Adaptive Database Caching with DBCache

C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Contact email: {mohan, maltinel}@almaden.ibm.com

Abstract

Caching as a means to improve scalability, throughput, and response time for Web applications has received a lot of attention. Caching can take place at different levels in a multi-tier architecture. However, due to the increased personalization and dynamism in Web content, caching at the bottom of the multi-tier architecture – the database level – can be beneficial to all upper levels because of better reusability of the cached data. DBCache supports database caching at both the edge of content delivery networks and at the middle-tier of Web site infrastructures. Its on-demand caching functionality provides the key feature for adaptive and transparent data caching for Web applications.

1 Introduction

Modern e-commerce Web sites, due to their transactional nature and increased personalization, are database driven. As shown in Figure 1, they are typically built on top of a broad range of middleware technologies including network load balancers, Web and application servers, and database servers. System design for a high volume Web site is challenging not only because it involves a variety of different technologies but also because of its performance requirements for peak access times. Scalability can be increased by replicating application server nodes in the middle tier. However, with this design, the backend database tends to become the bottleneck soon. Thus, applying caching solutions for high-volume Web sites to improve scalability, response time, and reliability has received a lot of attention in the Web and the database communities.

DBCache [1] supports the creation of persistent database caches based on DB2's federated query processing capabilities. DBCache allows the specification of *cache tables* that either cache the entire content or a subset of the tables from the backend database server. Federated database capabilities, particularly nicknames, are used to access the backend database. If a query cannot be answered from the cache, it is automatically routed to the backend database. The names of the cache tables can be the same as the backend tables, in which case no changes to existing application queries are required to make use of the cache.

DBCache provides two types of cache tables. *Declarative* cache tables are populated at definition time. They are implemented as user-maintained materialized views. Changes at the backend database are applied to the cache by using a data replication tool. Query routing is done at query compilation time using DB2's materialized view mechanism [9]. *Dynamic* cache tables, on the other hand, are populated at run-time based on the queries issued by the application. Changes at the backend database are detected and applied to the cache based on the respective cache content. Since the content may change over time, query routing is done at query execution time, to take the current content of the cache into consideration. In this paper, we'll focus on dynamic

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

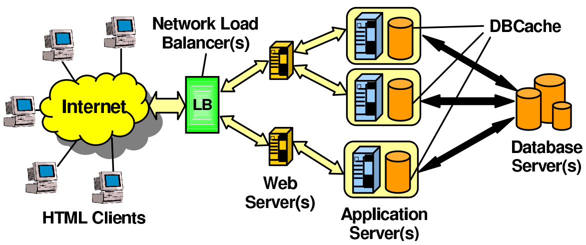

Figure 1: Database Caching with DBCache

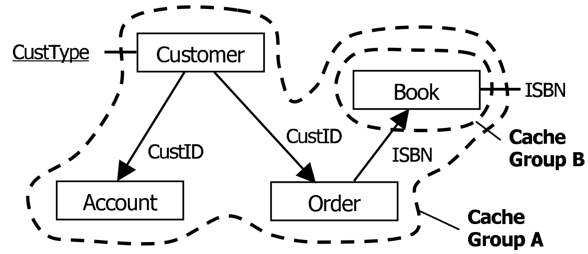

Figure 2: Cache Federation Example

caching and discuss as our main contributions: (1) a model to define a dynamic cache, (2) query plan generation for dynamic caching, and (3) asynchronous mechanisms for cache population and maintenance.

2 The Caching Landscape

Existing caching approaches for Web applications can be broadly classified as Web caching and database caching solutions. In the case of Web caching [7], Web objects such as, entire HTTP pages, page fragments, or JSP results are cached in the Web container of the application servers, the Web servers, a proxy cache, or an existing edge server. With database caching, data from the database server is cached in database caches closer to the application server to offload the backend database server. Even that reuse of complex Web objects may save some processing cost, with increased dynamism and personalization of modern Web sites, database caching offers higher reusability since the same base rows might be used for the generation of different Web objects.

