

Short Paper

Client's Cache Updating Using Idle Time of Clients in Relational Database Management Systems

YASAR-GUNERI SAHIN* AND HALIL-IBRAHIM BULBUL

**Department of Computer Engineering*

Yasar University

Izmir, Turkey

E-mail: yasar.sahin@yasar.edu.tr

Department of Computer Education

Gazi University

Ankara, Turkey

Although multi-tiered software development is taking the place in database application environments, client-server infrastructure has still been in use for years. Furthermore, both infrastructures still need to be improved in some aspects. One of the important aspects of improvement is caching. This paper addresses the issue of caching in relational database system within client-server structure and a method that can be used to keep clients' cache up to date. The main objective of the proposed caching methods is to use clients' idle time in order that clients' cache is up-to-dated and to use the clients' memories for caching. For this purpose, a new combined system which includes CCDT (Clients Cache Description Table) for organizing client cache updating, predicate-based caching method MNTC (Maximum Number of Tuples in Cache) that includes a managerial patch in server-side and small patches in clients-side, and TPCF (Time Period calculated from client Commit Frequency) for calculation of idle time of the clients, is presented.

Keywords: caching algorithm, cache description, database caching, cached updating, client-server structure

1. INTRODUCTION

Cache applications are extensively applied within the database management system infrastructure [1]. Some of those applications such as the relational database management systems (RDBMS) and the Object-Oriented Database Management Systems (OODBMS) have been used successfully in client/server architecture and web based applications software.

This study is focused on independent multi-clients with their server and the cache mechanism that can be used on a relational database. Client starts transactions and the server carries out the data management sharing. Existence of the query results in the local cache of the client contributing to an increase in the performance of distributed clients in data sharing.

Received February 14, 2006; revised July 28, 2006; accepted August 31, 2006.

Communicated by Ming-Syan Chen.

In the client-server architecture trade oriented relational databases cache applications are used with buffering method to reduce the amount of disk traffic in the frequently used disc blocks only at the side of server [2]. What is assumed here is that the clients have lower-capacities than servers. However, with the great achievements in technology, the clients now have increasingly larger memories and higher capacities to carry out numerous transactions. Technological developments in clients have eased the data and network traffic with the servers and have contributed to a reduction of time loss in reaction time. Thus it is no longer up-to-date to use cache mechanisms just on the server.

Recently several different client cache methods have been carried out [3, 4]. In those methods, transactions of cache loading, updating and changing were based on the logic of the IDs of objects. Some studies resemble our work. Some of them were “a predicate-based caching scheme” and “predicate locks” [5, 6]. Because of the execution cost, the predicate lock applications have not been regarded as successful applications and they also have a negative effect on the simultaneity [7]. These applications have lower performance because of the impediment of phantoms. Another similar study was the study of precision locks [8].

Other studies that were examined in this work are; the study of Larson and Yang (1987) and Sagiv and Yannakakis (1980) in which the first query evaluation had been carried out [9, 10], the studies for providing a restoration of constituted views [11-13], a study on the effects of the updating transactions on clients' caches [14], and the studies concerning cache applications carried out in clients' side [4, 15-27]. In this study, the essential point is it's considering the cache application carried on over the RDBMS, this is the main point that this study differs from others.

The approach of this study is that queries were executed on the server side and upon the information (necessary data for CCDT) gathered from the results of queries located on the client's side a Client Cache Description Table (CCDT) is constituted. CCDT includes information on the client caches. If a new query is a partial cache hit, then only the parts of this query that are non-existent shall be sent by the server and the cache would be updated. The server controlling the CCDT would carry out this transaction.

Despite the similarity of a predicate-based caching scheme (Keller and Basu 1996), there is a main difference between our and their study, that our method has a different commit protocol. An additional difference is the determination of the time of the transactions in case of a triggering or limiting transaction of a local update (such as insert, delete or update). There are two different methods presented in this study for the determination of committing time. One of those methods is the carrying out committed transaction immediately using CCDT that is prepared according to the query results. Another method is to carry out the transaction with the time period calculated from client commit frequency according to the commit frequency obtained from the updating transactions performed by the client.

2. COMMIT TIMING

The most serious problem between predicate-based caching algorithm and the usage of the CCDT obtained from queries arises in the updating process of CCDT and the client caches. In this section, the update problem arising during the usage of predicate-based caching as well as our approach to solve this problem is dealt with.

