

# Data Caching with Incremental Update Propagation in Mobile Computing Environments

H. Chung    H. Cho

Department of Computer Engineering  
Yeungnam University  
Kyongsan, Kyongbuk 712-749, Republic of Korea  
E-mail: hrcho@white.yeungnam.ac.kr

## Abstract

Users of mobile computers will soon have online access to a large number of databases via wireless network. Because of limited bandwidth, wireless communication is more expensive than wire communication. This implies that caching of frequently accessed data items will be an important technique that will reduce contention on the narrow bandwidth wireless channel. In this paper, we propose a new cache coherency scheme, named CCS-IUP (*cache coherency scheme with incremental update propagation*), that allows incremental update and use of derived caches. CCS-IUP is novel in the sense that it tries to avoid invalidating cached data; hence, the number of cache misses may be reduced. Furthermore, in CCS-IUP, the server propagates only relevant fractions (*increments*) of the modifications that affect the cached data, so that it can significantly reduce the amount of data transmission over the wireless network.

*Key Words and Phrases:* mobile computing, data caching, materialized view, update propagation, query processing

*CR categories:* H.2.4 (Database Systems), C.2.4 (Distributed Systems)

## 1 Introduction

Mobile computing - the use of a portable computer capable of wireless networking - will very likely revolutionize the way we use computers (Forman and Zahorjan, 1994; Imielinski and Badrinath, 1994). In particular, the mobile computing environment will give users with low powered palmtop machines capability of accessing databases over the wireless channels (Alonso and Korth, 1993; Imielinski and Badrinath, 1993). For example, while on the road, passengers will access airline schedules and weather information. Investors will access prices of financial instruments, salespeople will access inventory data, and callers will access location dependent data.

Sending access-requests from the mobile computer to the online database server may be expensive due to the limited uplink bandwidth, and also due to the fact that sending messages is a significant drain on the portable battery. This implies that each mobile user should access database in a way that minimizes communication. Furthermore, access to an online database may be unavailable when the user is out-of-range of the wireless network. These two problems can be alleviated by appropriate *data caching*, i.e., cache data in the local storage of mobile computers for later reuse (Barbara and Imielinski, 1994; Huang, Sistla and Wolfson, 1994). Data caching may lead to overall throughput improvement by making it possible to answer queries locally without competing for the scarce wireless bandwidth. For example, if a user frequently reads a data item  $x$  and  $x$  is updated infrequently, then it is beneficial for the user to cache a copy of  $x$  to his/her mobile computer. This way the reads access the local copy, and do not require communication.

Data caching is not a performance panacea, however. Because of data caching, several copies of a shared data item can exist in a database server and a number of mobile computers at the same time. Hence, to incorporate data caching, *cache coherency schemes* (CCSs) should be provided to guarantee that the cached data are maintained consistently. CCSs in mobile environments are more complicated than those of conventional client-server DBMS (Delis and Roussopoulos, 1993; Franklin, Carey and Livny, 1997). This is because (1) clients in mobile environments are often disconnected for prolonged periods of time due to the battery power saving measures, and (2) they are also frequently relocate between different cells and connect to different database server at different times. As a result, the server may no longer know which clients are currently located

under its cell and which of them are active.

## 1.1 Motivation

In mobile computing environments, there are two approaches of CCSs: *update invalidation* (CCS-I) and *update propagation* (CCS-P). In CCS-I, which was proposed by Barbara and Imielinski (1994), cached data of each client are invalidated if the client is informed that the data were updated by others. To inform clients, the server broadcasts a report (periodically or asynchronously) that includes updated data items. This results from the fact that wireless medium provides an excellent platform for broadcasting information to a massive number of users. In CCS-P, which was proposed by Huang et al. (1994), the server propagates updated data to the client, if the client caches the data. The propagation procedure can also be incorporated with the broadcasting mechanism.

We can compare the two approaches according to the number of bits that are transmitted in the channel both ways: downlink (from the server to the clients), and uplink (from the clients to the server). The tradeoff of two approaches is illustrated in Example 1.

**Example 1 (Tradeoff of Two CCSs):** Consider a communication scenario of a client and a server shown in Figure 1. Initially the client starts off with an empty cache. When an application of the client requests a query  $Q$ , it must be sent to the server due to empty cache. The server executes  $Q$  and returns the result  $R$  to the client. Then the client caches  $R$  in its memory. Now, suppose that some update operations are performed by the server and thus the result of  $Q$  is changed to  $R'$ . At this time, CCS-I makes the server broadcast an invalidation report that contains  $ID(R)$ , where  $ID(R)$  is an identity of  $R$ . Whereas, in CCS-P, the server sends the entire contents of  $R'$ . Since the size of  $ID(R)$  is usually smaller than that of  $R'$ , the amount of transmission in downlink can be reduced in CCS-I. If the client requests  $Q$  again, the order is reverted. In CCS-I, the client should send  $Q$  to the server since the cache was invalidated. Furthermore, the new result of  $Q$  (i.e.,  $R'$ ) should be sent to the client again. In CCS-P, however,  $Q$  can be processed locally since the client caches  $R'$ . As a result, CCS-I might suffer from a number of cache misses, which result in increasing data transmission in uplink and downlink.  $\square$

If the size of query result is small, the effect of invalidating the entire result or sending the

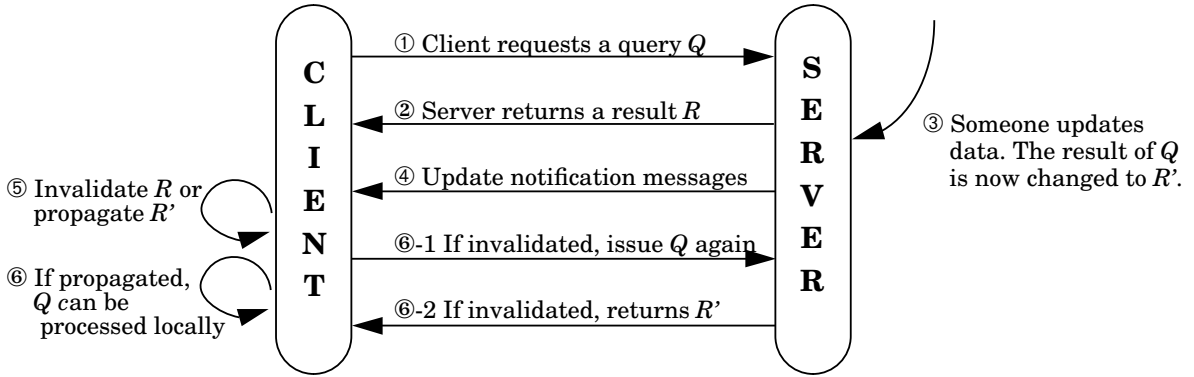


Figure 1: Communication scenario between a client and a server (example)

new result may not be significant. In fact, both CCS-I and CCS-P assume that queries are simple requests to read a numerical data item (stock data, temperature) or a textual data item (news). However, there may be a number of applications where the assumption of simple query does not hold. For example, salespeople could request catalog information for every household electrical appliances supported by his/her company. Investors may request a list of stocks each of which price has dropped during a week. In these applications, sending the entire query result must be very expensive operation; hence it should be avoided as far as possible.

