

# Cache Tables: Paving the Way for an Adaptive Database Cache

Mehmet Altinel<sup>1</sup> Christof Bornhövd<sup>1</sup> Sailesh Krishnamurthy<sup>2</sup> C. Mohan<sup>1</sup> Hamid Pirahesh<sup>1</sup>  
Berthold Reinwald<sup>1</sup>

<sup>1</sup> IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

<sup>2</sup> Comp. Sci. Div, Dept EECS, UC Berkeley, Berkeley, CA 94720

Contact email: {mohan,maltinel,cborn}@almaden.ibm.com

## Abstract

We introduce a new database object called *Cache Table* that enables persistent caching of the full or partial content of a remote database table. The content of a cache table is either defined *declaratively* and populated in advance at setup time, or determined *dynamically* and populated on demand at query execution time. Dynamic cache tables exploit the characteristics of typical transactional web applications with a high volume of short transactions, simple equality predicates, and 3-4 way joins. Based on federated query processing capabilities, we developed a set of new technologies for database caching: cache tables, "Janus" (two-headed) query execution plans, cache constraints, and asynchronous cache population methods. Our solution supports *transparent* caching both at the edge of content-delivery networks and in the middle-tier of an enterprise application infrastructure, improving the response time, throughput and scalability of transactional web applications.

## 1. Introduction

Transactional Web Applications (TWAs) have reached widespread use in modern enterprise application infrastructures [22]. Such applications are typically implemented with a broad range of technologies including network load balancers, HTTP servers, application servers, transaction-processing monitors and databases. In its simplest form, a TWA is realized with an HTTP server hosting presentation logic, and an application server hosting business logic (in the form of Java servlets or EJBs) that in turn obtains data by issuing queries to a relational database. Figure 1 depicts an example enterprise application configuration.

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment*

Proceedings of the 29<sup>th</sup> VLDB Conference,  
Berlin, Germany, 2003

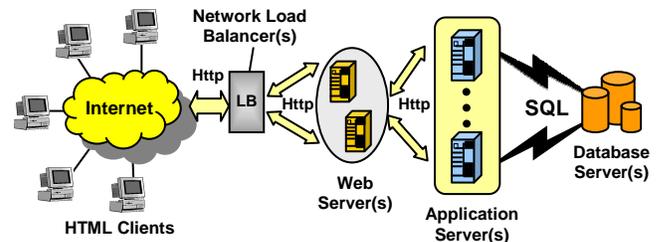


Figure 1: Typical Multi-Tier Enterprise Architecture

### 1.1 The Cache Jam

The various layers of the application infrastructure stack hurt its response time and scalability. The last few years have seen the wide use of caching static HTML pages and data as a technique to achieve better response time and scalability of interactive TWAs. Caching takes place at various stages: the cache of a client browser, forward and reverse proxy caches, nodes of content-delivery overlay networks, and in specialized object caches that are part of application business logic [21].

However, as TWAs get more dynamic with increased personalization and the need to deliver frequently updated information, such static caching techniques become less useful. High-volume web sites often serve highly personalized content to their users. As a consequence, the data they need to build web pages is very dynamic in nature and often cannot be profitably cached far away from the enterprise servers. For this reason, some enterprise applications run their business logic in the application server nodes deployed in remote data centers close to end users (these are also called "Web Hosting" services). What's more, the partnership between content-delivery network service providers (e.g., Akamai [2]) and application server vendors (such as IBM and BEA) will make it easier for companies to move more content and applications out of origin servers, thereby improving response time and reducing demands on in-house systems. The benefit of these approaches, however, will always be limited when the remote application servers try to access needed data from central backend databases.

Database caching is a promising technique to address the dynamic nature of TWAs [10][21]. Data stored in a database cache is accessed by the application using database queries in just the same way the backend

database is accessed. The database cache entity may itself be implemented in different ways. Semantic caching [9] and DBProxy [4] are approaches where results of queries are preserved in the cache and new queries are checked to see if they can be satisfied with the local data alone. In contrast, [23] and [24] describe systems where a full-fledged database server is co-located with the application server. One advantage of the latter approach is that a significant portion of logic (query parsing and analysis) that already exists in full-fledged systems can be exploited for managing the cache. Plus, this approach also allows caching other associated database objects, such as triggers, constraints, indices, stored procedures and so on which can be important not only for application performance and semantics but also for providing continued service for applications when backend databases are unavailable. Our approach in the DBCache project falls into this category. We are prototyping a database cache feature that is incorporated into a full-fledged DBMS, namely DB2. In our research prototype, we aim to turn a regular DB2 instance into a *transparent* database cache manager by modifying the engine code and leveraging existing federated database functionality.

Our solution aims to support database caching not only at mid-tier nodes of central enterprise infrastructures as in [23] and [24], but also at remote data centers and edge servers of content delivery networks. Given that there might be potentially a large number of database cache deployments in the latter scenario (e.g., Akamai's network currently has nearly 15,000 edge caching servers), declarative way of specifying table subsets can easily make the whole system unmanageable. In our solution, the database cache addresses this key problem by *adapting* to the system load by automatically choosing the data to cache. It is also vital that applications should be able to work seamlessly with database caches and not require any change in their existing queries.

Our on-demand caching solution exploits the typical characteristics of TWA queries. Our analysis of TWA query workloads (based on the Trade2 [18] and ECDW [3][19] benchmarks) shows that *equality predicates* on key or non-key columns (sometimes mixed with range predicates in the same query) and 3-4 way join queries are very common. In contrast, earlier work on semantic caching is best suited for range queries with no joins.

## 1.2 Database Cache Design Space

The easiest approach to implement a database cache would be to replicate the entire content of selected tables from the backend database. In this case, each cache table referred to in a query can be used without any further checking as long as stale data is acceptable. The simplicity of this approach attracted various database cache products including Oracle 9i Internet Application Server [23] and our earlier work in the DBCache project [3][19]. But, typically, front-end systems are much less powerful than backend systems making full-table caching difficult or

even sometimes impossible. Even for a powerful front-end system, large table sizes can easily make full-table caching infeasible due to increased replication and maintenance costs in the cache.

Sub-table caching, on the other hand, can provide an effective alternative by caching only the “interesting” parts of the backend tables. Materialized view technologies may seem a good match to implement a sub-table cache although they were developed for different purposes. In current database products, materialized views store precomputed query results, that are later used to speed up performance and data access of expensive queries [13][25].

Nicknames are references to remote tables that can be used in federated queries [15]. At first blush, it appears that no extra effort is needed to implement a sub-table cache by creating materialized views on nicknames. This way, existing materialized view matching mechanisms can be exploited to route queries to either cached tables (materialized views) or backend tables (nicknames) depending on query predicates. However, in a database cache this approach is too restrictive and ineffective for the following reasons:

- Materialized views require declarative specification. Once specified, the definition of materialized view content cannot change dynamically based on demand. Unfortunately, it is often impossible to know *a priori* exactly what to cache because of the dynamic nature of web applications (e.g., caching the content of a shopping cart or hot items of a product catalog in a typical e-commerce application).
- From the application viewpoint, cache tables must be semantically equivalent (i.e., peer level object) to the associated backend tables. One can argue that theoretically, a materialized view, derived from a single table, can inherit all the semantics (i.e., triggers, constraints, etc.) of its base table, and hence it can satisfy this requirement. However, as far as updates are concerned, cache tables and materialized views have clearly different semantics (our future vision of DBCache will include user updates performed directly in the cache database).

