* contains lower counts that *S1*, where *S*1 has three Adventure movies and *S*2 has only two.

So we again follow Step 4 which is to first download the attributes for category Adventure from *S2* to the mobile device and then send them to *S2* to see if they match with the other three movie titles in category Adventure in *S1* as in Figure 3.32.

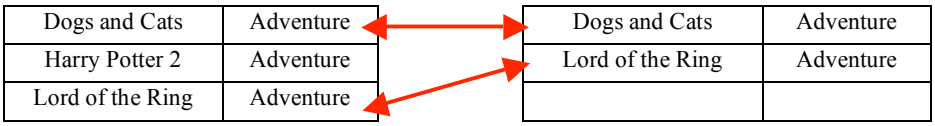


Figure 3.32: Comparison for Qualified Match

3.4 Conclusions

In this chapter, we have presented techniques relating to the processing of join queries in the mobile database environment. Our proposed algorithm takes into account mobile device side processing and server side processing with an extension to incorporate block-based processing. The contribution of this chapter is that we have investigated various forms of mobile side and server side processing, each with its own advantages and limitation. We consider a trade-off between server processing and data transfer cost. Two existing join techniques, namely sort-merge and hash join, have also been incorporated into the mobile side processing.

The contribution of this chapter extends to include block-based processing. In block-based processing, we incorporate the MDSP and SSP block processing, as well as the aggregate-based block processing. The main feature of the block-based processing is that the mobile users may have the capability to terminate the process at any point of time, as soon as the qualified matches have been obtained. Consequently, this reduces communication and processing costs, and addresses the limited visualization and storage space imposed by mobile devices.

Chapter 4

Mobile Top-k Join Query Processing

4.1 Introduction

This chapter addresses query processing to ensure efficiency for mobile devices with small screens. We want to ensure that the device will display high ranked information, which is commonly known as *Top-k*. In this case, there is no need to produce a complete list of query results. Our main concern is to address the limitation of small display screen and limited memory capacity on the mobile devices as well as low network bandwidth. Thus, all other unnecessary matched results would be ignored. This is also because complete output or browsing is not common and is impractical in mobile devices. Hence, it is crucial to return only top ranked results to mobile users.

Section 4.2 gives an overview of Top-k queries processing in the mobile environment. Then, two proposed methods for processing Top-k queries will be elaborated in Section 4.3, and finally Section 4.4 gives the conclusion and sums up the contribution.

4.2 Top-k: An Overview

A Top-k query is where the query will return only specific matches that have high rank (Ilyas, Aref, and Elmagarmid, 2003). In this thesis, since we are addressing multiple non-collaborative servers, the Top-k query is basically a Top-k join query, where the join operation produces high ranked results only. We are not addressing other Top-k queries as mentioned in the Literature Review chapter, but Top-k join only. Therefore, the join operation plays an important role in the query processing. The Top-k can be anything that the user specifies, where *k* represents a number, such as Top-3, Top-10, etc.

Top-k join query is extremely important in the mobile environment because the majority of the queries issued by mobile users are usually for the purpose of obtaining quick and important information only. Furthermore, they are restricted by several limitations that are unavoidable, such as low battery life span, limited memory and low network bandwidth. All these limitations constitute one of the several factors that limit the ability of having extensive processing time in the environment. Therefore, we need to take up only a short space of time by producing only the required important results as shown by a high ranking, and not every single join result.

In Top-k query processing, usually the ranking is based on numerical attributes, such as sales amount, box office amount, etc. An example query may be "List the Top-3 employees who made the most sales in the last month". There is a possibility that the ranking is based on a non-numerical attribute as well, such as film title. However, even though it physically appears as non-numerical, the ranking attribute (e.g. film title) can be associated with some sort of ranking in the sequence. For example, the first film is the top movie based on certain criteria, and the second film title is the next in the rank. Therefore, the query could be something like "List the Top-3 movies from various cinemas".

Therefore the ranking will always be based on some numerical figures whether it is a real figure in the attribute itself or just a sequence such as a ranking number. Because some lists of data may be already ranked, such as the top box office

movies, whilst other data sources can be raw (i.e. they are non-ranked, such as a list of movies that are in non-ranked order), Top-k join query processing must take into account whether or not the data sources have already been ranked locally.

An example of Top-k join query written in SQL statement is shown as follows:

```
SELECT *
FROM R1, R2,…,Rn
WHERE Join.Condition (R1,R2,…Rn)
ORDER BY f(R1.Score, R2.Score,...Rn.Score)
STOP AFTER ki
```

In the above SQL statement, function `f()` is a function that calculates the ranking based on user specified criteria, whereas the command `STOP AFTER` is a probationary command to demonstrate the use of Top-k query processing. Another similar command introduced by researchers is known as `LIMIT` which performs the same task.

In this thesis, our main concern is to perform a join processing to obtain a new rank from several lists of data.

Example 4.1: A mobile user query is to obtain the Top-3 restaurants in Melbourne from two servers where Server 1 is the list of restaurants ranked according to the amount of profit made, and Server 2 is the list of restaurants credited as tourist spots and ranked by the amount of profit as well. With these two lists of data, a new rank will be processed so that the result returned would be the Top-3 restaurants that are tourist accredited and located in Melbourne.

In this case, a function to calculate the new rank must be used. This function is user-defined. The following are two examples of formulas that can be used. Assuming that variables '*A*' and '*B*' represent the records from both servers, when these records are joined, a new value for the joined record must be calculated, such as: $(A+B)/2$. This is a common average formula. In another case, the user who invokes the query might want to give a different weight to each record, such as 30% and 70% to the respective records, and hence the average of the joined record is: $(0.3\times A) + (0.7\times B)$. This formula has a certain threshold value that is user-defined, and in this

example, the record from server *B* has a higher value in ranking because the weight is higher than that of the other record from server *A*.

The focus of this chapter is particularly on Top-k join processing in the mobile environment. One of the straightforward processing methods is to join everything and then sort it and finally process the Top-k. This is highly undesirable because it is well known, even in the traditional database query optimization, that join processing is expensive and should always be delayed whenever possible. Thus, a more desirable processing method is to delay the join as late as possible by obtaining the Top-k matches before getting the final join results. This will avoid processing the join beforehand.

Another core reason for delaying the join as late as possible is that the mobile memory size is small and therefore it may be difficult to produce the entire join results, let alone after the join processing. Therefore, when processing Top-k join, the following facts should be taken into account, which might be considered as advantages:

- Some input data (data sources) are already locally ranked, and
- Some servers may accept direct queries from mobile device.

However, other facts to be considered which appear to be constraints when processing Top-k join in a mobile environment are:

- Memory size of mobile device is limited,
- Battery life of mobile device is short, and
- Network bandwidth is slow causing extensive downloading time.

The following section proposes techniques that will process Top-k join assuming that the original data sources have already been locally ranked.

4.3 Top-k Join from Sorted Data Sources

The list of data returned from a server can be naturally sorted/ranked, or in another case, although the data is not originally ranked locally, if the server can accept a query that requests the data to be sorted, from an end result point of view, the server will provide the requested data in a ranked order. In the latter scenario, the mobile user sends a query to the server for a list of data in a particular sorting or ranking order.

This section introduces MDSP-based processing techniques whereby the data from each server is downloaded into the mobile device for a local Top-k join processing. In explaining the process, we use an example whereby the user wants to list the Top-3 box office movies from various cinemas; each cinema maintains its own server. We are using block-based processing, and assume that in this case each block size is 3 records.

When the first block from each server is sent to the mobile device memory, the local mobile processing can be carried out. In this thesis, we investigate two types of local processing techniques for Top-k where the input data sources come in a particular ranking order.

- Top-k Nested Loop Join
- Top-k Merge Join

4.3.1 Top-k Nested Loop Join

Using the example of "Listing top-3 box office movies from various cinemas", the first step is that the user would send a query to obtain the top-3 movies from each server. Because these two servers are non-collaborative to each other, the top-3 between the two different servers may be different. And when performing a local join, it can be done only if there is a match in the join attribute. Thus, to check whether or not the Top-3 movies from each server match, we will use a nested loop approach to check for the local match. The nested loop in this case is efficient since in this example, the number of records is very small.

Figure 4.1 gives an overview of the use of a nested loop in blocks of 3. Notice that once a block from each server is downloaded, the local nested loop join is done like any traditional nested loop processing. However, if Top-k results are not obtained from the first block, another set of blocks must be downloaded, in this case, one block from each server. Referring to Figure 4.1, the data from the second block of Server 1 must start the comparison with the first block of Server 2. This is necessary because the data from one server must be compared with the data of the other server. Therefore, the second block of Server 2 must also be compared with the previous blocks of Server 1 (e.g. block 1 Server 1). For clarity of presentation, the complete arrows representing each comparison are not shown in Figure 4.1.

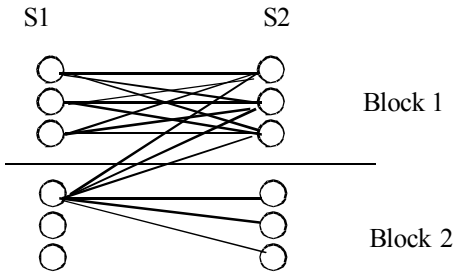


Figure 4.1: Nested Loop Join Processing

Figure 4.2 shows the downloading of block by block from both Server 1 and Server 2 to be matched using a nested loop approach. Note that each server gives the data that has already been ranked locally according to the qualification in each server. Therefore, the first three data refer to rank 1, 2, and 3. The example in Figure 4.2 shows that data item '*A*' from server 1 which is ranked 1 matched with item '*A*' from server 2 in rank 2. In this example, it shows 3 matches, after processing two blocks from each server. The black bullets show the matched records, whereas the white bullets show the non-matched records.

Next, we need to process them together to rank them into a new set of top-3 using the average formula for each of the records and the highest average will be the top followed by the second highest and third highest. Therefore, we can use a simple average formula: (Movie from Server 1 + Movie from Server 2) ÷ 2, assuming each

movie has an associated value, whether it be a sequence rank number or any other numerical figure, such as number of tickets sold.

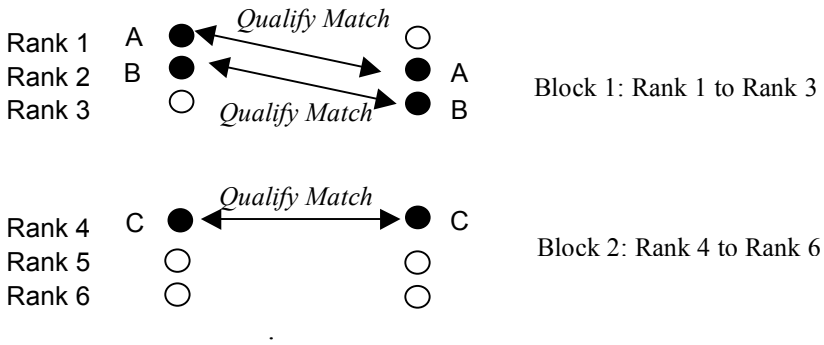


Figure 4.2: Nested Loop with Block-based

So, once a join match has been found (e.g. movie '*A*' from Server 1 rank 1 matches with the same movie from Server 2), a combined value is calculated (or averaged in this case). Once a new value has been calculated in each match, then the matched pairs can be sorted according to the new values.

Figure 4.3 gives an overview of how the proposed technique using local nested loop join works when we have obtained the top 3 records from 2 different servers to be matched and joined together. The intention is to loop through the records and get a qualified match and rank them in order to obtain the top-3 records.

By looping through records we first match '*A*' from Server *R* with '*B*' from Server *S* and since it does not match, we move on to matching '*A*' and '*C*' and again it does not match, so we move on to match '*A*' and '*A*' and now it matches, so we will keep it as one qualified match. We repeat the same thing again until we get three qualified matches. From the above example, the top-3 will be '*A*','*B*','*C*' because all these records exist in the two different servers. Now we get the top-3 we need to rank them.

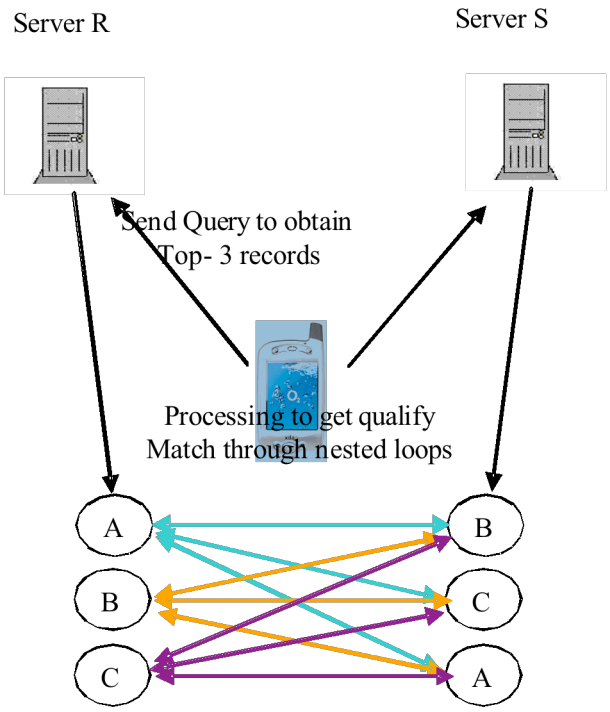


Figure 4.3: Data Source Sorted with Nested Loop

Nonetheless, there are two major pitfalls of this approach. The first pitfall is the issue of finding a match immediately. This is because, since we get only the top-3 movies from each server as the user has requested by issuing the mobile query, there is always a possibility that we might not get the exact top-3 to be joined. And if this happens, we will need to repeat the process in order to obtain the next 3 records and then check for re-joining again with the previous blocks of top-3 including the next current new 3-record block. And now with more and more data coming in, using the nested loop may no longer be as efficient as it was previously.

Figure 4.4 shows that if the first 3 records obtained from the servers do not match, then further processing will be needed. Assuming '*A*' and '*B*' records are already matched as top 1 and top 2 records, and now we need to find a qualifying match for the third record. Record '*C*' from Server *R* does not match with record '*D*' from Server *S*. Thus, we need to download the subsequent 3 records from each server. Now we get record '*D*' as a match between Server *R* and Server *S* and therefore it

becomes one of the top 3 records. The last step now is to rank them in order to obtain the new set of top 3 by using the same average formula as we used before.

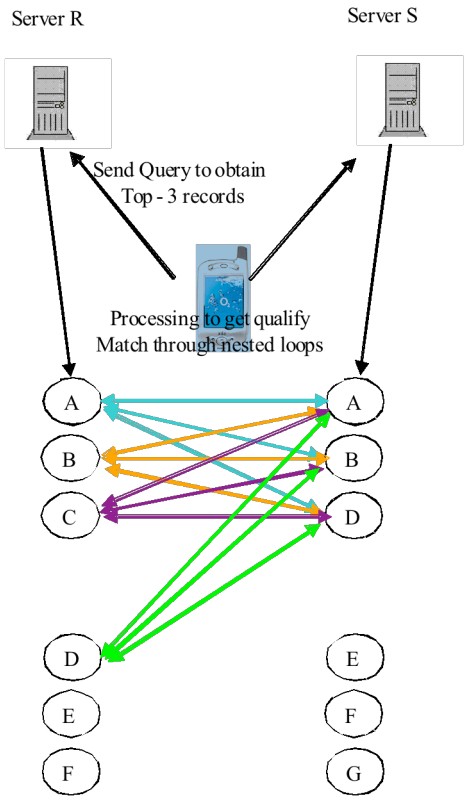
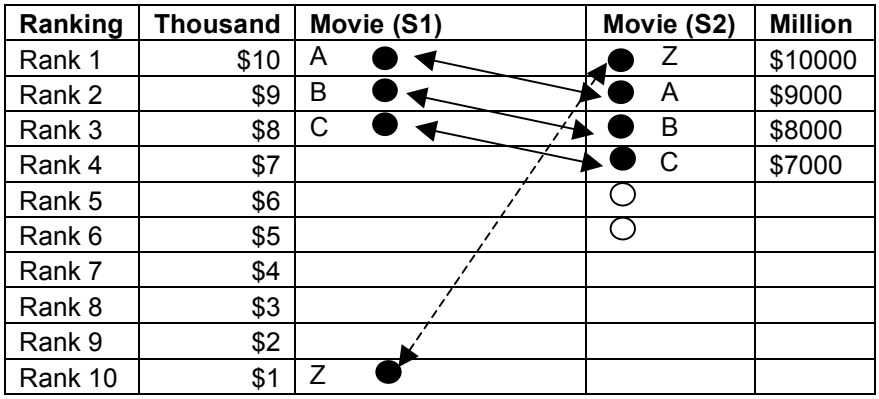


Figure 4.4: Problem of Nested Loop Join

The repetition of the nested loop takes more time and may increase the mobile cost. For that reason, the solution in this case would be to download a larger block which may mean, for example, downloading 10 records at a time rather than 3 records. The advantage of a bigger block is that there might be a good chance that all Top-k pairs will be obtained, and therefore, we do not need to download the second block. However, an obvious drawback is that we need to download a bigger block in the first place, and this incurs an overhead.

Another pitfall concerns the ranking where we assume that all downloads are done in one go, but the output result obtained may be wrong even though we have

managed to obtain our Top-k using the nested loop join. The following Figure 4.5 will explain the limitation of the approach.



Ranking	Thousand	Movie (S1)	Movie (S2)	Million
Rank 1	$10	A	Z	$10000
Rank 2	$9	B	A	$9000
Rank 3	$8	C	B	$8000
Rank 4	$7		C	$7000
Rank 5	$6			
Rank 6	$5			
Rank 7	$4			
Rank 8	$3			
Rank 9	$2			
Rank 10	$1	Z		

Figure 4.5: Another Problem of Nested Loop Join

From Figure 4.5, notice that we obtained the Top-3 which is '*A*', '*B*', '*C*' early in the process and if we average out the top 3 that were obtained at the beginning of the process, the result would be:

$A(10+9000)/2 = 4505$

$B(9+8000)/2 = 4004.5$

$C(8+7000)/2 = 3504$

However, if we stop immediately after this, then even though there may be a case where '*Z*' from S2 actually has a match from '*Z*' in S1, it will be overlooked and disregarded. And this will cause the processing of Top-k to become inaccurate because it appears that '*Z*' is supposed to have the highest average by using the average formula and should be ranked 1, even though its ranking from S1 may be the lowest.

$Z(1+10000)/2 = 5000.5$

Thus, the actual top-3 result should be as follows:

$Z = 5000.5$

$A = 4505$

$B = 4004.5$

And not

$A = 4505$

$B = 4004.5$

$C = 3504$

Therefore '*C*' should not be included in the Top-3 since its actually rank is 4 which we are not interested in. So, the proposed steps as illustrated in Figure 4.6 should be modified so that we do not stop the processing, even though in the early stage of the matching we have obtained the Top-k data. The solution is that all individual Top-k data from each server should be processed and so as to obtain a match.

Figure 4.6 demonstrates how this problem can be solved. Notice that we have already obtained the top 3 which are '*C*', '*D*',' '*E*'. However, we should not stop the matching process until all the white dots (i.e. non-matched records) on top of the matched dots do get a match or until there is no match left at all. In this example, the process should be stopped only when '*E*' from *S*1 and '*D*' from *S*2 have all previous records processed and matched.

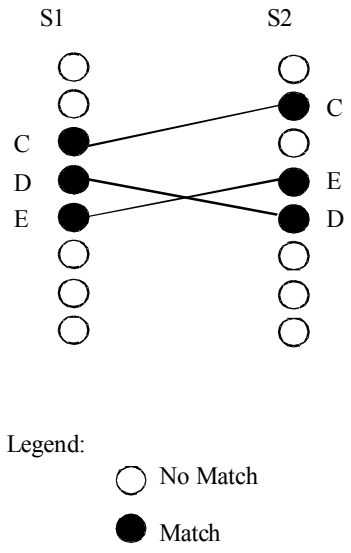


Figure 4.6: Solution to the matching problem of Nested Loop Join

Figure 4.7 shows that *S*1 has two items on top of the list that do not yet have any match and *S*2 also has two items that are yet to have a match. The idea is to ensure that any items that are above the last match must be processed. Using this method, we can ensure the Top-k results are obtained accurately by matching against all potential items without accidentally omitting any minor processing.

The algorithm of Top-k Nested-Loop Join is divided into smaller parts, each dealing with certain issues discussed above. Figure 4.8(a) show algorithm of the basic building block of the Top-k Nested Loop Join, which is taking one block from each server, and do a local nested loop join. Note that when a match is found, we need to check in particular two criteria: one is whether Top-k has been reached. If not, then the process naturally continues. If Top-k is reached, we need to check whether all previous records have matched. If not, then as explained above, the processing needs to continue, otherwise the entire process terminates.

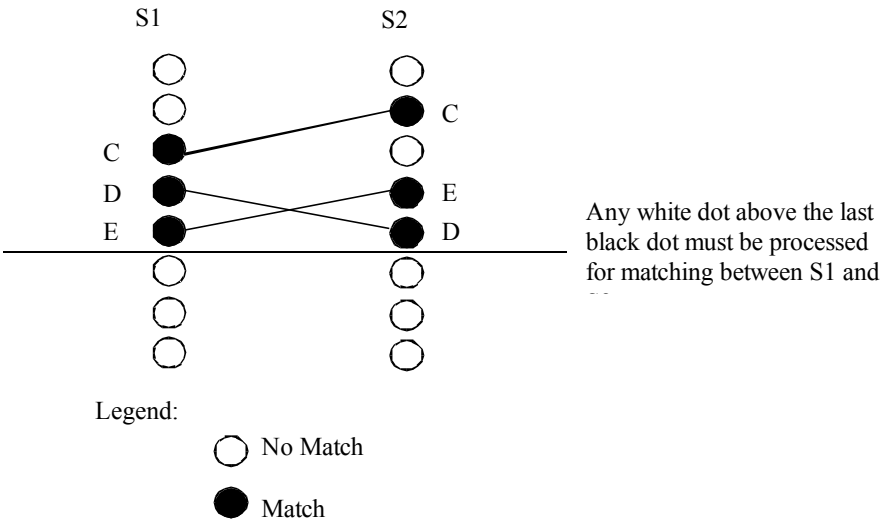


Figure 4.7: Processing all possible matches in Nested Loop

If more than one block is needed, Figure 4.8(b) shows the algorithm for the processing of multiple blocks. Therefore, the nested-loop is done at a record level at each block and at the block level when multiple blocks are processed.

```
Algorithm 4.1: Block Nested-Loop Join
Input: Blocks Ri and Sj
Output: Pairs of matches from Ri and Sj
1. For each record of Block Ri
2.    Read record from Block Ri
3.    For each record of Block Sj
4.       Read record from Block Sj
5.       Compare the join attributes
6.       If matched Then
7.          Calculate new combined value of the matched record
8.          Insert matched record in the Top-k result
9.          If Top-k is reached Then
10.            If all records above the current matched
               records have matched Then
11.               Terminate
12.            End If
13.         End If
14.      End If
15.   End For
16.End For
```

Figure 4.8(a): Block Nested-Loop Join Algorithm

```
Algorithm 4.2: Multiple Block Nested-Loop Join
Input: Blocks R1,2…n, and S1,2,…m
Output: Pairs of matches from R and S
1. For Block Ri = 1..n
2.    For Block Sj = 1..m
3.       Call Block Nested-Loop Join (Ri, Sj)
4.    End For
5. End For
```

Figure 4.8(b): Multiple Block Nested-Loop Join Algorithm

4.3.2 Top-k Merge Join

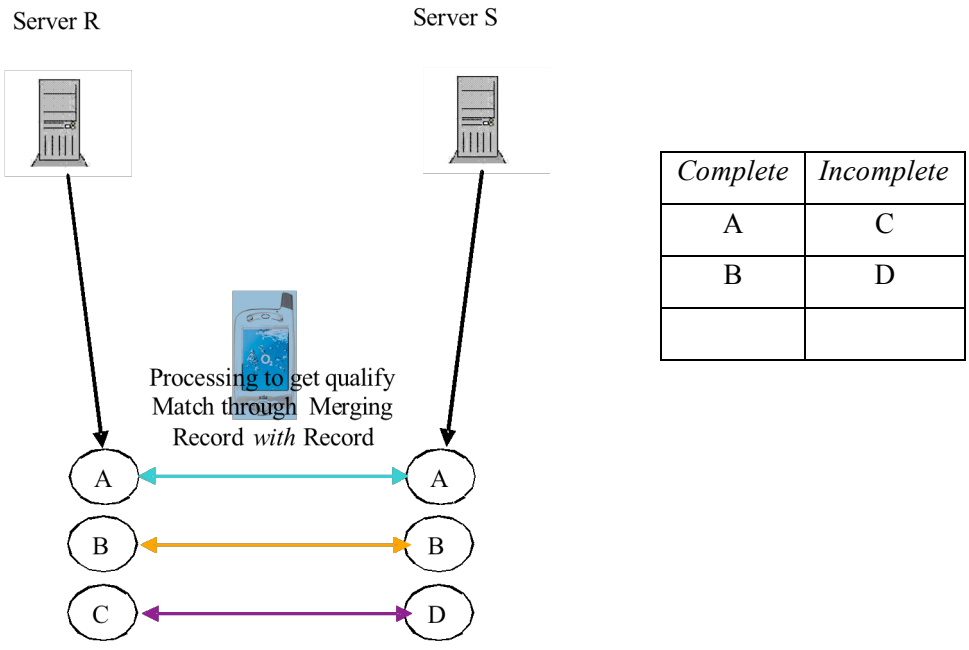
This is an alternative proposed technique for processing Top-k results which is instead of using the local nested loop to check for local matching, we will look at using the merging approach.

The main difference between this technique and the previous nested loop join technique is that in the merging process, we will compare records by merging them together one row at a time and classifying them into either the complete or incomplete list. So basically, if the first record appears to be matched from the 2 different servers

it will be categorized as *complete;* conversely, if the first record from each of the servers does not match, then it will be categorize as *incomplete*. And when another record has been sent to incomplete, it will then check within the incomplete list and if it findsa match, it will become complete.

For explanation purposes, we will incorporate block processing in this technique. Therefore, if there are not enough complete records in the first block, then it will continue to download the second block from each server and perform local matching using the merging approach to define which category the record will fall into. In determining the ranking from the complete category we will use the same average formula that we used in the first proposed technique and we will assume the records will be based on numerical attributes.

Figure 4.9 shows that if there is a match then it will be categorized in the complete list; else it will be categorized as incomplete and the next record will be re-matched and if it already has an existing match in the incomplete list, then it will be moved to the complete list.



Complete	*Incomplete*
A	C
B	D

Figure 4.9: Data Source Sorted with Merging

Since we need at least 3 records to qualify as Top-3, we still need 1 more record because in the complete list we currently have only 2 records which are '*A*' and '*B*'. Figure 4.10 shows that we download another set of 3 records to be matched and record '*D*' from Server R and Record '*E*' from Server S will go to the incomplete list. Since there is already an existing '*D*' in the incomplete list, it will become a match and therefore record '*D*' will move to the complete list.

The complete list now contains three records that satisfy our Top-3 criteria. And now having the top 3, we would like to rank them in order using the average formula as we used in our previous proposed techniques.

However, it is important to integrate the concept from the previous approach whereby we will ensure that the processing does not stop until all items on top of the last qualified match have been processed. In other words, never stop the matching process if there are still unmatched items which are above the matched Top-k. This is illustrated in Figure 4.11.

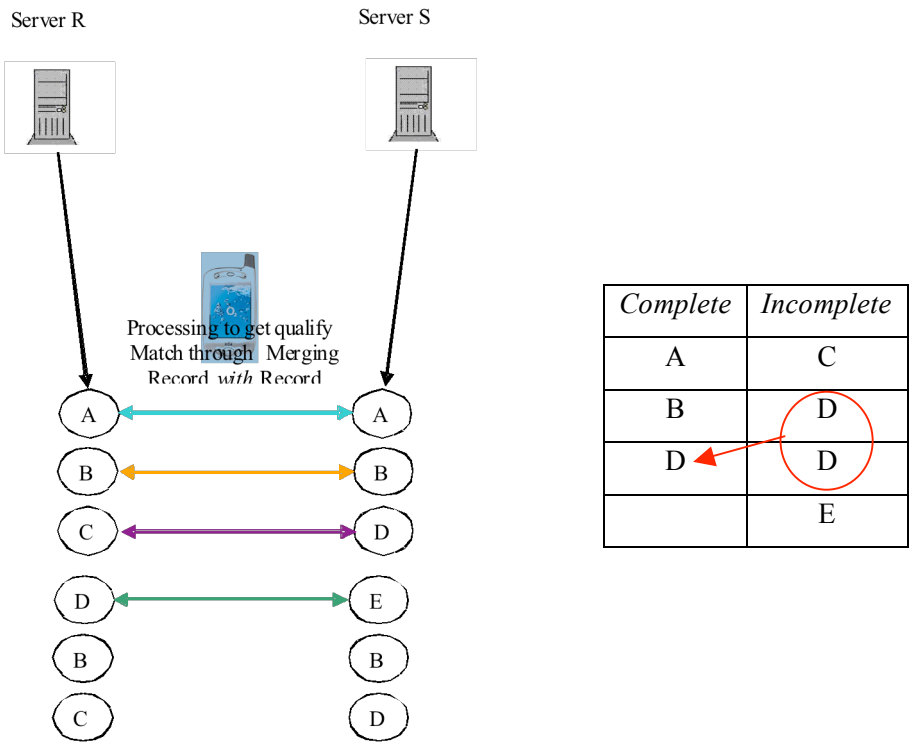


Figure 4.10: Obtaining 2nd block for Merging Process

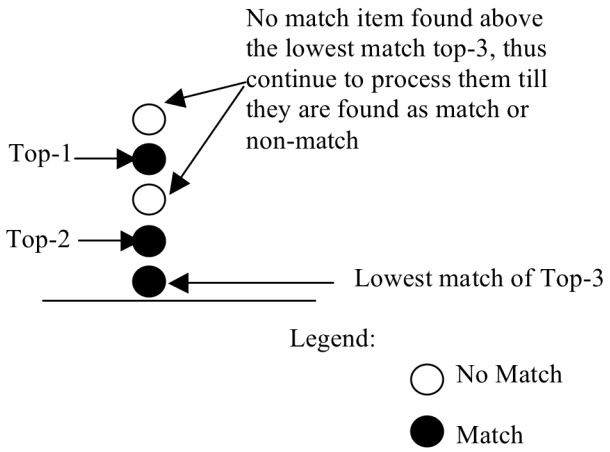


Figure 4.11: Unmatched records above the last matched records should be processed and only then does the processing stop.

In this way, this method helps to address some of the drawbacks of a nested loop, thereby increasing time and cost efficiency if there are a lot of records to be processed, and ensuring that all items have been processed, matched and placed into either the complete or incomplete lists to avoid obtaining the wrong Top-k results. Furthermore, another point is that in the merging approach, no issue arises regarding optimum block sizes since there is no need for the nested loop. This means that in performing the merging, the block size is unimportant because we are interested only in merging them and classifying them into the complete and incomplete categories.

The Top-k Merge Join algorithms are presented in Figure 4.12. Like those of the Top-k Nested Loop Join algorithms, the first part of the algorithm (see Figure 4.12(a)) deals with the actual record comparisons, which in this case is a merging process. However, note that this merging is rather different from the traditional merging, as the merging is done record by record and the advancing of the merging does not rely on whether or not a matched is found. Both records advance to the next record. The algorithm in Figure 4.12(b) is the main program which fetches each block when necessary. Each time, it calls the block merge join algorithm. If one of the blocks is already empty, then the block is nil, resulting in the other block being automatically included in the Incomplete list.

Algorithm 4.3: Block Merge Join

```
Input: Blocks Ri and Sj
Output: Pairs of matches from Ri and Sj
1. For each record k=1..n
2.    Read record from Block Ri(k)
3.    Read record from Block Sj(k)
4.    Compare the join attributes
5.    If matched Then
6.       Calculate new combined value of the matched record
7.       Insert matched record in the Top-k result which is
         the Complete list
8.       If Top-k is reached Then
9.          If all previous records are matched Then
10.            Terminate
11.         End If
12.      End If
13.   Else
14.      Search both records in the incomplete list
15.      If matched Then
16.         Calculate new combine value of the matched record
17.         Insert matched record in the Top-k result which is
            the Complete list
18.         If Top-k is reached Then
19.            If all previous records are matched Then
20.               Terminate
21.            End If
22.         End If
23       Else
24.         Put both records in the incomplete list
25.      End If
26.   End If
27.End For
```

Figure 4.12(a): Block Merge Join Algorithm

Algorithm 4.4: Multiple Block Merge Join

```
Input: Blocks R1,2…n, and S1,2,…m
Output: Pairs of matches from R and S
1. For each block k=1..n
2.    Call Block Merge Join (Rk, Sk)
3. End For
```

Figure 4.12(b): Multiple Block Merge Join Algorithm

Traditional Merging vs. Top-k Merging

Generally, there are several differences between traditional sort merge and Top-k merge, one main difference being that, in terms of the requirements, in traditional sort merge, the data must be sorted according to the join attribute that is usually numerical. The way the traditional sort merge works is that when matching between two records from two different tables, for instance Table A and Table B, where the two records are different, it is based on looking at which record contains a smaller numerical value and that will be the record that needs to be moved to the next record of the table (Elmasri and Navathe, 2007). This can be illustrated in Figure 4.13.

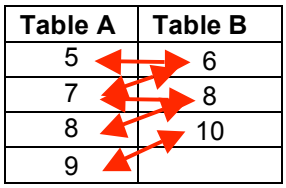


Figure 4.13: Traditional Merging Comparison

In Figure 4.13, the first record which is '5' from Table A (*A(5)*) and '6' from Table B (*B(6)*) are not the same and therefore we need to move to the next record for comparison. Since A(5) is not the same as B(6), then the pointer of Table A will move to the next record which is now A(7). The pointer reacts according to whichever record from the two tables has the smaller value. With the new pointer at A(7), we now make a comparison with the value B(6). Again there is no match, so now B(6) is smaller than A(7), so we will move to the next record of Table B which is now B(8) and compare it with A(7). Since there is no match again, and A(7) is smaller than B(8), we move to the next record on Table A which is now A(8) and compare it with B(8). Now there is a match which we are interested in. In such cases when a match has been obtained, both records from their respective tables will be moved to the next record together and will now compare between A(9) and B(10). The same process is repeatedly used in traditional sort merge.

The only assumption that needs to be made when using this approach is that there are no duplicates in the data as such when the data is primary key data. In the

situations when duplicates are involved (e.g. non-PK), then the nested loop will be used between the duplicate data itself. This can be illustrated in Figure 4.14a.

As can be seen in Figure 4.14b, value '8' occurs twice in Table *A* and Table *B* respectively and with the occurrence of these duplicate values, the nested loop is used and the results of the join between Table *A* and Table *B* using the nested loop will be as in Figure 4.14c.

Table A	Table B
5	6
7	8
8	8
8	10
9	

Figure 4.14a: Traditional Nested Loop Comparison

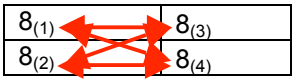


Figure 4.14b: Traditional Nested Loop Comparison for Duplicate Data

$A(8_1),B(8_3)$
$A(8_1),B(8_4)$
$A(8_2),B(8_3)$
$A(8_2),B(8_4)$

Figure 4.14c: Results of Traditional Nested Loop Comparison

However, what we are interested in this chapter is the Top-k merging approach and the way it works is different from, and less complicated than, the traditional sort merge. The data is not sorted based on the join attribute, but rather it is sorted based on the ranking. To adhere to this requirement, the traditional sort merge

cannot be used. The comparison for the qualified match between the different records in different tables is much simpler because we just need to compare them in pairs and move on to the next record in pairs when there is match or no match rather than needing to check which record has a smaller value before moving to the next record. This is illustrated by Figure 4.15.

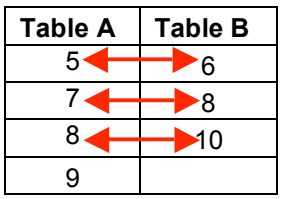


Figure 4.15: Top-k Merging Comparison

By comparing in pairs the records from both tables, such as with the first record comparing *A*(5) and *B*(6), there is no match, so we will put those two records in a table call '*Incomplete*'. Then moving to the second record for each table respectively, we get *A*(7) and *B*(8), and again there is no match, so we put them in '*Incomplete*'. The next record *A*(8) and *B*(10), again we put them in the '*Incomplete*' list. And looking at the '*Incomplete*' list, we see that *B*(8) has been placed there previously and now there is *A*(8), so we know they are a match and they will now be placed in the '*Complete*' list.