2.1 Commit Timing Problem

Let (C_i) and (C_j) be two individual clients on a client-server system. Commit timing problem arises when client (C_i) creates a transaction for a big sized range query on a relation, and if, client C_j concurrently updates a tuple on the same relation and commits this update transaction.

In such a case, the CCDT data on the server does not possess any cache information regarding C_j (since the query transaction initiated by C_i has not been completed yet). As the information required to tell C_i about the update process committed by C_j is not on CCDT yet, the update committed by C_j will not be contained in the result set just obtained by C_i . Therefore, the requested range query will not contain the correct result set.

For example, considering a scenario in which client (C_i) has started a query transaction on a student relation existing in a student database as follows;

```
Select * From Student_Records
Where Student_Id > 4001000 and Student_Id < 4010999.
```

And also considering that client (C_j) concurrently performs an update as follows and commits it;

```
Delete From Student_Records Where Student_Id = 4001002.
```

In this situation, where the above SQL statements start sequentially, C_i will try to reach approximately 10,000 student records. Meanwhile, C_j deletes the record of *Student_Id* 4001002 and commits this transaction before query of C_i completion. If the query requested by C_i has not been completed yet, then, this will mean that the commit on the CCDT made by C_j will have been completed before the transaction initiated by C_i and therefore no cache update signal will be send by CCDT to the clients.

Thus, since C_i is not aware of the update committed by C_j yet, the query result placed in C_i 's cache will contain 10,000 students instead of the correct record number of 9,999 which will be different from the correct results. In addition, if C_i ignores C_j update concurrently, then the query result has again 10,000 records instead of 9,999 therefore, wrong set is placed to client's cache.

This scenario seems that there is no problem in DBMS, because concurrent transactions are consistent if they are serialize able, but in the client cache side (C_1, \dots, C_n) this situation is a problem. In this scenario, since C_i has not been updated after C_j commitment, if C_i wants to create new request for same query then C_i must be re-updated fully or partially and therefore redundant reading and updating are required. In order to avoid these redundant transactions, we offer a new approach that is using CCDT.

2.2 Suggested Solutions

One of the ways to solve the problem given in section 2.1 is to lock the entire relation during a range query transaction. However, in case the range query transaction lasts long, then this will cause other clients to wait for a long time. Similarly, if more than one client performs a range query on the same relation, it will be even harder for other clients

to reach this relation. Therefore locking the entire relation will not ensure a reasonable solution to the problem.

The solution we propose is to ensure that clients making range query requests perform necessary changes on the CCDT at the time of their request instead of making cache updating on the relevant the CCDT after completing range query transaction. These clients will, after the end of the transaction and before updating their cache, re-check the CCDT to see if there have been any recent changes. In addition, they will check the CCDT and perform updating their cache during their idle time, which will be calculated by means of a formula obtained depending on the committing frequency. Thus, idle times of clients are utilized in an efficient manner.

Moreover, the total number of cached updates on the clients is restricted in this solution by using MNTC (Maximum Number of Tuples in Cache) patches, thus allowing automatic update instruction to the clients by the MNTC manager that supervises this number.

3. PROPOSED METHOD: CCDT (MODIFIED SCD)

This section deals with how clients update the CCDT (*Client Cache Description Table*), which is proposed and developed in order to keep client caches updated, and what procedures they follow in their query and update transactions.

CCDT located on the server, like SCD (Server Cache Description) in “predicate based caching scheme”, keeps information about the contents of the cache on the client’s- side [5]. In this study, a structure as given in Table 1 below is used. The contents of this table vary according to the application areas.

Table 1. Server CCDT attributes.

Attribute	Description	Length (Bytes)
Client_No	Client id	5
Relation_Name	Relation name	30
Tuple_Count	The number of tuples in the cache	4 (Long integer)
SQL	SQL Statement (If the cache content is a query result set)	1,000 (Text)
Updated_Items	Items of any update on the cache, if any.	1,000 (Binary)

In this Table, the field shown with *Client_No* represents the client to which the cache data belongs, *Relation_Name* represents the relation to which the data on the client cache belongs to, *Tuple_Count* represents the number of tuples existing on the cache in case the content is the result of the query made, SQL attribute is the SQL statement used to obtain the data on the cache in case the data on the cache has been obtained by query, and *Updated_Items* is the list of tuples occurring in case the data in the cache are updated by other clients.