In this paper, we propose a *CCS with incremental update propagation* (CCS-IUP) that allows incremental update and use of derived caches. Unlike CCS-I, CCS-IUP tries to avoid invalidating cached data; hence, the number of cache misses may be reduced. Furthermore, CCS-IUP propagates only relevant fractions (*increments*) of the modifications. This significantly reduces data transmission over the network as the server only transmits the increments affecting the cached data, compared with CCS-P where query results are continuously transmitted in their entirety.

The rest of the paper is organized as follows. Section 2 presents the model of caching and broadcasting. Then we describe the broadcasting algorithm and the cache reconstruction algorithm in Section 3 and Section 4, respectively. Section 5 presents optimization strategies that should be addressed to improve the performance of CCS-IUP. Section 6 describes previous works that are related to the paper. We conclude the paper in Section 7.

## 2 The Model

The database is a set of base relations  $R_i$ . Each relation consists of a number of records. To each record of a relation  $R_i$ , a unique record identifier ( $RID$ ) is assigned. Note that  $RID$  attribute may be hidden to the end user. Assume we have a large number of clients (i.e., mobile users) residing in a cell. Each client issues queries that may be selections, projections, or joins on base relations. The intelligent user interface such as spreadsheets may shield the actual query formats. In this case, there are lower level modules, called *database drivers*, that link the application to the actual DBMS. Note that the result of such query may consist of a large number of records, unlike the assumption of simple queries adopted in CCS-I and CCS-P. The clients exhibit a large degree of data locality, querying a particular subset of the database repeatedly. We assume that the database is updated only by the server. This assumption is not really necessary but it considerably simplifies the description of our algorithms.

Clients can cache a portion of the database. They can do this in a disk (if they are equipped with it), or any storage system that survives power disconnections, such as flash memories. We regard a cached data as a *materialized view* that is created as a result of query issuing in the client. Queries specify their target set of objects using *query predicates*, as in the WHERE clause of a SELECT-FROM-WHERE SQL query. We allow general SELECT-PROJECT-JOIN queries over one or more relations, with the restriction that  $RID$  attributes of all records of relations participating in a query must include in the query result. We feel this is not overly restrictive, since a query posed by the user that does not satisfy this constraint may optionally be augmented by a client to retrieve  $RID$  attributes from the server. Possible performance benefits are then (1) records need not be stored in duplicate, and (2) the materialized view may be *self-maintainable* (Gupta and Mumick, 1995) with respect to deletions to base relations. The latter benefit is mostly desirable in the sense that views can be maintained using only the materialized view (without base relations stored in the database server).

The specification of the cached data can be formally described as follows: Let  $R_i$  and  $R_j$  be two base relations maintained in the server database. Then the cached data can be one of the views  $\mathcal{V}_\sigma$ ,  $\mathcal{V}_\pi$ , and  $\mathcal{V}_\bowtie$  that are derived via relational operators *selection*  $\sigma$ , *projection*  $\pi$ , and *join*

$\bowtie$ , respectively be:

$$\begin{array}{ll}
 \text{selection view:} & \mathcal{V}_{\sigma(P)} = \sigma_P(R_i) \\
 \text{projection view:} & \mathcal{V}_{\pi(\omega)} = \pi_{\omega}(R_i) \\
 \text{join view:} & \mathcal{V}_{\bowtie} = R_i \bowtie R_j
 \end{array}$$

where  $P$  implies the selection predicate and  $\omega$  is an attribute list of the projection. The view created by intersection, union, and difference can be maintained similarly with the join algorithm and not included in this paper.

Figure 2 illustrates a stock trading database that consists of two base relations and three views of the relations. Each view corresponds to  $\mathcal{V}_{\sigma}$ ,  $\mathcal{V}_{\pi}$ , and  $\mathcal{V}_{\bowtie}$ , respectively. In Figure 2, we assume that the  $RID$  attribute of each view represents  $RID$  of the corresponding record of base relations. This implies that a join view may include several  $RID$  attributes. In Figure 2,  $\mathcal{V}_3$  has two  $RID$  attributes,  $RID_C$  for “Company” relation and  $RID_S$  for “Stock” relation.

By regarding the cached data as a view, we can *reconstruct* the view if any records of base relations are updated. The reconstruction procedure performs with three information: old view that was cached, query that created the view, and information for updated records that were broadcasted by the server. The detailed algorithm is shown in Section 4.

We assume that the server has no information about which clients are currently under its cell and what are the contents of their cache. This assumption is well suited in mobile computing environments due to the high frequency of disconnection and mobility of clients. The server therefore should broadcast an *update report* that contains one or a number of updated records. There are two alternative methods for broadcasting: *asynchronous* and *synchronous* (Barbara and Imielinski, 1994). In asynchronous method, the server broadcasts the update report immediately after changes to the records of base relations occur. Whereas, synchronous method is based on periodic broadcasting of the update report. Synchronous method is advantageous when a client is reconnected after some periods of disconnection. Asynchronous method does not provide any latency guarantees for the querying client; if the client queries a data item after the disconnection period then it either has to wait for the next report (with no time bound on the waiting time) or has to submit the query to the server (cache miss). In case of synchronous method, however, there is a guaranteed

**Company**

<i>RID</i>	<i>cname</i>	<i>field</i>	<i>capital</i>
1	IBM	computer	300,000
2	HANIL	bank	20,000
3	DEC	computer	100,000
4	SEOUL	bank	30,000
5	UNIVAC	computer	50,000
6	KIA	car	70,000
7	TAEGU	bank	10,000
8	PUSAN	bank	10,000

**Stock**

<i>RID</i>	<i>sno</i>	<i>cname</i>	<i>price</i>	<i>old-price</i>
1	100	IBM	358.25	360.50
2	101	DEC	295.50	285.00
3	102	HANIL	38.00	38.00
4	103	SEOUL	52.25	53.50
5	104	UNIVAC	175.75	180.00
6	105	TAEGU	27.50	28.00
7	106	KIA	94.50	90.25
8	107	PUSAN	27.75	27.00