The database cache itself may be implemented in different ways. With Semantic Caching [4] and DBProxy [2] actual query results are cached and new queries are checked to see if they can be satisfied from the cache alone. Query result caching has been proven efficient for complex OLAP type of queries. Another approach is the caching of parts or whole backend tables in a full-fledged database server. Microsoft's MTCache [5] and our previous work in DBCache [6] allow backend-table caching based on explicitly specified materialized views. Unfortunately, it is often impossible to know a priory exactly which table subset to cache because of the dynamic nature of Web applications (e.g., which are the "hot" items from the product catalog?). Thus, with DBCache we allow the specification of dynamic cache tables that cache data based on incoming user queries. In addition, similar to the caching approach in TimesTen [8], we support the declaration of related tables for cache population and maintenance. This way, DBCache offers flexible and adaptive database caching for dynamically changing user access patterns.

3 DBCache Dynamic Caching Model

We have to guarantee that the result of a query using the cache is the same as if it was executed against the backend database[1]. For this, the cache system has to make sure that the complete set of rows required to satisfy a given SQL predicate is completely in the cache. Our current prototype supports the evaluation of equality and equi-join predicates from the cache. We describe the possible content of dynamic cache tables, i.e., the predicates whose extensions should be kept in the cache, by using two types of cache constraints: referential cache constraints and cache keys.

For the evaluation of equality predicates from the cache, domain completeness of the corresponding cache table column is required. *Domain completeness* guarantees that for a given value of the column, the cache table contains all the rows from the backend table with the same value. Note that unique columns are domain-complete

[1]The query result may be slightly different due to the cache being slightly out-of-date (cache latency).

per definition. For non-unique columns, domain completeness needs to be enforced by the caching system. For a single cache table *CT*, domain completeness of a column *c* allows for the probing and evaluation of equality predicates of the form *CT.c = v*. If *v* is found in the cache it is guaranteed that the extension of this predicate is completely in the cache.

To support equi-joins in the cache, information is needed that tells us that if one cache table can be used in the join than the other table can be used too. For this purpose, we introduce the concept of a *referential cache constraint (RCC)*. An RCC defines a cache-parent to cache-child relationship between two join-compatible cache table columns. RCCs are enforced by the caching system. An RCC $CT_i.c_i \rightarrow CT_j.c_j$ guarantees that, when a row *r* with value *v* in c_i is in the cache, then all join partners of *r* via $CT_i.c_i = CT_j.c_j$ are also in the cache. Thus, RCCs allow the evaluation of predicates of the form ($CT_i.c_k = v$ and $CT_i.c_i = CT_j.c_j$), if c_k is domain complete.

The second type of cache constraint, the so-called cache keys, specify how cache tables are populated. Declaring a cache table column as a cache key assigns the domain completeness property to it. In addition, a cache key specifies a filling point for the cache table and, via (a chain of) RCCs, for other cache tables related to it. When a user query in an equality predicate refers a cache key value *v* which is currently not in the cache, evaluation of this predicate has to be done at the backend. As a consequence of such a cache miss, the caching system brings in all required rows for *v* from the backend database. The assumption here is that because of locality of reference a given cache key value indicates the likely reuse of the inserted rows.

A set of related cache tables which are (directly or via RCCs) populated by the values of one or more cache keys of a single cache table is called a *cache group*. A cache group can be understood as the representation of the processing context of one or more query types. Multiple cache groups may (partially) overlap in one or more cache tables thereby establishing a so-called *cache group federation*. Figure 2 shows an example of a federation consisting of two overlapping cache groups. Cache group A is filled through cache key *CustType* and contains all cache tables depicted. Cache group B is filled through *ISBN* and contains only the *Book* cache table.

Unrestricted use of cache constraints may result in performance and scalability problems because of cache population and cache maintenance overhead. Thus, we provide the following design rules and recommendation. Excessive cache population occurs when we have recurring load operations for the same cache table, which in the extreme case may result in caching the entire backend table. Obviously, cycles created by RCC definitions[2] in cache groups may pose this *recursive cache load* problem. We distinguish two types of cycles. In a *homogeneous* cycle for each participating table only one column is involved in the RCC cycle. A *heterogeneous* cycle is formed if one of the tables contains two columns used in the cycle. Homogeneous cycles are safe w.r.t. the recursive cache load problem. This is because starting with a given set of values a traversal does not require additional values for the participating columns. By the same argument, heterogeneous cycles are not safe since tables with two columns involved may require additional values for the column we started off from and, in turn, enforce an additional circulation. Therefore, as our first design rule, we do not allow heterogeneous cycles in cache group definitions.