## 1.3 Our Contributions

We introduce the notion of a *cache table* and show how to use this new database object to build a seamless, adaptive database cache. Our caching scheme allows us to take advantage of DB2's sophisticated distributed query processing power for database caching. As a result, the optimizer may choose to execute a query either at the local database cache or the remote backend server, or more importantly, it can partition the query into subqueries that can be distributed amongst both databases.

We show how entire tables (ideal for rarely changing data) as well as subsets of tables can be cached. Our significant contributions are (1) a database cache model based on our cache table concept that supports transparent

caching of declaratively specified as well as dynamically changing subsets of remote database tables, (2) a novel query rewrite method to translate input queries into an appropriate form that can process dynamic subsets at runtime, (3) asynchronous mechanisms to load cache tables on demand and to keep them consistent with respect to the updates performed at the backend database.

We propose a runtime solution that involves building a special query plan with three parts: a *probe*, a *local query* and a *remote query*. When such a plan is run, the probe is always executed first, and its result dynamically determines whether to run the local or the remote query next. Since this plan has two operational parts, we call it a two-headed or a *Janus* plan. The condition checked by the probe is created by using a set of new *cache constraints* in our system, which let us guarantee the correctness of the results. While the remote query is constructed with only nicknames, the local query may include both nicknames and cache tables so that the query can be partitioned and then distributed among both databases by the optimizer.

The rest of the paper is organized as follows: We start with a definition of cache tables in Section 2. Section 3 presents cache constraints that are a key part of our dynamic cache model. In Section 4, we describe our query engine modifications that let us make use of cache tables. Section 5 explains the techniques developed for the population and maintenance of cache tables. Performance evaluation of dynamic cache tables is provided in Section 6. We present related work on database caching in Section 7, and concluding remarks in Section 8.

## 2. Cache Tables

A Cache Table is a database object by which an end user can specify that a table (*cache table*) in a database (*cache database*) is a cache of a table (*backend table*) in another database (*backend database*). Each cache table is associated with a nickname that represents the corresponding backend table. The name of a cache table is the same as the backend table name, which is the target of the query in the original configuration. If the content of the cache table cannot answer the query, we transparently route the query to its respective backend table through the nickname. In DBCache, we provide two types of cache tables: declarative and dynamic. Although query compilation and maintenance mechanisms differ depending on the type, we allow mixed settings in a cache database.

### 2.1 Schema Setup in a Cache Database

To achieve transparent deployment of a database cache, we make no changes to the database schema as viewed by the applications. Each backend table is represented in the cache database schema either with a cache table or a nickname depending on whether caching is enabled for the backend table. Figure 2 shows an example cache schema setup. The names of the schemas and their elements are

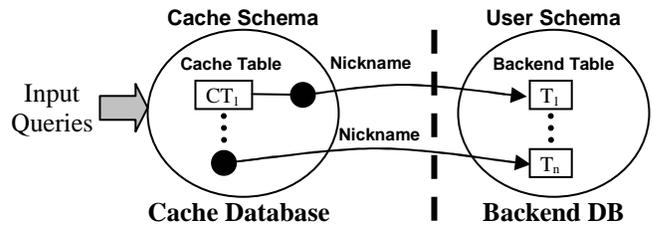


Figure 2: DBCache Schema Setup Example

created to be the same as their counterparts in the backend database<sup>1</sup>. Moreover, all the cache tables have exactly the same number and types of columns as those of their counterparts. As a result, the cache database schema resembles the actual user schema in the backend database, requiring no change in existing queries of the applications.

The main advantage of this setting is that for each cache table it allows us to easily cache other relevant logical and physical database objects associated with the backend table<sup>2</sup>. The cache database needs to have the logical objects (such as views, functions, constraints, stored procedures and so on) to be able to execute the queries locally, while the physical objects such as indexes are necessary for query performance.

### 2.2 Declarative Cache Tables

Declarative cache tables are useful when the desired content of the cache tables is known upfront. In this case, a declarative cache table is created with a predicate similar to a materialized view as shown in Figure 3. Note that when no predicate is provided, the entire table is cached from the backend database.

```
CREATE CACHE TABLE <Cache Table Name> AS
SELECT * FROM <Nickname Name>
WHERE <Predicate Definition>
```

Figure 3: Declarative Cache Table Creation

We implemented declarative cache tables by exploiting the existing materialized view support in DB2 [25] with some modifications. The details are given in Section 4.1.

### 2.3 Dynamic Cache Tables

Dynamic cache tables are populated on demand as dictated by the queries issued by the application. Hence, they eliminate the need for a DBA to specify what needs to be cached in cache tables. Instead, a dynamic cache table needs to be associated with a backend table through a nickname. The on-demand loading aspect provides a key feature needed for an adaptable cache. Moreover, dynamic cache tables can be used as a cornerstone to develop new

<sup>1</sup> Note that each cache table comes with an associated nickname whose name is uniquely generated by the cache system.

<sup>2</sup> Other database objects can easily be created for the cache tables by executing the same DDL statements that were used for the backend database.

methods to cache only “hot” items from backend databases. Figure 4 shows a DDL template to create a dynamic cache table.

```
CREATE CACHE TABLE <Cache Table Name>
FOR <Nickname Name>
```

**Figure 4:** Dynamic Cache Table Creation

Utilization of cache tables brings into play new challenges in their management and in query processing. In the following sections, we address these issues and describe our solutions. For declarative cache tables, we rely mostly on existing materialized view support in the DB2 engine. Hence, our main challenge in DBCache was to compile queries against dynamic cache tables and we have developed novel techniques for this purpose. Since our solutions are closely tied to representation of dynamic cache data in our design, we first explain the dynamic cache model and then show how we create query plans for cache tables.

### 3. Dynamic Cache Model

We describe the content of dynamic cache tables with cache constraint definitions. By observing cache constraints, we guarantee that the result of a query obtained by using the dynamic cache tables is the same as the result of the query if it were to be executed at the backend database (modulo the differences due to the cached data base being out-of-date). This property is our *correctness* principle in DBCache. Cache constraints help us to determine a set of dynamic cache tables that can be used in a query that satisfy the correctness principle. These tables are called *eligible* cache tables for the query.

There are two types of cache constraints enforced for dynamic cache tables: cache key constraints defined on columns of a cache table, and referential cache constraints that involve multiple cache tables. In the current prototype, we assume that DBAs specify these cache constraints. We plan to automate this task in the future.