As a summary, for Top-k merging, we compare the items in pairs between the two servers, and if they match, they are placed in the '*Complete*' list and if they are not matched then they are placed in the '*Incomplete*' list. And going through the '*Incomplete*' list if there is a match, together they will be placed in the '*Complete*' list in which all records are the qualified match.

4.4 Conclusions

The proposed technique in this chapter is targeted to obtain only the most required or the so-called high ranked information called Top-k results. This is the only information we want to display on the mobile device screen. In a real example, it is inefficient to display every single output to the mobile users because the users may be interested only in certain high ranked information. Therefore, processing the Top-k results is imperative.

Thus, in this chapter, we have presented Top-k join techniques aimed at saving transfer costs through processing high ranked data only. We have investigated Top-k join methods based on nested-loop and merge approaches. The algorithm considers that the data sources are already individually ranked in each server. Our merging method is also different from the traditional merging approach.

Chapter 5

Mobile Groupby-Join Query Processing

5.1 Introduction

This chapter addresses the second part of the thesis, which examines mobile aggregate query processing. In this chapter, we focus on groupby-join query processing on a mobile device. Fundamentally, groupby-join uses an aggregate function as its core processing. Aggregate query processing is useful in certain situations especially when the query is to summarize data. Since our focus in this thesis is related to multiple non-collaborative servers in a mobile environment, it is then desirable to study groupby queries in conjunction with join as the data sources are coming from various servers.

There are two types of groupby-join queries, depending on whether or not the join attribute is the same as the groupby attribute. The main objective in any query processing involving join operation is to delay the join operation as late as possible, so that the operands of the join operation will become smaller. Consequently, the join operation will not be as expensive. We still uphold this principle.

The aim of this chapter is then to investigate how the two groupby-join queries are processed efficiently in a mobile environment incorporating multiple non-collaborative servers. For the second type of the groupby-join queries, where the join attribute is different from the groupby attribute, we are particularly interested in

optimizing the query elapsed time, by delaying the join operation. This is different from the existing approaches, whose proponents claimed that this was not possible. Our work in this thesis proves the opposite, and subsequently performance gain can be expected.

Since this chapter focuses on these two types of groupby-join queries, the rest of this chapter is organized as follows. First, we provide an overview of the groupby-join queries in general, including the two different types of groupby-join queries (Section 5.2). Sections 5.3 and 5.4 present a detail discussion of the two types of groupby-join queries, including our proposed algorithms and some walkthrough examples. Finally, Section 5.5 gives the conclusion and sums up the contribution.

5.2 GroupBy-Join Queries: An Overview

As a further extension of the aggregate-based block join processing described in the previous chapter, we study a different kind of aggregate queries where join is also involved. Previously in Chapter 3, in our discussion of mobile join using aggregate block-based processing, we looked at the use of aggregation in mobile query processing. In this chapter, we extend the work by including join operations, and hence this query type is known as groupby-join queries. So, the main difference between this chapter and Chapter 3 is that in Chapter 3, the aggregate function is used to help the join queries by accommodating a block-based processing based on certain groups. In this chapter, the groupby operation is explicitly stated in the query itself through the `Group By` clause.

SQL queries in the real world are replete with group-by clauses and join operations. It is common for a GroupBy query to involve multiple tables. These tables are joined to produce a single table, and this table becomes an input to the group-by operation. We term these kinds of queries *GroupBy-Join* queries, that is, queries involving join and group by. For simplicity of description and without loss of generality, we consider queries that involve only one aggregate function and a single join.

Since two operations, namely group-by and join operations, are involved in the query, there are two different methods for executing the queries.

The first option is where the groupby attribute is same as join attribute (Taniar and Rahayu, 2001). This can be illustrated as in SQL command below where groupby attribute *Attr_C* is the same as the join attribute *S.Attr_C*.

```
Select S.Attr_C, count(*)
FROM R,S
WHERE R.Attr_A = S.Attr_C
GROUPBY S.Attr_C
```

The second option is when the groupby attribute is different from the join attribute (Taniar et al., 2004) and which can be illustrated as in the SQL command below where join attribute *S.Attr_C* is not the same as groupby attribute *S.Attr_D*.

```
Select S.Attr_C, count(*)
FROM R,S
WHERE R.Attr_A = S.Attr_C
GROUPBY S.Attr_D
```

These two types of groupby-join queries will be further described in more detail in the following two sections respectively.

5.3 GroupBy-Join (where Group-By Attribute = Join Attribute)

As the name suggests, the proposed technique works only when the join attribute is also one of the group-by attributes. Here, we are going to study how MDSP and SSP can be applied to groupby-join queries where the groupby attribute is the same as the join attribute.

Because the join and groupby attributes are the same, it then makes sense to do the grouping first so that the join operation will take the results of the grouping which is then joined with the other table. Because the groupby is carried out before the join, which is highly desirable from the query optimization point of view, we call this type of groupby-join queries "Groupby-Before-Join" queries. Note the word

'before' in between the groupby-join, indicating the order of the operations to be performed in executing the queries.

In the next sections, we study the use of MDSP and SSP in Groupby-Before-Join in a mobile environment incorporating multiple non-collaborative servers.

5.3.1 MDSP Based GroupBy-Before-Join

The proposed technique can be implemented on top of MDSP or SSP. The following will look at using MDSP as the base of the proposed technique which means heavy processing will rely locally on the mobile device itself. This implies that the results of queries from both servers will be downloaded to the mobile device for local processing. The algorithm for MDSP based Groupby-Before-Join is presented in Figure 5.1.

The algorithm in Figure 5.1 processes the groupby-before-join queries where the first table having the join attribute is located in Server 1, and the second table having the groupby attribute (which is also the join attribute) is located in Server 2. The local processing in a mobile device performs the final join operation.

Assuming that the groupby-before-join query is interested only in the aggregate of the group by attribute, the information downloaded from both servers is already adequate to be processed in the mobile device. The information from Server 1 is the join attribute values, whereas the information from Server 2 is the aggregate values of each group by attribute value. Figure 5.2 gives an example of the algorithm.

Algorithm 5.1: MDSP Groupby-Before-Join

```
Input:
   Server 1 table (R) having the join attribute
   Server 2 table (S) having the join/groupby atribute
1. Send query 1 to Server 1 (R) to obtain all join attributes
   Select join_attr into R
   From Table_R
   Order By 1;
2. Send query 2 to Server 2 (S) to obtain counts of all groups
   Select groupby_attr, count(*) into S
   From Table_S
   Group By groupby_attr
   Order By 1;
3. Local join processing is carried out in a the mobile device
   using a merging process
4. Store the matched results into Temp
```

Figure 5.1: MDSP based Groupby-Before-Join Algorithm

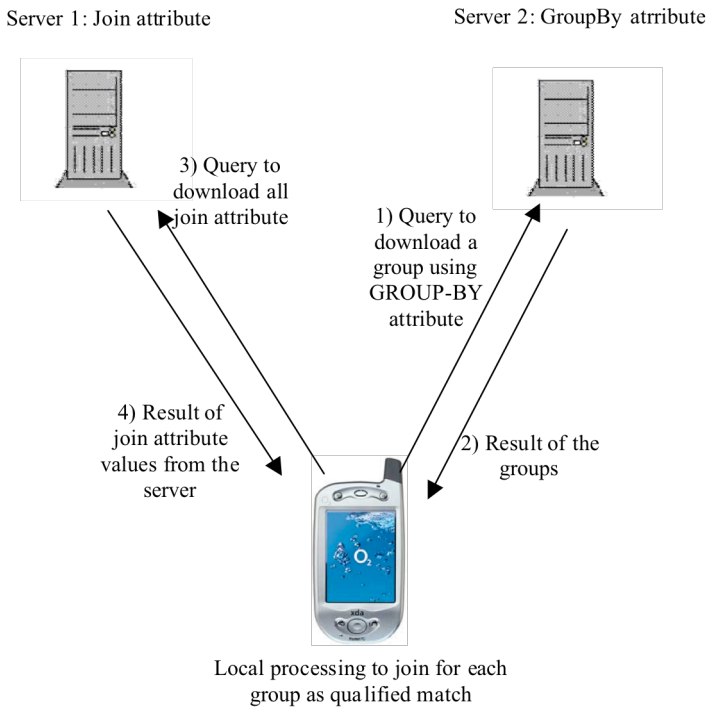


Figure 5.2: MDSP Groupby-Before-Join Technique

5.3.2 SSP Based GroupBy-Before-Join

Besides processing locally on the mobile device, the SSP model can also be incorporated whereby the processing power will be distributed to the server, rather than having the main joining process being done on the mobile device. There is a slight difference between the MDSP-version and the SSP-version for the GroupBy-Before-Join query processing. In the SSP version, the request to download is being sent only to the server having the groupby attribute (e.g. in this case the second server), whereas in the MDSP version, the request is sent to both servers. In the SSP version, join processing is done in the server not having the groupby attribute (e.g. in this case the first server), whereas in the MDSP version, the join processing is done locally on the mobile device. Figure 5.3 gives the algorithm for SSP based Groupby-Before-Join processing.

Algorithm 5.2: SSP Groupby-Before-Join

```
Input:
   Server 1 table (R) having the join attribute
   Server 2 table (S) having the join/groupby atribute
1. Send a query to Server 2 (S) to obtain counts of all groups
   Select groupby_attr, count(*) into S
   From Table_S
   Group By groupby_attr
   Order By 1;
2. Send the groups result obtained from Step 1 above to
   Server 1 (R) to be joined
3. Server 1 join the groups obtained from the query in Step 2
   above with their own local data.
4. Send the matched results to the mobile device
```

Figure 5.3: SSP based Groupby-Before-Join Algorithm

The above algorithm processes the groupby-before-join queries where the first table having the join attribute is located in Server 1, and the second table having the groupby attribute (which is also the join attribute) is located in Server 2. The final join processing is carried out in Server 1 having the join attribute only.

Apart from the location of the join processing, the way the join operation is carried out might also be different. In the MDSP version, since the data downloaded

from the two servers are already sorted, a merge operation can be conveniently carried out in the mobile device. In the SSP version, since the join operation is performed using more powerful computing and storage resources, it is likely that the server may choose to use a hash join algorithm. Figure 5.4 illustrates this algorithm.

In summary, to process the GroupBy-Before-Join queries, we first obtain the group using the group by attribute, and then only join it with the other data source to get the qualified matches. The MDSP and SSP versions indicate only where the final join processing is carried out.

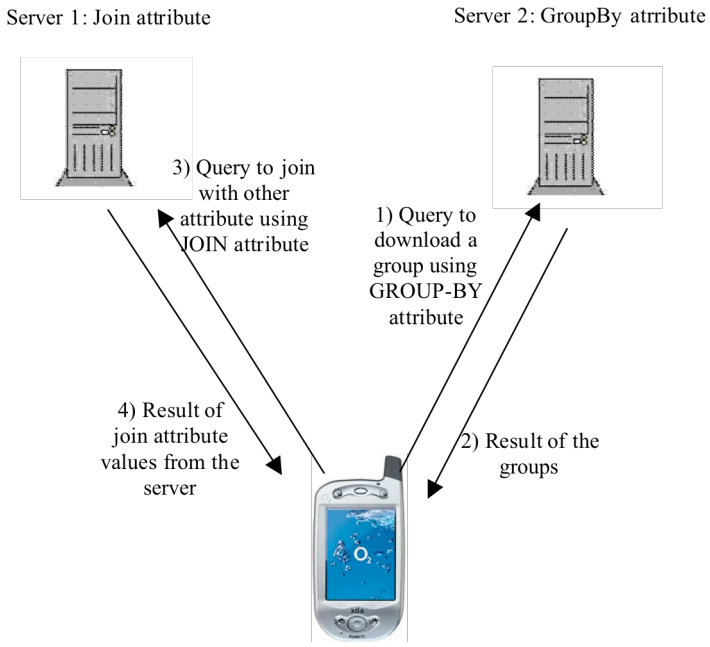
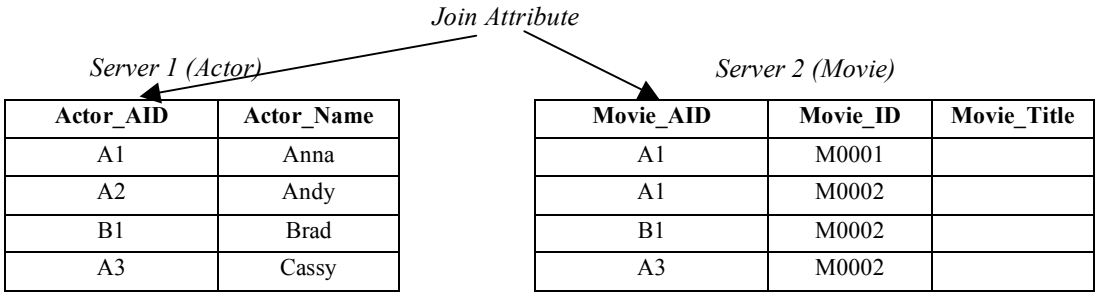


Figure 5.4: SSP based Groupby-Before-Join Technique

5.3.3 A Walkthrough Example

In order to demonstrate this version of the proposed technique, the following provides a comprehensive step-by-step walkthrough. Our aim is to find the number of movies of each actor that can be obtained from Table *Movie* together with the actor's name which can be obtained from Table *Actor* using the following sample data in Figure 5.5 obtained from two different servers.



Actor_AID	Actor_Name
A1	Anna
A2	Andy
B1	Brad
A3	Cassy

Movie_AID	Movie_ID	Movie_Title
A1	M0001	
A1	M0002	
B1	M0002	
A3	M0002	

Figure 5.5: Sample Data

Our aim is to obtain the count for the numbers of movies that the actors acted in which can be obtained from table Movies with viewing joining it with the Table Actor using the join attribute of Actor_AID and Movie_AID with the groupby attribute as Actor.AID. The following SQL Query is being issued:

```
SELECT Actor.AID, Actor.Name, COUNT(*)
FROM Actor, Movie
WHERE Actor.AID = Movie.AID
GROUP BY Actor.AID, Actor.Name
```

Note that the join attribute is the same as the group-by attribute which is Actor_AID.

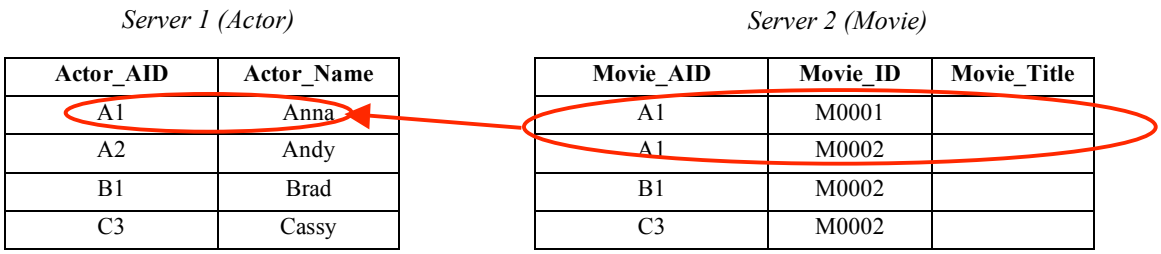
In the first step, we will first group by the Actor_AID from the table 'Movie and make a comparison with the table Actor and join it to obtain the name of that actor and return the output by using the aggregate function count to obtain the count of the number of movies in which that actor appeared.

Thus, for instance, the first group obtained from the table 'Movie would be Actor ID 'M0001' who acted in 2 movies of Movies 'M0001' and 'M0002' that will

return a count of 2. Next, with the first group that was obtained from using the GROUP BY function, we would compare it with the table 'Actor' and join it together with the actor name of Actor ID 'A1' which is 'Anna' as in Figure 5.6, and Figure 5.7 shows the output.

The above process is repeated for all groups obtained from Server 2, which in this case are: B1(1) and C3(1). The number in the brackets shows the count for the group. The joining process is then carried out as explained in Figure 5.6 above.

Since the main difference between the MDSP and SSP is only the location of the join operation, there is no need to further elaborate using an example. The example above already demonstrates how the group by operation is carried out before the join operation, with the primary aim of reducing the workload of the join operation.



Actor_AID	Actor_Name
A1	Anna
A2	Andy
B1	Brad
C3	Cassy

Movie_AID	Movie_ID	Movie_Title
A1	M0001	
A1	M0002	
B1	M0002	
C3	M0002	

Figure 5.6: Joining Matching Actor between Servers 1 and 2

Actor_ID	Actor_Name	Count(*)
A1	Anna	2

Figure 5.7: Final Output

5.4 GroupBy-Join (where Group-By Attribute ≠ Join Attribute)

In this version, the condition is that the groupby attribute is different from the join attribute. In the literature review, this kind of query is called "Groupby-After-Join" indicating that the join operation has to be carried out first, and then followed by the groupby. This turns out to be inefficient. In our case, we argue strongly that we need to do the groupby first and then followed by the join operation, whenever possible. Hence, we will no longer call this Groupby-After-Join.

Our proposal for processing this type of query is divided into two types:

- Double grouping
- Single grouping

The details will be explained in the next sections. But briefly, we can say that the double grouping method is where the grouping is done twice. The first grouping is done first followed by a join, and then finishes off with the second grouping. On the other hand, the single grouping is where the grouping is done first, and then followed by the join.

We need to emphasize two things: one is that the single grouping is different from the Groupby-Before-Join method explained in the previous section. The reason is that in the previous section, the join and groupby attributes are the same, whereas in this section, the join attribute is different from the groupby attribute. Consequently, the processing is different.

The second difference is that the method to be used, whether it be double grouping or single grouping, depends on the structure of the query itself. One thing that is certain is that in this section we deal only with the query where the join attribute is different from the groupby attribute. Secondly, we need to look at the relationship between the join and the groupby attributes. If there is a 1-many

relationship between the join and the groupby attributes, then we do the double grouping. Otherwise, we need to do the single grouping.

So, in short, we need to choose the correct method by considering the relationship between the join and the groupby attributes. In the next two sections, we will elaborate on these two methods: double grouping and single grouping, and then follow this with a walkthrough example.

5.4.1 Double Grouping Method

Consider the following Groupby-Join query:

```
Select S.groupby_attr, aggregate_func()
From Table R, Table S
Where R.join_attr = S.join_attr
Group By S.groupby_attr
```

In this query, the groupby attribute is in Table *S* identified by attribute name *groupby_attr*, whereas the join attribute from both tables is named *join_attr*. Note that the join attribute is different from the groupby attribute.

If the groupby attribute groupby_attr has a 1-*many* relationship with the join attribute join_attr, then a "Double Grouping" method can be applied. In this case, we strongly argue that we can do a grouping before the join, and hence it is NOT Groupby-After-Join anymore, but a Groupby-Before-Join. Because the term Groupby-Before-Join has been used in the previous section, where we focus on the Groupby-Join query where the join and the group by attributes are the same, in this section, we will not use the same term - we will use the term "Double Grouping", instead.

As mentioned previously, the Double Grouping method applies the grouping operation twice. After the first grouping, the join is carried out, and then finishes with a second grouping. As we can see here, the grouping is done before the join, and hence our claim that it is a groupby-before-join is still valid.

We also need to emphasize one more thing. In the previous method (namely Groupby-Before-Join), we introduced two versions, one based on MDSP, and the other based on SSP. For the Double Grouping and Single Grouping presented in this

section, we will combine the MDSP and the SSP, as some processing will be done in the server and rest in the mobile device.

The algorithm for the Double Grouping is presented in Figure 5.8. Note that the algorithm for the double grouping is very similar to that of the SSP-based Groupby-Before-Join, with one exception that there is a final grouping done in the mobile device.

Algorithm 5.3: Double Grouping

```
Input:
   Server 1 table (R) having the join attribute
   Server 2 table (S) having the join and groupby atributes,
                      which are different.
1. Get a group of records from the join attribute from
   Server 1 (R)
2. Send the groups to Server 2 (S) to be joined
3. Perform an aggregate function based on the groupby attr
4. Repeat steps 1-3 until all groups in steps 1 are processed
5. Send the results from step 3 to the mobile
6. The mobile will consolidate the groups by doing a second
   grouping operation
```

Figure 5.8: Double Grouping algorithm

The walkthrough example to be presented in Section 5.4.3 will clarify how the algorithm works using an example. However, in this section, we need to emphasize the difference between the SSP-based Groupby-Before-Join and the Double Grouping method. As mentioned earlier, the Double Grouping method has one main constraint, that is the groupby attribute with the join attribute has a 1-*many* relationship. It means that one groupby attribute value has a corresponding *many* join attribute values. If we then group the join attributes based on the groupby attributes, each groupby attribute will have many join attribute values. Then after that, a second grouping consolidates the grouping of the groupby attributes.

For example, if a groupby attribute value ‘A’ has several join attribute values ‘A1’, ‘A2’, ‘A3’, etc, and another groupby attribute value ‘B’ has several join attribute values ‘B1’, ‘B2’, ‘B3’, etc. Then it is possible to produce multiple groups of the join attributes, such as ‘A1’, ‘A2’, ‘A3’, ‘B1’, ‘B2’, ‘B3’, etc. Then the second

grouping will then consolidate all the As and all the Bs into one group. This is only possible if the groupby attribute has a 1-many relationship with the join attribute. A case study will further elaborate the Double Grouping method. This will be explained in Section 5.4.3.

5.4.2 Single Grouping Method

The Double Grouping described in the previous section has one condition where the join attribute value is associated with one groupby value, meaning that the groupby attribute contains the join attribute values. In this case, when grouping the join attribute values, for each groupby attribute value, we can easily identify one or more join attribute values.

If such a constraint does not hold, then the Double Grouping method cannot be applied. Hence, in this case, we need to use a Single Grouping method. The way the Single Grouping method works is that the grouping (like the block-based on Chapter 3) is done by the table not having the groupby attribute. The grouping is similar to the block-based method. In other words, there is no reduction in the number of records (not like the Double Grouping), but the grouping is carried out as a block-based only. Although there is no reduction in number of records, it is still beneficial to have a block-based method.

The way the Single Grouping works is as follows: firstly, group the non-groupby table based on one attribute. This group is then sent to the other server to be joined. Once the joined has been done, the aggregate function is applied. In other words, the Single Grouping method is a manifestation of the block-based approach as explained in Chapter 3. It is not a means of record number reduction. As a result, in terms of its performance, we can easily expect that the Double Grouping will deliver a much better performance than the Single Grouping.

Figure 5.9 gives the algorithm for the Single Grouping method. Note that the algorithm for the Single Grouping is similar to that of the Double Grouping, but without the second grouping.

```
Algorithm 5.4: Single Grouping
Input:
   Server 1 table (R) having the join attribute
   Server 2 table (S) having the join and groupby atributes,
                      which are different.
1. Get a group of records from the join attribute from
   Server 1 (R)
2. Send the groups to Server 2 (S) to be joined
3. Perform an aggregate function based on the groupby attr
4. Repeat steps 1-3 until all groups in steps 1 are processed
5. Send the results from step 3 to the mobile
```

Figure 5.9: Single Grouping algorithm

5.4.3 Walkthrough Examples

In this section, we will present case examples of how the double grouping and single grouping methods work.

In order to demonstrate the first scenario of this technique which involves double groupby processes, our aim is to get the count of the numbers of movies from a particular country which can be obtained from Table Actor and Table Movie respectively by using the following sample data obtained from 2 different servers of Actor and Movies. The following sample data as in Figure 5.10 will be used.

Server 1 (Actor)

Actor_AID	Actor_Name	Actor_Country
A1	Anna	USA
A2	Andy	HK
B1	Brad	USA
C3	Cassy	AUS
C4	Chan	HK
C5	Tom	HK
C6	William	NZ

Server 2 (Movie)

Movie_AID	Movie_ID	Movie_Title
A1	M0001	
A1	M0002	
A1	M0004	
A2	M0002	
A2	M0001	
B1	M0005	
B1	M0001	

Figure 5.10: Sample Data for Double Grouping Method

Thus, our aim is to get the count of the numbers of movies from a particular country and where the query containing the join attribute differs from the groupby attribute which is according to the SQL Query the groupby attribute is Actor.Country and the join attribute is Actor.AID. The following SQL Query is being issued:

```
SELECT Actor.Country, COUNT(*)
FROM Actor, Movie
WHERE Actor.AID = Movie.AID
GROUP BY Actor.Country
```

Note that the join attribute is different from the group-by attribute which is Actor_AID and Actor.Country respectively.

In this version of groupby where the join and groupby attributes differ, we will first group by the actor ID (Movie_AID) in the table Movie in which we will have records of 'A1' as 3 since there are 3 'A1' in the table Movie (see Figure 5.11).

After obtaining the count of the groupby of the AID, the next step is to join it with the table Actor so that we can obtain the other attributes from table Actor which is the Actor_Country for the associated Movie_AID. Therefore, we would send the Movie_AID of 'A1' to the table Actor and join it to get the value of Actor_Country which is USA that is associated with actor ID of 'A1' (refer to Figure 5.12).

Repeat the same thing for the next Movie_AID from the table Movie which is 'A2'. The count return for 'A2' is two records which means 'A2' has two movies in table Movie. And therefore again we will send the 'A2' that is group together from table Movie to the table Actor to match with the Actor_AID 'A2' to get its Actor_Country value which is 'HK' (see Figure 5.13).

Movie_AID
A1
A1
A1

A1 = 3 records

Figure 5.11: Grouping the First Group of Movie_AID

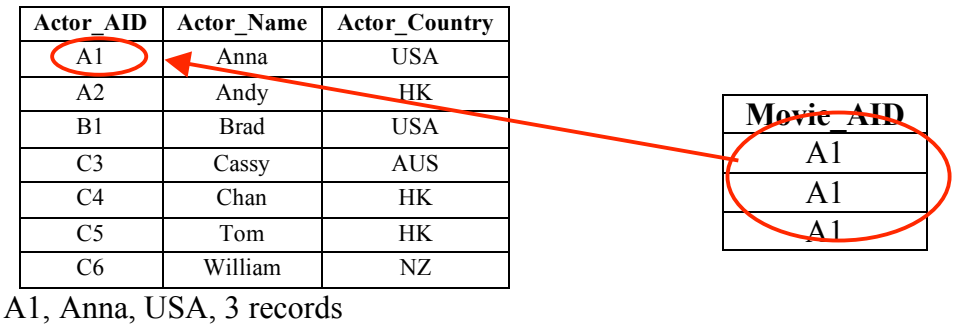


Figure 5.12: Joining Matching Actor between Servers 1 and 2 for the first group

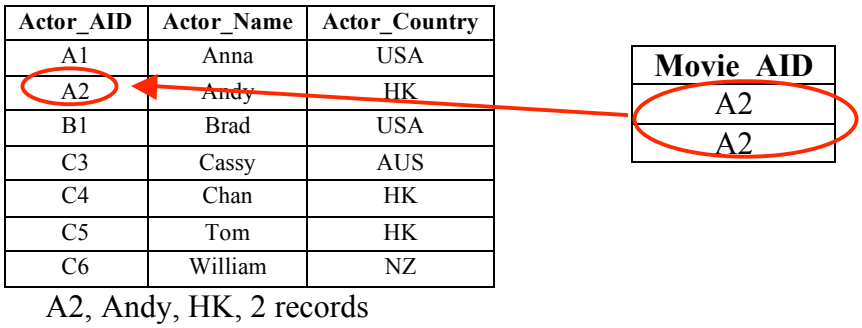


Figure 5.13: Joining Matching Actor between Servers 1 and 2 for the second group

Now, again repeat for the next Movie_AID which is 'B1'. The count return is two records meaning 'B1' has two movies in table Movie. Then again, we will send the 'B1' that has been count to the table Actor to match with Actor_AID 'B1' to get its Actor_Country value which is USA. This value is same as Actor_AID 'A1' which is 'USA'.

This is repeated until there are no more groups to be compared, and all those temporary results would have been stored in the mobile memory. We will have something similar to:

A1, USA, 3 records

A2, HK, 2 records

B1, USA, 2 records

And with all this data in the mobile device, we will process it locally by combining the information based on the country to get a final result of the count for the number of records for each separate country. This shows the additional local processing of groupby COUNT which returns the final output as follows:

Actor_Country	Count(*)
USA	5
HK	2

Figure 5.14: Final Output

Note that although the join attribute is not the same as the group by attribute, the group by attribute actually appears as a member of the join attribute whereby the group by value is associated with one join value. For example,

USA contains A1, B1

HK contains A2

The steps of processing can be summarized as follows:

Step 1: Download the join attribute from server 2 which is the Actor_ID using the GROUP BY function to group the Actor_ID from the Table Movie to the mobile device including the COUNT function.

Step 2: After obtaining the count of movies for particular Actor_ID, send the results of the count for that actor to Server 1 to process the join based on the join attribute Actor_ID to obtain the country of the actor.

Step 3: Locally process to group all related records based on the groupby attribute which in the above example is the Actor_Country where the additional local processing is to COUNT the final groupby to group them toghether.

Figure 5.15 shows the process of the GroupBy-Join based on the Double Grouping method.

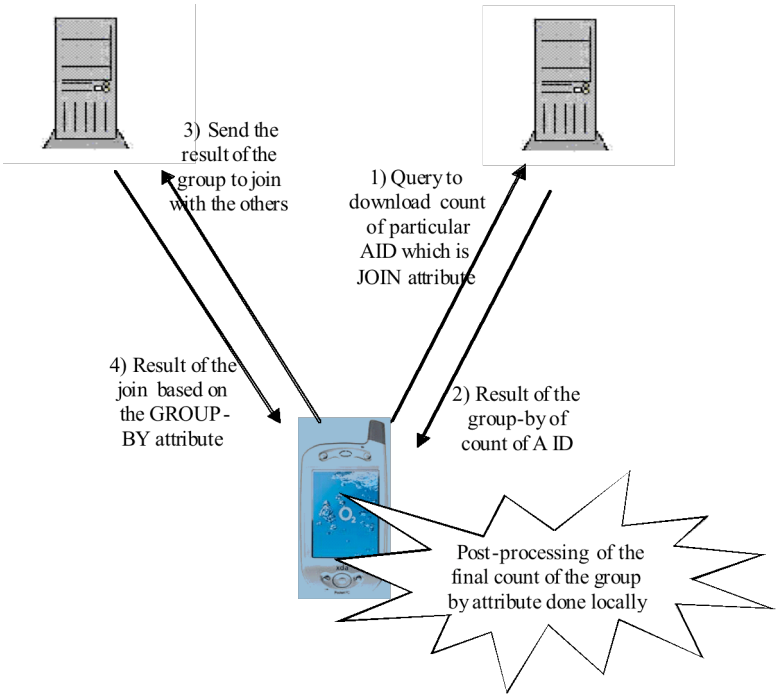


Figure 5.15: Double Grouping

The second scenario involves only a single grouping. We use the sample data presented in Figure 5.16.

Server 1(Actor)

Actor_AID	Actor_Name	Actor_Gender
A1	Anna	F
A2	Andy	M
B1	Brad	M
C3	Cassy	F
C4	Chan	M
C5	Tom	M
C6	William	M

Server 2(Movie)

Movie_AID	Movie_ID	Movie_Title
A1	M0001	
A1	M0002	
A1	M0004	
A2	M0002	
A2	M0001	
B1	M0005	
B1	M0001	

Figure 5.16: Sample Data for Single Grouping

In this example, our aim is to get the count of the numbers of actors and actresses from table Actor in each different movie. The actors and actresses are identified by their genders. The query can be expressed in the following:

```
SELECT Actor.AID,Actor.Gender COUNT(*)
FROM Actor, Movie
WHERE Actor.AID = Movie.AID
GROUP BY Actor.Gender
```

We will first group by the movie ID (Movie_ID) in the table Movie and based on the result as Figure 5.17, there are four different groups and in each group of movies, there may or may not be several actors/actresses.

Based on the above results of the groupby of the movie_ID, we now send it to the table Actor to be matched with the Actor_AID in table Actor to get their relevant genders by using the join attribute as in Figure 5.18.

After obtaining the first group by Movid_ID which is M0001, there are three actor/actress which are match against the table Actor in order to obtain the genders of the artists for that particular movie. Hence, by matching the M0001 Movie_ID with the table Actor to join them to obtain the gender, the following result as in Figure 5.19 is obtained.

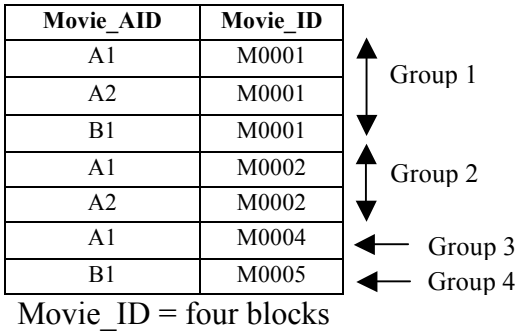


Figure 5.17: Four Different Blocks in S2

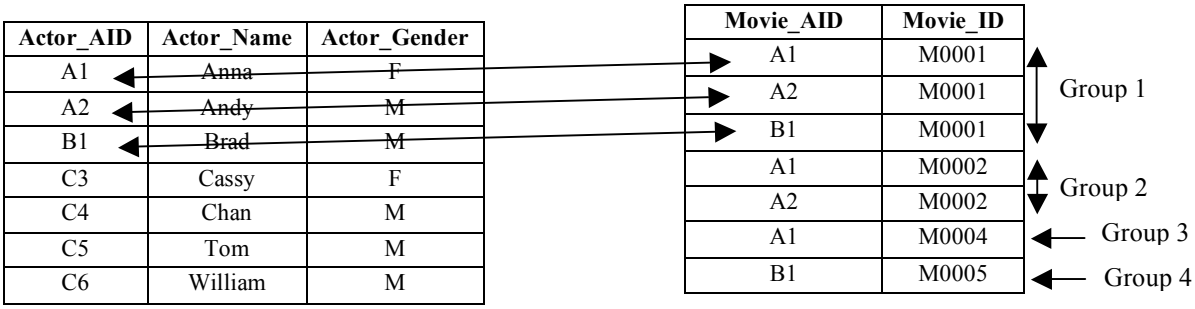


Figure 5.18: Performing Join for the 1st Group with same AID in Server 1

Actor_Gender		Movie_ID
F		M0001
M		M0001
M		M0001

Figure 5.19: Output of Join for the 1st Group

Repeat the same process the next block groupby of Movie_ID which is M0002. The results obtained would be as in Figure 5.20.

Actor_Gender		Movie_ID
F		M0002
M		M0002

Figure 5.20: Output of Join for the 2nd Group

After having repeated the above process until all movie_IDs have being grouped and compared for a qualifying match, the output would be as in Figure 5.21.

Actor_Gender	Movie_ID
F	M0001
M	M0001
M	M0001
F	M0002
M	M0002
F	M0004
M	M0005

Figure 5.21: Output for all the four Blocks

The last step is to count and groupby the gender that is within each block of movie_ID which produces results as in Figure 5.22.

Actor_Gender	Count(Gender)	Movie_ID
F	1	M0001
M	2	M0001
F	1	M0002
M	1	M0002
F	1	M0004
M	1	M0005

Figure 5.22: Final Output for all the Four Blocks in groups of Gender

5.5. Conclusions

Basically, the proposed technique presented in this chapter is an extended version of using MDSP and SSP to include an additional function which is the aggregate function. The aggregate function is deemed to help process the records into a whole certain value which maybe useful at times when the mobile users are only interested in obtaining for instance a count of certain records instead of viewing the value of the records themselves. Therefore, the aggregate function is meant to increase the options of mobile processing and help satisfy users' queries.

Thus, in this chapter, we have presented techniques for processing queries using group by joins to optimize the query processing strategy and adapt the query processing technique in the mobile environment. In each version of the techniques, there are sets of sub-techniques that have been proposed to apply to situations where different circumstances encountered.

In this chapter, we focused on two different types of Groupby-Join queries depending on whether or not the join attributes are the same as the groupby attributes. This raises the question of which operation will be carried out first, the groupby operation or the join operation. We have studied how these two types of queries can be carried out in a mobile environment involving multiple non-collaborative data sources.

The unique contribution of this chapter relates to the second type of the Groupby-Join queries where the join attribute is different from the groupby attribute. Traditionally, this type of query requires the join operation to be done before the groupby operation. However, in this chapter, we have demonstrated that it is possible to perform the grouping operation first before the join operation. We have proposed two techniques in this context: Double Grouping and Single Grouping. The efficiency of these techniques will be evaluated in the Performance Evaluation chapter in Chapter 7.

Chapter 6

Mobile Division Query Processing

6.1 Introduction

Apart from investigating the mobile join, Top-k join and groupby-join query processing in the mobile environment, it is crucial to look at another existing type of query, namely the division queries. This chapter investigates division queries in the mobile environment by resolving the current issues that are faced by the traditional division query processing. The main aim is to provide a comprehensive solution as well as addressing the shortcomings of the traditional division techniques.