$V_1$ : (selection view)

**select** \*  
**from** Stock  
**where** *price* < 100;

$V_1$				
<i>RID</i>	<i>sno</i>	<i>cname</i>	<i>price</i>	<i>old-price</i>
3	102	HANIL	38.00	38.00
4	103	SEOUL	52.25	53.50
6	105	TAEGU	27.50	28.00
7	106	KIA	94.50	90.25
8	107	PUSAN	27.75	27.00

$V_2$ : (projection view)

**select** *sno, price*  
**from** Stock;

$V_2$		
<i>RID</i>	<i>sno</i>	<i>price</i>
1	100	358.25
2	101	295.50
3	102	38.00
4	103	52.25
5	104	175.75
6	105	27.50
7	106	94.50
8	107	27.75

$V_3$ : (join view)

**select** *sno, cname, price*  
**from** Company, Stock  
**where** Company.*cname* = Stock.*cname*  
**and** Company.*field* = "bank";

$V_3$				
<i>RID<sub>C</sub></i>	<i>RID<sub>S</sub></i>	<i>sno</i>	<i>cname</i>	<i>price</i>
2	3	102	HANIL	38.00
4	4	103	SEOUL	52.25
7	6	105	TAEGU	27.50
8	8	107	PUSAN	27.75

Figure 2: Stock trading database

latency due to the periodic nature of the synchronous broadcast. Hence we adopt the synchronous broadcasting method.

In the synchronous broadcasting method, a client has to listen the update report first in order to conclude whether its cache is valid. Note that this adds some latency to query processing. If there are some update operations on the base relations, the cache should be reconstructed. The answer to a query will therefore reflect any updates to the base relations during the interval in which the query was posed. Note that this is the case even if the query predates the update during the interval.

### 3 Broadcasting Algorithm

CCS-IUP consists of two algorithms for broadcasting and view reconstruction, respectively. The broadcasting algorithm describes a communication protocol between a server and clients. The view reconstruction algorithm specifies how each client can reconstruct its cached view. In this section, we describe the broadcasting algorithm of CCS-IUP. The view reconstruction will be described in the next section.

Queries involving base relations are transmitted to and processed initially by the server. When the result of a query is cached into a view for the first time, this “new” view is *bound* to the base relations used in extracting the result. For each view  $\mathcal{V}$ , the client keeps the following four items to record such binding:

- $PR$  - the set of participating base relations of  $\mathcal{V}$
- $ATT$  - the set of attributes of  $\mathcal{V}$
- $COND$  - the applicable conditions on  $PR$
- $TS(\mathcal{V})$  - a timestamp

The notion of the binding is well matched to the semantics of SELECT-FROM-WHERE in SQL statement. Specifically, the value of  $PR$  comes from the relation names in FROM clause, and the attribute list of SELECT clause can be used to fill  $ATT$ . Similarly, the value of  $COND$  can be collected from the WHERE clause. The condition is essentially the filtering mechanism that decide what are the qualifying records for this client. The timestamp indicates that the view is correct at time of the timestamp.

For example, suppose a client caches three views of Figure 2. Then the binding information of each view can be set to as Figure 3. The value of  $COND$  of  $\mathcal{V}_2$  is set to NULL, because the WHERE clause is not defined. The value of  $ATT$  of  $\mathcal{V}_1$  is set to “\*” that means the view includes every attribute of base relation.

For each base relation participating to the cached view, the client maintains an *update propagation log* and a timestamp. Let  $UPL_i$  be the update propagation log of a base relation  $R_i$ .  $UPL_i$

<i>View</i>	<i>PR</i>	<i>ATT</i>	<i>COND</i>	<i>TS</i>
$V_1$	{Stock}	*	$price < 100$	1,000
$V_2$	{Stock}	{ <i>sno, price</i> }	NULL	2,000
$V_3$	{Company,Stock}	{ <i>sno, cname, price</i> }	Company. <i>cname</i> = Stock. <i>cname</i> <b>and</b> Company. <i>field</i> = "bank"	3,000

Figure 3: Binding information of each view (example)

is partitioned as *incremental deletion logs* ( $D_i$ ) and *incremental insertion logs* ( $I_i$ ). A modification of a record is treated as a pair of a deletion and an insertion. Each record of  $D_i$  is a pair of  $RID$  and  $TS$ , where  $RID$  is an identifier of a deleted record and  $TS$  is a timestamp representing when the record was deleted. On the other hand, each record of  $I_i$  includes every attribute of  $R_i$  and one additional attribute for timestamping ( $TS$ ). The timestamp of each relation indicates a time of the last update propagation record. The value is initially set to 0.

The server broadcasts the update report periodically at times  $T_k$ . The update report consists of information about records that have been changed in the last  $\Delta$  seconds. The value of  $\Delta$  is set to an integral times of broadcasting period. Note that the choice of the value of  $\Delta$  has a significant effect on the amount of data transmission. Intuitively, if the value of  $\Delta$  is too large then the size of propagation report could increase, because the old update logs may be broadcasted a number of times together with the new update logs. Furthermore, the large size of propagation report may result in the increased processing overhead for the client to check the report. On the other hand, if the value of  $\Delta$  is too small, the client could face an increasing number of cache misses when it tries to reconnect. The detailed discussion about the size of  $\Delta$  can be found in Barbara and Imielinski (1994).

The update report is defined formally as a list  $\Phi$  of which element is a 4-tuple ( $R_i, RID, VAL, TS$ ) where

- $R_i$  - the relation identifier that includes the record of  $RID$
- $RID$  - the record identifier that was inserted or deleted during the interval  $[T_k - \Delta, T_k]$

- *VAL* - the value of record if it was inserted; otherwise (i.e., deleted) it is null
- *TS* - a timestamp indicating when the record was updated lastly

Upon receiving the list of  $\Phi$ , the client compares  $R_i$  to its binding information to decide whether to keep each  $\Phi$  or not. Also the client has a list of queries  $Q_i$  that have been issued in the interval  $[T_{k-1}, T_k]$ . The client also keeps a variable  $T_l$  that indicates the last time it received a report. The formal description of the broadcasting algorithm is now shown in Figure 4.