#### 3.1 Cache Keys

A *cache key* is a cache table column whose values identify the records that are cached in the cache table. A cache key does not have to be unique; instead, all the values in a cache key column must be *domain complete*. This property guarantees that for any value of a domain-complete column, the cache table contains all the rows from the backend table that contain this value. Note that unique columns (hence, primary key columns as well) of a cache table are domain complete by definition. If a cache key is defined on a non-unique column, DBCache satisfies the domain completeness property for that column by fetching all required records from the backend and loading them into the cache table (details are provided in Section 5).

For a single cache table  $CT_i$ , the domain completeness property guarantees the correctness of *equality* predicates

in the form of “ $CT_i.c_x=value$ ”, where  $c_x$  is a domain-complete column. In this case, we can say that  $CT_i$  is an eligible table.

Cache keys are explicitly defined in the system and their definitions clearly state the intention to trigger on-demand loading for missing cache key values. Figure 5 shows a DDL template to create a cache key for a dynamic cache table.

```
ALTER TABLE <Dynamic Cache Table Name>
ADD CACHE KEY <Column Name>
```

**Figure 5:** Adding a Cache Key to a Dynamic Cache Table

#### 3.2 Referential Cache Constraints

We have developed another type of cache constraint that guarantees the correctness of equi-joins between cache tables. Basically, we define a new relationship type between cache tables that is specific to the cache database.

Using the domain completeness property, we can determine a set of eligible cache tables that could be used to answer queries involving single table equality predicates as described in the previous section. But if the query also includes other cache tables participating in equi-join predicates, how can we determine their eligibility? We need additional information that tells us that if one cache table is eligible in an equi-join then the other is too. We have developed the notion of referential cache constraints to address this requirement. From this perspective, they are semantically different from referential integrity constraints. Our on-demand loading mechanisms described in Section 5 always enforce the referential cache constraints while populating the cache tables. Hence, the cache system always guarantees to keep correlated values in the cache tables consistent.

A *Referential Cache Constraint* (RCC) can be defined between *any* columns of two cache tables depending on whether or not a join operation is possible between them. An RCC creates a cache-parent/cache-child relationship between two cache tables. When there is an RCC between a column  $c_n$  of a cache table  $CT_i$  (cache-parent) and column  $c_m$  of another cache table  $CT_j$  (cache-child), it indicates that for any value of  $c_n$  in  $CT_i$ ,  $CT_j$  includes all rows having that value in their  $c_m$  column. But for a row of  $CT_j$ , the associated row(s) in  $CT_i$  may or may not exist. Note that this parent-child relationship is completely cache specific in the sense that it is defined from the join processing point of view. To illustrate this, suppose there is an *equi-join* predicate  $T_i.c_n=T_j.c_m$ , where  $T_i$ ,  $T_j$  are backend tables whose cache tables are  $CT_i$ ,  $CT_j$  respectively, and  $CT_i$  is the cache-parent of  $CT_j$  through columns  $c_n$  and  $c_m$ . If we know that  $CT_i$  is an eligible cache table for use in a local query plan then so is  $CT_j$ . In short, it is safe to execute the join  $CT_i.c_n=CT_j.c_m$  in the cache database. Figure 6 shows an example DDL template to create RCCs in DBCache.

```

ALTER TABLE <Dynamic Cache Table1>
ADD CACHE REFERENCE FROM <Column Name1>
TO <Dynamic Cache Table2>(<Column Name2>)

```

**Figure 6:** DDL for Adding a Referential Cache Constraint

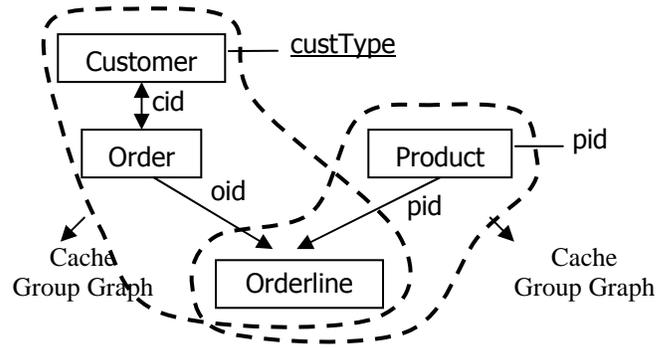
### 3.3 Cache Groups

In DBCache, we use the term *cache group* to identify a set of related cache tables whose content is (directly or transitively) populated by the values of one or more cache keys of a *single* cache table, called the *root table*. The tables in a cache group that are reachable from the root table via RCC constraints are called *member tables*. The cache group notion helps us organize the cache tables in a way that (1) we can recognize the application contexts more easily, and (2) as explained in the following section, we can detect potential problems caused by conflicting cache constraint definitions. If a cache table does not contain any cache key (i.e., if it is not a root table), it must be a member of at least one cache group. Otherwise the cache system cannot populate that cache table. In such a rare case, the cache table can be populated manually and still be made use of due to unique/primary key columns. In a cache database, a cache group may be completely covered by another one if its root table is a member of that cache group. Or, some cache groups may overlap by sharing one or more member tables. Note that cache groups are implicitly constructed based on cache constraints (i.e., they are not declaratively specified). In that respect, our cache group concept is very different from the one introduced in TimesTen [24].

We represent a cache group as a directed graph, called *cache group graph*, where nodes denote cache tables and edges denote RCCs. The direction of an edge for an RCC is from a cache-parent to a cache-child table. The graph may also contain bi-directional edges indicating that there are two RCCs on the same columns of cache tables in both directions. Each row in one cache table requires having corresponding rows in the other table. A (unidirectional) *path* is formed from a source table to a target table in the graph by following the directions of a set of RCCs. Each participant table must be traversed only once in a path. But a path can start and end with the same table. Such a path is called *cycle*. Note that each bi-directional edge corresponds to a cycle.

Cache keys are represented in the graph as annotations to the node representing the root cache table. Underlined cache keys denote non-unique cache keys. Notice that a cache group graph corresponds to a reachability graph of the root table node, which can reach all other nodes representing members of the cache group. When two or more cache groups are connected to each other via overlapping members, combined representation of the cache group graphs is captured in a *connectivity graph*.

Figure 7 shows a connectivity graph which includes two cache group graphs shown inside dashed lines. The content of the cache groups are identified by *custType*



**Figure 7:** Connectivity Graph Example

and *pid* cache keys of their respective root tables. For example, when a “Gold” customer is cached based on the “*custType=Gold*” predicate, the system guarantees that all gold customers, their orders and orderline rows will be cached, benefiting join queries involving these tables.

### 3.4 Issues with Cache Constraints

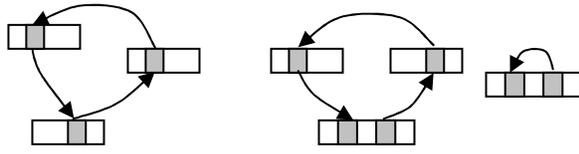
Uncontrolled use of cache constraints can cause an unexpected amount of data to be loaded into the cache database. This not only causes severe performance problems during population, but also maintenance problems later in the system (e.g., during cache invalidation). Unexpected cache load occurs when there are recurring load operations for the same cache table. We call this phenomenon the *recursive cache load* problem. In the extreme case, this problem can cause loading the entire content of the backend table. A cache group is called *safe* if the setting of cache constraints does not cause any recursive cache loads in any of its cache tables. In the next two sections, we address unsafe conditions and define a set of rules to exclude them in a cache group.

