Middle-tier Database Caching for e-Business [*]

Qiong Luo[1] Sailesh Krishnamurthy[2] C. Mohan[3] Hamid Pirahesh[3]
Honguk Woo[4] Bruce G. Lindsay[3] Jeffrey F. Naughton[1]

Contact email:{mohan@almaden.ibm.com, qiongluo@cs.wisc.edu}

Abstract

Scaling up to the enormous and growing Internet population with unpredictable usage patterns, E-commerce applications face severe challenges in cost and manageability, especially for database servers that are deployed as those applications' back-ends in a multi-tier configuration. Middle-tier database caching is one solution to this problem. In this paper, we present a simple extension to the existing federated features in DB2 UDB, which enables a "regular" DB2 instance to become a ***DBCache*** without any application modification. On deployment of a ***DBCache*** at an application server, arbitrary SQL statements generated from the unchanged application hitherto intended for a back-end database server, can be answered: at the cache, at the back-end database server, or at both locations in a distributed manner. The factors that determine the distribution of workload include the SQL statement type, the cache content, the application requirement on data freshness, and cost-based optimization at the cache. We have developed a research prototype of DBCache, and conducted an extensive set of experiments with an E-Commerce benchmark to show the benefits of this approach and illustrate tradeoffs in caching considerations.

1 Introduction

Various caching techniques have been deployed to increase the *performance* of multi-tier web-based applications in response to the ever-increasing scale of the Internet. Such applications typically achieve a measure of *scalability* with application servers running on multiple (relatively cheaper) systems connecting to a single database system. This, however, does not solve the scalability problem for back-end database servers. Middle-tier database caching (shown as the gray box in Figure 1) is one of these caching techniques, which is deployed in the middle, usually at the application server, of a multi-tier web site infrastructure when the back-end database is a bottleneck. Example commercial products include the Database Cache of Oracle 9i Internet Application Server [19] and TimesTen's Front-Tier [22].

In a multi-tier e-Business infrastructure, middle-tier database caching is attractive because of improvements to the following attributes:

[*] Work done at IBM Almaden Research Center
[1] Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706
[2] Computer Science Division, Department of EECS, University of California Berkeley, Berkeley, CA 94720
[3] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
[4] Department of Computer Sciences, University of Texas-Austin, Austin, TX 78712

(1) *Scalability:* by distributing query workload from back-end to multiple cheap front-end systems.

(2) *Flexibility:* with QoS (Quality Of Service) control where each cache hosts different parts of the backend data, e.g., the data of *Platinum* customers is cached while that of ordinary customers is not.

(3) *Availability:* by continued service for applications that depend only on cached tables even if the back-end server is unavailable.

(4) *Performance:* by potentially responding to locality patterns in the workload and smoothing out load peaks.

Using a general-purpose industrial-strength DBMS for middle-tier database caching is especially attractive to e-Businesses, even though there have been special-purpose solutions (for example, e-Bay uses its own front-end data cache [18]). This is mainly due to crucial business requirements such as reliability, scalability, and manageability. For instance, an industrial-strength DBMS closely tracks SQL enhancements, and provides a variety of tools for application development. More importantly, it provides transactional support, multiple consistency levels, and efficient recovery services. Finally, an ideal cache should be transparent to the application that uses it, and this is difficult to achieve with a special-purpose solution.

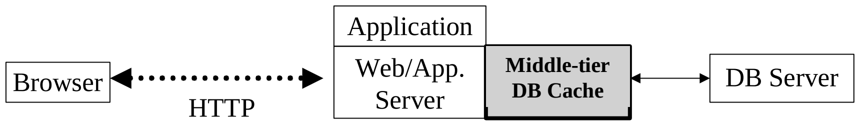

Figure 1: *Single-connection* Approach to Database Caching

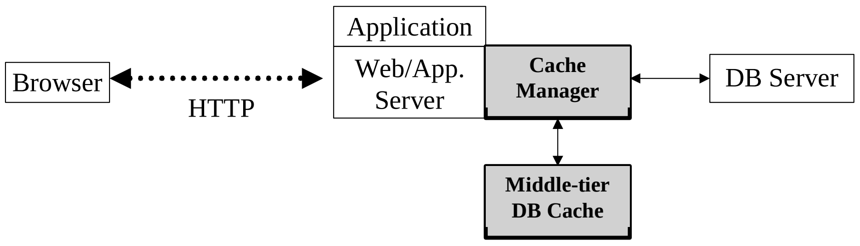

Figure 2: *Double-connection* Approach to Database Caching

The existing commercial approaches ([19], [22]) of using a general-purpose industrial strength DBMS involve an architecture such as in Figure 2. In this approach, there is a small layer (Cache Manager) that directs application database queries to different databases, and the DBCache instance has no knowledge that it is being used as a DBCache. In practice, the Cache Manager could be part of a call-level interface library [19] or special purpose code embedded in the application itself [22]. Since the application has two separate connections to different databases, we will call this the *double-connection* approach to distinguish it from the configuration in Figure 1, which we will call the *single-connection* approach. In the latter case, the DBCache is fully aware that it serves as a proxy for the backend database server. We will further distinguish the former case into *application-aware double-connection* [22] and *application-transparent double connection* [19]. As we will see repeatedly in this paper, this design choice affects many aspects of the database.

Many research questions arise in using a full-fledged database engine for middle-tier database caching, and the answers to some of them affect the relevance of others. In decreasing order of importance, these are:

(1) What are the performance bottlenecks in e-Business applications, or in other words, are we addressing the right problems by focusing on database caching?

(2) Will performance be acceptable using a commercial DBMS as a middle-tier data cache? Features such as transactional semantics, consistency, and recovery come with some overhead. What features can be dispensed with in such an environment?

(3) What database caching schemes are suitable for e-Business applications?

(4) How can a database caching scheme be implemented in a commercial database engine and how does it perform under realistic e-Commerce workloads?

(5) What is the impact of running a database server in the same computer as an application server?

(6) How can we generalize these results to other kinds of web applications?

Most of these questions remain open, partly due to the diversity of e-Commerce applications and the complexity of these systems.

In this paper, we attempt to answer some of these questions. We start with examining the opportunities in e-Commerce applications for middle-tier database caching by running an e-Commerce benchmark on typical web site architectures. We observe that this benchmark generated a large number of simple OLTP-style queries, their table accesses were highly skewed on a few read-dominant tables, and there was a clear separation between write-dominant tables and read-dominant tables. We find that web application clones could scale up to heavy loads and this leaves the backend database server to eventually become the performance bottleneck in the system.

We then explore how to extend DB2 so that it can be used as a middle-tier database cache. We take the single-connection approach, because we think it is important to leverage existing features of the DBMS engine and not produce special purpose code outside of the engine. By extending DB2's federated features we turned a DB2 instance into a table level database cache without changing user applications. The novelty of this extension is that query plans at the cache may involve both the cache and the remote server based on cost estimation. Through experiments, we showed that the overhead of adding a full-strength DBMS as a middle-tier database cache was insignificant for e-Commerce workloads. Consequently, middle-tier database caching improved users response time significantly when the backend database server was heavily loaded.

The remainder of the paper is organized as follows. In Section 2, we present our prototype middle-tier database cache (called *DBCache*) that leverages existing features of federated technology in a commercial DBMS engine (DB2). In Section 3, we describe our overall evaluation methodology with an e-Commerce benchmark. We then present our experimental results to show the performance impact of middle-tier database caching (Section 4). We discuss related work in Section 5 and conclude in Section 6 with our agenda for future research.

2 Turning DB2 into a DBCache

In this section, we will first discuss our design considerations for the DBCache. Then, we present our cache initialization tool and our modification to the DB2 engine.