This chapter is organized as follows. Section 6.2 provides an overview of the division queries followed by Section 6.3 which introduces the first proposed algorithm to address the minor issue of traditional division techniques by looking at new techniques that can be adapted to the mobile environment. Section 6.4, which will be our main contribution in this chapter, looks at another proposed algorithm that concentrates on a more major issue: proposing division techniques for the mobile environment that are able to tackle groupings between the multiple non-collaborative servers. Finally, Section 6.5 concludes the chapter.

6.2 Division Query: An Overview

There are two different types of division queries studied in this chapter, namely: relational division and multiple group division.

The relational division query has long existed as the type of query that is powerful enough to be used in the mobile environment (Date, 1995). Basically, a traditional relational division query is a query to find attributes in one relation that satisfy *all* of a given list of criteria from another relation. Figure 6.1 illustrates a division query between two tables: table *R* and table *S* based on attribute *attr*2 of table *R* and attribute *attr*1 of table *S* whereby there exist records of the dividend table indicated by attr1 in which their attr2 has all of the records in the divisor table.

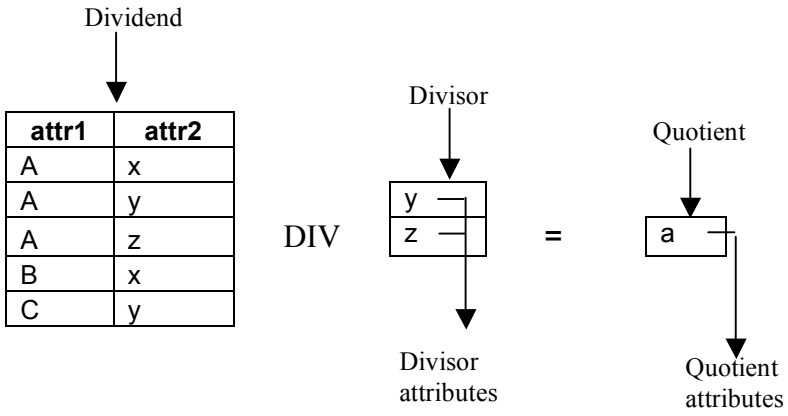


Figure 6.1: Relational division operation

Based on the similar concept of division query that relates to query requests in obtaining results that satisfy all of a given criteria in specific relation, this second type of division query is known as the *multiple group division*, The difference is that it processes a division query where both sources of data are groups. Comparing it with the relational division, notice that in Figure 6.1 the divisor contains only individual attributes that are of single value without any grouping. However, in the multiple group division, the divisor would be groups. Figure 6.2 illustrates a multiple group division query between two tables: table *R* and table *S* based on attribute *attr*2 of table

R and attribute *attr*1 of table *S* the divisor table has multiple groups of divisors as indicated by *attr2* in the divisor table.

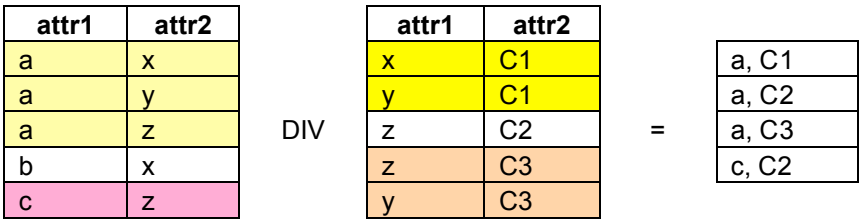


Figure 6.2: Multiple group division operation

6.3 Relational Division Techniques

The first proposed technique is to address the problem mentioned earlier in the Literature Review chapter (Chapter 2), that is, adapting the existing division query processing techniques to a mobile environment. This technique appears as an extended version of the block-based processing which incorporates the division query operation into the basic techniques of MDSP-Block and SSP-Block in the previous chapter.

Generally, mobile device side processing with division operation deals with acquiring information from one server in terms of blocks and sending it to the mobile device to be processed locally in executing the division operation.

Hence, in this version of division query processing on the mobile device side, a full list would be downloaded from one server to the mobile device and another list would be downloaded in blocks from another server and, with the two lists currently in the mobile device, the division operation will be executed to determine the qualified match.

However, another option is to use the server side processing as the base for the division operation. This is another possible extension of the proposed technique utilizing the division operation to transfer the processing to the servers instead of having it done locally on the mobile device which may use up too much memory.

Therefore, for instance, we would now download in blocks from Server 1 and send it to Server 2 for carrying out the division operation and determining the qualified match. And with the qualified match determined by Server 2, it would return the results to the mobile device.

6.3.1 Count-Based Relational Division Technique with MDSP

Now, we would like to demonstrate how the relational division works in a mobile environment. Figure 5.1 shows two tables that have been obtained from two different servers that contain different tables holding different types of records. For instance, Server 1 represents the table recording the movies in a year that are showing in respective cinemas while Server 2 represent the table that lists all movies that received awards. We will be using the following tables as a running example. Note that the data in Figure 6.3(a) is not necessarily sorted on the basis of any attribute. The sample data is used only to assist readers to understand the example.

Cinema_ID	Movie_ID
A	1
A	2
A	3
A	4
A	5
A	6
A	8
B	2
B	1
C	3
C	10
C	8
C	9
C	1

(a) Table 1 in Server 2

Movie_ID
1
2
3
6
8

(b) Table 2 in Server 2

Figure 6.3 Tables from two different Servers

In this example, our aim is to find cinemas which show all the movies in the awards list. In other words, we need cinemas that have all movies in the awards list. We know that this means that each cinema must have at least the same count as, or

greater count than, the movies in the awards list. Therefore, in this example, our first step is to obtain the count of the awards list which means we will download the count of the divisor which is from Server 2 into the mobile device. Then, we will send a query to obtain the count of the dividend for each block of cinemas using the GroupBy cinemas function. Since there are three different cinemas, there are three blocks of counts that are to be downloaded from Server 1 into the mobile device.

Now that we have the two lists of counts on the mobile device as in Figure 6.4, we need to locally compare them to check which cinema contains Count(Movie_ID) the same as, or greater than, the Count(Movie_ID) obtained from the divisor which is the awards list.

Cinema_ID	Count(Movie_ID)
A	7
B	2
C	5

Count(Movie_ID)
5

Figure 6.4: Counts from both dividend and divisor

Figure 6.4 shows that only Cinemas '*A*' and '*C*' might contain all the movies that exist in the movie awards list. Thus, we will now discard cinema '*B*' because it can never be a potential query result.

Next, we need the divisor. So we will now send a query to the Server which is the awards list to obtain the details of the awards which in this case is the actual data of the Movie_ID itself as in Figure 6.5b. We do the same thing for the dividend; we will now send a query to the Server 1 to obtain the full details record for the potential qualifying match which in this case only involves Cinemas '*A*' and '*C*' because we know Cinema '*B*' can never satisfy the division condition based on the previous count matching comparison we done as in Figure 6.5a. This shows that we have made it possible to eliminate unnecessary downloads to the mobile device.

The comparison between the dividend and the divisor data is to make a final comparison based on the Movie_ID to ascertain which cinema actually has all

Movie_ID that exist in the Movie_ID in the awards list. In Figure 6.5a we can see that Cinema_ID '*A*' contains 7 movies 5 of which exist in the awards list as in Figure 6.5b. This shows that Cinema_ID '*A*' has satisfied the division condition and therefore will be our qualified match. On the other hand, Cinema_ID '*C*' has 5 movies but only 3 of these exist in the awards list. This makes it a non-qualifying match because we are interested only in a cinema that shows all the movies in the awards list. Therefore, the results for the query asking which cinema shows all movies that received awards are as shown in Figure 6.5c.

Cinema_ID	Movie_ID
A	1
A	2
A	3
A	4
A	5
A	6
A	8
C	3
C	10
C	8
C	9
C	1

Figure 6.5a: Dividend

Movie_ID
1
2
3
6
8

Figure 6.5b: Divisor

Cinema_ID
A

Figure 6.5c: Quotient

The count-based relational division algorithm is shown in Figure 6.6. Note that the algorithm is divided into four phases.

- In Phase 1, the count request is sent to the respective servers (Dividend and Divisor Servers).
- In Phase 2, counts from the respective servers are compared. This is needed in order to eliminate unqualified quotient groups.
- In Phase 3, the actual records are requested from the respective servers. Since we are going to use a merging approach, the divisor data is requested to be sorted, and the dividend data is sorted based on the quotient attribute (primary sort) and then for each quotient attribute, the divisor attribute is sorted (secondary sort).

- In Phase 4, the local merging is carried out. Quotient table will be produced to store the division results.

Algorithm 6.1: Count-based Relational Division MDSP version

```
Input: Dividend from server R and Divisor from server S
Output: Quotient at the mobile device

// Phase 1: Count Request:
1. Send a count request to Server S (Divisor)
   Select Count(*)
   From Table_S;
2. Receive the divisor count and store it to CountS
3. Send a group count request to Server R (Dividend)
   Select QuotientAttr, Count(*)
   From Table_R
   Group By QuotientAttr;
4. Receive the group count and store it to CountR temp table

// Phase 2: Count comparison
5. For each group in CountR table
6.   If group count in CountR table NOT >= CountS Then
7.     Delete record from CountR table
8.   End If
9. End For

// Phase 3: Qualifying group download
10.Send a request to Server S (Divisor) to download the data
11.Send CountR temp table to Server R and request to send the
   actual data according to CountR
// Phase 4: Record comparison
12.For each QuotientAttr
13.  Read a record
14.  Perform merging between QuotientAttr record and Divisor
15.  If all divisor records have match Then
16.    Store the QuotientAttr into the Quotient Table
17.  End If
18.End For
```

Figure 6.6: Count-based Relational Division based on MDSP Algorithm

6.3.2 Count-Based Relational Division Technique with SSP

The only difference between the SSP-Division and MDSP-Division is where the processing takes place. Therefore, instead of having the division operation being done on the mobile device locally, we will have it done on the server. Our aim is still the same which is to get the results of a query for the cinema which shows all the movies that received awards based on the award list.

In the SSP, we could maximize our memory efficiency slightly because we would distribute some processing to the server instead of having everything done locally on the mobile device. Firstly, we would obtain the count of the divisor and then send it to Server 1 which is the dividend for processing to compare the count between the dividend and the divisor so that only the count of the dividend that is same as, or greater than, the count of the divisor will be kept as a potential qualify match. This is where the difference lies, since in the MDSP base, we would have to download both count lists but in the SSP, we download only one count list.

This process can be illustrated as in Figure 6.7. In other words, count comparison is done in the dividend server. This ends the first phase. The next phase is the actual data comparison.

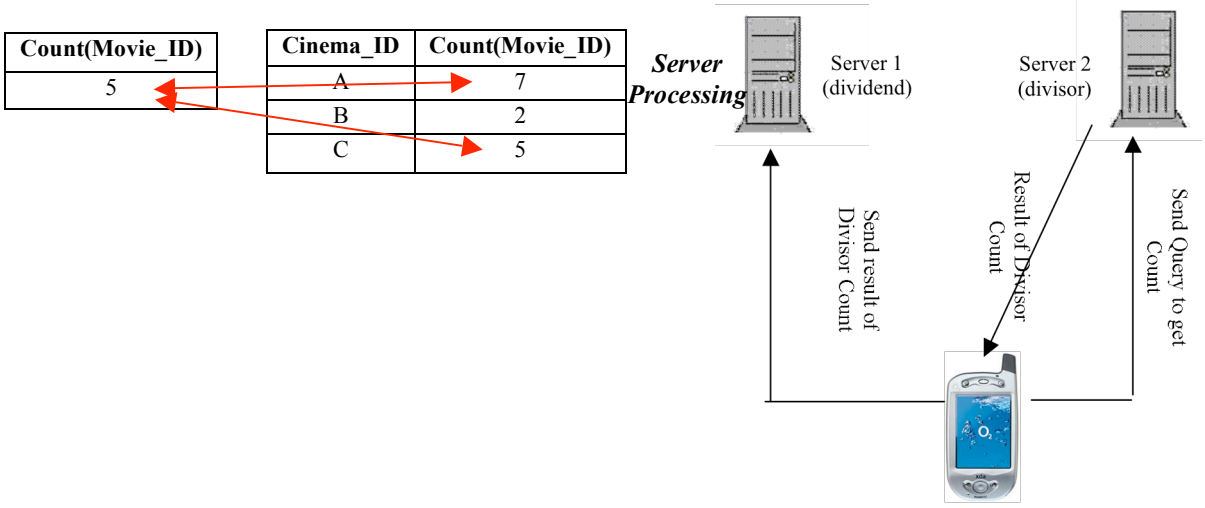


Figure 6.7: Count-comparison done in the Dividend Server

By processing and obtaining the potential qualifying match based on the count comparison, the result will now be returned to the mobile device together with its full details which are the Cinema_ID and Movie_ID. With the result on the mobile device, we now send it to Server 2 which is the divisor so that a final processing at the server to obtain the qualify match can be done. This is where the server will process the list obtained from mobile device and compare it with its award list to check all Movie_ID from the mobile device that exist in the Movie_ID in the server. Once the server has finished processing, the final output will be returned to the mobile device as the qualify match. This process is illustrated in Figure 6.8.

There is another option for processing the above division query whereby, instead of sending the dividend to the server for processing, the divisor is sent for processing. This avoids the need for block processing which is required when sending the dividend. This is because, by sending the divisor, the whole list will be sent at once rather than needing to be broken down into blocks as in the case of the dividend.

The choice of selecting whether to send the dividend or the divisor for processing in the server side will depend greatly on circumstances. For instance, if the dividend does not contain a long list that needs to be broken down into blocks, then it may be a wise choice to send the dividend. However, if the list of the divisor appears to be carrying a large amount of data, then downloading the divisor to be sent to the server for processing may not be the better choice.

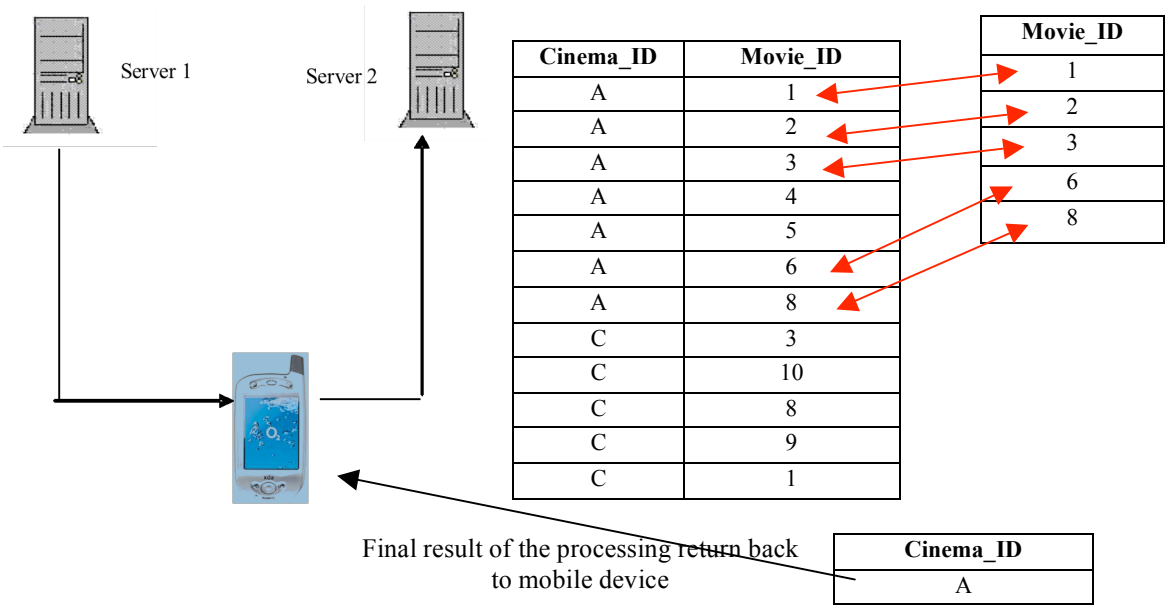


Figure 6.8: data-comparison done in the Divisor Server

Algorithm 6.2: Count-based Relational Division SSP version

```
Input: Dividend from server R and Divisor from server S
Output: Quotient at the mobile device

// Phase 1: Count Request
1. Send a count request to Server S (Divisor)
   Select Count(*)
   From Table_S;
2. Receive the divisor count and store it to CountS
3. Send the divisor count to Server Dividend to process

// Phase 2: Count comparison in the Server Dividend
4. The Server Dividend groups the quotient attribute
5. Count each group
6. Compare each group count with CountS received from mobile

// Phase 3: Qualifying group download from Server Dividend
7. Qualified dividend records are downloaded to mobile device

// Phase 4: Record comparison in Server Divisor
8. Qualified divident records are sent to Server Divisor for
   comparison of mathces.
9. Quotient tables as the result of the comparison are sent
   to mobile
```

Figure 6.9: Count-based Relational Division based on SSP Algorithm

In summary, the Relational Division uses the following steps:

Step 1: Obtain the count from the divisor server.

Step 2: Send the count of the divisor to the dividend in the server to compare the count of the dividend server and discard those that contain a count less than the count of the divisor. The server will count for each of the groups and compare the count received from the mobile device. It will process only those counts that are the same or greater and then it will send the result back to the mobile device.

Step 3a: Option 1: Using MDSP: Upon obtaining the counts that are either the same or greater between the divisor and dividend and sending them to the mobile device, a query will then be sent to the divisor server to obtain a full list of details so that a local comparison matching can be done to see whether or not they qualify. This is done in blocks within the dividend.

Step 3b: Option 2: Using SSP: Obtain the qualify match of the result in Step 2 together with its full details and send it to the divisor server for final comparison of the qualify match. This means that the results of the count that are the same or greater obtained from Step 2 will be downloaded to the mobile device and included with the other attributes records. This is done in blocks within the dividend.

Step 4: Repeat the same process until all blocks that qualify based on the count within the dividend have been compared and processed.

As shown above, count-based relational division can be either integrated with the mobile device side processing (MDSP) or server side processing (SSP). There is only a slight difference between using the MDSP or SSP as outlined in Steps 3a and 3b, where MDSP does the main processing in the mobile, whereas SSP does it in the divisor server.

6.4 Multiple Group Division Techniques

We will now consider the next proposed technique which tackles a major problem of existing techniques as highlighted previously in Chapter 2. Our investigation of related work revealed that previous researchers have not considered group division in the mobile environment. In this section, we propose two different methods for tackling group division.

The proposed techniques are called:

- Sort Merge Multiple Group Division
- Aggregate Multiple Group Division

Each of the above techniques aims to answer the query of "Listing the cinemas that show ALL movies in at least one category". This query will use the following tables as in Figure 6.10 to illustrate the process of obtaining the results.

Cinema_ID	Movie_ID
A	1
A	3
A	4
A	5
B	1
B	2
B	6
C	2
C	3
C	6

Movie_ID	Category
1	Drama
2	Drama
2	Action
3	Action
6	Action
3	Horror
4	Horror

Figure 6.10: Sample data for Multiple Group Division

Note the data for the multiple group division query has an additional group attribute in the Divisor. In the above example, the additional attribute is the Category attribute which acts as a group attribute for the multiple division. This attribute will also appear in the Quotient table. Therefore, the Quotient table will have two

attributes: the original quotient attribute from the dividend table and this new group attribute from the divisor table. The former is now called the first quotient attribute, whereas the latter is now called the second quotient attribute, producing a Quotient table with two attributes.

6.4.1 Sort-Merge Multiple Group Division

As in the example shown in Figure 6.11, using the Sort-Merge Multiple Group Division, we would first request to download all movies from all categories in the divisor which contain the Movie_ID and Category records. When issuing the query to download the records, ensure that movies from each category are sorted in ascending order.

Movie_ID	Category
1	Drama
2	Drama
2	Action
3	Action
6	Action
3	Horror
4	Horror

Figure 6.11: Download of the Divisor Table
(divisor attribute values within each quotient attribute are sorted)

Cinema_ID	Movie_ID
A	1
A	3
A	4
A	5
B	1
B	2
B	6
C	2
C	3
C	6

Block 1
Block 2
Block 3

Figure 6.12: Download of the Dividend Table
(Dividend attribue values are sorted for each quotient attribute value)

The next step is to download the dividend in a block manner. The block is defined for each cinema (using the above example). In other words, each separate cinema, regardless of how many movies it holds, is considered as a block. The same as for Step 1, ensure that the Movie_ID for each cinema is sorted as in Figure 6.12.

Step 3 is the local processing itself. With two lists of records which are the divisor and a block of the dividend that are downloaded one at a time in the mobile device, the merging process will be carried out. The process will merge one block of the dividend with all blocks of the divisor.

In the process of merging the first block of the dividend with the whole divisor, we will first compare the Cinema '*A*' with Category '*Drama*' to check each element. This means to merge *A*(1,3,4,5) and *Drama*(1,2). After merging in order to ascertain whether or not they match, move on to the next group of the divisor which is comparing Cinema '*A*' with Category '*Action*'. This process is repeated until all categories in the divisor have been matched against the block of the dividend that has been downloaded. This can be seen in Figure 6.13.

After a comparison of the one block of the dividend with all divisor groups has been completed, Step 4 is to download the subsequent block of the dividend which in this case is Cinema '*B*' and repeat the same process of comparing them with the whole divisor.

This process will finish once all blocks from the dividend have been downloaded and compared with the divisor.

Cinema_ID	Movie_ID
A	1
A	3
A	4
A	5

Movie_ID	Category
1	Drama
2	Drama
2	Action
3	Action
6	Action
3	Horror
4	Horror

Figure 6.13: Merging first block of Dividend with the Divisor

Algorithm 6.3: Sort-Merge Multiple Group Division

```
Input: Dividend from server R and Divisor from server S
Output: Quotient at the mobile device

// Phase 1: Download Divisor
1. Send a request to Server S (Divisor) to download divisor
   Select *
   From Table_S
   Group By Table_S.quotient
   Order By Table_S.divisor;

// Phase 2: Download one dividend block at a time and process
2. For each block in Dividend
3.   Send a request to Server R (Divident) to download one
     divident block, sorted by divisor attribute
4.   Compare with the divisor
5.   If matched Then
6.     Store the match in the Quotient
7.   End If
8. End For
```

Figure 6.14: Sort-Merge Multiple group Division Algorithm

The following is the summary of the steps for Sort Merge Multiple Group Division:

Step 1: Download the divisor that is large enough to fit into the main memory of the mobile device. If the main memory fits the whole divisor, then download the whole thing. Otherwise, if the main memory fits only 2 blocks of divisor (for example, the 'Drama' and 'Action' categories) then download only these 2 blocks.

Step 2: Download a block of the dividend (e.g. Cinema '*A*').

Step 3: Process the comparison of the single block of the dividend with whatever divisor is in the main memory.

Now we address the issue of limited memory in the mobile device. Often it may not have enough memory space to hold the whole block of the divisor that has been downloaded from the server to the local memory. Therefore, to solve this problem, there are two options in Step 4 from which to select in order to save memory.

Step 4a: Download another block of the divisor if in Step 1 it has only partly been downloaded (For example, if not all categories from the divisor can be fitted into main memory).

Step 4b: The other option is to process by downloading all dividend blocks first instead of downloading al the divisor first.

As a summary, the two options of processing the multiple group division query using the proposed sort merge have a trade-off between them. In Option 1, we download all divisors for each block of dividend; whereas in Option 2, we download all dividend blocks for each block in divisor. Option 1 would appear to be more efficient if the divisor is smaller than the dividend; whereas Option 2 would be more desirable if the dividend is smaller than than divisor. Thus, when selecting which option to use for processing the group division queries, ascertain whether the divisor or the dividend is smaller and then select the right option accordingly for the processing.

6.4.2 Aggregate Multiple Group Division Technique

In this proposed technique, we make use of the count for each group using the aggregate count function. Therefore, first we obtain the count for each block in the dividend and the divisor. This is done by sending a query to the respective servers to obtain the two lists of data in sorted order based on the value of the count.

For instance, Figure 6.15 illustrates the list of data from the dividend and divisor that are to be downloaded to the mobile device.

Once the mobile device has the information of the count, the mobile can then execute the pruning phase. Based on the count information, we can actually prune the comparison. In this pruning phase, the comparison between the count of the dividend and divisor is performed. The condition is that the count of the group in the dividend has to be at least the same as the count of the group in the divisor. Thus, only the count of the dividend that is same or greater than the divisor qualifies. Figure 6.16 indicates how the comparison is being done.

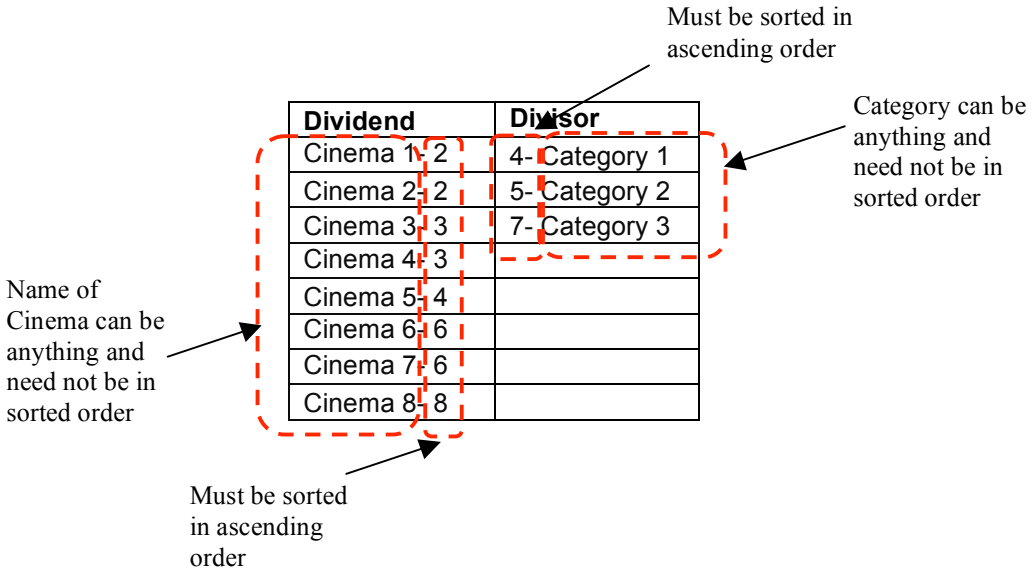


Figure 6.15: Complete the dividend for each divisor block

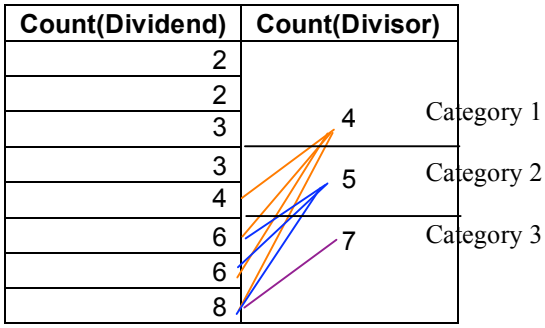


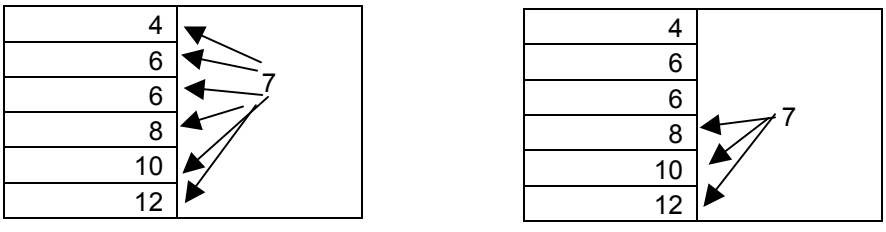
Figure 6.16: Pruning Phase

In the pruning phase as illustrated in Figure 6.16, category 1 in the divisor shows a count of 4 that needs to be compared with the count of the cinema which is the dividend starting with count 4 moving downwards (e.g. 4, 6, 6, and 8) as indicated by the arrow, whereas category 2 with count of 5 needs to be compared only with cinemas with counts 6,6 and 8. As for Category 3 with count of 7, this needs to be compared only with cinemas with counts of 8. Therefore with this scheme, cinemas with counts 2, 2, 3, 3 are eliminated right away.

Once the pairs have been formed, the mobile device can request the details from the dividend and divisor to download the details in sorted form to be merged. This process is then similar to what we did in sort-merge multiple group division technique. Therefore, as can be seen, this technique is an enhanced version of the previous technique. It appears to be more efficient since, even though we need to get all divisors downloaded, not all the dividends will be needed. Furthermore, the divisor with a large count needs to be compared only with the dividend with a large count. This is not a pure nested loop approach but rather, a pruning approach. This is because a high divisor count such as '7' will be compared only with the count of a dividend that is the same or higher which is '8' in the example given above.

To differentiate the pruning approach and the pure nested loop approach, Figure 6.17(a) and (b) show the difference between the pure nested loop approach and pruning approach whereby in a pure nested loop, the divisor '7' has to be compared with every single record in the dividend which is 4, 6, 6, 8, 10 and 12; whereas in the pruning, divisor '7' is compared only with counts of the dividend of 8, 10 and 12. The others such as 4, 6, 6 are automatically eliminated.

Figure 6.18 shows the algorithm for the aggregate-based multiple group division. In the first phase, the counts from both dividend and divisors are obtained. In the second phase, the pruning is in action. The result of this phase is the candidate pairs. In the third phase, each of the candidate pairs is sent to the respective servers to obtain the full data. The comparison will then be carried out in the mobile device.



(a) Pure nested-loop **(b) Pruning phase**

Figure 6.17: Pure Nested Loop vs. Pruning Phase

Algorithm 6.4: Aggregate-based Multiple Group Division

```
Input: Dividend from server R and Divisor from server S
Output: Quotient at the mobile device
// Phase 1: Count Request:
1. Send a count request to Server S (Divisor)
   Select QuotientAttr2, Count(*)
   From Table_S
   Group By QuotientAttr2
   Order By Count(*);
2. Receive the divisor count and store it to CountS
3. Send a group count request to Server R (Dividend)
   Select QuotientAttr1, Count(*)
   From Table_R
   Group By QuotientAttr1
   Order By Count(*);
4. Receive the group count and store it to CountR temp table
// Phase 2: Pruning phase
5. For each record in CountS
6.    For each record in CountR
7.       If CountR.count >= CountS.count Then
8.          Put both QuotientAttr from CountR and CountS into
            Candidate Pairs
9.       End If
10.   End For
11.End For
//  Phase 3: Comparison
12.For each pair in Candidate Pairs
13.  Send QuotientAttr1 to Server R (Dividend) and
     Download the records
14.  Send QuotientAttr2 to Server S (Divisor) and
15.  Download the records
16.  Compare the two quotients
17.  If matched then
18.    Store both quotients into the Quotient Table.
19.  End If
20.End For
```

Figure 6.18: Aggregate-based Multiple Group Division Algorithm

6.5 Conclusions

The techniques proposed in this chapter are divided into two different types where the first type is an improvement of the previous relational division technique adapted into the mobile environment using the MDSP and SSP. The other is an extension method whereby we introduced the multiple grouping division techniques. In the multiple

group division, we make use of the sort-merge and counting techniques. This multiple group division is the main contribution in this chapter, since none of the existing works focus on multiple group division. The contribution is further enhanced by its relevance to a mobile environment.

The performance of the multiple group division is further enhanced by having a pruning phase, where the counts of dividend groups that are at least the same or higher than the count of the divisor groups will be processed. With this pruning, we can prune away many records in the dividend table which do not have the potential to be included in the query results.

Chapter 7

Analytical Model and Experimentation

7.1 Introduction

In order to measure the effectiveness of the proposed algorithms, it is necessary to provide cost models that can describe the behaviour of each model. Although the cost models may be used to estimate the performance of a query, the primary intention is to use them for comparison purposes. The cost models also serve as tools to examine every cost factor in more detail, so that right decisions can be made when adjusting the entire cost components to increase overall performance. It is the aim of this chapter to analyse each algorithm through performance evaluation.

This chapter is organized as follows. Section 7.2 briefly introduces the basic cost components and cost notations. These are basically the variables used in the cost equations. In Section 7.3, basic cost calculation is explained, followed by basic costs for mobile query operations in Section 7.4. Section 7.5 drills down the analytical models for each of the proposed algorithms presented in this thesis. Section 7.6 presents experimental results, and the summary is presented in Section 7.7. Finally, Section 7.8 concludes the chapter.

7.2 Basic Cost Components

Analytical models are cost equations/formulas that are used to calculate the elapsed time of a query. A cost equation is composed of variables which are substituted with specific values at run-time of the query. These variables denote the cost components of query processing. The processing paradigm is a mobile device wirelessly connected to a couple of independent servers, which are non-collaborative, and each of the servers may receive/process/answer queries from the mobile client. Cost equations are composed of a number of components, particularly:

- Data parameters
- Systems parameters
- Query parameters
- Time unit costs
- Communication costs

Each of these components is represented by a variable, to which a value is assigned at run-time. The notations used are shown in Table 7.1 (Taniar et al., 2008). The original cost notations in Taniar et al. (2008) are for parallel database processing. In this thesis, these are adapted for mobile query processing.

Symbol	Description
Data Parameters	
R	Size of table
$\|R\|$	Number of records in table R
Systems Parameters	
P	Page size
H	Hash table size
Query Parameters	
π	Projectivity ratio
σ	Selectivity ratio
Time Unit Cost	
IO	Effective time to read a page from disk
t_r	Time to read a record in the main memory
t_w	Time to write a record to the main memory
Communication Cost	
T_{up}	Time to upload data from a mobile device to a server
T_{down}	Time to download data from a server to a mobile device

Table 7.1: Cost Notations

7.2.1 Data Parameters

There are two important data parameters (Taniar et al., 2008; Taniar, Rahayu and Ekonomosa, 2001):

- Actual size (in bytes) of the table (R).
- Number of records in a table ($|R|$), and

Data processing is based on the number of records. For example, the evaluation of an attribute is performed at a *record level*. On the other hand, systems processing, such as I/O (read/write data from/to disk) is done at a *page level*, where a page normally consists of multiple records. In terms of their notations, for the actual size of a table, a capital letter, such as R, is used. If two tables are involved in a query, then letters R and S are used to indicate tables 1 and 2, respectively. Table size is measured in bytes. For the number of records, an absolute notation is used. For example, the number of records in table R is indicated by $|R|$. Again, if table S is used in the query, $|S|$ denotes number of records in this table.

In data communications, the size is measured in bytes. If the message transferred is a query, the size of the query in bytes becomes the parameter. If the message transferred (either upload or download) is a number of records, the total size of records in bytes becomes the parameter.

7.2.2 Systems Parameters

There are two important systems parameters (Taniar et al., 2008; Taniar and Rahayu, 2006), namely:

- Page size (P) and
- Hash table size (H).

Page size, indicated by P, is the size of one data page in bytes, which contains a batch of records. When records are loaded from disk to main memory, they are not loaded record by record, but page by page. To calculate the number of pages of a given table, divide the table size by the page size. For examples, R = 4 Gigabytes (=

4×1024^3 bytes) and P = 4 Kilobytes (= 4× 1024 bytes), $R/P = 1024^2$ number of pages. Since the last page may not be a full page, the division result must normally be rounded up.

Hash table size, indicated by H, is the maximum size of the hash table that can fit into the main memory. This is normally measured by the maximum number of records. For example, H = 10,000 records. Hash table size is an important parameter in query processing. In a mobile environment, where the size of main memory is very limited, it is likely that the data cannot fit into the main memory all at once, because normally the size of the main memory is much smaller than the size of a database. Therefore, in the cost model, it is important to know the maximum capacity of the main memory, so that the number of times that a batch of records needs to be swapped in and out from the main memory to disk can be precisely calculated. The larger the hash table, the less likely that record swapping will be needed, thereby improving overall performance.

In server processing, although the size of the main memory of a server is relatively much larger that that of a mobile device, hash table size is also an important parameter, especially when the database size is larger than the available main memory.

7.2.3 Query Parameters

There are two important query parameters (Taniar et al, 2008; 2004), namely:

- Projectivity ratio (π), and
- Selectivity ratio (σ).

Projectivity ratio π is the ratio between the projected attribute size and the original record length. The value of π ranges from 0 to 1. For example, assume that the record size of table R is 100 bytes and the output record size is 45 bytes. In this case, the projectivity ratio π is 0.45 or 45%. In the MDSP and SSP techniques, where in some cases, only PK is needed, the proportion of the PK size and the record size is then reflected by the projectivity ratio.