#### 3.4.1 Dangerous Paths

It is easy to see that when a cache group graph contains a cycle, then there is a danger of recursive cache load for each participant table. A safe condition that prevents the recursive cache load problem for a cycle is when a *single* column of each participant table is used during the traversal. Such a cycle is called *homogeneous cycle*. Note that each bi-directional edge creates a homogeneous cycle. On the other hand, a *heterogeneous cycle* is formed by a path if one of the participant tables contains two or more columns used in the traversal. Figure 8 shows different examples of each cycle type. Heterogeneous cycles pose a potential recursive cache load problem, hence as a precondition, they are not allowed. We set the following rule to exclude them in the cache group definitions:

**Rule-1:** A cache group graph must not include any heterogeneous cycles.

Note that our example does not have any recursive cache load problem since there isn’t any heterogeneous cycle. But, there would be one, if Product table was a cache-child of Orderline through a column other than *pid*



(a) A Homogeneous Cycle (b) Heterogeneous Cycles

**Figure 8:** Cycle Examples in a Cache Group Graph

(e.g., an RCC from order location of **Orderline** to product manufacturing location of **Product**). If this is allowed, when a new row is loaded in the **Product** table, all the corresponding **Orderline** rows having the same **pid** will be loaded. Then, for each new **Orderline** row, we may have to load a new **Product** row due to the new RCC. This may in turn force us to load more rows for the **Orderline** as each new **Product** row requires all corresponding **Orderline** rows. As a result, we may have to repeat all these operations, and in the extreme case we may load the entire **Product** and **Orderline** tables.

### 3.4.2 Implications of Domain Completeness Property

Although the domain completeness property provides key functionality for the correctness of equality predicates, complicated situations may arise because of its semantics. In particular, enforcing the domain completeness property on non-unique columns in a cache table may lead to the recursive cache load problem.

In DBCache, a cache table column is not explicitly defined as domain complete. Instead, it implicitly becomes domain complete if one of the following conditions is satisfied.

- (1) if the column is a unique (or primary key) column,
- (2) if a cache key is defined for the column,
- (3) if the column is involved in a homogenous cycle,
- (4) if the column is the only column used in the RCCs where its cache table is participating as a cache-child, and the cache table does not contain any cache key defined on another column.

(1) and (2) address domain-complete columns created as a direct consequence of the definition. The following theorem shows how domain-complete columns result from (3).

**Theorem:** All the columns involved in a homogeneous cycle are domain complete.

**Proof:** Suppose that there is a homogeneous cycle HC in a cache group graph G, including n cache tables:  $CT_1(C_1) \rightarrow CT_2(C_2) \rightarrow \dots \rightarrow CT_n(C_n) \rightarrow CT_1(C_1)$ . When a cache table  $CT_i$  is populated with a set of rows having a set of values  $v_1, \dots, v_m$  in the column  $C_i$ , then for each value  $v_j$ , the table  $CT_{i+1}$  will be populated with all the rows having  $v_j$  in the column  $C_{i+1}$ . Thus  $CT_{i+1}(C_{i+1})$  will be domain complete. Similarly  $CT_{i+2}$  will be populated with all the rows containing the values  $v_1, \dots, v_m$ , making  $CT_{i+2}(C_{i+2})$  domain complete. Eventually, the newly loaded values

$v_1, \dots, v_m$  in  $CT_{i+1}(C_{i+1})$  will cause  $CT_i$  to be populated with all the rows including those values in column  $C_i$ , making it domain complete.

In our example in Figure 7, the **cid** column of **Order** table is involved in a homogeneous cycle, and hence is domain complete according to (3).

Note that the domain completeness property created by (4) is different than others in the sense that such a column cannot coexist with any other domain-complete column. In other words, the domain completeness property is destroyed when the cache table contains a new domain-complete column. In the example, **oid** and **pid** columns of **Orderline** table are not domain complete as there are two RCCs. But with the absence of one of them, the column used in the remaining RCC will be domain complete in **Orderline**.

For a given cache table, there is no limit on the number of domain-complete columns as long as they are unique. But at most one non-unique column can be domain complete. To explain why such a restriction is needed, let's assume that two non-unique columns are domain complete in a cache table. When we insert a set of rows into the cache table for a specific value in the first cache key column, to satisfy the domain completeness property of the second column we may have to load more sets of rows from the backend for each new value in the second column. These new rows may force us to do another round of loads to satisfy the domain completeness property of the first column and so on. As a result, satisfying domain completeness for both columns can become unmanageable, and in the extreme case we may end up loading the entire backend table. As a result, we have the following rule for a cache table:

**Rule-2:** A cache table must not have more than one non-unique domain-complete column.

The domain completeness properties caused by (1) and (4) are irrelevant for this rule: For (1), the columns are unique, and for (4), two domain-complete columns cannot coexist as explained above. In order to enforce Rule-2 for domain-complete columns that may be created by (2) and (3), we do not allow the following situations in our model:

- A cache table having more than one non-unique cache key.
- A cache table having a non-unique cache key and at the same time having one or more non-unique columns which are involved in homogeneous cycles.
- A cache table having two or more non-unique columns which are involved in homogeneous cycles.

A new cache constraint is created in the system only if its addition does not violate Rule 1 and 2.

### 3.4.3 Other issues

Selectivity of non-unique cache keys is also an important factor for the usability of dynamic caching. If the cache keys are chosen from low-selectivity columns

(e.g., gender), the system will be forced to load a large amount of data dynamically. In general, it is better to use declarative cache tables for such cases.

We are planning to develop a cache advisor tool to setup cache key constraints based on a given query workload. Without having such a workload as an input, we can still automate the process by taking hints from the backend database schema. Primary keys and referential integrity (RI) constraints can give us an idea about selecting cache keys and RCCs. Especially, if there are some RI constraints replicated at the cache database, special attention must be paid to the loading order of cache tables during on-demand loading. Moreover, in general, it is reasonable to expect that there is a join operation through the RI columns in a query workload. Therefore, when there is an RI relationship between two cache tables in the cache database, we always map it to an RCC and we always keep RCCs and RI constraints consistent. In the mapping process, the cache table that contains the foreign key becomes a cache-parent and the other becomes a cache-child. We can also create an additional RCC by switching the roles to increase the likelihood of eligibility of both cache tables in a join operation. This way, we can handle joins between cache tables while satisfying RI constraints in the cache database. However, due to the recursive cache load problem, these RCCs can not always be created together. In such cases, we may have to choose either keeping RI constraints between cache tables or the capability to execute joins from cache-parent to cache-child. This decision must normally be made by the DBA as each application’s requirements might be different.

## 4. Query Compilation for Cache Tables

In this section, we present our modifications to the DB2 query compiler to generate query plans for cache tables. It is important to note that the focus of this paper is on read-only queries. Although there is some support in DBCache to handle updates at the database cache, we do not address it here due to space limitations.

