Data Warehousing Support for Mobile Environment

Ken C.K. Lee

M.Phil.

The Hong Kong Polytechnic University

October, 1999

Abstract

Over the last decade, the rapid advances in wireless communication technology and enhancement of portable computers have led to an evolution of mobile database applications. Typically, a mobile environment is built upon a cellular network which is a combination of a back-bone wired network and a set of small *wireless cells*. In the wired network, the *database servers* and the *base stations* are fixed in their locations. In the wireless cell, the *mobile clients* dynamically make connections to the base stations. To access information, a client queries one or more database servers via any connected base station. The relationship among these three entities could be roughly viewed as a three-tier client/server architecture. The base station works as a middle-tier to serve the other two entities. In this thesis, we generalize such an architecture into a framework of Mo*bile* W*arehousing* S*ystem* (**MoWS**) and investigate issues for improving the performance of client query processing in this new environment.

Very often, the population of mobile clients is huge compared with that of database servers in the mobile environment. A server being accessed by numerous clients will be overloaded. The query processing performance could not be guaranteed. In order to enhance the performance, a common approach is to replicate data in distributed hosts, from where clients can access information. In our research, we propose to maintain useful information pertaining to a subset of databases, which is of common interest to a handful of clients, in a form of materialized database view in the base stations. We intend to equip with the base stations ability to help answering client queries instead of merely directing the requests to the server; thus the server loading can be relieved to a large degree. We term each base station, which serves as a data repository for clients, *mobile data warehouse*.

One of the most important issues regarding data replication is data consistency maintenance. The replicated data becomes stale when the source is updated. In the scope of MoWS, we study a pull-based view update scheme for a mobile data

warehouse to request differential changes from the source database servers. In order to demonstrate the capability of addressing the view update anomaly problem due to server autonomy and asynchronous database updates, the correctness and complexity of our pull-based update scheme are studied.

In addition, we address the client query processing limitation over a narrow bandwidth and unreliable wireless channel. Accessing remote database server over a wireless channel will make it suffer from a lot of communication overhead. The narrow bandwidth amplifies data transfer latency while the unrelability causes a client occasionally disconnected. To cater for this problem, client caching is recommended. Most conventional caching schemes, such as page-based or record-based, are unable to assist clients to determine if there is sufficient cached data to answer their queries, thus forcing them to contact the server for possibly missing data. In this thesis, we suggest the use of a semantic caching scheme in which every query result associated with a semantic description is cached in a mobile client as a data block. By reasoning with the specification of an initiated query and the semantic description of the cached data block, a client becomes intelligent to assert whether the cache can contribute to answering the query completely and deduce what is missing from the cache. The main drawback of our scheme is an introduction of dynamic cache granularity that complicates the cache manipulation. We propose several cache management techniques for our semantic caching scheme.

Shortly concluded, MoWS is a hierarchical data replication framework in which information in database servers is replicated in the base stations and the mobile clients as materialized view and cache respectively. The strength of this design is that it enables mobile clients to answer their queries with little dependency on the wireless channel. In addition, mobile data warehouses are able to serve a mass of mobile clients, thus sharing the server loading. To quantify the performance of MoWS, we implement a prototype. We conduct a series of experiments based on the prototype,

along side with the appropriate quantitative analysis. From the results, we identified scenarios where our approach is beneficial, resulting in shorter response time, better cache hit, smaller transmission cost and lower storage overhead. These results demonstrate the effectiveness of our proposed schemes and the suitability of MoWS.

Acknowledgement

Looking back what I have done in these years, it is just like a long and tough journey. Now it seems to come closer to the finish line but I know it is a point of another start. Along the journey, I was accompanied by many. They let me know I am not alone in the research planet. In this moment, I would like to thank some of them.

First of all, I express my sincere gratitude to my supervisors, Dr. Hong-Va Leong and Dr. Antonio Si for their guidance and encouragement. Without their support, my research work cannot be carried out so smoothly. Also I am grateful to other research group members, Mr. Boris Chan, Mr. Stanley Yau and Mr. Jimmy Chim for their sharing, their working closely with me all through. In addition, I would like to thank my close friends, Mr. Chris Tsang and Mr. Daniel Duan for our being room-mates during the period in PQ720, a very nice office we studied in. Besides, I thank some of research students and departmental staffs for their kindness. They all provide me a warm and comfortable environment to study and to build up myself. Last but not the least, I would like to thank my family. They encouraged me and are willing to listen to my problem. This thesis is dedicated to all of them.

Contents

Chapter 1

Introduction

Over the past decade, the advances in wireless communication technology and improvement of low-cost portable computers have caused the emergence of a mobile computing paradigm [24, 37]. In this paradigm, users can communicate with each other with their portable computers while they are not required to stay in fixed positions and plug their computers to the network. This concept fosters a new direction of personal computing development which is proposed by the consortium of several world-wide IT companies in Bluetooth specification [31]. The objective of Bluetooth is to provide a standardized open platform to develop wireless network applications. Those applications will enable users to access, or even control remote resources with their hand-held computers (or probably the third generation mobile phone handset). The most important feature is that users will be able to retrieve up-to-date information anywhere at any time. Now, we are going to investigate the potential problems brought about the mobile computing paradigm in remote resources manipulation and research a new set of solutions.

1.1 What Is Mobile Database?

Within the context of the newly emerged mobile computing paradigm, users are free to roam about and are able to access remote database information over a wireless channel. In addition, the introduction of mobility capability has fostered and witnessed the development of a new class of database applications. An underlying database system supporting these database applications is called "Mobile Database". There are several motivative application examples. An investor can obtain timely up-to-date stock information, vital for his/her investment decision in the rapid changing financial market. Connectivity to a corporate database would allow a salesperson to demonstrate and inquire product information and stock level during traveling. Navigational data as well as traffic information could assist en route drivers in route planning and route selection, as in the Intelligent Transportation Systems (ITS) [16, 40].

The network configuration typically adopted in the mobile database is outlined in Figure 1.1. It consists of two main components [37]: a large back-bone wired network and a collection of small wireless networks. The wired network connects *database servers* and *base stations*, while an individual wireless network connects one base station and a set of *mobile clients*. In the wired network, both database servers and base stations are resided in fixed locations and communicate via a high speed data transmission link with rate varying from 10 Megabits per second (Mbps) in Ethernet to over 100 Mbps in ATM [24]. In a wireless network, a base station connects several mobile clients with a wireless channel of several kilobits per second, typically 19.2 kbps, data transmission rate.

Between wired network and wireless network are base stations which serve both database servers and mobile clients as middle-wares. They could convey messages from the wireless network to the wired network and vice versa. Because of decreasing signal strength along the distance away from the base station, the coverage of a wireless network is very limited. The area covered by a wireless network is referred

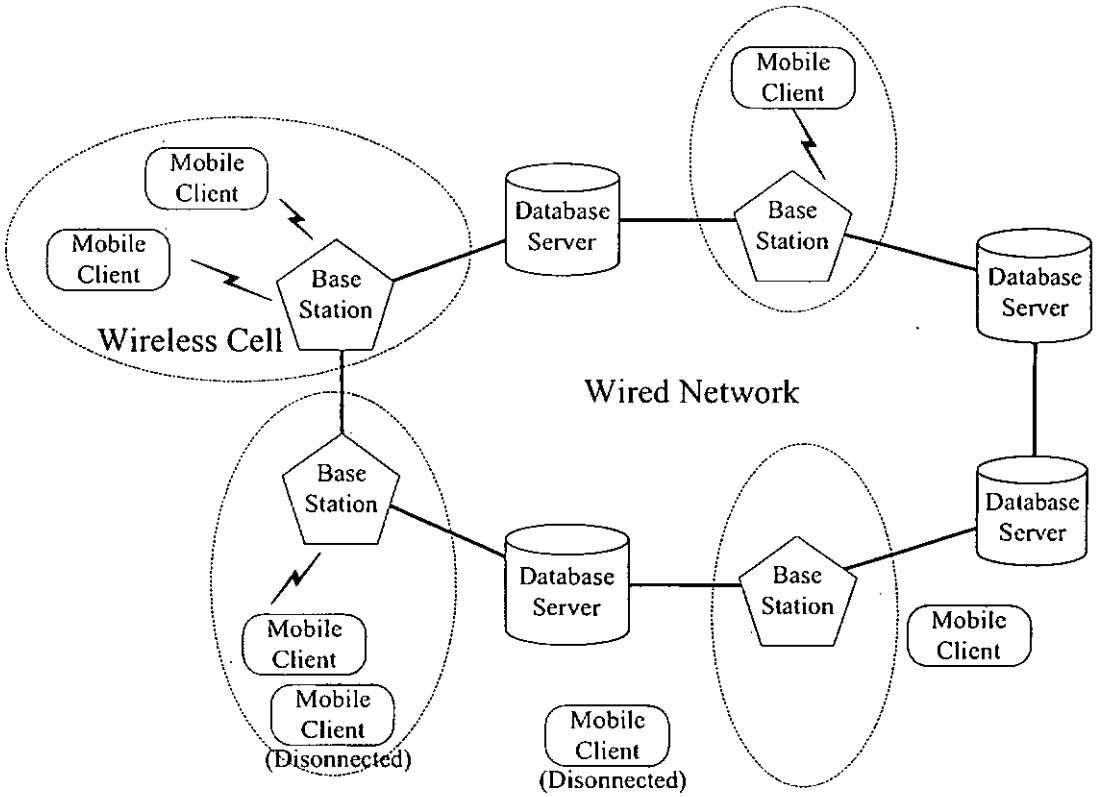


Figure 1.1: The Cellular Network Architecture

to as a cell. This network configuration, therefore, gains its name "*Cellular Network*" as a number of cells are tightly packed together. The overall coverage of the cellular network is expansible by incorporating more number of cells. This expansibility feature provides a very strong foundation to large-scale mobile database applications as well as world-wide mobile telephony services. For instance, GSM, TDMA and CDMA support a telephony service around the world by their underlying cellular structure.

Regardless of the presence of base stations, the infrastructure seems closely similar to a conventional client/server database architecture. Database servers provide information to clients. A client initiates a query to a database server, and the database server evaluates the query and replies an answer to the initiated client. Then one

would ponder, "Does it mean we can just reuse the existing mechanisms adopted in the conventional environment in this new environment?" The answer is absolutely "no". Unlike the conventional architecture, the interaction of clients and servers is now loosely coupled due to the clients being *weakly connected* in the wireless networks. Moreover, the population of mobile clients would be very huge compared to that of database servers. Imagine that a database server would be potentially queried by a million of clients around the world through GSM. Such high network overhead and an explosion of client population demands a set of new solutions.

1.2 Challenges in the Mobile Databases

The new issues of the mobile environment such as the large client population and the underlying network constraints, dictate the needs of new techniques in the system design and implementation. First of all, the database servers will be overloaded as a mass of clients query them. This will greatly affect the server response and hence the performance of client query processing cannot be guaranteed. Replicating data in distributed sites is a feasible and practical solution. To a certain degree, client querying secondary site instead of the server will relieve server loading. In the mobile environment, base stations will be appropriate candidiate to have replicated data since they are located in the reasonably beneficial position between clients and servers. They can answer some of the client queries and reduces the amount of queries forwarded to server. Then the issue to organize the replicated data in the base stations and the way to maintain replicated data consistency becomes challenging. We will discuss these issues in this thesis.

Secondly the network constraints in the mobile environment refers to the bandwidth limitation and the unreliability of the wireless channel which imposes a lot of communication overhead in remote client query processing. Narrow bandwidth leads

to a long transmission latency. This, in turn, prolongs the response time. Unreliability causes clients occasionally disconnected. It forces all communication activities to be suspended and even terminated. Those low bandwidth and low relability factors are the main bottleneck of system performance and they are not so influential in conventional client/server environment. To improve the query processing performance, the degree of network dependency of clients should be minimized. Client caching can alleviate the problem by providing certain amount of data for local query processing. However, most conventional caching schemes are item-based, that is, using a page or a record as a caching unit. They lack semantic information about the data. Every time when a query is initiated, a client itself cannot assert whether the cache can provide sufficient data to answer the query; this will force the client to contact the server for possibly missing data. This cripples the function of client caching. A new caching scheme is highly desirable. In this thesis, we will discuss a new semantic caching scheme which relates data according to its semantics into a data block.

In addition to those mentioned above, the emergence of the mobile database raises other issues such as client battery power limitation, communication asymmetry and client mobility. Client battery power limitation discourages clients to stay in an active mode for a long period. Clients would have only to operate in a doze mode most of the time in order to save power. The effect of battery power limitation contributes to an asymmetric communication infrastructure in this environment. The bandwidth of downlink from base stations to clients is always much larger than that of uplink in the reverse direction. It favors client tuning, not transmitting. It, in turn, encourages the use of broadcast to disseminate information of common interest. Moreover, client mobility requires client location information management and it fosters a class of location-dependent applications. Not all those issues are addressed in our study but we consider them as a future enhancement of our existing works.

1.3 Objectives of the Research

In our research, our focus is on addressing the two main problems mentioned in Section 1.2. In order to alleviate server overloading problem, we propose to have a database view derived from an arbitrary number of database servers, and it is materialized in base stations. We call those base stations with materialized views *mobile data warehouses*. The warehouse can process queries initiated by clients within the cell it serves. To maintain views updated properly, we derive a pull-based view update scheme for the warehouses. The warehouse requests differential changes from source database servers. We demonstrate the effectiveness and correctness of our scheme in the presence of a view update anomaly problem, which arises due to server autonomy and asynchronous database updates.

In order to relax the contention on the wireless channel and to equip mobile clients the capability to process queries locally when they are disconnected, we investigate the feasibility of a semantic query caching scheme. The benefit of the scheme is that it can relate individual cached data blocks, each of which is associated with a semantic description. Through reasoning with the specification of an initiated query and the semantic description of a cached data block, a client becomes able to examine if the query can be entirely answered using the cached data. We demonstrate mechanisms for transforming queries to reuse cached data and to fetch missing data from the server. However, our scheme induces the dynamic granularity of cached data blocks that complicates cache manipulation. We propose several cache management techniques to deal with it.

Strictly speaking, materialized view and client caching are two different forms of data replication. We apply these two forms in the mobile environment in a framework of **Mo***bile* **W***arehouse* **S***ystem* (**MoWS**). In the following chapters, we are going to discuss the system design and the implementation of the prototype of MoWS. To justify the suitability of MoWS, we evaluate the performance of each individual

proposed scheme based on the prototype and the appropriate quantitative analysis.

1.4 Thesis Organization

The remainder of this thesis is organized in 7 chapters. Chapter 2 will review previous literatures about mobile data management techniques, caching schemes and view management techniques. In Chapter 3, we shall describe an overall picture of MoWS with the detail system and operational specifications. In Chapter 4, we shall derive a scheme for a warehouse to request differential changes to its materialized view which is derived from single relation by giving its definition and its last update time. Furthermore the scheme is generalized to handle a view derived from multiple relations, no matter whether the relations are located in one database server or scattered in multiple servers. In Chapter 5, we shall discuss the semantic caching scheme. We shall outline how a query can be answered using the semantic cache, and discuss a new set of cache management techniques. In Chapter 6, we shall describe the implementation methodology and optimization techniques realized in MoWS. Next, in Chapter 7, we shall present an experimental evaluation with our prototype to show the suitability of the MoWS. Finally we shall summarize our work, conclude our contributions, and state our future plan in Chapter 8.

Chapter 2

Related Work

2.1 The WIND Project

MoWS is one of the major components in our research group's **WIND** (**WI**reless **N**etworked **D**atabase) project. The objective of **WIND** is to investigate a series of data management schemes, and to integrate those feasible schemes into a unified framework in order to enhance the effectiveness and efficiency of query processing (or even transaction processing) in the mobile database. The communication paradigms studied include pure server data broadcasting [51, 52, 54] and mobile client/server point-to-point communication [14, 49]. In broadcast schemes, clients are responsible to tune required data from a broadcast channel that is shared among different clients. Usually the broadcast carries information of common client interests. Over a broadcast channel, how data can be securely accessed by an appropriate client [70] and how a client evaluates its query [69, 72] were studied. Besides, in point-to-point schemes, the studies of two different client data replication schemes such as data caching in the context of OODB [14, 53, 55, 71] and maintenance of materialized database views [49] were conducted. Over a point-to-point wireless channel, clients access the database server for their particular interests. The issues of maintaining replicated data con-

sistency were discussed in [14, 71]. To avoid unnecessary message to confirm data expiry, we adopt an adaptive refresh time estimation based on database update frequency; that concept is borrowed from the idea proposed in *Leases file* [26]. Each data is associated with a server estimated refresh time. When a refresh time of a data expires, accessing that cached data would pay for the risk of reading stale data. Extending from study of materialized database views in [48, 49], the issue about how a client becomes able to self answer queries, and the support of query processing and view maintenance for multiple database servers are our focus in MoWS.

2.2 Related Work in Mobile Data Management

The differences between mobile database and conventional distributed database were discussed in [22], and the important impacts to mobile data management were also discussed in [36]. Mainly, the problems are the narrow bandwidth, limited mobile client power supply, client mobility and disconnection. They dictate the need of new data management schemes, including broadcasting strategies, efficient data caching and location management.

Data broadcast strategies include broadcast disk [1], broadcast indexing [68] and broadcast organization [35]. Clients listen to a broadcast to capture their interested information. Usually the broadcast is composed of information favored by majority, so not all required information can be obtained. Supplementary to a broadcast is an on-demand channel, or so-called a *back channel* which is a point-to-point channel. Over the back channel, clients' particular interests are obtained. In a practical environment, the allocation of channels for data broadcasting and for on-demand access becomes a key factor to the performance. This kind of channel allocation issue was studied in [2].

Scanning an entire broadcast or accessing over a dedicated back channel to pick up

required information would be time-consuming due to the lengthy data transmission latency in the mobile environment. Client caching is highly recommended. One of the earlier work studying the mobile data caching is [9]. Mobile clients maintain data in their caches and listen to the broadcast to invalidate or refresh the stale cache. The signature-based schemes has been proposed in [46, 50]. Besides, our research group proposed an alternative cache management scheme [14] based on refresh time estimation which can reduce communication overhead.

Location information management of mobile clients is another important issue. Initially it is emphasized for an efficient lookup mechanism in the mobile phone system. To yield faster client position lookup, a hierarchical cellular network with location register was suggested in [7]. It finds a client position by traversing the hierarchical location registers, in which the register of each level maintains location information of clients in a particular geographical zone, and the zone represented by each level subsumes that represented by all its child level. Also the issue of querying the spatial information in a mobile environment was discussed in [34]; example queries like "finding a close neighbor and who is a doctor?" are considered.

To enable operations to be performed during client disconnection period is another research issue in a mobile environment. With cached data in the client local storage, some of the queries can still be processed even when a client is disconnected. In Coda file system [42], version control is adopted during disconnected operation and a reintegration of updates is performed upon reconnection. Supporting file access in a mobile environment in the presence of weak connectivity has been introduced in later version of Coda through a new "write disconnected" state of operation [58]. Coda, however, is focused on file access rather than database record access.

2.3 Related Work in Caching Schemes

Over the last few years, caching in client-server database system has been studied quite extensively, mainly in the context of *data-shipping* model on which *item-base caching* schemes operate. A client loads a set of data items from a server by explicitly submitting an identifier list. As clients are equipped with powerful processors and large storage devices, the *query-shipping* database retrieval technologies have been improved. A client passes a query of a predefined syntax like ANSI SQL [80] to a server to gain the query result it wants. Hence, *semantic-based caching* becomes more feasible. The survey of *data-shipping* and *query-shipping* can be found in [10].

2.3.1 Navigational and Associative Query

A query is referred to as "*navigational*" if its retrieval mechanisms is from one data item related to another and so on. Usually it is common in the context of OODB. For a query accessing different pieces of information, it first accesses from an object and traverses one object's pointers to other objects recursively until the required information is obtained.

Totally different to the navigational query is the associative query. The associative query models user requirement in a predicate. The predicate is usually expressed as a set of boolean expressions. Each boolean expression could be a comparison between an attribute value and a constant, or between two attribute values. All retrieved records are related and they should fulfill the predicate.

2.3.2 Item-based Caching

In item-based caching schemes [14], all data items are of fixed size, which are often pages or records. Each item is a basic unit with a unique identifier. A page (record) is assigned with a page (record) identifier. Storage, retrieval and maintenance operates

at the client side based on those identifiers. Very often, those identifiers do not bear any meaning to the corresponding items. Therefore a client cannot make use of it to determine whether the local cache is sufficient to answer queries because the cache contains a subset of a database. Without contacting the server, the client can assert if there is missing result from cache.

2.3.3 Item-based Associative Caching

To support the client self-answering the queries, index-based associative caching is proposed that it supports the associative access by the item-based caching schemes. The schemes require a client to maintain an index page (or for short, index) that is a mapping between attribute values of data items to their corresponding item identifiers. When an associative query is initiated, the client examines the predicate, looks through an index to fetch local cached items and determine what data items are missing. Then it requests the server for the missing items if any. This mechanism is a comprehensive way to reuse the existing item-based caching schemes and support associative query evaluation in the client side. However, it has severe drawbacks. First, it is only applicable for the execution of an associative query whose predicates are with indexed attributes. Second, the maintenance of index consistency will suffer from very high index update overhead.

2.3.4 Semantic Query Caching

To improve the self-answering capability, cached data is collected and associated with the semantics. Then cache retrieval and manipulation are based on the semantics. Such new caching schemes are called semantic-based. One example is semantic query caching [47]. In semantic query caching schemes, every query result is cached as a unit with the description about what data is within that unit. Unlike index-based associative caching, semantic query caching does not need to maintain an index as well

as a huge number of identifiers. It maintains a relatively small amount of semantic descriptions instead. Therefore, it does not pay for the overhead of maintaining index consistency. To check whether a query can be answered entirely using the local cache, query reasoning that examines the semantic of the cache and the requirement of the query is used rather than index lookup.

The origin of semantic query caching is query reasoning in query optimization, especially multiple query plan optimization. Redundant database access and computation are minimized as the common intermediate results are identified and evaluated [23] as early as possible. Based on the similar concept, a query can reuse the result of another query as its intermediate result to improve the efficiency. Techniques to determine whether the result of one "conjunctive query" can be reused include query containment and query equivalence [38, 56, 73].

Semantic Query Caching in a Centralized Environment

Semantic query caching schemes in centralized database systems [15] store query results in primary memory to reduce disk access. Caching record pointers was shown to yield better performance than caching the whole records. Obviously since a small volume of cache storage can keep more record pointers than data records, it allows fast record lookup. The idea is similar to index-based associative caching but the index is dynamically created. However, caching record pointers might not be profitable in a distributed environment, especially in the mobile environment when the network bandwidth is low. This is simply because saving a remote pointer in a client's cache, be it in memory or storage, does not help in reducing the transmission overhead of the remote records. Furthermore, the absence of data record in the cache will render the cache almost useless in the event of a network disconnection.

Semantic Query Caching in a Distributed Environment

Semantic caching schemes in a distributed database system were discussed in [20, 21]. In [21], records of a relation are dynamically clustered and stored according to client's interest. Tuples are mapped into a multidimensional semantic space. Each cluster of tuples is cached as a unit and is described by a *restricted condition.* Comparing the predicate of a query with the restrict condition of a cache, a client becomes able to check what tuples are available and what are missing. Similarly, a relation is divided along tuples statically into several chunks [20], according to some predefined ranges on certain attributes. A query is processed by retrieving a set of chunks covering those required tuples. The hierarchical indexing structure of chunks improves query transformation efficiency. However, unnecessary overhead would be resulted especially when only a small portion of chunk is used. Also we note that, very often, clients will only request a certain subset of attributes rather than a whole tuple. This restriction on projecting selected attributes were not addressed in [20, 21]. In a mobile environment, the *project* operator is very useful in pruning unnecessary attributes from incurring extra communication overhead and response time. In MoWS, we extend the caching model proposed in [20] to cater for the conjunctive projection-selection-join queries derived from multiple relations and multiple database servers.

In additional to those mentioned, the most related works include ADMS± [64] and A*Cache [10]. In [64], a *ViewCache* scheme that uses the notions of *extended logical access path* and *incremental access methods* was proposed. Based on the *ViewCache* scheme, query result cached in a client was discussed in [65]. Updates are logged in the server. Any client initiated query reusing cache would explicitly call for cache refreshment from the server. The server then computes and propagates the differential changes to the client. Next, the A*Cache is using query results as caching units and several optimization techniques were used in [10]. The cache refreshment is by server eager updates notification.

Furthermore, authors in [66] discussed the design of an intelligent cache manager, named *WATCHMAN*, for caching query results in a data warehouse environment. However, it is limited in the read-only environment.

2.4 Related Work in View Management

By associating a semantic description with each caching unit in a client cache, the entire cache, in effect, can be regarded as a repository of materialized views in mobile clients [77]. It is therefore natural to apply view management techniques to manipulate the cache. Also in the warehouse, there are mainly a collection of materialized views. To handle view management, two aspects need to be addressed. First, a stale materialized view must get updated correctly and efficiently. Second, storage should be efficiently utilized.

2.4.1 View Updating

Existing view update mechanisms include view recomputation and incremental view update [49]. View recomputation regenerates a new view; overwrites the stale one every time. Incremental view update incorporates the differential changes to a view since it is last updated. Usually, database update in the mobile environment is quite infrequent [33], thus the latter approach is more favorable since the transmission overhead of small view changes should be less than that of an entire new view. Every time a view needs to be refreshed, a client will send the view definition to the server to compute and deliver the changes.