In our cache model, a cache table column becomes domain complete if: (1) it is a unique (or primary key) column, (2) a cache key is defined for it, (3) it is involved in a homogeneous cycle[3], or (4) it is the target of the only incoming RCC and its table does not contain a cache key defined on another column (aka *induced domain completeness*). Enforcing domain-completeness for non-unique columns may lead to recursive cache population. Let's assume a cache table with two non-unique domain complete columns c_1 and c_2. If we insert rows for a new value for c_1, to ensure domain completeness of c_2 we need to insert more rows for each new value in c_2, which in turn may require to bring in new rows to ensure domain completeness of c_1 and so forth. Thus, as our second design rule, we allow at most one non-unique domain-complete column per cache table.

[2]We define a cycle as a path from a cache table back to the same table by following a set of RCCs. At this point, each participating table must only be visited once.

[3]For the initiating column, i.e., the column through which the cycle is populated domain completeness is enforced by the caching system. For all other columns in the cycle induced domain completeness applies.

However, we pose no limit on the number of unique domain-complete columns.

Low-selectivity cache key columns (e.g., gender) can trigger the loading of large amounts of data that is never used later. Thus, the use of high-selectivity cache keys (e.g., customer ID) is recommended which allows a more precise population of the cache.

4 Janus Query Plans

Since the content of dynamic cache tables is not described by any predicate values and may change between subsequent executions of the same query, the decision of whether to route the query to the cache or the backend database must be made at runtime. For this, a special kind of query execution plan called *Janus* plan is prepared at query compilation time. A Janus plan consists of two query plan alternatives. The *local* plan refers to all dynamic cache tables that, dependent on the current content, might be usable for query execution. For tables for which no cache table exists, the local plan refers via a nickname to the corresponding backend table.[4] The *remote* plan, on the other hand, uses only nicknames to enable execution of the query at the backend database. Both query plans are tied together with a conditional switch operator which contains a simple subquery, called the *probe* query. Usability of cache tables in the probe and the local query plan is decided based on cache constraints. Details are given in [1].

Execution of a Janus plan is done by first executing the probe query which performs a simple existence check for the cache tables referred to in the local query to find out whether or not the local plan can be used to answer the given query. If the cache contains the necessary data the local query plan is executed, if not we fall back to the remote query plan. The probe query is constructed based on the equality predicates on domain complete columns as given in the user query. The property of domain completeness, as introduced in Section 3, ensures that the probe is sufficient to guarantee correctness – if a single record of the probe query result is found in the cache, it is guaranteed that it is safe to execute the local query plan.

The probe query is created as a scalar subquery and its results can potentially be reused by the local query. Thus, as shown in [1], the extra overhead of the probe query is relatively low and justified by the benefits of executing a local as opposed to a remote query.

5 Cache Population

Each execution of the remote query in a Janus plan is considered a cache miss, and the cache key values that failed the probe query are used for on-demand cache population by a dedicated background process called the DBCache cache daemon.[5] Cache miss information is passed on to the cache daemon by sending an internal message as part of remote query plan execution. However, cache tables are not populated as part of the remote query. This is because cache constraints may require the loading of a (relatively) large amount of additional data into a number of cache tables; also it would implicitly turn the given user query into an insert statement. Both would significantly increase execution time and change the semantics of the original user query.

For each cache miss message, the daemon generates (at most) one insert statement per cache table in the affected cache group and then executes these statements in a single database transaction. As a first step for statement generation, the set of *qualifying rows* for each cache table, i.e., the rows that need to be brought in because of defined cache constraints. Qualifying rows for a cache table are represented as (nested) subqueries that refer to the corresponding backend tables, is determined. Qualifying rows for the root table CT_0 of the cache group are determined by the given cache key value and cache keys defined on CT_0. As we follow RCCs $CT_i.c_{from} \rightarrow CT_{i+1}.c_{to}$ the qualifying rows subquery for CT_i is used as the starting point to determine the

[4]Note that this may result in a distributed executed of the query using DB2's federated database functionality.