### 4.1 Query Plans for Declarative Cache Tables

We exploit existing materialized view matching mechanisms for declarative cache tables. For that reason, declarative cache tables are created as materialized views (with special properties) over nicknames so that during query compilation, the view matching mechanism can route queries either to the local cache database or to the backend database. There is, however, one obstacle to activate this mechanism: In our setting, queries refer to cache tables, whereas query routing always occurs only from base tables (i.e., nicknames) to the materialized views (i.e., declarative cache tables), not vice versa. To overcome this problem, we implemented a name replacement mechanism that takes effect in the database engine after semantic processing of the queries (i.e., constraint checking, trigger processing, etc.) is done.

Basically, we replace each declarative cache table reference with its corresponding nickname in the query so that the existing view matching mechanism can be activated to route the query.

### 4.2 Generating Query Plans for Dynamic Cache Tables

The decision on choosing dynamic cache tables over nicknames to answer a query must be done at runtime since the content of dynamic cache tables may change between subsequent executions of queries. In our solution, we create two plan alternatives for each query during query compilation. The first plan considers all possible dynamic cache tables usable (it may include nicknames for other tables) in the query, hence it is called a *local plan*. The second plan is constructed only with the nicknames to enable remote query execution. Then, both plans are tied together with a conditional switch operator as shown in Figure 9. We name this new class of query plans *Janus* plans.

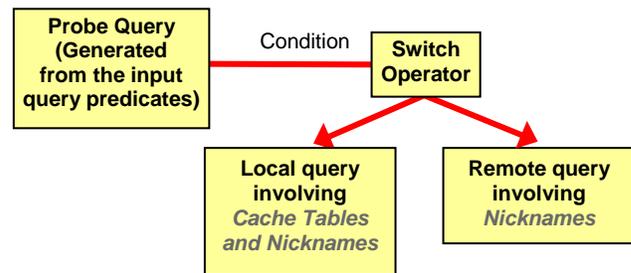


Figure 9: Janus Plan to Handle Dynamic Cache Tables

The switch condition contains a subquery, called the *probe query*, that is used to decide at runtime which leg of the Janus plan to execute. In other words, the execution of a Janus plan starts with executing the probe query followed by either the local or the remote plan depending on the probe query result. The probe query performs a data access only for each potential cache table to find out whether or not it can be used to answer the input query. Therefore, by setting up the probe query properly, Janus plans provide the needed mechanism for making a runtime decision of whether or not the cache tables can be utilized, while avoiding costly query recompilation.

A Janus plan is constructed in four steps:

- (1) We process the initial query plan to convert it into a remote plan, which contains only nicknames. As explained in step 3, during local plan generation, we switch back as many of these nickname references as possible to dynamic cache tables. This method ensures that the query is executable even if the Janus plan cannot be created. For example, when the currency setting of the cache database indicates that the applications cannot tolerate out-of-date data, no attempt is made to generate a Janus plan, resulting in retrieving the data from the backend database.

- (2) A probe query is generated by checking all the equality predicates<sup>3</sup> to determine whether they can participate in the probe query condition. If no such predicate is found, then the process is aborted at this point. A detailed description of the probe query generation is given below.
- (3) The input query graph is cloned and in the clone nicknames are replaced with the corresponding cache table names (Details are provided below). Basically, the clone becomes the “local” query plan and the unaltered plan remains as the “remote” query plan. Note that the local query may contain both cache tables and nicknames. This may result in distributed execution of the query using DB2’s federated database functionality.
- (4) A switch operator is inserted at the top of the query plan<sup>4</sup>. Local, remote and probe query plans are plugged into the switch operator.

The cost of data access for determining whether or not cache tables are usable might seem too expensive. However, as shown next, the probe query has a very simple structure and its results can be potentially reused by the local query. So, we anticipate that the extra overhead of the probe query will be acceptable considering the benefits of executing a local query as opposed to a remote one. We verified this claim with a set of performance experiments, whose results are given in Section 6.

### Creating Probe Query and Local Query Plans:

The probe query determines whether or not the input query can be executed in the local cache. It is created as a scalar subquery to keep its execution cost at a minimum and to find out its result with a simple existence check operation. As stated before, probe query predicates are constructed only from the equality predicates of the input query<sup>3</sup>. The definition of domain completeness ensures that the scalar subquery in the probe is sufficient to guarantee the correctness. Thus, if a single record of the probe query result is found in the local cache, it is guaranteed that it is safe to execute the local query.

A probe query is constructed by examining the equality predicates of the input query using the domain completeness property. During this process, we also find out initial set of cache tables and predicates usable in the local query.

Formally speaking, given a query  $Q$ , let  $T_1, T_2, \dots, T_n$  be the subset of tables used in  $Q$  that have corresponding cache tables  $CT_1, CT_2, \dots, CT_n$  and  $P_1 \wedge P_2 \wedge \dots \wedge P_m$  be all of

<sup>3</sup> That does not mean that DBCache can handle the queries that have *only* equality predicates. Our bottom line requirement here is that there must be *at least one* equality predicate including a domain-complete column in the query.

<sup>4</sup> One may ask why Janus plans are not created separately for each base table in the query graph. Although this will eliminate the need for a probe query, the dynamic nature of the switch operator provides only dynamic statistics which makes further query optimization very difficult. This is an open question and may require further research.

the equality predicates in  $Q$ . Initially, let the sets  $T_{local}$  and  $P_{local}$  be empty.  $T_{local}$  represents our eligibility set. Every  $CT_i$  and  $P_k$  can be used in the probe query iff there is a  $P_k$  of the form “ $T_i.C_x = value_i$ ”, where  $C_x$  is a domain-complete column of  $CT_i$ . In this case, the sets are updated as  $T_{local} = T_i \cup T_{local}$ , and  $P_{local} = P_k \cup P_{local}$ . After all the equality predicates in  $Q$  have been processed, the condition of the probe query becomes:

EXISTS(PQ<sub>1</sub>)  $\wedge$  ...  $\wedge$  EXISTS(PQ<sub>s</sub>), where  $s = |T_{local}|$   
And each PQ<sub>i</sub> is a subquery created as:

```
SELECT 1
FROM CTi
WHERE CTi.Cx = valuei (where “Ti.Cx = valuei”  $\in$  Plocal)
```

After the dynamic cache tables for the probe query are determined, the following checks are done for join predicates to find out whether any other potential dynamic cache table  $CT_j$  can take part in the local query:

- For every equi-join predicate  $P_k$  of the form “ $T_i.C_x = T_j.C_y$ ”, if  $T_i \in T_{local}$  and  $CT_i$  is a cache-parent of  $CT_j$  in an RCC through the columns  $CT_i.C_x$  and  $CT_j.C_y$ , then update the set  $T_{local} = T_j \cup T_{local}$
- For every equi-outer-join predicate  $P_k$  of the form “ $T_i.C_x(outer) = T_j.C_y$ ”, if  $T_i \in T_{local}$  and  $CT_i$  is a cache-parent of  $CT_j$  in an RCC through the columns  $CT_i.C_x$  and  $CT_j.C_y$ , then update the set  $T_{local} = T_j \cup T_{local}$

These steps are repeated until no more dynamic cache tables can be added to  $T_{local}$ . Finally, each table  $T_i \in T_{local}$  is replaced with its respective dynamic cache table  $CT_i$  in the local query plan.