View update mechanisms could be roughly classified into *immediate*, *deferred* and *snapshot* [18]. An immediate view is one that is updated at once when there is a change at the server. Very often, maintaining an immediate view requires a source to notify a client whenever modifications are performed [12]. A deferred view is one

that is updated on demand when there is a need [17], say, a query accessing the view. Finally, a snapshot view is one that is updated periodically [57], such as hourly or daily. In the mobile environment, maintaining an immediate view is difficult because it is difficult for a client continuously listening to the database server to keep track of all updates, and for a server to contact those disconnected clients. There might be some updates missed [9]. A deferred or snapshot view is, therefore, more appropriate. In MoWS, we intend to maintain the materilized view when there is a need, so deferred approach is chosen.

Recently, there have been many algorithms proposed for incremental view updating [8, 29, 39, 45, 61], addressing a relational database model. The idea of incremental recomputation of a relational expression is proposed and the notions of insertion set and deletion set for efficient view updating are defined in [61] and supplemented by [27]. In [8], each tuple of a relation is associated with a tag, whose value is either "yes" or "no", indicating whether a tuple of a relation participates in deriving a particular view. The objective of using tags is to eliminate unnecessary message passing in a distributed environment if the database updates only involve the tuples with a "no" tag. The problem with this approach is that for each relation, it dedicates one extra tag attribute for each view derived from that relation. However, it is inflexible when there is a redefinition of views [28] and a change in the number of views that will lead to the modification of the database schema. In [39], the changes in a relation are determined with the help of a timestamp and a *backlog relation*. A timestamp is associated with each tuple. Its value is set to the transaction time of an update operation. A backlog relation is a file containing the operation detail on each tuple. A vacuum facility is provided to prevent the backlog relation from infinite growth.

The use of a *count* attribute associated with every resultant tuple of a view, indicating the number of tuples in the deriving relations that produce the same tuple in the view is mentioned in [29]. A modified union operator, "$\uplus$", is also provided to

operate on two sets of count associated tuples. The count attribute together with the $\uplus$ operator ensures the correctness of a materialized view. In our algorithm, we will also make use of the count attribute and the $\uplus$ operator.

The differential log file to handle updates on a materialized view is proposed in [67]. A log file is similar to the backlog relation in [39]. It records the types of operations and the changes on a tuple. During view updating, the file is scanned and the view changes are calculated. This work, however, only focuses on a restricted set of view definitions and it requires that tuples are assigned unique identifiers. Similar log scanning techniques for view updating can be found in [45].

In a warehouse, queries and view updates can be carried out at the same time. To resolve the conflict and maintain the correctness of the query results, an on-line warehouse view maintenance is proposed [62].

2.4.2 View Update Anomaly Problem

The view update anomaly problem was first addressed by [83] and subsequently investigated in [4, 84] and they discussed their algorithms in the context of five coherence requirements, namely, convergence, weak consistency, consistency, strong consistency and completeness in the order of increasing strictness. The anomaly problem emerges owing to server autonomy and concurrent asynchronous database updates. All these proposed approaches are applied in a situation where the database server has to notify every client whenever modifications are performed. Then each client formulates a query to ask for relative changes to the view [12]. In this environment, the database server has to be aware of which views are derived from it, and it has to be always connected to.

In [83], the *ECA* is proposed. When there is a pending request not yet replied from the server, a client formulates a query to ask for view change with compensation which cancels out the effect contributed by the last update. Extended from the single server

model, a family of *Strobe* algorithms are discussed in [84] for a view derived from multiple servers. Then in [4], the *SWEEP* algorithm is proposed to do compensation in the client side; an improved algorithm called *Nested SWEEP* buffers database acknowledgement and carries out view updating satisfying strong consistency. The detail discussion and comparison of those proposed algorithms is in [82]. Unlike these works, our approach to update a view derived from multiple servers needs to ask for changes from individual servers and performs a sequence of propagation, to be discussed in Chapter 4.

2.4.3 Storage Management

As discussed in [30, 41, 81], a set of sub-views are selected to be maintained instead of their final complete views. Obviously, admitting all views independently (*full materialization*) can provide the best performance in processing queries. However, there is a high possibility that there exists substantive overlapping sub-views among different views. Full materialization will thus incur a high storage overhead. With any materialized view updating, redundant data transmission is required.

By contrast, there is a cost saving alternative by separately storing overlapping sub-views and non-overlapping sub-views termed *semi-materialization* [41]. Original views become virtually defined on the materialized portions. The mixture of materialized and virtual views are arranged in a hierarchical structure [18]. Existing heuristic approaches to decide which views to be materialized [30, 81] are based on a static environment where all queries and views are predefined and access probability is known. They are inappropriate in a mobile environment, since they do not consider any storage capacity constraint which is a critical limitation of many mobile clients. Researches in the WATCHMAN project [66] studied their scheme including cache replacement algorithm and cache admission algorithm. It showed that their approach outperforms traditional LRU replacement scheme. In [20], replacement scheme is

extended to cater for the mobility of the mobile clients. However, both of them did not address cache update in their contexts. In our work, we consider the effect of storage capacity on caching and quantify the effect of applying replacement policy for cache management.

2.5 Related Work in Other Areas

Besides those issues discussed before, we considered others that are related to our work such as data replication, relation fragmentation and allocation, as well as concurrency control in the distributed database and multidatabase query optimization and query caching.

2.5.1 Data Replication

In a distributed database system, data could be replicated in several sites to reduce communication traffic cost. Very often, the decision made on data allocation is based on historical access information. In a rapid changing environment, an adaptive data replication scheme was proposed in [78]. The algorithm dynamically changes the data replication and data allocation during run time. The cost model of the algorithm considers overall read-write access in the whole system. When the frequency of read access gets higher than that of write access, more data replica is preferred, otherwise less replica is desirable. In addition, the concurrent data replica access is controlled by means of some algorithms like typical *majority consensus* [75] and *quorum voting* [5].

2.5.2 Relation Fragmentation and Fragment Allocation

Relation fragmentation is to partition a relation into sub-relations such that the portion of the relation could be separately allocated in sites where there are a lot of

accesses initiated. That intends to reduce communication traffic between the operation site and data allocation site. The way to make a fragment along tuples is called "horizontal" while along attributes, the fragmentation is named "vertical". Very often, we use both together and call it "hybrid" fragmentation. For semantic cache management in MoWS, a coarse query result would be partitioned into smaller pieces, that is similar to the idea of fragmentation but in our approach, partitioning is dynamically performed.

2.5.3 Concurrency Control

In our current work, we assume query processing and view updating are operating concurrently in the warehouse, the conflict and the inconsistency problem are resolved by using the scheme proposed by [62] that employs the ideas of two phase locking mechanisms and multi-version control protocols [59] in the distributed database. Besides, in the distributed database, other concurrency control protocols have been proposed and proved to produce serializable histories. The two well-known protocols are strict two phase locking and timestamp ordering [44]. To enhance concurrency, relaxed serializability and multi-versioning were suggested. One of the proposed relaxed forms of serializability is *epsilon serializability* [63, 79]. The way to isolate the two main related issues, multiversion control and concurrency control in distributed databases, was discussed in [6].

2.5.4 Multidatabase Query Optimization and Query Caching

Integrating multidatabases is a popular trend in a distributed database environment [76]. Since the data organization among the databases might be different, to provide a unified access interface, an additional middle-ware called *mediator* is provided [60]. Through the mediators of different servers, client application accesses information as if from one single database. When a query is posed, a mediator for-

mulates query plans for executions [32]. On the multidatabase access model, query caching is proposed in [3, 25] and we borrow the idea in client query processing with semantic caching and mobile data warehouse performs like a mediator in MoWS.

Chapter 3

System Model

The MoWS is developed upon the cellular network configuration. Mobile clients initiate queries to a base station which maintains a materialized database view derived from multiple databases, serving as a mobile data warehouse. Each client also maintains a cache for self query processing. This forms a hierarchy of data replication. We term this a framework of '**Mo***bile* **W***arehouse* **S***ystem*' (**MoWS**). Before exploring much detail, we would like to outline the picture illustrating the system and operation models in this chapter.

3.1 The MoWS Environment

In the MoWS, there are three types of entities: *mobile clients*, *base stations* and *database servers*. Each mobile client has processing power and non-volatile memory for query processing and caching. Each base station maintains a copy of client interested part of database from database servers in form of a materialized view and it serves numerous mobile clients. We refer this base station as a "mobile data warehouse". Each database server provides remote information access. The following describes the characteristics of the environment.

Autonomy: Mobile clients, warehouses and database servers are assumed to be autonomous. Each database server operates independently from one another. We assume there is no global update transaction operating on two or more servers [4]. Any update operation does effect on one server only. On the other hand, mobile clients are neither aware of other mobile clients nor of the cache contents of other clients. Each mobile client contacts solely with a local warehouse via a dedicated wireless channel in a wireless cell. In addition, the responsibility of a warehouse is to serve mobile clients and to contact server for view updating. The materialized view of each warehouse is independent from that of others.

Global Clock Synchronization: Although all database servers are independent, we assume all their activities are synchronized using various global clock synchronization algorithms [19].

Database Model: We assume that in the environment, all database are of the relational model. In addition, we do not make any assumption on the concurrency control or recovery mechanism in database access from a database server. We just require that when accessing a database server or a warehouse, the consistency of database state must be guaranteed.

Query Evaluation Capability: The mobile clients, warehouses and database servers are each assumed to possess query evaluation capabilities. When data is not available, query could be passed from clients to the warehouses and from the warehouses to the database servers. This feature exploits the utilization of resources of clients and warehouses, and parallel query executions. To a certain extent, workload from database servers is shared among mobile clients and warehouses. The overall performance can, therefore, be naturally enhanced.

Stateless Database Servers and Data Warehouses: The data warehouses and database servers are assumed to be stateless. Database servers do not keep information about from which a materialized view is derived. Warehouses do not maintain information about what data is cached in mobile clients and where the clients are currently located. This property reduces the mobile client state information storage and management overhead in the warehouses. It minimizes the cost in maintaining the state information consistency after clients recover from disconnection.

FIFO Message Passing: The message delivery follows the first-in-first-out discipline. The requests submitted by a given entity first are expected to be arrived at the destination first. Also the replies received by a given entity is in the same order as the corresponding requests it submitted. These characteristics can be achieved by associating a sequence number in all request/reply messages, or by means of logical clock or vector clock.

Sequential Client Query Initiation: Of a client, a new query initiation is permitted only after the previous one was finished. Each client query is supposed to be atomic, i.e., two consecutive queries are supposed to be independent. This assumption does not mean that there is no concurrent query initiation in the environment. Queries originated from multiple clients can reach a warehouse nearly at the same time.

Disconnection: We assume that wireless network disconnection does not occur when a client query is being processed. This assumption will be relaxed in future but not in this thesis. We also assume that the disconnection between a warehouse and database servers in the wired network seldom happens or even the disconnection in the wired network lasts for only a neglectable short period.

3.2 Formal Specification

3.2.1 Workgroup

In the mobile environment, there is a set of database servers, $\mathcal{DS} = \{DS_1, DS_2, \cdots, DS_{|\mathcal{DS}|}\}$, a set of base stations, working as warehouses, $\mathcal{W} = \{W_1, W_2, \cdots, W_{|\mathcal{W}|}\}$ together with a set of mobile clients, $\mathcal{MC} = \{MC_1, MC_2, \cdots, MC_{|\mathcal{MC}|}\}$. We consider an abstraction of a cooperation among multiple database serves, one warehouse and a pool of multiple clients as a workgroup w. Formally w is represented as a triple $<\mathcal{MC}_w, W_w, \mathcal{DS}_w>$ in which $\mathcal{MC}_w$ ($\subseteq \mathcal{MC}$) is a set of mobile clients who are served by warehouse W_w in a wireless cell, while W_w maintains a materialized view derived from $\mathcal{DS}_w$ ($\subseteq \mathcal{DS}$) which is a set of database servers. Notice that in one workgroup, there is only a warehouse. Without loss of generality, we will focus on the an operation of a workgroup in the following discussion.

3.2.2 Relational Database Model

Our research considers a relational database model. In a database DB, we assume there is a set of base relations, i.e., $DB = \{R_1, R_2, \cdots, R_{|\mathcal{DB}|}\}$. Each base relation R_i is a collection of tuples defined by a scheme A_i of n arbitrary attributes, i.e., $A_i = \{a_{i,1}, a_{i,2}, \cdots, a_{i,n}\}$. A group of attributes called key attribute, $K_i \subseteq A_i$, is used to identify a tuple in the relation. Corresponding to each attribute of all tuples the value domain $D_{i,j}$ of an attribute $a_{i,j}$ denotes a set of valid values. Given a tuple r of a relation R_i, its value denoted by $r[a_{i,j}]$ should be within $D_{i,j}$, i.e., $r[a_{i,j}] \in D_{i,j}$. Intuitively each tuple of a relation R_i can be regarded as a point in an n-dimensional space, $D_{i,1} \times D_{i,2} \cdots \times D_{i,n}$, in which $D_{i,x}$ will constitute the x^{th} orthogonal dimension. Then a subset of relation forms a smaller n-dimensional space, which is called the semantic subspace.

3.2.3 View Definition and Query Specification

In this stage, we consider the definition of a materialized view maintained in a warehouse to be of a conjunctive projection-selection-join expression [43] (that operand relations could be located in different database servers). Its formal expression is as follows:

$$V \stackrel{def}{=} \pi_{A_V}(\sigma_{Cond_V}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m))$$

where A_V is a subset of attributes from all relations, i.e., $A_V \subseteq \mathcal{A}$ where $\mathcal{A} = \cup_{i=1}^{m} A_i$. The $Cond_V$ is a list of predicates in a conjunctive normal form; $(v(a_1) \in d_{V,1}) \wedge (v(a_2) \in d_{V,2}) \wedge (v(a_{|\mathcal{A}|}) \in d_{V,|\mathcal{A}|})$, generalized as $\wedge_{i=1}^{|\mathcal{A}|}(v(a_i) \in d_{V,i})$ in which $v(a_i)$ is a value of an attribute a_i and $d_{V,i}$ is a sub-domain of D_i. Therefore, the values $r[a_i]$ of a retrieved tuple r should be equal to one of elements in $d_{V,i}$. Unless specified, otherwise each $d_{V,i}$ is defaulted to D_i. A condition $Cond_V$ is said to be *satisfiable* if all value domains are non-null sets. A view defined with a non-satisfiable condition results in no tuple. Here, the join operation is supposed to operate on a chain of different relations. No cyclic nor tree join is taken into current consideration but we will study these kinds of join arrangements in our extension work.

Further, we consider the specification of a client query $\mathcal{Q}$ which is evaluated on the materialized view in the warehouse, is expressed in a conjunctive selection-projection format, too:

$$\mathcal{Q} = \pi_{A_Q}\sigma_{Cond_Q}(V)$$

where A_Q is a subset of attributes of the view, i.e., $A_Q \subseteq A_V$, $Cond_Q$ is a condition of generalized expression, $\wedge_{i=1}^{|A_V|}(v(a_i) \in d_{Q,i})$ and its operand is the materialized view, V in the warehouse.

3.2.4 Database Update

Not only specifying the view definition and the query specification, we also concern the specification of database update operations. Each database update makes changes to a relation in a database server. It could be in general an insertion, a deletion or a modification. First, for an insertion, Ins, a new tuple is added into a relation R_i, that is,

$$R_i \leftarrow R_i \cup \{\{value_1, value_2, \cdots, value_{|A_i|}\}\}$$

in which each $value_x$ $(1 \leq x \leq |A_i|)$ is a value within default value domain of an attribute a_x, that is, $value_x \in D_{i,x}$. Second, for a deletion, Del, a set of tuples are removed from a relation R_i as they fulfill a certain selection condition and it is expressed as

$$R_i \leftarrow R_i - (\sigma_{Cond_{Del}}(R_i))$$

where $Cond_{Del}$ is $\wedge_{j=1}^{|A_i|}(v(a_{i,j}) \in d_{Del,j})$ and $d_{Del,j} \subseteq D_j$. Third, for a modification, Mod, certain attribute values of some tuples in a relation R_i satisfying certain criteria are altered. The operation is

$$\vartheta_{\exists j \Rightarrow 1 \leq j \leq |A_i| : a_{i,j} \leftarrow value_j}(\sigma_{Cond_{Mod}}(R_i)).$$

where ϑ is an update operator and condition $Cond_{Mod}$ is $\wedge_{j=1}^{|A_i|}(v(a_{i,j}) \in d_{Mod,j})$.

3.3 Workgroup Architecture and Operation

3.3.1 Overview

The overview of a workgroup architecture is shown in Figure 3.1. Within a wireless cell, a mass of mobile clients are served by the warehouse providing them a materialized view, which contains common client interested database items, derived from multiple database servers. Conceptually, the workgroup architecture can be regarded as

two coupling parts: a warehouse cooperates with multiple deriving database servers to maintain a materialized view in the wired network, and mobile clients initiates query to the warehouse, through which clients access the materialized view as if they retrieve information from the servers via a dedicated point-to-point wireless channel. When a client query arrives, the warehouse serve it with the view. To update the cache contents, we utilize the view maintenance mechanism proposed in warehouse view maintenance. In addition, client cache enables self query processing. In the following, we will outline the operation model of these two parts in two subsequent subsections respectively.

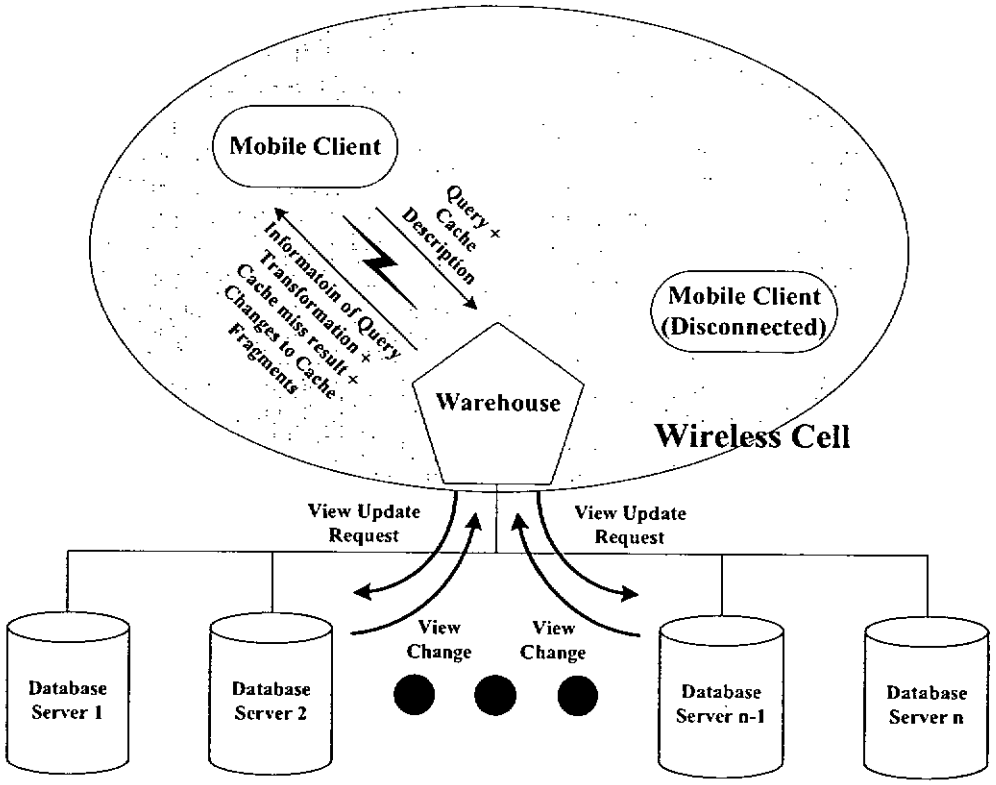


Figure 3.1: Workgroup Architecture

3.3.2 Warehouse View Maintenance

We assume that the warehouse materializes a view from several stateless database servers. Due to the autonomy of servers and asynchronous concurrent database updates, a view update anomaly problem arises [83]. To address it, existing view update mechanisms are based on server-push technology that requires the warehouse always listening to the server notification. They synchronize the updates and view update events based on the order of server notifications in FIFO channels. However, we have considered that database servers could be stateless; they do not know which warehouse maintains a view derived from them. They could not actively inform any entities about the database changes. Thus, complementing these push-based mechanisms, we derived a pull-based view update mechanism, in which a warehouse asks the servers about the changes. The initiation of the requests can be carried out on demand basis or periodically. Our pull-based view update relies on timestamps and it is based on an assumption that there is some clock synchronization algorithm to order the database activities. Before detail discussion of our approach in Chapter 4, we consider the following example to illustrate the view update anomaly problem.

Example # 3.1

We consider at the very beginning, there are three database servers, DS_1, DS_2 and DS_3 having relations R_1, R_2 and R_3 respectively. The contents of the relations are:

R_1

$a_{1,1}$	$a_{1,2}$
1	3
2	3
3	4

R_2

$a_{2,1}$	$a_{2,2}$
3	7
4	5
0	1

R_3

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
5	6	3
7	8	8
6	4	8

The materialized view, V, maintained in a warehouse at time t' is defined as $\pi_{a_{1,1},a_{3,1},a_{3,2}}\ (\sigma_{(v(a_{1,1}\in\{1,2\})\wedge(v(a_{3,2}\in\{5,\cdots,\infty\})}\ (R_1 \bowtie_{a_{1,2}=a_{2,1}} R_2 \bowtie_{a_{2,2}=a_{3,1}} R_3))$. Its content

should be:

V

a_1	a_2	a_3
1	7	8
2	7	8

Notice that this initial view content is correct as it can reflect the current content of database servers. Suppose at another time t'' ($t' < t''$), the warehouse contacts all servers for view updating, but there is only one database DS_1 that has been changed, it notifies the changes as a deletion of a tuple $\{2,3\}$ plus an insertion of tuple $\{2,4\}$. At that time, the contents of all relations are:

R_1

$a_{1,1}$	$a_{1,2}$
1	3
3	4
2	4

R_2

$a_{2,1}$	$a_{2,2}$
3	7
4	5
0	1

R_3

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
5	6	3
7	8	8
6	4	8

Based on the DS_1's notification, the warehouse sends two queries to DS_2: $R_2 \bowtie \{\{2,3\}\}$ and $R_2 \bowtie \{\{2,4\}\}$ to determine the corresponding changes. Then DS_2 evaluates the queries and delivers the answers: $\{\{2,3,3,7\}\}$ and $\{\{2,4,4,5\}\}$ to the warehouse. With same mechanism, the warehouse poses the queries $R_3 \bowtie \{\{2,3,3,7\}\}$ and $R_3 \bowtie \{2,4,4,5\}\}$ to DS_3. Unfortunately, before the queries are arrived at DS_3, the contents of DS_2 and DS_3 are changed individually. Their new contents become:

R_1

$a_{1,1}$	$a_{1,2}$
1	3
3	4
2	4

R_2

$a_{2,1}$	$a_{2,2}$
3	7
0	1
4	6

R_3

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
7	8	8
6	4	8
6	7	3

Thus DS_3 determine the results as {{2,3,3,7,8,8}} and {} respectively. After projecting out the required attributes, the net change of deletion of {{2,7,8}} and an insertion of nothing is determined and incorporated into the current view. Then an incorrect view content is resulted:

V

a_1	a_2	a_3
1	7	8

Notice that the correct view corresponding to the contents of all relations at time t'', that means the correct view content, should be:

V

a_1	a_2	a_3
1	7	8
2	5	6

The main causes of the problem is asynchronous database updates in DS_3 occurring during view updating.

□

3.3.3 Client Caching

Next we discuss how the processing of a client queries performed, and we briefly describe the idea of a semantic query caching scheme in MoWS. Every time after a query is processed, the result is cached. Since it covers a portion of the materialized view in the warehouse from which it is derived, the stored result is referred to as a "*cache fragment*", which is the basic caching unit. To describe the data contained therein, each cache fragment is associated with a semantic description that is exactly the expression of a query yielding the result as it is evaluated. Most of the time, the description of the entire cache (*cache description*), in turn, is a collection of

semantic description of all cache fragments. In addition, for management purpose, each cache fragment is associated with a summarized profile of access pattern, control information, such as the size of the cache fragment and its last update time. Usually the profile is condensed into a single value as replacement score for the sake of cache replacement.

When a wireless network is temporarily not available, no query of the client will be submitted for evaluation. The client cannot conduct any communication activities and has to evaluate queries using local cache only. When the client examines that all required data for a query is available in the cache, it answers the query locally and the query is said to be *completely self-answerable*. However, when only a portion of a query can be locally evaluated, the query is said to be *partially self-answerable*. Sometimes, if the data requirement of user applications is not so strict, partially self-answerable queries would be regarded acceptable. For example, looking a restaurant near the current location pf a mobile client does not dictate complete result, but only "reasonable" results.

When the wireless connection is available, the mobile client can initiate a query to the warehouse together with the cache description. The purpose of sending a cache description is to inform the warehouse what data is in the client cache. Based on the cache description, the warehouse transforms a query into a semantically equivalent construct of sub-queries, some of which could be evaluated in the client and some are required to be executed in the warehouse. Also, the warehouse determines the updates to the client cache so as to maintain cache consistency. Meanwhile when the query is being processed in the warehouse, the client pre-loads some possibly required cache fragments from the local cache. After the completion of query evaluation in the warehouse, the information of *query transformation*, *cache miss results* and *changes to cache fragments* are delivered back to the client. The changes to cache fragments are the differential changes to be incorporated in the corresponding cache fragments.