Selectivity ratio σ is a ratio between the total output records, which is determined by the number of records in the query result, and the original total number of records. Similar to π, selectivity ratio σ also ranges from 0 to 1. For example, suppose initially there are 1,000 records ($|R|$ = 1,000 records), and the query produces 4 records. The selectivity ratio σ is then 4/1,000=1/250=0.004.

Selectivity ratio σ is used in many different query operations. To distinguish one selectivity ratio from the others, a subscript can be used. For example, σ_g indicates the number of groups produced by a groupby operation. Using the above example, the selectivity ratio σ of 1/250 (σ = 0.004) means that each group collects an average of 250 original records.

If the query operation involves two tables (for example, in a join operation), a selectivity ratio can be written as σ_j, for example. The value of σ_j indicates the ratio between the number of records produced by a join operation and the number of records of the Cartesian product of the two tables to be joined. For example, $|R|$ = 1,000 records and $|S|$ = 500 records, if the join produces 5 records only, then the join selectivity ratio σ_j is 5 / (1,000 × 500) = 0.00001.

Projectivity and selectivity ratios are important parameters in query processing, as they are associated with the number of records before and after processing; additionally, the number of records is an important cost parameter, which determines the processing time in the main memory.

7.2.4 Time Unit Costs

Time unit costs relate to the time taken to process one unit of data (Taniar et al., 2008). They are:

- time to read or to write a page from or to disk (*IO*);
- time to read a record in the main memory (t_r);
- time to write a record to the main memory (t_w); and
- time to perform a computation in the main memory.

Time taken to read/write a page from/to disk is basically the time associated with an input/output process. The variable used in the cost equation is denoted by *IO*. Notice that *IO* works at a page level. For example, to read a whole table from disk to main memory, divide table size and page size, and then multiply by the *IO* unit cost ($R/P \times IO$).

The time taken to write the query results into a disk is very much reduced as only a small subset of *R* is selected. Therefore, in the cost equation, in order to reduce the number of records as indicated by the query results, *R* is normally multiplied by other query parameters, such as π and σ.

Times taken to read and write a record in/to main memory are indicated by t_r and t_w, respectively. These two unit costs are associated with reading records, which are already in the main memory. These two unit costs are also used when obtaining records from the data page. Notice now that these two unit costs work at a record level, not at a page level.

Finally, the time taken to perform a computation in the main memory varies from one computation type to another. But basically, the notation is *t* followed by a subscript, which denotes the type of computation. Computation time in this case is the time taken to compute a single process in the CPU. For example, the time taken to hash a record to a hash table is shown as t_h, and the time taken to add a record to a current aggregate value in a group by operation is denoted as t_a.

7.2.5 Communication Costs

Communication costs can generally be categorized into elements:

- Upload cost per byte (T_{up}), and
- Download cost per byte (T_{down}).

Communication or data transfer costs are denoted by T_{up} and T_{down}, respectively. Both work at a byte level, as with the disk. It is well accepted that the download speed, indicated by the download cost, is faster than the upload speed. The actual data transfer cost will be determined by the amount of data that is transferred along the wireless medium.

7.3 Basic Cost Calculations

The previous section explains the cost components involved in mobile query processing. Each mobile query can be decomposed into a number of operations. In this section, basic operations and their cost calculations are given. Query operation normally follows these steps (Taniar et al., 2008):

- data loading (scanning) from disk;
- transferring records from data page to main memory;
- data computation;
- writing records (query results) from main memory to data page; and
- data writing to disk (in some cases).

7.3.1 Disk Operations

The first step corresponds to the last step, where data is read and written from and to the disk. As mentioned earlier in this chapter, disk reading and writing is based on page (i.e. I/O page). Several records on the same page are read/written as a whole.

The cost components for disk operations are the size of table (R or a reduced version of R), page size (P), and the I/O unit cost (IO). R and P are needed to calculate the number of pages to be read/written, whereas IO is the actual unit cost.

If all records are being loaded from a server disk, then we use R to indicate the size of the table read. Once the size of table is known, we can calculate the total cost of reading the data page from the disk as follows:

$$Scanning\ Cost = R / P \times IO \tag{7.1}$$

In some cases, the query results may need to be stored on disk. This is particularly relevant when the size of the main memory is insufficient to store

temporary query results. The writing disk cost is similar to the scanning cost. The main difference is that we need to determine the number of pages to be written, and this can be far below R, as some or many data have been eliminated or summarized by the data computation process.

Adjusting equation (7.1) to be adopted for the writing cost, we need to introduce cost variables that imitate the data computation process in order to determine the number of records in the query results. In this case, we normally use the selectivity ratio σ, and the projectivity ratio π. The use of these variables in the disk writing cost depends on the algorithms, but normally the writing cost is as follows:

$$Writing\ Cost = (data\ computation\ variables \times R) / P \times IO \tag{7.2}$$

where the value of the data computation variables is between 0.0 and 1.0. The value of 0.0 indicates that no records exist in the query results, whereas 1.0 indicates that all records are written back.

These two equations are general and basic cost models for disk operations. The actual disk costs depend on each query operation.

7.3.2 Main Memory Operations

Once the data has been loaded from the server disk, the record has to be removed from the data page and placed in main memory (the cost associated with this activity is called a *select cost*). This step also corresponds to the second last step – that is, before the data is written back to the disk, the data has to be transferred from the main memory to the data page, so that it will be ready for writing to the disk (this is called a *query results generation cost*).

Unlike disk operations, main memory operations are based on records, not on pages. In other words, $|R|$ is used instead of R.

The select cost is calculated as the number of records loaded from the disk times reading and writing unit costs to the main memory (t_r and t_w). The reading unit

cost is used to model the reading operation of records from the data page, whereas the writing unit cost is to actually write the record, which has been read from the data page, to main memory. Therefore, a select cost is calculated as follows:

$$Select\ Cost = |R| \times (t_r + t_w) \tag{7.3}$$

The query results generation cost is similar to the select cost, like the disk writing cost is to the disk reading cost. In the query results generation cost, there are two main important differences in particular. One is that the unit time cost is the writing cost (t_w) only, and no reading cost (t_r) is involved. The main reason is that the reading time for the record is already part of the computation, and only the writing to the data page is modelled. The other important element, the same as for the disk writing cost, is that the number of records in the query results must be modelled correctly, and additional variables must be included. A general query results generation cost is as follows:

$$Query\ Results\ Generation\ Cost = (data\ computation\ variables \times |R|) \times t_w \tag{7.4}$$

The query results generation operation may occur many times depending on the algorithm. The intermediate query results generation cost in this case is the cost associated with the temporary query results at the end of each step of data computation operations. The cost of generating the final query results is the cost associated with the final query results.

7.3.3 Data Computation

The main process in any database processing is data computation. What we mean by data computation is the performance of some basic database operations, such as searching, sorting, grouping, filtering of data. Here, the term computation is used in the context of database operation.

As data computation works in main memory, the cost is based on the number of records involved in the computation and the unit computation time itself. Each data computation operation may involve several basic costs, such as unit costs for hashing, for adding the current record to the aggregate value, and so on. However, generally, the data computation cost is a product of the number of records involved in the computation ($|R|$) and the data computation unit costs (t_x, where x indicates the total costs for all operations involved). Hence, a general data computation cost looks like this:

$$Data\ Computation\ Cost = |R| \times (t_x) \tag{7.5}$$

The above equation assumes that the number of records involved in the data computation is $|R|$. If the number of records has been reduced due to previous data computation, then we must insert additional variables to reduce $|R|$. Also, the data computation unit cost t_x must be spelt out in the equation, which may be a sum of several unit costs.

7.4 Mobile Query Processing - Main Cost Components

The previous two sections discuss in detail the basic cost components needed to perform any analytical evaluation of a proposed algorithm. These cost components are the basic building block and the foundation of our mobile query processing

performance evaluation. This section focuses on mobile query processing. As shown throughout the thesis, mobile query processing incorporates multiple independent servers which are non-collaborative, and the processing can be divided into three main elements:

- transfer costs,
- server costs, and
- mobile device costs.

Transfer costs, as the name states, deal with the overall cost of the communication between the mobile device and the servers. The server costs are the cost incurred by the server when carrying out some operations. The mobile device costs are somewhat similar to the server costs, with the main difference being that the mobile device costs are very much influenced by the limited capacity of the mobile device itself making the cost must higher in comparison with the server cost, especially when performing the same kind of operation.

The details of these three costs are explained in the following sections. These costs will be explained in the context of mobile query processing involving independent non-collaborative servers.

7.4.1 Transfer Costs

Transfer costs are associated with data or query transfer through a wireless medium. There are mainly three types of transfer costs:

- static query sending costs,
- data parameter query sending costs, and
- query results receiving costs.

The first two costs are upload costs which are the costs associated with uploading something by the mobile device to a server. Conversely, the last cost is a cost associated with downloading data from a server. The details of these three transfer costs are explained separately as follows:

(*a*) *Static Query Sending Costs*

A query is sent by a mobile device to a server in order to obtain the requested data. For example, a mobile user wants to obtain a list of movies from a server. This type of query is called a static query, as it can be expressed in an SQL or any query language or facilities provided by the server. The transfer cost associated with this is basically the cost of sending the request itself.

The transfer unit cost is T_{up} since we are uploading a request. The actual transfer cost is a product between the transfer unit cost T_{up} and the length of the query itself. If the length of the query is expressed as Q, then the static query sending cost is:

$$Q \times T_{up} \tag{7.6}$$

The query Q is measured in bytes. Since Q is normally quite small, it is often negligible. Even if a block-based approach is used in the algorithm whereby a query is sent every time a mobile device requesting a block of data from the server, the repeated transfer cost is relatively very small in comparison with actual record transfer. However, for the sake of completeness, if a query is sent to download one data block each, the static query transfer cost is expressed as follows:

$$\sum_{i=1}^{n} Q_i \times T_{up} \tag{7.7}$$

where n is the number of queries sent by the mobile device to the server.

(*b*) *Data Parameter Query Sending Costs*

In many algorithms, data obtained from one server needs to be sent across to the other server. This type of request is slightly different from the static query. In static query, the request is general and the query is rather short, whereas in the data parameter

query, the query has a long list of requests which is basically the data downloaded from the other server. For example, a list of movies has been obtained from one server, and this list of movies, which can be very long, is about to be sent to another server to be matched with movies in the other server. In other words, the initial list of movies obtained from the first server forms a data parameter to the query. Therefore, the length of such query depends on the length of its parameter, which in this case is a list of movies.

There are two main components of data parameter query sending costs. One is a fixed cost associated with the query itself, and the second is a variable cost associated with the data parameter. However, in general, the sending query cost is still: $Q \times T_{up}$. However, Q is divided into Q_1 which is the length of the static part of the query, which is normally very small, and Q_2 which is the data parameter and can be very long depending on how much data we send from the mobile device to the server.

In general, the total cost for data parameter query sending is:

$$(Q_1 + Q_2) \times T_{up},$$

$$\text{where } Q_2 = (\textit{data computation variables} \times R) \tag{7.8}$$

For example, if a list of PK of all records from the first server is to be uploaded to the second server, the data computation variable is the size of PK (in comparison with the entire record size). In other words, the length of PK is represented by a projectivity ratio π. Hence, $Q_2 = (\pi \times R)$.

In terms of its cost contribution to the overall query elapsed time, data parameter query sending costs contribute much more than the negligible static query sending cost, due to the variable part of the data parameter query sending costs depending on the number of records being sent by the mobile device to the server.

(*c*) *Query Result Receiving Costs*

As stated earlier that this transfer cost is a cost for downloading data from server. Hence, the transfer unit cost T_{down} is used, instead of T_{up}. However, there is a degree of similarity between this receiving cost and the data parameter query sending cost; that is, both are dependent on how much data is being uploaded/downloaded.

In general, the total receiving cost is:

$$data\ size \times T_{down} \tag{7.9}$$

The *data size* is influenced by the output of a server. The details of server operation costs are explained in the following section. If the server is only retrieving data from its local disk without any filtering, then the table size R is the *data size* for the above equation. If there is some filtering, whether it be projection (in the case of only certain attributes being downloaded) or selection (in the case of only certain records being downloaded), data computation variables must be applied to reduce the original data size.

7.4.2 Server Costs

Server costs are associated not only with SSP algorithms where the main processing is done in the server. Even in any MDSP algorithms, the server plays a role, such as retrieving data from its local disk, or even sorting the requested data by a mobile device.

Server costs in mobile query processing exist in a number of forms, including:

- data retrieval costs,
- sorting costs,
- counting and grouping costs, and
- joining costs.

(*a*) *Data Retrieval Costs* ($T_{retrieve}$)

Data retrieval costs are the basic costs incurred by reading the data from the disk. To some degree, this has been explained in the previous section. Therefore, equation (7.1) for the scan cost can be used to calculate data retrieval costs. Once the data has been scanned from the disk, it needs to be placed in the main memory, and hence equation (7.3) for the select cost should be applied.

However, if the server receives some parameters whereby only those records satisfying these parameters need to be read, after equations (7.1) and (7.3) are applied whereby each of the records is retrieved from disk and placed in the main memory, further data computation in the server main memory needs to take place. In this case, equation (7.5) needs to be calculated in order to filter out undesired records or to choose the required records. Once the desired records have been selected, equation (7.4) for writing costs can be applied.

Assuming that linear search is used, meaning that each record is linearly scanned and evaluated whether it is the desired record, the complete cost notations are expressed as follows:

- *Scanning cost*: the cost of loading half of the records (on average) from disk to main memory: $R / P \times IO$.

- *Select cost*: the cost of obtaining a record from data page: $|R| \times (t_r + t_w)$.

- *Comparison cost*: the cost of comparing a record with the search predicate: $|R| \times t_c$, where t_c is a time unit cost for search predicate comparison.

- *Result generation cost*, the cost to write found records to the data page: $\sigma \times |R| \times t_w$, where σ is the search query selection ratio.

The above cost equations are the complete versions of the equations (7.1), (7.3), (7.5), and (7.4) previously described. The total cost is the sum of the costs of scanning, selecting the data page, making a comparison, and generating results. Once

the required data has been retrieved from the server local disk, they are ready to be transferred to the mobile device which requested the data.

(*b*) *Sorting Costs* (T_{sort})

In many cases, the data requested by a mobile device to the server must be sorted. Therefore, simple data retrieval as mentioned above is not enough. Once the data has been retrieved, it needs to be sorted before sending it to the mobile device. In the literature, there are two kinds of sorting: *internal sorting* and *external sorting* (Elmasri and Navathe, 2007; Taniar et al., 2008).

Internal sorting is where sorting takes place totally in the main memory. The data to be sorted is assumed to be small and fits the main memory. *External sorting,* on the other hand, is where the data to be sorted is large and resides in secondary memory. Thus, external sorting is also known as file sorting. In databases, since data is stored in tables (or files) and is normally very large, database sorting is therefore an external sorting. Therefore, we assume that when a mobile device sends a query to retrieve data and present it in a sorted order (Order By clause in the SQL), an external sorting must be done by the server.

There are two main cost components for external sort, namely, the costs relating to I/O and those relating to CPU processing. The I/O costs are the disk costs, which consist of load cost and save cost. These I/O costs are as follows.

- *Load cost* is the cost of loading data from disk to main memory. Data loading from disk is done by pages.

  $$Load\ Cost = Number\ of\ pages \times Number\ of\ passes \times Input\ Output\ unit\ cost$$

  where $Number\ of\ pages = (R/P)$, and

  $$Number\ of\ passes = (\lceil log_{B-1}\ (R/P/B) \rceil + 1)$$

  Hence, the above load cost becomes:

  $$(R/P) \times (\lceil log_{B-1}\ (R/P/B) \rceil + 1) \times IO$$

- *Save cost* is the cost of writing data from the main memory back to the disk. The save cost is actually identical to the load cost, since the number of pages loaded from the disk is the same as the number of pages written back to the disk. No filtering to the input file has been done during sorting.

The CPU cost components are determined by the costs involved in getting records out of the data page, sorting, merging, and generating results, which are as follows.

- *Select cost* is to obtain a record from the data page, which is calculated as the number of records loaded from the disk times reading and writing unit cost to the main-memory. The number of records loaded from the disk is influenced by the number of passes, and therefore equation (7.6) above is applied here to calculate the number of passes.

$$|R| \times Number\ of\ passes \times (t_r + t_w)$$

- *Sorting cost* is the internal sorting cost which has a $O(N \times \log_2 N)$ complexity. Using the cost notation, the $O(N \times \log_2 N)$ complexity has the following cost.

$$|R| \times \lceil log_2 (|R|) \rceil \times t_s$$

The sorting cost is the cost of processing a record in pass 0 only.

- *Merging cost* is applied to the first pass onwards. It is calculated based on the number of records being processed, which is also influenced by the number of passes in the algorithm, multiplied by the merging unit cost. The merging unit cost is assumed to involve a *k*-way merging where searching for the lowest value in the merging is incorporated in the merging unit cost. Also, bear in mind that 1 must be subtracted from the number of passes, as the first pass (i.e. pass 0) is used by sorting.

$$|R| \times (Number\ of\ passes - 1) \times t_m$$

- *Generating result cost* is the number of records being generated or produced in each pass before they are written to disk multiplied by the writing unit cost.

$$|R| \times Number\ of\ passes \times t_w$$

(*c*) *Counting and Grouping Costs* ($T_{groupcount}$)

Counting and grouping costs are associated with aggregate functions and groupby operations. If the query simply asks to perform an aggregate function without a groupby clause, we assume that one group is performed. If the query involves a groupby operation whereby a number of group is produced by the query and each group produces one aggregate value, then we assume *n* number of groups is produced.

It is well known that the hash-based aggregate technique is the most efficient technique for groupby and aggregate operations (Taniar et al, 2002). Therefore, in this case we assume that when the server is performing a groupby operation and an aggregate function, a hash-based technique is applied.

There are two additional time unit costs that need to be used:

- t_h, Time to compute hash value, and
- t_a, Time to add a record to current aggregate value

Basically, the costs consist of the following components: scan cost, select cost, aggregate cost, overflow cost, and generating result cost. The first two costs are well-known and have been described in the above section. The final cost for generating result cost has also been presented. Therefore, the main unique costs are the aggregate cost which is the center of the groupby and aggregate operation, and sometimes an overflow cost might also occur, especially when the main memory size is rather small.

The complete cost components are as follows.

- *Scan cost* is the cost for loading data from local disk in the server: $(R / P) \times IO$

- *Select cost* is to obtain records from the data page, which is calculated as the number of records loaded from the disk times the reading and writing unit cost to the main-memory: $|R| \times (t_r + t_w)$

- *Local aggregation* involves reading, hashing, and computing the cumulative value, which is given by the number of records in main-memory times the reading, hashing, and computation unit costs.

$$|R| \times (t_r + t_h + t_a)$$

The hashing process is very much determined by the size of the hash table that can fit into the main-memory. If the memory size is smaller than the hash table size, we normally partition the hash table into multiple buckets whereby each bucket can fit perfectly into main-memory. A hashing technique can be roughly explained as follows (Taniar et al., 2008).

a. The records are read and hashed into a hash table based on the *Group By* attribute. The first record hashing to a new value adds an entry to the hash table, and the subsequent matches update the cumulative result as appropriate.

b. If the entire hash table cannot be fitted into the allocated memory, the records are hash partitioned into multiple buckets, and all but the first bucket are spooled to the disk.

c. The overflow buckets are processed one by one as in step (a) above.

In this scenario, we must include the I/O cost for reading and writing overflow buckets, which is as follows.

- *Reading/Writing of overflow buckets* cost is the I/O costs associated with the limitation of main-memory to accommodate the entire hash table. This cost includes the costs of reading and writing records not processed in the first pass of hashing (Taniar et al., 2008).

$$\left(1-\min\left(\frac{H}{\sigma\times|R|},1\right)\right)\times\left(\pi\times\frac{R}{P}\times 2\times IO\right) \tag{7.10}$$

The first term of the above equation can be explained as follows. For example, if the maximum hash table size H is 10 records, selectivity ratio σ is ¼, and there are 200 records ($|R|$), the number of groups in the query result will be equal to 50 groups ($\sigma \times |R|$). Since only 10 groups can be processed at a time, we need to break the hash table into 5 buckets. All buckets but the first are spooled to disk. Hence, 80% of the groups (1 – (10/50)) is overflow. Should there be only fewer than, or equal to, 10 groups in the query result, the first term of the above equation would be equal to 0 (zero), and hence there would be no overhead.

The second term of the above equation is explained as follows. The constant 2 refers to two input/output accesses: one is for spooling of the overflow buckets to disk and two is for reading the overflow buckets from disk. Notice that the record size is reduced by the projectivity ratio π, because in the hash table only the projected attributes are kept, not the whole record.

- *Generating result records* cost is the number of selected records multiplied by the writing unit cost: $|R| \times \sigma \times t_w$

(*d*) *Joining Costs* ($T_{hashjoin}$)

For the join operation in the server, since a hash-based join is the most efficient join algorithm, it is assumed that a hash-based join is used in the server. The cost of the local join using a hash-based join comprises three main phases: data loading from each processor, the joining process (hashing and probing), and result storing in each processor.

The data loading consists of scan costs and select costs:

- *Scan cost* = $((R / P) + (S / P)) \times IO$

  Note that in the above equation, both tables *R* and *S* are assumed to reside in the local disk of the server. However, if only one table resides in the server, and the other table is actually sent by a mobile device (in SSP techniques), then the scan cost is applied to only one table, either *R* or *S*.

- *Select cost* = $(|R| + |S|) \times (t_r + t_w)$

  Again, assuming that one table is sent by a mobile device and the other table resides in the local disk of the server, only one table component either $|R|$ or $|S|$ is used in the above equation.

In short, the loading costs may be identical to the loading costs described previously. The main cost is actually in the join process itself. The join process itself is basically incurring hashing and probing costs, which are as follows:

- *Join* costs involve reading, hashing, and probing:

  $$(|R| \times (t_r + t_h) + (|S| \times (t_r + t_h + t_j))$$

The process is basically reading each record *R*, and hashing it to a hash table. After all records *R* have been processed, records *S* can be read, hashed, and probed. If they are matched, the matching records are written out to the query result.

The hashing process is very much determined by the size of the hash table that can fit into main memory. If the memory size is smaller than the hash table size, we

normally partition the hash table into multiple buckets whereby each bucket can perfectly fit into main memory. All but the first bucket are spooled to disk.

Based on this scenario, we must include the I/O cost for reading and writing overflow buckets, which is as follows.

- *Reading/Writing of overflow buckets* cost is the I/O cost associated with the limited ability of main memory to accommodate the entire hash table. This cost includes the costs for reading and writing records not processed in the first phase of hashing.

$$\left(1-\min\left(\frac{H}{|S|},1\right)\right)\times\left(\frac{S}{P}\times 2\times IO\right) \tag{7.11}$$

Although this looks similar to the previously mentioned overflow cost, there are two significant differences. One is that only *S* is included in the cost component, because only the table *S* is hashed; and, secondly, the projection and selection variables are not included, because all records *S* are hashed.

The final cost is the query results storing cost, consisting of generating result cost and disk cost.

- *Generating result records* cost is the number of selected records multiplied by the writing unit cost: $|R| \times \sigma_j \times |S| \times t_w$

Note that the cost is reduced by the join selectivity factor σ_j, where the smaller the selectivity factor, the lower the number of records produced by the join operation.

- *Disk cost* for storing the final result is the number of pages needed to store the final aggregate values times the disk unit cost, which is:

$$(\pi_R \times R \times \sigma_j \times \pi_S \times S / P) \times IO$$

As not all attributes from the two tables are included in the join query result, both table sizes are reduced by the projectivity ratios π_R and π_S. Also note that disk cost is optional as, in many algorithms, the join results are sent back to the mobile device and are not stored locally in the server's disk.

7.4.3 Mobile Device Costs

Mobile device costs are the costs associated with processing in the mobile device itself. Basically, the server costs explained in the previous section are applicable. However, the main differences are:

- *Higher computation unit cost*

  Since the processing speed of the mobile device is much slower than that of the server, it is likely that the actual processing speed will have a high value. The computation unit cost variable itself is the same; only the actual figure is different. For example, the t_x computation unit cost, such as t_r, t_w, t_h and t_a, in the mobile device has a much higher figure, compared with that of the server.

- *Smaller main memory size*

  The main memory size of a mobile device is not comparable with that of the server which has a much larger main memory size. Consequently, the hash size table H of the mobile device has a much smaller figure. As a result, for some operations, memory overflow is often encountered, and hence flushing to non-volatile memory is needed. It is therefore expected that memory overflow is much needed by a mobile device.

  Still in the area of memory size, if the table size that has been downloaded is much larger than the sum of any memory hierarchy (main memory and non-volatile memory) of a mobile device, the algorithm will

crash because of the inability of the mobile device to handle such as a large volume of data.

- *Relatively infrequent disk access*

  Since the data is located at the server, the mobile device needs to request only the data to be downloaded. Once the data has been received by the mobile device, the mobile device is ready to carry out some operation. The data that arrives at the mobile device will be kept in the mobile memory, and as a result, mobile disk access is often unnecessary. In other words, mobile device costs focus on mobile operations, one of which is merging operation.

  Since the results of a query operation are not written on the mobile non-volatile memory, disk writing cost is also not applicable. We assume that the query results are displayed on the mobile screen, and hence, only result cost is generated.

- *Inapplicable operations*

  We assume that the mobile device will not carry out any sorting operation. If the data needs to be sorted, it is assumed that the requested data comes in a sorted order. In other words, sorting will be done by the server.

In a mobile device, the main costs are then: joining, grouping, and merging costs. Each is explained as follows:

(*a*) *On-Mobile Joining and Grouping Costs* ($M_{hashjoin}$ and M_{group})

Both joining and grouping operations are still applicable to a mobile device. In terms of their costs, the cost equations of joining and grouping operations of the server are the same as those for the mobile device.

Without the disk access, the joining cost in the mobile device is: $(|R| \times (t_r + t_h) + (|S| \times (t_r + t_h + t_j))$, whereas the overflow cost remains the same. And the *Generating result records* cost is: $|R| \times \sigma_j \times |S| \times t_w$

The grouping cost for the mobile device is $|R| \times (t_r + t_h + t_a)$. The overflow cost also remains the same. The generating result cost is then $|R| \times \sigma \times t_w$.

Note that, in general, the same cost equations of the server are applied to the mobile, except that disk access costs are eliminated, unless overflow is encountered.

(*b*) *On-Mobile Merging Costs* ($M_{merging}$)

Merging operation is more common in the mobile device. One condition of merging is that the data needs to be in order. This can be achieved by requesting the server to sort the data. Once the data has arrived in a sorted order, the mobile device can carry out a merging process.

The complexity of merging is known to be linear, that is $O(N+M)$, assuming that N and M are the two data parameters of the merging process. Hence, the merging cost is:

$$(|R| + |S|) \times (t_r + t_c)$$

where t_c is the comparison cost. The generation result cost is

$$|R| \times \sigma_j \times |S| \times t_w$$

In the merging, there is no overflow cost. In summary, merging is quite efficient, provided that the heavy sorting operation is done previously by the server. In comparison, a hash-based join might need more memory in order to avoid overflow problem. However, with a sort-merge join, if the sort has already been outsourced to the server, the merging will be much more efficient, and hence the mobile device should opt for the merging option.

7.5 Analytical Models of the Proposed Algorithms

The previous sections have described the basic cost equations which form the foundation for the cost of each proposed algorithm in this thesis. In this section, each of the proposed algorithms will be dissected in order to study the behaviour of the algorithm.

The proposed algorithms in this thesis have been studied in detail in the previous chapters, covering (*i*) mobile join query processing, (*ii*) mobile Top-k join query processing, (*iii*) mobile groupby-join query processing, and (*iv*) mobile division query processing. In each category, a number of algorithms have been proposed. Therefore, in this section, we will address the cost models for each algorithm in the categories mentioned above.

7.5.1 Mobile Join Query Processing Cost Models

For mobile join query processing, in the earlier chapter we presented four major groups of algorithms, namely:

- MDSP (Mobile Device Side Processing), that comes in three flavours
- SSP (Server Side Processing), in two different formats
- BBP (Block-based Processing) for MDSP and SSP
- ABJP (Aggregate Based Block Join) for MDSP and SSP

(*a*) *MDSP Cost Models*

There are three MDPS versions, called MDSP1, MDSP2, MDSP3. In summary, for mobile side processing, MDSP1 obtains all the complete records from the two respective servers; MDSP2 receives a list of PK from one server and the complete records from the other server; and, MDSP3 receives a list of PK from the two servers. Once the data has been obtained, the processing at the mobile device side starts.

Further data requested depends on the type of processing and only for the matched records.

First, we would look at the various cost models that are divided into (*a*) *transfer time*, (*b*) *server time* and (*c*) *mobile time* for the different versions of mobile device side processing (MDSP).

Transfer Costs

As mentioned earlier, there are three different transfer or communications costs, namely static query sending cost, data parameter sending cost, and records receiving costs.

- *Static Query Sending Costs:*
  Using equation (7.6), the static query sending costs for *MDSP*1 are as follows:
  $Q_1 \times T_{up}$, for sending one query to the first server,
  $Q_2 \times T_{up}$, for sending another query to the second server.
  If the query is of the same length, whereby $Q_1 = Q_2$, then the static query sending cost for MDSP1 is: $(2 \times (Q \times T_{up}))$

  The static query sending costs for *MDSP*2 and *MDSP*3 are identical to those of the above: $(2 \times (Q \times T_{up}))$, although the actual queries for each MDSP are slightly different. However, in terms of the length of the query which needs to be sent to the respective server, it is to be approximately equal in length.

- *Data Parameter Query Sending Costs:*
  There is no data parameter query sending cost for *MDSP*1. Hence, the cost is nil.

  For *MDSP*2, note that one server only sends a list of PK for processing in the mobile device. After processing in the mobile device to form a list of matched records, if additional information from the server that initially sent the PK

only is required, then this incurs a data parameter query sending cost as in equation (8), which is as follows: $Q_3 \times T_{up}$

Q_3 is further divided into Q_{31} that represents fixed size query and Q_{32} represents values of variables that vary based on the number of qualified matches that are obtained. Q_{32} can be formulated as follows: $\pi \times \sigma \times R$, where π indicates the proportion of record R that needs to be downloaded, and σ indicates that only selected records are needed. These selected records exist in the matched query results processed in the mobile device.

For *MDSP*3, the data parameter query sending cost is double that of *MDSP*2, simply because in MDSP3, initially the mobile device only downloaded a list of PKs from both servers.

- *Record Receiving Costs:*

For *MDSP*1, the record receiving costs are the costs incurred because of downloading records from both servers. Assume the first server has table R and the second server has table S. Therefore, for *MDSP*1, the download cost for returning all results from the first server to the mobile device as in equation (7.9) is: $R \times T_{down}$, and the download cost for the second server is: $S \times T_{down}$

For *MDSP*2, the cost is very similar to that of MDSP1, except that the record receiving cost from the first server is only for the PKs. Hence, the costs are:
$(\pi \times R \times T_{down}) + (S \times T_{down})$
where $(\pi \times R)$ indicates the fraction of the record length is the PK itself.
If non-PK attributes need to be downloaded for the matched records, there will be an additional record receiving cost, which is:
$\sigma \times \pi \times R \times T_{down}$

The selectivity ratio s indicates that only the matched records are downloaded, whereas the projectivity ratio π indicates that only certain attributes (i.e. non-PK attributes) are downloaded.

For *MDSP*3, the record receiving cost is very similar to those of *MDSP*2, that is: $(\pi \times R \times T_{down}) + (\pi \times S \times T_{down})$
where the projectivity ratio π for both tables indicates the fraction of the table is the PK.

To download the details of the matched records, similar to MDSP2, when necessary the non-PK attributes are needed from both servers. Hence, the receiving records cost for MDSP3 is: $\sigma \times \pi \times (R + S) \times T_{down}$

The transfer costs (or communication costs) for the three MDSP techniques can be summarised as follows:

$$
\begin{aligned}
MDSP1 = \\
&(2 \times (Q \times T_{up})) + \\
&((R + S) \times T_{down})
\end{aligned}
\tag{7.12}
$$

$$
\begin{aligned}
MDSP2 = \\
&(2 \times (Q \times T_{up})) + \\
&((\pi \times R \times T_{down}) + (S \times T_{down})) + \\
&((Q_{31} \times T_{up}) + (\pi \times \sigma \times R \times T_{up})) + \\
&(\sigma \times \pi \times R \times T_{down})
\end{aligned}
\tag{7.13}
$$

$$
\begin{aligned}
MDSP3 = \\
&(2 \times (Q \times T_{up})) + \\
&((\pi \times R \times T_{down}) + (\pi \times S \times T_{down})) +
\end{aligned}
$$

$$((2 \times Q_{31} \times T_{up}) + (\pi \times R \times T_{down}) + (\pi \times S \times T_{down})) +$$
$$(\sigma \times \pi \times (R + S) \times T_{down})$$

(7.14)

Server Costs

For MDSP, the server costs basically involve retrieval costs and sorting costs only, since the main processing is carried out in the mobile device. When the mobile device sends its query requesting a list of data, there are two options: retrieve all data without sorting, or retrieve all data and sort them. If the requested data are not sorted, the mobile processing will adopt a hash-based join; whereas, if the requested data are sorted by the server, the mobile processing will perform a merging.

Apart from this difference, non-sorting data retrieval is definitely used in MDSP2 and MDSP3 when retrieving all non-PK attributes of the matched records. Both non-sorting retrieval and sorting retrieval costs are explained as follows:

- *Non-Sorting Retrieval Costs:*

  For *MDSP*1, the non-sorting retrieval cost is:

  $(R + S) \times T_{retrieve}$

  where $T_{retrieve}$ is the unit cost for disk retrieval, as described earlier in Section 7.4.2 (a).

  For *MDSP*2, the non-sorting retrieval cost is:

  $((\pi \times R) + S) \times T_{retrieve}$

  to retrieve the original records, and the following cost to retrieve non-PK attributes of the matched records:

  $(\pi \times \sigma \times R) \times T_{retrieve}$

  Note that the two projectivity ratios are not equal. The first projectivity ratio indicates the proportion of the size of PK to the overall record size, whereas the second one indicates exactly the opposite.

For *MDSP*3, the non-sorting retrieval cost is:

$((\pi \times R) + (\pi \times S)) \times T_{retrieve}$

For retrieving the non-PK attributes of the matched records it is:

$((\pi \times \sigma \times R) + (\pi \times \sigma \times S)) \times T_{retrieve}$

- *Sorting Retrieval Costs:*

  For *MDSP*1, the sorting retrieval cost is:

  $(R + S) \times T_{sort}$

  where T_{sort} is the unit cost for sorting, as described earlier in Section 7.4.2 (b).

  Note that for *MDSP*1, we need to choose only whether we adopt a sorting or non-sorting retrieval. In other words, there is only one retrieval cost to be applied.

  For *MDSP*2, the retrieval of the PK of table *R* follows the non-sorting retrieval, which is: $(\pi \times R) \times T_{retrieve}$

  Whereas the table *S* follows the sorting cost, which is:

  $(S) \times T_{sort}$

  For *MDSP*3, there is no sorting cost, because when retrieving the PK of both tables, we use the non-sorting retrieval cost.

The server costs for the three MDSP techniques can be summarised as follows:

$$MDSP1 = (R + S) \times T_{retrieve}$$

or

$$MDSP1 = (R + S) \times T_{sort} \tag{7.15}$$

$$MDSP2 = ((\pi \times R) + S) \times T_{retrieve} + (\pi \times \sigma \times R) \times T_{retrieve}$$

or

$$MDSP2 = ((\pi \times R) \times T_{retrieve}) + ((S) \times T_{sort}) + (\pi \times \sigma \times R) \times T_{retrieve} \tag{7.16}$$

$$MDSP3 = (((\pi \times R) + (\pi \times S)) \times T_{retrieve}) + (((\pi \times \sigma \times R) + (\pi \times \sigma \times S)) \times T_{retrieve}) \quad (7.17)$$

Mobile Costs

For MDSP, the mobile costs are heavy as they are the actual processing costs in the mobile device. Basically there are two types of MDSP processing, depending on whether the incoming data is sorted or not sorted. If the incoming data is sorted, then the mobile device performs a merging operation, otherwise a hash-based join operation.

- *Merging Costs:*
  For *MDSP*1, the merging cost is:
  $(|R| + |S|) \times M_{merge}$
  where M_{merge} is a unit cost of the merging operation on the mobile device which can be found in more detail in Section 7.4.3 (b). Note that an absolute value of *R* and *S* indicate number of records in the respective table.

  For *MDSP*2 and *MDSP*3, the merging cost is also the same as that of *MDSP*1. Although in MDSP2, table *R* is only the PK, in terms of its number of records, it does not matter whether the record is a full record or just an attribute of the record. The number of records is still the same. Hence, the merging costs for both MDSP2 and MDSP3 are identical to those of MDSP1.