Note that in this algorithm, the way of checking probe predicates at runtime has the consequence that if one of the cache table’s predicates fails, then none of the cache tables will be used. In the future, we will explore using the subset of eligible cache tables under such a condition.

## 5. Cache Table Population and Maintenance

In this section, we present population and maintenance mechanisms for cache tables. These mechanisms are different for declarative and dynamic cache tables.

### 5.1 Mechanisms for Declarative Cache Tables

To populate declarative cache tables initially and to keep them up-to-date later, DBCache relies on DPropR utility [16] which is IBM’s asynchronous data replication tool for relational data. DPropR consists of two independent programs, a data change capture program and an apply program. Based on subscription settings, the capture program detects changes in a source database and notifies the apply program. Using the predicates given during the creation of declarative cache tables, we automatically configure the replication subscriptions. When the capture and apply programs start running, the declarative cache tables are loaded with the data from their counterparts in

the backend database and asynchronously updated at the specified frequency.

## 5.2 Mechanisms for Dynamic Cache Tables

In this section, we present the on-demand loading feature, and accompanying cache invalidation mechanisms to keep dynamic cache tables consistent with the backend database. Figure 10 illustrates the components developed for this purpose.

### 5.2.1 On-demand Loading of Dynamic Cache Tables

Each execution of a remote query in a Janus plan corresponds to a cache miss in DBCache. The cache key values that have failed the probe query are used to perform on-demand cache loading. To extract those values, we attach a special user defined function (with no side effects) to the remote query. However, we don't populate the cache tables immediately, because cache constraints may require loading an unknown amount of additional data into an unknown number of cache tables. As this operation may cause severe performance problems, we pass on the cache key values along with associated cache table information to a cache daemon by creating a non-persistent MQ message in the user defined function. The daemon runs as a lower priority background process and checks the cache constraints and issues the required insert statements asynchronously against relevant cache tables.

The basic idea behind our cache population algorithm is to prepare (at most) one insert statement per cache table in a cache group, and then to execute these statements in a *single* transaction in cache-parent-to-child order of the affected tables. Statement preparation is done using the following procedure.

- (1) For each received cache key value, we determine the set of rows that need to be inserted into the corresponding table  $CT_0$  (a.k.a. *qualifying rows* of  $CT_0$ ) by considering all cache keys of  $CT_0$ . For each defined cache key we need to guarantee domain completeness. Note that non-unique cache keys result in loading multiple rows for  $CT_0$ .
- (2) Starting from cache table  $CT_0$ , for every RCC constraint  $CT_i \rightarrow CT_{i+1}$ , we determine the qualifying rows for  $CT_{i+1}$  based on the qualifying rows for  $CT_i$  and the cache keys defined on  $CT_{i+1}$ . The set of qualifying rows for  $CT_{i+1}$  is the set of cache-child rows corresponding to the qualifying rows set of  $CT_i$  plus all the rows necessary to satisfy domain completeness properties due to the cache keys of  $CT_{i+1}$ . Recursively, for all outgoing edges from  $CT_{i+1}$ , we repeat step (2). Note that if we encounter a non-unique cache key or if we have multiple incoming edges for a cache table, we might have to revisit cache tables and expand their set of qualifying rows.

Qualifying rows for a cache table are represented as a (nested) subquery. Thus, the insert statement for each

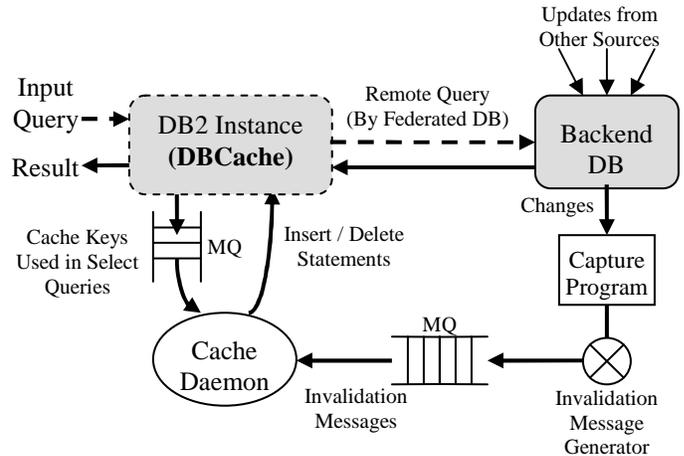


Figure 10: Components for On-demand Loading and Maintenance

visited cache table contains a select subquery on the nickname to retrieve all the qualifying rows that do not already exist in the table.

### 5.2.2 Cache Invalidation

The content of dynamic cache tables is invalidated as they get updated/deleted in the backend database. As in the setting for declarative cache tables, updates are detected by the capture program of the DPropR utility [16]. However, instead of using the apply component, we utilize the cache daemon to process updates for dynamic cache tables. The capture program provides facilities to access updated rows. As shown in Figure 10, we generate invalidation messages and send them to the cache daemon. Upon receiving such a message, the daemon creates delete statements according to the cache constraints and issues them against the cache database. Once the invalidated data is discarded from the cache, updated rows can be reloaded as new requests are processed. In the future, we plan to explore updating the cached data rather than invalidating it as well as a way of discarding unused data from the cache.

## 6. Experiments

In the experiments we did not focus on how much response time improvement we can achieve with database caching. There is a large number of related work in the literature that experimentally show benefits of caching in the Internet environment [4][8][11][20][21]. Instead, we performed a set of experiments to evaluate the overhead of Janus plans for dynamic cache tables. We do not report any experimental results for declarative cache tables here since their implementation relies on materialized view mechanisms, and performance issues for materialized view selection and matching were also well studied in the literature [13][25]. Our goal for dynamic cache table experiments was twofold: (1) to measure the additional runtime cost incurred by probe query execution and the switch operator in the Janus plans, (2) to measure the overhead of on-demand loading operation for Janus plans.