The cache miss results are the required data for the query that are not available in the cache. The information of query transformation is a piece of control information, which guides the client to construct a complete query result from the existing cache fragments that are loaded from the local cache and the cache miss results. Having received the reply, the client refreshes all cache fragments and then constructs the query result. Next, certain cache miss results are cached for later use requiring the cache to accommodate them. The important issues in the semantic query caching scheme like query transformation, cache coherence and cache storage management will be discussed in Chapter 5.

3.4 Chapter Summary

In this chapter, we gave an outline of the characteristics of MoWS environment and stated our assumptions. We then formalized the system model in several aspects, the definition of a workgroup, relation database model on which view definition, and query specification as well as database update were discussed. Further we briefly described the operation of a workgroup and outlined the idea of several important issues to be explored in next 2 chapters.

Chapter 4

Warehouse View Maintenance

View maintenance is a very important issue in MoWS. In our model, the database servers are assumed to be stateless; they are not aware of existence of a warehouse and of its view contents. So to update a materialized view, we will discuss a pull-based view update approach in this chapter. When there is a need, say, a new query that demands the most recent results is evaluated on the view or periodically, a warehouse sends its view definition to servers for determining differential change.

In the following, we will first outline how to enable a database server to determine the change to a materialized view provided that a view definition and a last view update time are available. We will then discuss the way to update a materialized view derived from more than one relations in a single server. We will further extend the basic mechanism to our WAVE algorithm catering for updating a view derived from multiple servers.

4.1 Stateless Server

All database servers are assumed to be stateless. They do not maintain warehouse information. In order to equip a server with a capability to determine the change of

the underlying database with respect to a materialized view definition, a special view update scheme is required. In this section, we will discuss the use of two time tags, *creation time tag* and *deletion time tag* in relation change determination.

In the database server, a pair of relations are used instead of a single base relation. They are called *currency relation* and *history relation*. The currency relation contains tuples describing the current database state while the history relation collects deleted tuples removed from the currency relation. Remember, all client queries are evaluated only on the currency relation. The function of the history relation is to reconstruct the past database state, that is useful in the context of a temporal database [74]. The currency relation is derived from the base relation whose schema has n attributes. It has appended an extra attribute called *creation time tag*, c_T, which records the insertion time of the associated tuple into the currency relation. The new schema could become $\{a_{i,1},a_{i,2},\cdots,a_{i,n},c_T\}$. For notational convenience, we retain R_i and A_i, representing the currency relation and its schema respectively. Corresponding to the currency relation, a history relation denoted by $\tilde{R}_i$ has similar schema $\tilde{A}_i$ with yet another one extra attribute called *deletion time tag*, d_T, so the schema becomes $\{a_{i,1},a_{i,2},\cdots,a_{i,n},c_T,d_T\}$. The deletion time tag is to record the deletion time of the associated tuple removed from the currency relation into the history relation. To limit the growth of a history relation, a vacuuming facility is employed to purge the old tuples periodically [39].

Example # 4.1

As shown in Example # 3.1, relations are stored in separate database servers. Each of them is substituted by a pair of currency relation and history relation. We assume that the creation time tags of all tuples in the currency relations are set to t_0 and all history relations are initially empty. Then the contents of the currency relations and the history relations of individual servers are presented:

DS_1 :

R_1

$a_{1,1}$	$a_{1,2}$	$c\tau$
1	3	t_0
2	3	t_0
3	4	t_0

$\tilde{R}_1$

$a_{1,1}$	$a_{1,2}$	$c\tau$	$d\tau$
[Empty]			

DS_2 :

R_2

$a_{2,1}$	$a_{2,2}$	$c\tau$
3	7	t_0
4	5	t_0
0	1	t_0

$\tilde{R}_2$

$a_{2,1}$	$a_{2,2}$	$c\tau$	$d\tau$
[Empty]			

DS_3 :

R_3

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$c\tau$
5	6	3	t_0
7	8	8	t_0
6	4	8	t_0

$\tilde{R}_3$

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$c\tau$	$d\tau$
[Empty]				

□

Updates in a database server could be either an insertion, *Ins*, a deletion, *Del*, or a modification, *Mod* on individual currency relations. Insertion causes a tuple to be added into a currency relation whereas deletion causes a tuples to be removed from a currency relation to the history relation. In our approach, modification of a tuple is considered as a deletion of the tuple instantly followed by an insertion of a new tuple with modified values. Therefore we concentrate the discussion on the insertion and deletion operations. Each such operation transforms the database from one state to another. For notational convenience, we denote the content of a relation R_i, at time t as R_i^t, while R_i will refer to the relation at the most current time now, i.e., R_i^{now}.

A tuple inserted into R_i is denoted as r_{Ins}. The creation time tag of r_{Ins} is automatically set to timestamp of the insertion, t_{Ins}, i.e. $r_{Ins}[c\tau] = t_{Ins}$. Similarly, a

tuple deleted from R_i is denoted as r_{Del}. The deleted tuple, r_{Del}, will be removed from R_i into the corresponding history relation $\tilde{R}_i$ and the deletion time tag is taken from the timestamp of the deletion, t_{Del}, i.e., $r_{Del}[d\tau] = t_{Del}$. As modifying a tuple is treated as a deletion-insertion pair, modifying any attribute of a tuple r_{Mod} of R_i into r'_{Mod} results in moving r_{Mod} to $\tilde{R}_i$, followed by inserting a new tuple r'_{Mod} with the new attribute values into R_i.

Example # 4.2

Suppose there are three update transactions $T_{1,1}$ at time t_1, $T_{2,1}$ at t_2 and $T_{3,1}$ at t_3 performed in three database servers where $t_0 < t_1 < t_2 < t_3$. They are

Begin Transaction $T_{1,1}$ on DS_1

$R_1 \leftarrow R_1 - (\sigma_{v(a_{1,1})\in\{2\}\wedge v(a_{1,2})\in\{3\}}(R_1))$;

$R_1 \leftarrow R_1 \cup \{2, 4\}$;

End Transaction $T_{1,1}$

Begin Transaction $T_{2,1}$ on DS_2

$R_2 \leftarrow R_2 - (\sigma_{v(a_{2,1})\in\{4\}}(R_2))$;

$R_2 \leftarrow R_2 \cup \{4, 6\}$;

End Transaction $T_{2,1}$

Begin Transaction $T_{3,1}$ on DS_3

$\vartheta_{a_{3,1}\leftarrow 6, a_{3,2}\leftarrow 7}\ (\sigma_{v(a_{3,1})\in\{5\}\wedge v(a_{3,2})\in\{6\}\wedge v(a_{3,3})\in\{3\}}(R_3))$;

End Transaction $T_{3,1}$

After the transactions are executed, the contents of the currency relations and the history relations are changed. The contents of the relations become:

DS_1 :

R_1

$a_{1,1}$	$a_{1,2}$	$c\tau$
1	3	t_0
3	4	t_0
2	4	t_1

$\tilde{R}_1$

$a_{1,1}$	$a_{1,2}$	$c\tau$	$d\tau$
2	3	t_0	t_1

DS_2 :

R_2

$a_{2,1}$	$a_{2,2}$	$c\tau$
3	7	t_0
0	1	t_0
4	6	t_2

$\tilde{R}_2$

$a_{2,1}$	$a_{2,2}$	$c\tau$	$d\tau$
4	5	t_0	t_2

DS_3 :

R_3

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$c\tau$
7	8	8	t_0
6	4	8	t_0
6	7	3	t_3

$\tilde{R}_3$

$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$c\tau$	$d\tau$
5	6	3	t_0	t_3

□

4.2 View Updating for a Single Relation

To update a materialized view, we should first study how to deduce the change of a relation in a database server since the view was created/lastly updated. Above all, we first introduce two terms called *insertion set* and *deletion set* with respect to a particular reference time, t_{ref}. The insertion set of a relation R_i, denoted as $\triangle_{t_{ref}} R_i$, is a set of tuples currently remaining in R_i and inserted into R_i after t_{ref}. Similarly, the deletion set of a relation, R_i, since t_{ref} is denoted as $\bigtriangledown_{t_{ref}} R_i$. It is defined as the set of tuples created before t_{ref} but removed from the relation after t_{ref}. Tuples belonging to these two sets could be easily identified by the creation time tag and

deletion time tag:

$\triangle_{t_{ref}} R_i = \{r \in R_i | r[c\tau] > t_{ref}\}$, and

$\triangledown_{t_{ref}} R_i = \pi_{\tilde{A}_i - \{d\tau\}} (\{r \in R_i | r[c\tau] < t_{ref} \wedge r[d\tau] > t_{ref}\})$.

The tuples belonging to the deletion set do not contain the deletion time tag. However, for clarity, we will leave out the outer projection operation on $(\tilde{A}_i - \{d\tau\})$ in the following discussion. With definitions of $\triangle_{t_{ref}} R_i$ and $\triangledown_{t_{ref}} R_i$, we can derive the relationship between R_i at a particular time, t_p and R_i at time t_{ref}, where $t \geq t_{ref}$ by

$$R_i^{t_p} = R_i^{t_{ref}} - \triangledown_{t_{ref}} R_i \cup \triangle_{t_{ref}} R_i, \quad \text{or}$$

$$R_i^{t_p} = R_i^{t_{ref}} \cup \triangle_{t_{ref}} R_i - \triangledown_{t_{ref}} R_i, \quad \text{since } \triangle_{t_{ref}} R_i \cap \triangledown_{t_{ref}} R_i = \emptyset.$$

Inversely, the relationship could be

$$R_i^{t_{ref}} = R_i^{t_p} - \triangle_{t_{ref}} R_i \cup \triangledown_{t_{ref}} R_i, \quad \text{or}$$

$$R_i^{t_{ref}} = R_i^{t_p} \cup \triangledown_{t_{ref}} R_i - \triangle_{t_{ref}} R_i.$$

Example # 4.3

Suppose at time t'' (where $t_0 < t' < t_1 < t'' < t_2$ such that only $T_{1,1}$ was executed and view was created at t'), the change of individual relations in different database servers with respect to time t' are:

DS_1 : $\triangle_{t'} R_1 = \{2,4,t'\}$

$\triangledown_{t'} R_1 = \{2,3,t'\}$

DS_2 : $\triangle_{t'} R_2 = \emptyset$

$\triangledown_{t'} R_2 = \emptyset$

DS_3 : $\triangle_{t'} R_3 = \emptyset$

$\triangledown_{t'} R_3 = \emptyset$

□

To enable a server to determine the change, each view V should be associated with a timestamp, $V.time$, to remark the creation/last update time. Thus, when refreshing the view, the definition as well as the timestamp should be supplied to the server. For the view V derived from a relation R_i with a definition, $\pi_{A_V}(Cond_V(R_i))$, to be refreshed at time t_{update}, we can have an updated view definition:

$$V^{t_{update}} = \pi_{A_V}(\sigma_{Cond_V}(R_i^{V.time} - \nabla_{V.time}R_i \cup \triangle_{V.time}R_i))$$

in which $R_i = R_i^{V.time} - \nabla_{V.time}R_i \cup \triangle_{V.time}R_i$. By distributing the relational algebra operations over the expression, the view definition can be changed to :

$$\begin{aligned} V^{t_{update}} = {} & \pi_{A_V}(\sigma_{Cond_V}(R_i^{V.time})) - \pi_{A_V}(\sigma_{Cond_V}(\nabla_{V.time}R_i)) \cup \\ & \pi_{A_V}(\sigma_{Cond_V}(\triangle_{V.time}R_i)). \end{aligned}$$

By associating the *count* attribute with every distinct tuple of each the above relational algebra expression, i.e., $A_V \cup \{count\}$, we obtain a simplified form:

$$V^{t_{update}} = V^{V.time} \uplus \nabla_{V.time}V \uplus \triangle_{V.time}V,$$

where the $V^{V.time}$ is the current view maintained in the warehouse, i.e., the view was created/lastly updated at time $V.time$. Notice that the definition of $V^{V.time}$ corresponds to $\pi_{A_V}(\sigma_{Cond_V}(R_i^{V.time}))$. The remaining two terms, $\nabla_{V.time}V$ and $\triangle_{V.time}V$, define respectively two sets of tuples to be removed from and inserted into the current view. The former corresponds to the relational algebra expression: $\pi_{A_V}(\sigma_{Cond_V}$ $(\nabla_{V.time}R_i))$ while the later corresponds to the expression: $\pi_{A_V}(\sigma_{Cond_V}(\triangle_{V.time}R_i))$.

The additional *count* attribute is important in maintaining the correctness of a view. This is because multiple tuples having the same values of A_V in R_i will contribute to one single tuple in the view, V; deleting one such tuple in R_i will result in removing the only tuple in the view without the *count* attribute. The value of

the *count* attribute in $\nabla_{V.time}V$ is negative since it represents the sets of tuples to be removed. Further notice that we have replaced the operators "−" and "∪" with associative "⊎" operator defined in [29]. The ⊎ operator works on two relations, R_a and R_b. Both R_a and R_b have same set of attributes, A, as well as the *count* attribute. $R_a \uplus R_b$ results in a new relation, $R_{R_a \uplus R_b}$ which has same set of attributes, A and the *count* attribute. Its content is defined by:

$$
\begin{aligned}
&R_{R_a \uplus R_b} \quad = \\
&\{ \quad r \mid ((\exists r_a \in R_a \wedge \exists r_b \in R_b \wedge \\
&\qquad r[A] = r_a[A] \wedge r[A] = r_b[A] \wedge r[count] = r_a[count] + r_b[count]) \vee \\
&\qquad ((\exists r_a \in R_a \wedge \forall r_b \in R_b \wedge r[A] = r_a[A] \wedge r[A] \neq r_b[A] \wedge r[count] = r_a[count]) \vee \\
&\qquad (\forall r_a \in R_a \wedge \exists r_b \in R_b \wedge r[A] = r_b[A] \wedge r[A] \neq r_a[A] \wedge r[count] = r_b[count])) \wedge \\
&\qquad (r[count] \neq 0)). \ \}
\end{aligned}
$$

Even though the currency relation and the history relation do not include the *count* attribute, in processing ⊎ operation, an implicit *count* value 1 is assumed. Since $V^{V.time}$ is the current view before the update. The server only needs to compute and to transmit $\nabla_{V.time}V \uplus \triangle_{V.time}V$, collectively denoted by $\delta^{t_{update}}_{V.time}V$, to the warehouse. This entity defines the net change that needs to be incorporated into the current view. For clarity, $\delta^{\mathsf{now}}_{V.time}V$ is just represented as $\delta_{V.time}V$. After the view is updated, $V.time$ is set to t_{update}.

Example # 4.4

Recalling from the previous example, a view derived from the relation relations is refreshed. The view definition V is first rewritten into three single-server definitions V_1, V_2 and V_3. Each component change with respect to the time t' is determined at time t''.

$\delta_{t'}V_1 = \{\{2,4\}[+1],\{2,3\}[-1]\}$

$\delta_{t'}V_2 = \emptyset$

$\delta_{t'}V_3 = \emptyset$

□

4.3 View Update for Multiple Relations

It is easy to generalize our algorithm for a view derived from multiple relations. We assume that there are m relations resided in one single database server and an order is imposed among relations where we considered an operand relation to be a predecessor of another relation when that relation is stated on the left side of another in the join chain. The view, V is defined as $\pi_{A_V}\ (\sigma_{Cond_V}(R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_m))$. We assume that the view was lastly updated at time $V.time$ and the view updating is performed at time t_{update} $(V.time \leq t_{update})$. Reordering the relational operations, we could rewrite the definition into: $V^{V.time} = \pi_{A_V}\ (\pi_{A_{V_1}}(\sigma_{Cond_{V_1}}(R_1)) \bowtie \pi_{A_{V_2}}(\sigma_{Cond_{V_2}}(R_2)) \bowtie \pi_{A_{V_3}}(\sigma_{Cond_{V_3}}(R_3)) \bowtie \cdots \bowtie \pi_{A_{V_m}}(\sigma_{Cond_{V_m}}(R_m)))$. Denoting $\pi_{A_{V_j}}(\sigma_{Cond_{V_j}}(R_j))$ by $V_j^{V.time}$, the view definition could be transformed into: $V^{V.time} = \pi_{A_V}\ (V_1^{V.time} \bowtie V_2^{V.time} \bowtie V_3^{V.time} \bowtie \cdots \bowtie V_m^{V.time})$. $V_j^{V.time}$ could be considered as the j^{th} component of the view derived from the currency relation R_i at time $V.time$.

Next, we discuss the way to determine the overall change to a materialized view derived from multiple relations in a single database server. Corresponding to V_i, $\delta_{V.time}V_i$ can be determined as previously discussed. The newly updated view becomes a projection of $((V_1^{V.time} \uplus \delta_{V.time}V_1) \bowtie (V_2^{V.time} \uplus \delta_{V.time}V_2) \bowtie (V_3^{V.time} \uplus \delta_{V.time}V_3) \bowtie \cdots \bowtie (V_m^{V.time} \uplus \delta_{V.time}V_m))$ on A_V. Each entry, $V_j^{V.time} \uplus \delta_{V.time}V_j$, is equivalent to V_j, the content of the j^{th} component of the view derived from the current content of currency relation R_i. The updated view content is simply the projection on A_V of

$$\left(\begin{array}{llllll}
 & \delta_{V.time}V_1 & \bowtie V_2 & \bowtie V_3 & \bowtie \cdots & \bowtie V_m \\
\uplus & V_1^{V.time} & \bowtie \delta_{V.time}V_2 & \bowtie V_3 & \bowtie \cdots & \bowtie V_m \\
\uplus & V_1^{V.time} & \bowtie V_2^{V.time} & \bowtie \delta_{V.time}V_3 & \bowtie \cdots & \bowtie V_m \\
\uplus & \cdots & & & & \\
\uplus & V_1^{V.time} & \bowtie V_2^{V.time} & \bowtie V_3^{V.time} & \bowtie \cdots & \bowtie \delta_{V.time}V_m \\
\uplus & V_1^{V.time} & \bowtie V_2^{V.time} & \bowtie V_3^{V.time} & \bowtie \cdots & \bowtie V_m^{V.time}
\end{array}\right)$$

The last entry constitutes the existing view content before the updating, i.e., $V^{V.time}$. The rest of the entries constitute the net change of the view, $\delta_{V.time}V$. Each entry represents the net change on a component of the view, $\delta_{V.time}V_j$, accumulating the change by joining other components. Starting from V_1, $\delta_{V.time}V_1$ joins with V_2, V_3 and so on, until all changes are accumulated. We denote each such entry, the *accumulated component net change* on a component V_j by $\partial_{V.time}V_j$. Each entry can be expressed in a unified form:

$$\partial_{V.time}V_j \quad = \quad (\bowtie_{l=1}^{j-1} V_l^{V.time}) \bowtie \delta_{V.time}V_j \bowtie (\bowtie_{r=j+1}^{m} V_r).$$

It is important to state the result of joining two relations, R_a and R_b, with the *count* attribute by multiplying the *count* values. Detail specification of $R_{R_a \bowtie R_b}$ is stated as:

$$\begin{aligned}
R_{R_a \bowtie R_b} = \{\ r | & \exists r_a \in R_a \wedge \exists r_b \in R_b \wedge \\
& r_a[J_a] = r_b[J_b] \wedge r[A_a] = r_a[A_a] \wedge r[A_b] = r_b[A_b] \wedge \\
& ((r_a[count] \geq 0 \wedge r_b[count] \geq 0 \wedge r[count] = r_a[count] \times r_b[count]) \vee \\
& (r[count] = -|r_a[count] \times r_b[count]|))\ \}
\end{aligned}$$

where J_a and J_b are the join attributes. When joining two tuples, either one with a *count* of negative value would produce a resultant tuple with a negative *count* value. From above, the overall change is represented as

$$\delta_{V.time}V \quad = \quad \uplus_{j=1}^{m} \partial_{V.time}V_j.$$

The net change of the view, $\delta_{V.time}V$ will be determined by performing a $\uplus$ operation on the individual accumulated net change. The net change of the view will then be incorporated into the existing materialized view in the warehouse and the last view update time is set to new update time, t_{update}.

4.4 View Update for Multiple Servers

In the previous sections, we have outlined how to determine the change of a view derived from a single relation as well as multiple relations in a single server. Extending the result again, we discuss here how to update a materialized view derived from more than one database servers. To update a view, the view definition is rewritten into sub-view definitions. For the sake of simplicity, we assume that each database server has only one relation. The problem, thus, becomes similar in nature to that for multiple relations within a single database server but with a distributed control. There are m database servers, DS_1, DS_2, DS_3, $\cdots$, DS_m and each DS_i contains a currency relation, R_i and a history relation $\tilde{R}_i$. V is defined as : π_{A_V} (σ_{Cond_V} $(R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_m)$).

To record the creation/last update time, the timestamp of a view is expressed in a vector, that is $V.time = < t_{p_1}, t_{p_2}, t_{p_3}, \cdots, t_{p_m} >$. Each element t_{p_j} of $V.time$ refers to the creation/last update time at the database server DS_j. For $V^{V.time} = \pi_{A_V}$ (σ_{Cond_V} ($R_1^{t_{p_1}} \bowtie R_2^{t_{p_2}} \bowtie R_3^{t_{p_3}} \bowtie \cdots \bowtie R_m^{t_{p_m}}$)), we could rewrite it: $V^{V.time}$ $= \pi_{A_V}$ ($\pi_{A_{V_1}}(\sigma_{Cond_{V_1}}(R_1^{t_{p_1}})) \bowtie \pi_{A_{V_2}}(\sigma_{Cond_{V_2}}(R_2^{t_{p_2}})) \bowtie \pi_{A_{V_3}}(\sigma_{Cond_{V_3}}(R_3^{t_{p_3}})) \bowtie \cdots \bowtie$ $\pi_{A_{V_m}}(\sigma_{Cond_{V_m}}(R_m^{t_{p_m}}))$). Substituting $\pi_{A_{V_j}}(\sigma_{Cond_{V_j}}(R_j^{t_{p_j}}))$ with $V_j^{t_{p_j}}$, the view definition could be transformed into : $V^{V.time} = \pi_{A_V}$ ($V_1^{t_{p_1}} \bowtie V_2^{t_{p_2}} \bowtie V_3^{t_{p_3}} \bowtie \cdots \bowtie V_m^{t_{p_m}}$). Every $V_j^{t_{p_j}}$ could be considered as the j^{th} component derived from the database server DS_j.

Now, assume that a warehouse intends to refresh its view. The new update time $V.time'$ is another vector of time, $<t_{q_1},t_{q_2},t_{q_3},\cdots,t_{q_m}>$ in which $t_{p_j} \leq t_{q_j}$. For each V_j, the warehouse obtains $\delta_{t_{p_j}}^{t_{q_j}} V_j$. The time t_{q_j} helps to freeze the database state even there is an update occurring at time after t_{q_j}. The updated view should be equal to a projection on A_V of $(V_1^{t_{p_1}} \uplus \delta_{t_{p_1}}^{t_{q_1}} V_1) \bowtie (V_2^{t_{p_2}} \uplus \delta_{t_{p_2}}^{t_{q_2}} V_2) \bowtie (V_3^{t_{p_3}} \uplus \delta_{t_{p_3}}^{t_{q_3}} V_3) \bowtie \cdots \bowtie (V_m^{t_{p_4}} \uplus \delta_{t_{p_m}}^{t_{q_m}} V_m)$. Alternatively, each $(V_j^{t_{p_j}} \uplus \delta_{t_{p_j}}^{t_{q_j}} V_j)$ corresponds to $V_j^{t_{q_j}}$, i.e., the content of the sub-view definition derived from database server DS_j at t_{q_j}. To simplify, we obtain the expression

$$\left(\begin{array}{llllll}
 & \delta_{t_{p_1}}^{t_{q_1}} V_1 & \bowtie V_2^{t_{q_2}} & \bowtie V_3^{t_{q_3}} & \bowtie \cdots & \bowtie V_m^{t_{q_3}} \\
\uplus & V_1^{t_{p_1}} & \bowtie \delta_{t_{p_2}}^{t_{q_2}} V_2 & \bowtie V_3^{t_{q_3}} & \bowtie \cdots & \bowtie V_m^{t_{q_3}} \\
\uplus & V_1^{t_{p_1}} & \bowtie V_2^{t_{p_2}} & \bowtie \delta_{t_{p_1}}^{t_{q_1}} V_3 & \bowtie \cdots & \bowtie V_m^{t_{q_3}} \\
\uplus & \cdots & & & & \\
\uplus & V_1^{t_{p_1}} & \bowtie V_2^{t_{p_2}} & \bowtie V_3^{t_{p_3}} & \bowtie \cdots & \bowtie \delta_{t_{p_m}}^{t_{q_m}} V_m \\
\uplus & V_1^{t_{p_1}} & \bowtie V_2^{t_{p_2}} & \bowtie V_3^{t_{p_3}} & \bowtie \cdots & \bowtie V_m^{t_{p_m}}
\end{array}\right)$$

The last entry, $V_1^{t_{p_1}} \bowtie V_2^{t_{p_2}} \bowtie V_3^{t_{p_3}} \bowtie \cdots \bowtie V_m^{t_{p_m}}$, constitutes the view content at $V.time = <t_{p_1},t_{p_2},t_{p_3},\cdots,t_{p_m}>$. The rest of the entries constitute the net change of the view, $\delta_{V.time}^{V.time'} V$. Notice that each entry again constitutes the accumulated component net change of a component V_j, $\partial_{t_{p_j}}^{t_{q_j}} V_j$, and can be unified into a generalized form:

$$\partial_{t_{p_j}}^{t_{q_j}} V_j \quad = \quad (\bowtie_{l=1}^{j-1} V_l^{t_{p_l}}) \bowtie \delta_{t_{p_j}}^{t_{q_j}} V_j \bowtie (\bowtie_{r=j+1}^{m} V_r^{t_{q_r}}).$$

The overall net change, $\delta_{V.time}^{V.time'} V$, should be $\uplus_{1 \leq j \leq m} \partial_{t_{p_j}}^{t_{q_j}} V$. The mechanism for updating a view derived from multiple database servers is similar to that derived from a single server with multiple relations. The only difference is that in the latter case, all ∂V_j could be determines at one database server, since all relations reside in a single server. However, in this situation, determining ∂V_j requires the warehouse sending

extra messages to the database servers, because the relations come from different servers.