The client initiates a transaction for a query request, as shown in Fig. 1 (a). Before this request is processed by Database Management System, a new record is created on the CCDT in line with the request information and the request is added in this record with

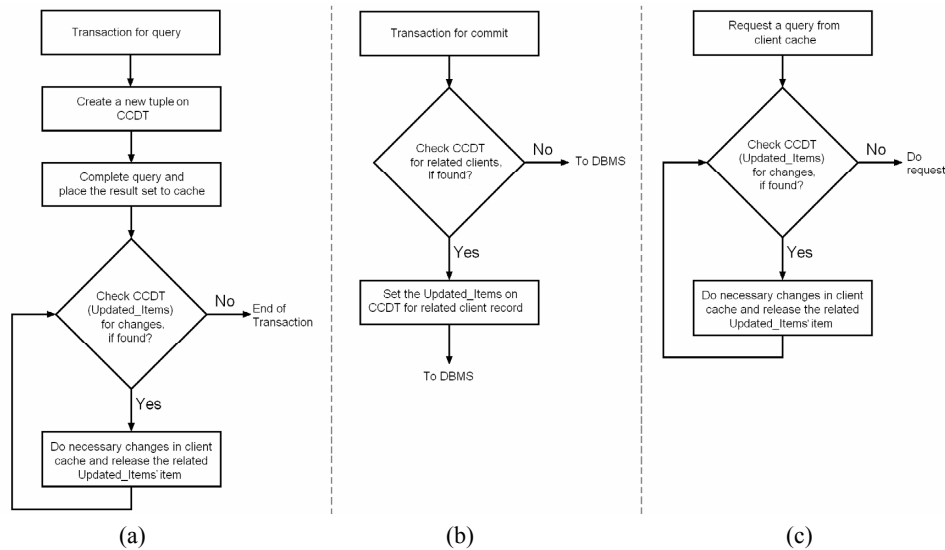


Fig 1. (a) CCDT workflow for a query transaction; (b) CCDT workflow for a commit transaction (update); (c) CCDT workflow for a cache hit.

necessary information such as client number, relation name *etc.* Next, DBMS gets the request for the query. After the result set obtained, it is placed on the client cache. After this step, client, which requests the query, checks *Updated_Items* attribute on the record created for itself in CCDT. If any update has been committed on any item, then the same update is performed also on the client cache through *Updated_Items'* item. After the update, the update information existing in the relevant *Updated_Items'* item is deleted.

This procedure sequence is continued until no data remain in the *Updated_Items* attribute. Absence of any data in *Updated_Items* attribute means that update necessity no longer exists, with all the updates being completed.

Fig. 1 (b) shows a flowchart about client (C_j) that commits a related update. If the update requested by a client applying for commit directly affects the cache of another client (query requesting), an update information such as *tuple_no*, *attribute_name etc.* is inserted in the *Updated_Items* item of the tuple on the CCDT, belonging to the client (requesting the query) that will be affected from this commit. This procedure ensures the background to inform the affected client of this update process. If the query on the affected client (query requesting) has not been completed yet, then the CCDT will wait the query result. If the query result has been placed into the client cache, the CCDT will inform this client of the changes, which will be performed by the same client. Then, the changes made in the *Updated_Items* item will be deleted.

If multiple clients ($C_k, k = 1, \dots, r$) execute updates on the same relations concurrently, in this situation, all clients are queued according to FIFO and they update *Updated_Items* of CCDT with items no, which affected by update, sequentially. Since CCDT has single tuple for a particular SQL of C_i , no client can keep CCDT busy forever (because, clients read and write CCDT in a certain instant), and C_i has placed in a certain place in queue therefore, C_i checks the CCDT and applies the updates at any instant. In

this situation, some of updates (updates recorded to CCDT after checking of C_i) will be ignored C_i temporarily, and these updates will be up to dated at any idle time of C_i in future.

If a new query is a cache hit, as can be seen in Fig. 1 (c), before using directly the data on its cache, it will check the *Updated_Items* item of the tuple on CCDT belonging to itself to see whether there is any change. After all the necessary changes have been handled, the client cache will be updated and it will finally contain the correct data that can be used for query by client without any need to the server.