## 6.1 Experiment Settings

We picked two tables as cache tables and generated three types of queries for these tables from IBM's Trade2 J2EE Benchmark [18]. The benchmark models an online brokerage firm providing web-based services such as login, buy, sell, get quote, and more. Different types of queries helped us to understand the overhead of Janus plans in various application scenarios. The first query type is a simple select statement with unique cache key (created on the primary key column) access, the second type again is a simple select statement but with non-unique cache key access, and the third type is a join query involving two tables as shown in Table 2. We created the same primary keys and indices at the cache database. Table 1 and Figure 11 show the cache and backend database settings for the experiments.

| Table Name       | Rows    | Primary Key    | Indices |
|------------------|---------|----------------|---------|
| TradeHoldingBean | 227,117 | (Userid, Indx) | Symbol  |
| TradeQuoteBean   | 5,000   | Symbol         | -       |

**Table 1:** Cache Tables and their Backend Database Settings

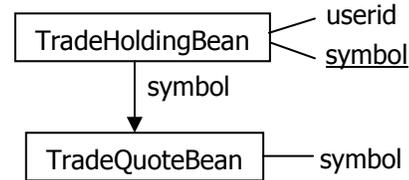
|                                    |  |
|------------------------------------|--|
| <b>Unique Cache Key Access</b>     | SELECT T1.Symbol, T1.Price, T1.Details<br>FROM TradeQuoteBean T1<br>WHERE T1.Symbol = ?  |
| <b>Non-unique Cache Key Access</b> | SELECT T1.UserID, T1.Symbol, T1.Quantity,<br>T1.Price<br>FROM TradeHoldingBean T1<br>WHERE T1.Symbol = ?   |
| <b>Join Query</b>                  | SELECT T1.UserID, T1.Symbol, T1.Price,<br>T1.Quantity, T2.Details<br>FROM TradeHoldingBean T1, TradeQuoteBean T2<br>WHERE T1.UserID = ? AND T1.Symbol =<br>T2.Symbol |

**Table 2:** Queries Used in the Experiments

All experiments were performed on two IBM IntelliStation Netfinity 3500 machines with 1GHz Intel P4 CPU, 1GBytes of memory and Windows 2000 operating system. Our cache and backend database machines were connected in a local area environment. Our DBCache research prototype [7] was implemented using the DB2 v8.1 code base.

### 6.2 Janus Plan Overhead: Cache Hit Case

In this experiment we compared performance of a Janus plan against directly querying local cache tables. Note that the latter corresponds to pure execution of the local query plan in a Janus plan. Therefore, the difference between the numbers showed us how much overhead was introduced by the probe query and switch operator when a cache hit occurs. We populated all cache tables with full backend data so that the probe query always finds the required cache key value in the cache tables, hence triggering the



**Figure 11:** Cache Constraint Definitions in the Experiments

local query plan execution. Figure 12-A shows the results of this experiment.

As seen from the graph and the table, the overhead dramatically drops as the local query plan gets relatively more costly than the probe query in the Janus plan. Even for a simple query including only a single join operation, the overhead is less than 1%. Considering that a typical TWA workload will contain many similar or more complex queries as the dominant cost factor, the overall overhead of Janus plans will be minimal when a cache hit occurs. We further reduced the overhead of Janus plans for simple queries accessing a single table by combining the probe and the local query.

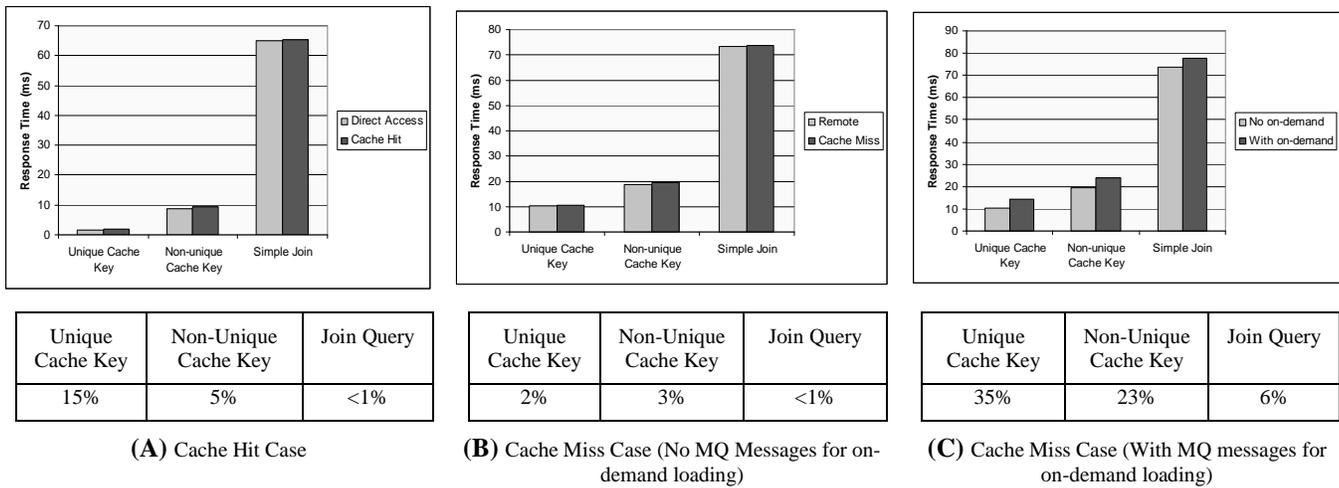
### 6.3 Janus Plan Overhead: Cache Miss Case

We expect the overhead of Janus plans to be even more negligible in a cache miss case due to high network costs for the execution of the remote query plan. This time we first measured pure remote execution of the queries and then compared these numbers with those of Janus plans. The difference tells us the overhead. To guarantee a cache miss, we issued queries against initially empty cache tables, and used different constant values in each query.

Note that when a cache miss occurs there are two cases. In the first case, if the probe query tests only domain-complete column values, then only the remote query is executed and the results are returned without populating any cache tables. On the other hand, if the test occurs for cache keys, then, in addition, we create MQ messages to send missing cache key values to the cache daemon. Therefore, we conducted two sets of experiments to evaluate the performance in each case.

Figure 12-B shows the results for the cache miss case when no population occurs. As expected, the overhead is less than in the cache hit case, since the network cost became main factor in query response times, reducing the effect of Janus plans.

In the last experiment, we enabled on-demand loading for each cache miss and generated MQ messages for the cache daemon. Note that this is actually an extreme case since under normal circumstances, the number of cache misses will decrease as the cache tables get populated. We observed a considerable overhead for simple queries caused by the MQ mechanism to pass cache key values. However, as the remote query plan gets relatively more costly, the overhead dramatically drops. The results of this experiment are shown in Figure 12-C.



**Figure 12: Experimental Results: Measuring Overhead of Janus Plans**

Overall, the experiments verified that dynamic cache tables add only minimal overhead in the cache database. Considering the benefits of on-demand database caching in terms of better response time, higher scalability and availability on the Internet, the overhead will be negligible.

## 7. Related Work

Oracle [23] and TimesTen [24] offer database cache products that are directly related to our work. Similar to our previous DBCache solution [3][19], Oracle’s approach involves full-table caching using a full-fledged database server in the middle-tier with updates fed through replication. Their solution ensures that other objects such as stored procedures and user-defined functions get deployed in the middle-tier from the backend as well. Although this approach has the advantage of considerable application transparency, it is not very adaptive and requires considerable cache management tasks on the part of administrators. TimesTen Front-Tier [24], on the other hand, allows sub-table level caching and update queries at cache databases. However, applications must be aware of the cache content and choose the target database (i.e., cache or backend) accordingly. Moreover, Front-Tier is restricted to work only with Oracle as the backend database. A cache group notion similar to ours was first introduced by TimesTen. However, their cache group definition is solely based on referential integrity constraints of the backend database and is less powerful.

One distinctive feature of DBCache from these products is that it can do distributed query execution. In DBCache, the user query can be executed at either the local database cache or the remote backend server, or more importantly, the query can be partitioned and then distributed to both databases for cost optimal execution.