4.5 The WAVE Algorithm

Based on the mechanism described in Section 4.4, we derive an algorithm called WAVE for the purpose. The algorithm earns its name since we regard each i^{th} component of a view V as a computation pulse starting at DS_i, spreading towards DS_1 and DS_m along the join chain of relations. The direction of the i^{th} computation pulse from i to 1 is said to be *left* while that from i to n is said to be *right*, assuming that the join chain of relation from R_1 to R_m is enumerated as a linear form arranged from left hand side to right hand side. To ensure the correctness, a vector of timestamps is used.

4.5.1 Basic WAVE Algorithm

The simplest version of WAVE algorithm is the basic WAVE algorithm. The implementation of the basic WAVE algorithm in database servers and a warehouse are depicted in Figure 4.1 and Figure 4.2 respectively. The presentation follows an object-oriented program style. The interaction between them is through object method invocation. Implementation of each remote method invocation is supposed to involve a pair of messages, i.e., a request from the caller to the callee and a response from the callee to the caller.

When there is a need to refresh a view, the warehouse function ViewUpdate accepts two parameters, V^{def} of type ViewDefinition and T declared as an array of CLOCK respectively. They represent the view definition and the creation/last update time. Next, the definition is rewritten into n sub-view definitions and stored in an array of definitions, v. To each database server presented by DatabaseServer[i], $v[i]$ is sent

```
MODULE DatabaseServer

    GLOBAL   R    :  CURRENCY_RELATION;
             R̃    :  HISTORY_RELATION;
             clock :  CLOCK; /* CURRENT DATABASE SERVER TIME */

    PROCEDURE Update(u :  DatabaseUpdate)
        perform u on R and R̃;
    END PROCEDURE

    FUNCTION LocalChange(T:Clock) → ViewChange
        RETURN   π_{Ã−{cτ,dτ}} (σ_{dτ>T∧cτ≤T} (R̃))
                 ⊎ π_{A−{cτ}} (σ_{cτ>T} (R));
    END FUNCTION

    FUNCTION AccumulateChange(V^def :  ViewDefinition,
             T :  Clock,δv :  ViewChange) → ViewChange
        /*   TO FREEZE THE DATABASE STATE AT TIME T,
             THE CHANGES SINCE T IS REMOVED */
        RETURN   (Evaluate(V^def,π_{A−{cτ}}R)
                 ⊎ −Evaluate(V^def,LocalChange(T))) ⋈ δv;
    END FUNCTION

    FUNCTION EvaluateLocalChange(V^def:ViewDefinition, T:Clock)
             → ViewChange,Clock
        RETURN   Evaluate(V^def,LocalChange(T)),clock;
    END FUNCTION

END MODULE
```

Figure 4.1: The Implementation of the WAVE Algorithm (Database Server)

```
MODULE Warehouse
    GLOBAL   V : View;
             V^def : ViewDefinition;
             V.time : array of CLOCK;

    FUNCTION ViewUpdate(V^def : ViewDefinition, T : array of CLOCK)
             → ViewChange,array of CLOCK
        VAR  v : array of ViewDefinition;
             i,l,r,n : INTEGER;
             t : array of Clock;
             ∂v : array of ViewChange;
             δV : ViewChange;
        δV ← ∅;
        v,n ← Decompose(V^def);
        PARALLEL FOR i ← 1 TO n
            ∂v[i], t[i] ← DatabaseServer[i].EvaluateLocalChange(v[i],T[i]);
        END PARALLEL FOR

        PARALLEL FOR i ← 1 TO n
            FOR l ← i − 1 DOWNTO 1 /* LEFT DIRECTION */
                ∂v[i] ← DatabaseServer[l].AccumulateChange(v[l],∂v[i],T[l]);
            END FOR
            FOR r ← i + 1 TO n /* RIGHT DIRECTION */
                ∂v[i] ← DatabaseServer[r].AccumulateChange(v[r],∂v[i],t[r]);
            END FOR
        END PARALLEL FOR

        FOR i ← 1 TO n
            δV ← δV ⊎ ∂v[i];
        END FOR

        RETURN π_{A_{V^def}}(δV),t;
    END FUNCTION

    PROCESS ViewMaintenance
        VAR δV : ViewChange;
            T : array of CLOCK;
        δV, T ← ViewUpdate(V^def,V.time)
        V ← V ⊎ δV;
        V.time ← T;
    END PROCESS
END MODULE
```

Figure 4.2: The Implementation of the WAVE Algorithm (Warehouse)

together with corresponding last update time $t[i]$ for the server through an invocation of remote object function EvaluateLocalChange. After completion of the function, the local change and the most current database time are returned and stored in variable $\partial v[i]$ and $t[i]$ respectively. To determine the net effect, $\partial v[i]$ is further propagated to other servers by calling function AccumulateChange along two directions. Finally the net change is accumulated by performing $\uplus$ on those individual changes and is maintained in δV. Analyzing the implementation, the number of message passing is simply the twice of total number of object methods invoked, that is 2 $\times$ (total number of EvaluateLocalChange called + total number of AccumulateChange called). The total number of times of calling EvaluateLocalChange is just n while that of calling AccumulateChange is $n \times (n-1)$. Thus total number of messages is exactly $2 \times n^2$. The message complexity is therefore $O(n^2)$.

4.5.2 Optimized WAVE Algorithm

Observation from the basic WAVE algorithm tells us that there could be a lot of ways to improve the efficiency in terms of the number of messages and transmission volume by exploring the parallelism and the use of semi-join reducer. It is possible to group certain computation pulses in one direction since all pulses should visit all database servers once. The optimized WAVE algorithm is devised and shown in Figure 4.3 and Figure 4.4 (any function that is not stated is assumed to be identical as that in the basic WAVE algorithm). After determining all local database changes, the computation pulses start from both ends, 1^{st} database server and the n^{th} database server. For instance, to accumulate net change along right propagation. The warehouse first contacts the 2^{nd} database server and propagates the join attributes of the local change $\partial f[1]$, together with the sub-view definition $f[2]$ and $t[2]$. Then, the accumulated component net change is determined and stored in $\partial R[1]$. Next to contact the 3^{rd} database server, the join attributes of the local change of $\partial f[2]$ and that of

$\partial R[1]$ are propagated with $f[3]$ and $t[3]$. Then the change are eventually stored in $\partial R[1]$ and $\partial R[2]$. This procedure goes on until the n^{th} database server is reached. At the end, all ∂R elements store parts of the accumulated component net change after the right direction propagation. Similarly, all ∂L elements store parts of the accumulated component net change after the left direction propagation originated at n^{th} database server. Finally the net change to a view is computed by performing $\uplus$ on all $\partial L[i] \bowtie \partial f[i] \bowtie \partial R[i]$. In the optimization, the number of messages is reduced a lot. The total number of times of calling EvaluateLocalChange remains at n. However the function OptimizedAccumulateChange is invoked $(n-1)+(n-1)$ times since there are two propagation directions. Finally the message complexity is $O(n)$.

```
MODULE DatabaseServer

    ...
    FUNCTION AccumulateChange(v^def:ViewDefinition,T:array of CLOCK,
             δv:array of ViewChange,j:INTEGER)
             → array of ViewChange
        VAR ∂v   :  array of ViewChange;
            i    :  INTEGER;
            R^T  :  RELATION;

        R^T ←    (Evaluate(v^def, π_{A-{cT}} R)
                 ⊎ −Evaluate(v^def,LocalChange(T[i])))
        FOR i ← 1 TO j
            ∂v[i] ← R^T ⋉ δv[i]; /* SEMI-JOIN*/
        END FOR

        RETURN ∂v;
    END FUNCTION

    ...

END MODULE
```

Figure 4.3: The Implementation of the Optimized WAVE Algorithm (Database Server)

```
MODULE Warehouse
    ...
    FUNCTION ViewUpdate(V^def : ViewDefinition,T : array of CLOCK)
            → ViewChange, array of CLOCK
        VAR v : array of ViewDefinition;
            i,j,l,r,n : INTEGER; t : array of Clock; δV : ViewChange;
            ∂v,∂L,∂l,∂R,∂r : array of ViewChange;

        v,n ← Decompose(V^def);
        PARALLEL FOR i ← 1 to n
            ∂v[i], t[i] ← DatabaseServer[i].EvaluateLocalChange(v[i],T[i]);
        END PARALLEL FOR
        FOR i ← 1 to n
            /* ANY RELATION "R" JOIN WITH IDENTITY, RESULT "R" */
            ∂L[i] ← IDENTITY; ∂R[i] ← IDENTITY;
        END FOR
        PARALLEL
            FOR j ← n − 1 DOWNTO 1 /* LEFT DIRECTION */
                FOR l ← 1 TO n − j − 1
                    ∂l[l] ← π_JoinAtt(∂L[n − l + 1]); END FOR
                ∂l[n − j] ← π_JoinAtt(∂v[j + 1]);
                ∂l ← DatabaseServer[j].AccumulateChange(v[j],T[j],∂l,n − j);
                FOR l ← j + 1 TO n
                    ∂L[l] ← ∂l[l − j] ⋈ ∂L[l]; END FOR
            END FOR
            FOR i ← 2 TO n /* RIGHT DIRECTION */
                FOR r ← 1 TO i − 2
                    ∂r[r] ← π_JoinAtt(∂R[r]); END FOR
                ∂r[i − 1] ← π_JoinAtt(∂v[i − 1]);
                ∂r ← DatabaseServer[i].AccumulateChange(v[i],t[i],∂r,i − 1);
                FOR r ← 1 TO i − 1
                    ∂R[r] ← ∂R[r] ⋈ ∂r[r]; END FOR
            END FOR
        END PARALLEL

        FOR i ← 1 TO n
            δV ← δV ⊎ (∂L[i] ⋈ ∂v[i] ⋈ ∂R[i]);
        END FOR
        RETURN π_{A_{V^def}}(δV),t;
    END FUNCTION
    ...
END MODULE
```

Figure 4.4: The Implementation of the Optimized WAVE algorithm (Warehouse)

4.5.3 Correctness of WAVE Algorithms

We have discussed basic WAVE and optimized WAVE algorithms and we showed the message complexity accordingly. Now we would like to discuss what level of coherence requirement our algorithms can fulfill. In [83], five coherence requirements are stated namely *convergence*, *weak consistency*, *consistency*, *strong consistency* and *completeness* in the order of increasing strictness. Let the state (content) of a database source at any time t be denoted by ss_t and the state (content) of a materialized view at any time t be denoted by mv_t. Convergence requires that at a final time t_f, the time when all finite database updates finished, mv_{t_f} should reflect the corresponding database state, i.e., ss_{t_f}. In the meantime, the requirement of weak consistency is stricter than that of convergence. It requires that the set of states of the materialized view should reflect a certain set of states of the database source. Next, the consistency requires that any pair of materialized view states is in the same order as the corresponding pair of database states, that means $mv_{t_i} < mv_{t_j} \Rightarrow ss_{t_x} < ss_{t_y}$ $(i < j \wedge x < y)$ where "$a < b$" means a partial order between state a and b, more specific, a before b, and where mv_{t_i} and mv_{t_j} reflect ss_{t_x} and ss_{t_y} respectively. Strong consistency requires both consistency and convergence. Finally, completeness requires every change database state is reflected by that of materialized view and in the exact order. However, this requirement is too strong and might not be necessary in most practical applications. In accordance with those requirements, our WAVE is able to fulfill strong consistency. Our algorithms is based on the use of timestamps to resolve the update anomaly problem. The timestamps are incrementally updated every time a warehouse asks database sources for their local changes. Therefore, a materialized view state must reflect a database state in the same temporal order. That satisfies the consistency requirement. Using our algorithm, the warehouse repeatedly asks the database sources for the changes even the database update are being performed or all database updates have finished such that the warehouse can eventually capture

the final state of the database source. That satisfies the convergence requirement. Therefore our algorithm guarantees the strong consistency requirement. To end up this section, we demonstrate with an example how our optimized WAVE algorithm can do view updating and fulfill the strong consistency requirement.

Example # 4.5

Using the optimized WAVE algorithm, the update anomaly problem can be resolved. We demonstrate the running of the algorithm based on Example 3.1. *V.time* is initiated to $< t', t', t' >$. Just as before, the warehouse gets the change and the time from DS_1: {<2,4>[1],<2,3>[-1]} with t'', the current database times of DS_1 while those from DS_2 and DS_3 are empty and t'' for the first contact with servers. Those local changes and times are stored in the array ∂v and t.

Then the warehouse propagates changes in both left and right directions. For the right propagation, the warehouse first sends the join attributes {<4>,<3>} together with the definition of V_2 stored in $v[2]$ and time t' in $t[2]$ to DS_2. Notice that the join attribute does not require attachment of *count* since they are used to get the quantified tuples. Then DS_2 replies with {<4,5>[1],<3,7>[1]} which is stored in $R[1]$. To perform further propagation, the join attributes {<5>,<7>} from $R[1]$ and that of $\partial v[2]$ (that is empty) are sent along with the sub-view definition, $f[3]$ and time $t[3]$ to DS_3. Suppose at that time, $T_{3,1}$ is executed before request arrives. After DS_3 computation (the effect of database updates are compensated by $R^{t''} \leftarrow R - \triangle_{t''} R \cup \triangledown_{t''} R$), the tuples {<5,6>[1],<7,8>[1]} are returned. At last, the accumulated change {<4,5,5,6>[1],<3,7,7,8>[1]} are stored in $R[1]$, and $R[2]$ remains empty.

Since the local changes of V_2 and V_3 stored in $\partial f[2]$ and $\partial f[3]$ are empty, the left propagation would result in empty $L[2]$ and $L[3]$. After all, the accumulated change due to the local change of DS_1 is just {<2,4[1]>,<2,3>[-1]} $\bowtie$ {<4,5,5,6>[1], <3,7,7,8>[1]}. At the end, the final result after the projection of required attributes of the joined result is {<2,5,6>[1],<2,7,8>[-1]}. After incorporating the change to

V, the content becomes:

V

$a_{1,1}$	$a_{3,1}$	$a_{3,2}$
1	7	8
2	5	6

and the $V.time$ is set to $< t'', t'', t'' >$.

□

4.6 Chapter Summary

In this chapter, we discussed how a stateless server becomes able to determine the change of a view when a view definition and a last update time are provided. Based on this idea, we extended the mechanism to determine the change to a view derived from multiple relations even the relations are located in one or more servers. The implementation of the mechanism is called WAVE, which propagates local changes at each server to other servers to gather the accumulated net change. Further, we discussed the optimized version of the algorithm in which the propagation of all pulses in one direction are grouped. And we showed that the optimized version is more efficient in terms of less number of messages.

Chapter 5

Semantic Query Caching

Semantic query caching scheme uses semantic information to manage and manipulate the entire cache. To exploit the cache semantics, each cache fragment (cached previous query result) is associated with a semantic description, which is exactly the expression of the deriving query. Based on query reasoning, it is capable for a client to determine how to reuse the cache fragments in answering queries and to examine what is missing from the cache. To reuse a cache fragment and to represent the missing data, a query is rewritten into a probe query, extracting data from the cache fragment, and a supplementary query submitted for the warehouse evaluation. This query rewriting procedure is referred to as *query transformation.*

In semantic query caching scheme, the sizes of all cache fragments are not fixed. This dynamic cache granularity property renders cache accommodation more complicated. Cache replacement is no longer as simple as discarding one fragment to admit a new one. In addition, a cache fragment might contain data of different access frequency. To efficiently utilize the cache storage, portion of a cache fragment of higher access frequency is preferred for retainment while that of lower access frequency is needed to be discarded. This issue raises a need of *cache decomposition* that divides a cache fragment into several smaller ones such that the divided fragments contain

data of same access frequency. To maintain the cache coherence, the cache description is sent to the warehouse every time a query is initiated. In MoWS, the mechanism of determining the differential change from the warehouse is similar to warehouse view maintenance against a database server, so we will not discuss this issue in this chapter.

In the following sections, we focus the issue of query transformation and cache accommodation which are based on the assumption that the query and cache fragments are all derived from the same view in the warehouse.

5.1 Relationship between a Cache Fragment and a Query

Query transformation is a mechanism to rewrite a query into a semantically equivalent construct of sub-queries. Evaluation of such equivalent construct of sub-queries yields an identical result as that of original query. Some sub-queries are evaluated in the warehouse while others are evaluated in the mobile client. This query rewriting mechanism is implemented in both warehouse and mobile clients. In client connection mode, a mobile client sends a query together with a cache description to the warehouse where query transformation is carried out. In disconnection mode, query transformation is forced to run in the client.

For notational convenience, we denote the cache of a mobile client, MC by C, which consists of a set of cache fragments, i.e. $C = \{F_1, F_2, \cdots, F_{|C|}\}$. Upon defining the cache, we are now at the position to discuss how to make use of cache fragments in answering a query. As there is a cache fragment available, query transformation could now be started as identifying the required data from each cache fragment that could satisfy a query and defining the semantics of the data missing from the fragments. There exists various relationships between the content of a cache fragment, F_i, and

the result of a query (or simply query), Q, when both of them are defined on the same operand. These relationships are depicted in Figure 5.1. In the figure, the white and dark grey blocks represent a cache fragment and a query respectively, while the light grey block represents the data common both to cache fragment and query. Any horizontal sub-block represents a collection of tuples while any vertical sub-block denotes a set of attribute values.

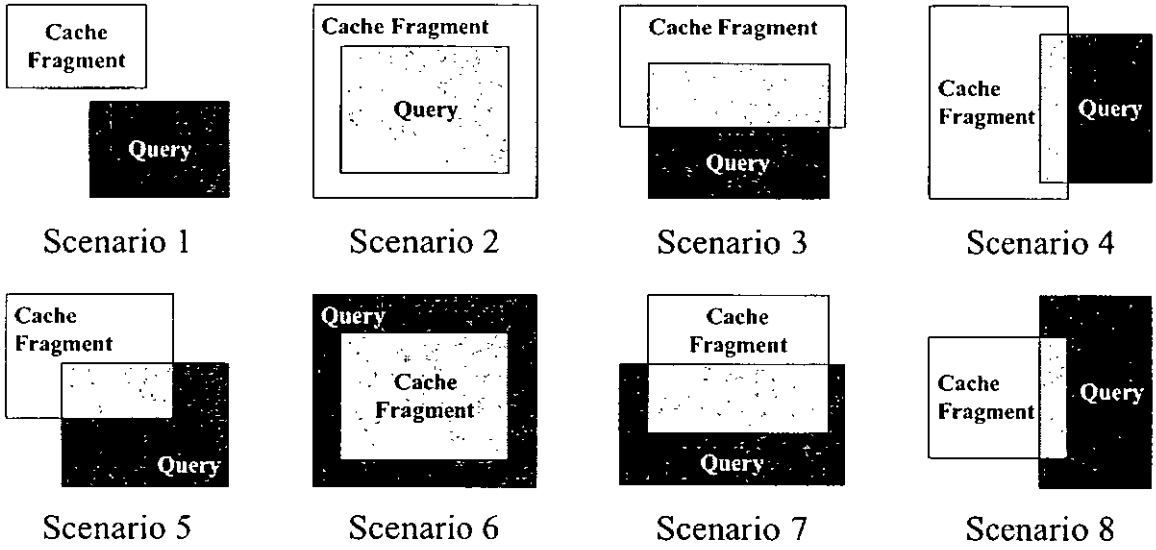


Figure 5.1: Relationships between a Cache Fragment and a Query

The possible relationships between them can be categorized into eight scenarios :

Scenario 1: A query and a cache fragment are disjoint. This implies that the cache fragment contains no data required by the query.

Scenario 2: A query is fully enclosed within a cache fragment, implying that the required result are completely available in the cache fragment.

Scenario 3: A horizontal sub-block of a query is covered by a cache fragment.

Scenario 4: A vertical sub-block of a query is covered by a cache fragment.

Scenario 5: This is a more generic case as a combination of Scenarios 3 and 4.

Scenario 6: This is a complementary case to Scenario 2. A cache fragment is fully enclosed within a query. Since tuples and attributes are unordered, by moving all tuples belonging to the top sub-block to the bottom and moving all attributes belonging to the left sub-block to the right, this scenario could be reduced to Scenario 5.

Scenario 7: This is a complementary case to Scenario 3. Again, attributes are unordered; the vertical sub-block could be re-arranged, reducing to Scenario 5.

Scenario 8: This final case is complementary to Scenario 4. Rearranging the horizontal sub-block, it also reduces to Scenario 5.

In all scenarios, except Scenario 1, a cache fragment can be used to answer a query to a certain degree. We generalize all eight scenarios by transforming a query, Q, into two sub-queries: a probe query, Q^P, and a supplementary query, Q^S. A probe query is responsible for retrieving data from a cache fragment that is the light grey block. A supplementary query, by contrast, is responsible for retrieving data from the server which is the dark grey block.

Example # 5.1

To make the presentation easier to understand, we illustrate an example where a warehouse maintains a view V. The schema A of V contains 7 attributes, i.e., $A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$, each of which is of type integer and the domain is ranging from $\{-\infty,\cdots,\infty\}$ and we assume a_1 is the key of the schema. At the very beginning, the mobile client initiates a query, Q_0, where expression is $\pi_{a_1,a_5,a_6}(\sigma_{(v(a_1)\in\{1,2\})\wedge(v(a_6)\in\{5,\cdots,\infty\}}(V))$. After evaluation, the result of the query is cached as a cache fragment, F_0 and its semantic description is set to the specification of Q_0. Subsequently, F_0 could be reused to answer client initiated queries.

□

5.2 Query Transformation

5.2.1 Query Transformation with One Cache Fragment

To determine how a cache fragment can be used in answering a query, let us first examine the concept of *query intersection* between two queries, Q_a and Q_b, on the same operand view V with a schema A. The intersection between them, denoted by $Q_a \cap Q_b$, defines the common projection attributes of those tuples that are shared by the two queries. Formally, the specification of $Q_a \cap Q_b$ is

$$\pi_{A_{Q_a} \cap A_{Q_b}}(\sigma_{Cond_{Q_a} \wedge Cond_{Q_b}}(V))$$

where the projection attributes $A_{Q_a} \cap A_{Q_b}$ refers to the common projection attributes while $Cond_{Q_a} \wedge Cond_{Q_b}$ defines the combined condition. It means $((v(a_1) \in d_{Q_a,1}) \wedge (v(a_1) \in d_{Q_b,1})) \wedge ((v(a_2) \in d_{Q_a,2}) \wedge (v(a_2) \in d_{Q_b,2})) \wedge \cdots \wedge ((v(a_{|A|}) \in d_{Q_a,|A|})$, in short, $\wedge_{i=1}^{|A|} (v(a_i) \in (d_{Q_a,i} \cap d_{Q_b,i}))$, according to which common tuples from both queries are qualified. With the definition of query intersection, all the above eight scenarios can be handled with a uniform framework.

Scenario 1: In the first scenario, the result of query Q is disjointed from the content of cache fragment F_i. The result of the probe query, Q^P, should be an empty set, $\emptyset$, while the supplementary query is set to the original query, i.e., $Q^S = Q$. The way to determine the disjointness of Q and F_i is based on the testing of the emptiness of their intersection. The intersection is empty when either the set of common attributes (except key attributes) is empty, $(A_Q - K) \cap A_{F_i} = \emptyset$, or the resultant condition, $Cond_Q \wedge Cond_{F_i}$, is not satisfiable that means they never reason 'true' by all means.

Example # 5.2

Based on Example # 5.1, suppose a query $Q_1 = \pi_{a_1,a_5,a_6}\ (\sigma_{v(a_1)\in\{1,2,3\}\wedge v(a_6)\in\{-\infty,\cdots,4\}}\ (V))$, is issued and the previous result of query Q_0 is maintained as a cache fragment F_0 in the cache which description is Q_0. Using F_0, Q_1 is examined whether it is self-answerable. After the examination of the query intersection, Q_1 cannot reuse F_0 since the combined condition is not satisfiable even though there are common projection attributes. □

Scenario 2: The second scenario represents the situation where query Q is completely self-answerable. This scenario occurs when the intersection between Q and F_i is the query itself, i.e., $Q \cap F_i = Q$ and one additional constraint must be fulfilled that involved attributes in all predicates of $Cond_Q$ should appear in the cache fragment projection or the value domain is exactly equal to that of corresponding predicate in the cache fragment, i.e., $\forall(v(a_j) \in d_{Q_j})$ appears in $Cond_Q : a_j \in A_{F_i} \vee d_{Q,j} = d_{F_i,j}$. This condition ensures that the tuples extracted from the cache fragment are exactly what Q requires. In case the constraint does not hold, the projection of key attributes of those tuples satisfying combined condition, $Cond_Q \wedge Cond_{F_i}$, from the server should be required to filter out the unwanted tuples from the probe query using semi-join operator. Finally, the result of the supplementary query, Q^S, is empty, and the probe query, Q^P, is set to $\pi_{A_Q \cap A_{F_i}}(\sigma_{Cond'_Q}(F_i))$ where $Cond'_Q$ is a condition in which the predicates of those attributes absent in the cache fragment projection are removed. Formally, $Cond'_Q = \wedge_{\forall(v(a_j)\in d_{Q,j})\text{ appear in }Cond_Q \wedge a_j \in A_{F_i}}(a_j \in d_{Q,j})$.

