DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures

Christof Bornhövd[1] Mehmet Altınel[1] Sailesh Krishnamurthy[2] C. Mohan[1] Hamid Pirahesh[1] Berthold Reinwald[1]

[1] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

[2] Computer Science Division, EECS Dept., UC Berkeley, Berkeley, CA 94720

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

Multi-tier infrastructures have become common practice for implementing high volume web sites. Such infrastructures typically contain TCP load balancers, HTTP servers, application servers, transaction-processing monitors, and databases. Caching has been widely used at different layers of the infrastructure stack to improve scalability and response time of e-business applications. The majority of existing caching mechanisms target only static HTML pages or page fragments. However, as web applications become more dynamic through increased personalization, these caching techniques turn out to be less useful. Consequently, as more application requests result in increased querying and updating of backend database servers, scalability limits are often reached.

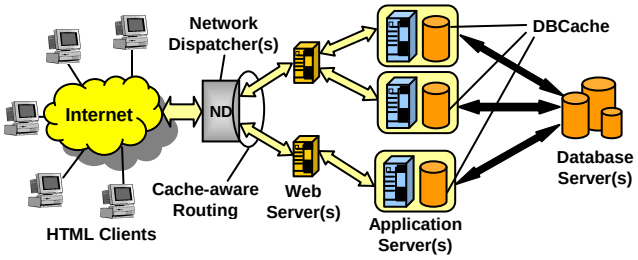

Figure 1: Database Caching with DBCache

Middle-tier database caching is a promising approach to overcome the above problem. Our research prototype is based on DB2 UDB. We introduce a new database object called *Cache Table* that enables persistent caching of entire tables, or subsets of tables. Cache tables come in two flavors. The content of a cache table is either defined declaratively and populated in advance at setup time (*declarative cache tables*), or the content is determined dynamically and populated on demand based on incoming queries (*dynamic cache tables*). In this demonstration, we focus on dynamic cache tables and present the following features. Figure 1 shows our system setup.

Query Plans for Dynamic Cache Tables. For dynamic caching, the decision of whether the corresponding cache tables can be used to answer a given query must be made at query execution time since the content of cache tables may change between subsequent query executions. In our solution, we create a special query plan that consists of three parts: a local plan, a remote plan, and a simple probe query. The *local plan* includes all cache tables that could potentially be used. The *remote plan* refers to backend tables only. At plan execution time, the *probe query* uses the current content of the table to determine whether the local or the remote plan should be executed.

Cache Constraints. We support two types of cache constraints to help us determine whether a set of cache tables can be used for a given query. A *cache key constraint* defines a column that determines which rows should be cached. It specifies that for any value in the cache key column, the cache table contains either all of the corresponding rows from the backend database, or none. The constraint ensures that a simple scalar probe query is sufficient to determine what data is in the cache and hence whether to use the local or the remote plan. *Referential cache constraints* involve two tables and require that for all rows in a parent cache table all the corresponding child rows are also cached. This constraint guarantees the correctness of equi-joins between cache tables.

Cache Population and Maintenance. Cache key values that have failed the probe query are used to perform on demand cache loading. Cache loading is performed asynchronously by a cache daemon that enforces cache constraints. The daemon runs as a lower priority background process that fetches rows from the backend database, and issues the required data modification statements. Cached records are invalidated if they are updated/deleted in the backend database. Updated/deleted records are detected by the capture program of DB2's DPropR replication utility and are propagated to the cache daemon. The daemon then issues the corresponding statements against the cache database.

Cache-aware Routing. Since different tables or subsets might be cached at different application nodes, HTTP request routing needs to take into consideration what data is cached at which node to maximize the cache hit ratio. We have extended the existing content-based routing functionality of IBM's Network Dispatcher to allow the execution of user-defined functions in the rule evaluation phase of content-based routing. For DBCache, this functionality is used to execute custom logic that routes a given user request to the appropriate application server node. This routing logic can be as simple as a hash function based on user login ID, or can be based on additional routing data cached locally at the load balancer.

We demonstrate these DBCache features using IBM's Trade2 J2EE benchmark, which models an online brokerage firm. Trade2 consists of Java classes, Java Servlets, Java Server Pages, and Enterprise Java Beans.

SIGMOD 2003, June 9-12, 2003, San Diego, CA.