[5]Here the execution of application-specific policies is possible, e.g., "only start caching if we have seen the same cache miss n times in the last m seconds".

```
INSERT INTO UserSchema.Customer
SELECT * FROM NN.Customer
WHERE CustType = 'gold'

INSERT INTO UserSchema.Account
SELECT * FROM NN.Account
WHERE CustID IN (SELECT CustID
                 FROM NN.Customer
                 WHERE CustType = 'gold')

INSERT INTO UserSchema.Order
SELECT * FROM NN.Order
WHERE CustID IN (SELECT CustID
                 FROM NN.Customer
                 WHERE CustType = 'gold')
  AND (OrdID) NOT IN (SELECT OrdID FROM
                      UserSchema.Order)

INSERT INTO UserSchema.Book
SELECT * FROM NN.Book
WHERE ISBN IN (SELECT ISBN FROM NN.Order
               WHERE CustID IN
                 (SELECT CustID
                  FROM NN.Customer
                  WHERE CustType = 'gold'))
```

Figure 3: Insert Statement Generation

```
DELETE FROM UserSchema.Order
WHERE CustID IN (SELECT CustID
                 FROM UserSchema.Customer
                 WHERE CustID = 'cid:500')
  AND (ISBN) NOT IN (SELECT ISBN
                     FROM UserSchema.Book)


DELETE FROM UserSchema.Account
WHERE CustID IN (SELECT CustID
                 FROM UserSchema.Customer
                 WHERE CustID = 'cid:500')


DELETE FROM UserSchema.Customer
WHERE CustID = 'cid:500'
```

Figure 4: Delete Statement Generation

qualifying rows for CT_{i+1}. The final insert statement for each cache table is based on the qualifying rows for the respective table and results in a federated statement. This way, the daemon does not have to deal with intermediary query results, i.e., actual rows. For our example from Figure 2, a cache miss for *CustType = "gold"* would result in the insert statements shown in Figure 3.

Cache miss messages need not be persistent. If a pending miss message is lost, another miss message for the same cache key value is sent. We require a primary key for each cache table. Thus, if a cache miss message is sent multiple times (i.e., because a second request for the same cache key occurred before the first cache miss was processed) the resulting primary key constraint violation causes the daemon to skip the second message.

6 Cache Maintenance

The content of dynamic cache tables needs to be updated as the corresponding tables at the backend database are updated. Changes at the backend are detected by the capture program of IBM's data replication tool which sends according maintenance messages to the DBCache cache daemon. All changes from a transaction at the backend are sent as one message to preserve transaction boundaries. Also, since losing maintenance messages might result in inconsistent cache content, messages from the replication tool have to be stored persistently.

Upon receiving a maintenance message, the daemon generates appropriate insert, update and delete statements and issues them against the cache database. The algorithms used to generate these statements are similar to the one described for cache population in Section 5. All changes of a transaction at the backend database are applied to the cache within the same database transaction. Note that changes at the backend tables might be relevant for the cache database or not based on the current cache content. In our existing prototype, filtering out irrelevant changes is done by the daemon during statement execution. In particular, the following four cache maintenance tasks are performed by the daemon.

Insert at the backend database: A new row has to be inserted into the cache only if (a) for any value of a column that is the target of an incoming RCC we have the corresponding cache parent row(s) in the cache, or if (b) for a non-unique cache key of the target table we already have rows with the same key value in that table. Otherwise, the insert at the backend database is irrelevant for the cache database. This way, we only bring in new cache key values based on an insert at the backend if existing cache constraints require us to do so. If the daemon tries to insert an already existing row, (e.g., because a cache miss message overtook a cache maintenance message[6]), the resulting primary key violation causes the daemon to skip the maintenance message. The insert statement for the root table is based on the insert at the backend database and the defined cache constraints. For all other cache tables statement generation is done exactly as for cache population.

Update at the Backend Database. For update operations we distinguish two cases. (a) If the update is only on columns that are neither the source or target of an RCC nor cache key columns we can do the update by simply updating the corresponding row in the cache table. (b) If the update is on at least one cache key or RCC column we might have to delete and/or insert other rows from/into the cache in addition to the actual update. For the latter case, we can translate the update into a delete of the old row, using the algorithm for backend deletes, followed by an insert of the updated row, using the algorithm for backend inserts. If an update message is received by the daemon before the affected row was brought into the cache (because of a cache miss) the update message is discarded. This is not a problem since at the time the corresponding cache miss message is processed the latest version (i.e., the one reflecting the update seen before) of the row is fetched.