When the client requesting a query gives up its request, it will delete the relevant tuple on the CCDT, thus eliminating the need of being informed of any further updates. Thanks to these procedures, the CCDT will be able to know all the cache data on all clients and in case of any update, it will ensure that the update will be made by the clients. Therefore, client caches will be kept up-to-date continuously because of CCDT.

4. MNTC MANAGER ON SERVER

The purpose of MNTC (Maximum Number of Tuples in Cache) is to restrict the number of tuples updated by the clients using cached update method, which consists of collection of insert, delete or update processes and of committing all of them in a single transaction. The purpose of this restriction is to execute update transactions automatically, thus ensuring the changes to be made on CCDT in order. In addition, the total number of cache updates should be restricted, because predicting or calculating idle time of some clients are very hard, or sometimes some of clients might not have idle time. In this situation, restriction of the total number of cache updates is a useful way to say that, it is time to up-to-date.

As shown in Fig. 2, the MNTC system comprises of a MNTC manager located on the server and MNTC patches located on the clients. The system operates by first utilizing the data kept by MNTC manager on the MNTC table. The MNTC manager first permits the client requesting an update then determines the lock(s) on the tuple(s) on which an update is to be made and places these locks on the MNTC table. If the transaction is an insertion, it will not require any lock, but it will be written on the MNTC table containing the new tuple that has been inserted by client and waiting to be committed. MNTC manager knows which client will perform update transaction on which relation. In case a client (C_i) requests a query and there is a possible update by another client (C_j) known by MINTC, MNTC sends a signal to the latter client (C_j) that is likely to perform the procedure to finish the process. Thus, the MNTC manager forces the latter client (C_j) to respond this signal by committing, rollback or cancelling the transaction.

At the same time, the MNTC manager will force clients to perform multiple commits in idle time, via a function depending on commit frequency of the clients, in order to reduce the possibility of encountering locked tuples by cached update processes existing on clients during queries. Thus, the idle time will be utilized more efficiently. In addition, MNTC manager is also responsible, in coordination with CCDT, from sending to MNTC patches the data about the update processes to be performed, depending on relation and on client.

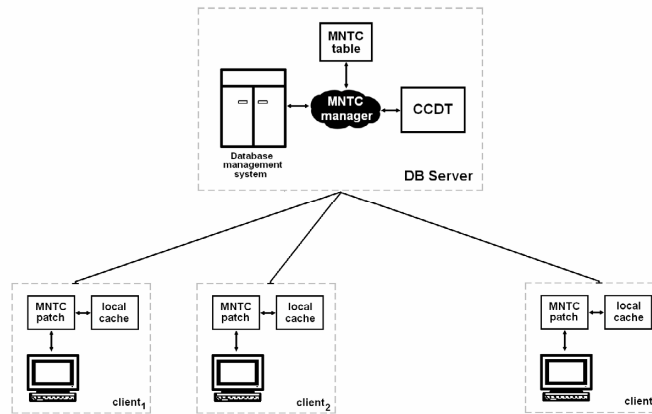


Fig. 2. MNTC schema.

MNTC patches on the clients are also responsible from monitoring forced commit or cancel signals sent by MNTC manager and from performing necessary processes. Besides, these patches are responsible from automatically performing the commit in case the total number of cached updates on the client reaches the number reported by MNTC manager.

Using a dynamically changing value instead of using a fixed value in order to determine the number of tuples automatically to be kept in cache will give some advantages depending on the situation. At the same time, it will be possible for the client to permanently update the records in the cache without having to wait a MNTC signal for the commit.

A simple formula has been presented in this study, aiming to allow necessary time range for performing commit signal automatically during operation. This formula may be developed further in order to calculate optimum time.

Considering the above formula;

$$TPCF = TTC - TSC \quad (1)$$

where, TPCF: Time Period calculated from client Commit Frequency, TSC: Time Spend during a Transaction for Commit (Calculated on the average), TTC: Average Time interval between Two Commit transactions on the same client.

TPCF is a time value used to calculate idle time intervals between transactions. With this value, the intervals during which the clients request update transaction are approximately calculated, and the idle periods interval between two transactions are determined dynamically and continuously by MNTC manager. TPCF is kept in a record separately for each client by MNTC manager and reported to MNTC patches. MNTC Manager is also responsible from TPCF synchronization calculated for MNTC manager clients. If there are not any priorities involved, this synchronization is performed by FIFO method.