If the difference between the current report timestamp ( $T_k$ ) and  $T_l$  is more than  $\Delta$ , then the client purges all the caches it maintained. This is because the client has been disconnected very long time, and thus there may be some updates that are not broadcasted to the client and the current report does not include them, too. If the difference is smaller than  $\Delta$ , the client checks whether  $\Phi$  needs to be kept. The client can skip  $\Phi$  if there are no views that correspond to  $\Phi$  or  $\Phi$  has been reflected already. Otherwise,  $\Phi$  is appended to the appropriate incremental logs of the base relation. After checking all  $\Phi$ s, the client also updates  $T_l$ . Now the client can process its queries that have been issued during the interval. If the result of the query is not cached, then it should be forwarded to the server. Otherwise, the query can be processed locally. Before processing the query, the cached view should be reconstructed if the participating base relations have been updated. As a result of view reconstruction, the timestamp of the view is changed to the maximum timestamp of base relations.

Example 2 illustrates the broadcasting procedure for the stock trading database of Figure 2.

**Example 2 (Broadcasting Procedure of CCS-IUP):** Suppose a client  $\mathcal{C}$  caches three views of Figure 2, and the binding information of each view is constructed as Figure 3. Furthermore, the timestamp of both base relations at  $\mathcal{C}$  are supposed to be 3900, which means that  $\mathcal{C}$  keeps every update log that has been made before 3900. Suppose that an update report is broadcasted to  $\mathcal{C}$  at 4500 (i.e.,  $T_k = 4500$ ) and the report consists of the following four logs:

---

```

if ( $T_k - T_l > \Delta$ ) { drop entire cache } /* reconnected after long periods of disconnection */
else {
  for every element ( $R_i, RID, VAL, TS$ ) in  $\Phi$  {
    if ( $\exists \mathcal{V}_c$  such that the client caches  $\mathcal{V}_c$  and  $R_i \in PR_c$ ) {
      if ( $TS < TS(R_i)$ ) { skip  $\Phi$  } /* The log has been reflected already */
      else {
        if ( $VAL$  is null)  $D_c = D_c \cup \{(RID, TS)\}$ 
        else  $I_c = I_c \cup \{(VAL, TS)\}$ 
         $TS(R_i) = TS$  /* increments the timestamp of base relation */
      }
    }
  }
   $T_l = T_k$  /* Reset the timestamp of the client to the current time */
}
for every query  $Q_i$  {
  if (the result of  $Q_i$  is not cached) /* cache miss */
    { request  $Q_i$  to the server }
  else {
    Let  $\mathcal{V}_i$  be the cached result of  $Q_i$ 
    if ( $\exists PR_i$  such that  $D_i \neq \emptyset$  or  $I_i \neq \emptyset$ ) { /* Base relation has been updated */
      call view reconstruction
       $TS(\mathcal{V}_i) = \text{MAX}_{R \in PR_i}(TS(R))$  /* view's TS is set to maximum TS of relations */
    }
    Use  $\mathcal{V}_i$  to answer  $Q_i$ 
  }
}

```

---

Figure 4: Broadcasting algorithm of CCS-IUP

$\Phi_1$ :	[Stock, 8, NULL, 4000]
$\Phi_2$ :	[Stock, 8, (8, 107, 'PUSAN', 26.00, 27.00), 4200]
$\Phi_3$ :	[Company, 2, NULL, 4300]
$\Phi_4$ :	[Company, 2, (2, 'HANIL', 'bank', 25000), 4400]

In this case,  $\mathcal{C}$  should keep every log of the update report. This is because (1) each log corresponds to either 'Stock' relation or 'Company' relation both of which participate to the views cached in  $\mathcal{C}$ , and (2) the timestamp of each log is greater than that of base relation maintained in  $\mathcal{C}$ . Since  $\Phi_1$  and  $\Phi_3$  are deletion logs, they are appended to the incremental deletion logs of 'Stock' relation and 'Company' relation, respectively. On the other hands,  $\Phi_2$  and  $\Phi_4$  are appended to the incremental insertion logs of the corresponding base relation. As a result of the log appending, the timestamp information is changed. Specifically,  $TS(\text{Stock})$  is set to 4200, and  $TS(\text{Company})$  is set to 4400. The timestamp of  $\mathcal{C}$  is also changed to 4500, which is the value of  $T_k$ .  $\square$

A novel optimization strategy about the broadcasting algorithm is that it is possible not to drop the entire cache even though the difference between  $T_k$  and  $T_l$  is more than  $\Delta$ . This is achieved by adopting the notion of *update history window*  $\mathcal{W}$ , where  $\mathcal{W} > \Delta$  (Wu, Yu and Chen, 1996; Cai, Tan and Ooi, 1997). The basic idea is that the server also maintains update logs for as far back as  $\mathcal{W}$  seconds. If the disconnection time of a client ( $T_k - T_l$ ) is between  $\Delta$  and  $\mathcal{W}$ , then the client may request missing logs to the server. After receiving the logs, the client can incrementally reconstruct its cache. If the disconnection time is larger than  $\mathcal{W}$ , then it implies that the client's cache is totally invalid and the query has to be processed from scratch. As a result, the cache miss ratio can be reduced significantly if the server keeps update logs for large  $\mathcal{W}$ .

## 4 View Reconstruction Algorithm

CCS-IUP reconstructs the cached view if all of the following three conditions are satisfied: (1) an application of a client issues a query, (2) the result of the query is cached in the client, and (3) some records of the participating base relations have been updated.

We first introduce some notations for incremental logs. Given  $D_i$  and  $I_i$  of a base relation  $R_i$ ,

let  $D_i^\pi$  means  $\pi_{RID}(\sigma_{TS>TS(\mathcal{V}^*)}(D_i))$  and  $I_i^\pi$  means  $\pi_{RID,VAL}(\sigma_{TS>TS(\mathcal{V}^*)}(I_i))$ , where  $\mathcal{V}^*$  represents  $\mathcal{V}_{\sigma(P)}$  or  $\mathcal{V}_{\pi(\omega)}$ , respectively. As a result, both  $D_i^\pi$  and  $I_i^\pi$  include only update logs that are not reflected to the view. Now, we can reconstruct the selection view ( $\mathcal{V}_{\sigma(P)}$ ) and the projection view ( $\mathcal{V}_{\pi(\omega)}$ ) as follows:

$$\begin{aligned}\mathcal{V}_{\sigma(P)} &= (\mathcal{V}_{\sigma(P)} \ominus D_i^\pi) \cup \sigma_P(I_i^\pi) \\ \mathcal{V}_{\pi(\omega)} &= (\mathcal{V}_{\pi(\omega)} \ominus D_i^\pi) \cup \pi_\omega(I_i^\pi)\end{aligned}$$

where  $R \ominus S = R - \pi_R(R \bowtie S)$ . For both cases of reconstructing  $\mathcal{V}_{\sigma(P)}$  and  $\mathcal{V}_{\pi(\omega)}$ , we first remove the records of  $D_i^\pi$  found in the view, and then insert new (qualified) records of  $I_i^\pi$  to the view. This holds because the unary relational operators of selection and projection can be distributed over the union operator.