- *Hash-based Join Costs:*
  For all MDSP techniques, the hash-based join cost is:
  $(|R| + |S|) \times M_{hashjoin}$

where $M_{hashjoin}$ is a unit cost for a hash-based join operation on the mobile device which can be found in more detail in Section 7.4.3 (a).

(*b*) *SSP Cost Models*

There are two SSP versions, called SSP1 and SSP2. For SSP, the mobile device obtains the data from one server and sends it to the other server for processing. The main difference between SSP1 and SSP2 is the data downloaded from one server is the entire record (SSP1) or only the PK (SSP2). If it is the latter, after processing, the non-PK details from server one must be downloaded later.

Like those of MDSP, we will look various cost models in terms of: (a) *transfer time*, (b) *server time* and (c) *mobile time* for the two versions of SSP. In SSP, it is obvious to expect that the server time would dominate the entire processing time. However, for completeness, we will look into the transfer and mobile times as well.

Transfer Costs

Again, we will consider the three transfer costs for SSP.

- *Static Query Sending Costs:*
  The static query sending costs for *SSP*1 and *SSP*2 are as follows:
  $Q_1 \times T_{up}$, for sending one query to the first server.
  This is a simple cost for sending a query to the first server.

- *Data Parameter Query Sending Costs:*
  The data obtained from server one needs to be sent to server two for processing, thereby incurring a data parameter query sending cost. For *SSP*1, the data parameter sending cost is dominated by the entire table *R*, whereas for *SSP*2, the sending cost is only for the PKs. Again, there are two parts to the data parameter query sending costs: fixed part which is the query itself, and

the variable part which contains the data. The entire data parameter sending cost is:

$Q_2 \times T_{up}$

Q_2 is divided into Q_{21} that represent fixed size query and Q_{22} represent variables values that vary based on the number of qualified matches that are obtained.

For *SSP*1, $Q_{22} = R$, and

For *SSP*2, $Q_{22} = \pi \times R$.

For *SSP*1, after the processing, matched records need to download non-PK attributes from server one. Hence:

$Q_4 \times T_{up}$

where $Q_{32} = \pi \times \sigma \times S$.

Note that ($\pi \times \sigma$) indicates the selectivity ratio of the matched records.

- *Record Receiving Costs:*

  The record receiving costs from server one for *SSP*1 and *SSP*2 are:

  SSP1 = $R \times T_{down}$

  SSP2 = $\pi \times R \times T_{down}$

  Once the processing has been completed in server two, the matched records are sent to the mobile device. The costs for both *SSP*1 and *SSP*2 are identical, which is:

  $\sigma \times S \times T_{down}$

  where $\sigma \times S$ indicates the selectivity ratio of table *S* from server two is selected.

  Additionally for *SSP*1, since we need to obtain non-PK attributes from server one, it incurs a records receiving cost, which is:

  $\sigma \times \pi \times R \times T_{down}$

As a summary, the transfer costs (or communication costs) for the SSP techniques are as follows:

$$
\begin{aligned}
SSP1 = \\
&(Q_1 \times T_{up}) + \\
&(R \times T_{down}) + \\
&((Q_{21} \times T_{up}) + (R \times T_{up})) + \\
&(\sigma \times S \times T_{down})
\end{aligned}
\tag{7.18}
$$

$$
\begin{aligned}
SSP2 = \\
&(Q_1 \times T_{up}) + \\
&(\pi \times R \times T_{down}) + \\
&((Q_{21} \times T_{up}) + (\pi \times R \times T_{up})) + \\
&(\sigma \times S \times T_{down}) + \\
&((Q_{21} \times T_{up}) + (\pi \times \sigma \times S \times T_{up})) + \\
&(\pi \times \sigma \times R \times T_{down})
\end{aligned}
\tag{7.19}
$$

Server Costs

For SSP, the server costs basically involve retrieval costs and join costs. The retrieval cost is imposed by server 1, whereas the join cost is imposed by server 2. The retrieval, either the PK or the entire record, is done through simple retrieval without sorting, whereas the join operation is done in a hash-based join manner.

- *Non-Sorting Retrieval Costs:*
  For *SSP*1, the non-sorting retrieval cost is:
  $(R) \times T_{retrieve}$

where $T_{retrieve}$ is the unit cost for disk retrieval, as described earlier as in Section 7.4.2(a).

For *SSP*2, the non-sorting retrieval cost is applied to the PK only, indicated by the projectivity ratio which is the proportion of the PK to the entire record.
$(\pi \times R) \times T_{retrieve}$

After the processing, if non-PK attribute values of the matched record from server 1 need to the retrieve, the retrieving cost for this is:
$(\pi \times \sigma \times R) \times T_{retrieve}$

- *Joining Costs:*

  For *SSP*1, the joining cost is:
  $(R + S) \times T_{hashjoin}$
  where $T_{hashjoin}$ is the unit cost for joining operation, as described earlier in Section 7.4.2 (d).

  For *SSP*2, the joining cost is:
  $((\sigma \times R) + S) \times T_{hashjoin}$

As a summary, the server costs for the two SSP techniques are as follows:

$$
\begin{aligned}
SSP1 = \\
& R \times T_{retrieve} + \\
& (R + S) \times T_{hashjoin}
\end{aligned}
\tag{7.20}
$$

$$
\begin{aligned}
SSP2 = \\
& (\pi \times R) \times T_{retrieve} + \\
& ((\sigma \times R) + S) \times T_{hashjoin} + \\
& (\pi \times \sigma \times R) \times T_{retrieve}
\end{aligned}
\tag{7.21}
$$

Mobile Costs

The mobile device basically sends a request to server 1. This cost is covered in the transfer cost. Once the mobile device has received the data from server 1, it sends the data to server 2 for processing, and this is covered by the data parameter query sending cost, as well as the server cost which is the actually processing itself by server 2. In other words, there is no processing done in the mobile device, and hence SSP does not incur any mobile costs.

(*c*) *BBP Cost Models*

There are two Block-Based Processing models: one is MDSP-based and the other is SSP-based. The processing of block-based and non-block-based is actually very similar.

In MDSP, for example, the records downloaded which must have been sorted by the server come in a number of blocks, one block at a time. There are two consequences. One is that the user is able to terminate the process at any point in time, without finishing the processing of all blocks. Hence, we save processing time. A second consequence is that if the user decides to process all blocks, then there will be no difference between block-based and non-block-based, since all data is processed anyway.

For SSP, it is the same thing. The records downloaded from server 1 come into several blocks and the block is sent and processed to server 2 as it comes. The same two consequences apply: that is, the user is able to terminate the process at any point of time without downloading all blocks from server 1; and secondly, if all blocks are ultimately processed, there will be no difference between the block-based and the non-block-based.

However, there is one major difference between the block-based and the non-block-based - that is, the number of queries sent by the mobile device to the server. For MDSP for example, one request sent by the mobile device is for downloading one

block. If there are *n* blocks to be downloaded, then there will be *n* requests or queries. The same applies to the SPP. The processing costs for the block-based MDSP and SSP are as follows. For simplicity, we assume that MDSP1 and SSP1-like are used in the block-based.

- *Block-based MDSP Costs:*

  The static query sending cost is the cost to send the query to download one block each from the first server. Hence it is:

  $Q_{R1} \times T_{up}$
  $Q_{R2} \times T_{up}$
  …
  $Q_{Rn} \times T_{up}$

  The total upload cost of sending the number of queries that are needed for downloading all blocks as in equation (7.7), can be summarized as follows:

  $$\sum_{i=1}^{n} Q_{Ri} \times T_{up}$$

  The same is applied to the second server, for which the static query sending cost is:

  $$\sum_{i=1}^{n} Q_{Si} \times T_{up}$$

  For the actual data, the cost sent by the server is:
  $R_1 \times T_{down}$
  $R_2 \times T_{down}$
  …
  $R_n \times T_{down}$
  which is equal to:

  $$\sum_{i=1}^{n} R_i \times T_{down}$$

where n is the number of blocks downloaded from server 1 (table R).

The same applies to the second server, which is:

$$\sum_{i=1}^{m} S_i \times T_{down}$$

where m is the number of blocks downloaded from server 2 (table S).

The server cost is basically retrieving (and sorting the data) from the server local disk, before it is being sent to the mobile device. The entire table is retrieved by the respective server and sorted. Once the data is retrieved and sorted, the data can then be transfer block-by-block to the mobile device. Hence, the non-sorting retrieval cost for the non-block MDSP1 can be used. The mobile cost is basically the merging cost in the mobile device. Again, the total cost of merging if all blocks are downloaded and processed, is the same as the merging cost of the non-block MDSP1 technique.

- *Block-based SSP Costs:*

  The static query sending cost for the SSP block-based is half of that of the MDSP block-based, since the query is only sent to server 1. Hence the cost is:

  $$\sum_{i=1}^{n} Q_{Ri} \times T_{up}$$

  After the query has been sent to the first server, the server has to respond with retrieving the requested data. If all blocks are requested, then the retrieving cost will be the same as that of the server retrieving cost of the non-block-based SSP.

  Once the data is retrieved from the local disk in server 1, the data transfer to the mobile device is done block-by-block. Again, if all blocks are being sent to the mobile device, the transfer cost for the SSP block-based is the same as that of the non-block-based.

  Once the data has been received by the mobile device, each block is sent to the second server for processing. In terms of the variable cost, which consists of

the records itself, the block-based cost would be the same as for the non-block-based. The only difference is the fixed part, which is multiplied by the number of queries. Although this is relatively very small, nevertheless, there is a slight difference.

Once the data has been received by the second server, it is then joined with the local data in the second server (table *S*). In server 2, the join will be a hash-join process. This can done as follows: Table *S* is hashed into a hash table. Once a block of table *R* called R_1 arrives, the records of R_1 are hashed and probed into the hash table *S*. This is repeated for all incoming blocks from table *R*. Based on this process, the hash-join cost will be the same as that of the non-block-based.

Once the join operation has been completed in server 2, the results are sent back to the mobile device. The query result sending cost will also be identical to the non-block-based.

(d) ***Aggregate Based Block Join Cost Models***

There are two version of the aggregate based block join: MDSP-based and SSP-based. The main difference in terms of its processing strategy between the non-aggregate based (block-based or non-block-based) and the aggregate based block join is that in aggregate based block join, there is a preprocess in the join operation; that is, to group a number of records based on a certain additional attribute and to count the number of records in the group. Based on the counts for each group, it can be determined whether the entire group should proceed with the join process or be eliminated entirely. This process is certainly attractive if the groups' composition between the two tables is rather different, so that the join operation can be reduced. For example, tables *R* and *S* have two groups or categories each, and one of the categories from both tables is entirely different. As a result, only one group from each table will be joined. Therefore, the efficiency of the aggregate based block join is very

much dependent on the group composition, in order to take advantage of the full benefit it offers.

Since the main difference lies in the grouping, the counting, and later the pruning operations, in studying the behaviour of these algorithms, in the cost models, we focus only on the grouping and counting processes. Once the grouping, counting, and pruning are complete, the actual join process will follow the block-based MDSP/SSP as described in the previous section.

MDSP Aggregate Based Block Join Costs

For the transfer costs, since the static query sending cost does not contribute too much to the overall processing cost, we focus only on the transfer costs that involve the sending and receiving of records. Hence, we call this "record transfer" costs.

- *Record Transfer Costs:*

  In MDSP aggregate based block join, each server sends the group counts to the mobile device. Using the parameter σ_g to indicate the selectivity ratio of the groups and the table size, the number of groups formed by each server contributes to the total record transfer cost (which is the groupby transfer cost) which is as follows:

  $(\sigma_{g1} \times \pi \times R \times T_{down}) + (\sigma_{g2} \times \pi \times R \times T_{down})$

  Since the number of groups produced by each server is likely to be different, we use two selectivity ratios called σ_{g1} and σ_{g2}, whereas the projectivity ratio π indicates that only a fraction of the record size is downloaded.

The server cost is basically incurred by the retrieval, the grouping, and the counting of each group. The group and group count will be presented in a sorted order based on the group. The algorithm that is used to perform the grouping and counting by the server is generally a hash-based aggregate operation.

- *Server Grouping/Counting Costs:*

  The details of the grouping/counting unit cost have been discussed earlier, involving hashing and aggregating. For simplicity, we call the unit cost $T_{groupcount}$.which can be found in more detail in Section 7.4.2(c). Hence, the server cost for grouping and counting is as follows:

  $(R + S) \times T_{groupcount}$

Once the groups have been downloaded from each respective server to the mobile device, the mobile device can then do a merging between the two lists of groups to eliminate groups that exist only on one list.

- *Mobile Merging Costs:*

  The mobile merging cost depends on the number of groups that are produced by each server, and this is reflected in the number of groups being downloaded (transferred) from the server to the mobile device, as indicated by the transfer cost.

  The merging operation was discussed earlier, and for simplicity, we introduce a merging unit cost, which is $M_{merging}$.which can be found in more detail in Section 7.4.3(b) The capital *M* indicates that this is the mobile cost.

  Since the merging operation is a linear operation, the mobile merging cost is very similar to the group transfer cost, with the difference of the merging unit cost being used, and not a transfer (or download) unit cost. Hence, the mobile merging cost is as follows:

  $(\sigma_{g1} \times \pi \times R \times M_{merging}) + (\sigma_{g2} \times \pi \times R \times M_{merging})$

Once this pre-processing operation has been completed, the next process is the same as that of the block-based MDSP. The only difference is that the number of records involved in the join, is reduced. We shall introduce another selectivity ratio to

reduce the original number of records in table *R* and table *S*. Apart from this, the costing equations will be the same as those for the block-based MDSP.

SSP Aggregate Based Block Join Costs

Similar to MDSP, we focus only on the grouping and counting processes in the aggregate block-based join. There are similarities and of course differences between the MDSP aggregate based join and the SSP aggregate based join.

The similarity is that both servers do the grouping and counting, and the groups and counts are sent to the mobile device.

The difference, as for any other SSP, is that the joining operation is carried out in one of the servers. The reason that both servers initially need to send the groups and the counts to the mobile device is that the mobile device decides for each group, from which group (from which server) the records should be downloaded. As explained in the algorithms, the count of each group will determine the server that needs to send the data to the mobile device. For example, for the first pair of groups, if the group from server 1 is smaller than the group from server 2, then the records of this group from server 1 will be downloaded and then transferred to server 2 for processing. The same process applies to each pair.

- *Record Transfer Costs, Server Grouping/Counting Costs, and Merging Costs:*
  From the costing view point, the record transfer cost for SSP will be identical to that of MDSP, which is: $(\sigma_{g1} \times \pi \times R \times T_{down}) + (\sigma_{g2} \times \pi \times R \times T_{down})$.
  As a result, the server initial grouping and counting cost will also be the same, that is: $(R + S) \times T_{groupcount}$.

  Once the mobile device receives the groups and counts, to identify which block to download from one server and to send to the other server, each pair of groups needs to be compared, and hence the merging cost should be used, which is: $(\sigma_{g1} \times \pi \times R \times M_{merging}) + (\sigma_{g2} \times \pi \times R \times M_{merging})$.

Therefore, the main difference between the MDSP and the SSP aggregate based block join lies in the process after the groups and counts have been received and processed by the mobile device.

In the MDSP version, the records of both groups are downloaded to be processed in the mobile; whereas in the SSP version, the records of one group are downloaded to be sent to the other server. Therefore, for the SSP version, the next process is similar to the block-based SSP version.

In summary, the block-based approaches give the mobile user the opportunity to terminate the process at any point in time, because the data is downloaded and processed block-by-block. However, if all blocks are processed, there will be only a slight difference in terms of the processing costs between the non-block and block-based. And in fact, the block-based will incur a slight overhead due to the repeated querying costs. In other words, the block-based approach will benefit the entire processing time only if not all blocks are processed.

7.5.2 Mobile Top-k Join Query Processing Cost Models

There are two main algorithms proposed for mobile Top-k join query processing: one is based on a nested-loop and the other is based on merge. Note that our main contribution to the Top-k join is that we have formulated that, although top-k results are obtained during the process, the process should not stop if there are any high ranked records (records that are ranked higher than the bottom top-k records) that have not had any matches. However, in this section, we will discuss the cost models of the proposed algorithms.

(*a*) *Top-k Nested-Loop Join Cost Models*

In our Top-k join processing, the server sends records to be processed on the mobile side one block at a time. The joining process, which follows a nested-loop approach is

done on the mobile side. The main difference between Top-k nested-loop and the general nested-loop join, is that in Top-k join, the process is done block-by-block, and if the final Top-k records are obtained, the entire joining process terminates. Therefore, there is a factor that the join process may not need the entire records from both servers. Note that in our algorithm, we assume that the mobile device which sends a request to the server to send a block of records, the data in the server itself is already ranked.

There are two main components of our Top-k nested-loop join costs: (*i*) server time which covers the retrieving of records by the server, and transfer time which covers the download (transfer) time of these records, and (*ii*) mobile time which covers the local join itself.

- *Server and Transfer Costs:*

  In the worst case, all records from both servers need to be retrieved and sent to the mobile device. Hence the server time is:

  $(R + S) \times T_{retrieve}$

  Following this, the transfer cost to send these records to the mobile device is:

  $(R + S) \times T_{down}$

  The above is true for all records that are retrieved and transferred, even though the records are retrieved and transferred one block at a time.

  The best case is where only one block of records is retrieved and transferred to the mobile device, whereby after processing one block, the desired Top-k join results are already obtained, and consequently the entire process terminates.

  In practice, however, it is in between. Sometimes it leans toward the best case, and at others it leans toward the worst case. Therefore, in this case we use a selectivity ratio to indicate number of records retrieved. Therefore, the server time and the transfer time are as follows:

  Server cost = $\sigma \times (R + S) \times T_{retrieve}$

  Transfer cost = $\sigma \times (R + S) \times T_{down}$

Say for example, that the selectivity ratio is 50% - that means that only half of the records are retrieved and downloaded, saving the other half.

- *Mobile Costs:*

  The mobile performs a normal nested-loop join based on the available block. If only a certain percentage of records are downloaded and processed, then the cost is based on this percentage of records, which is indicated by the selectivity ratio as mentioned above. Therefore, the parameters for the join operation are: $(\sigma \times R)$ and $(\sigma \times S)$.

  So far the joining operation used in the algorithm is either a hash-based or a sort-merge. A nested-loop join in known to have a square complexity, as stated in Chapter 2 (e.g. $O(N \times M)$). Therefore, the nested-loop join cost for the mobile is:

  $((\sigma \times R) \times (\sigma \times S)) \times M_{nestedloop}$

  The first two parameters indicate that it may be the case that only a fraction of both tables R and S are involved in the nested-loop join, whereas the multiplication indicates the square complexity of the nested-loop join algorithm as indicated by the complexity analysis in Chapter 2. The unit processing cost $M_{nestedloop}$ is the processing unit cost to process a nested-loop join.

(*b*) *Top-k Merge Join Cost Models*

Top-k merge join is also very similar to the nested-loop. It has two main components: the server/transfer time, and the mobile time.

- *Server and Transfer Costs:*

  The server and transfer costs are identical to those of the Top-k nested-loop, which are:

  Server cost = $\sigma \times (R + S) \times T_{retrieve}$

Transfer cost = $\sigma \times (R + S) \times T_{down}$

Regardless of whether it is a nested-loop or a merge join, the records need to be retrieved from the server's local disk and transferred to the mobile device.

- *Mobile Costs:*

  The main difference between the two methods lies in the mobile cost, whereby in Top-k merge join, a merging cost is applied, not a nested-loop. Note that the difference between a merging and a nested-loop is that the merging operation is a linear operation, whereas the nested-loop has a square complexity.

  Applying a merging cost to the Top-k merge join cost, the cost is:

  $((\sigma \times R) + (\sigma \times S)) \times M_{merge}$

  Note that the plus sign is used between variables R and S to indicate that it has a linear complexity, meaning that each record is accessed only once. The processing unit cost is also changed to M_{merge}.

7.5.3 Mobile GroupBy-Join Query Processing Cost Models

For mobile groupby-join query processing, there are two major groups of algorithms that we have presented in the earlier chapter, namely:

- GroupBy-Join, where join attribute = groupby attribute, and
- GroupBy-Join, where join attribute ≠ groupby attribute.

(a) *GroupBy-Join Cost Models, where Join Attribute = GroupBy Attribute*

As in the previous section, we focus only on the number of records being involved in the process, as the number of records determines the entire process. Based on this, there are three important measures in the GroupBy-Join query processing, whereby

the join attribute is the same as the group by attribute. These three measures also indicate the main steps of the process. They are: (*i*) number of groups and their counts produced by server 1 – this incurs three different costs: server processing time for the server to retrieve and produce the groups and the counts, download transfer time from server 1 to the mobile device, and upload transfer time from the mobile device to server 2 to be joined, (*ii*) number of records that participated in the joining operation in server 2.

- *Number of Groups and Their Counts:*

  As stated above, there are three stages which involve groups and their counts. The first one is the server time to retrieve the records from the local disk and produce the groups and their counts. Assuming that the hash-based aggregate algorithm is used, the cost for grouping and counting is:

  $R \times T_{groupcount}$

  where $T_{groupcount}$ is the server processing unit cost to produce the grouping and their counts.

  The data transfer cost from the server to the mobile device is then calculated as follows:

  $\sigma_g \times \pi \times R \times T_{down}$

  Lastly, the data transfer cost from the mobile device to the second server is:

  $\sigma_g \times \pi \times R \times T_{up}$

  Note that the last two equations are similar, except for the transfer unit cost, where one is the download unit cost, and the other is the upload unit cost.

- *Server Join Cost:*

  Once the second server has received the groups and their counts from the mobile device, it can start the joining process. The parameters for the joining process are the entire table *S* and the groups and counts from table *R*. Hence, the joining cost in the second server is calculated as follows:

$((\sigma_g \times \pi \times R) + S) \times T_{hashjoin}$

The first parameter is the number of groups and their counts, whereas the second parameter is the entire table *S*.

(b) ***GroupBy-Join Cost Models, where Join Attribute ≠ GroupBy Attribute***

There are two versions of this groupby-join processing. The first version is where the groupby operation is executed twice. The first groupby operation is done in one table, which is then joined with the second table, and then performs the second groupby. The second version is where the grouping is also done in the first table, but the group is not aggregated (or counted). The grouping is done merely for the joining purpose which is done next. The aggregating and counting is done after the joining process is completed

As previously discussed, we focus on the number of records affected by the process. The cost models for the two abovementioned versions are as follows:

Double Grouping Method Costs – Versions 1:

There are four important stages in the double grouping method. The first one is the number of groups and their counts produced by server 1. This incurs the server time, download (transfer) time, and upload (transfer) time. The second one is the joining process in the second server. The third is the join results which are downloaded to the mobile device (transfer time). Finally, the mobile device performs a post-processing which is the final counting.

- *Number of Groups and Their Counts*

  The server time is: $R \times T_{groupcount}$

  The download transfer time is: $\sigma_g \times \pi \times R \times T_{down}$

  The upload transfer time is: $\sigma_g \times \pi \times R \times T_{up}$

- *Joining Cost*

  The server two time to perform a join operation is: $((\sigma_g \times \pi \times R) + S) \times T_{hashjoin}$

- *Join Results Download Cost*

  The join result download cost is: $\sigma_j \times ((\sigma_g \times \pi \times R) + S) \times T_{down}$

  where σ_j indicates the join selectivity factor.

- *Final Counting Cost at the Mobile Device*

  The final counting cost at the mobile device is:

  $\sigma_j \times ((\sigma_g \times \pi \times R) + S) \times M_{compare}$

  where $M_{compare}$ is the unit cost to read and group records.

Single Grouping Method Costs – Versions 2:

Since there is no aggregating in the first phase (note that the grouping in the first stage is only for the block-based purposes), all records from the first server are sent block-by-block to the second server. Therefore, there is only one important cost in this method; that is, the server join cost including the join itself and the grouping as well. However, the entire table R still needs to be retrieved (and grouped) and sent to the second server to start the joining process.

- *Number Records Retrieved and Sent*

  The server time is: $R \times T_{retrieve}$

  The download transfer time is: $R \times T_{down}$

  The upload transfer time is: $R \times T_{up}$

- *Joining Cost*

  The joining cost in server 2 is: $(R + S) \times T_{hashjoin}$

7.5.4 Mobile Division Query Processing Cost Models

For mobile division query processing, there are two major groups of algorithms that we have presented in the earlier chapter, namely:

- Relational Division (count-based MDSP and count-based SSP), and
- Multiple Groups Division (Nested-loop and Aggregate-based).

(*a*) *Relational Division Cost Models*

There are two versions of the relational division query processing: count-based MDSP and count-based SSP. The general structure of the process consists of these stages: (*i*) count collection, (*ii*) count comparison, (*iii*) records loading, and (*iv*) the actual record group comparison.

For the MDSP version, count collection is done by the two servers which send their counts to the mobile device. There will be one count from the divisor table, but there will be multiple groups from the dividend table. The count comparison is performed at the mobile side. The qualified groups will need to load the records from the respective servers, and then the record group comparison is performed on the mobile side after obtaining the necessary records from the two respective servers.

For the SSP version, the count collection is also done by the two servers. However, only the count from the divisor server is sent to the mobile device, which is then passed through to the dividend server. Hence, the count comparison processing is done at the dividend server (as opposed to the mobile side as in the MDSP version). The actual record comparison, including the records loading, is done either in the dividend server or in the divisor table. Either way, the records from one server need to be transferred to the other server via the mobile device. Since the number of divisor records is generally smaller than the dividend records, it is desirable that the divisor table be sent to the dividend server via the mobile device.

MDSP Count-based Relational Division Costs

As there are four major components of the relational division processing, the cost models will be discussed in terms of these three components.

- *Counting and Count Collection Costs:*

  The counting process is done by the server, and hence it is a server cost. The server counting cost is then (assume table R is the dividend table, and table S is the divisor table):

  $(R + S) \times T_{groupcount}$

  The groups and counts are then transferred to the mobile device. The transfer cost is then:

  $(\sigma_g \times \pi \times R \times T_{down}) + (G_S \times T_{down})$

  The selectivity ratio σ_g indicates the proportion of the number of groups and the entire table R, whereas a new variable G_S indicates the size of the count record from the divisor table S, which is rather negligible. Therefore, the count download is normally dominated by the dividend.

- *Count Comparison Costs:*

  Once the mobile device has received the group counts from both servers, the count comparison is done by going through the group counts of the dividend and checking whether each group count is larger than the divisor count. If it is, then it is accepted, but if not, the group will be discarded from the dividend group. Since the cost will be dominated by the dividend group, it is basically linear to the size of the dividend groups (number of groups in the dividend table), which is:

  $\sigma_g \times \pi \times R \times M_{comparison}$

- *Record Loading Costs:*

  All records from the divisor table (server) need to be retrieved. The server retrieval cost and the record download cost are:

Server retrieval cost = $S \times T_{retrieve}$

Record download (transfer) cost = $S \times T_{down}$

Not all records from the dividend table (server) need to be retrieved and downloaded. This is because during the group comparison on the mobile side, some groups which have a lower count than that of the divisor table will be discarded immediately. The records from these groups will not be retrieved at all. However, in the dividend server, all records need to be scanned from disk before performing some selection. Therefore, the server retrieval cost is and the record download (transfer) costs are:

Server retrieval cost = $R \times T_{retrieve}$

Record download (transfer) cost = $\sigma \times R \times T_{down}$

The selectivity ratio of R indicates the proportion of number of records selected to be downloaded.

- *Record Comparison Costs:*

  The record comparison is done on the mobile side. This is basically a merging operation. However, the merging is done at the group level: one group from the dividend compared with the divisor. Hence, the divisor needs to be accessed as many times as there are groups in the dividend. Hence, the merging cost in the mobile side is:

  $((\sigma \times R) + (numdividendgroups \times S)) \times M_{merging}$

  The left-hand side term indicates that all records from the qualified groups from the dividend are accessed, and the second term in the above equation refers to the repeated access to the divisor as many times as there are the number of groups from the dividend. The processing unit cost is $M_{merging}$.

SSP Count-based Relational Division Costs

Similarly to the MDSP version, we focus on the four aspects.

- *Counting and Count Collection Costs:*

  Since the counting and count collection is done by both servers, the server cost for this is the same as that for the MDSP version, which is:

  $(R + S) \times T_{groupcount}$

  However, only the count of the divisor table is sent to the mobile device which is then passed through to the dividend server. Hence, the transfer cost is:

  $G_S \times (T_{down} + T_{up})$

  We can expect this cost to be very small and often negligible from the point of view of the overall cost.

- *Count Comparison Costs:*

  Count comparison is now done at the dividend server. From the costing point of view, it is almost identical to that of the MDSP version, with only one exception; that is, the processing unit cost is now $T_{comparison}$, instead of $M_{comparison}$, because it is done at the server side.

  $\sigma_g \times \pi \times R \times T_{comparison}$

- *Record Loading and Record Comparison Costs:*

  The record loading from the divisor table has the same cost as that of the MDSP version, with an addition that the divisor table needs to be uploaded to the dividend server. Hence, the total cost is:

  Server retrieval cost = $S \times T_{retrieve}$

  Record download (transfer) cost = $S \times T_{down}$

  Record upload (transfer) cost = $S \times T_{up}$

  Since the record loading for the dividend table is done in the server, the record loading cost for the dividend table will incorporate the record comparison as well. Although conceptually not all records from the dividend need to be compared, since some of them have already been eliminated during the count comparison, all records still need to be scanned from the server local disk and perform some selection. Another point is that, similar to the MDSP version,

the divisor will be accessed as many times as there are groups in the dividend. However, if we use a hash-based join in the server, the record comparison cost will be the join cost between the dividend and the divisor table which is:
$(R + S) \times T_{hashjoin}$

(*b*) *Multiple Groups Division Cost Models*

There are two versions of the multiple groups divisions: the sort-merge based and the aggregate-based.

Sort-Merge Multiple Group Division Costs

For the sort-merge version, there are only two major cost elements: (*i*) records loading, including sorting by the server, and (*ii*) record comparison in the mobile side.

- *Record Loading and Sorting Costs:*
  There are a few cost components for record loading and sorting. The first one is the server time, which includes record retrieval and sorting. The sorting itself already covers the retrieval which is:
  $(R + S) \times T_{sort}$
  Note that in the algorithm, although the dividend is downloaded one group at a time, at the end all groups must be downloaded anyway. Therefore, both complete tables *R* and *S* must be retrieved and sorted by the server.

  Subsequently, these records must be downloaded to the mobile side, which is:
  $(R + S) \times T_{down}$

  Note that when the size of main memory of the mobile device is not large enough to hold all records of the divisor, the divisor table needs to be chopped into several blocks, one block being enough for the main memory of the

mobile device. On the other hand, there will be no problem with the dividend, since we are downloading one block at a time.

However, as a consequence of the divisor table being chopped into several blocks, the entire dividend needs to be downloaded as many blocks as there are in the divisor. Hence, the transfer cost incurs some overhead depending on the number of times the entire dividend needs to be downloaded. In this case, we introduce a variable called *Replicate* which is greater than or equal to 1. If it is equal to 1, it means that there is no need to repeatedly download the dividend, since the entire divisor fits into the mobile device. However, if the divisor is divided into several blocks, this is when the dividend needs to be downloaded several times.

In the algorithm, we have explained that we may need to save some time by keeping the last group of the current block of the divisor. Hence, a slight saving of time can be expected which is determined by the value of *Replicate*.

Therefore, the revised transfer time considering the factor that the divisor table may not fit entirely into the mobile side is:
$(Replicate \times R + S) \times T_{down}$

- *Record Comparison Costs:*
  The record comparison cost is a mobile cost. If all records from both divisor and dividend fit into the memory of the mobile device, then the merging cost is simply:
  $(R + S) \times M_{merging}$

  However, again, if the divisor table does not entirely fit into the main memory of the mobile device, the *Replicate* variable needs to be used to factor the repeated access to the dividend table. Therefore, the cost is:
  $(Replicate \times R + S) \times M_{merging}$

Aggregate-Based Multiple Group Division Costs

The cost for the aggregate-based multiple group division is greatly similar to the count-based MDSP relational division. Basically, we request both servers to provide the mobile device with the groups and counts. Note that the only difference between the multiple groups and the relational division is that in the multiple group, the divisor will produce a number of groups and counts, instead of just one count as in the relational division.

Then the mobile device will do the group comparison. In the algorithm, we call this the pruning step. Basically, the outcome of this step is that some of the groups in the dividend will not be processed in the record comparison due to the inadequacy of the count.

Finally, the groups that are worth downloading will be retrieved and transferred to the mobile device for final record comparison.

- *Counting and Count Collection Costs:*

  The counting process done by the server involves a server time, which is:

  $(R + S) \times T_{groupcount}$

  The groups and counts are then transferred to the mobile device. The transfer cost is then:

  $(\sigma_{g1} \times \pi \times R \times T_{down}) + (\sigma_{g2} \times \pi \times S \times T_{down})$

  The selectivity ratio σ_g indicates the proportion of the number of groups from each table. Note that since the divisor may also produce a number of groups, it is indicated similarly to the dividend.

- *Count Comparison Costs:*

  This is literally the pruning step, and the cost is as follows:

  $(\sigma_{g1} \times \pi \times R \times M_{comparison}) + (\sigma_{g2} \times \pi \times S \times M_{comparison})$

- *Record Loading Costs:*

  Records from the groups which are not disqualified from the pruning phase will need to be retrieved from the respective server and downloaded to the mobile device.

  The retrieval cost is a sorting one, which is:
  $(\sigma \times R \times T_{retrievesort}) + (\sigma_g \times S \times T_{retrievesort})$

  And the download cost is:
  $(\sigma \times R \times T_{down}) + (\sigma_g \times S \times T_{down})$

  The selectivity ratio indicates the proportion of the number of records selected for retrieval and download.

- *Record Comparison Costs:*

  The record comparison is done on the mobile side. This is basically a merging operation. Although it is a simple merging, some groups from the divisor may need to be merged with a number of groups in the dividend. Therefore, the *Replicate* variable needs to be used. Hence, the cost is:
  $(Replicate \times R + S) \times M_{merging}$

7.6. Performance Evaluation Results

The aim of performance evaluation is to evaluate the effectiveness of each of the proposed techniques described in this thesis. The analytical models presented in the previous section analyze the elements of each processing component. These models are then incorporated into a simulation model, and the results of our simulation experimentations are presented in the following sections. We group the results based on the four key contributions in this thesis, which include mobile join processing,

mobile Top-*k* join processing, mobile groupby-join processing, and mobile division query processing.

In the simulation, we incorporate the analytical models presented earlier, and distributions functions are used, including normal, skewed, and random data distribution to simulate the record and processing distribution. In the experimentations, we particularly focus on communication costs (data transfer costs), and processing costs (server and mobile processing costs). The results are presented as follows:

7.6.1. Performance of Mobile Join Query Processing

For mobile join query processing presented in Chapter 3, we have conducted experimentations on the five basic mobile join techniques (MDSP1/2/3 and SSP1/2) to evaluate the communication or data transfer costs under several conditions and circumstances, the processing costs combining the server, mobile processing costs, the transfer cost as well as the processing cost for the block-based techniques compared with the aggregate block join.

Section (a) below presents the experimentation results examining the communication/data transfer costs of MDSP1/2/3 and SSP1/2. Section (b) below presents the experimentation results of the processing costs, covering server and mobile processing costs.

The block-based processing is examined in Section (c) and Section (d) below, where Section (c) examines the communication/data transfer costs, whereas Section (d) looks at the processing costs.

All of the proposed techniques earlier described in Chapter 3 are examined in this section, in terms of the communication and processing costs.

(*a*) Performance Results for the Communication/Data Transfer Costs of the Five Basic Mobile Join Techniques

Communication costs, or data transfer costs, are one of the main factors in mobile query processing. Hence, examining communication costs for each of the five mobile join query processing techniques, namely: MDSP1/2/3 and SSP1/2, is critical in order to understand the efficiency of each technique.

We have studied the communication costs, especially, in three areas: (*i*) the impact of selectivity ratio on the size of the information downloaded from the servers, (*ii*) the impact of the size of primary key on the overall record size on transfer costs, and (*iii*) the impact of an increasing number of records on the communication costs. By examining the performance results as plotted in the following four graphs (see Figure 7.1(a)-(d)), we have the following observations.

In studying the impact of selectivity ratio on the overall information size from the servers, we performed two experimentations: in the first, the size of the table in server 1 (i.e. table *R*) is much bigger than the size of table in server 2 (i.e. table *S*); in the second, both tables from both servers are equal in size. These are shown in Figure 7.1(a) and Figure 7.1(b) respectively.