Deletion from the Backend Database. If a cached row is deleted from the backend database, we also have to remove this row from the corresponding cache table. In the case of a cache parent row, we actually do not need to delete all corresponding cache children to preserve cache consistency. However, in spite of this, we also delete cache children if (a) the cache child table does not have a cache key, and (b) the cache children are not shared by a cache parent of the same or another cache group. Since garbage collection is done through cache keys, these rows would become unreachable for later garbage collection. If a delete message is seen by the daemon before the corresponding cache miss message that would bring in the affected row, the delete message will have no effect. This does not cause a problem since at the time the cache miss message is processed the deleted row would simply not be found at the backend database. Similar to statement generation for inserts, the main idea behind delete statement generation is to generate one delete statement per cache table, and then to execute these statements in a single database transaction. The delete statement for a given cache table is generated based on the non-unique cache keys defined on this table and the qualifying rows identified by the delete message or the qualifying rows of its cache parent tables. For our example from Figure 2, the deletion of customer row *CustID = "cid:500"* would result in the delete statements shown in Figure 4. Note that we do not generate a delete statement for the *Book* cache table since it has a cache key defined.

Garbage Collection. Periodic garbage collection done by the cache daemon directly deletes rows from the cache based on cache key values determined to be "good" candidates for eviction. The purpose is to delete entire cache group instances ("trees" of related rows) to free up caching space. Identification of candidates for eviction can be based on popularity (i.e., how many cache hits did occur for this cache group instance), recency (i.e., how long has the cache group instance been in the cache), and space consumption (i.e., how big is the cache group instance). Statement generation for garbage collection is done in a very similar way as for backend deletes with only the following two differences. (a) In case of the eviction from a root table that has a non-unique cache key, we have to delete *all* rows that have the corresponding cache key value to conserve domain completeness. This is true even if the eviction is not for a value of the non-unique cache key. (b) For garbage collection and backend deletes alike, if deletion of a cache parent is required we also delete all corresponding cache children that are not referred to by a cache parent of another cache group. For garbage collection, however, the only exception is if the cache group that shares child rows is fully contained in the group for which we perform deletes. In this case we also delete the shared rows which results in a more aggressive deletion for garbage collection. From the cache daemon's point of view, garbage collection corresponds to backend deletes. Thus, conflicts based on different execution orders of different maintenance operations are not an issue for the reasons given in the previous paragraphs.

[6] In the current prototype all incoming messages for the same cache federation are processed by the daemon in sequential order to guarantee cache consistency and to reduce the likelihood of deadlocks.

7 Open Research Issues

In this section we will identify open research issues in the area of dynamic database caching. Although the issues are discussed in a DBCache-specific context, they pose problems relevant to constraint-based sub-table caching in general.

Cache Maintenance – Update Filtering. Whether a given update at the backend database is relevant for a particular cache depends on the current cache content. To determine which changes at the backend need to be applied to which caches, we have considered two architectural alternatives. (a) In the first case, capture sends a maintenance message for every update at the backend database to every frontend cache, and the respective cache daemon determines which operations to be apply to the cache database[7]. Since this approach, in general, can introduce significant message traffic between the backend and the cache database, it might only be applicable to applications with very low update ratio, or to settings with only a few frontend caches and with high network bandwidths. (b) In the second case, determining which update operations need to be applied at which cache is done at the backend database. Thus, only relevant maintenance messages are sent from the backend to the affected caches and no additional filtering at the frontends is required. This approach requires a complete representation of cache constraints and keeping close track of the cache content at the backend database for every cache served. It might be beneficial for applications with higher update ratio by significantly reducing the message traffic. However, it requires additional processing at the backend database hampering our goal of unburdening the backend server. Although for a complete implementation an adaptive solution that shifts filtering between cache and backend database nodes is desirable, the following two main questions remain open. First, how can we organize and maintain information at the backend about the current cache contents without overwhelming the backend server? Second, based on a given update ratio, cache sizes, and number of caches, which of the two alternatives is superior?

Local Updates. In the initial prototype, all update queries are executed at the backend database and are applied indirectly to the cache through the cache daemon. As a result, applications do not see the effects of their updates within the same transaction. This may not be acceptable for certain classes of applications. We distinguish three update modes: (a) direct update of backend database, indirect update of local cache (via replication tool), (b) direct update of backend and local cache (requires 2PC support), and (c) direct update of local cache, indirect update of backend database (via replication tool). A database caching system has to offer these different update options so that applications can choose the best fit for their update semantics. Moreover, there is a need for new mechanisms for detecting whether an update operation is relevant to the local cache. There is also a need to change probing logic in the Janus plans. The reason is that due to effects of update operations (i.e. deleting a row), checking content of cache tables in the probe may select the wrong plan option. Hence, applications may end up not seeing the effects of their updates. One solution to this problem is to probe some form of metadata instead of actual table content. Open questions in this case are, what kind of metadata would be needed to represent the cache content? How this information can be kept up-to-date with respect to dynamically changing cache content? What kind of race conditions may arise and how can they be avoided?