2.1 Design Considerations

The goal of our DBCache is to improve the performance and scalability of web-based applications by distributing query processing to the clones of the applications and the underlying application servers (as Figure 3). When the DBCache is collocated with the application server, it serves as a database server for the node it resides in. Each DBCache can cache part of the data in the backend database server and share the server workload. With server workload distributed to multiple DBCache nodes, the overall user response time is improved when the backend server is under heavy load.

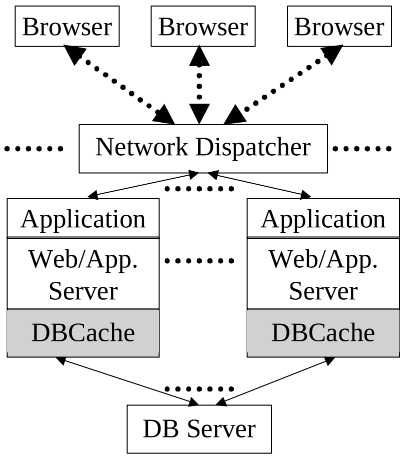

Figure 3: Deploying DBCache

With this goal in mind, we examine the design requirements, and our choice of caching schemes and existing mechanisms.

2.1.1 Requirements

The first requirement in our design of DBCache is that neither the application, nor the underlying database schema should have to change. Firstly, it is desired that the decision to deploy a DBCache could be made for an arbitrary shrink-wrapped application by local administrators who do not have access to the application source code. Secondly, requiring the application to be cognizant of the DBCache would result in increased complexity, which is undesirable especially given that cost and maintainability are already major problems in such environments. We aim to make it easy for database administrators to set up the cache database schema, and make the DBCache to be transparent to the applications at run time. Notice that in [22] any part of the application that uses the DBCache for performance needs to retarget SQL queries to the specific dialect supported by the DBCache. In general, it is desirable that the DBCache instance is capable of understanding any SQL statement that the back-end can handle. With *application-transparent double connection* [19] this is not much of an issue as the Cache Manager can easily redirect special case queries to the back-end server.

The second requirement is to support *reasonable* update semantics. Update transactions increase resource contention at the backend database server as well as the cost for cache consistency maintenance. Fortunately, E-commerce applications have "high browse-to-buy" ratios (read-dominant) and high tolerance for slightly out-of-date data. This allows us to defer update propagation to the cache so that it affects on-line transactions as little as possible. Nevertheless, time limits on the deferral of cache synchronization are necessary to ensure "reasonable" freshness of cached data.

Other requirements include support for failover of incoming requests to a failed cache node to another cache-node, and dynamic cache node addition and removal. These requirements are aimed at increasing system availability, manageability, and incremental changes to capacity.

2.1.2 Choice of Caching Schemes

Given the requirements, we have the task of choosing a caching scheme for DBCache. We categorize caching schemes by the unit of logical data (base or derived) that is cached as follows: full table, a subset of a table, an intermediate query result, or a final query result. Although (full) table level caching can be viewed as a full table scan query, it is the only scheme among the four that needs only schema information of the cached table. In contrast, other schemes need to know extra information, such as the query definitions that correspond to the current cache content. So, we regard the latter three as query result caching and consider caching a subset of a table as a special case of an intermediate query result.

Table level caching has several advantages, with the most definitive ones being easy deployment and the ability to answer arbitrary queries on cached tables. However, updates on a table must reach all nodes that cache this table within a *reasonable* amount of time. If the queries are expensive, table level caching does not save any computation even on a cache hit unless the access paths are different and/or the cache node is less loaded. In comparison, query result caching schemes may save expensive computation on a cache hit. But the downside is that they require complex schema definitions for deployment, complex query rewriting at runtime, and complex update logic to minimize the number of cache nodes to synchronize.

Note: We do not consider the *single-connection* and *double-connection* approaches to be different schemes. They are merely different designs for the same scheme – i.e., table level caching.

The relative performance of these caching schemes is determined by the characteristics of the data and workload. From our observations on an E-Commerce benchmark and anecdotal knowledge of real world e-Businesses, table level caching seems to be sufficient for these applications. The simple OLTP-type queries do not need complex intermediate result caching, the small number of frequently queried tables serve as easy candidates for table level caching; and the clean separation of read-dominant tables and write-dominant tables enables selectively caching tables to reduce update propagation costs.

Accordingly, as the first step of turning DB2 into a middle-tier data cache, we explore table level caching. Other techniques such as subset caching and intermediate result caching in the form of materialized views and final query result caching at a call-level interface library (such as a JDBC driver) are also on-going work at IBM but are out of the scope of this paper.

2.1.3 Leveraging Existing Mechanisms

Having decided on table level caching, our problem is the following: *given a web application that cannot be changed, its backend database schema, and its database workload, generate the middle-tier cache database schema, and process the SQL statements utilizing both the cache and the backend database*. As discussed earlier, we have two alternatives: one is to build a special purpose cache query processor from scratch, and the other is to leverage existing technology in DB2. In our research prototype, we chose the latter route for the following reasons. Firstly, there is an interesting match between existing federated features and the query routing function needed in our cache. Secondly, we prefer to reuse existing features and not build yet another query processor. Finally, our approach also allows us to handle distribute queries effectively, where it is possible for the optimizer to decide what portion of the query should be processed in the front end and what portion is in the backend. In comparison, in the *application-transparent double-connection approach*, new functionality is limited to the Cache Manager and outside the database engine proper.

The federated features in DB2 V7 first appeared in IBM's DataJoiner [10] product. Users can access IBM and some non-IBM databases, relational data and non-relational data, as well as local data or remote data through a single federated DB2 database. The local database, called a ***federator***, translates a user query over a local "alias" for remote data into a distributed query to remote data sources. When setting up a federated database, users need to create references in the local database to remote data sources, for example, a ***node*** to identify a remote host, a ***server*** for a remote database on the host, and a ***nickname*** for a table or view in the remote database.

If we use a federator to model a cache, and a remote data source to be the back-end database, we instantly have almost all the desired query routing capability that we need. We can therefore, design the cache schema to be such that all cached tables are local tables, and all un-cached tables to be nicknames of the back-end tables. SQL statements submitted to the cache are compiled as usual; if a statement involves nicknames, the federated features of DB2 will estimate the cost of query execution at the remote server, decide on predicate pushdown based on the cost estimation, and generate a distributed query plan.

Our prototype currently does not support updates in the front-end. Instead, users need to issue the "*set passthru <remote-server>*" command and then issue the updates on the referenced remote tables. In *passthru* mode, the federator blindly forwards the statements to the specified remote server. This allows users to update remote data sources directly and frees the federator from maintaining multi-site updates. As we will see, this restriction necessitates some modification to the DB2 engine in order to use it as a middle-tier DBCache without changing user applications. Nevertheless, even without this restriction, we would still need something special to route all updates to the backend even when the affected tables exist in the cache. It should be noted that the passthru functionality is present in DataJoiner to handle those cases where users may choose to entirely bypass the frond end and send the requests directly to the backend. DBMSs often have unique extensions to the standard SQL. Using passthru, one can exploit these features.

Like the approaches in [19] and [22], we aim to keep the data in our DBCache consistent using standard replication techniques. In our approach, we use DataPropagator/Relational (DPropR) [11]. DPropR is IBM's tool for asynchronous data replication for relational databases. In conjunction with DataJoiner, DPropR can also support non-relational data replication and replication between non-IBM and IBM databases. DPropR consists of three independent programs: an administration program, a data change ***capture*** program, and an update ***apply*** program. The three programs communicate with one another through a set of tables called ***control tables***. Users can "subscribe" replication requests using GUI tools and the subscription information is stored in the control tables. Subscriptions can be on a set of tables, possibly with some selection predicates on tables. Users can also specify the frequency of update propagation, the minimum size of each data transfer, among other options. For IBM data sources, DPropR uses a log reader to capture changes; for non-IBM data sources, DPropR uses triggers on the source database to capture changes. The apply program is a "humble" database application operating on the control tables and the target tables.