Example # 5.3

Next, we consider another query, Q_2, which is stated as :
$\pi_{a_1,a_5}\ (\sigma_{(v(a_1)\in\{1,2\})\wedge(v(a_5)\in\{-\infty,\cdots,5\})\wedge(v(a_6)\in\{5,\cdots,\infty\})}\ (V))$. The non-empty intersec-

tion of F_0 and Q_2 as $\pi_{a_1,a_5}\ (\sigma_{(v(a_1)\in\{1,2\})\wedge(v(a_5)\in\{-\infty,\cdots,5\})\wedge(v(a_6)\in\{5,\cdots,\infty\})}\ (V))$ is examined. Then the probe query is produced as $\pi_{a_1,a_5}\ (\sigma_{(v(a_5)\in\{-\infty,\cdots,5\})}\ (F_0))$, in which the predicate corresponding to a_1 and a_6 is omitted since the domains restricted in a_1 and a_6 of Q_2 are exactly matched to that of F_0. Also, the presence of a_5 in the fragment supports the evaluation of the predicate $(v(a_5) \in \{-\infty, \cdots, 5\})$. The tuples retrieved from F_0 are exactly what Q_2 required, no extra projection of key attributes from server are demanded. So, Q_2 is completely self-answerable. □

Scenario 3: In this scenario, some tuples required by query Q could be found in the cache fragment F_i. The probe query, Q^P, and the supplementary query Q^S, represent two disjoint horizontal portions of the complete query result. We call the supplementary query, a *horizontal supplementary query*, Q^H. Queries of this category will have a non-empty intersection $Q \cap F_i$ such that the projection attributes of the query $A_Q \cap A_{F_i} = A_Q$ and $Cond_Q \wedge Cond_{F_i}$ is satisfiable but $Cond_Q \neq Cond_{F_i}$. The probe query, Q^P, and the horizontal supplementary query, Q^H, are defined as $\pi_{A_Q}(\sigma_{Cond_Q \wedge Cond_{F_i}}(F_i))$ and $\pi_{A_Q}(\sigma_{Cond_Q \wedge \neg Cond_{F_i}}(V))$ respectively. Since the negation of a predicate in a conjunctive form, $\neg Cond_{F_i}$, produces another predicate in a disjunctive form that violates our initial consideration on conjunctive projection-selection query, we use another method to evaluate this negation. For the operand, V, with a schema, A, we should have $|A|$ horizontal supplementary sub-queries (that is recursive bisection of $|A|$-dimensional semantic space). Collectively, a horizontal supplementary query, Q^H is the union of all sub-queries, i.e., $Q^H = \cup_{y=1}^{|A|} Q^{H,y}$. The condition $Cond_{Q^{H,y}}$ of each $Q^{H,y}$ $(1 \leq y \leq |A|)$ is expressed as $\wedge_{l=1}^{y-1}(a_l \in (d_{Q,l} \cap d_{F_i,l})) \wedge (a_y \in (d_{Q,y} - d_{F_i,y})) \wedge_{r=y+1}^{|A|}(a_r \in d_{Q,r})$. Thus the expression of each $Q^{H,y}$ is $\pi_{A_Q}(\sigma_{Cond_{Q^{H,y}}}(V))$. Finally the results to the original query Q will thus be the union of both queries i.e., $Q^P \cup Q^H$.

Example # 5.4

Comparing with F_0, a new query, Q_3, is examined with specification π_{a_1,a_5} $(\sigma_{v(a_1)\in\{1,2,3\}\wedge v(a_6)\in\{4,\cdots,\infty\}}\ (V))$. A non-empty intersection, $Q_3 \cap F_0$, $\pi_{a_1,a_5}\ (\sigma_{v(a_1)\in\{1,2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V))$ is resulted. The probe query, Q_3^P, is π_{a_1,a_5} (F_0) while the horizontal supplementary query Q_3^H contains 7 sub-queries (since there are totally 7 attributes) but only two of them have satisfiable conditions. They are $\pi_{a_1,a_5}\ (\sigma_{(v(a_1)\in\{3\})\wedge(v(a_6)\in\{4,\cdots,\infty\})}\ (V))$ and $\pi_{a_1,a_5}\ (\sigma_{(v(a_1)\in\{1,2\})\wedge(v(a_6)\in\{4\})}$ $(V))$. □

Scenario 4: Here, only certain attributes required by query Q could be found in F_i. The probe query, Q^P, and the supplementary query, Q^S, represent two disjoint vertical portions of the complete query result. We call the supplementary query, a *vertical supplementary query*, Q^V. Queries of this type will have a non-empty intersection $Q \cap F_i$ such that $(A_Q \cap (A_{F_i} - \mathcal{K})) \subset A_Q$ (excluding equality) and $Cond_Q \wedge Cond_{F_i} = Cond_Q$, i.e., $\forall j : 1 \leq j \leq |A| \Rightarrow (d_{Q,j} \cap d_{F_i,j}) = d_{Q,j}$.

The vertical supplementary query, Q^V, is defined as $\pi_{(A_Q - A_{F_i})\cup\mathcal{K}}(\sigma_{Cond_Q}(V))$. The probe query, Q^P, is defined as $\pi_{A_Q\cap A_{F_i}}(\sigma_{Cond'_Q}(F_i))$ that is closely similar to Scenario 2. Then the results to the original query, Q, will be defined by $Q^P \bowtie_{\mathcal{K}} Q^V$.

Example # 5.5

Another query, Q_4, is specified as $\pi_{a_1,a_3,a_5}\ (\sigma_{v(a_1)\in\{1,2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V))$. A non-empty intersection, $Q_4 \cap F_0$, $\pi_{a_1,a_5}\ (\sigma_{v(a_1)\in\{1,2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V))$ is yielded. The probe query, Q_4^P, is $\pi_{a_1,a_5}\ (F_0)$ while the vertical supplementary query Q_4^V is $\pi_{a_1,a_3}\ (\sigma_{(v(a_1)\in\{1,2\})\wedge(v(a_6)\in\{5,\cdots,\infty\})}\ (V))$. □

Scenario 5: In this generalized scenario (including Scenario 6 to 8), the original query Q needs to be transformed twice into a vertical and a set of horizontal

supplementary queries so as to identify precisely what data needs to be obtained from the server. The horizontal supplementary query, Q^H, is defined in the same way as in Scenario 3, while the vertical supplementary query Q^V, is defined similar to Scenario 4. The result of the original query Q is defined as $(Q^P \bowtie_{\mathcal{K}} Q^V) \cup Q^H$. It is shown in Figure 5.2.

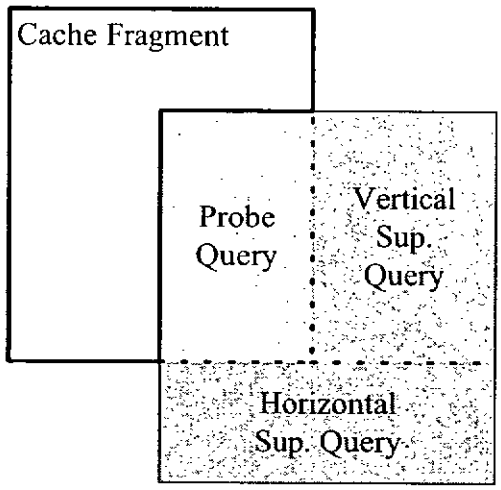


Figure 5.2: Query Transformation with a Cache Fragment

Example # 5.6

The specification of a new query, Q_5, is $\pi_{a_1,a_3,a_5}\ (\sigma_{v(a_1)\in\{2,3\}\wedge v(a_6)\in\{4,\cdots,\infty\}}\ (V))$, an intersection with F_0 produces non-empty result. The probe query, Q_5^P, is $\pi_{a_1,a_5}\ (\sigma_{v(a_1)\in\{2\}}\ (F_0))$. The vertical supplementary query, Q_5^V, is $\pi_{a_1,a_3}\ (\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V))$ while two horizontal supplementary sub-queries are : $\pi_{a_1,a_3,a_5}\ (\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{4\}}\ (V))$ and $\pi_{a_1,a_3,a_5}\ (\sigma_{v(a_1)\in\{3\}\wedge v(a_6)\in\{4,\cdots,\infty\}}\ (V))$.

□

In the remaining discussion, we use the generalized scenario (Scenario 5). A query reusing a cache fragment is transformed into three main sub-queries, a probe query, a vertical supplementary query and a horizontal supplementary query.

5.2.2 Query Transformation with Multiple Cache Fragments

As there is a sufficient number of reusable cache fragments, a query could be transformed repeatedly. We outline a function called QueryTransform in Figure 5.3. The function is invoked with two parameters: Q of type Query and C of type CacheFragmentDescription, both of which are representing the expressions of a query and a collection of cache fragment descriptions respectively. First of all, a temporary result for transformed query T is initialized to the original Q. In a condition that Q cannot be rewritten, T remains equal to Q that implies no cache fragment could be used. Next, to reduce the computation overhead, it must exclude those non-reusable cache fragments by evaluating the intersection between a query and a cache fragment description. Then one cache fragment is picked up for transforming a query into a probe query, a vertical supplementary query and a collection of horizontal supplementary queries with a function Transform whose logic has been discussed in Section 5.2. Furthermore, the rest of the reusable cache fragments, $U - \{u\}$, are used to transform the supplementary query by calling QueryTransform again.

Example # 5.7

To illustrate the recursive query transformation, let us consider there are two cache fragments available in the cache. One is the result of Q_0, F_0, from Example # 5.1 and another is the result of Q_1, F_1, from Example # 5.2. They are disjoint. We reuse the query Q_5 from Example # 5.6 to be transformed with F_0 first and then F_1, the resultant transformed query should be

$$(\pi_{a_1,a_5}(\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}(F_0)) \bowtie \pi_{a_1,a_3}(\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}(V))) \cup$$
$$(\pi_{a_1,a_3,a_5}(\sigma_{v(a_1)\in\{3\}\wedge v(a_6)\in\{5,\cdots,\infty\}}(V)) \cup$$
$$\pi_{a_1,a_5}(\sigma_{v(a_1)\in\{2,3\}\wedge v(a_6)\in\{4\}}(F_1)) \bowtie \pi_{a_1,a_5}(\sigma_{v(a_1)\in\{2,3\}\wedge v(a_6)\in\{4\}}(V))))$$

□

```
Function QueryTransform(Q:Query,C:set of CacheFragmentDescription)
    → TransformedQuery
    VAR T : TransformedQuery;
        U : set of CacheFragmentDescription;
        u : CacheFragmentDescription;
        Q^P, Q^V : Query;
        Q^H : set of Query;
        q : Query;
        i,j : INTEGER;

    T ← Q;
    U ← ∅;

    FOR ALL c ∈ C /* FILTER ALL REUSABLE CACHE FRAGMENT */
        IF (non-empty intersection between Q and c)
            U ← U ∪ {c};
        ENDIF
    ENDFOR

    IF U ≠ ∅ THEN
        PICK u FROM U

        Transform(Q,u) → Q^P, Q^V, Q^H;
        /* USING A CACHE FRAGMENT TO TRANSFORM A QUERY
           INTO A PROBE, A VERTICAL AND A HORIZONTAL SUBQUERY */

        T.Q^P ← Q^P;
        T.Q^V ← QueryTransform(Q^V,U − {u});
        FOR ALL q ∈ Q^H
            T.Q^{H,i} ← QueryTransform(q,U − {u});
        ENDFOR
    ENDIF
    RETURN T

EndFunction
```

Figure 5.3: Query Transformation Function

5.3 Cache Management

In evaluating a continuous stream of queries, being able to retain frequently accessed cached data in the cache is highly desirable. However, not all data in a cache fragment has the same access frequency. Intuitively, blindly maintaining a whole cache fragment would reduce the cache hit because of extra storage overhead incurred in keeping those less frequently accessed data. Coarse cache granularity thus decreases effective cache space utilization. In our research, we intend to cater for the issue of identifying and extracting accessed data in a portion from a cache fragment, subdividing a fragment into smaller ones and discarding sub-fragments not being reused. This subdivision mechanism is known as *cache decomposition.*

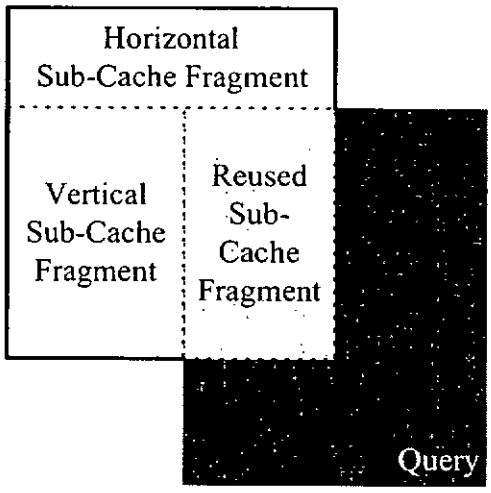


Figure 5.4: Decomposition of a Cache Fragment Reused by a Query

In detail, our cache decomposition scheme is similar to query transformation. Consider Figure 5.4 that is our generalized scenario of query transformation, a cache fragment, F_i, is reused by a query Q. It can be then divided into three pieces. The reused sub-cache fragment, F_i^U, is simply the reused portion by a query. The horizontal sub-cache fragment, F_i^H represents the non-reused tuples, while the vertical sub-cache fragment, F_i^V, refers to the remaining non-reused attributes of the reused

tuples. Implementation of cache decomposition could be performed by migration of data from the original cache fragment to the new sub-cache fragments and removal of the old fragment. Migrating appropriate data from the original cache fragment to the new sub-cache fragments is achieved through the evaluation of *extracting queries* on the cache fragment. The nature of extracting queries are exactly the same as that of the probe query. In the scenario, three different extracting queries will be posed.

An extracting query that extracts data from the original cache fragment to a reused sub-cache fragment, F_i^U, is the probe query, $Q^P = \pi_{A_Q \cap A_{F_i}}(\sigma_{Cond'_Q}(F_i))$. The remaining extracting queries which extract data to a vertical sub-cache fragment F_i^V and to a horizontal sub-cache fragment F_i^H that is $\cup_{y=1}^{|A_{F_i}|} F_i^{H,y}$, are similar to those mentioned in Scenario 5 in Section 5.2. Denoting predicates in $Cond'_Q$ by $(v(a'_1) \in d'_{Q,1}) \wedge (v(a'_2) \in d'_{Q,2}) \wedge \cdots, (v(a'_{|A_{F_i}|}) \in d'_{Q,|A_{F_i}|})$, where $a'_i \in A_{F_i}$, an extracting query of F_i^V can be defined $\pi_{(A_{F_i} - A_Q) \cup K}(\sigma_{Cond'_Q}(F_i))$ and an extracting query of each horizontal extracting query, $F_i^{H,y}$, is $\pi_{A_{F_i}}(\sigma_{Cond_{F_i^{H,y}}'}(F_i))$ where $Cond'_{F_i^{H,y}}$ is equal to $\wedge_{l=1}^{y-1}(v(a'_l) \in (d'_{F_i,l} \cap d_{Q,\alpha})) \wedge (v(a'_y) \in (d'_{F_i,y} - d_{Q,\beta})) \wedge_{r=y+1}^{|A_{F_i}|}(v(a'_r) \in d_{F_i,\gamma})$ in which $\exists \alpha, \beta, \gamma : 1 \leq \alpha < \beta < \gamma \leq |A| : (v(a_\alpha) \in d_{Q,\alpha})$, $(v(a_\beta) \in d_{Q,\beta})$ and $(v(a_\gamma) \in d_{Q,\gamma})$ appear in $Cond_Q \wedge (a'_l = a_\alpha \wedge a'_y = a_\beta \wedge a'_r = a_\gamma)$.

After a fragment is decomposed, each new sub-cache fragment becomes an independent unit, with its own semantic description and access information. Its description is defined by combining the selection condition of its corresponding extracting query and that of original cache fragment. Therefore, the description of the reused sub-cache fragment, F_i^U, is $\pi_{A_Q \cap A_{F_i}}(\sigma_{Cond'_Q \wedge Cond_{F_i}}(V))$ while the description of the vertical sub-cache fragment, F_i^V, and horizontal sub-cache fragment, $F_i^{H,y}$, are $\pi_{(A_{F_i} - A_Q) \cup K}(\sigma_{Cond'_Q \wedge Cond_{F_i}}(V))$ and $\pi_{(A_{F_i} - A_Q) \cup K}(\sigma_{\neg Cond'_Q \wedge Cond_{F_i}}(V))$. The horizontal sub-cache fragment, in turn, is a set of fragments whose predicates are of conjunctive form. Each of them is expressed as $\pi_{A_{F_i}}(\sigma_{Cond_{F_i^{H,y}} \wedge Cond_{F_i}}(V))$.

To select fragments to be discarded in order to free space for new fragments if

needed, each cache fragment is assigned a replacement score in accordance with different replacement schemes. When a cache fragment is accessed, its score is updated. After cache decomposition, new scores are assigned to individual sub-fragments. We assign the reused sub-cache fragment a new score but other sub-cache fragments will inherit the original score.

Example # 5.8

Recalling from Example # 5.6, a cache fragment, F_0 is reused by a query, Q_5. The three extracting queries of reused sub-cache fragment, vertical sub-cache fragment and horizontal sub-cache fragment are $\pi_{a_1,a_5}\ (\sigma_{v(a_1)\in\{2\}}\ (F_0))$, $\pi_{a_1,a_6}\ (\sigma_{v(a_1)\in\{2\}}\ (F_0))$ and $\pi_{a_1,a_5,a_6}\ (\sigma_{v(a_1)\in\{1\}}\ (F_0))$. Finally the description of each sub-cache fragment are:

$$
\begin{aligned}
F_0^U &: \pi_{a_1,a_5}\ (\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V)) \\
F_0^V &: \pi_{a_1,a_6}\ (\sigma_{v(a_1)\in\{2\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V)) \\
F_0^H &: \pi_{a_1,a_5,a_6}\ (\sigma_{v(a_1)\in\{1\}\wedge v(a_6)\in\{5,\cdots,\infty\}}\ (V))
\end{aligned}
$$

□

However, over-decomposition is the drawback of cache decomposition. When a coarse cache fragment is decomposed into a large number of finer sub-cache fragments, cache utilization will be reduced since a lot of space is used up to store the descriptions but not the data. The uncontrolled decomposition would cause the number of cache fragments to grow indefinitely. To address this problem, we have considered two possible approaches. One of them is to limit the degree of cache decomposition by setting a *decomposition threshold* which is an indication of cache granularity. Here, it is expressed as a ratio of the cache size. When its size is greater than the threshold, the cache fragment is said to be coarse; otherwise it is fine. Therefore if the size of a cache fragment is smaller than the threshold, decomposition of such fragment is prohibited.

By the way, we prevent a cache fragment from getting smaller and smaller by

setting decomposition threshold but this threshold has no effect in controlling the size of incoming supplementary query results. To cater for this issue, we have considered the second approach by combining several smaller cache fragments into one larger cache fragment. This mechanism is called *cache fusion*. In our implementation, when there is a query which reuses certain number of finer cache fragments and supplementary query results, the final query result instead of the finer fragments are maintained in the cache. For replacement score assignment, a new cache fragment will be assigned a new score. Applying both cache decomposition and cache fusion in cache accommodation is termed *cache re-organization*.

5.4 Chapter Summary

In this chapter, we discussed how to reuse a cache fragment to answer a query if there exists an intersection between them. To extract the required data from a cache fragment, a probe query is defined. The missing data is represented by a supplementary query. When other cache fragments are available, the supplementary query could be further transformed.

When admitting a new query result, certain cache fragments need to be replaced. Owing to a variable cache granularity, cache replacement is not so straightforward as one-to-one substitution. In addition to precisely maintaining frequently accessed data in a cache, we propose two management techniques, namely, cache decomposition and cache fusion. Cache decomposition divides a large cache fragment into smaller ones that contain data of same access frequency. However, it will introduce a problem of over-decomposition. On one hand we prevent a cache fragment from being decomposed into a number of smaller and smaller pieces. On the other hand, we use cache fusion to combine certain number of smaller cache fragments to a large one to improve the query transformation efficiency and reduce the storage overhead of

maintaining semantic information. With respect to the two techniques, we discussed the replacement score assignment schemes to mark the scores.

Chapter 6

Prototype Implementation

6.1 Development Methodology

The MoWS is composed of three main classes of entities : mobile clients, base stations and database servers. Each of them is equipped with its individual separate software module, namely, client subsystem, warehouse subsystem and server subsystem. They are all developed using Microsoft Visual C++ 5.0. Depicted in Figure 6.1 is the prototype architecture.

A client subsystem has three components: a query processor, a query transformation logic and a cache manager. Every query initiated from user applications is first input to the query processor. The query processor then processes the query with either the warehouse subsystem in the connection mode or the query transformation logic in the disconnection mode. The query transformation logic is to transform a query reusing cache fragments. The cache manager is responsible to manipulate the cache storage for keeping the cache fragments as well as their descriptions and access control information. It implements both cache decomposition and cache fusion. Linkage between a client subsystem and a warehouse subsystem is through Digital RoamAbout Wireless LAN and the communication protocol used is

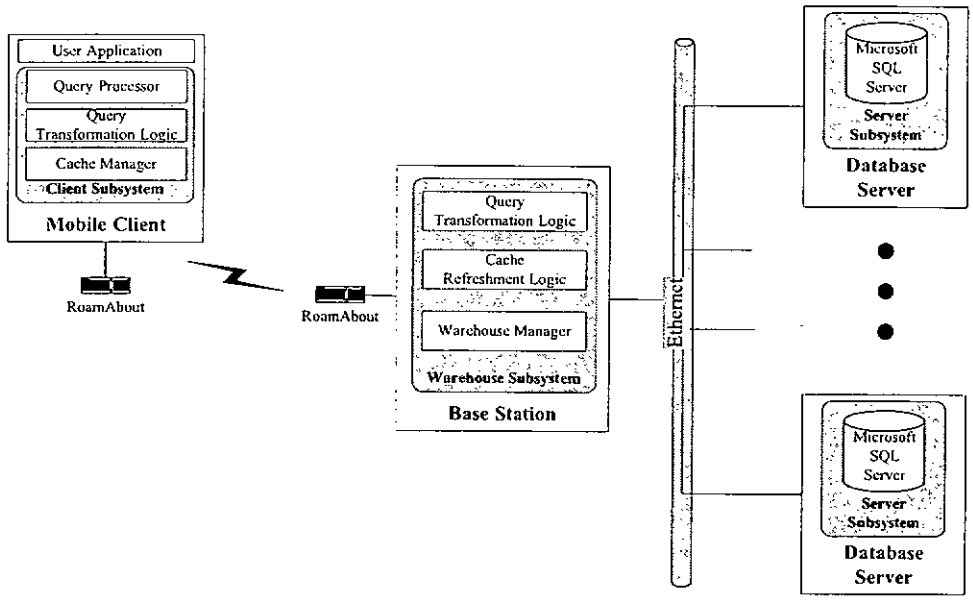


Figure 6.1: Prototype Architecture

TCP/IP. To reduce the communication and storage overhead of a query or a cache fragment description, we propose a representation to stand for a query, the pattern of the representation is called *bit-stream query representation* and it will be discussed in Section 6.2. Supported by the bit-stream representation, query transformation can be efficiently performed. We will discuss the mechanism in Section 6.3 and the implementation detail of storage management of the client subsystem in Section 6.4.

In a warehouse subsystem, there are a query transformation logic, a cache refreshment logic and a warehouse manager. The warehouse database storage used is Microsoft SQL Server 6.5. The implementation of the query transformation logic is same as that in the client subsystem. The cache refreshment logic is used to determine the changes to the client cache. Finally the warehouse manager is to keep the materialized view updated. It implements our WAVE algorithm.

At last, a server subsystem is developed to implement our WAVE algorithm and contains one database management system. The connection between a warehouse subsystem and a server subsystem is through TCP/IP communication protocol over

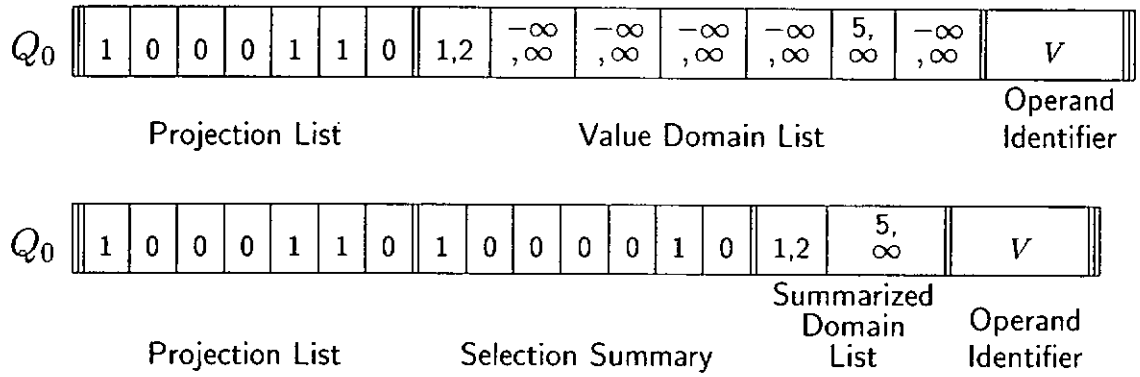
an Ethernet.