Note that projection requires more work for processing both  $I_i$  and  $D_i$ . For the insertions, the incremental merge has to make sure that it does not generate any duplicates. For the deletions, it must either eliminate a record or not, depending on the existence of duplicates of the record. In CCS-IUP implementations, we recommend that each client caches every duplicate of a projection. Then the duplicate is eliminated when the query result is returned to the user. This can simplify both the structure and the incremental maintenance of client's cached view.

Given  $D_i$  and  $I_i$  of a base relation  $R_i$ , and  $D_j$  and  $I_j$  of a base relation  $R_j$ , we can reconstruct the join view ( $\mathcal{V}_{\bowtie}$ ) as follows: Let  $D_i^\pi$  means  $\pi_{RID}(\sigma_{TS>TS(\mathcal{V}_{\bowtie})}(D_i))$  and  $I_i^\pi$  means  $\pi_{RID,VAL}(\sigma_{TS>TS(\mathcal{V}_{\bowtie})}(I_i))$ .

$$\begin{aligned}\mathcal{V}_{\bowtie} &= ((R_i \ominus D_i^\pi) \cup I_i^\pi) \bowtie ((R_j \ominus D_j^\pi) \cup I_j^\pi) \\ &= ((R_i \bowtie R_j) \ominus (D_i^\pi \times D_j^\pi)) \cup ((R_i \ominus D_i^\pi) \bowtie I_j^\pi) \cup ((R_j \ominus D_j^\pi) \bowtie I_i^\pi) \\ &= (\mathcal{V}_{\bowtie} \ominus (D_i^\pi \times D_j^\pi)) \cup ((R_i \ominus D_i^\pi) \bowtie I_j^\pi) \cup ((R_j \ominus D_j^\pi) \bowtie I_i^\pi)\end{aligned}$$

Note that  $\mathcal{V}_{\bowtie}$  cannot be reconstructed using the update propagation logs only. In other words, the computation of  $\mathcal{V}_{\bowtie}$  for  $R_i$  and  $R_j$  requires the contents of both base relations (due to the second and third terms of the last equation). There are two alternatives for solving the problem: (1) to

invalidate  $\mathcal{V}_{\bowtie}$  in case of updates, and (2) to cache both  $R_i$  and  $R_j$  together with  $\mathcal{V}_{\bowtie}$ . The alternative (1) would result in less storage overhead and lower communication cost. However, it may suffer from frequent cache misses, and thus  $\mathcal{V}_{\bowtie}$  should be frequently transferred from the server. Note that the size of  $\mathcal{V}_{\bowtie}$  is usually much larger than  $R_i$  and  $R_j$ . In addition, if the client caches both  $R_i$  and  $R_j$ , it can support other queries for the relations without communication to the server. This is why we choose alternative (2). In Section 5, we describe special kinds of updates which do not require base relations to reconstruct the join views.

Example 3 illustrates the view reconstruction procedures for the stock trading database of Figure 2.

**Example 3 (View Reconstruction Procedure of CCS-IUP):** Like Example 2, a client  $\mathcal{C}$  is supposed to cache three views of Figure 2. Suppose that the update report of Example 2 is the only one that was broadcasted after the creation of each view. Then the incremental logs of each base relation may be constructed as follows:

$D_{Stock}$ :	[8, 4000]
$I_{Stock}$ :	[(8, 107, 'PUSAN', 26.00, 27.00), 4200]
$D_{Company}$ :	[2, 4300]
$I_{Company}$ :	[(2, 'HANIL', 'bank', 25000), 4400]

The reconstruction of  $\mathcal{V}_1$  is rather simple. Since  $\mathcal{V}_1$  is a selection view and it is defined on 'Stock' relation, both  $D_{Stock}$  and  $I_{Stock}$  need to be checked. Remember that  $TS(\mathcal{V}_1)$  is 1000 (see Figure 2). So  $D_{Stock}^\pi$  and  $I_{Stock}^\pi$  are equal to  $D_{Stock}$  and  $I_{Stock}$ , respectively. Since  $\mathcal{V}_1$  has a record with  $RID$  8, the record is removed from  $\mathcal{V}_1$  as a result of  $(\mathcal{V}_1 \ominus D_{Stock}^\pi)$ . In case of  $I_{Stock}^\pi$ , the value of 'price' attribute is smaller than 100. So the log of  $I_{Stock}^\pi$  should be appended to  $\mathcal{V}_1$ . Figure 5(a) shows the new  $\mathcal{V}_1$  after the reconstruction.

The projection view  $\mathcal{V}_2$  can also be reconstructed like to the selection view. In this case, a record with  $RID$  8 is removed first and then a new log from  $I_{Stock}^\pi$  is inserted to  $\mathcal{V}_2$ . Figure 5(b) shows the new  $\mathcal{V}_2$  after the reconstruction.

As we have described previously, the join view  $\mathcal{V}_3$  cannot be reconstructed without the base relations. So in this example, we assume that  $\mathcal{C}$  caches the base relations of  $\mathcal{V}_3$  when  $\mathcal{V}_3$  is created