In our approach, as in [19], all updates *must* happen in the back-end, and so we chose a replication scheme (based on DPropR) for update propagation between the back-end database and the DBCache. This ensures that all write operations take place at the same location with no possibilities for asynchronously-performed conflicting updates. In future work, we will address other options for handling updates to improve availability and performance. In addition, even if an active transaction holds a write lock against a cached data item in the back-end, that does not preclude another transaction from reading a *transaction-consistent* older version of the same data item in another front-end. This way, we exploit relaxed consistency requirements of e-Business applications and avoid the overhead of two-phase commit transactional consistency maintenance.

2.2 Cache Initialization

Implementing table level caching using DB2 consists of two pieces of work. The first is a tool for initializing the cache, including choosing tables to cache, and setting up update propagation tools for them. The second is to modify the DB2 engine so that it can consider user specified data freshness requirements and route requests to the cache or to the backend database. In addition, for reasons explained below, DPropR also has to be modified slightly. One of our design requirements for DBCache is that the existing web applications and their database schema should not change when deploying a DBCache. As this is a basic requirement for usability, we implemented a tool called ***DBCacheInit*** to automatically create the database schema for a cache database and initialize it.

First, the tool collects necessary access authorization information about the backend database, such as the server name, the backend database name, and user/password information. Then, it uses that information to examine the catalog of the backend database, and further collects information about existing tables, views, indexes, referential integrity constraints, and so on. Information about triggers is not collected, as triggers are only involved with updates, and in this version, we want all updates to happen only at the back-end. Stored Procedures may contain update statements and so we don’t deploy them at the DBCache either. In DB2, user-defined functions may not update database content. However, in general, the back-end and front-end nodes may be on different platforms, and so for simplicity the tool does not attempt to re-deploy user-defined functions in the front-end.

Ideally, after collecting information about the backend database, the tool should examine a snapshot of a typical workload consisting of SQL queries, and decide which tables to cache. This is a similar problem to that solved by DB2's Index Advisor, which recommends indexes based on query workload and available disk space. So, in our tool, we presume that selection of the cached tables versus uncached tables is provided a-priori.

Once the tables to cache are determined, the tool then creates a cache database with the same name as that of the backend database, unless specified otherwise, and replicates the to-be-cached tables at the cache database. For each table that is not to be cached, the tool creates a nickname for it at the cache database, with the same name as the corresponding table at the back-end. In addition, all views in the back-end are recreated in the front-end. By setting up names of cached tables and nicknames identical to their counterparts at the backend database, the user application does not need to change or even be aware of the existence of the cache database. The DB2 federated query processor will decide how to process queries.

In addition, indexes on cached tables can be replicated to the cache database. Indexes on non-cached tables, are replicated as "index specification only", where actual indexes are not created, but the cache database is informed about the existence of a real index at the back-end. This lets the cache database choose a potentially better plan.

Finally, for the cached tables, we need to load their initial data. We also need to set up replication subscriptions for them so that when the tables in the backend database change, the cached tables will be brought up to date asynchronously. We use DPropR for this purpose. When the cached tables are subscribed for update propagation, and the capture and apply programs start running, the cached tables are loaded with the data from their counterparts in the backend database and are updated asynchronously at the specified frequency.

2.3 Inside DBCache

2.3.1 DBCache Mode

Because we are modifying a regular DB2 into a DBCache, we need a way to indicate to the DBMS instance that it will be used as a cache. We have two choices over where to set this cache mode: either at the DBMS instance level, or on a per-database basis. A cache mode at the instance level is simpler to implement while a cache mode at the database level is more flexible. If the cache mode is on a per database basis, the DBMS will make query routing decisions according to which database the queries are on. This allows a DBCache to manages both “real” local databases and local databases that serve as a cache of a remote database. The downside is that it is less efficient than a DBMS level setting if the instance is deployed purely as a cache. In our current prototype, we use a DBMS level cache mode setting, and may change to a database level setting in the future.

Another issue related to the cache mode setting is whether we allow a cache DBMS instance to be served by multiple remote DBMS instances. Ideally, this should be the case since federated DB2 can serve queries that access multiple remote data sources. Unfortunately, federated DB2 does not support multiple site updates yet while our DBCache will face this problem if it allows for multiple remote servers. Therefore, we support only one remote server per DBCache instance in the current stage. The cache database is aware of the identity of the remote server and uses it in its query processing decisions.

Finally, even in the cache mode, applications sometimes may need to access the most-up-to-date data (for instance, shopping cart), and sometimes need to issue updates that are targeted to the cache database itself (for example, the apply program of DPropR). For the case of accessing the most-up-to-date data, we allow users to change the value of an existing special register called "*REFRESH AGE*" to specify their data currency requirement.[*] For the operations targeted at the local database, we provide a command "*set passthru local*" so that any operations after that and before a "set passthru reset" are executed locally even in a cache mode. We will present more details for passthru in the following section.

2.3.2 Auto-Passthru

When the DBCache instance detects the cache mode setting, it will route requests to the cache database, to the backend database, or to both places for distributed query processing. We achieve this by introducing an "automatic passthru", or *"auto-passthru"* mechanism based on DB2's existing passthru mechanism.

The existing mechanism relies on the commands *"set passthru <remote-server-name>"* and "*set passthru reset".* All statements submitted after a passthru session has been turned on and before it has been reset, are sent to the specified remote server directly. The exception to this is *another* "set passthru" command. If a user sets passthru to Server A and then sets passthru to Server B before resetting passthru, the statements before "set passthru B" are sent to Server A directly, and after "set passthru B" to Server B directly, implicitly ending the passthru to Server A. When a "set passthru reset" is issued later, the passthru mode to B is then ended. Note that this model is different from a truly nested or a stack model.

Unlike the existing passthru mechanism, we do not depend on explicit passthru set and reset commands, as that will require applic\ation modification. Instead, *auto-passthru* takes place in the cache mode. Three factors affect where a statement is executed: (1) statement type, whether it is a UDI (Update/Delete/Insert) or a query (Select), (2) the current value of the "REFRESH AGE" register, which indicates the user's tolerance for out-of-date data, and (3) any nicknames in the query.

More specifically, if a statement is a UDI, *auto-passthru* sends it through to the remote server. This is to avoid conflicts due to multi-site updates as well as to bypass the update restrictions on nicknames. If a statement is a read-only query, *auto-passthru* examines the current value of "REFRESH AGE" to decide further: if the value is 0, it means that the user has requested the most-up-to-date data. In this case, auto-passthru will send the statement

[*] This value can be set or changed without necessarily modifying the application program.

through to the backend database server to ensure the freshness of the data. Otherwise, the auto-passthru mechanism will allow the query to be executed locally at the cache. Interestingly, if a query is routed to the local database but involves nicknames, then the existing federated query processing takes over: if the query involves any cached tables, then a distributed query plan is generated; otherwise, a remote-only plan is chosen. Finally, statements other than a UDI or a query, such as Data Definition Language (DDL) statements, are directly passed to the backend database on the assumption that it is what the user desired. The philosophy here is that the user is in general unaware of the existence of the cache, and so any schema change should be effected at the back-end database.

For situations where a user is aware of the cache's existence (such as creation of local indexes), we need a way to capture the user's intent. An example of a situation where operations are explicitly targeted at the cache database is the DPropR apply program. This application propagates data updated on the back-end database to the cache database. Since DpropR reuses the SQL API, the cache DBMS engine has no way of knowing that these updates are targeted at the cache database, and so we have to ensure that *auto-passthru* does not send them to the back-end database again. Other administration activities over the cache database face the same problem. We solve this problem by providing an SQL statement *"set passthru local"*. Applications use this command to indicate that the following statements should be executed locally even in the cache mode. Like a normal passthru, this can be turned off with the *"set passthru reset"* command. In our case we modified the DPropR apply program to let it issue "set passthru local" command right after it sets up a connection to the cache database for applying changes.