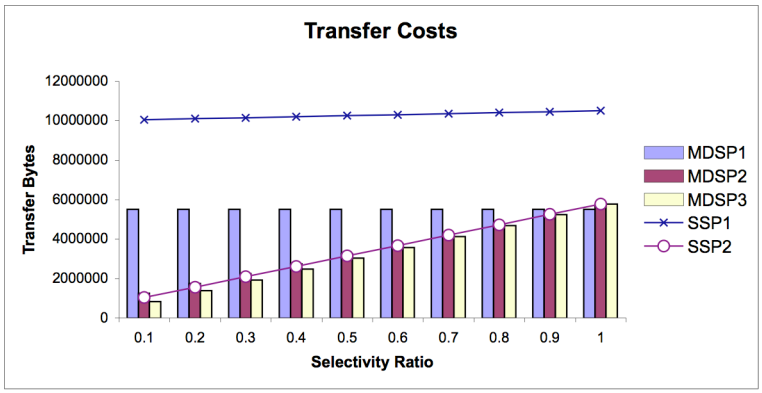


Figure 7.1(a). Impact of selectivity ratio when tables from both servers are of different sizes

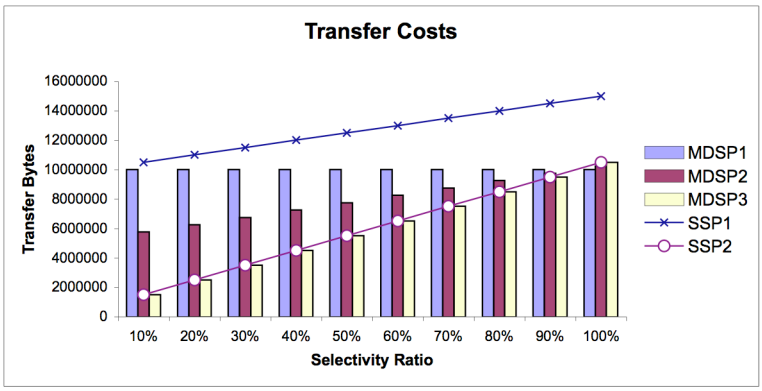


Figure 7.1(b). Impact of selectivity ratio when tables from both servers are almost of equal size

Figure 7.1(a) shows that when there is a large difference between the size of the two tables, the transfer costs for MDSP1 and SSP1 are quite constant, regardless of the selectivity ratio. The cost is dominated by the download of the original table from the server. In the case of MDSP1, the download is from both servers, but in SSP1, although the download is from one server, the mobile has to upload the same data to the other server. Additionally, for SSP1, results of the query have to be downloaded to the mobile, making the communication costs more expensive than MDSP1, because in MDSP1, the main data transfer cost is incurred by downloading tables from both servers only. SSP1 appears to be the most ineffective and it incurs the highest transfer costs compared with other techniques. As for the others (MDSP2/3 and SSP2), there is a clear gradual increase as the selectivity ratio increases, meaning that transfer cost increases with the percentage of match (selectivity ratio). Although these three techniques offer a similar effectiveness, MDSP3 is slightly better than the rest, although it might be ineffective when the selectivity ratio is extremely high.

Figure 7.1(b) shows the transfer costs when the size of the tables from both servers is almost equal. There are a few similarities and differences between Figure 7.1(a) and Figure 7.1(b). In terms of the ranking efficiency of the five methods, MDSP3 is still better than the rest, and SSP1 is still the worst; moreover, the SSP1 transfer cost now increases as the selectivity ratio increases. The increase is due to the

number of matched records which need to be downloaded to the mobile device after the server processing. Also, note that MDSP2 suffers from an increase in the communication cost for a similar reason. On the other hand, for MDSP3, because initially we download only the primary keys from both tables, instead of the full tables, the increase of overhead for the increasing selectivity ratio is quite small.

Figure 7.1(c) studies the impact of the proportion of the primary key size to the record size. Although this does not affect MDSP1 and SSP1, as both download the full tables, it affects the others, especially MDSP2/3 and SSP2, because these three techniques download the primary key data in the first instance. Consequently, MDSP2 and MDSP3 perform the best. SSP2, on the other hand, offers better performance when the primary key size is small.

Figure 7.1(d) shows the impact of various sizes of the tables from the two servers. Note the increases of MDSP1 and SSP1 which are phenomenal. MDSP2 also increases its transfer cost, but the most efficient are still MDSP3 and SSP2.

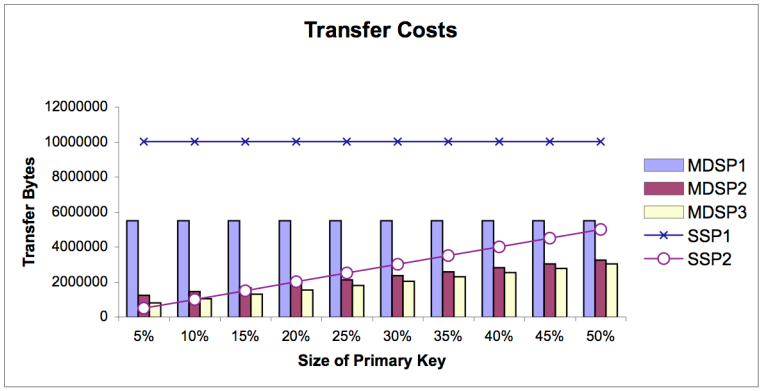


Figure 7.1(c). Impact of size of primary key on transfer costs

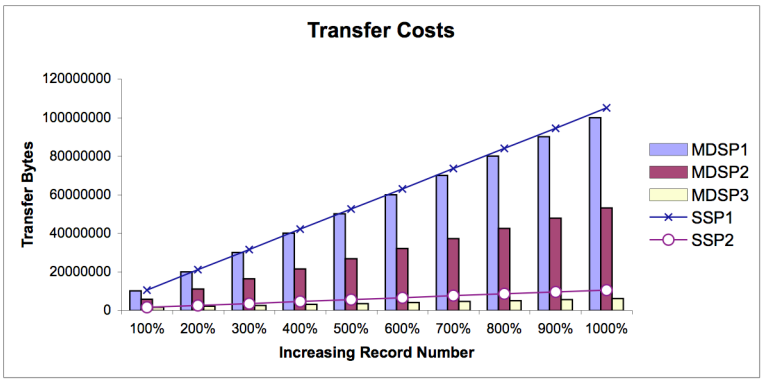


Figure 7.1(d). Impact of increasing number of records on transfer costs

From this study, it can be noted that SSP2 and MDSP3 outperform the other techniques, in terms of their data transfer costs. Therefore, if server processing is not possible due to the constraint imposed by the mobile environment where the server serves only data broadcasting but does not accepts users' queries, then an MDSP3-like approach is the most appropriate.

(*b*) Performance Results for the Processing Costs of the Five Basic Mobile Join Techniques

The processing costs are comprised of server and mobile processing costs. Although MDSP1/2/3 techniques mainly process the query on the mobile device side, the server costs are incurred when tables are retrieved from the server. Similarly to the above study on data transfer costs, for the processing costs, we also examine the impact of the selectivity ratio, proportion of the primary key size to the overall record size, and the increasing number of records. Moreover, we additionally examined the use of hash and sort-merge join in MDSP1. The hash-join does not require the data to be sorted, whereas the sort-merge join requires the data from the servers to be sorted first in the server.

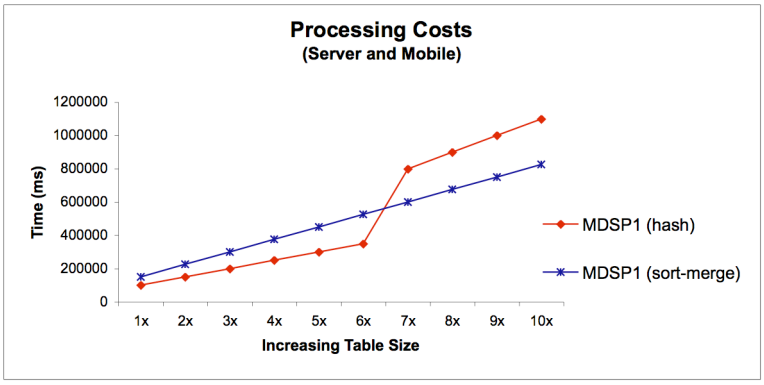


Figure 7.2(a). Hash vs. Sort-Merge for MDSP1

Figure 7.2(a) shows a comparison between hash and sort-merge join for MDPS1. It shows that the hash-join method is quite effective, especially when the number of records to be processed in the mobile device is quite small and fits into the mobile device main memory. Once the table size increases (e.g. double, triple, etc), the hash-join processing may incur some memory overflow overheads which increases the overall processing costs dramatically. On the other hand, the sort-join method performs consistently over the increase of number of records. The main processing costs were the sorting costs in the server and the merging cost in the mobile device. The result in Figure 7.2(a) concludes that in the context of mobile device processing, it is desirable to share some heavy process with the most powerful machines, namely the server. This is the case with the sort-merge join where the sorting is carried out by the server. As a result, in some cases, the hash-join performs worse than the sort-merge join, since the hash-join method solely rely on mobile processing, and does not take an advantage that the server may be able to do some preliminary work. This observation is one of the main contributions of the thesis, and in particular for the first sub-topic of the thesis focusing on mobile join query processing (see Chapter 3).

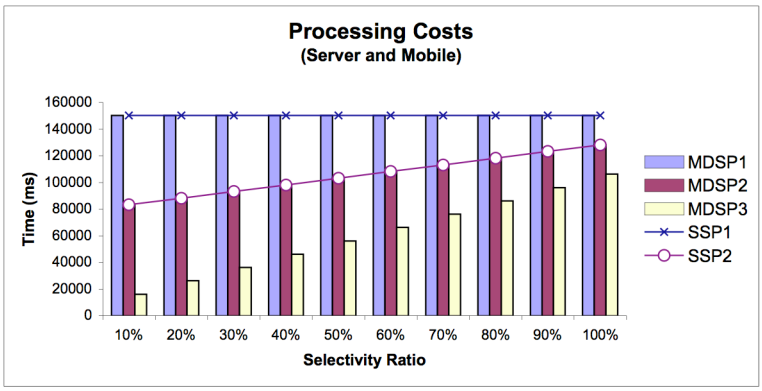


Figure 7.2(b). Impact of selectivity ratio to processing costs

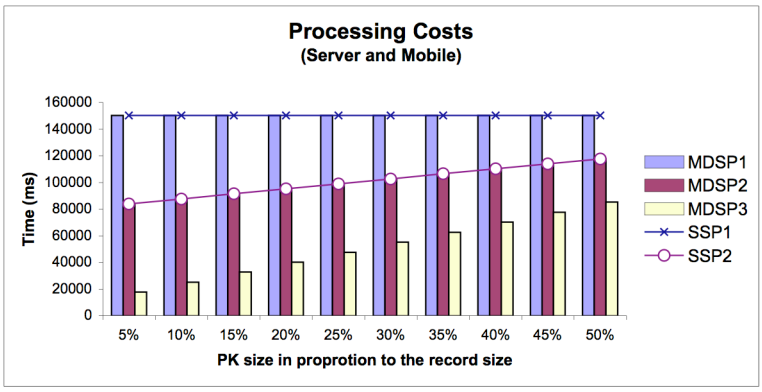


Figure 7.2(c). Impact of size of primary key to processing costs

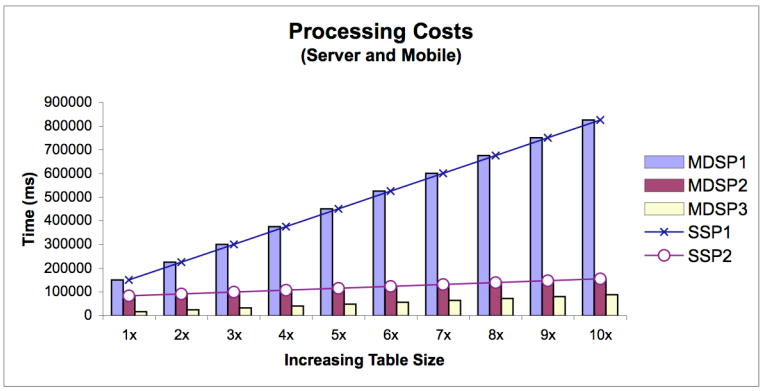


Figure 7.2(d). Impact of increasing number of records to processing costs

Figures 7.2(b)-(d) show the processing costs when several variables, including selectivity ratio, primary key size, and record numbers, are increased. Figure 7.2(b) shows that the performance of both MDSP1 and SSP1 is quite poor disrespecting the selectivity ratio. MDSP2 and SSP2 are also inefficient either. This is due to the

increase of the results to be produced, as the selectivity increases. The best performance is delivered by MDSP3. Figure 7.2(c) shows a very similar pattern to Figure 7.2(b). The only minor difference is that MDSP2/3 and SSP3 perform slightly better.

Figure 7.2(d) shows that when the number of records is extremely large, MDSP1 and SSP1 are not worth exploring. The other three techniques, MDSP2/3 and SSP2 cope quite reasonably well with the increase of record numbers. This is due to that these methods download the primary key data only in the first instance. The increase of the overhead would be due to the number of matches as the result of the query, in which non-primary key data needs to be downloaded. However, if the selectivity ratio is relatively small, the impact of number of records to the overall transfer costs will be minimal, and this is shown by the performance of MDSP2/3 and SSP2.

Overall, from the processing point of view, MDSP3 processing time is the most efficient. This is due to the minimum processing cost to join primary keys from both the tables from both servers. Having small size of operands of the join operation is shown to offer the most efficient processing costs.

(*c*) Performance Results for the Communication/Data Transfer Costs of the Block-based Mobile Join Techniques

For the block-based mobile join techniques, we compare the performance of the block-based *BBP* techniques and the aggregate block join *ABJP*. Both are block-based processing as explained in Chapter 3, but *ABJP* groups the block-based on an aggregate function, whereas *BBP* groups each block statically.

We are particularly interested in examining two factors in block-based processing, particularly, the impact of groups that is being pruned due to the elimination of a number of groups before the join processing takes place, and the impact that selectivity ratio brings. Since there are two techniques, MDPS and SSP, for each block-based, we show 4 graphs (see Figures 7.3(a)-(d)), two each on group pruning and selectivity ratio.

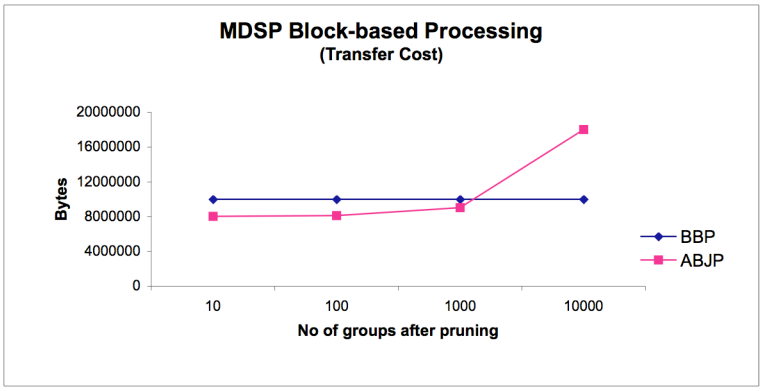


Figure 7.3(a). Impact of number of groups pruned in MDSP block-based transfer costs

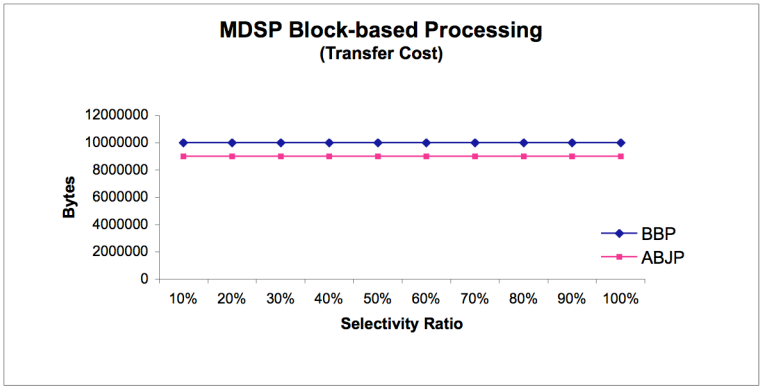


Figure 7.3(b). Impact of selectivity ratio in MDSP block-based transfer costs

Figure 7.3(a) shows that when the there are a lot of groups being pruned by the initial grouping, ABJP technique performs quite well compared with BBP. However, when there are so many groups to be processed, ABJP perform poorly. We need to reiterate that when the initial groups are formed in ABJP, it is likely that some groups are naturally eliminated, and hence do not contribute to the join process. This group pruning is not applicable to BBP, since in BBP, all groups are processed. Therefore, the performance of BBP is constant as shown in Figure 7.3(a).

Figure 7.3(b) shows that the transfer costs of both BBP and ABJP are constant despite the increase of the selectivity ratio. This is because the transfer cost is not affected so much by how many records are being downloaded. Nevertheless, ABJP

performs slightly better than BBP, as also indicated in the previous figure (Figure 7.3(a)).

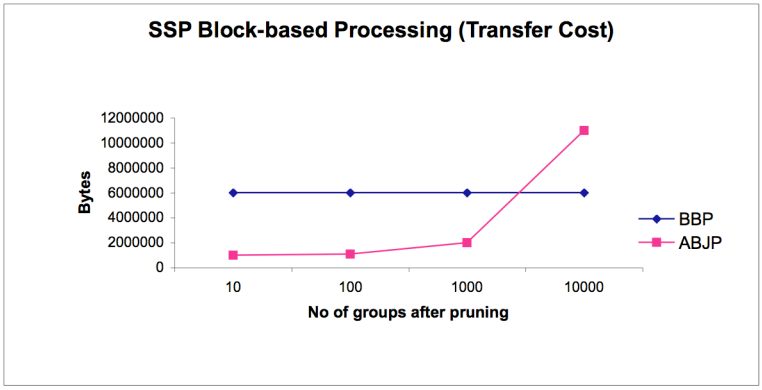


Figure 7.3(c). Impact of number of groups pruned in SSP block-based transfer costs

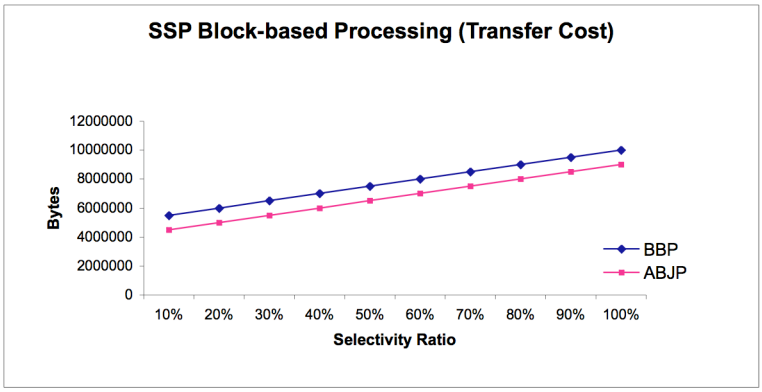


Figure 7.3(d). Impact of selectivity ratio in SSP block-based transfer costs

Figure 7.3(c) focuses on the SSP techniques in both BBP and ABJP styles. The trend shown in Figure 7.3(c) is quite similar to that of Figure 7.3(a), with a slight difference that in SSP, the ABJP performance is much better than in BBP. Note that the difference can be up to 6 folds compared to the MDSP. The similarities however are that BBP is constant, and transfer cost for ABJP increases steeply when the pruning is not in effect.

Figure 7.3(d) shows the impact of the selectivity ratio. Although both BBP and ABJP increase linearly, this is quite different from Figure 7.3(b) where both are constant. In SSP, the transfer cost overhead is due to the number of matches to be

downloaded after the server processing. It does not matter whether it is BBP or ABJP, the SSP block-based method will have more transfer cost if there are more matched records. On the other hand, for MDSP block-based, because the transfer cost is not affected by how many records that are matched by the query, as a result, the transfer costs for MDSP are quite constant.

Overall, ABJP performs slightly better than BBP in both MDSP and SSP techniques. The only draw back is that when the pruning of the initial groups is not effective, only then the transfer cost for ABJP will be expensive. In the experimentations, we assume that the user does not terminate the process without getting all join query results, and consequently, the original block method (BBP) does not offer much benefits compared with the non-block methods. However, ABJP is able to offer some advantages by pruning records from some aggregate groups, and from this point of view, ABJP can be attractive.

(*d*) Performance Results for the Processing Costs of the Block-based Mobile Join Techniques

Because the performance of the block-based is quite constant when varying the selectivity ratio as shown previously in Figures 7.3(b) and 7.3(d) only, in the processing costs, covering the server and mobile processing costs, we focus on the impact of group pruning to the processing costs.

Figure 7.4(a) shows that in terms of the processing costs, ABJP is only as good as BBP when the group pruning is very effective. Basically the main processing cost is the retrieval and the grouping costs done by the server when the mobile device requests the groupings. In this case, all records must be retrieved from the server disk to be processed by the server. The mobile processing cost is quite minor because of not only the pruning but also the absence of disk access. The records received from the servers are immediately processed in the mobile device through the merging process without the need to access any local disk on the mobile device, because the data is not stored in mobile device local disk. This is applicable to both BBP and ABJP. The increase in processing cost in ABJP when the group pruning is ineffective

is due to the mobile processing cost, because more records need to be merged. On the other hand, the processing cost for BBP is only slightly affected by the group pruning, which cannot be seen clearly in the graph.

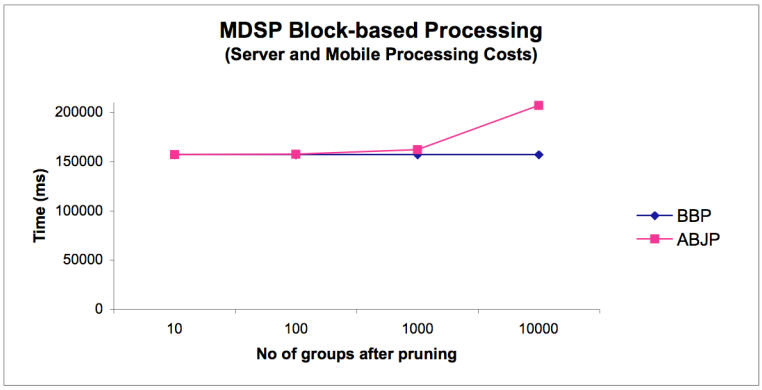


Figure 7.4(a). Impact of group pruning in MDSP block-based processing costs

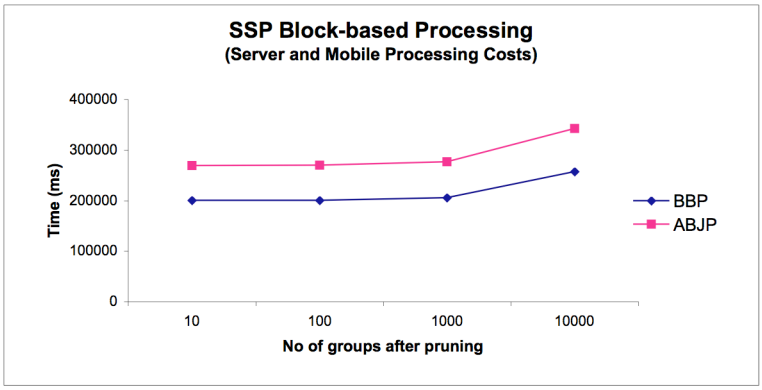


Figure 7.4(b). Impact of group pruning in SSP block-based processing costs

Figure 7.4(b) shows the SSP processing costs. The graph shows that BBP is comparable with ABJP. The difference is very little. When the pruning is quite effective, most of the processing costs will be dominated by the data retrieval and grouping in the server, which is mainly the disk access cost. Also note that BBP the increase in the processing cost when the group pruning is ineffective can been seen more clearly compare with the MDSP method in Figure 7.4(a).

Comparing the processing costs between MDSP and SSP, SSP processing cost is more expensive. This is because in SSP, the initial group pruning which is done in

the mobile device is not fully beneficial, because in the server (SSP), only one side is pruned, whereas the other side is the table from the server which is doing the server processing job. On the other hand, in MDSP, the group pruning receives a full benefit, because if any side from the join is pruned, then the join processing in the mobile will be very light. This is not the case with SSP. Additionally, the SSP ABJP processing cost is worse than SSP BBP, and this is due to the server join processing overhead which is

7.6.2. Performance of Mobile Top-*k* Join Query Processing

For mobile Top-*k* join, we have done a number of studies. One is to examine the impact of block size (*k*) to the communication cost. It is noted in Chapter 4 that if the block size is large enough that the Top-*k* results are generated, then there is no need to download additional data blocks. Conversely, if the block size is small which subsequently results in more blocks to be downloaded becomes necessary. Another study on block size (*k*) is to determine the impact to the processing cost in term of number of records processed.

Apart from the block size (*k*), we did study on the number of records processed or percentage of record processed to the overall processing cost and mobile main memory requirements. In Top-*k* join processing, as explained in Chapter 4, that there is a stopping condition when all records ranked higher than the lowest matched ranked records have been processed. Consequently, not all records may need to be processed in order to correctly generate Top-*k* results. Therefore, the study on the impact on what percentage of number of records processed to the overall performance is important.

(*a*) Performance Results for Mobile Top-*k* Join based on the Block Size

Figure 7.5(a) shows the impact of the block size (*k*) to the communication costs. It also shows the percentage of records processed in order to generate the Top-*k* results. We studied the extreme size of the block size. Block size=1 means that individual record is a block, whereas the block size=100 means that the entire table is considered one block. In this study, the number of records in both servers is only 100 records.

Note that when the entire table is one block, the communication cost is quite poor. This is understandable because although not all records will contribute to the final Top-*k* ranking, all records must be downloaded. With a reduce block size, the communication cost is also reduced, until a point in time where the reduction in block size is unnecessary.

At the other extreme, when each individual record is considered a block, a lot of blocks need to be downloaded, and for each block downloaded, the mobile device needs to send a query to obtain a block. Hence, the more blocks to be downloaded the more queries need to be sent. The overhead decreases quite steeply when the block size increases up to around 10. And then the communication costs are quite constant before it peeks again. The point where the communication costs start to increase indicate the point where it is roughly equal to the required records processed to generate the Top-*k* results. For example, in the graph, we reported 4 cases: 30%-50% and 90% number of record processed. Looking at the 90% record processed to generate the Top-*k* results, the communication costs will only increase if the block size is more than 90%. If the query needs to process 30% of the total number of records, then the block size more than 30% will generate more expensive communication costs. It is known that predicting percentage of records to be processed is difficult. However, because the communication costs are quite constant after an initial drop, therefore, it is reasonable to set the block size to be small, just after the initial drop in the communication cost. After this, there is a degree of freedom, before the communication costs pick up again.

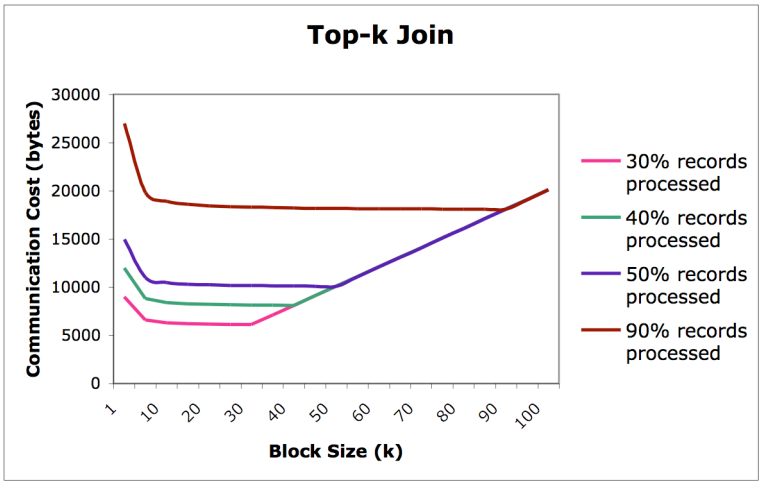


Figure 7.5(a). Impact of block size (*k*) to communication costs

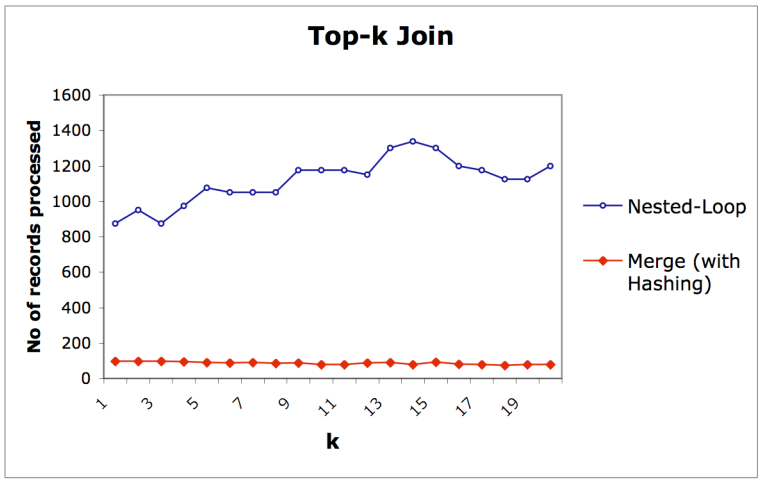


Figure 7.5(b). Impact of block size (*k*) to number of records processed

Figure 7.5(b) shows a second study of the block size. In this case, we compared the performance of the nested-loop and merge with hashing approaches. We used a random number generator to generate the probability for matching records. It clearly shows that nested-loop performs poorly compared to the merge with hashing technique. The nested-loop process increases with the increase of block size. The increase of block size means that more records need to be compared to generate the query results. Although in some cases, especially good cases, whereby the Top-*k* results are generated quite quickly without much comparison, even though the block size is small. However, after running the experimentations repeatedly and averaging

the results, it clearly shows that the bigger the block size the more records to be processed.

On the other hand, for the merging with hashing technique, the increase in number of records processed cannot be seen clearly from the graph, because they are quite low. Therefore, it looks like that the merge with hashing technique offers quite a constant performance. The merge with hashing technique offers a much better performance because of quite a linear complexity of the algorithm, compared with the nested loop with exponential complexity. Therefore, it is quite common to expect that the merging technique is the best choice.

(*b*) Performance Results for Mobile Top-*k* Join based on the Percentage of Records Processed

The second study on Top-*k* join is to examine the impact of number of records processed. Obviously that when the number of records processed to generate the Top-*k* results increases the performance degrades. But we would like to examine to what extent that this is true.

Figure 7.6(a) shows that the merge with hashing technique increases the processing cost quite steadily, compared with that of the nested-loop approach. Even when all records need to be processed, the merge with hashing technique performs quite well, whereas the nested-loop hits the roof. This again shows the superiority of the merge with hashing technique for mobile Top-*k* join query processing.

Figure 7.6(b) shows that impact to mobile memory requirement. The mobile memory requirement can be explained as follows. When a block from both servers has been downloaded, they occupy the main memory of the mobile device. If a second block needs to be downloaded to the mobile device because the Top-*k* records have not been generated from the first block, if we use a nested loop approach, both first and second blocks need to stay in the mobile, because the nested-loop process itself will compare records across different blocks. Consequently, the more blocks need to be downloaded, the more space in the mobile main-memory needs to be used.

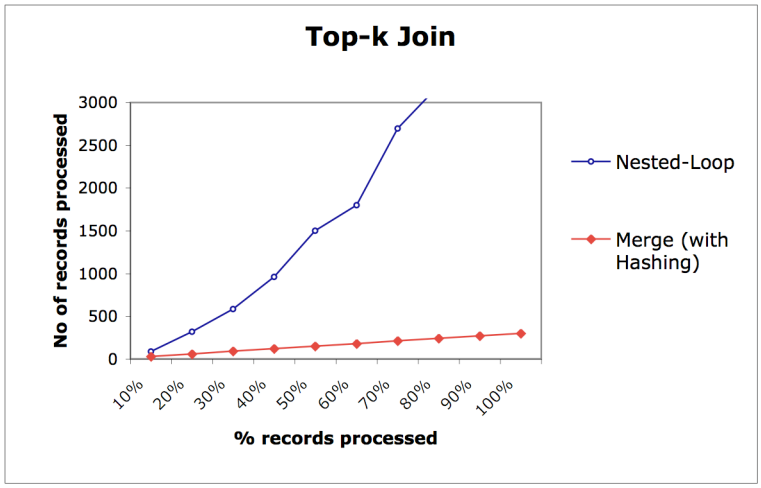


Figure 7.6(a). Impact percentage of records processed to number of records processed

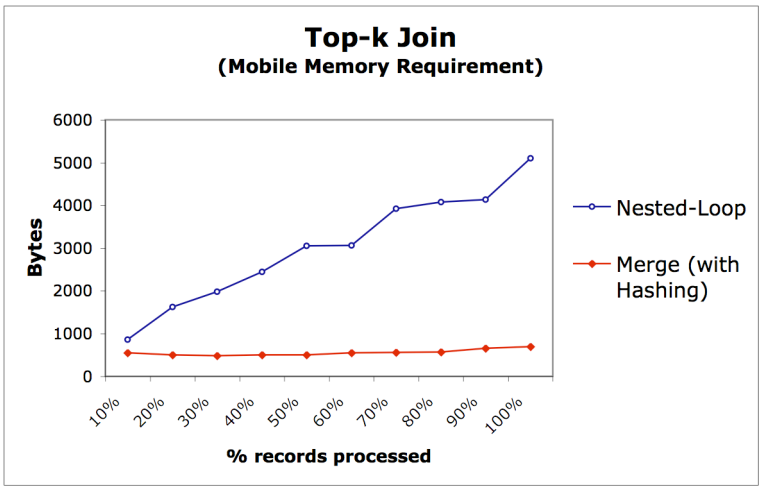


Figure 7.6(b). Impact percentage of records processed to mobile memory requirement

Conversely, if we use a merge approach, when a second block is downloaded, the first block can be removed from the mobile main-memory. The only exception is that in merging, we need to maintain an '*Incomplete*' list as described in Chapter 4. If we use a hash table to maintain the incomplete list, then the merging will only incur an overhead to store the hash table. However, this overhead is relatively small compared to the overhead in nested-loop. Therefore, the graph shows that the merging technique's performance is relatively quite stable.

7.6.3. Performance of Mobile Groupby-Join Query Processing

For mobile groupby-join query processing, we have studied the two groupby-join query category, in particular, the groupby-join attribute where the join attribute = groupby attribute, and where the join attribute ≠ groupby attribute. For the former group, we have proposed a groupby-before-join technique, as described in Chapter 5, whereas for the latter group, we have come up with two techniques: double grouping and single grouping. Since processing group by as early as possible is more desirable, in the performance evaluation, we examined the impact of number of groups to the performance; both transfer and processing costs. Some of the experimentation results are presented as follows:

(*a*) Performance Results for Mobile Groupby-Join based on Join attribute = Groupby attribute

There are two proposed methods for groupby-join where join attribute = group by attribute. Our methods are based on MDSP and SSP. In our proposed techniques, we employ a strategy whereby the group by operation is carried out first, before the join operation. In the experimentations, we compared our proposed technique with an existing technique whereby there is no notion of groupby-before-join, implying that the join operation is carried out and then groups the join results to produce the final query results.

Figures 7.7(a)-(b) focus on the impact of number of groups to transfer costs, using MDSP and SSP respectively. Both figures show that when the number of groups is low, meaning that the groupby table has been summarized into only a few groups, the proposed groupby-before-join techniques for both MDSP and SSP outperform the traditional techniques. Figure 7.7(a) shows that the performance of the proposed groupby-before-join technique is quite stable when the group numbers is below 1000. Note that the total number of records used in the experimentations was 10,000 records. This proves that performing the groupby operation first whenever possible give performance benefits. On the other hand, for the traditional groupby-after-join technique, the performance is quite constant, regardless the number of

groups. This is because the performance is occupied by the join operation itself, and hence the number of groups produced by the group by operation plays a minor role only. The performance of the proposed groupby-before-join will degrade when the reduction in the number of groups produced by the group by operation is only very little, that is when the number of groups is large. This means that each group consists of a few number of records only. The performance of the proposed technique under this condition will not deliver a strong performance, compared with when each group consists of a large number of records.

Figure 7.7(b) shows a similar pattern to Figure 7.7(a). However, there are a few main differences. Once is that the performance of the proposed technique is extremely good compared with the proposed technique using the MDSP approach. This can be explained as follows. Using the SSP approach, the proposed technique downloads only the groups from one server, and then upon receipt, the groups are sent to the other server. If the number of groups is low, then the transfer cost will also be low. For the MDSP approach as shown previously in Figure 7.7(a), the mobile device needs to download the groups from one server and the complete table from the other, making the transfer cost quite substantial compared with that of the SSP approach. Hence for the MDSP approach, most of the transfer cost will be taken by the download of the full table from the non group by table.