<p>(a) selection view</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="5"><math>V_1</math></th> </tr> <tr> <th><math>RID</math></th> <th><math>sno</math></th> <th><math>cname</math></th> <th><math>price</math></th> <th><math>old-price</math></th> </tr> </thead> <tbody> <tr><td>3</td><td>102</td><td>HANIL</td><td>38.00</td><td>38.00</td></tr> <tr><td>4</td><td>103</td><td>SEOUL</td><td>52.25</td><td>53.50</td></tr> <tr><td>6</td><td>105</td><td>TAEGU</td><td>27.50</td><td>28.00</td></tr> <tr><td>7</td><td>106</td><td>KIA</td><td>94.50</td><td>90.25</td></tr> <tr><td>8</td><td>107</td><td>PUSAN</td><td>26.00</td><td>27.00</td></tr> </tbody> </table>	$V_1$					$RID$	$sno$	$cname$	$price$	$old-price$	3	102	HANIL	38.00	38.00	4	103	SEOUL	52.25	53.50	6	105	TAEGU	27.50	28.00	7	106	KIA	94.50	90.25	8	107	PUSAN	26.00	27.00	<p>(b) projection view</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="3"><math>V_2</math></th> </tr> <tr> <th><math>RID</math></th> <th><math>sno</math></th> <th><math>price</math></th> </tr> </thead> <tbody> <tr><td>1</td><td>100</td><td>358.25</td></tr> <tr><td>2</td><td>101</td><td>295.50</td></tr> <tr><td>3</td><td>102</td><td>38.00</td></tr> <tr><td>4</td><td>103</td><td>52.25</td></tr> <tr><td>5</td><td>104</td><td>175.75</td></tr> <tr><td>6</td><td>105</td><td>27.50</td></tr> <tr><td>7</td><td>106</td><td>94.50</td></tr> <tr><td>8</td><td>107</td><td>26.00</td></tr> </tbody> </table>	$V_2$			$RID$	$sno$	$price$	1	100	358.25	2	101	295.50	3	102	38.00	4	103	52.25	5	104	175.75	6	105	27.50	7	106	94.50	8	107	26.00	<p>(c) join view</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th colspan="5"><math>V_3</math></th> </tr> <tr> <th><math>RID_C</math></th> <th><math>RID_S</math></th> <th><math>sno</math></th> <th><math>cname</math></th> <th><math>price</math></th> </tr> </thead> <tbody> <tr><td>2</td><td>3</td><td>102</td><td>HANIL</td><td>38.00</td></tr> <tr><td>4</td><td>4</td><td>103</td><td>SEOUL</td><td>52.25</td></tr> <tr><td>7</td><td>6</td><td>105</td><td>TAEGU</td><td>27.50</td></tr> <tr><td>8</td><td>8</td><td>107</td><td>PUSAN</td><td>26.00</td></tr> </tbody> </table>	$V_3$					$RID_C$	$RID_S$	$sno$	$cname$	$price$	2	3	102	HANIL	38.00	4	4	103	SEOUL	52.25	7	6	105	TAEGU	27.50	8	8	107	PUSAN	26.00
$V_1$																																																																																																	
$RID$	$sno$	$cname$	$price$	$old-price$																																																																																													
3	102	HANIL	38.00	38.00																																																																																													
4	103	SEOUL	52.25	53.50																																																																																													
6	105	TAEGU	27.50	28.00																																																																																													
7	106	KIA	94.50	90.25																																																																																													
8	107	PUSAN	26.00	27.00																																																																																													
$V_2$																																																																																																	
$RID$	$sno$	$price$																																																																																															
1	100	358.25																																																																																															
2	101	295.50																																																																																															
3	102	38.00																																																																																															
4	103	52.25																																																																																															
5	104	175.75																																																																																															
6	105	27.50																																																																																															
7	106	94.50																																																																																															
8	107	26.00																																																																																															
$V_3$																																																																																																	
$RID_C$	$RID_S$	$sno$	$cname$	$price$																																																																																													
2	3	102	HANIL	38.00																																																																																													
4	4	103	SEOUL	52.25																																																																																													
7	6	105	TAEGU	27.50																																																																																													
8	8	107	PUSAN	26.00																																																																																													

Figure 5: The example views after reconstruction

in  $\mathcal{C}$ . Then we can reconstruct  $\mathcal{V}_3$  by (1) removing records of  $\mathcal{V}_3$  that correspond to  $D_{Stock}^\pi$  or  $D_{Company}^\pi$ , and (2) inserting the results of join operations on  $Company'$  with  $I_{Stock}^\pi$  and  $Stock'$  with  $I_{Company}^\pi$ .  $Company'$  is created by deleting records of 'Company' that correspond to  $D_{Company}^\pi$ .  $Stock'$  can be created similarly with 'Stock' and  $D_{Stock}^\pi$ . The reconstructed  $\mathcal{V}_3$  is shown in Figure 5(c). □

The view reconstruction algorithm is similar to the incremental access method (IAM) for views, which was proposed by Roussopoulos (1991). IAM has been developed for implementing views in a centralized relational DBMS. The main difference of IAM and view reconstruction algorithm of CCS-IUP is the method of representing views. In IAM, each view is represented as a collection of indices that point to the records of base relations required to materialize the view. While this representation method is well suited in the centralized DBMS due to the low storage overhead and smart handling of incremental updates, it must result in lots of data transmission in mobile computing environments. If the cached view contains only pointers to actual records of base relations (stored in server), materializing the view requires transmitting the actual records from the server to the client. This reduces or diminishes the benefits of caching. Therefore, in CCS-IUP, each view stores the actual result of a query as a snapshot. A previously constructed snapshot can be retrieved directly without any data transfer.

## 5 Optimization

In this section, we consider optimization strategies about query processing and reducing downlink transmission that should be addressed to improve the performance of CCS-IUP.

### 5.1 Query Processing

#### 5.1.1 Cache Completeness

In CCS-IUP, queries are used to form predicates that describe the cache contents. Then the subsequent query at the client may be satisfied in its local cache if we can determine that the query result is entirely contained in the cache. This issue is often called *cache completeness*, which was introduced firstly by Keller and Basu (1996) in the area of client-server DBMS. For example, suppose a client caches a view  $\mathcal{V}_1$  of Figure 2, and the client should process a query that selects ‘Stock’ relation each of which ‘price’ attribute is less than 50. Note that the query can be processed with  $\mathcal{V}_1$ , since the condition of the query is more restrict than the condition attached to  $\mathcal{V}_1$  (i.e.,  $COND(\mathcal{V}_1)$ ). This implies that queries with more restrict condition can be processed without (expensive) wireless communication.

To achieve the cache completeness, we should be able to check whether a query can be processed with local cached view. This problem may be translated into checking inclusion relationships between a condition of a query and that attached to the view. One idea is to describe the condition as a range of the maximum acceptable value (MAV) and the minimum acceptable value (MIV). For two conditions  $C_1$  and  $C_2$  on the same attribute,  $C_1$  is included to  $C_2$  if  $MAV(C_1) \leq MAV(C_2)$  and  $MIV(C_1) \geq MIV(C_2)$ . The similar idea has been introduced for *predicate locking* used for concurrency control in traditional database systems. However, predicate locking uses the notion of the predicate by preventing two locks with *intersecting* predicates from being held concurrently. In case of projection, it should also be checked whether the attribute list of a query is included in that of a view. However solving this problem is rather simple, and thus we omit the detailed description.