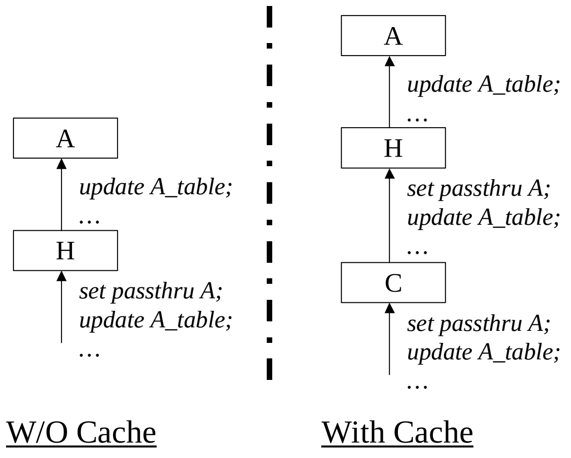

Figure 4: Handling User-Specified Passthru

Finally, we also need to solve the problem of an explicit passthru that is specified by the user application. In other words, the use of passthru for a DBCache should not be intrusive. The following scenario (in Figure 4) illustrates the problem. Suppose a user application uses its backend database H as a federator for some other data source A and has a set passthru statement "set passthru A". Now the administrator decides to deploy a database cache C in front of its backend database H. When the "set passthru A" command comes to C, auto-passthru needs to record this information, and pass this statement itself and the following statements to the backend database H. This is because the cache database C should not bypass the backend database H and forward subsequent SQL requests directly to A. In fact, the cache database C might have no knowledge of the data source A. In a real-world environment, web application servers often live in a DMZ (De-Militarized Zone) and may not be able to access the same servers that the original database H can.

To put it together, we show the state transition graph comparing the original passthru and our auto-passthru in Figure 5. Note: (1) The italics (e.g. *read-only*) above each box denotes which SQL requests (excluding passthru set/reset and commit/rollback, which are processed specially) are handled at the cache node. The remaining commands are handled at the backend node. (2) The “PT DS(i*)” state means a mode of passing thru to the most recently specified passthru data source.

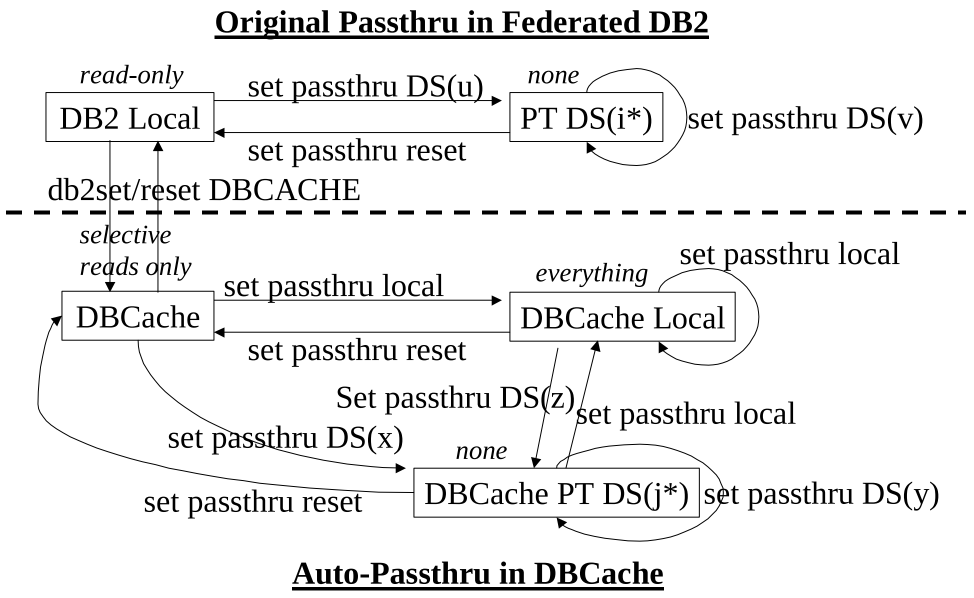

Figure 5: Comparison of Passthru State Transition

The two states above the dotted line are for DB2 in its non-DBCache mode (regular federated feature). In the “DB2 Local” state, the local DB2 processes almost all kinds of SQL statements, except giving error messages for UDI operations on nicknames. On a “set passthru DS(u)” request, the local DB2 transits into a “PT DS(i*)” state where i*=u. In this state, the local DB2 passes every following request to the backend DS(i*) until a “set passthru reset” request comes. If another “set passthru DS(v)” request comes in this state, the local DB2 stays in the “PT DS(i*) state where i*=v. In the federated environment, the local DB2 talks to one data source at any point of time (no real nesting) and remembers only the most recently specified data source for passthru. On a “set passthru reset”, the local DB2 will go back to the “DB2 Local” state.

The three states below the dotted line are for DB2 in its DBCache mode. After the DBA issues the dbcache-mode-setting command via “db2set”, the local DB2 enters the “DBCache” state, in which the local DB2 will handle reads on local tables selectively (only when refresh-age = any). Upon a “set passthru local” command, the local DB2 enters the “DBCache Local” state, in which all later requests including UDIs are handled locally. From this “DBCache Local” state, upon a “set passthru DS(z)”, the local DB2 enters the “DBCache PT DS(j*)” state, in which the local DB2 passes commands to the backend database and the backend database in turn passes the commands to the specified data source. The “DBCache PT DS(j*)” state can also be reached from the “DBCache” state upon a “Set Pasthru DS(x)” command. Proper handling of application issued “set passthru” is important since, as we explained before, even in DataJoiner passthru was supported to allow exploitation of non-standard features of remote data sources.

3 Evaluation Methodology

There are at least two alternative ways to examine the performance impact of middle-tier database caching in e-Business applications. One alternative is to apply our prototype in the field and perform case studies. Unfortunately, this is seldom viable for various business reasons. Moreover, with the diversity of these applications and workloads, it may be difficult to gain insights from case studies. The other alternative is to pursue simulation studies. The problem there is that it may be extremely difficult to model the complex running environments of e-Commerce applications.

Therefore, we chose to pursue a middle-of-the-road approach to test out our ideas. We used hardware and software components that are popular in real e-Commerce applications to build our testing environment. We chose an e-Commerce benchmark called ECDW (Electronic Commerce Division Workload) to be the test target application. This benchmark was developed and is used internally by the WebSphere Commerce Server Performance group at IBM Toronto Lab. It is similar to the TPC-W benchmark [23], but has more features that are typical in e-Commerce applications.

We tested the ECDW workload in several typical server-side configurations. We chose to use production DB2 in all configurations, instead of using production DB2 for some configurations and using our DBCache prototype for some other configurations with a middle-tier database cache. The main reason was that we wanted to compare the caching scheme results with the non-caching results without worrying about the effects of implementation differences. Therefore, for configurations with middle-tier database caching, we created special database schema at the middle-tier to simulate the effects of caching. By "simulating" table level caching using the existing DB2 federated features, it is sufficient to show how this scheme performs.

Disclaimer: *The usage of the ECDW benchmark throughout this paper is for us to gain insights in an e-Commerce application and test our ideas. It was neither intended for nor should be in any way viewed as the best possible performance results for any specific IBM or non-IBM products.*

3.1 Benchmark Description

The ECDW benchmark was designed through close interactions with a wide range of customers in order to reflect the key characteristics of real world e-Commerce applications. It simulates web users accessing an on-line shopping mall. The application used is IBM's WebSphere Commerce Suite (WCS) [13], and Segue Software's SilkPerformer [21] tool to simulate the user workload and test performance. We describe WCS, SilkPerformer, and ECDW itself in order.