The second difference is that the transfer costs for the traditional groupby-after-join as shown in Figure 7.7(b) go up notably when the number of groups produced is large. This is due to the results of the query which need to be downloaded to the mobile device. On the other hand, for the MDSP approach in Figure 7.7(a), the traditional technique is not influenced greatly by the large number of groups, because the process is already done in the mobile device, and hence the final query results are already in the mobile device. It does not require any further data transfer.

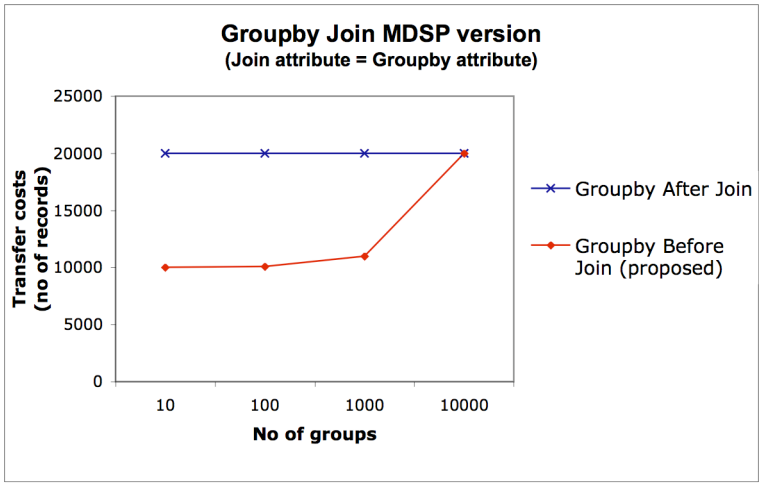


Figure 7.7(a). Impact of number of groups to MDSP transfer costs

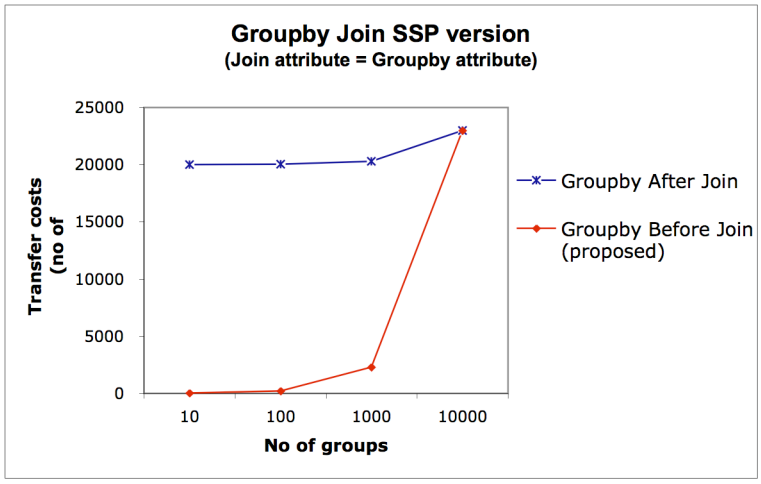


Figure 7.7(b). Impact of number of groups to SSP transfer costs

Figures 7.8(a)-(b) focus on the impact of number of groups to processing costs covering server and mobile processing. For the proposed groupby-before-join techniques, we examined the use of sort approach and hash approach. The same is applied for the traditional technique using groupby-after-join technique.

The sort approach is where the groups produced by the group by server and the entire table by the other table are first sorted by the servers. Consequently, upon receipt, the mobile device when performing the join, it can employ a merge operation. On the other hand, if the servers do not sort the groups (in the group by server) and the entire table (in the other server), once the mobile receives the required

information, the join operation in the mobile device needs to employ a hash-join technique. The results from mobile join query processing presented in the earlier sections conclude that when the data is large, the hash-join technique may not be suitable to be used in a mobile device. It is then desirable in this case that the sorting is pre-done by the server so that the mobile device can do a merging only. We examined this also in groupby-join query processing.

Figure 7.8(a) shows that both approaches (sort and hash) of the proposed technique (groupby-before-join) outperform both approaches of the traditional technique (groupby-after-join). This strengthens the results shown previously in the transfer costs where the proposed technique outperforms the traditional technique.

Now let us examine the behaviour of the sort and hash approaches of the proposed technique. Note that the hash approach performs better than the sort approach when the number of groups is small. When the number of groups is small, the join operation which takes two operands - the groups from one table and the entire records from the other table - the hash approach is able to hash a small number of groups to a hash table without any memory overflow. The entire table from the second server will be hashed and probed only to the hash table, and this does not incur any memory overflow either. Hence, the performance of the hash approach is excellent when the number of groups is small as shown in Figure 7.8(a). But when the number of groups becomes larger, there is a point where the hash table for the groups will no longer fit into the main memory of the mobile device, and this will incur a memory overflow overhead. This can be seen as a sharp rise in processing costs when using the hash approach of the proposed technique; therefore, the sort approach delivers a better performance. With the sort approach, the processing costs will increase at a later point in time, only when the number of groups is large. This is due to the actual merging operation in the mobile device which needs to merge more records to produce the query results.

For the traditional techniques, whether they be the sort or hash approaches, the performance is generally worse than the proposed techniques. There are a couple of things to note for the traditional techniques. One is that the hash approach is very expensive. One reason for this is the memory overflow overhead, because the entire

tables from the two servers need to be joined in the limited main memory mobile device. Hence, the number of groups is irrelevant. For the sort approach, since the sorting is carried out by the respective servers, the mobile device does only the merging operation, and because merging does not incur any main memory overflow overhead, it delivers a better performance than the hash approach. Nevertheless, the traditional techniques generally cannot deliver a better performance than the proposed techniques.

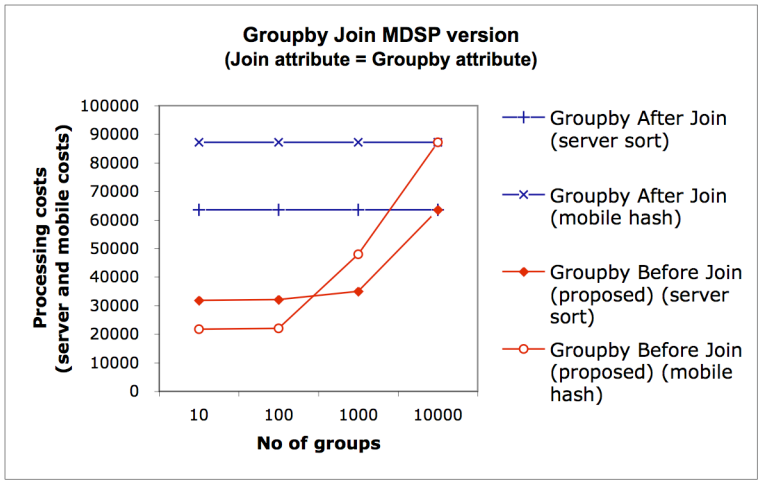


Figure 7.8(a). Impact of number of groups on MDSP processing costs

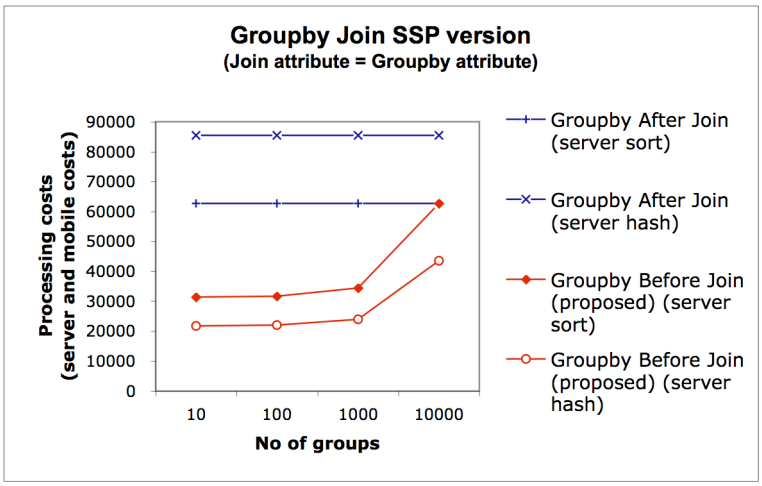


Figure 7.8(b). Impact of number of groups on SSP processing costs

Figure 7.8(b) shows the performance for SSP. Although the trend is quite similar to that of MDSP (see Figure 7.8(a)), there are several differences. In SSP, the join operation is done by the server. The first server of course does the grouping (for the proposed approach) and the results are sent to the second server to do a join operation. Since the server has a bigger main memory, the memory overflow overhead for the hash approach is quite small. Therefore, as shown in Figure 7.8(b), the hash approach consistently delivers a better performance than the sort approach. The second difference is that in Figure 7.8(a), when there is main memory overflow in the mobile device, the hash approach performance becomes worse and at some point in time when the number of groups is very large, is worse than the traditional techniques which rely solely on join operation before groupby. This does not happen in SSP, because main memory overflow in the server is reduced.

Overall, the proposed techniques, whether they be MDSP or SSP based, deliver a better performance than the traditional techniques. The level of superiority of the proposed techniques is of course influenced by how much pruning can be achieved before the join operation is carried out. If the pruning is high, meaning that the number of groups is small, then the proposed techniques will be able to deliver top performance. The performance will slightly degrade when the pruning is no longer that high.

(*b*) Performance Results for Mobile Groupby-Join based on Join attribute ≠ Groupby attrribute

For groupby-join where the join attribute is different from the groupby attribute, the existing techniques are heavily dependent on the groupby-after-join techniques. In this thesis, we argue that it is possible to perform a group by operation in many circumstances, and hence we need to pursue this direction. The results in the previous section on the performance of the proposed groupby-before-join method are very promising, providing an increased incentive for pursuing a groupby-before-join technique for groupby-join operation where the join attribute is different from the groupby attribute.

We have divided groupings into two categories, namely double grouping and single grouping. We have studied these two categories, particularly in terms of the impact of the number of groups on the performance. Figures 7.9(a)-(b) examine the performance of the double grouping category, whereas Figures 7.10(a)-(b) depict the performance of the single grouping category.

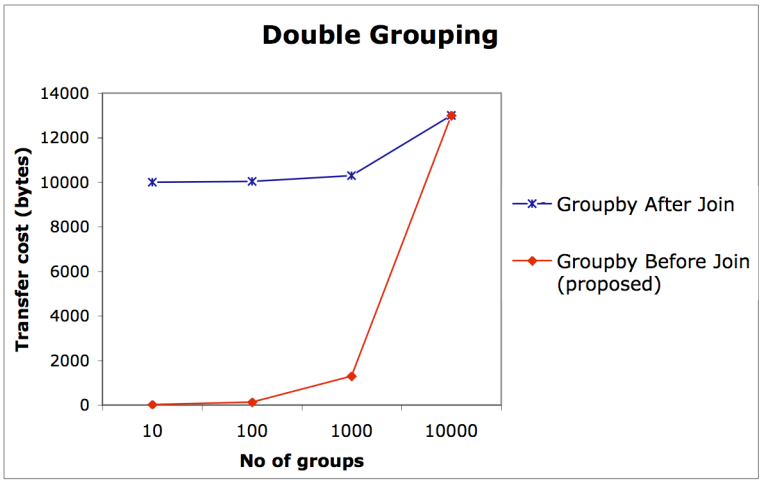


Figure 7.9(a). Impact of number of groups on transfer costs in double grouping

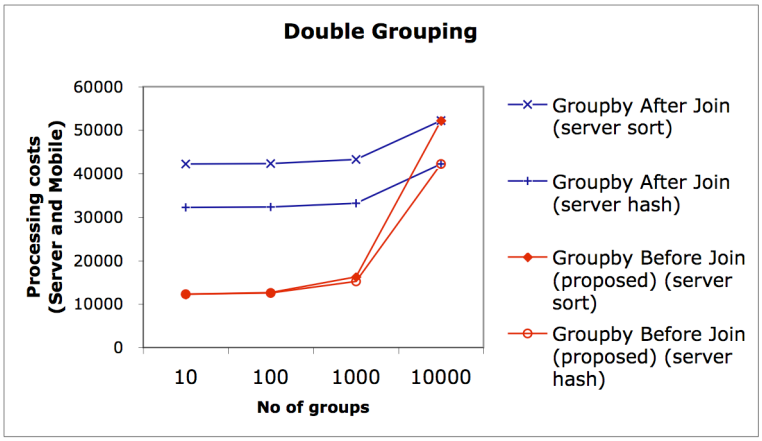


Figure 7.9(b). Impact of number of groups on processing costs in double grouping

Figure 7.9(a) shows the impact of the number of groups on transfer costs. The trend is quite similar to that indicated by the previous graphs. However, there is one

notable difference: the performance of the proposed groupby-before-join is extremely good. We need to reiterate that the proposed method combines the MDSP and SSP approaches. In the double grouping method, the first grouping is done by the first server, then the groups are passed to the first server through the mobile device, and the join is carried out by the second server. The second grouping is carried out in the mobile device. Since the first grouping operation and the join operation are carried out by the server, if the join results are quite small due to a low selectivity ratio and small number of groups produced by the first grouping operation, then the transfer cost will also be very low.

On the other hand, the traditional approach performs the join operation first. Hence, the transfer cost is based purely on the entire table size. Regardless of whether an MDSP approach or an SSP approach is used, the transfer cost will be excessive, as is shown in Figure 7.9(a).

Figure 7.9(b) studies the processing costs. As in the previous section, we studied the processing costs based on sort and hash. Note that the performance of both sort and hash approaches for the proposed method is quite comparable. This is because the join operation is done in the second server. The first server performs the first grouping operation, and the second server carries out the join operation. Although the unit cost for hashing and sorting is different, there is not much difference in the overall cost, because one of the operands in the join operation which is the number of groups is quite small. The sorting or the hashing of the other table requires a similar processing time, because the sorting operation uses a hashing technique as well, since in the experimentations, the number of records does not exceed the available main memory of the server; hence, an internal sorting method is used.

For the traditional technique, the hash approach gives a slightly better performance than the counterpart sort approach. This is again because the main memory overflow in the server is negligible, although there is no pruning done prior to the join operation.

Overall, the proposed technique based on groupby-before-join technique gives a better performance compared with the traditional groupby-after-join technique. The

use of groupby-before-join technique in this category of query is one of the main contributions of this thesis, and the performance of the proposed technique is also superior.

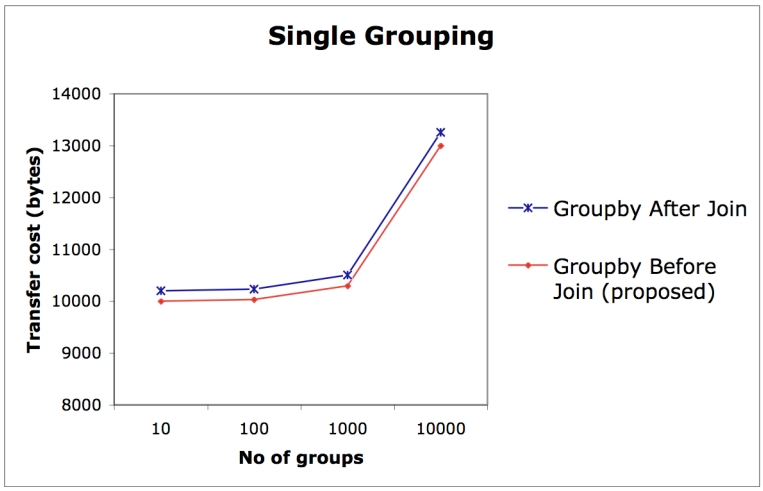


Figure 7.10(a). Impact of number of groups on transfer costs in single grouping

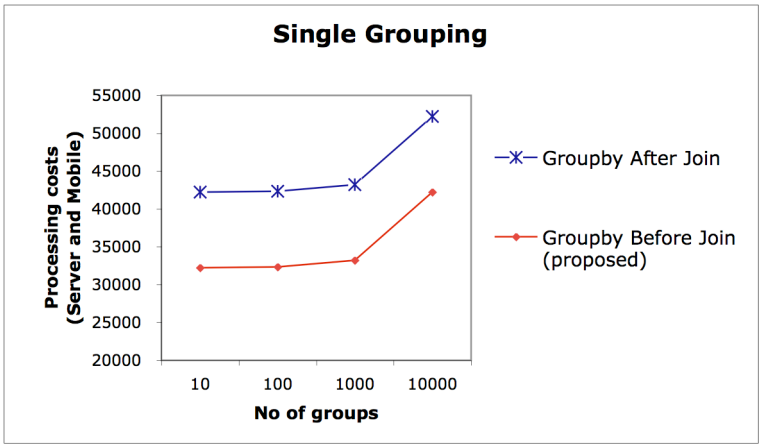


Figure 7.10(b). Impact of number of groups on processing costs in single grouping

For the single grouping methods, we also examined their performances. Figure 7.10(a) shows that the proposed method does not give a significantly better performance than the traditional technique. We need to reiterate that in the single grouping technique, the first server which does the grouping does not aggregate the groups. So, basically all records from the first server actually contribute to the transfer

cost, and this is not any different from the traditional technique. This explains why the single grouping approach of the proposed technique does not perform any better than the traditional technique.

Figure 7.10(b) shows that the processing performance of the proposed technique is consistently better than that of the traditional technique. This is due to the grouping produced by the first server, which is then used in the merging operation by the second server in the joining process. However, since the transfer unit cost is much higher than processing costs, the transfer costs will dominate the overall performance. As a result, both the proposed technique and the traditional technique are quite comparable in terms of their overall performance. In short, the groupby-before-join method for groupby-join queries where the join attribute is different from the groupby attribute performs well with the double grouping option. Hence, it is critical to identify whether a groupby-join query, where the join and the groupby attributes are different, falls into the double grouping option. If it does, then our proposed method can deliver an excellent result.

7.6.4. Performance of Mobile Division Query Processing

For mobile division query processing, we studied the two areas of relational division and multiple group division. The relational division queries are quite common in relational databases, but we have additionally proposed multiple group division for mobile query processing. Some of our experimental results are presented in the following:

(*a*) Performance Results for Mobile Relational Division Queries

In the experimentations, we have focused particularly on the impact of dividend size (number of records in the dividend table) and the impact of selectivity ratio, on both transfer and processing costs.

In mobile relational division query processing, we have proposed three techniques, namely MDSP relational division, SSP dividend, and SSP divisor. The difference between the two SSP techniques is where the division is carried out; that is,

in the dividend server or in the divisor server. In the performance graphs, we compare the three techniques to examine the efficiency of each.

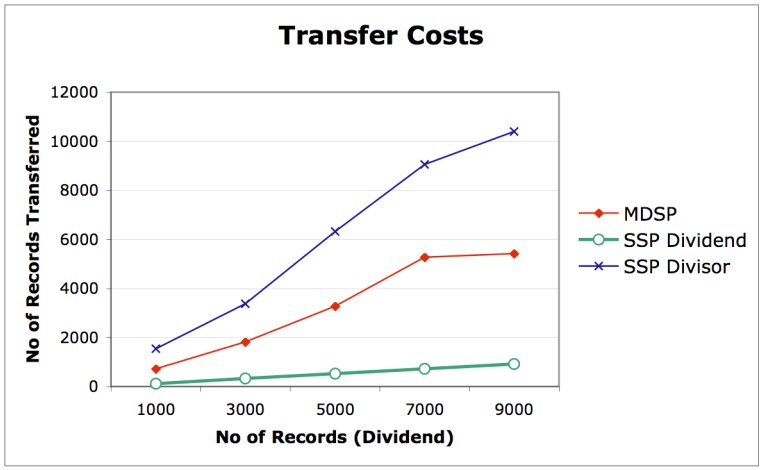


Figure 7.11(a). Impact of dividend size on transfer costs

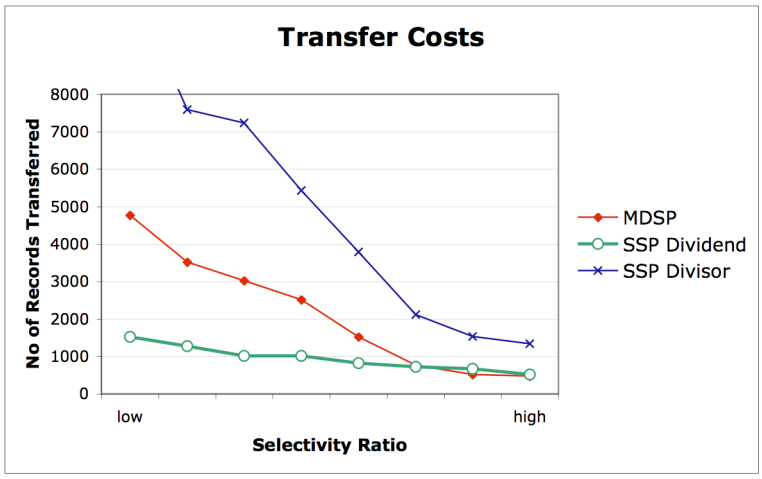


Figure 7.11(b). Impact of selectivity ratio on transfer costs

Figures 7.11(a)-(b) show the performance in terms of the data transfer costs. Figure 7.11(a) shows that SSP dividend gives the most efficient performance. In SSP dividend, the divisor table which is relatively smaller than the dividend, is sent to the dividend table to be processed. Because the divisor table is small, the transfer cost is then low. At the other extreme, SSP divisor is the most expensive. This confirms our

initial observation made in Chapter 3 about mobile join query processing: that is, when using the SSP model, we need to choose the smaller table to be transferred to the other server.

The MDSP technique in Figure 7.11(a) shows a medium-level performance. In MDSP, the transfer cost will also be dominated by the dividend table, similar to the SSP divisor. The only difference is the final query results; when using the SSP version, the results have to be transferred from the server to the mobile, and this incurs additional transfer cost.

Figure 7.11(b) shows the impact of selectivity ratio on the transfer cost. Again, SSP dividend gives the best performance, even when the selectivity ratio is quite low. A low selectivity means that the selectivity pruning is low, implying that the results are large. The MDSP and SSP dividends are both quite good when the pruning is high. When the pruning is high, using MDSP, some of the dividend records are not downloaded because they are pruned away. Whereas with the SSP dividend, the low transfer cost is a consequence of the small number of query results. The SSP divisor is very expensive, again due to the data transfer from the dividend to the divisor server.

Figures 7.12(a)-(b) studied the processing costs based on the two variables: dividend size and selectivity pruning. Figure 7.12(a) shows that the SSP divisor's processing cost is the lowest. This is in contrast to the transfer cost, where the SSP divisor is the most expensive. The low processing cost of SSP divisor is because the some of the records from the dividend have already pruned by the group counting comparison done on the mobile side. Due to this, the division processing in the divisor is discounted. For the SSP dividend, there is no group elimination, because all dividend records need to be accessed from the local disk in the dividend server anyway. So the processing cost will not be as low as that for the SSP divisor. For MDSP, the processing cost is the most expensive, simply because of a limitation in resources in the mobile device. In the experimentations, the divisor table was very small, and therefore we did not encounter any main memory overflow. Nevertheless, because of the difference in processing speed between server and mobile, the poor performance of MDSP reflects this fact.

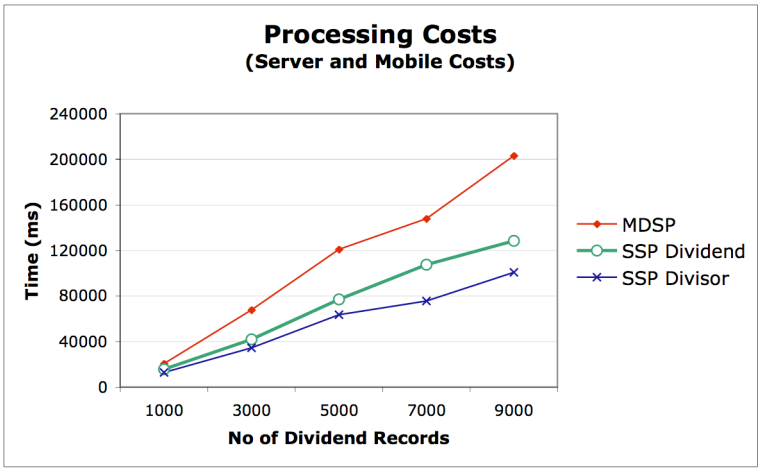


Figure 7.12(a). Impact of dividend size on processing costs

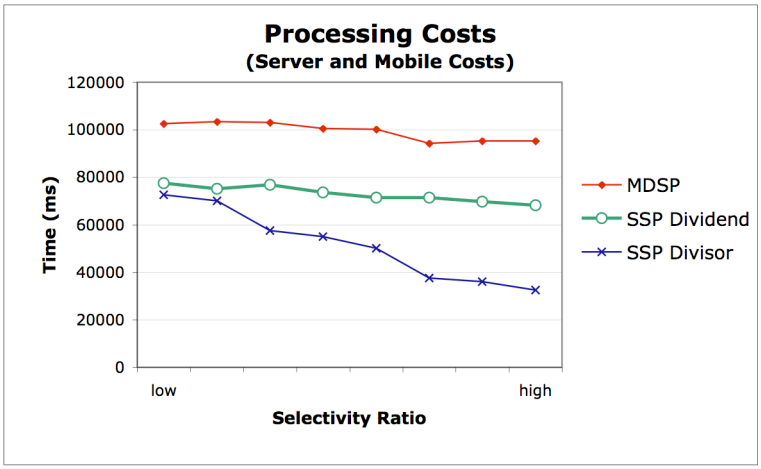


Figure 7.12(b). Impact of selectivity ratio on processing costs

Figure 7.12(b) shows the processing costs due to the selectivity pruning. Again, the SSP divisor shows its superiority; whereas, the SSP dividend does not improve so much even when the pruning is high. This is because, when the division operation is done in the dividend table, the pruning of the dividend does not affect the entire processing. For MDSP, it is very similar due to the more expensive mobile processing.

Overall, there is a contradictory result especially for the SSP divisor, which is high in transfer cost but quite efficient in the processing cost. However, because the

transfer unit cost is much more expensive than the processing unit cost, SSP divisor overall performs poorly. We need to rely on the SSP dividend in processing mobile relational division query. For the MDSP approach, the performance is not too bad. MDSP is a good option, if the mobile environment does not permit any query to be processed by the server (if the server does a data broadcast only). MDSP performance is acceptable especially if the divisor table is quite small, since this is not likely to incur any main memory overflow in the mobile processing.

(*b*) Performance Results for Mobile Multiple Group Division Queries

Mobile multiple group division query processing is one of the main contributions of this thesis. We have proposed two processing techniques: one is based on sort-merge, and the other on count aggregate. In the experimentations, we have examined the performance of these two techniques. Additionally, we have studied the effect of pruning in the aggregate-based of multiple group division, since pruning is one of the optimisation techniques we developed for multiple group division query processing.

Figures 7.13(a)-(b) demonstrate the effect of pruning in the aggregate-based multiple group division query. We compare this with the other two methods (sort-merge and the non-pruning aggregate-based).

Figure 7.13(a) shows the impact of the pruning level on the transfer costs. Since the sort-merge and the non-pruning aggregate-based are not affected by the pruning level, there is only one value in the performance graph. Figure 7.13(a) shows that as the pruning level increases (becomes high), the aggregate-based with pruning performance drops significantly. In contrast, the sort-merge and the aggregate without pruning are not affected by the pruning level as they do not possess the pruning features. The pruning in the aggregate-based technique works by first requesting from the servers the group summary, which is then analyzed and compared in the mobile device. Only those groups which will be likely to contribute to the final query results will then be requested from the servers. As a result, if most of the groups have already been pruned away at this stage, fewer records need to be downloaded, resulting in a low data transfer cost, as happened with the aggregate-based with pruning as shown

in Figure 7.13(a). On the other hand, the other two methods do not include any group comparisons to prune away non-qualified groups.

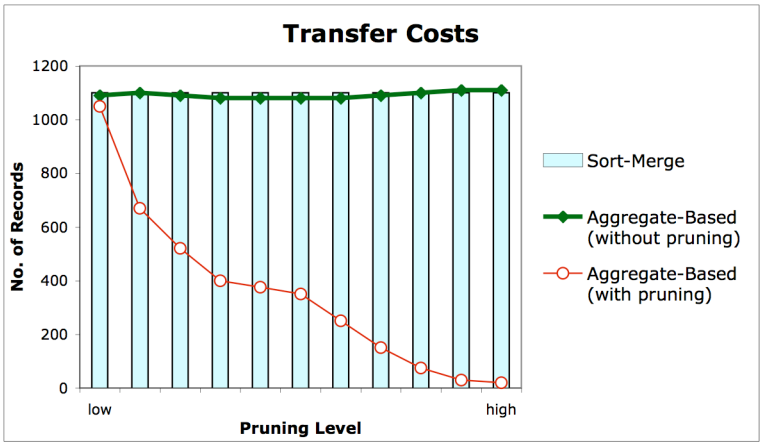


Figure 7.13(a). Impact of pruning level on transfer costs

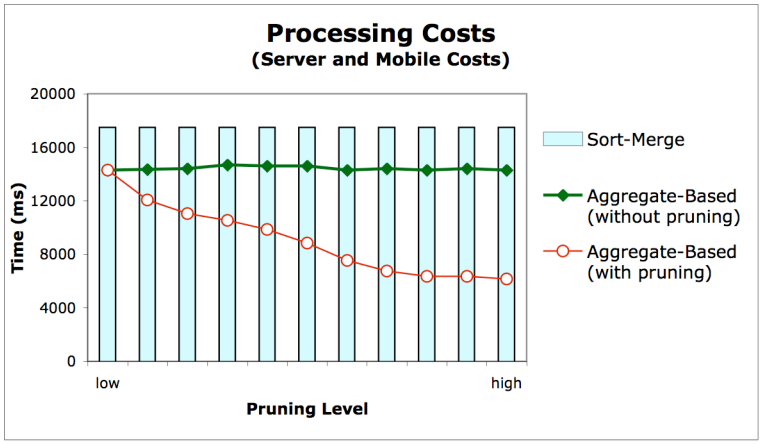


Figure 7.13(b). Impact of pruning level on processing costs

Figure 7.13(b) shows the impact of pruning on the processing costs covering server and mobile processing costs. Note that the trend is quite similar to transfer costs, only with a number of differences. One is that the drop in the processing cost of the aggregate-based with pruning technique is not as sharp as that of the transfer cost. This is due to the majority of the server cost having been absorbed by the retrieval and the grouping cost in the beginning. The other difference is that the aggregate-based without pruning performs slightly better than the sort-merge technique. This is

because the sorting unit cost is slightly higher than the aggregate unit cost, resulting in only a slight improvement. Nevertheless, the aggregate-based technique should employ the pruning method in order to take full advantage of the aggregate-based.

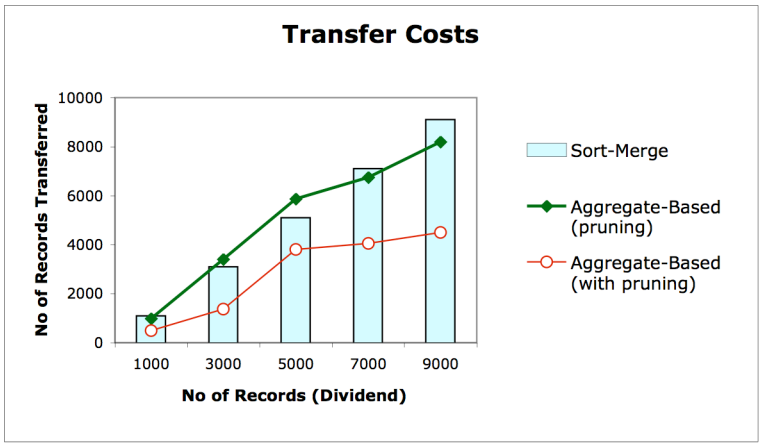


Figure 7.14(a). Impact of size of dividend on transfer costs

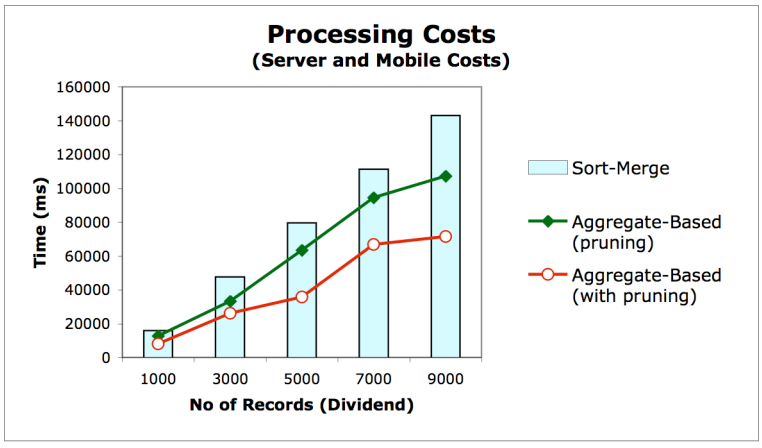


Figure 7.14(b). Impact of size of dividend on processing costs

Figures 7.14(a)-(b) show the impact of the size of dividend table on the transfer and processing costs. We can certainly expect that the cost will increase as the number of records in the dividend table grows. However, we would like to compare the three methods.

Figure 7.14(a) shows that the transfer costs of the sort-merge and aggregate without pruning are quite comparable. We run a simulation to generate the synthetic data randomly, and the results for these two techniques are quite comparable, although Figure 7.14(a) shows that the sort-merge incurs a higher transfer cost. However, the most important observation is that the aggregate-based with pruning performs the best. It is roughly around 60% of the other two techniques. The saving demonstrates the benefits of pruning at the initial grouping stage.

Figure 7.14(b) shows the processing cost. It is clear that the sort-merge processing cost is the worst of the three, and the aggregate-based with pruning saves between 20%-30% of the processing costs compared with the other two techniques.

Both graphs demonstrate that the performance of the aggregate-based with pruning is quite promising, although the effectiveness of the technique depends on how much pruning can be done. Nevertheless, even when the amount of pruning is small, some performance benefits are still produced, thereby indicating that it is beneficial to add a pruning stage to the aggregate-based method. On the contrary, the sort-merge performance is not attractive.

7.7. Summary of Evaluation

A number of aspects emerged from the experimentations.

- Since *data transfer costs* dominate the overall processing cost, it is desirable to reduce the communication costs by transferring data from the server to the mobile device as little as possible. In the experimentations, MDSP3, where the initial data download is restricted to the primary keys only, delivers the best performance. For SSP, SSP2 delivers a better performance than SSP1, for the same reason. Furthermore, since multiple servers are involved, the server that has the smaller table size is used to download the data.
- For the *processing costs*, which include server and mobile processing costs, the aim was to have the server perform the heavy processing, with the mobile device doing the lighter processing. This is achievable by employing a sort-

merge processing in MDSP, whereby the sorting is done by the server and the merging is carried out by the mobile device. This tends to give a better performance than the traditional hash-join method. It is important to study processing costs separately, in order to understand the processing behaviour of the server and the mobile device, due to the processing and storage constraints imposed by the mobile device. Taking these into account, MDSP3 delivers the best performance, as the join operands for the operation are small.

- For the block-based processing in mobile join processing, because all data needs to be processed, the efficiency is not achieved due to the block-based processing itself, but due to the *level of pruning* done prior to the processing. As a result, the aggregate-based join processing (ABJP) performs slightly better than the original block method (BBP). This supports the abovementioned points on data transfer costs, whereby the communication costs should be decreased by reducing the number of records transferred from the server to the mobile (and vice versa), and this is achieved through the pruning method in ABJP. The use of the pruning method is also advantageous in the MDSP because, if any side from the join is pruned, the joining process in the mobile will be very light. This is not the case with SSP because the pruning is only half-sided.
- For the Top-k join processing, the *merge with hashing* method outperforms the nested-loop method. The combined method of merging and hashing offers a much better performance because of the linear complexity of the algorithm. Furthermore, when using the merge with hashing method, the *memory utilization* is rather small, making this highly suitable for mobile devices with small storage space.
- For the Groupby-Join whereby the join attribute is the same as the groupby attribute (known as *Groupby-Before-Join*), the performance relies on the amount of pruning that is done before the join. This also supports the above claim that the overall processing costs can be decreased if the operands to the processing become small. Pruning in Groupby-Before-Join is achieved through the groupby processing done before the join processing. The pruning

level is determined by the number of groups formed during the groupby process, the smaller the number of group the higher the pruning level, and consequently, performance gains.