6.2 Bit-Stream Query Representation

In our prototype implementation, a query is represented in a form called *bit-stream query representation.* It has two representation formats: normal form and compact form. The normal form is mainly composed of three parts: a projection list, and a value range list and an operand view identifier. For a schema of the operand view containing n attributes, the projection list is associated with a binary list of n bits long. The on-mode of i^{th} bit indicates that the query requires the i^{th} attribute to become projected. The value domain list is a list of n min-max value pairs. Each min-max value pair is composed of two integers, bounding the set of elements selected within the whole sub-domain. This is based on an assumption that all attribute values are ordinal or they could be ordered. Finally, the operand identifier refers to the materialized view in the warehouse.

To show the format of the representation, we consider query, Q_0, defined in Example # 5.1. It is defined as a projection on a_1, a_5 and a_6 and restriction on the value range of a_1 from 1 to 2 and that of a_6 from 5 to ∞. At the end, the operand identifier records the materialized view, V. Furthermore, shortening the bit-stream query representation can be done by replacing a whole domain list with a summarized domain list. Very often the domain list might be lengthy but there might be no constraint on a particular set of attributes in a typical query, and a typical domain list might contain some default min-max pairs, i.e., unconstrained domains. A selection summary is introduced to indicate precisely which attribute has a constrained domain, i.e., non-default min-max pairs. The selection summary has the same size of the projection list. If the i^{th} attribute has a non-default min-max pair, the i^{th} bit of the selection summary will be turned on; otherwise, it is off. The domain list thus

could only contain the min-max pairs for those non-default domains. This produces a compact form. The normal form and compact form of Q_0 are inter-changeable and they are shown in Figure 6.2.



Normal Form (Above), Compact Form (Below).

Figure 6.2: Bit-Stream Query Representation of Query Q_0

6.3 Query Transformation Using Bit-Stream Query Representation

The bit-stream query representation is not only compact in size but also is very helpful in query transformation. In this section, we will present the way to use it in query reasoning. First, with the normal form of bit-stream representation, the process of determining if a cache fragment is reusable to a query, i.e., examining their intersection, could be simplified into two simple steps: performing a bit-wise **AND** operation on the two projection lists, followed by merging the range of the value domain lists. The merging can be done by identifying the least upper bound and the greatest lower bound of each min-max value pair of both lists. Then, once the non-empty intersection is determined, the probe and supplementary queries can be defined in the similar way. The resultant transformed queries of Q_5 (Example 5.6)

reusing F_0 are shown in Figure 6.3. The results of Q_0 is cached as cache fragment F_0. To decide if F_0 is reusable to Q_5, intersection $Q_5 \cap F_0$ needs to be determined. This is achieved by performing a bit-wise **AND** operation on the two projection lists as well as merging the ranges of the two value domain lists. As shown in Figure 6.3, the intersection between F_0 and Q_5 is non-empty.

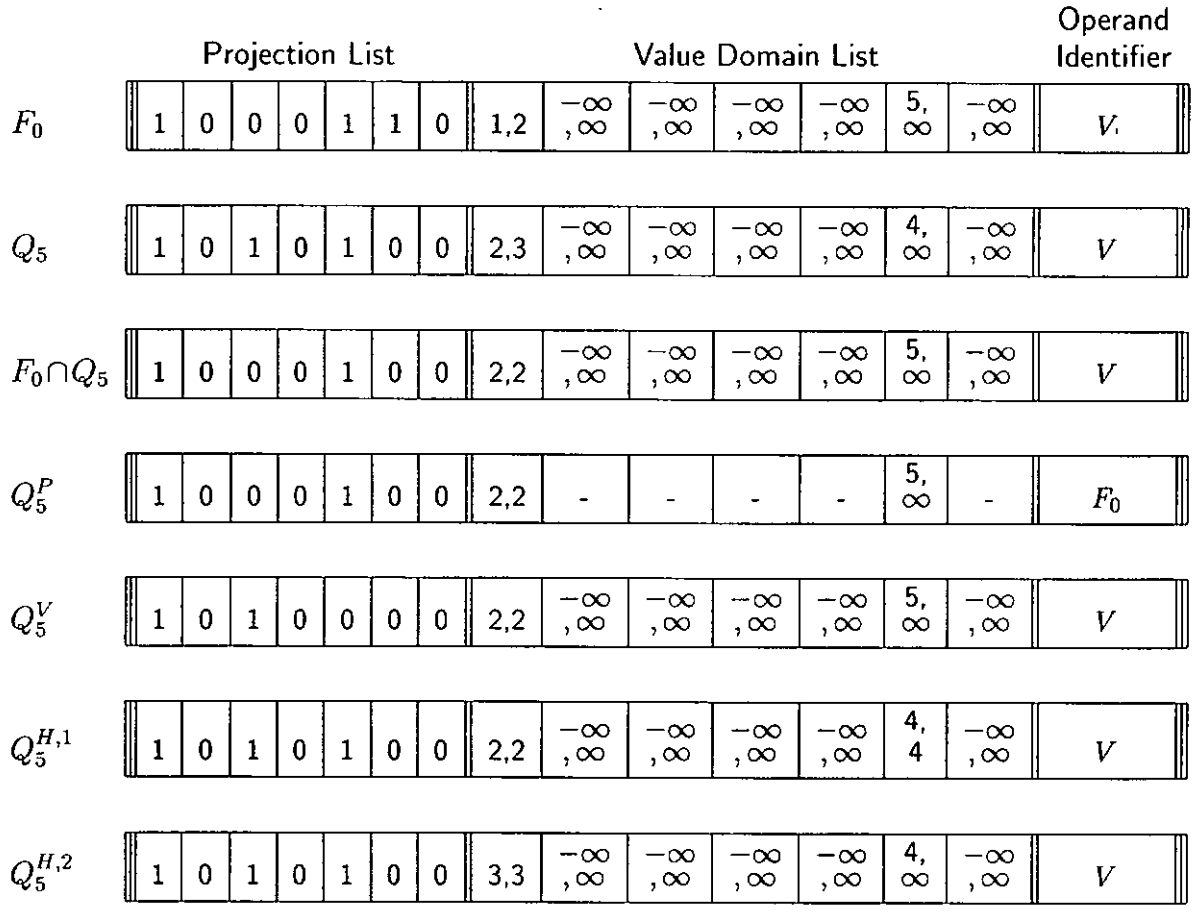


Figure 6.3: Query Transformation using Bit-Stream Query Representation

The probe query, Q_5^P, is formulated by replacing the operand identifier of the original query with the cache fragment. If the i^{th} bit of the projection list from the operand cache fragment F_0 is turned *off*, removing the i^{th} min-max value pair from the original query Q_5. For the vertical supplementary query, Q_5^V, its projection list is

the difference of the projection lists, i.e., $A_{Q_5} - A_{F_0}$, plus the key attribute(s), while its domain list is equal to that of the intersection.

For each horizontal supplementary sub-query, $Q_5^{H,i}$, its projection list is the same as the original query, i.e., A_{Q_5}, while its domain list is determined as follows. The j^{th} $(j < i)$ min-max value pair is equal to that of the intersection. The i^{th} min-max value pair is determined from the intersection of the i^{th} min-max value pair of F_0 and the negation of i^{th} min-max value pair of Q_5. The negation of a min-max pair, in our implementation, yields two pairs, they are ranged from the default domain minimum to the greatest lower bound - 1 and from the least upper bound + 1 to the default domain maximum. The j^{th} $(j > i)$ min-max value pair is just the original j^{th} min-max value pair of Q_5. The idea is depicted in Figure 6.3. Note that those supplementary queries which condition are not satisfiable are not shown in the figure.

6.4 Cache Storage Management

In the earlier phase of our implementation, we had developed our prototype using existing database management software packages like Microsoft Access as our cache storage management. However, the overall performance is not satisfactory as we expected in the preliminary feasibility studies. Although it offers a comprehensive application programming interface, the underlying database system has involved too much necessary overhead in computation and storage in the run time. It is all due to the excessive system resources used in their internal database activities that we do not require in the simple buffer management for caching data.

Instead, we implemented our own buffering system to maintain cache fragments and cache control information. First, each cache fragment is maintained in an individual binary file. A cache fragment is stored as a sequence of tuples, each of which is a collection of attributes. The space required in storing a cache fragment in the

buffer should be $\sum_{\forall t \in \text{tuples of cache fragment}} \sum_{\forall a \in \text{attributes of } t}$ (size of a). Second, cache fragment information and its control information are maintained as an entry in a specific file named 'CacheSummary'. The structure of a typical entry is shown in Figure 6.4.

Name	Data Type	Size	Description
CacheID	Long	4 bytes	Cache fragment identifier (unique)
SummarySize	Short	2 bytes	Size of the CacheSummary entry
Cache fragment description in a compact Form:			
ProjectionList	Bytes	8 bytes	Projection list
SelectionSummary	Bytes	8 bytes	Selection summary
Summarized value domain list (repeated for n Min-Max Pairs):			
MinValue	Long	4 bytes	Minimum value of a min-max pair
MaxValue	Long	4 bytes	Maximum Value of a min-max pair
OperandRelation	Long	4 bytes	Operand relation identifier
NumAttributes	Short	2 bytes	Number of projection attributes
NumTuples	Long	4 bytes	Number of tuples
FragmentSize	Long	4 bytes	Cache fragment size
UpdateTime	Long	4 bytes	Database server update time
LastRefTime	Long	4 bytes	Last cache fragment reference time
ReplacementScore	Long	4 bytes	Replacement score

Figure 6.4: Structure of a CacheSummary Entry

Each entry is composed of a cache identifier, a cache fragment description, control information such as fragment size, last update time, number of tuples, number of attributes, last reference time and replacement score. Since the cache fragment description is in a compact size that depends on the selection summary, the size of

each entity is not fixed. To facilitates file fetching in CacheSummary, each entry has been associated with a 'SummarySize' stating the entry length. It can be, in turn, interpreted as the offset to next cache fragment summary from the start of current entry. Collectively, a single entry requires 56 bytes for a cache fragment derived from a single-attribute selection query.

Chapter 7

Performance Evaluation

To justify the suitability of MoWS, we conduct performance evaluation based on the prototype, and quantitative analysis. The following 2 sections present the experiment result of our proposed semantic caching scheme, and the quantitative analysis of our pull-based view update mechanism.

7.1 Performance Study on Semantic Query Caching

7.1.1 Experiment Setup

In evaluating the performance of semantic query caching scheme, we employ one mobile client and one warehouse connected through a wireless channel of typical 19.2kbps. The hardware configuration of these entities is shown in Table 7.1.

We measure the performance of the caching scheme using a modified Wisconsin benchmark [11]. The original benchmark offers a relation schema containing 13 4-byte-long integers and 3 52-byte-long character strings. In order to make sure that all queries will have results of the same size where there are fixed number of tuples selected and fixed number of attributes projected, the benchmark schema is modified with 52 4-byte-long integers only. The number of tuples selected and the number

of attributes projected consitute the *selectivity* and *projectivity* respectively. Now, the warehouse contains one materialized view which is a pair of benchmark relations, **WIS** and a corresponding history relation, $\tilde{\mathbf{WIS}}$. The relation **WIS** contains 10,000 tuples. We intend to keep the warehouse and mobile client storage small, in order to make it feasible to run a large number of experiments. The schema of **WIS** contains 52 integers of 4-byte long plus a creation time tag of 4-byte. Each tuple of **WIS** has a size of 212 bytes. The schema of $\tilde{\mathbf{WIS}}$ contains 52 integers of 4-bytes long plus a creation time tag and a deletion time tag, both of them are 4-byte long. The size of each tuple of $\tilde{\mathbf{WIS}}$ is 216 bytes. Other benchmark properties are kept unchanged, i.e., *Unique2* is perfectly cluster-indexed; *Unique1* is the key attribute and is indexed but non-clustered. Altering selectivity and projectivity varies the query size in the experiments.

Item	Mobile Client	Warehouse
CPU Speed	Intel Pentium 133MHz	Intel Pentium-II 450MHz
Memory: RAM	32 MBytes	128 MBytes
Harddisk	3.2GBytes	9.1GBytes

Table 7.1: Hardware Configuration

A stream of client queries is generated according to 80/20 rule such that 80% of queries will access 20% of the tuples and 20% of the attributes, while 20% of the query will access the remaining 80% of the tuples and 80% of the attributes. The two sets of data items are termed *hot region* and *cold region* [1] respectively. We follow the skewed hot access pattern and the changing skewed hot access pattern. We model the skewed hot access pattern by fixing the hot region for all queries while we model the

[1] Although the center-point of a hot query is mostly confined within the hot region [14], certain portion of a large query might touch the cold region. This renders a large number of hits in that region adjacent to the hot region

changing skewed hot access pattern by progressively shifting the hot region towards other set of tuples along the stream of queries. We have considered three replacement schemes in our experiment, namely LRU (Least Recently Used), MRU (Most Recently Used), LSR (Largest Size Remains) and considered the case where no cache is used NoCaching (no query result cached). Each experiment is carried out for several times by supplying a query stream consisting of 50 warm-up queries followed by 300 queries based on which the result is collected.

The selection criteria of each query will involve the attribute *Unique2* only, while the set of projection attributes contains p attributes, including the key and selection attribute; they are *Unique1* and *Unique2*. The projectivity of a query ranges from 8, 12 to 16, while the selectivity ranges from 1000, 1500 to 2000 tuples. The cache storage size at a client is fixed at 128K. For the largest query with a selectivity of 2000 tuples and a projectivity of 16 attributes, nearly 98% storage space is used to store the result. To cater for a dynamic cache granularity, techniques like cache decomposition and cache fusion discussed in Chapter 5 are used. In the following experiments, we vary the decomposition threshold from 100%, 10%, 1% and 0.1%. Also we fuse small cache fragments into a large one when a certain number of small cache fragments are reused. That number is called *fusion threshold*. In the experiment, it is set to 1, 5 and 10.

To be conservative, we only consider all database updates to be modifications in our experiment. When a set of tuples in the warehouse are being modified, the values of key attributes and selection attributes are kept unchanged such that this will not affect the query selectivity, and the database size can be maintained invariant. Also we are concerned that updates are usually infrequent and each time only a small number of tuples are modified. To investigate the effect of database updates in MoWS, we specify a *query/update ratio* which is a ratio between the number of queries and the number of updates. For instance, a ratio of 10:1 means that there would be on

average 1 update for every 10 queries. To conclude this subsection, we summarize all parameters in Table 7.2.

Parameter	Values
Size of a Database Relation	10000
Cache Storage Size	128k bytes
Number of Projection Attributes (Projectivity)	8, 12, 16
Number of Tuples Selected (Selectivity)	1000, 1500, 2000
Number of Tuples Updated (U-Selectivity)	50, 100
Query/Update Ratio	10:0, 10:1, 10:2, 10:3
Access Pattern	CSH (Changing Skewed Hot), SH (Skewed Hot)
Size of Hot Region	20% of tuples and 20% of attributes
Decomposition Threshold	100%, 10%, 1%, 0.1%
Fusion Threshold	1, 10, 100
Replacement Scheme	LRU, MRU, LSR, NoCaching
Wireless Network Bandwidth	19.2Kbps

Table 7.2: Parameters and Values

7.1.2 Performance Metrics

To quantify the performance, we mainly focus on four performance metrics. The first and the most important one is the *elapsed time*, that is, the duration taken from the time a query is initiated to the time a query result is completely obtained and the cache are completely updated. The second metric is the *bandwidth consumption* that measures how much the bandwidth of the wireless channel is used to process a query. The third one is the *cache hit ratio* that is targeted to explore how much of a query can be answered by the mobile client. The last metric is the *answerability rate*

which is used to indicate how many queries are considered locally answerable during disconnection as a certain expected cache hit is fulfilled, that is supposing a minimum required hit ratio of h, we measure the number of queries which have a hit ratio of at least h as the answerability rate.

7.1.3 Experiment Result

Experiment #1. Study on different replacement schemes

In the first experiment, we investigate the benefit of semantic query caching for no database update, i.e., query/update ratio is set to 10:0. We differentiate the performance among two classes of queries, namely SH (Skewed Hot) and CSH (Changing Skewed Hot) with four different replacement schemes, namely, NoCaching, LRU, MRU and LSR. The results are presented in Figure 7.1 and Figure 7.2. The effect of cache decomposition is not studied in this experiment. Recall that decomposing a cache fragment into sub-fragments is performed only when the size of a cache fragment is larger than the defined threshold. A decomposition threshold of 100% will prohibit decomposition from occurring. Therefore, the decomposition threshold is set to 100% here. The effect of the decomposition threshold will be studied in Experiment #2. Also, no cache fusion is studied here.

We measure three performance metrics, including *elapsed time*, *bandwidth consumption*, and *cache hit ratio*. The results are depicted in Figure 7.1 and Figure 7.2, each arranged as an array of graphs. The first row depicts the elapsed time, the second row the hit ratio and the third the bandwidth consumption. The different columns depicts the measurements of the metrics with different projectivity. In the figures, the elapsed time of semantic caching completely outperforms NoCaching irrespective of the query class and replacement scheme. It performs the worst because all query results should be loaded from the warehouse. So, a higher bandwidth consumption and zero cache hit is explainable. Regardless of replacement schemes, SH produces a

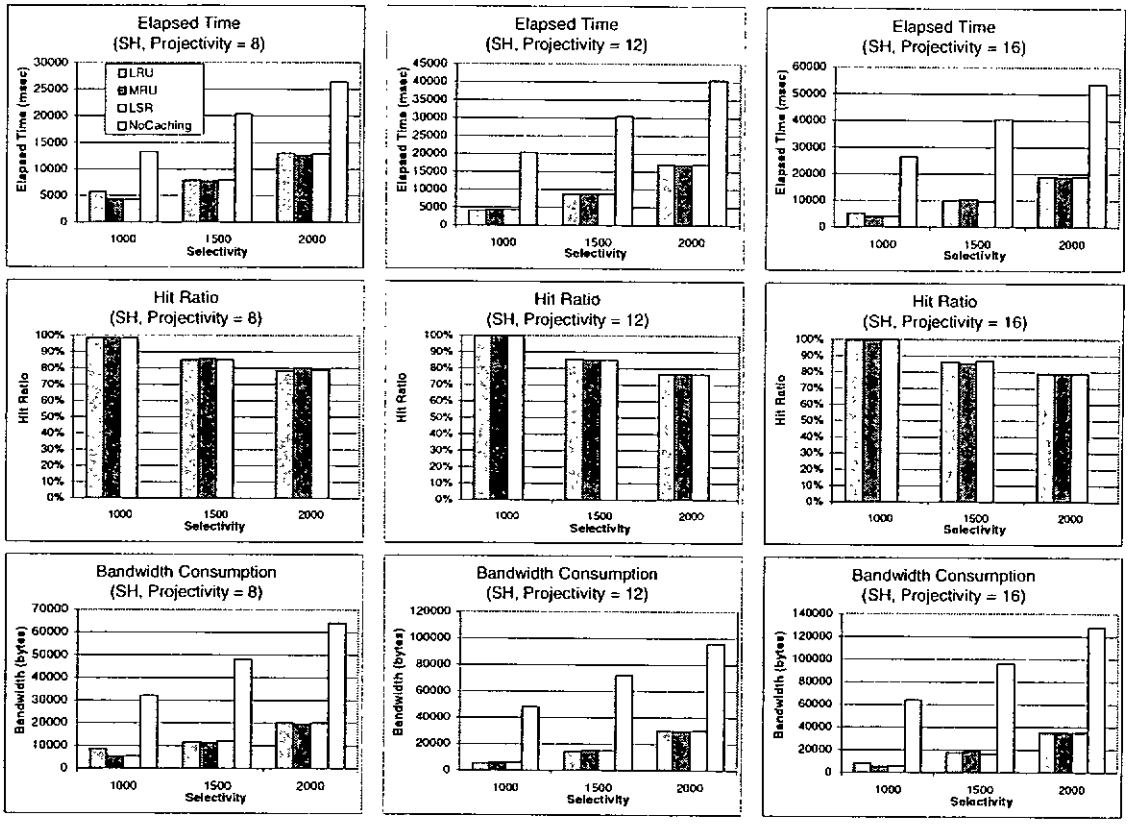


Figure 7.1: Experiment #1 Result (Skewed Hot)

better performance than CSH. The SH queries yield a shorter elapsed time since most of the result can be loaded from the cache that is reflected in cache hit ratio. Also less bandwidth consumption is obviously expected.

With respect to each access pattern, we discover that among three replacement schemes, LRU performs very well, next is LSR and the worst is MRU in all metrics for CSH. But for SH, MRU performs a little bit better, next are LSR and LRU, both performing more or less the same. Besides, we observe that as the query size increases for CSH, i.e., when selectivity increases for same projectivity, the cache hit ratio greatly drops. The main reason is that a coarse cache granularity prevents the cache from adapting to the changes in access patterns. Accommodation of new cache fragments

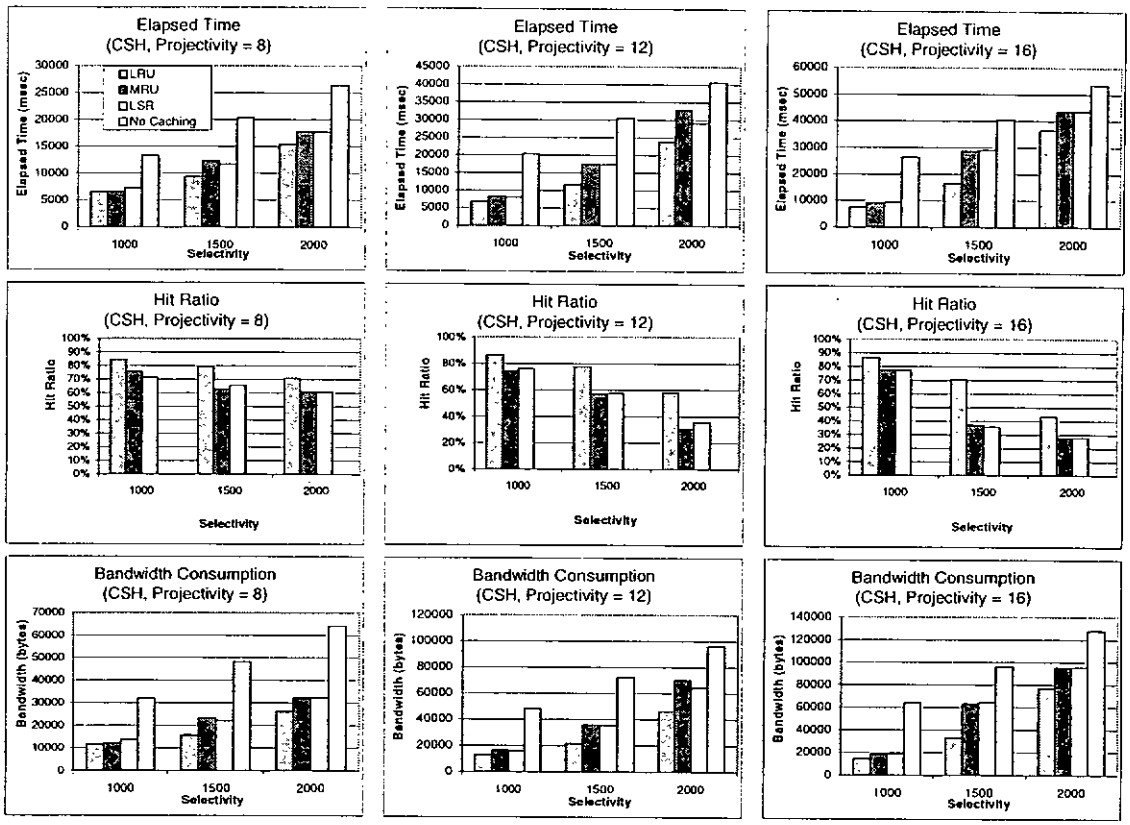


Figure 7.2: Experiment #1 Result(Changing Skewed Hot)

becomes less flexible, but for SH, no such phenomenon is observed.

Experiment #2. Effect of cache decompostion

In the second experiment, we study the effect of the cache decomposition only on the performance for no database update, i.e, query/update ratio is set to 10:0. We restrict the selectivity of queries to 2000 and the projectivity of queries to 16. From the results of Experiment #1, we are performing a worst-case study here. Notice that NoCaching does not have any effect with respect to our approach, so we do not show its result here.

We experiment with the decomposition threshold ranging from 100%, 10%, 1%

and 0.1% of the cache size. Notice that with a threshold set to 100%, no cache decomposition is performed. In this experiment, it is used as a reference to illustrate the effect of cache decomposition. Here, we only illustrate the elapsed time as well as the cache hit ratio, since users probably are more interested in these two metrics. The results are depicted in Figure 7.3 and Figure 7.4. Notice that we compare LRU and MRU since they perform better than LSR observed from Experiment #1.