WCS is an integrated solution used by e-commerce sites in various industries. A sample set of customers are: BuyUSA, InfinityQS, Mazda's Competition Parts Program, Milwaukee Electronic Corporation, and IBM's own shopIBM site (www.ibm.com). It provides services for creating, customizing, running, and maintaining on-line stores throughout their entire lifespan of operations. On the database side, it has more than 500 tables, many indexes, constraints, and triggers. There are tools to create the database schema and load in data. On the application side, it uses an Enterprise Java Bean (EJB) framework so that developers can program database accesses without being directly bound to the underlying database schema. Furthermore, customers can and do extend the database schema as well as the application, by creating new columns and tables and new EJBs.

SilkPerformer is a load and performance testing tool for e-business web applications. It emulates workloads that testers specify, such as number of concurrent users, testing period, testing scenarios, and other options. The tool warms up the testing environment, does the measurement, and finishes with a proper shutdown process. All the measurements are performed on the client side; in the normal configuration, no instrumentation is done at the web server, application server, or backend database server. The measurement output includes throughput, response time, user-defined counters, user-defined timers, and other numbers, both on a running basis and on an aggregation basis.

The ECDW benchmark uses around 300 tables in the WCS schema. The database size can be small (10,000 items, 650MB), medium (30,000 items, 2GB), or large (50,000 items, 3.5GB). The "*regular shopping scenario*" is defined in a SilkPerformer script, which depends on two variables – user type and shop flow. The user types are: new registering user (5%), existing registered user (10%), and guest user (85%). The shop flows are: browsing (88%), browsing and adding to a shopping cart (5%), browsing and preparing an order (2%), and browsing and buying (5%). These ratios were obtained through customer interactions, and thus attempt to mimic common "browse to buy" ratios at real shopping sites.

The benchmark measures web interactions and web transactions. A ***web interaction*** corresponds to a user conducting a specific operation at a browser that may involve a few mouse clicks and possibly some user input. For instance, a *LogOn* web interaction is one in which a registered user clicks on the "Log on" link, fills out her information, and clicks on the submit button. Or a guest user clicks on a link to a product item to view the details of that item. A ***web transaction*** corresponds to an HTTP *session* – a series of user operations, which includes multiple web interactions.

Figure 6 shows the regular shopping scenario of the benchmark. Each box represents a web interaction. A web transaction in the regular shopping scenario consists of the following sequence of web interactions: 1) Go to the front page of the store. 2) If the user is a new registered shopper, register with the site; if the user is an existing registered shopper, log on to the site; otherwise (a guest shopper), go to the next step directly. 3) Browse a few items. 4) If the current shop flow is not just browsing, but also preparing an order or buying, loop a few times adding some products browsed into

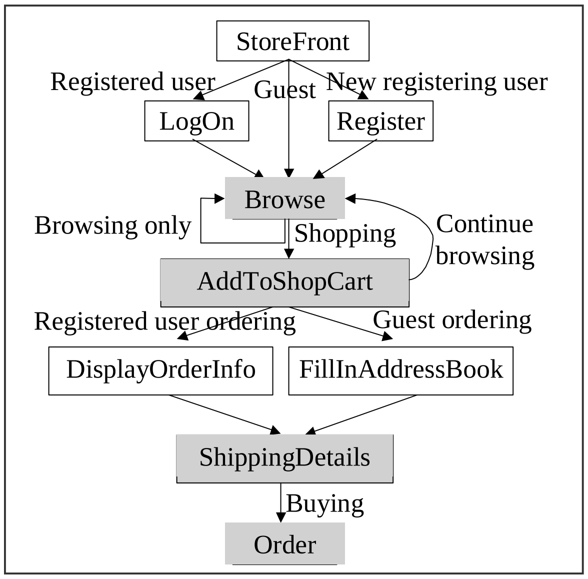

Figure 6: Regular Shopping Scenario

the shopping cart, and browsing a few items again. 5) If the current shop flow needs to prepare an order, open and fill out the address book for a guest shopper or directly display order information for registered users, and generate the detailed shipping information for all users. 6) If the current shop flow is to buy, finish the order.

Since there are four different shop flows in the regular shopping scenario, a web transaction may end after one of the following four web interactions (grayed boxes as in Figure 6): Browse, AddingToShopCart, ShippingDetails, and Order correspondingly.

3.2 Server-side Topologies

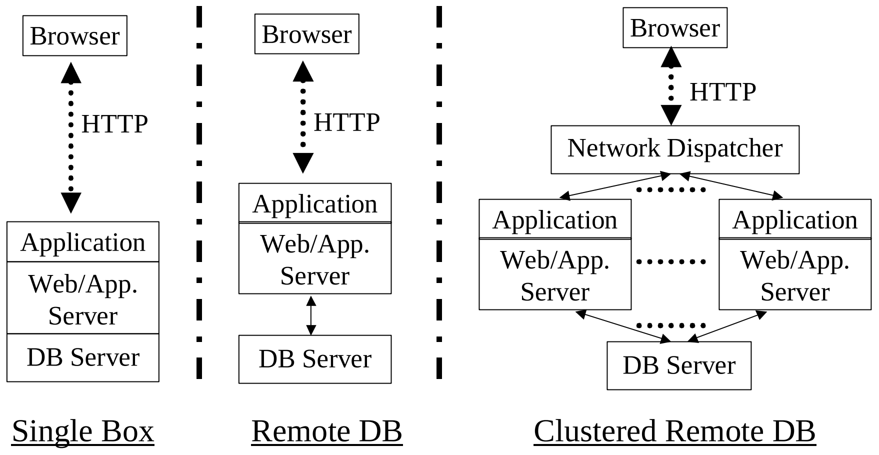

Figure 7: Three Non-Caching Server-side Topologies

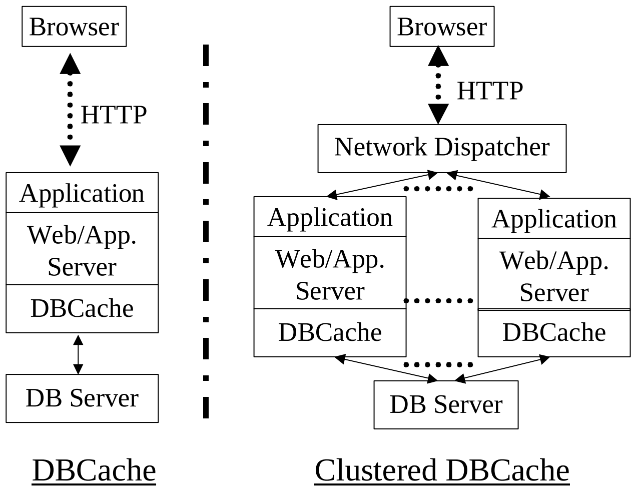

Figure 8: Two Caching Server-side Topologies

We tested on five server-side topologies, two of them with a middle-tier database cache, and three of them without. The three topologies that do not have a middle-tier cache (shown in Figure 7) are the following: (1) single box, in which the web/application server and the backend database server are on the same machine; (2) remote DB, in which these two components are on two machines; (3) clustered remote DB, in which there are multiple web/application server machines to communicate with the same backend database server. The HTTP requests are distributed to the web application server machines in round-robin fashion.through a network dispatcher or some mechanisms like that.

The two topologies that do have middle-tier caching are shown in Figure 8. DBCache topology adds a middle-tier database cache to the Remote DB topology, and Clustered DBCache adds a middle-tier database cache to each web application server in the Clustered Remote DB topology. Note that the single box topology in Figure 7 is essentially a DBCache topology with a 100% cache hit ratio.