- For the Groupby-Join where the join attribute is different from the groupby attribute, the *Double Grouping* method proves that it is worthwhile to perform the group operation before the join operation, which was thought to be impossible by many of the existing methods. Because the groupby operation is done first, consequently, there is a significant reduction in the communication costs, again due to the pruning as in the Groupby-Before-Join. On the other hand, the *Single Grouping* method does not deliver much performance gain, because the grouping done with this method does not reduce the size of the operation, as the grouping is merely for block-based processing purposes, and not for aggregating/summarizing the input data.
- For the *Relational Division* operation, since the processing itself is a heavy task, the SSP method is the best option. It is also important to choose the correct server to download, whether it be the divisor server or the dividend server. Because the transfer cost is a dominant factor, SSP dividend method delivers a better performance due to the lowest communication costs.
- For the *Multiple Group Division* operation, the *pruning* method used in the aggregate-based is very attractive. Even when the pruning level is low, the performance is still very good, in terms of transfer costs and processing costs.

There are two important lessons drawn from the experimentation results:

1. The *communication costs* should be reduced by *pruning* input data prior to the main processing. In the mobile join operation, the aggregate-based join processing performs some pruning by removing any non-candidates for the join. In the Groupby-Join, the pruning is done through the groupby operation done before the join, and in the division, the pruning is achieved especially in our proposed aggregate-based multiple group division.

2. The *processing and storage costs* should be reduced by keeping and processing as small an amount of data as possible in the mobile device. In Top-k join, this is achieved by the utilizing a block-based combined with a hashing technique, where only the active block is stored in the memory of the mobile device with a small overhead of the unprocessed records. The processing costs can also be reduced by outsourcing the heavy processing to the server, like the sort-merge MDSP.

7.8 Conclusions

This chapter basically has provided a complete guide to the cost models that are used to simulate a wide range of proposed techniques in providing the results to evaluate the performance of each of the proposed techniques. The main components that are of interest in terms of evaluating the performance are the communication cost and the processing cost involving the cost of processing done by both the server and the mobile. Each proposed technique has been proven by the simulated graphs which demonstrate the level to which one technique can outperform another in certain circumstances which may be determined for example, by whether or not the records size between the servers is similar. Thus, different conditions will give a different performance result. Therefore, this chapter provides proof of the level of performance of each proposed technique when applied under different conditions.

Chapter 8

Conclusions

8.1 Introduction

This thesis investigated mobile query processing incorporating multiple non-collaborative servers. The main aim of this research was to study performance improvement as well as extending the use of traditional query processing techniques to be adaptable in the mobile environment. This thesis also focuses on more advanced types of queries that are useful in addressing some of the well-known problems of the mobile environment. Attention is focused on two major areas of the query processing technique: *join query processing* and *aggregate query processing*. The two query types are the most common ones, especially since multiple data sources are involved. As this thesis focuses on multiple independent data sources in a mobile environment, these two queries are very important, and hence it is appropriate and necessary to investigate them thoroughly. In addition, the performance evaluation and analysis of the results for each of the proposed algorithms have been carried out.

8.2 Summary of the Research Results

The main research result of this thesis demonstrates how mobile query processing can be executed when multiple independent servers are involved. This is achieved by

formulating various types of query processing algorithms for a number of different possible mobile queries that may arise, particularly join queries which include basic join, and Top-k join. Other types of queries such as aggregate query, which include GroupBy join and division queries have also been investigated.

Join is an important problem to solve, more importantly in the mobile environment, where the communication cost is expensive due to low bandwidth. Hence, minimization of the communication cost is always the primary aim. This can partially be answered through Top-k join queries, and that is why Top-k join queries have also been investigated in this thesis.

In particular, the small display limitation in the mobile devices is addressed in the second part of the thesis focusing on mobile aggregate query processing. Because the focus is on multiple data sources, the aggregate queries are discussed in conjunction with join queries, and hence we focus on Groupby-Join queries. The use of grouping and aggregate functions, like counting, is critical in processing division queries. As a result, division queries are also the focus in this thesis.

The research presented in this thesis has addressed the limitations of the mobile environment including the mobile device itself as well as solving the outstanding problems for each of the existing queries techniques to be adaptable in the mobile environment. The achievements of this research are summarized as follows.

- **Mobile Join Query Processing**

  The main motivation in proposing mobile join query processing is to enable data from multiple independent servers in the mobile environment to be processed and joined so that a new form of information can be obtained. The proposed mobile join algorithms take the base concept of traditional join, sort merge and hash approaches to be used in the mobile environment. The algorithms are built to be used either as locally process in the mobile device which is known as mobile device side processing (*MDSP*), or in conjunction of server side processing (*SSP*). In addition, the block-based approach was being investigated with the aim of

helping to minimize the communication costs, as well as taking into account the small display screen of a mobile devices and its limited memory capacity.

The main contributions of mobile join query processing are twofold. One is that we have demonstrated how mobile join queries are processed through the mobile side processing as well as the server side processing. The second contribution, which is the main contribution in this context, is that through our analysis of the performance evaluation, we have been able to conclude that the sort-merge join method may outperform the hash-join method, especially when the mobile memory is very small. With the sort-merge join method, the sorting is outsourced to the server which has a more powerful resource, leaving the merging operation to be done in the mobile device. Whereas, the hash-join performs poorly when there is a memory overflow in the mobile device due to a small memory capacity.

We can summarize that in an environment where there is a server and a client, if the operation can divided into two, it is desirable to allocate a heavy process to the server and let the client perform the lightweight processes. In our case, this can be achieved by using the sort-merge join approach. In the hash-join approach, the server does a lightweight job which is record retrieval, and the client does a heavyweight job which is the hashing and probing. This is an undesirable job allocation. On the other hand, with sort-merge join, the heavyweight task of sorting is done by the server, whereas the lightweight job of merging is done by the client, which is a desirable job division.

- **Mobile Top-k Join Query Processing**

Research on Top-k join queries has been an active research area in a non-mobile environment. This thesis looked at Top-k join queries in the mobile environment with its capability to address specific mobile issues such as limited screen display, limited battery life or low bandwidth. Top-k join query result is where the queries return only specific matches that qualify as important information. By having multiple independent servers, the purpose is to obtain the Top-k information based

on the different multiple lists obtained from the servers regardless of whether the original obtained data are sorted or non-sorted. It is used to circumvent the problems of limited memory, short battery life span and low bandwidth due to its ability to display only the few top results to the mobile users. This is because, with the goal of delivering limited results, it reduces the time of processing as well as utilizing less memory and screen space to display it. Thus, mobile devices do not need a large memory capacity in order to keep a whole long list of results to be displayed, and users do not have to wait a long time (which drains the battery) since only the top few results need to be displayed.

The main contributions of our Top-k join query investigation are also twofold: one is that we have investigated how a Top-k join is performed in a mobile environment. In this case, we have introduced the use of nested-loop and sort-merge approaches. Although the nested-loop is simple, it is necessary in order to illustrate the completeness of the proposed methods. The second contribution is that we have formulated rules whereby Top-k join processing will not stop until a certain condition has been reached. And this condition is not derived by the result, but by the records unprocessed. We have thoroughly explained that the process should not stop when there are unmatched records in a higher rank. The process is stops only when all higher ranked records have been processed and the Top-k join results have been produced. This is a necessary condition in order to ensure the accuracy of the Top-k join results.

- **Mobile GroupBy-Join Query Processing**

The next type of query that we contributed in this thesis is another extension of the mobile join query processing where a collection of proposed algorithms using aggregate functions are being presented to deal with groupby-join queries. The idea is to incorporate aggregate query processing in the mobile environment using the existing aggregate functions and to propose algorithms that will maximize the efficiency of the processing and at the same time respond to the user queries. Using the *MDSP* and *SSP* foundation, these techniques also took into

consideration whether the types of queries should have the join process first or the group-by process first. This is determined by whether the join attribute and group by attribute are similar or different. Nevertheless, these queries also incorporate the block-based approach that is believed to minimize communication costs and address the small display and limited memory issues.

There are two main contributions in this context. One is that we have proposed two types of processing methods, the first of which addresses the Groupby-Join queries where the join attribute is the same as the groupby attribute. In this case, the groupby operation is carried out first, then followed by the join. Hence the name, Groupby-Before-Join.

Our second and main contribution deals with the Groupby-Join where the join attribute is different from the groupby attribute. Traditionally, this is called Groupby-After-Join because the join is done first and is then followed by the groupby. We propose that it is possible to do the grouping first before the join. Hence, our results are much better than the traditional ones. In order to achieve this, we have proposed two methods, namely, Double Grouping and Single Grouping. Although the single grouping method does not significantly improve performance compared with the traditional method, our Double Grouping method outperforms the traditional methods.

- **Mobile Division Query Processing**

Traditionally, this universal quantification has been lacking in research, especially in cases where large databases are involved. Thus, the existing techniques need to be studied so that they can be enhanced or modified for use in the mobile environment. Basically, it is a query request to find an attribute in one relation that satisfies all of a given list of criteria from another relation. We examine ways to overcome the shortcomings of the traditional division techniques and apply them to the mobile environment incorporating multiple non-collaborative servers. We also extended the traditional division query processing by considering multiple

group division query processing, whereby the divisor is not a single group from a relation, but multiple groups from a single relation. This type of query in particular is common in multiple independent servers query processing in a mobile environment.

Our contributions are also twofold in this regard. Firstly, we focused on the relational division based on sort-merge and aggregate functions. Our second contribution is the main one, which is multiple group division. None of the existing work has considered multiple group division. In addition to this major contribution, we enhance the performance by introducing some pruning techniques so that unnecessary records will be pruned away, and with consequent gains in performance.

8.3 Future Research

Many avenues exist for further research, both theoretical and practical, as indicated in the following paragraphs.

***Mobile User Interface Complexities*:** Throughout the whole thesis, we have not looked into the limitation of mobile screen display or issues of mobile user interface. However, it will be beneficial to have a friendly user interface, and additionally, with a small screen display of a mobile device, it is often useful to take this into account so that the query results to be displayed on the small display screen are easily readable and navigable. Also, another worthwhile area to be investigated relates to having a good interactive command that transfers some control to the mobile users enabling them to continue or abort certain queries that are currently being processed.

***Mobile Approximate Query Processing*:** Approximate queries are both relevant and important in a mobile environment because several limitations of a mobile environment may prevent the returning of particular query results accurately.

In addition, there are times when certain data sets that are supposed to qualify as a match between servers are actually being processed as a disqualified match due to a slight inaccuracy. This inaccuracy or wrongly disqualified data can cause some unavoidable mobile issues, such as noise in the data sets, differential occurrence within the multiple servers such as the attribute value standard, data structure, and data entry. Therefore, a future research aim may be the formulation of algorithms that take into account the above issues.

***Mobile Location Dependent Queries*:** The explosive growth of location-dependent applications has shown that research in this domain is beneficial, especially in regards to mobile users who are not stationary and their present location is often of interest. Whenever a user moves from one location to another location, the objects being queried can turn out to be different according to their geographical coordinates. Hence, location-dependent applications play an important role. It is important to show to the mobile user, who is frequently moving from one location to another, that the queries he or she sends depend on the location that he or she is querying from. This also includes further researching into designing algorithms that are able to process queries in an efficient way that involves data that are obtained from multiple, different locations.

***Mobile Query with Cached Data*:** The use of effective caching management in mobile databases is believed to improve the efficiency of the query processing. Due to the limitations of a typical mobile environment which includes limited bandwidth and instability of the wireless network which cause frequent disconnection, caching of the frequently accessed data in a client's local storage becomes significant in improving the performance and data availability of data access queries. This is made available by caching the frequently accessed data items in the local mobile device storage as well as when frequent disconnections occur. The query can still be partially processed from caches and at least some of the results from the previous queries can be returned to the users. This is because the mobile device is able to keep the existing data and if the user needs the same exact data, the downloading can be minimized if the mobile device recognizes that the data has been previously loaded into the device.

Thus, it is important to have cache management because often a user may download similar data repeatedly from the same source.

Bibliography

Acharya, D., Vijay Kumar, Gi-Chul Yang: DAYS mobile: a location based data broadcast service for mobile users. *The ACM Symposium on Applied Computing* (*ACM SAC*) 2007: 901-905

Akbarinia, R., Esther Pacitti, Patrick Valduriez: Reducing network traffic in unstructured P2P systems using Top-k queries. *Distributed and Parallel Databases* **19**(2-3): 67-86 (2006)

Al-Mogren, A.S., Margaret H. Dunham: BUC, a Simple yet Efficient Concurrency Control technique for Mobile Data Broadcast Environment. *DEXA Workshop* 2001: 564-569

Angiulli, F., Clara Pizzuti: An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data Knowl. Eng.* **53**(3): 263-281 (2005).

Angiulli, F., Clara Pizzuti: Top-k Closest Pairs Join Query: An Approximate Algorithm for Large High Dimensional Data. *International Database Engineering and Applications Symposium (IDEAS)* 2004: 103-110.

Au, M-W., Edward Chan, Kam-yiu Lam: Concurrency Control for Mobile Systems with Data Broadcast. *Journal of Interconnection Networks* **2**(3): 253-267 (2001)

Balke, W-T., Wolfgang Nejdl, Wolf Siberski, Uwe Thaden: Progressive Distributed Top k Retrieval in Peer-to-Peer Networks. *The IEEE International Conference on Data Engineering (ICDE)* 2005: 174-185

Bose, I., Wang Ping, Mok Wai Shan, Wong Ka Shing, Yip Yee Shing, Chan Lit Tin, Shiu Ka Wai: Databases for Mobile Applications. *Encyclopedia of Database Technologies and Applications* 2005: 162-169

Cao, G.: A Scalable Low-Latency Cache Invalidation Strategy for Mobile. *IEEE Trans. Knowl. Data Eng.* **15**(5): 1251-1265 (2003)

Cao, G.: On Improving the Performance of Cache Invalidation in Mobile Environments. *MONET* **7**(4): 291-303 (2002a)

Cao, G.: Proactive Power-Aware Cache Management for Mobile Computing Systems. *IEEE Trans. Computers* **51**(6): 608-621 (2002b)

Cao, H., Shan Wang, Lingwei Li: Location dependent query in a mobile environment. *Inf. Sci.* **154**(1-2): 71-83 (2003)

Cao, P., Zhe Wang: Efficient top-K query calculation in distributed networks. *PODC* 2004: 206-215

Celik, A., JoAnne Holliday, Zachary Hurst: Data Dissemination to a Large Mobile Network: Simulation of Broadcast Clouds. *Mobile Data Management* 2006: 37

Chakrabarti, K., Michael Ortega-Binderberger, Sharad Mehrotra, Kriengkrai Porkaew: Evaluating Refined Queries in Top-k Retrieval Systems. *IEEE Trans. Knowl. Data Eng.* **16**(2): 256-270 (2004)

Chan, B.Y.L., Antonio Si, Hong Va Leong: Cache Management for Mobile Databases: Design and Evaluation. *The IEEE International Conference on Data Engineering (ICDE)* 1998: 54-63

Chan, D., John F. Roddick: Context-Sensitive Mobile Database Summarisation. *Australian Computer Science Conference (ACSC)* 2003: 139-149

Chang, Y-I., Che-Nan Yang: A complementary approach to data broadcasting in mobile information systems. *Data Knowl. Eng.* **40**(2): 181-194 (2002)

Chang, Y-I., Shih-Ying Chiu: A Hybrid Approach to Query Sets Broadcasting Scheduling for Multiple Channels in Mobile Information Systems. *J. Inf. Sci. Eng.* **18**(5): 641-666 (2002)

Chaudhuri, S., Luis Gravano, Amélie Marian: Optimizing Top-k Selection Queries over Multimedia Repositories. *IEEE Trans. Knowl. Data Eng.* **16**(8): 992-1009 (2004)

Chaudhuri, S., Luis Gravano: Evaluating Top-k Selection Queries. *The International Conference on Very Large Data Bases (VLDB)* 1999: 397-410

Chen, C-M., Yibei Ling: A Sampling-Based Estimator for Top-k Query. *The IEEE International Conference on Data Engineering (ICDE)* 2002: 617-627

Chen, M-S., Kun-Lung Wu, Philip S. Yu: Optimizing Index Allocation for Sequential Data Broadcasting in Wireless Mobile Computing. *IEEE Trans. Knowl. Data Eng.* **15**(1): 161-173 (2003)

Chow, C-Y., Hong Va Leong, Alvin T. S. Chan: Cache Signatures for Peer-to-Peer Cooperative Caching in Mobile Environments. *The International Conference on Advanced Information Networking and Applications AINA* (volume 1) 2004: 96-101

Chow, C-Y., Hong Va Leong, Alvin T. S. Chan: Group-Based Cooperative Cache Management for Mobile Clients in a Mobile Environment. *The International Conference on Parallel Processing (ICPP)* 2004: 83-90

Chrysanthis, P. K., Evaggelia Pitoura: Mobile and Wireless Database Access for Pervasive Computing. *The IEEE International Conference on Data Engineering (ICDE)* 2000: 694-695

Chuang, P-J., Ching-Yueh Hsu: An Efficient Cache Invalidation Strategy in Mobile Environments. *The International Conference on Advanced Information Networking and Applications AINA* (volume 2) 2004: 260-263

Date, C.J.: *An Introduction to Database Systems*, 6th Edition. Addison-Wesley 1995

Datta, A., Debra E. VanderMeer, Aslihan Celik, Vijay Kumar: Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users. *ACM Trans. Database Syst.* **24**(1): 1-79 (1999)

Deng, B., Yan Jia, Shuqiang Yang: Efficient Non-Blocking Top-k Query Processing in Distributed Networks. *DASFAA* 2006: 880-889.

Dunham, M.H., Vijay Kumar: Location Dependent Data and its Management in Mobile Databases. *DEXA Workshop* 1998: 414-419

Elmagarmid, A.K., in Jing, Abdelsalam Helal, Choonhwa Lee: Scalable Cache Invalidation Algorithms for Mobile Data Access. *IEEE Trans. Knowl. Data Eng.* **15**(6): 1498-1511 (2003)

Elmasri, R., Shamkant B. Navathe: *Fundamentals of Database Systems*, 5th Edition. Addison Wesley 2007

Fagin, R., Ravi Kumar, D. Sivakumar: Comparing Top k Lists. *SIAM J. Discrete Math.* **17**(1): 134-160 (2003)

Fratarcangeli, C.,: Technique for Universal Quantification in SQL. *SIGMOD Record* **20**(3): 16-24 (1991)

Graefe, G., Richard L. Cole: Fast Algorithms for Universal Quantification in Large Databases. *ACM Trans. Database Syst.* **20**(2): 187-236 (1995)

Han, K.H., Jai-Hoon Kim, Young-Bae Ko, Won-Sik Yoon: An Energy Efficient Broadcasting for Mobile Devices Using a Cache Scheme. *International Conference on Computational Science* 2004: 598-601

He, Y., Yanfeng Shu, Shan Wang, Xiaoyong Du: Efficient Top-k Query Processing in P2P Network. *DEXA* 2004: 381-390.

Heuer, A., Astrid Lubinski: Database Access in Mobile Environments. *DEXA* 1996: 544-553

Hu, Q., Dik Lun Lee: Adaptive Cache Invalidation Methods in Mobile Environments. *The International Conference on High Performance Distributed Computing (HPDC)* 1997: 264-273

Hu, Q., Dik Lun Lee: Cache algorithms based on adaptive invalidation reports for mobile environments. *Cluster Computing* **1**(1): 39-50 (1998)

Huang, J-L., Ming-Syan Chen, Wen-Chih Peng: Broadcasting Dependent Data for Ordered Queries without Replication in a Multi-Channel Mobile Environment.

The IEEE International Conference on Data Engineering (ICDE) 2003: 692-694

Huang, J-L., Ming-Syan Chen: Dependent Data Broadcasting for Unordered Queries in a Multiple Channel Mobile Environment. *IEEE Trans. Knowl. Data Eng.* **16**(9): 1143-1156 (2004)

Huang, J-L., Wen-Chih Peng, Ming-Syan Chen: SOM: Dynamic Push-Pull Channel Allocation Framework for Mobile Data Broadcasting. *IEEE Trans. Mob. Comput.* **5**(8): 974-990 (2006)

Hung, J-J., Yungho Leu: An Energy Efficient Data Reaccess Scheme for Data Broadcast in Mobile Computing Environments. *The International Conference on Parallel Processing ICPP* Workshops 2003: 5-12

Hung, J-J., Yungho Leu: Efficient Index Caching Schemes for Data Broadcasting in Mobile Computing Environments. *DEXA Workshops* 2003: 139-143

Hwang, S-W., Kevin Chen-Chuan Chang: Optimizing Access Cost for Top-k Queries over Web Sources: A Unified Cost-based Approach. *The IEEE International Conference on Data Engineering (ICDE)* 2005: 188-189

Ilyas, I.F., Walid G. Aref, Ahmed K. Elmagarmid: Supporting Top-k Join Queries in Relational Databases. *The International Conference on Very Large Data Bases (VLDB)* 2003: 754-765.

Ilyas, I.F., Walid G. Aref, Ahmed K. Elmagarmid: Supporting top-k join queries in relational databases. *VLDB J.* **13**(3): 207-221 (2004).

Kahol, A., Ramandeep Singh Khurana, Sandeep K. S. Gupta, Pradip K. Srimani: An Efficient Cache Maintenance Scheme for Mobile Environment. *The International Conference on Distributed Computing Systems (ICDCS)* 2000: 530-537

Kim, H-S., Hwan-Seung Yong: Personalized cache management for mobile computing environments. *Inf. Process. Lett.* **87**(4): 221-228 (2003)

Kimelfeld, B., Yehoshua Sagiv: Finding and approximating top-k answers in keyword proximity search. *Principle of Database Systems (PODS)* 2006: 173-182

Kottkamp, H-E., Olaf Zukunft: Location-aware query processing in mobile database systems. *The ACM Symposium on Applied Computing* (*ACM SAC*) 1998: 416-423

Kumar, M., Sajal K. Das: Flexible Integrated Cache for Efficient Information Access in Mobile Computing Environments. *Infrastructure for Mobile and Wireless Systems* 2001: 69-77

Lai, K.Y., Zahir Tari, Peter Bertók: Cost Efficient Broadcast Based Cache Invalidation for Mobile Environments. *The ACM Symposium on Applied Computing* (*ACM SAC*) 2003: 871-877

Lam, K-Y., Edward Chan, Hei-Wing Leung, Mei-Wai Au: Concurrency control strategies for ordered data broadcast in mobile computing systems. *Inf. Syst.* **29**(3): 207-234 (2004)

Lam, K-Y., Edward Chan, Joe Chun-Hung Yuen: Approaches for broadcasting temporal data in mobile computing systems. *Journal of Systems and Software* **51**(3): 175-189 (2000)

Lam, K-Y., Edward Chan, Joe Chun-Hung Yuen: Broadcast Strategies to Maintain Cached Data for Mobile Computing System. *The International Conference on Entity-Relationship (ER) Workshops* 1998: 193-204

Lee, C., Chao-Chun Chen: "Cache and Carry" for Location Management in Mobile Information Systems. *Distributed and Parallel Databases* **16**(2): 165-192 (2004)

Lee, C-C., Yungho Leu: Efficient data broadcast schemes for mobile computing environments with data missing. *Inf. Sci.* **172**(3-4): 335-359 (2005)

Lee, C-H., M. Chen: Using Remote Joins for the Processing of Distributed Mobile Queries. *DASFAA* 2001: 226-233

Lee, C-H., Ming-Syan Chen: Processing Distributed Mobile Queries with Interleaved Remote Mobile Joins. *IEEE Trans. Computers* **51**(10): 1182-1195 (2002)

Lee, C-I., Cheng-Jung Tsai: An efficient approach to extracting and ranking the top K interesting target ranks from Web search engines. *Informatica* (Slovenia) **25**(3): (2001)

Lee, G., Shou-Chih Lo: Broadcast Data Allocation for Efficient Access of Multiple Data Items in Mobile Environments. *MONET* **8**(4): 365-375 (2003)

Lee, K.C.K., Hong Va Leong, Antonio Si: Adaptive semantic data broadcast in a mobile environment. *The ACM Symposium on Applied Computing* (*ACM SAC*) 2001: 393-400

Lee, K.C.K., Hong Va Leong, Antonio Si: Semantic Data Broadcast for a Mobile Environment Based on Dynamic and Adaptive Chunking. *IEEE Trans. Computers* **51**(10): 1253-1268 (2002)

Lee, Y.K., Woong-Kee Loh, Yang-Sae Moon, Kyu-Young Whang, Il-Yeol Song: An Efficient Algorithm for Computing Range-Groupby Queries. *DASFAA* 2006: 483-497.

Li, C., Kevin Chen-Chuan Chang, Ihab F. Ilyas, Sumin Song: RankSQL: Query Algebra and Optimization for Relational Top-k Queries. *SIGMOD Conference* 2005: 131-142

Liu, J., Liang Feng, Chao He: Semantic link based top-K join queries in P2P networks. *The International Conference on World Wide Web (WWW)* 2006: 1005-1006.

Liu, J., Liang Feng, Yunpeng Xing: A pruning-based approach for supporting Top-K join queries. *The International Conference on World Wide Web (WWW)* 2006: 891-892.

Lo, E., Nikos Mamoulis, David Wai-Lok Cheung, Wai-Shing Ho, Panos Kalnis: Processing Ad-Hoc Joins on Mobile Devices. *DEXA* 2004: 611-621

Loh, Z.X., Tok Wang Ling, Chuan-Heng Ang, Sin Yeung Lee: Adaptive Method for Range Top- k Queries in OLAP Data Cubes. *DEXA* 2002: 648-657.

Loh, Z.X., Tok Wang Ling, Chuan-Heng Ang, Sin Yeung Lee: Analysis of pre-computed partition top method for range top-k queries in OLAP data cubes. *CIKM* 2002: 60-67.

Luo, Z.W., Tok Wang Ling, Chuan-Heng Ang, Sin Yeung Lee, Bin Cui: Range Top/Bottom k Queries in OLAP Sparse Data Cubes. *DEXA* 2001: 678-687.

Malladi, R., Karen C. Davis: Applying Multiple Query Optimization in Mobile Databases. *Hawaii International Conference on System Sciences HICSS* 2003: 294

Mamoulis, N., Panos Kalnis, Spiridon Bakiras, Xiaochen Li: Optimization of Spatial Joins on Mobile Devices. *Advances in Spatial and Temporal Databases, 8th International Symposium* (*SSTD)* 2003: 233-251

Marian, A., Sihem Amer-Yahia, Nick Koudas, Divesh Srivastava: Adaptive Processing of Top-K Queries in XML. *The IEEE International Conference on Data Engineering (ICDE)* 2005: 162-173

Marsit, N., Abdelkader Hameurlain, Zoubir Mammeri, Franck Morvan: Query Processing in Mobile Environments: A Survey and Open Problems. *International Conference on Distributed Frameworks for Multimedia Applications (DFMA)* 2005: 150-157

Matsunam, H., Tsutomu Terada, Shojiro Nishio: A Query Processing Mechanism for Top-k Query in P2P Networks. *The IEEE International Conference on Data Engineering (ICDE) Workshops* 2005 (online proceedings: http://www.informatik.uni-trier.de/~ley/db/conf/icde/icdew2005.html#MatsunamTN05)

Mei, J., Richard B. Bunt: Adaptive File Cache Management for Mobile Computing. *Mobile Data Management* 2003: 369-373

Metwally, A., Divyakant Agrawal, Amr El Abbadi: Efficient Computation of Frequent and Top-k Elements in Data Streams. *The International Conference on Database Theory (ICDT)* 2005: 398-412

Michel, S., Peter Triantafillou, Gerhard Weikum: KLEE: A Framework for Distributed Top-k Query Algorithms. *The International Conference on Very Large Data Bases (VLDB)* 2005: 637-648

Negri, M., Giuseppe Pelagatti, Licia Sbattella: Semantics and Probelms of Universal Quantification in SQL. *Comput. J.* **32**(1): 90-91 (1989)

Nejdl, W., Wolf Siberski, Uwe Thaden, Wolf-Tilo Balke: Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks. *International Semantic Web Conference* 2004: 137-151

Omiecinski, E.: Parallel Relational Database Systems. *Modern Database Systems* 1995: 494-512

Ozakar, B., Franck Morvan, Abdelkader Hameurlain: Mobile join operators for restricted sources. *Mobile Information Systems* **1**(3): 167-184 (2005)

Özsu, M.T., Patrick Valduriez: Distributed and Parallel Database Systems. *The Computer Science and Engineering Handbook* 1997: 1093-1111

Peng, W-C., Jiun-Long Huang, Ming-Syan Chen: Dynamic Leveling: Adaptive Data Broadcasting in a Mobile Computing Environment. *MONET* **8**(4): 355-364 (2003)

Peng, W-C., Ming-Syan Chen: Design and Performance Studies of an Adaptive Cache Retrieval Scheme in a Mobile Computing Environment. *IEEE Trans. Mob. Comput.* **4**(1): 29-40 (2005)

Peng, W-C., Ming-Syan Chen: Efficient Channel Allocation Tree Generation for Data Broadcasting in a Mobile Computing Environment. *Wireless Networks* **9**(2): 117-129 (2003)

Peng, W-C., Ming-Syan Chen: Query Processing in a Mobile Computing Environment: Exploiting the Features of Asymmetry. *IEEE Trans. Knowl. Data Eng.* **17**(7): 982-996 (2005)

Pissinou, N., Kia Makki, William J. Campbell: On the design of a location and query management strategy for mobile and wireless environments. *Computer Communications* **22**(7): 651-666 (1999)

Qun, C., Andrew Lim, Yi Zhu: Index and Data Allocation in Mobile Broadcast. *DEXA* 2001: 785-794

Ramakrishnan, R. and Johannes Gehrke: *Database Management Systems*. 2nd Edition. McGraw-Hill 2000

Rantzau, R., Leonard D. Shapiro, Bernhard Mitschang, Quan Wang: Algorithms and applications for universal quantification in relational databases. *Inf. Syst.* **28**(1-2): 3-32 (2003)

Rantzau, R., Leonard D. Shapiro, Bernhard Mitschang, Quan Wang: Universal Quantification in Relational Databases: A Classification of Data and Algorithms. *The European Conference on Database Technology (EDBT)* 2002: 445-463

Sato, K., Soichi Hasegawa, Shigeaki Tagashira, Keizo Saisho, Akira Fukuda: Performance Modeling of Layered-Data Delivery for Mobile Users through Broadcast/On-Demand Hybrid Communication. *Mobile Data Management* 2001: 284-286

Shi, S., Jianzhong Li, Chaokun Wang: Using PR-Tree and HPIR to Manage Coherence of Semantic Cache for Location Dependent Data in Mobile Database. *The International Conference on Web Age Information Management (WAIM)* 2002: 72-79

Silberschatz, A., Henry F. Korth, S. Sudarshan: *Database System Concepts*, 5th Edition. McGraw-Hill Book Company 2006

Silberstein, A., Rebecca Braynard, Carla Schlatter Ellis, Kamesh Munagala, Jun Yang: A Sampling-Based Approach to Optimizing Top-k Queries in Sensor Networks. *The IEEE International Conference on Data Engineering (ICDE)* 2006: 68

Tan, K-L.: Organization of Invalidation Reports for Energy-Efficient Cache Invalidation in Mobile Environments. *MONET* **6**(3): 279-290 (2001)

Taniar, D., Clement Leung, Wenny Rahayu, Sushant Goel: *High Performance Database Processing and Grid Databases*, Wiley, 2008.

Taniar, D., Rebecca Boon-Noi Tan, Clement H. C. Leung, Kevin H. Liu: Performance analysis of "Groupby-After-Join" query processing in parallel database systems. *Inf. Sci.* **168**(1-4): 25-50 (2004)

Taniar, D., Wenny Rahayu, Hero Ekonomosa: Performance Evaluation of Parallel GroupBy-Before-Join Query Processing in High Performance Database Systems. *High Performance Computing and Networking (HPCN) Europe* 2001: 241-250.

Taniar, D., Wenny Rahayu: Parallel "GroupBy-Before-Join" Query Processing for High Performance Parallel/Distributed Database Systems. *The International Conference on Advanced Information Networking and Applications AINA* (1) 2006: 693-700.

Taniar, D., Wenny Rahayu: Parallel Processing of "GroupBy-Before-Join" Queries in Cluster Architecture. *The International Conference on Cluster Computing and the Grid (CCGRID)* 2001: 178-185.

Taniar, D., Yi Jiang, Kevin H. Liu, Clement H. C. Leung: Parallel Aggregate-Join Query Processing. *Informatica* (Slovenia) **26**(3): (2002).

Vlach, R.: Efficient Execution Strategies for Mobile Procedures Querying Distributed Databases. *Distributed Objects and Its Applications (DOA)* 2000: 99-108

Waluyo, A.B., Bala Srinivasan, David Taniar: Data Dissemination in Mobile Databases. *Encyclopedia of Information Science and Technology* (volume 2) 2005: 691-697

Waluyo, A.B., Gabriel Goh, David Taniar, Bala Srinivasan: On Building a Data Broadcasting System for Mobile Databases. *The International Conference on Advanced Information Networking and Applications AINA* 2005: 538-543

Waluyo, A.B., Bala Srinivasan, David Taniar: A Taxonomy of Broadcast Indexing Schemes for Multi Channel Data Dissemination in Mobile Database. *The International Conference on Advanced Information Networking and Applications AINA* (volume 1) 2004a: 213-218

Waluyo, A.B., Bala Srinivasan, David Taniar: Location Dependent Queries in Mobile Databases. *International Conference on Information and Knowledge Engineering (IKE)* 2004b: 362-370

Waluyo, A.B., Bala Srinivasan, David Taniar: Optimizing Query Access Time over Broadcast Channel in a Mobile Computing Environment. *The International Conference on Embedded and Ubiquitous Computing (EUC)* 2004c: 439-449

Waluyo, A.B., Bala Srinivasan, David Taniar: Optimal Broadcast Channel for Data Dissemination in Mobile Database Environment. *Advanced Parallel Programming Technologies (APPT)* 2003: 655-664

Wei, G., Jun Yu, Hanxiao Shi, Yun Ling: A Semantic-Driven Cache Management Approach for Mobile Applications. *International Conference on Computational Science* (3) 2006: 184-191

Wolfson, O., Bo Xu, Huabei Yin, Hu Cao: Searching Local Information in Mobile Databases. *The Intl Conference on Data Engineering (ICDE)* 2006. (online proceedings: http://www.informatik.uni-trier.de/~ley/db/conf/icde/icde2006.html#WolfsonXYC06)

Xie, T., Chaofeng Sha, Xiaoling Wang, Aoying Zhou: Approximate Top-k Structural Similarity Search over XML Documents. *Asia-Pacific Web Conference (APWeb)* 2006: 319-330.

Xin, S., Jiawei Han, Hong Cheng, Xiaolei Li: Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach. *The International Conference on Very Large Data Bases VLDB* 2006: 463-475

Xu, J., Xueyan Tang, Dik Lun Lee: Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments. *IEEE Trans. Knowl. Data Eng.* **15**(2): 474-488 (2003)

Xuan, P., Subhabrata Sen, Oscar González, Jesus Fernandez, Krithi Ramamritham: Broadcast on Demand: Efficient and Timely Dissemination of Data in Mobile Environments. *IEEE Real Time Technology and Applications Symposium* 1997: 38-48

Yiu, M.L., Xiangyuan Dai, Nikos Mamoulis, Michail Vaitis: Top-k Spatial Preference Queries. *The IEEE International Conference on Data Engineering (ICDE)* 2007: 1076-1085

Yu, H., Hua-Gang Li, Ping Wu, Divyakant Agrawal, Amr El Abbadi: Efficient Processing of Distributed Top-k Queries. *DEXA* 2005: 65-74

Yuen, J.C.H., Edward Chan, Kam-yiu Lam, Hei-Wing Leung: Cache Invalidation Scheme for Mobile Computing Systems with Real-time Data. *SIGMOD Record* **29**(4): 34-39 (2000)

Zhang, Q., Yu Sun, Xia Zhang, Xuezhi Wen, Zheng Liu: TOP-k Query Calculation in Peer-to-Peer Networks. *The ASIAN Computer Science Conference (ASIAN)* 2005: 127-135

Zheng, B., Jianliang Xu, Dik Lun Lee: Cache Invalidation and Replacement Strategies for Location-Dependent Data in Mobile Environments. *IEEE Trans. Computers* **51**(10): 1141-1153 (2002)

Zhu, M., Dimitris Papadias, Jun Zhang, Dik Lun Lee: Top-k Spatial Joins. *IEEE Trans. Knowl. Data Eng.* **17**(4): 567-579 (2005)