As can be seen in Fig. 3, TPCF is less than TTC, by at least as much as TSC. In other words, in order that a new transaction can be finalized, TPCF should be at least as

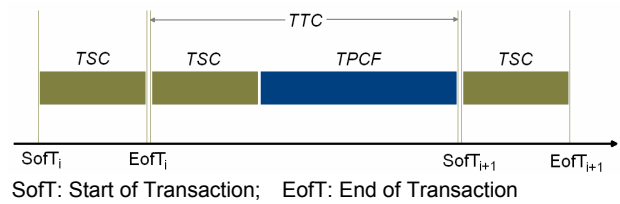


Fig. 3. TPCF time diagram.

long as the time required for a transaction normally. In cases where the period of TPCF is less than the transaction time to be newly created, it is considered appropriate to stop the procedure for a new transaction and to continue with it after the new transaction has been completed. MNTC manager changes TPCF dynamically and continuously, and reports it to the MNTC patches.

Other important aspect of TPCF is that, it ensures frequent check of CCDT by clients at certain intervals through TPCF. Therefore, when there is a query result in a client cache (see section 3) and if this result set is required at any time, there will be no need to re-check CCDT, thus saving time and a reduction in traffic is ensured.

5. EXPERIMENTAL RESULTS

The results of experiments performed during the application phase are dealt with in this section. The cache application in the study and the time required for obtaining the query transactions are shown in this section. The results have been gathered under the condition of test environment showed in Tables 2 and 3.

Table 2. Experiment environment.

Tools	Description
DBMS	Oracle 8.1.7i
DB Connection Object	ADO
DB	Student Registration DB
Number of Tuples in Student Records Relation	34,000
Number of Clients	25
DB Server CPU	Intel Xeon (Single)
Network	100 Mbps Switched
Software	Student Registration Module (Developed with Borland Delphi)

Table 3. Constraints and restrictions.

Options	Value
Isolation Level	Cursor Stability
Marshall Option	Marshal Modified Only
Lock Type	Optimistic
Cursor Location	Client
Cursor Type	Dynamic

5.1 When the Client Cache is Empty

The Fig. 4 (a) shows the elapsed time for monitoring the results of query transactions performed when the client is empty. Here the average number of tuples to be included by each query is between 1,000 and 11,000. As it can be seen from the figure, due to the additional time required for updating the cache information, using cache takes slightly more time than without cache application, since there is no need for such an update.

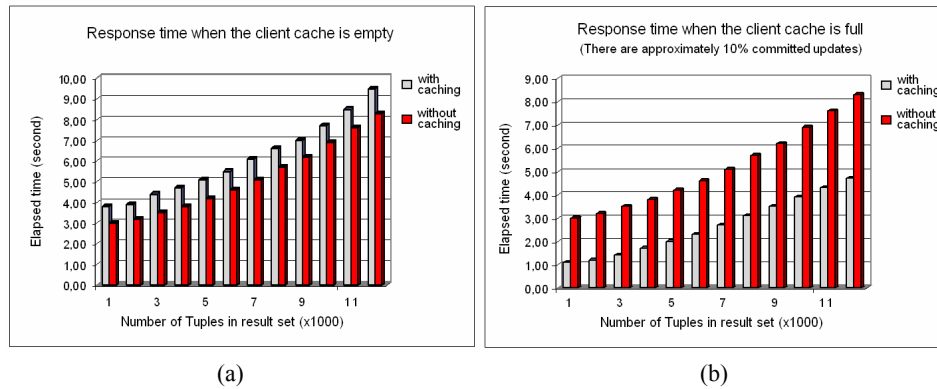


Fig 4. (a) Query result set response time when client cache is empty; (b) query result set response time when client cache is full and assuming approximately 10% committed updates.

5.2 When the Client Cache is Full

This section provides the time required for responses when the client caches are filled with similar queries beforehand. Following the cache update, changes were made to the tuples over the relation that will change query content by 10% (committed updates).

As it can be seen from the Fig. 4 (b), due to the stabilization of the cache level at 10%, significant decreases have been provided in response time. Thus, an improvement is obtained in this application when it is compared with the applications without the cache use.