### 5.1.2 Join Processing with Partial Information

A view that takes the join of two or more relations ( $\mathcal{V}_{\bowtie}$ ) is not *self-maintainable* with respect to updates, i.e., the view cannot be maintained without requiring access to any base relations (Gupta and Mumick, 1995). Since it may not be acceptable to invalidate  $\mathcal{V}_{\bowtie}$  or to cache all of the base relations, we should develop an optimized view maintenance plan that consumes both network bandwidth and storage capacity as little as possible. The plan should address the following subproblems: detection of *irrelevant updates*, (2) detection of *autonomously computable updates*, and (3) *re-definition* of the view. The followings describe each subproblems in more detail.

An update operation into a base relation is *irrelevant* to a view if it causes no record to be updated, inserted or deleted into the view (Blakeley, Coburn and Larson, 1989). Suppose a join view  $\mathcal{V}_{\bowtie}$  of which bound information is  $PR_{\bowtie}$  (the set of participating base relations of  $\mathcal{V}_{\bowtie}$ ),  $ATT_{\bowtie}$  (the set of attributes of  $\mathcal{V}_{\bowtie}$ ), and  $COND_{\bowtie}$  (the applicable conditions on  $\mathcal{V}_{\bowtie}$ ). An update operation of a record  $r$  into a base relation  $R_u \in PR_{\bowtie}$  is irrelevant to  $\mathcal{V}_{\bowtie}$  if and only if  $COND_{\bowtie}(r)$  is unsatisfiable. For example, suppose a  $\mathcal{V}_{\bowtie}$  is defined as  $\pi_{a,b,d}(\sigma_{d>30}(A \bowtie B))$  where the relation schemes are  $A(a,b,c)$  and  $B(c,d,e)$ . In this case, if a record  $(20, 20, 20)$  is inserted into  $B$ , then it is irrelevant to  $\mathcal{V}_{\bowtie}$  since the value of  $d$  is 20 which is not greater than 30. Note that irrelevant updates are worth to be detected since they can be omitted in the procedure of view reconstruction without resulting in any inconsistencies.

If an update operation is not irrelevant to a view, then some data from the base relations may be necessary to update the view. An important case to consider is one in which all the required data is contained in the view itself. Updates of this type is called as *autonomously computable updates* (Blakeley et al., 1989; Tompa and Blakeley, 1988). For example, suppose a  $\mathcal{V}_{\bowtie}$  is defined as  $\pi_{b,c,d}(\sigma_{b>e}(A \bowtie B))$  where the relation schemes are  $A(a,b,c)$  and  $B(c,d,e)$ . Suppose also the  $\mathcal{V}_{\bowtie}$  contains a record  $(45, c_1, d_1)$ . If a record  $(a_1, 50, c_1)$  is inserted into relation  $A$ , we can infer that the  $\mathcal{V}_{\bowtie}$  should have the additional record  $(50, c_1, d_1)$ . This inference can be made, using the current view maintenance and the inserted record, by observing that if  $e_1$  is less than 45 then it is also less than 50. As a result, the two base relations are not used and the insertion is autonomously computable. Detecting autonomously computable updates is important since the new state of the

view can be computed solely from the view definition, the current state of the view, and the updated record (i.e., without accessing base relations).

Even though the updated record is not irrelevant and not autonomously computable, it is possible not to invalidate the entire view. Instead the definition of the view may be changed to reflect that the view does not include the inserted view. For example, suppose a  $\mathcal{V}_\times$  is defined as  $\pi_{a,b,d}(A \bowtie B)$  where the relation schemes are  $A(a, b, c)$  and  $B(c, d, e)$ , and the key of  $A$  is  $a$ . If a record  $(20, 20, 20)$  is inserted into  $A$ , it is not irrelevant and not autonomously computable. In this case, the definition of  $\mathcal{V}_\times$  is changed as  $\pi_{a,b,d}(\sigma_{a \neq 20}(A \bowtie B))$ . Now a query  $\pi_{a,b}(\sigma_{a > 20}(A \bowtie B))$  can be executed with accessing  $\mathcal{V}_\times$  only.

### 5.1.3 Query Trimming

The performance of CCS-IUP can be further optimized for queries that cannot be processed with local caches only. The basic idea is to *reconstruct* the WHERE clause of the query so that the server sends only the query results that are not cached in the client. For example, in Figure 2, a query on  $\mathcal{V}_1$  with condition ‘ $price < 150$ ’ may be reconstructed to a new query with ‘ $100 \leq price < 150$ ’. Since the cached data are not sent again from the server, the amount of communication can be reduced. However, sending only the new query from the client may result in database inconsistency with regard to the concurrency control. This problem can be resolved simply by sending both the original query and the new query to the server, and the server returns the result of the new query while concurrency control is performed on the original query.

## 5.2 Reducing Downlink Transmission

We now consider the issue of reducing the amount of update reports broadcasted from the server. The size of update report in CCS-IUP is larger than that in CCS-I, since CCS-IUP tries to propagate updates to each client; in contrast, CCS-I just invalidates each client’s cache in case of updates. To reduce the size of update reports, the server may keep track of each client’s cache information. This is achieved by maintaining the queries requested from each client. In this case, the server need not broadcast updates that do not affect any caches in all clients.

One novel idea about this issue is *to relax the consistency of caches* by allowing some degree of value divergence from the data in the server. For example, if clients are caching stock information of Figure 2, it may be perfectly acceptable to use values that are not completely up to date, as long as they are within 0.5 percent of the true prices. This allows updates to be broadcasted from the server to the clients more efficiently (e.g., when the server is lightly loaded, or by batching updates together).

In this context, the notion of *quasi-copies* which was introduced by Alonso, Barbara and Garcia-Molina (1990) may be considered. A quasi-copy is a cached value that is allowed to deviate from the central value in a controlled way. However, the quasi-copy should be guaranteed to meet a certain predicate (named *quasi-caching predicate*). Several types of inconsistency predicate may be defined: *delay condition* (e.g., the cache must not be more than one hour old than the central value), and *arithmetic condition* (e.g., the cache's value must not be off by more than 10 percents of that of the central value). With regard to the delay condition, the update logs may not be broadcasted promptly when the update is occurred. This implies that the server may broadcast the update report so that it can use the communication bandwidth more efficiently and the server bottleneck may be prevented. The notion of arithmetic condition allows the server to broadcast an item only if it changes more than the prescribed limit. This will also reduce the number of the update logs that should be broadcasted.