Garbage Collection. To preserve cache consistency, garbage collection for DBCache has to be done at cache group instance level and not at the level of individual rows. This poses two major challenges. (a) Cached objects, i.e., cache group instances, are of varying size; and (b) different from most Web object caching scenarios, dependencies between cached objects (in the form of overlapping cache groups) have to be taken into consideration. For different object sizes we can retreat to existing function-based strategies as used for Web object caching [7]. For overlapping cache groups expensive filtering is required to prevent deletion of rows shared by multiple cache group instances. Main open issues for garbage collection in the context of cache groups are:

[7] Our current prototype follows this approach.

First, since eviction is based on statistics like number of cache hits or size of cache group instances, what would be a reasonable trade-off between accuracy and cost for collecting necessary statistics? Second, in the presence of overlapping cache groups and cache maintenance operations, how can we efficiently attribute cache hits (i.e., the use of rows from the cache to answer a given query) to cache group instances?

Tooling. Efficient use of constraint-based database caching requires the provision of powerful tools to support cache setup and adaptation. We envision the development of a cache advisor tool to propose and automatically setup cache tables and cache constraints based on a given query workload. If no such workload can be provided, the tool can still automate the process by taking hints from the backend database schema, i.e., primary keys and referential integrity constraints can be a good starting point for selecting cache keys and RCCs. In addition, tooling is required to adapt a given cache setup at run-time in response to changed user behavior by adding or dropping cache elements. Close monitoring of cache performance is required to support well informed manual corrections. Open issues in this area include the question for appropriate cost models to estimate cache population/maintenance cost vs. benefit for different cache setup alternatives. These models form the basis for automatic decisions on cache setup, and thus on-demand database caching.

8 Summary

DBCache dynamic cache tables provide adaptive on-demand database caching for high-volume e-commerce Web sites. In this paper, we briefly discussed our main contributions: (1) a model to define dynamic caches, (2) query plan generation for dynamic caching, and (3) asynchronous mechanisms for cache population and maintenance. For a more detailed discussion of the dynamic cache model, and DBCache query plan generation see [1]. We would like to thank Nesime Tabul, Prakash Linga, Sailesh Krishnamurthy, Bruce Lindsay, Dan Wolfson, and Theo Haerder for their contributions to the DBCache project, and the IBM SWG for many fruitful discussions.

References

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: *Cache Tables: Paving the Way for an Adaptive Database Cache*, VLDB'03, Berlin, Germany, Sep. 2003

[2] K. Amiri, S. Park, R. Tewari, S. Padmanabhan: *DBProxy: A Dynamic Data Cache for Web Applications*, ICDE'03, Bangalore, India, Mar. 2003

[3] C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: *DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures*, (Demo Description), SIGMOD'03, San Diego, CA, Jun. 2003

[4] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, M. Tan: *Semantic Data Caching and Replacement*, VLDB'96, Mumbai (Bombay), India, Sep. 1996

[5] P. Larson, J. Goldstein, J. Zhou: *Transparent Mid-tier Database Caching in SQL Server*, (Demo Description), SIGMOD'03, San Diego, CA, Jun. 2003

[6] Q. Luo, S. Krishnamurthy, C. Mohan, H. Woo, H. Pirahesh, B. Lindsay, J. Naughton: *Middle-tier Database Caching for e-Business*, SIGMOD'02, Madison, WI, Jun. 2002

[7] S. Podlipnig, L. Böszörmenyi: *A Survey of Web Cache Replacement Strategies*, ACM Computing Surveys, Vol. 35, No. 4, Dec. 2003, pp. 374-398

[8] The TimesTen Team: *Mid-tier Caching: The FrontTier Approach*, SIGMOD'02, Madison, WI, Jun. 2002

[9] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata: *Answering Complex SQL Queries Using Automatic Tables*, SIGMOD'00, Philadelphia, PA, May 2000