5.3 The Results with Respect to Number of Attempts

Fig. 5 (a) shows elapsed time required for the transactions that are carried out when the client cache is full (*i.e.* starting from the 2nd attempt into the same relation) and elapsed time for the transaction that carried out as a new attempt in each separate case. As it can be seen, performing transactions in each and every attempt contributes to some time reducing.

Fig. 5 (b) shows time obtained when the client cache is empty starting from the first attempt. Since, there is no need for a time required for client cache update in the applications carried out without cache use, the first application time here is lower. However, as

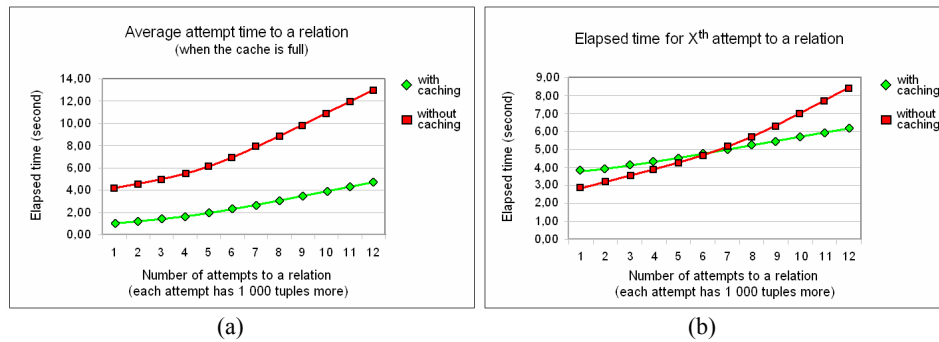


Fig 5. (a) Average attempt time to a relation (when the cache is full); (b) elapsed time for x^{th} attempt to a relation.

the number of attempts increase, the difference between them is lower and the time required for the applications performed with cache use reduces after the 6th attempt. As the number of attempts increase, the time loss decreases.

Therefore the time required to obtain the result set in the query applications will be decreased with the use of cache, after a certain time through which the caches will be filled with similar queries.

6. CONCLUSION AND FUTURE WORKS

It is known that there have been attempts to obtain time and traffic cutting by using predicate-based caching method employed in RDBMS on client-server structures. In this study, we tried to introduce a solution-proposal to the client cache update problem encountered during the usage of SCD by employing CCDT. Predicate-based caching method was improved using MNTC manager, MNTC patches and TPCF in order to keep client caches up-to-dated through utilizing idle time of clients. In addition, a simple TPCF formula is presented to predict the idle time of clients. Although, the experiments showed that calculation of idle time of client is not easy, usage of MNTC helps the clients to update their updates (such as insert, delete and update) waiting for commit in case of that TPCF calculation is hard. Furthermore, it is determined that presented method is more useful when the clients' cache is not empty and when the number of attempts on the same query more than 5. Finally, with the usage of presented method, many benefits like time saving and reduction in traffic can be acquired especially in environments which are having intensive transactions.

As a further work on the TPCF formulation can be considered. In addition to this, an improvement may be made in the coordination the clients by improving attributes on CCDT. More, a new method can be investigated to add Join queries to CCDT structure. Additionally some other works may be carried out on CCDT, MNTC Manager and TPCF in order to make system consistency better.