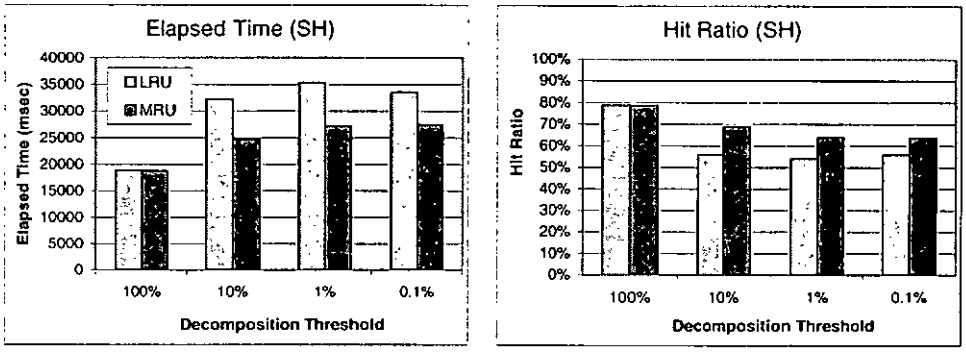


Figure 7.3: Experiment #2 Result (Skewed Hot)

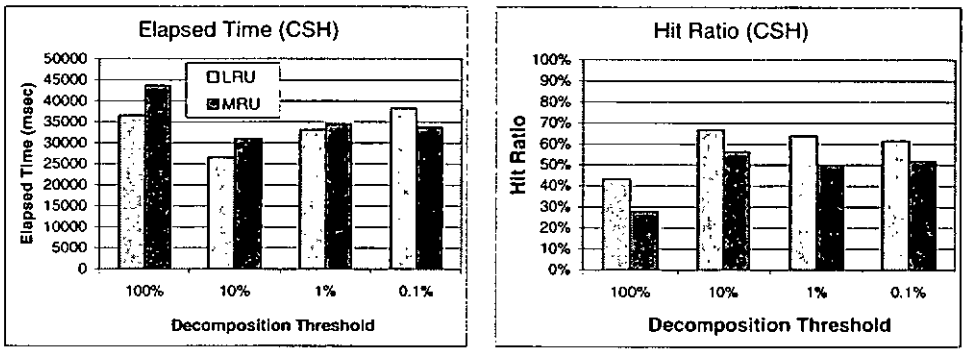


Figure 7.4: Experiment #2 Result (Changing Skewed Hot)

We observe from Figure 7.3 with for SH, the application of cache decomposition, longer elapsed time is resulted. The reason is that cache decomposition would produce smaller fragments. Enhancement of the cache adaptability does not benefit for such stable access pattern. Rather, it introduces the additional overhead in processing cache decomposition. Also from the figure, we can see that the hit ratio drops when

extra storage space is used to maintain the semantic information and the presence of duplicated key attributes.

However, from Figure 7.4, we observe that in general for CSH, LRU performs better than MRU for all decomposition thresholds. And the performance is improved by applying cache decomposition. The elapsed time is shorter and the cache hit ratio is higher when the threshold is at 10% and 1% (with a 10% threshold yielding the best performance). Further decomposition (threshold at 0.1%) will lead to a higher elapsed time and a lower cache hit. Indeed, cache decomposition will produce too many overly fine cache fragments, resulting in extra processing overhead in cache management, query transformation, result construction, extra storage overhead and transmission overhead. We discover that there are duplication of key attributes due to vertical decomposition of the cache at a smaller threshold.

Experiment #3. Study of self-answerability

We would like to study the answerability of queries as cache decomposition is employed in Experiment #2. In this experiment, we only test the CSH with varying decomposition thresholds since no decompostion produces the best performance for SH. The query selectivity and projectivity are fixed at 2000 and 16 respectively. Given a degree of self-answerability, h, we measure the number of queries which have a hit ratio over or equal to h. The degree is set to 50%+, 60%+, 70%+ and 80%+. The results are shown in Figure 7.5.

From the figure, we can observe that for LRU, when decomposition is disabled, i.e., with a threshold of 100%, only less than half (42%) of the queries can be answered with a hit ratio of at least 50%. This means that the cache content contains a lot of non-reusable data. In the figure, the best result is observed when the threshold is 10%. With a requirement that a partial result of over 70% should be found in the cache, almost 40% of the queries can be served. With a decomposition threshold of

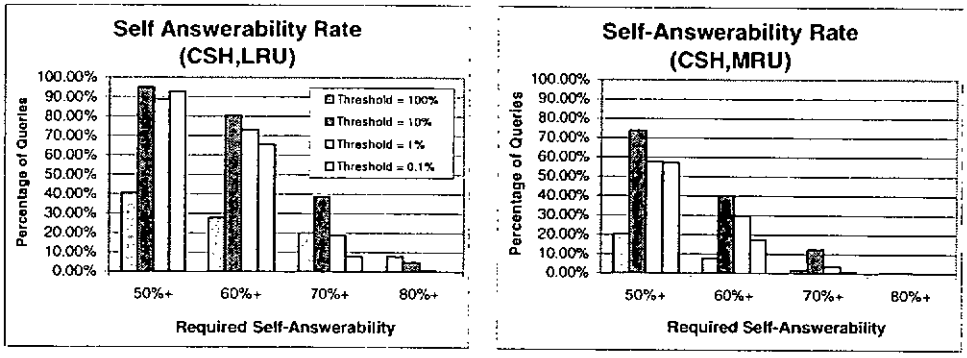


Figure 7.5: Experiment #3 Result

1% and 0.1%, more than 60% of the queries can be answered at a self-answerability of 60% or above but around 20% of the queries can be answered for self-answerability of 70%. In general, decomposition threshold set to 10% would increase self-answerability rate during disconnection, with more queries answerable at a higher designated cache hit ratio. For MRU, the same phenomenon could be observed and explained, but the overall performance is not so satisfactory as that for LRU.

Experiment #4. Effect of cache fusion

From the previous 2 experiments, we learned that the decomposition could enhance the performance of the cache in terms of all performance metrics. However, it introduces the problem of over-decomposition. To address this issue, we have discussed the cache fusion mechanism. When a query reuses certain number of small cache fragments, the final query result is admitted to the cache and replaces those reused fragments. We experiment the cache fusion with a *threshold* set to 1, 5 and 10. Also we restrict the experimental scope on CSH access with the query selectivity and projectivity fixed at 2000 and 16 respectively. The replacement scheme used is LRU. Also we vary the decomposition threshold, 10%, 1% and 0.1%. The result is shown in Figure 7.6.

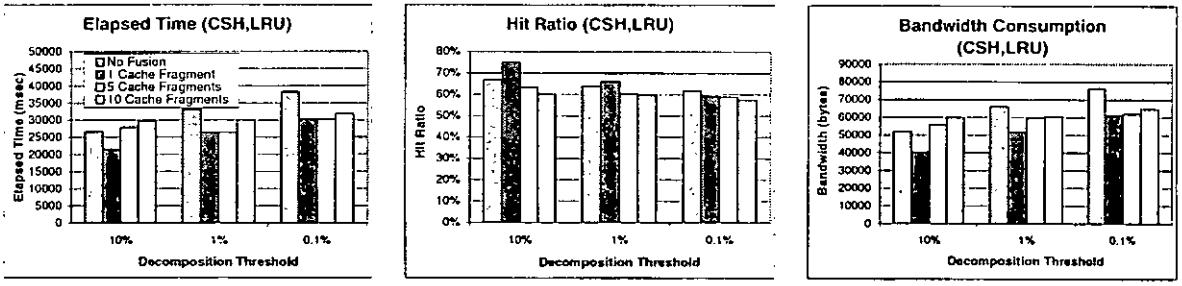


Figure 7.6: Experiment #4 Result

From the figures, we can see that using cache fusion for selected fusion threshold can yield a better performance compared with no cache fusion. With fusion threshold set to 1, the performance is the best in terms of the elapse time, the hit ratio and the bandwidth consumption. As the fusion threshold further increases to 5 and 10, the performance drops accordingly. The reason is that the larger fusion threshold would allow more number of smaller cache fragments to remain in the cache. Thus, the problem of over-decomposition does exist in such cases.

Experiment #5. Effect of database update

In the fifth experiment, we experiment the cache refreshment when a query is evaluated. In this experiment, we select three set of queries: the first is with selectivity of 1000 and projectivity of 8, the second is with selectivity of 1500 and projectivity of 12 and the third is with selectivity of 2000 and projectivity of 16. They represent three different classes of query sizes. Specific to the third one, we use cache decomposition with threshold set to 10% and cache fusion with threshold set to 1 in this experiment. This setting has been shown to yield the best performance previously. To study the effect of update in the source data, we vary the query/update ratio as well as the update selectivity. The query/update ratio is varied among 10:0, 10:1, 10:2, and 10:3, while the update selectivity is controlled at 50 and 100. Notice that when query/update ratio is set to 10:0, there is no database update. We contrast

the results against that of NoCaching to explore the suitability of our caching scheme for different update rates. In this experiment, we examine the elapsed time and bandwidth consumption. We do not show the cache hit ratio which is presented in Experiment #1.

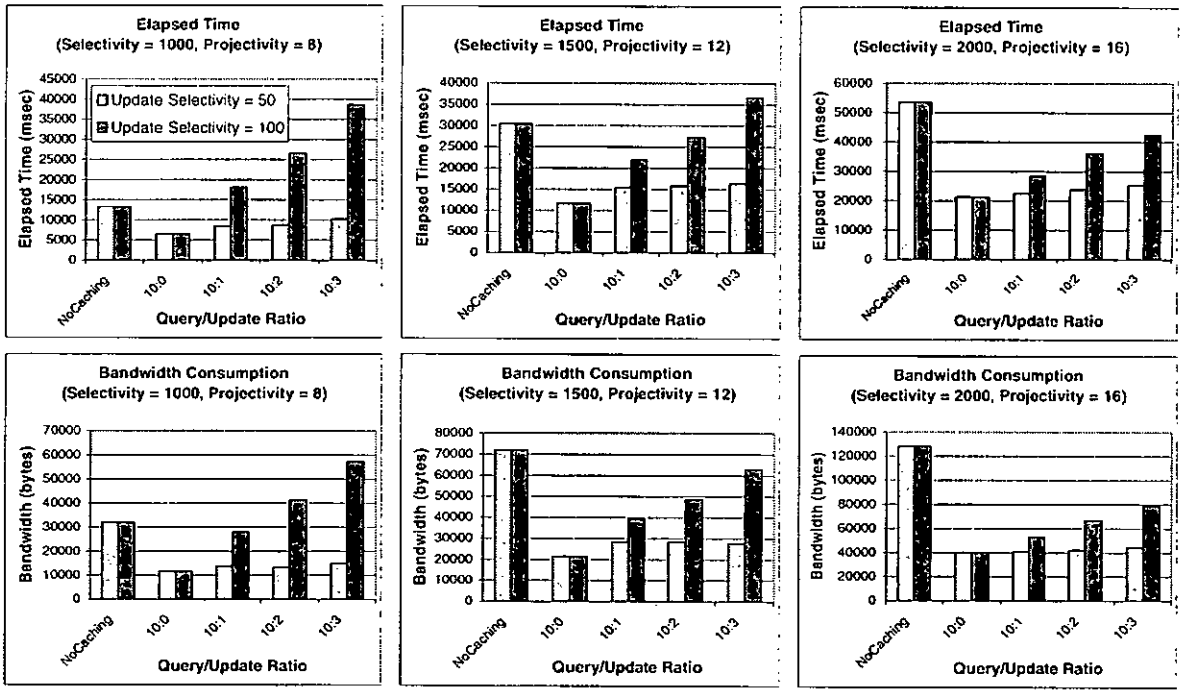


Figure 7.7: Experiment #5 Result

The result is depicted in Figure 7.7. From the figure, we observe that in general, for the update selectivity is 50, as a query/update ratio rises, the elapsed time increases slightly. However, for update selectivity is 100 and same rising of query/update ratio would cause a dramatic increase in the elapsed time. The reason is that when the update selectivity is small, the possibility of having a stale cache fragment will be low as expected. The overhead to maintain cache consistency will be less. However, when the update selectivity increases, the cached data will probably become invalidated quickly and it will suffer from more cache update overhead.

Also we observe from the figure that when the query size is small (selectivity =

1000, projectivity = 8), with high update selectivity and high query/update ratio, NoCaching produces a relatively good performance in terms of both elapsed time and bandwidth consumption. It is because NoCaching does not spend any overhead in maintaining cache consistency. It is not worth to mainatin the cache as it is always not valid. From the observation of the results, we can conclude that our scheme is suitable in the environment where update seldom occurs and the database usually undergoes relatively small changes. That matches the feature in the mobile environment that the update is infrequent and the degree of change in database is very small [33].

7.2 Performance Study on Pull-based View Update Mechanism

In this section, we will evaluate the performance of WAVE (optimized WAVE) and compare it with that of some previous approaches, in the context of pull-based view update environment, based on quantitative analysis. In our evaluation, we focus on the amount of data transferred (transmission overhead) between the database server and the warehouse as well as the storage overhead. For the ease of presentation, the projection and selection are assumed to be performed when all the tuples are received at the warehouse before sending to the mobile client. The evaluation parameters and their values are shown in Table 7.3.

The join factor, J represents the expected number of tuples of a relation that will be joined with one tuple in preceding relation. The degree of update is expressed as ρ. Each update is either an insertion, a deletion, or a modification. The degree of modification is expressed in term of μ, which is the operation out of the number of updates that modify a tuple. We assume that each update operates on a distinct tuple.

Parameter	Symbol	Value
Size of a database relation	C	10000
Size of a tuple	B_r	208 bytes
Size of a "count" attribute	B_c	1 bytes
Size of a time tag	B_t	4bytes
Size of a join attribute	B_j	4bytes
Size of a sub-view description	B_ξ	36 bytes
Number of database servers	m	1,2,3,$\cdots$,20
Join factor	J	0.5, 1, 1.5, 2, 2.5
Degree of update	ρ	0.5%, 1%, 1.5%, 2%, 2.5%
Degree of modification	μ	20%,40%,60%,80%,100%

Table 7.3: Parameters and Values

Evaluation #1. Study of WAVE performance varying the number of database servers and join factor

In the first evaluation study, we would like to compare the transmission overhead, in terms of number of bytes transferred in WAVE, with that in the base case, recomputation. The recomputation just involves issuing the view definition for evaluation again and the new view replaces the old view content. Intuitively, WAVE should perform better than recomputation since it involves only incremental computation while in recomputation, the whole view is re-evaluated and transmitted for each update request from the warehouse. With 100% degree of modification, all tuples updated are modified. In recomputation, in response to a request from a warehouse, each database server sends qualified tuples to the warehouse. With m database servers, the amount of data transferred is $mCB_r + mB_\xi$.

To compute the transmission overhead in WAVE, we divide the traffic into two

types, uplink traffic from the warehouse to the database servers and downlink traffic from the servers to the warehouse. We first calculate the number of tuples sent in the downlink from the database server to the warehouse. To determine the changes of a view, the warehouse will need to send its last update time and a sub-view description to a database server. This constitutes a total of $B_t + B_\xi$ bytes. With the degree of update ρ, the number of tuples modified at a database server since the cache was last updated is ρC. Assuming that all tuples being modified are distinct, the size of the changes in the relation is $2\rho C$ since there are ρC tuples deleted and ρC tuples created.

Consider the situation when database server i receives a request from a warehouse. It will send $2\rho C$ tuples to the warehouse. After projecting the $2\rho C$ tuples on the join attributes, the projected tuples will be forwarded to server $i-1$ and $i+1$ in two directions. Server $i-1$ will perform a semi-join operation between the projected tuples and its relation, resulting in $2\rho CJ$ tuples which will be sent back to the warehouse. Similarly, server $i+1$ also sends back $2\rho CJ$ tuples. This is repeated until server 1 and m are reached. The number of tuples transmitted over the downlink, D_i for server i is therefore:

$$\begin{aligned} D_i &= 2\rho C + \textstyle\sum_{l=1}^{i-1} 2\rho C J^l + \sum_{r=i+1}^{m} 2\rho C J^{r-i} \\ &= 2\rho C(1 + \textstyle\sum_{l=1}^{i-1} J^l + \sum_{r=i+1}^{m} J^{r-i}). \end{aligned}$$

Since the warehouse will send an update request to all m database servers, the total number of tuples transmitted over the downlink will be:

$$D = \sum_{i=1}^{m} D_i = \begin{cases} \frac{2\rho C}{(J-1)^2}[2J(J^m - 1) - m(J^2 - 1)] & \text{if } J \neq 1 \\ 2\rho C m^2 & \text{otherwise.} \end{cases}$$

Each tuple has a size of $B_r + B_c$ bytes. This results in a total of $D(B_r + B_c)$ bytes. The first reply from each database server will contain a time of B_t bytes. Since there are m such messages, this constitutes a total of mB_t bytes.

Now, consider the data being transferred over the uplink from the warehouse to the database servers. Consider again a request to a database server i. It will receive

$2\rho C$ tuples from server i. After projecting on the join attributes, the $2\rho C$ entries are sent to server $i-1$ and $i+1$. Similarly, $2\rho CJ$ projected tuples will be sent to server $i-2$ and $i+2$ and so on. The number of tuples sent is therefore:

$$\begin{aligned} U_i &= \sum_{l=1}^{i-1} 2\rho C J^{l-1} + \sum_{r=i+1}^{m} 2\rho C J^{r-i-1} \\ &= 2\rho C(\sum_{l=1}^{i-1} J^{l-1} + \sum_{r=i+1}^{m} J^{r-i-1}). \end{aligned}$$

The total number of tuples sent over the m sources will be:

$$U = \sum_{i=1}^{m} U_i = \begin{cases} \frac{4\rho C}{(J-1)^2}[(J^m-1) - m(J-1)] & \text{if } J \neq 1 \\ 2\rho C m(m-1) & \textit{otherwise.} \end{cases}$$

Since each join attribute has a size of B_j bytes, this results in UB_j bytes. In addition to the join attributes, each message to a database server contains a time and the sub-view description. There are m first round message and $2(m-1)$ subsequent messages for two propagation directions. So totally there are $(3m-2)(B_\xi + B_t)$ bytes . Summing all these up, the total amount of data transferred is:

$$\begin{cases} \frac{2\rho C(B_r+B_c)}{(J-1)^2}[2J(J^m-1) - m(J^2-1)] + \\ \qquad \frac{4\rho C B_j}{(J-1)^2}[(J^m-1) - m(J-1)] + \\ \qquad (4m-2)B_t + (3m-2)B_\xi & \text{if } J \neq 1 \\ 2\rho C m^2(B_r + B_c) + 2\rho C m(m-1)B_j + \\ \qquad (4m-2)B_t + (3m-2)B_\xi & \text{otherwise.} \end{cases}$$

We will investigate the effect of three parameters on the number of bytes transferred, namely, m, ρ and J. The results are shown in Figure 7.8 and Figure 7.9. Figure 7.8 shows the transmission overhead against the number of database servers, m. Figure 7.9 depicts the number of bytes transferred against the join factor, J. For comparison purpose, we also depict the transmission overhead for recomputation.

In Figure 7.8, J is maintained at 1, as a point of reference. We study the performance of WAVE on different rates of relation update, with ρ ranging from 0.5 to 2.5 with 0.5% interval. We also vary m from 1 to 20. The number of bytes transferred

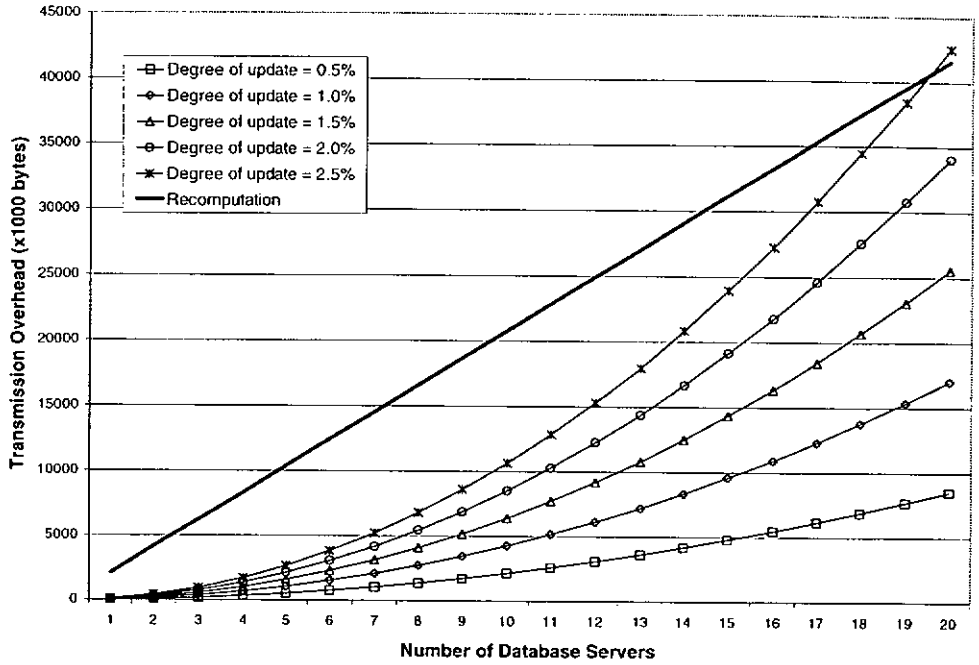


Figure 7.8: Transmission Overhead against Number of Database Servers

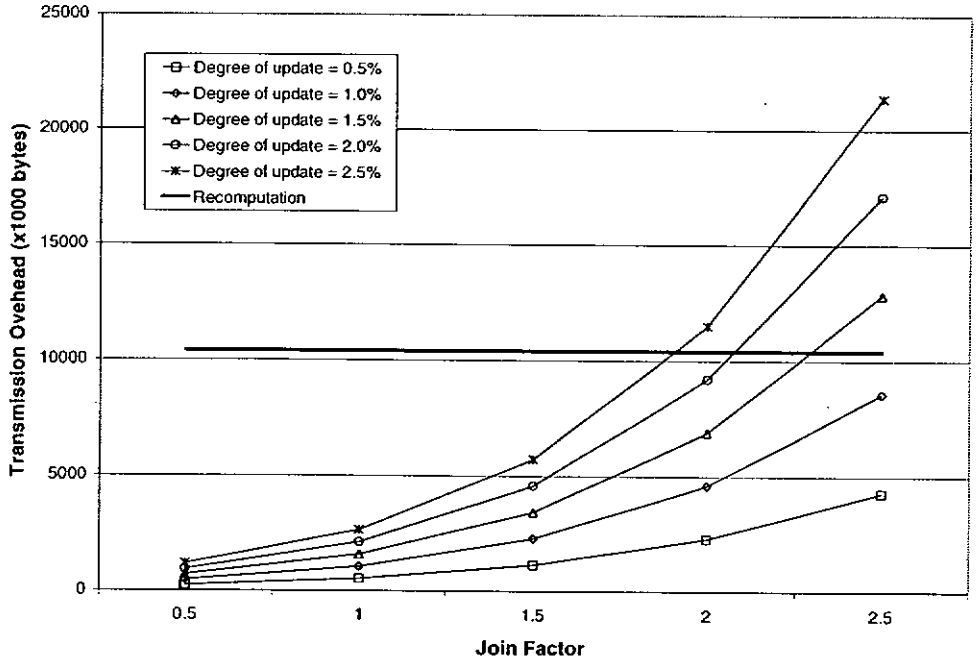


Figure 7.9: Transmission Overhead against Join Factor

is found to be directly proportional to the degree of update. With respect to m, the number of bytes transferred rises gradually as m increases in recomputation, but the growth is more rapid in WAVE. However, WAVE performs slightly worse than recomputation only when there are more than 19 data sources with a degree of update over 2.5%.

In Figure 7.9, m is fixed at 5. We compute the performance of WAVE with respect to different join factors from 0.5% to 2.5%. Since the same relations are transferred from the database servers to the base station, the number of bytes transferred in recomputation is independent of the join factor for the fixed number of servers. In contrast, the number of bytes transferred increases exponentially in WAVE. It can be observed that WAVE performs better than recomputation when the join factor is less than 2. As a result, an incremental cache refreshment would not be appropriate for relations and queries with a large join factor. However, for practical database applications, most of queries will only lead to a small join factor, especially when join operation is performed via foreign keys.

Evaluation #2. Comparing WAVE with T-Strobe in terms of transmission overhead

In the second evaluation, we would like to compare the transmisson overhead in WAVE with other incremental update mechanisms. Since apart from this work, we are not ware of any similar algorithm specifically designed for the pull-based view update environment, we decide to compare WAVE with T-Strobe [84] because T-Strobe is shown to be space and time efficient. Notice that T-Strobe and WAVE are designed for different environments, and thus, the comparison could only be used as a reference.