3.3 Test Environment Details

We used six computers in the tests. Four of them were IBM Netfinity 3500 server machines with an 800MHz Pentium III CPU and 1GB memory, and two of them were IBM IntelliStation workstations with 930 MHz Pentium III CPU and 512MB memory. The four server machines had Windows 2000 Server and the two workstations had Windows 2000 Professional. All machines had 20-30GB disk space. All machines were on a LAN with bandwidth of 100Mbits/second.

We installed an *instance* of IBM WebSphere Commerce Suite (WCS) V5.1 on each server system, which includes the IBM HTTP Server (repackaged Apache), WebSphere Application Server (WAS), and DB2 V7.1. We also deployed the ECDW store application (JSP files, HTML files, Java class files, EJB files, etc.) in the WCS instance on each sever system, and created the store database with the large data size (3.5GB) using the scripts and data that come with the benchmark. On one workstation computer, we installed SilkPerformer V3.5 to be used as the test driver (web client). On the other workstation machine, we installed IBM WebSphere Edge Server V3.6 to be used as a network dispatcher to distribute HTTP requests to multiple WAS servers.

We configured DB2 as specified by the ECDW benchmark with appropriate buffer pool and log buffer sizes. To intensify the testing for the database server, we set the *think time* (waiting time between web interactions) to be zero in the ECDW client test driver.

3.4 Database Workload Details

We are especially interested in the characteristics of the database workload from the application. This WCS-based benchmark is a canned application. We used DB2's dynamic SQL statement snapshot tool [12] to capture the SQL execution information at the database server side. It reports the SQL statement text (with '?' representing parameter markers – input variables that are bound at query run-time as opposed to compile-time), the total number of executions, total execution time, number of rows read, and other information.

We examined the SQL statements that were captured through the snapshot tool. For the 1-user regular shopping workload, there were 151 distinct query templates (with parameter markers and literals), consisting of 125 read queries, 14 insert statements, and 11 update statements. For the 30-user regular shopping workload, there were 388 distinct query templates, but still the same 14 inserts and 11 updates as in the 1-user workload. This was because while all the insertions and updates were issued as prepared statements with parameter markers, some queries (for example, checking orders of a particular registered user) were issued with literals and not parameter markers. As a result, the different workloads had a fixed numbers of insert and update templates, but had different numbers of selection templates. This large and varying number of query templates made it difficult for us to analyze the SQL query characteristics of the workloads. Since ordering and registering comprised only a small fraction of the workload, in later experiments we focused on browsing-only workloads, which we created by modifying the original regular shopping workloads.

In a browsing-only scenario, all users are guest shoppers and all transactions are browsing only. We also examined the SQL snapshots of 1-user and 30-user browsing-only workloads. The query templates in the browsing-only workloads were fixed. There were a total of 47 query templates, with 27 of them having parameter markers, and the other 20 not. All of them were simple OLTP style queries, with only 15 of them having joins among two to four tables and the other 32 being single-table selection queries. In total, the browsing only workloads involved 51 tables.

The number of executions of these query templates in browsing-only workload is shown in Figure 9. The top 12 most accessed query templates all had parameter markers in them. Four of them were joins and the other eight were selection queries. Collectively they accessed 15 tables (less than 1/3 of the involved tables of the workload) and consisted of 88% of the total number of SQL executions.

Moreover, in regular shopping workloads, we observed that there was a large degree of overlap between tables with inserts and updates – of the 11 tables with updates and 14 with inserts (there was no deletion in the workload), 9 had both inserts and updates. We observed that there was little overlap between read-only tables with queries and tables with updates and inserts. Only two tables ("userreg" and "users") were subject to inserts, updates and selects, only one table ("member") was both queried from and inserted into, and one table ("keys") was queried from and updated to. We also examined if any updates/inserts happened on the tables that were involved in the top 12 most frequent query templates. We found that there was only one such table (the table "users" accessed by the 6th most frequent query template).

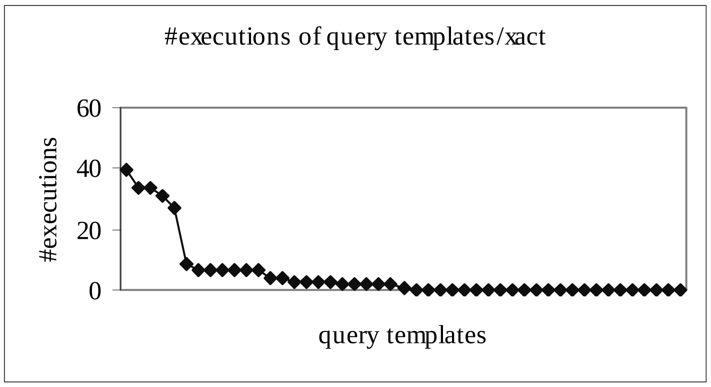

Figure 9: Number of Executions of Query Templates in Browsing

In summary, we observed that e-Commerce workloads had short query execution time, highly skewed popularity of tables, and clean separation of read-dominant and write-dominant tables. These characteristics make middle-tier database caching very attractive.

4 Experimental Results

First, we compared performance of regular shopping scenarios with that of browse-only scenarios. Then, we measured the overhead of adding a middle-tier database cache by using a database cache with 0% hit rate. Then, we cached tables for the top 6 most frequent queries at the middle-tier and measured its performance varying the workload on the backend database server. We then explored update propagation cost in the caching scheme. Finally, we examined the performance of clustered topologies.

4.1 Comparing Workload Characteristics

We tested the regular shopping scenario in the single box topology. We varied the number of concurrent users at the simulator and measured each user executing 100 web transactions. The backend database was restored after each run so that each run started with the same database content. The throughput is reported in terms of average number of web transactions per second, and the average response time is reported in terms of the number of seconds per web transaction. We also report the average response times (in seconds) per ***web interaction*** as "tBrowse", "tBuy", "tOthers", and "tOverAll". "tBrowse" refers to the time spent in browsing. "tBuy" includes the time spent in adding items to shopping carts, filling out address book, displaying order information, working out shipping details, and ordering. "tOthers" refers to the time spent in registered user logging on and new users registering. "tOverAll" is the weighted average response time per web interaction for all types of web interactions, with the weights being the occurrences of each type of interaction. These numbers are shown in Table 1. We also ran the browsing only workload on a single box server topology varying the number of concurrent users as shown in Table 2.

#users	1	5	10	30
#xacts/sec	0.5	0.8	0.8	1.0
secs/xact	1.9	6.2	11.2	30.1
TBrowse	0.1	0.5	0.8	3.0
TBuy	1.0	1.9	6.1	5.5
TOthers	0.2	1.0	3.7	3.0
TOverAll	0.2	0.7	1.2	3.2

Table 1: Regular Shopping Scenario on Single Box

#users	1	5	10	30
#xacts/sec	1.3	1.6	1.6	1.6
secs/xact	0.8	3.1	6.4	19.5
tOverAll	0.1	0.4	0.8	2.5

Table 2: Browsing-only Scenario on Single Box

Not surprisingly, both the throughput and response times (per web transaction and per web interaction) of the browsing-only workload improved over those of the regular shopping workload. But both scenarios followed the same pattern: the throughput increased slightly from 1 user to 30 users, while the response time consistently increased proportional to the increase of the number of users. Since browsing represents the majority of the total workload (tOverAll in regular shopping scenario follows closely with the tBrowse value), and browsing-only scenario is much simpler to test, most of the following experiments are focused on browsing-only workloads.

4.2 Examining Overhead of Adding a Front End Cache

Adding a middle-tier database cache at the application server creates overhead by consuming resources on the application server machine. This is a common concern, especially when the middle tier database cache is a full-strength DBMS not a lightweight query processor, so we examine this overhead.