## 6 Related Work

Several techniques to provide incremental updates at clients have been recently investigated for relational DBMSs with a central server and multiple clients connected through LAN (Delis and Roussopoulos, 1993; Keller and Basu, 1996). In both techniques, query results (i.e., materialized views) are cached into each client. Then a subsequent query at the client may be satisfied in its cache if it is possible to determine that the query result is entirely contained in the cache. To maintain each client's cache consistently, they assume that the server keeps track of the cache descriptions of each client; hence, the server can determine which clients should be informed the updates of base relations. Furthermore, since the server and the clients are connected through

LAN, sending enough information for view derivation (Delis and Roussopoulos, 1993) or sending recomputed view after invalidation (Keller and Basu, 1996) would not have prohibitive overhead. Note that this assumption does not hold in the mobile computing environment because (1) there may be a large number of mobile computers attached to the database server, (2) each mobile computer is often disconnected or relocate, and (3) wireless communication is more expensive than wire communication due to limited bandwidth.

In the mobile computing environment, Cai et al. (1997) have extended CCS-IUP proposed in the previous version of this paper (Chung and Cho, 1996). They first proposed two criteria, dissemination of update report and view reconstruction, for classifying the cache coherency schemes. With regard to the dissemination of update report, there are two alternatives: update propagation (UP) and update invalidation (UI). With regard to the view reconstruction, only the client may perform the reconstruction (C), or both the server and the client can collaborate to reconstruct the view (CS). According to the criteria, they propose the following four combinations: UP-C, UP-CS, UI-C, UI-CS. Note that the basic idea of UP-C is similar to that of CCS-IUP. On the basis of the simulation study, they conclude that CS-based schemes may perform better than C-based schemes due to powerful processing capacity of the server and less downlink transmission. However, their simulation results are not completely convincing in the sense that they did not consider any query optimization strategies described in Section 5. Note that in C-based schemes there are more chances to reduce uplink transmission with the query optimization strategies, since the client contains every update log in C-based schemes. Furthermore, they did not consider the case where there are large number of clients. In this case, the server may suffer from frequent query requests in CS-based schemes.

## 7 Concluding Remarks

In this paper, we proposed a new cache coherency scheme, named CCS-IUP (*cache coherency scheme with incremental update propagation*) for mobile computing environments. Unlike the previous approaches in mobile computing environments, where invalidating or propagating the entire cache in case of updates, CCS-IUP allows incremental update and use of derived caches. As a

result, CCS-IUP can reduce the number of cache misses compared to the invalidating approach. Furthermore, compared to the propagating approach, CCS-IUP can significantly reduce the amount of data transmission over the wireless network, since the server transmits only the relevant fractions (*increments*) of the updates that affect the cached data.

Any successful implementation must be based on a good conceptual structure and design. In this paper, we have attempted to address some major conceptual and design issues. We are developing an experimental testbed to evaluate the viability of our approach, using a prototype of a CCS-IUP based mobile computing environment. Detailed design of our experiments is currently in progress. Simulation studies to compare the performances of alternative caching schemes for large numbers of clients and queries are also planned.

Apart from the planned performance studies, many other important issues remain unexplored in this paper. Work currently in progress addresses implementation questions on suitable indexing technique for conditions of queries and views, performance tuning, local index creation for cached views, and effective transaction processing for the views. Development of analytical system models, heuristics for effective conservative and liberal approximations of cache descriptions, and intelligent query-containment algorithms for determining cache completeness are topics for future efforts.

## References

- ALONSO, R., BARBARA, D. and GARCIA-MOLINA, H. (1990): Data Caching Issues in an Information Retrieval System, *ACM Trans. on Database Syst.*, 15(3), September: 359-384.
- ALONSO, R. and KORTH, H. (1993): Database System Issues in Normadic Computing, in *Proc. of ACM SIGMOD*: 388-392.
- BARBARA, D. and IMIELINSKI, T. (1994): Sleepers and Workaholics: Caching Strategies in Mobile Environments, in *Proc. of ACM SIGMOD*: 1-12.
- BLAKELEY, J.A., LARSON, P. and TOMPA, F.W. (1986): Efficiently Updating Materialized Views, in *Proc. of ACM SIGMOD*: 61-71.

- BLAKELEY, J.A., COBURN, N. and LARSON, P. (1989): Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, *ACM Trans. on Database Syst.*, 14(3), September: 369-400.
- CAI, J., TAN, K. and OOI, B.C. (1997): On Incremental Cache Coherency Schemes in Mobile Computing Environments, in *Proc. of 13th Int'l Conf. on Data Eng.*: 114-123.
- CHUNG, H. and CHO, H. (1996): Data Caching with Incremental Update Propagation in Mobile Computing Environments, in *Proc. of 1st Aust. Workshop on Mobile Computing, Database, and Appl.*: 42-52.
- DELIS, A. and ROUSSOPOULOS, N. (1993): Performance Comparison of Three Modern DBMS Architectures, *IEEE Trans. on Softw. Eng.*, 9(2), February: 120-138.
- FORMAN, G. and ZAHORJAN, J. (1994): The Challenges of Mobile Computing, *IEEE Comput.*, 27(4), April: 38-47.
- FRANKLIN, M.J., CAREY, M.J. and LIVNY, M. (1997): Transactional Client-Server Cache Consistency: Alternatives and Performance, *ACM Trans. on Database Syst.*, 22(3), September: 315-363.
- GUPTA, A. and MUMICK, I.S. (1995): Maintenance of Materialized Views: Problems, Techniques, and Applications, *IEEE Data Eng. Bull.*, 18(2), June: 3-18.
- HUANG, Y., SISTLA, P. and WOLFSON, O. (1994): Data Replication for Mobile Computers, in *Proc. of ACM SIGMOD*: 13-24.
- IMIELINSKI, T. and BADRINATH, R. (1993): Data Management for Mobile Computing, *SIGMOD RECORD*, 22(1), March: 34-39.
- IMIELINSKI, T. and BADRINATH, R. (1994): Mobile Wireless Computing, *Comm. ACM*, 37(10), October: 19-28.
- KELLER, A. and BASU, J. (1996): A Predicate-Based Caching Scheme for Client-Server Database Architectures, *VLDB J.*, 5(1), January: 35-47.

ROUSSOPOULOS, N. (1991): An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis, *ACM Trans. on Database Syst.*, 16(3), September: 535-563.

TOMPA, F.W. and BLAKELEY, J.A. (1988): Maintaining Materialized Views without Accessing Base Data, *Information Syst.*, 13(4), April: 393-406.

WU, K., YU, P.S. and CHEN, M. (1996): Energy-Efficient Caching for Wireless Mobile Computing, in *Proc. of 12th Int'l Conf. on Data Eng.*: 336-343.