Query result caching is a similar approach to database caching in the sense that the cache content is checked with backend database queries. But, unlike database caches, the

cache can store only partial backend table data, not any other relevant database objects such as triggers, constraints, stored procedures, etc. The earliest query caching work was semantic caching [9], which described ways in which a client proxy might cache the results of queries executed in a remote database. The big disadvantage of semantic caching was that it worked only for range queries and did not address joins. As we saw in the analysis of TWA queries, both these query styles are very common. Further, semantic caching did not address the impact of updates of cached data in the remote server.

The DBProxy project [4] offers an improvement on semantic caching by supporting common TWA queries and building an infrastructure to invalidate out of date data in the cache. However, DBProxy suffers from the other disadvantage of semantic caching – conventional RDBMS architectures support complex SQL queries. This means that the cache implementation plays “catch-up” always lagging behind the syntax supported in the database server. This lack of transparency makes it difficult for application developers to adopt such approaches.

Most query result caches use materialized view technology to store and match the cache content, although materialized views were first developed for improving query performance in data warehouses and OLAP applications [12][13][25]. As we stressed throughout the paper, our cache table concept goes beyond that of materialized views by providing richer semantics and supporting dynamically changing content.

There are other caching methods that take place in various forms ranging from HTML pages to business object caches at different layers of transactional web application infrastructures. Proxy caches like [5][8][20] and EJB caches in IBM’s WebSphere [17] and BEA’s WebLogic [6] application server products are a few examples. [21] and [22] provide a good survey of existing products and ongoing research efforts in this area.

Finally, there is a countless number of caching methods proposed in different database contexts.

Examples include caching in client/server databases [11], in mediator systems [1], in database middleware systems [14], and so on. However, these methods are geared towards solving specific performance problems in their application domains.

## 8. Conclusions

The focus of the DBCache project has been on developing the core functionality for cache tables. We have implemented sophisticated mechanisms inside the DB2 query engine not only to cache static subsets of backend tables but also dynamically changing, workload-driven subsets. Once the set of dynamic cache tables and cache constraints have been specified, DBCache can asynchronously populate the cache tables on demand. Caching data from backend database servers in such an adaptable fashion is a key feature needed to deploy database caches at remote data centers or at the edge of content delivery networks. As a result, businesses will be able to move processing outside of their central infrastructures, improving the response time, throughput and scalability of transactional web applications.

Our long term goal for the DBCache project is to achieve a highly efficient, scalable, zero-admin database cache. The cache table concept presented in this paper is an important step towards this goal. We plan to design new tools that will ease the deployment of DBCache as well as new techniques that will iteratively refine the cache settings at runtime by adding or dropping cache elements. For example, DBCache will actively monitor results of the workload, determine new potential cache table candidates, create cache constraints and start caching data for them. We are now laying the foundations for this truly adaptable database cache.

## Acknowledgements

We would like to thank Nesime Tatbul, Qiong Luo, Honguk Woo, Larry Brown, Bruce Lindsay, Dan Wolfson and Theo Haerder for their contributions to the DBCache project. We are also benefited from fruitful discussions with IBM SWG team including Mary Roth, Eileen Lin, and George Lapis.

## REFERENCES

- [1] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems", SIGMOD'96, Montreal, Canada, June 1996
- [2] Akami Technologies, <http://www.akamai.com/>
- [3] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Lindsay, H. Woo, L. Brown, "DBCACHE: Database Caching For Web Application Servers", (Demo Description), SIGMOD'02, Madison, WI, June 2002
- [4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "DBProxy: A Dynamic Data Cache for Web Applications", ICDE'03, Bangalore, India, March 2003
- [5] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, T. Zhong, "Web caching for database applications with Oracle Web Cache", SIGMOD'02, Madison, WI, June 2002
- [6] BEA WebLogic Application Server, <http://www.bea.com/products/weblogic/server/index.shtml>
- [7] 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, June 2003
- [8] K. S. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal, "Enabling Dynamic Content Caching for Database-Driven Web Sites", SIGMOD'01, Santa Barbara, CA, June 2001
- [9] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava and M. Tan, "Semantic Data Caching and Replacement", VLDB'96, Mumbai (Bombay), India, September 1996
- [10] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, K. Ramamritham, D. Fishman., "A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration", VLDB'01, Rome, Italy, September 2001
- [11] M. J. Franklin and M. Carey, "Client-server caching revisited", in Readings in database systems (3rd ed.), M. Stonebraker, J. M. Hellerstein (Edtrs), Morgan Kaufmann Publishers Inc., 1998
- [12] J. Goldstein and P. Larson, "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution", SIGMOD'01, Santa Barbara, CA, June 2001
- [13] A. Gupta and I. S. Mumick (Editors), "Materialized Views: Techniques, Implementations, and Applications", The MIT Press, 1999.
- [14] L. M. Haas, D. Kossmann, I. Ursu, "Loading a Cache with Query Results", VLDB'99, Edinburgh, Scotland, September, 1999
- [15] IBM DB2 DataJoiner, <http://www-4.ibm.com/software/data/datajoiner/>
- [16] IBM DB2 DataPropagator, <http://www-4.ibm.com/software/data/DPropR/>
- [17] IBM WebSphere Application Server, <http://www-3.ibm.com/software/webservers/appserv/>
- [18] IBM WebSphere Performance Benchmark Sample (Trade 2 Application), [http://www-3.ibm.com/software/webservers/appserv/wpbs\\_download.html](http://www-3.ibm.com/software/webservers/appserv/wpbs_download.html)
- [19] Q. Luo, S. Krishnamurthy, C. Mohan, H. Woo, H. Pirahesh, B. G. Lindsay, J. F. Naughton, "Middle-tier database caching for e-Business", SIGMOD'02, Madison, WI, June, 2002
- [20] Q. Luo and Jeffrey F. Naughton, "Form-Based Proxy Caching for Database-Backed Web Sites", VLDB'01, Rome, Italy, September 2001
- [21] C. Mohan, "Caching Technologies for Web Applications", VLDB'01, Rome, Italy, September 2001. [http://www.almaden.ibm.com/u/mohan/Caching\\_VLDB2001.pdf](http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf)
- [22] C. Mohan, "Application Servers and Associated Technologies", VLDB'02, Hong Kong, China, August 2002. [http://www.almaden.ibm.com/u/mohan/AppServersTutorial\\_VLDB2002\\_Slides.pdf](http://www.almaden.ibm.com/u/mohan/AppServersTutorial_VLDB2002_Slides.pdf)
- [23] Oracle Corporation, "Oracle Internet Application Server Documentation Library" [http://technet.oracle.com/docs/products/ias/doc\\_index.htm](http://technet.oracle.com/docs/products/ias/doc_index.htm)
- [24] The TimesTen Team, "Mid-tier Caching: The FrontTier Approach", SIGMOD'02, Madison, WI, June 2002
- [25] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata, "Answering Complex SQL Queries Using Automatic Summary Tables", SIGMOD'00, Philadelphia, PA, May 2000