We configured WAS (WebSphere Application Server) on a server machine to let it use a local DB2 server, and used the DBCacheInit tool to create a database of all nicknames in the local DB2 referencing the back-end database in a remote DB2 on another server machine. This makes the local DB2 act as a middle-tier DBCache with a 0% cache hit rate. We compared the performance of this "*dbcache0*" configuration and the remote DB configuration to examine the overhead.

We compare the throughput and web transaction response time of these three configurations for varying number of concurrent users in Figure 10. The remote DB configuration is shown as "remote", and the database cache with all nicknames "dbcache0". As expected, dbcache0 was always worse than the remote DB case, because the backend server was not overloaded. This is because all the queries at the cache are misses and every query goes through two database servers. This made the performance of dbcache0 around one half of the remote DB case under a light load (less than 10 users). But when the number of concurrent users increased to 30, the difference became much less significant. This shows that although using a full strength DBMS as a middle-tier database cache adds some overhead, this overhead is insignificant when the server is fully loaded.

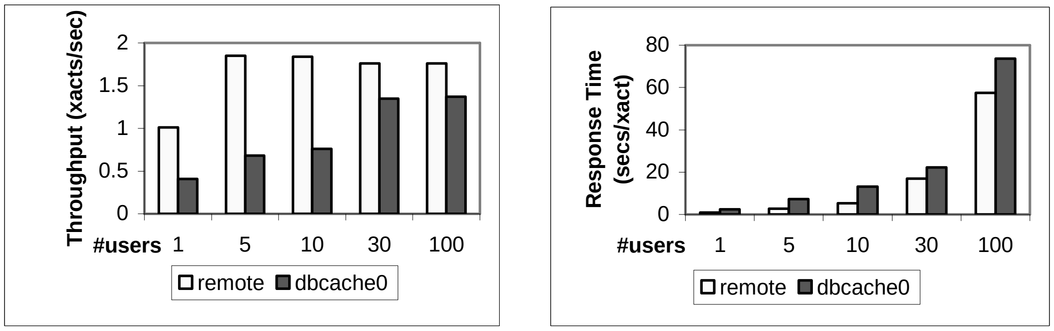

Figure 10: Overhead of Adding a Front End Cache with a 0% Cache Hit Rate

4.3 Examining Server Workload Sharing

In real world scenarios, e-business applications have a large number of online users, and the load can be vary by a factor of 100 in daily operations [5]. When the backend database server is more heavily loaded, caching in the front ends is even more important to improve users' response time. Therefore, we set up a middle-tier cache database at a WAS machine as the front end and measured its performance varying the workload on the backend server.

From previous investigation on the browse-only workloads, we observed that the accesses of different query templates were highly skewed. We selected the top 6 most used query templates and cached in the local database the eight tables that they accessed: in the local database. Queries on these eight tables consisted of ***71%*** of the database queries in the browse-only scenario. In the later experiments, we also used this cache configuration for all DBCache cases.

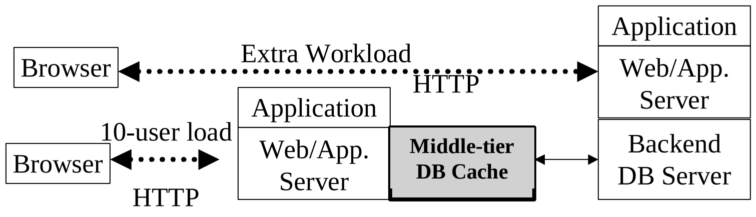

Figure 11: Setup for Varying Server Workload

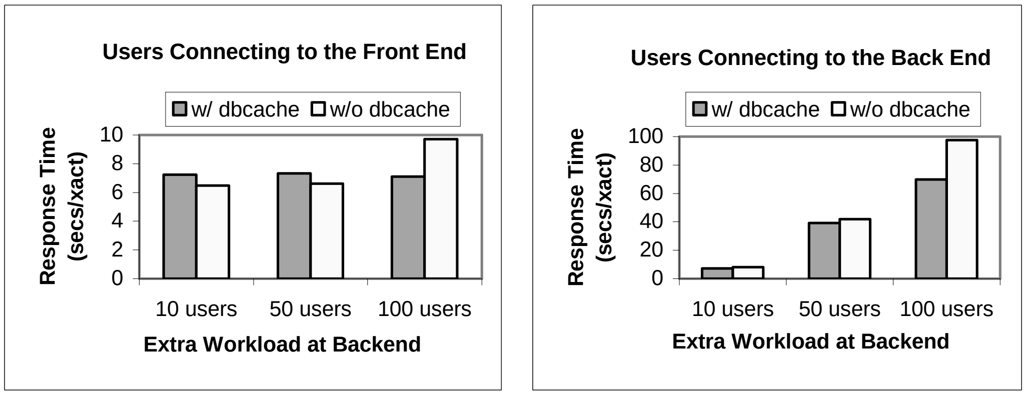

Figure 12: Caching Effect With Varying Server Workload

The setup for varying the backend server workload in the DBCache topology is shown in Figure 11. The goal is to vary the backend server workload and examine how caching can help. We achieved this by sending an extra browsing only workload to the backend directly. The setup for varying the backend server workload in the Remote topology is similar except that there is no middle-tier dbcache at the front end. Both the front end response time and the backend response time were measured at the test driver (web client side). We compare the response times in the dbcache case with those in the remote case.

In Figure 12, we see that when the extra workload on the backend database was 10 or 50 users, adding a cache at the application server did not help. But when the number of extra users on the backend reached 100, caching started to make a difference. The front end response time was improved because the cache sheltered its users from the overloaded backend database server, and the backend response time was improved because the cache shared its server workload. Due to resource constraints, we were not able to test more than 100 users. But we believe that this caching benefit will be even more significant when the backend database server is more heavily loaded.

4.4 Examining Update Propagation Cost

This experiment was to examine how much performance impact the asynchronous update propagation process had on the on-line query performance. We set up DPropR on the backend database server and the WAS server with a DBCache. The capture program was running on the backend database server, and the apply program on the cache database. The cache database still had the eight tables cached and the other tables uncached. Since the cached "users" table was updated frequently in a regular-shopping scenario to update the "lastSession" field with the timestamp of the last log-on session for each registered user, we subscribed the "users" table for update propagation with the minimum frequency of 1 minute.