REFERENCES

1. C. Mohan, "Application servers and associated technologies," in *Proceedings of the 28th International Conference on Very Large Data Bases*, http://www.almaden.ibm.com/u/mohan/AppServersTutorialVLDB2002_Slides.pdf, 2002.
2. *Oracle 7 Server Concepts Manual*, Oracle Corporation, 1992.
3. Y. Wang and L. A. Rowe, "Cache consistency and concurrency control in a client-server DBMS architecture," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991, pp. 367-377.
4. W. K. Wilkinson and M. A. Neimat, "Maintaining consistency of client-cached data," in *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990, pp. 122-133.
5. A. M. Keller and J. Basu, "A predicate-based caching scheme for client-server database architectures," *The VLDB Journal*, Vol. 5, 1996, pp. 35-47.
6. K. P. Esparan, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communication of the ACM*, Vol. 19, 1976, pp. 624-633.
7. J. Gray and A. Reuter, "Isolation concepts," *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 403-406.
8. J. R. Jordan, J. Banerjee, and R. B. Batman, "Precision locks," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1981, pp. 143-147.
9. P. A. Larson and H. Z. Yang, "Computing queries from derived relations: theoretical foundation," Research Report No. CS-87-35, Computer Science Department, University of Waterloo, CA, 1987.
10. Y. Sagiv and M. Yannakakis, "Equivalences among relational expressions with the union and difference operators," *Journal of ACM*, Vol. 27, 1980, pp. 633-655.
11. J. A. Blakeley, P. A. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1986, pp. 61-71.
12. S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991, pp. 577-589.
13. A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993, pp. 157-166.
14. J. A. Blakeley, N. Coburn, and P. A. Larson, "Updating derived relations: detecting irrelevant and autonomously computable updates," *ACM Transactions on Database System*, Vol. 14, 1989, pp. 369-400.
15. M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data caching tradeoffs in client-server DBMS architecture," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991, pp. 357-366.
16. M. J. Franklin, M. J. Carey, and M. Livny, "Local disk caching in client server database systems," in *Proceedings of the 19th International Conference on Very Large Data Bases*, 1993, pp. 641-655.
17. D. Lomet, "Private locking and distributed cache management," in *Proceedings of*

- the 3rd International Conference on Parallel and Distributed Information Systems*, 1994, pp. 151-159.
18. N. Roussopoulos and H. Kang, "Preliminary design of ADMS±: a workstation-mainframe integrated architecture for database management systems," in *Proceedings of the 12th International Conference on Very Large Data Bases*, 1986, pp. 355-364.
 19. A. Delis and N. Roussopoulos, "Performance and scalability of client server database architectures," in *Proceedings of the 18th International Conference on Very Large Data Bases*, 1992, pp. 610-623.
 20. T. K. Sellis, "Intelligent caching and indexing techniques for relational database systems," Technical Report No. CS-TR-1927, Computer Science Department, University of Maryland, College Park, U.S.A., 1987.
 21. N. Kamel and R. King, "Intelligent database caching through the use of page-answers and page-traces," *ACM Transactions on Database Systems*, Vol. 17, 1992, pp. 601-646.
 22. A. Khalil and P. Sanghyun, "Scalable template-based query containment checking for web semantic caches," in *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003, pp. 493-504.
 23. J. Fernandez, A. Fernandez, and J. Pazos, "Optimizing web services performance using caching," in *Proceedings of the International Conference on Next Generation Web Services Practice*, 2005, pp. 157-162.
 24. Q. Ren, M. H. Dunham, and V. Kumar, "Semantic caching and query processing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, 2003, pp. 192-210.
 25. P. Bodorik, D. Jutla, and Y. Lu, "Interoperable server-based cache consistency algorithm," in *Proceedings of the International Database Engineering and Applications Symposium*, 2004, pp. 312-231.
 26. B. Y. Chan, A. Si, and H. V. Leong, "A framework for cache management for mobile databases: design and evaluation," *Distributed and Parallel Databases*, Vol. 10, 2001, pp. 23-57.
 27. B. Signer, A. Erni, and M. C. Norrie, "A personal assistant for Web database caching," in *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, LNCS 1789, 2000, pp. 64-78.

Yasar-Guneri Sahin received his Ph.D. degree in Technology Education, M.S. degree in Computer Education from Gazi University, Ankara, Turkey, in 2005 and 1999, respectively, and his B.S. degree in Computer Engineering from Middle East Technical University, Ankara, Turkey, in 1994. He has been an Assistant Professor of Computer Engineering Department, Yasar University, Izmir, Turkey, since 2005. He has worked as General Manager in Yagusa Computer Co. for 10 years, and developed many government software projects. His research interests include database management systems, distance education, software engineering, and programming languages.

Halil-Ibrahim Bulbul received his Ph.D. degree in Educational Technology from Ankara University Ankara, Turkey, M.S. degree in Technology education from California University of PA, U.S.A., in 1997 and 1990, respectively, and his B.S. degree in

Technology Education from Gazi University, Ankara, Turkey, in 1985. He has been an Assistant Professor of Computer Education Department, Gazi University, Ankara, Turkey, since 1997. His research interests include educational technologies, e-learning, distance education, educational software design, and database management systems.