We fix m to 5 and compare the transmission overhead in terms of the number of bytes transferred in T-Strobe with WAVE when ρ ranges from 0.5% to 2.5%. Again, we assume a 100% degree of modification. We also vary J from 0.5 to 2.5. The results

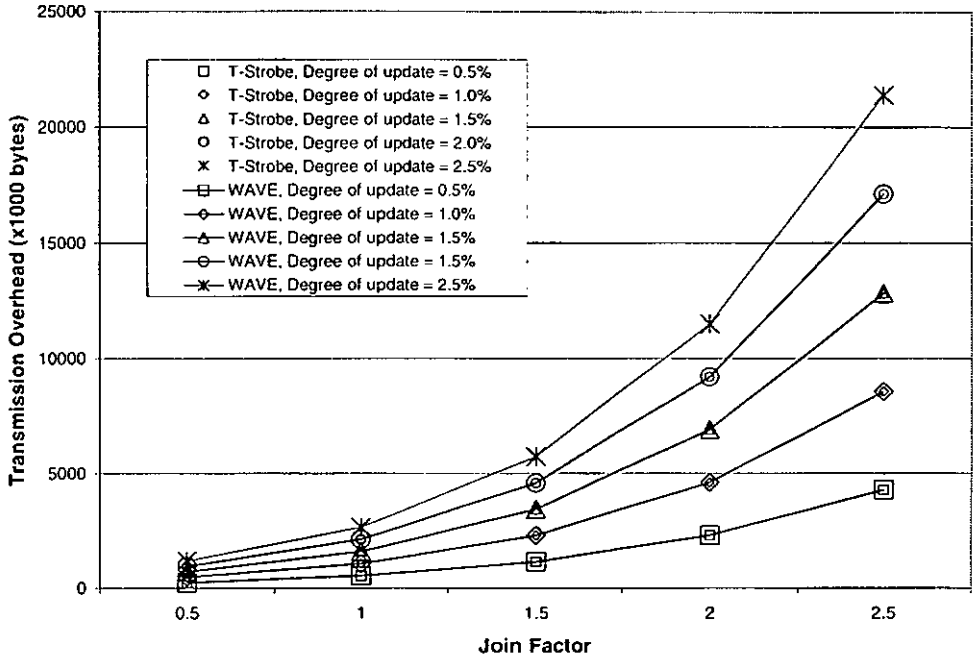


Figure 7.10: Comparison of WAVE and T-Strobe

are shown in Figure 7.10. We could observe that the number of bytes transferred in the two approaches are almost identical with only non-observable differences.

Evaluation #3. Comparing WAVE with other algorithms in terms of storage overhead

In our last evaluation, we would like to quantify the storage overhead imposed by WAVE. We specifically compare the storage overhead of WAVE with the differential file approach (differential) proposed in [67] since both approaches are based on logging update information applied on a relation and will require extra storage overhead. We also compare with T-Strobe because T-Strobe is quite space efficient.

For differential, each entry in the log file includes an updated tuple, an operation code and two timestamps. In addition to our notations in the previous evaluations, we denote the size of an operation code as B_{op} of 1 byte. The size of each entry in the log file is thus $B_{log} = B_r + B_{op} + 2B_t$. The size of a log file depends on the number

of entries in the log file which in turn, depends on the number of update operations. Assuming that we have the same number of insert and delete operations, the number of insert and delete operations is therefore $\frac{1-\mu}{2}\rho C$. Since each modification operation will constitute two entries in the log file, we compute the total number of entries in the log file, $C_{log} = 2\rho C\mu + \frac{1-\mu}{2}\rho C + \frac{1-\mu}{2}\rho C = (1+\mu)\rho C$. The size of the log file will therefore be $C_{log}B_{log} = (1+\mu)(B_r + B_{op} + 2B_t)\rho C$.

T-Strobe requires storage for holding all the updated tuples within a transaction. If we assume the degree of update ρ within a transaction, the storage required to hold the updated tuples will be equal to $\rho C B_r$.

The storage overhead in WAVE includes the time tag of the currency relation as well as history relation. The storage overhead of the time tag in the currency relation is CB_t. The size of each tuple in the history relation is equal to $B_r + 2B_t$. Again, we assume the same insert and delete operations, percentage of delete operations will be equal to $\frac{1-\mu}{2}$. The number of entries in the history relation is thus equal to $\rho C(\mu + \frac{1-\mu}{2}) = \rho C\frac{1+\mu}{2}$, giving a total size of $\frac{1}{2}\rho C(1+\mu)(B_r + 2B_t)$.

Figure 7.11 depicts the storage overhead when the degree of update, ρ, ranges from 0.5% to 2.5% of C, i.e., the number of update operations, ρC, ranges from 50 to 250. The percentage of modification operations, μ, ranges from 20% to 100% of ρC.

As shown in Figure 7.11, all three algorithms require extra storage which increases linearly with the degree of modifications and the degree of update. However, the storage requirement of WAVE increases much less rapidly than that of differential. The saving of storage in WAVE is mainly due to the fact that our history relation only maintains one tuple for every modification while in differential, two entries will be maintained in the log file for each modification.

Push-based algorithms usually require minimal storage as updated information are pushed to the client. Therefore, T-Strobe requires less storage than WAVE in general. However, we observe that when the degree of update reaches 2.5% and the

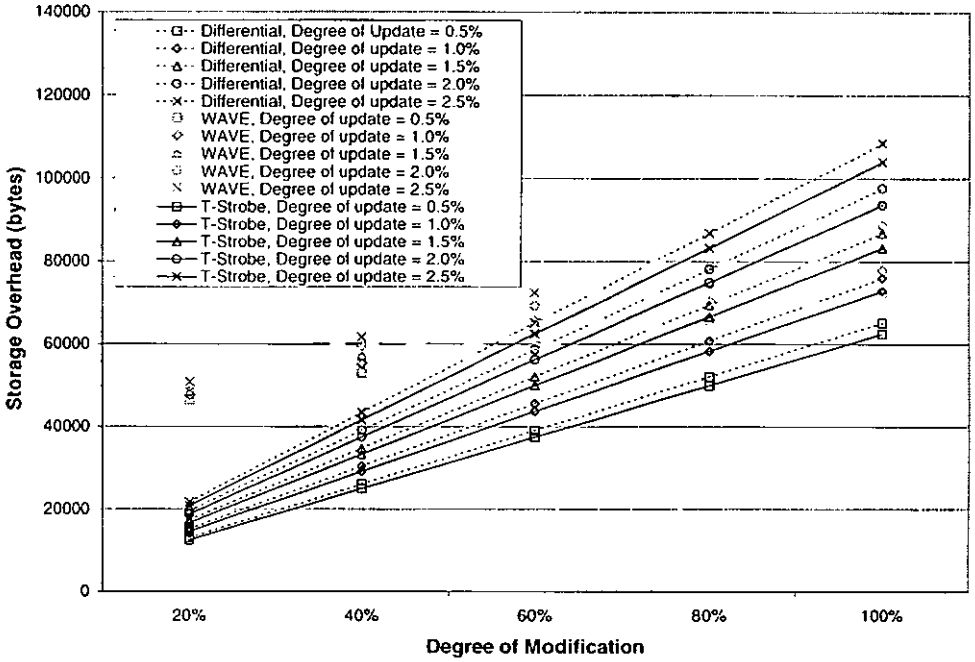


Figure 7.11: Storage Overhead against Degree of Modification

degree of modification reaches 80%, WAVE becomes more space efficient. This is because T-Strobe only requires storage for holding update information within a single transaction. When the degree of update and modification is small, it requires less storage than WAVE because WAVE requires extra storage to hold the time tags of each tuple. However, when the degree of update increases, WAVE outperforms T-Strobe because T-Strobe will require the same amount of extra storage for each inserted tuple, just like any kind of updated tuple. In WAVE, no extra storage is needed for an inserted tuple, except for the creation time tag information.

7.3 Discussion

In the first part of this chapter, we demonstrate the effectiveness of our semantic query caching scheme through a set of experiments. It has a good performance in terms of the elapse time, the cache hit ratio and the bandwidth consumption. In

a read-only environment where there is no database update, for a changing skewed hot access pattern, LRU performs the best while for a skewed hit access pattern, MRU is shown a little superior to others. Catering for a "large" cache fragment, cache decomposition with suitable threshold settings can enhance the performance of the cache to the changes in the access pattern. This could be observed from the self-answerability rate. Further, to alleviate the problem of over decomposition, cache fusion is used to coalesce smaller fragments to a larger one. That further increases the cache hit ratio and reduces bandwidth consumption and shortens elapsed time. The performance of semantic query caching scheme is satisfactory when update rate and update selectivity is low. This is a typical characteristic of the mobile environment.

Using a quantitative analysis, our proposed WAVE algorithm is shown to be effective and efficient. For a small join factor among different relations, low degree of update but with a higher degree of modifications, our WAVE performs efficiently in terms of less transmission overhead and small storage overhead compared with differential log and T-Strobe.

Chapter 8

Conclusion

8.1 Contribution of the Thesis

We proposed a pull-based view update mechanism in which the warehouse requests the differential changes from the stateless servers. Based on the use of creation time tag and deletion time tag in two relations: a currency relation and a history relation, a stateless server becomes able to determine the respective changes of a relation. We derived a mechanism to deduce the respective changes to a view, as long as the changes of a relation can be determined. Then we elaborated the mechanism from a single relation to multiple relations resident in one database server. Based on a similar idea, we extended the mechanism to our WAVE algorithm to handle view updating over multiple relations in different database servers. We demonstrated the correctness of our algorithm in the presence of view update anomaly problem which arises due to autonomy of servers and concurrent updates. To reduce number of messages, we developed an optimized version of WAVE algorithm in which all individual view accumulation change requests are grouped according to propagation directions to reduce number of messages.

We discussed our semantic query caching scheme. Every time a query result is

maintained in the client cache as a data block that we call *cache fragment*. The cache fragment is a basic caching unit. Each cache fragment is associated with a description that is the specification of the deriving query. We then discussed the idea of *query intersection*, which helps us to reason whether a query can be answered using a cache fragment, and illustrated the scenarios of reusing cache fragment to answer a query. We described the query transformation mechanism that rewrites a query into a probe query and a supplementary query when a cache fragment is reusable. The probe query retrieves required data from the cache fragment and the supplementary query represents the missing data. Then the complete query result can be constructed from both query results. If there is another fragment reusable, the supplementary query can be further transformed.

We proposed two cache manipulation techniques, namely, cache decomposition and cache fusion to handle cache management, which is complicated by a dynamic cache granularity. Usually a coarse cache fragment might contain data of different access frequency. To enhance the cache hit, those data items of lower access frequency need to be extracted from the cache fragment and to be disposed. In the thesis, our cache decomposition mechanism is used to divide a large cache fragment into several smaller sub-fragments. The idea of cache decomposition is similar to that of query transformation. To avoid a smaller cache fragment from being further decomposed, which induces excessive overheads in storing semantic information and processing query transformation and result construction, we suggested the use of decomposition threshold as an indicator. When the size of a cache fragment is smaller than the threshold, and the decomposition of the cache fragment is prohibited. In constrast to cache decomposition, our cache fusion mechanism is to fuse smaller fragment into one larger fragment. In our implementation, we use the final query result to substitute for the reused smaller cache fragments.

In the thesis, we described the implementation methodology of our prototype.

To precisely represent a query, we introduced a bit-stream pattern of two forms: normal form and compact form. Using the bit-stream query representation, query transformation can be performed in a structural way. Besides, we discussed our cache storage management by storing a cache fragment in a binary file in an existing file system.

Finally, to demonstrate the feasibility of our MoWS, a set of experiments was conducted and the representative sets of results were presented. Our experimental results clearly showed the suitability of MoWS in the mobile environment.

8.2 Future Work

Many important issues related to MoWS remain unexplored in the thesis. There are interesting questions that give rise to several implementation alternatives and performance optimizations. Below, we discuss a few such issues and outline directions for future research.

Semantic Data Broadcast: In MoWS, the caching model is based on point-to-point communication paradigm. If server data broadcast with semantics is available, that is, data being broadcast is described by means of semantic information, a part of a query result can be obtained from the broadcast. Then our query transformation mechanism can be further extended to rewrite a query into several sub-queries which load data from the cache, fetch data from the server and capture required data from the broadcast. It will make our work sound and complete.

Query Transformation Optimization: In the thesis, we did not consider how to transform a query in an optimized way. When a number of cache fragments are used to transform a query in different orders, the resulting transformed query could be so different. In the worst case, a query could be transformed into a

collection of overly fine sub-queries. To admit all those sub-query results, the cache would contain a lot of cache fragments. Also the construction of a query result from overly fine sub-query results incurs more computation overhead. So an optimization technique needs to be incorporated in the transformation mechanism. Currently we are studying some strategies focusing on the minimization of the number of sub-queries, less computation overheads in result construction and so on.

Cache Maintenance: Cache update mechanism is used to refresh those stale cache fragments. However, updating a cache fragment by incorporating differential changes might not be absolutely beneficial, as a large portion of a cache fragment is outdated. Then reusing this stale cache fragments is not recommended. In this case, query transformation mechanism should be supplemented with a filter to invalidate those stale fragments instead of refreshing them.

Besides, a fragment is decomposed when a query reuses it. This immediate approach might not be effective to locate the portion of a cache fragment with a higher access frequency. If sufficient access history of a cache fragment is maintained to conclude a suitable way to decompose the cache fragment, a better decomposition decision can be made. This will be one of the enhancements in our cache management.

Versioned Cache: In current work, the entire cache is refreshed every time a query is initiated. When disconnection occurs, a query entirely answered using the cache fragments can still be considered correct even the cache fragments may not be most up-to-date but they all belong to one consistent database state.

However, updating all cache fragments no matter whether they are reused by a query would incur high update overheads. Optimization can be acheived by selectively refreshing those reused cache fragments. Updating those required

cache fragments of a query will reasonably incur less overhead. Then another serious problem will arise that some fragments are updated and some remain stale. So, the cache will become a collection of cache fragments of different versions. A query reusing those cache fragments of different versions might not be guaranteed to be correct again if the choice of the version is not judicial.

If it is not a strict data requirement for user applications, relaxed consistency criteria could be considered. For example, some of alternatives are *Divergence Control* [13, 79]. In a cache, the version difference among cache fragments should be bounded. The checking based on timestamps of data items is discussed in [13]. So even though a query reuses those cache fragments of different version, the incorrectness is limited in a certain degree. Sometimes, inconsistency could be open-bounded depending on the nature of the application domains. If the data is used to indicate general information rather than to support critical decision making, the less accurate result can be accepted.

Heterogeneous Databases: Currently we assume all databases are of relational model. In a practical environment, it is not a must. An extension would be concentrated on different database models. Then the query processing and the view update mechanism would be very different and complicated

Complicated Query Specification and View Definition: Currently, we have only considered the view definition of a conjunctive projection-selection with a chain join of operand relations while the query is specified as a projection-selection on the materialized view. Besides a basic construct, we would like to study other complicated features such as involving aggregation functions and a recursive query (view), which is very common in ordinary database applications. Examples of aggregation functions include sum, average and maximum of certain attributes as well as a group-by function. A recursive query (view) is

one whose operand is another query (view). Catering for those kinds of features, query transformation and view update mechanism will definitely become more complicated.

Bibliography

[1] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Prefetching from a Broadcast Disk. In *Interntional Conference on Data Engineering*, pages 276–285, 1996.

[2] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing Push and Pull for Data Broadcast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 183–194, 1997.

[3] S. Adali, K.S. Candan, Papakonstantinou, and V.S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 137–148, 1996.

[4] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417–427, 1997.

[5] D. Agrawal, Omer Egecioglu, and Amr El Abbadi. Analysis of Quorum-Based Protocols for Distributed (k+1) – Exclusion. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):533–537, 1997.

[6] Divykant Agrawal and Soumitra Sengupta. Modular Synchronization in Distributed, Multiversion Databases: Version Control and Concurrency Control. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):126–137, 1993.

[7] B.R. Badrinath and T. Imielinski. Replication and Mobility. In *The 2^{nd} IEEE Workshop on Management of Replicated Data*, 1992.

[8] J. Bailey, G. Dong, and M. Mohania. Efficient Incremental View Maintenance Using Tagging in Distributed Database. Technical Report 95/37, Department of Computer Science, University of Melbourne, 1995.

[9] D. Barbara and T. Imielinski. Sleepers and Workaholics : Caching Strategies in Mobile Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, May 1994.

[10] J. Basu. *Associate Caching in Client-Server Databases.* PhD thesis, Department of Computer Science, Stanford University, 1998.

[11] D. Bitton, D.J. DeWitt, and C. Turbyfill. Benchmarking Database Systems A Systematic Approach. In *Proceedings of VLDB Conference*, pages 8–19, November 1983.

[12] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

[13] Boris Chan. Cache Consistency Management in Mobile Distributed Environment. Master's thesis, Department of Computing, The Hong Kong Polytechnic University, 1998.

[14] Boris Y.L. Chan, Antonio Si, and Hong V. Leong. Cache Management for Mobile Databases : Design and Evaluation. In *Interntional Conference on Data Engineering*, pages 54–63, February 1998.

[15] C.M. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer : Integrating Query Result Caching and Matching. In *Proceedings of EDBT Conference*, pages 323–336, March 1994.

[16] Manhoi Choy, Hong V. Leong, and Mei-Po Kwan. On Real-time Distributed Geographical Database Systems. In *Proceedings of the 27^{th} Hawaii International Conference on System Science, Volumn IV*, pages 337–346, 1994.

[17] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for Deferred View Maintenace. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469–480, 1996.

[18] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, and Inderpal Singh Mumick. Supporting Multiple View Maintenance Policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 405–416, 1997.

[19] F. Cristian, H. Aghili, and R. Strong. Clock Synchronization in the Presence of Omissions and Performance Faults, and Processors Joins. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, 1986.

[20] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Diversh Shrivastava, and Michael Tan. Semantic Data Caching and Replacement. In *Proceedings of VLDB Conference*, pages 330–341, August 1996.

[21] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching Multidimensional Queries Using Chunks. In *Proceedings of SIGMOD Conference*, pages 259–270, June 1998.

[22] Margaret H. Dunham and Abdelsalam (Sumi) Helal. Mobile Computing and Databases: Anything New? *SIGMOD Record*, 24(4):5–9, December 1995.

[23] S.J. Finkelstein. Common Subexpression Analysis in Database Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–245, 1982.

[24] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[25] Parke Godfrey and Jarek Gryz. Semantic Query Caching for Heterogeneous Databases. In *Proceedings of the 4th KRDB Workshop*, 1997.

[26] C.G. Gray and D.R. Cheriton. Leases : An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of SOSP*, pages 202–210, 1989.

[27] Timothy Griffin, Leonid Libkin, and Howard Trickey. An Improved Algorithm for the Incremental Recomputation of Active Relational Expression. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[28] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting Materialized Views after Redefinitions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 211–222, 1995.

[29] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[30] H. Gupta. Selection of views to Materailize in a Data Warehouse. In *Proceedings of ICDT*, pages 98–112, 1997.

[31] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf J. Joeressen, and Warren Allen. Bluetooth: Vision, Goals, and Architecure. *ACM MC2R*, 1(2):38–45, October 1998.

[32] Chun-Nan Hsu and Craig A. Knoblock. Reformulating Query Plans For Multidatabase Systems*. In *Proceedings of the International Conference on Information and and Knowledge Management*, pages 423–432, November 1990.

[33] Y. Huang, P. Sistla, and O. Wolfson. Data Replication for Mobile Computers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–24, June 1994.

[34] T. Imielinski and B.R. Badrinath. Querying a Highly Mobile Distributed Environments. In *Proceedings of International Conference on Very Large Data Bases*, pages 41–52, August 1992.

[35] T. Imielinski, S. Viswanathan, and B.R. Badrinath. Data on Air : Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, 1997.

[36] Tomasz Imielinski and B.R. Badrinath. Data Management for Mobile Computing. *SIGMOD Record*, 22(1):34–39, March 1993.

[37] Tomasz Imielinski and B.R. Badrinath. Mobile Wireless Computing : Challenges in Data Management. *Communication of ACM*, 37(10):18–28, 1994.

[38] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of Conjunctive Queries: Beyond Relations as Sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.

[39] C.S. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Database with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):461–473, 1991.

[40] Ronald K. Jurgen. Smart cars and highways go global. *IEEE SPECTRUM*, pages 26–36, May 1991.

[41] Magdi N. Kamel and Susan B. Davidson. Semi-Materialization : A Performance Analysis. In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, pages 125–135, January 1991.

[42] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computing Systems*, 10(1):3–25, February 1992.

[43] Henry F. Korth and Abraham Silberschatz. *Database System Concept.* McGraw Hill, 2 edition, 1991. QA76.9.D3K67.

[44] H. Kung and J. Robinsion. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[45] Wilburt Juan Labio and Hector Garcia-Molina. Efficient Snapshot Differential Algorithm for Data Warehousing. In *Proceedings of International Conference on Very Large Data Bases*, pages 63–74, 1996.

[46] D. Lee and W. Lee. On Signature Caching of Wireless Broadcast and Filtering Service. In *Proceedings of the 2nd International Mobile Computing Workshop*, pages 15–24, March 1996.

[47] Ken C.K. Lee, Hong V. Leong, and Antonio Si. Semantic Query Caching in a Mobile Environment. *ACM MC2R*, 3(2):28–36, April 1999.

[48] Ken C.K. Lee, Hong Va Leong, and Antonio Si. Incremental Maintenance for Dynamic Database-Derived HTML Pages in Digital Libraries. In *Proceedings of International Conference on Information and Knowledge Management*, pages 20–29, 1998.

[49] Ken C.K. Lee, Antonio Si, and Hong V. Leong. Incremental View Update for a Mobile Data Warehouse. In *Proceedings of the 13th ACM Symposium on Applied Computing*, pages 394–399, February 1998.

[50] W. Lee and D. Lee. Using Signature Techniques for Information Filtering in Wireless and Mobile Environments. *Special Issue on Database and Mobile Computing, Journal on Distributed and Parallel Database*, 4(3):205–227, July 1996.

[51] Hong V. Leong and Antonio Si. Data Broadcsating Strategies over Multiple Unreliable Wireless Channels. *Mobile Communications : Technology, Tools, Applications, Authentication and Security*, pages 31–38, September 1995.

[52] Hong V. Leong and Antonio Si. Data Broadcasting based on Statistical Operators. *IEE Electronics Letters*, 32(21):1951–1953, 1996.

[53] Hong V. Leong and Antonio Si. A Semantic Caching Mechanism for Mobile Database. *International Journal of Computers and Their Applications*, 4(2):21–34, August 1997.

[54] Hong V. Leong and Antonio Si. Database Caching over the Air-Storage. *The Computer Journal*, 40(7):401–415, 1997.

[55] Hong V. Leong and Antonio Si. On Adaptive Caching in Mobile Databases. In *Proceedings of 12th ACM Symposium on Applied Computing*, pages 302–309, February 1997.

[56] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 95–104, May 1995.

[57] B. Lindsay, C. Mohan, H. Pirahesh, and O. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 53–60, 1986.

[58] L.B. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15^{th} ACM Symposium on Operating System Principles*, pages 143–155, December 1995.

[59] V. Hadzilacos P.A. Berstein and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesly, Reading, Mass, 1987. QA76.9.D3B48.

[60] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Acorss Hetergeneous Information Sources. In *Interntional Conference on Data Engineering*, pages 251–260, March 1995.

[61] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expression. *IEEE Transactions on Knowledge and Data Engineering*, pages 337–341, 1991.

[62] Dallen Quass and Jennifer Widom. On-Line Warehouse View Maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, May 1997.

[63] Krithi Ramamritham and Calton Pu. A Formal Charaterization of Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[64] N. Roussopoulos. The Incremental Access Method of View Cache : Concepts, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[65] N. Roussopoulos, C.M. Chen, and S. Kelly. The ADMS Project : View "R" Us. *IEEE Data Engineering Bulletin*, 18(2):19–28, 1995.

[66] Peter Scheuermann, Junho Shim, and Radek Vingralek. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *Proceedings of International Conference on Very Large Data Bases*, pages 51–62, 1996.

[67] A. Segev and J. Park. Updating Distributed Materialized Views. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):173–184, 1989.

[68] Narayanan Shivakumar and Suresh Venkatasubramanian. Time- and Energy- Wireless Broadcasting. *ACM-Baltzer Journal of Mobile Networks and Nomadic Application (MONET)*, 2(2):129–140, 1996.

[69] Antonio Si and Hong V. Leong. Query Processing and Optimization for Broadcast Database. In *Proceedings of 7th International Conference on Database and Expert Systems Applications*, pages 899–914, September 1996.

[70] Antonio Si and Hong V. Leong. View Access Control for Object-Oriented Database in a Broadcast Environment. In *Proceedings of the Object-Oriented Database Systems Symposium*, pages 3–12, July 1996.

[71] Antonio Si and Hong V. Leong. Adaptive caching and refreshing in mobile databases. *Personal Technologies*, 1(3):156–170, September 1997.

[72] Antonio Si and Hong V. Leong. Query Optimization for Broadcast Database. *Data and Knowledge Engineering Journal*, 29:351–380, 1999.

[73] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering Queries with Aggregation Using Views. In *Proceedings of International Conference on Very Large Data Bases*, pages 318–329, 1996.

[74] A.U. Tansel, J. Clifford, S. Gadia, A. Segev, and R. Snodgrass. *Temporal Databases – Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company, Inc, 1993. QA76.9.D3T4125.

[75] R.H. Thomas. A Majority Consensus Approach To Concurrency Control for Multiple Copy Database. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[76] Jennifer Widom. Integrating Hetergeneous Databases : Lazy or Eager? *Computing Survey*, 28(4):91, 1996.

[77] O. Wolfson, P. Sistla, S. Dao, K. Narayanan, and Ramya Raj. View Maintenance in Mobile Computing. *SIGMOD Record*, 24(4):22–27, December 1995.

[78] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(4):255–314, 1997.

[79] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence Control Algorithm for Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):262–274, 1997.

[80] ANSI X.3.135-1992. *American National Standard for Information Systems – Database Language – SQL*, November 1992.

[81] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithm for Materialized View Design in Data Warehousing Environment. In *Proceedings of International Conference on Very Large Data Bases*, pages 136–145, August 1997.

[82] T. Yurek. Efficient View Maintenance at Data Warehouses. Master's thesis, Department of Computer Science, University of California at Santa Barbara, 1997.

[83] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.

[84] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1996.