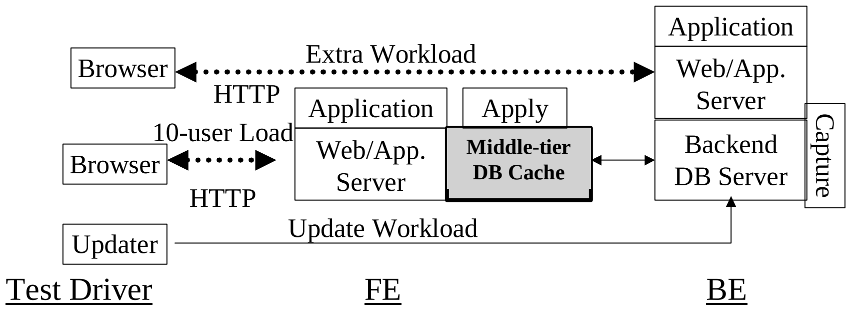

Figure 13: Setup of DPropR with Varying Server Workload

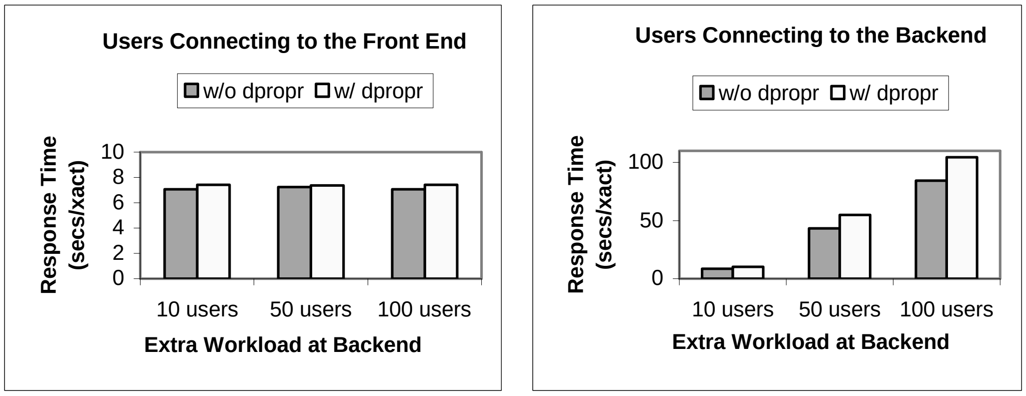

Figure 14: Update Propagation Cost with Varying Server Workload

We still used the same extra workloads on the backend as in the previous experiment, and examined the update propagation cost in this setting. Besides sending a 10-user browsing-only workload to the front end WAS server with a DBCache and sending extra browsing-only workload to the backend database server with a WAS clone directly, we also sent an update workload on the "lastSession" field of the "users" table to the backend database server (shown in Figure 13). This "lastSession" field was updated with the current timestamp to simulate the actual update in a regular-shopping scenario when a registered user logged on. The update workload was executed without any waiting time between consecutive updates. The update throughput was measured to be 40-60 updates/second depending on the server load. We measured the performance impact of update propagation on the browsing workloads by measuring the response times at the simulated browsers.

We compared the with-dpropr-running case with the without-dpropr-running case when the same updater was updating the backend database server. Figure 14 shows that in general asynchronous update propagation did not add significant overhead to the response time, although the capture program on the backend database server incurred around 20% overhead to the query workload when the extra load on the server was 100 users. The overhead caused by the apply program was low because the apply program batched up updates for each propagation interval (1 minute), and each experiment lasted 20-30 minutes. The overhead caused by the capture program was reading the log and, when a log record relating to a subscribed table is found, performing an SQL insert into the "changed data" table (one of the control tables used by DPropR).

4.5 Clustering Web Application Servers

Finally, we compared the performance on clustered topologies ("2-WAS" and "3-WAS") with that on corresponding non-clustered topologies ("1-WAS") to see how they were scaling with the number of WAS machines. Figure 15 shows the throughput and response times of a 30-user browsing-only workload when the backend database is heavily loaded under a CPU hog program.

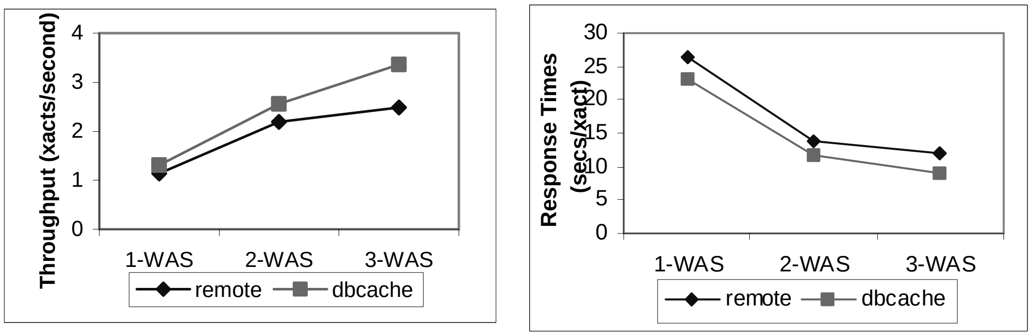

Figure 15: Varying Number of WAS machines

When the number of WAS machines increased, both clustered topologies improved the user response time, but clustered dbcache improved the throughput more than clustered remote DB topology. This implies that (1) For the clustered remote DB topology, simply increasing the number of application servers does not scale up the entire system under a heavy load, and causes the back-end database server to become the bottleneck. (2) Clustered DBCache topology shares the backend database server workload, and it can scale up throughput better by adding more cache nodes. We are interested in adding more WAS nodes to further examine the scale-up effect for clustered topologies.

4.6 Discussion

By running the benchmark and its modifications in various configurations, we show that both application servers and back-end database servers can be bottlenecks under different workloads. Application servers are mostly CPU intensive under e-Commerce workloads, but they can scale to a large number of users by replicating (together with replicated applications) to multiple nodes. In general, single node commercial database servers consume much less CPU resource than application servers, but they can also become a bottleneck under heavy loads.

One approach to scaling the back-end database when it is a bottleneck is to use a more expensive SMP/MPP system – while this approach helps increase the scalability of the system, it does not address the performance, flexibility and availability concerns listed in Section 1. It is also more expensive compared to the DBCache approach where cheaper and less reliable machines can be used to run the application servers with DBCache.

Due to resource constraints, we were not able to test more than 100 simulated users, more than 3 WCS nodes, or separating the application servers from the DB server by a wide area network. However, from the trends shown in the experiments, we believe that middle-tier database caching on the application servers can improve server scalability. If these data caches are deployed with edge servers, they can also bring content closer to users and improve performance in terms of response time. Performance can also be enhanced when the application server and the database are geographically separated by a wide-area network, as is common for many customers. Finally, by continuing to provide limited service based on cached data, this approach also increases availability of the web site. Many issues relating to database caching in application and edge servers are discussed in [18].

5 Related Work

Products most relevant to ours are the Database Cache of Oracle’s 9i IAS (Internet Application Server) [19] and TimesTen's Front-Tier [22]. Oracle's Database Cache caches full tables using a full-fledged Oracle DBMS, and relies on replication tools to asynchronously propagate updates from the backend database to the cache. TimesTen's Front-Tier is a caching product based on their in-memory database technology. One advanced feature of Front-Tier is that users can create cache views at the Front-Tier, which can be a *subse*t of tables and *join* views. Unlike Oracle and our DBCache, updates are performed at the Front-Tier cache, and propagated to the backend database at transaction commit time (or the propagation to the backend can also be done asynchronously).

A major difference between our work and these existing products is that our cache has distributed query processing capability. This is because we leverage DB2's federated features so that query plans at the cache can involve both sites in a cost-based manner [20]. In contrast, Oracle’s query routing happens at the OCI layer before a statement reaches the cache database. Consequently, the statement is either entirely executed at the backend database or entirely at the cache database. Similarly, applications using TimesTen's Front-Tier must be aware of the cache content and issue queries on cached content and on the backend database separately.

Caching for data-intensive web sites have been recently studied in [2], [3], [7], [16], [17], and [24]. They focused on caching dynamically generated web pages, HTML fragments, XML fragments, or query results from outside of a DBMS (except [24] investigated using the backend database to cache intermediate query results as materialized views). Our focus is to engineer a full-strength DBMS into a middle-tier database cache from inside out, and improve availability and performance for applications without making any changes to them.

Finally, previous work on materialized views [9] and caching for heterogeneous systems ([1], [4]), client server database systems ([6], [14]), and OLAP systems [8] are also relevant to our work. Most of the techniques proposed in these papers are suitable for specific types of applications, for example, keyword based search, mobile navigation, or computation intensive OLAP queries. Compare to these applications, e-Commerce applications are usually simple OLTP-style queries but requires reliability, scalability, and maintainability. Consequently, we choose simple table level caching using an industrial strength DBMS.

6 Conclusions and Future Work

We have examined the opportunities in e-Commerce applications for middle-tier database caching by running an e-Commerce benchmark on typical web site architectures. We observed that e-Commerce applications generated a large number of simple OLTP-style queries, their table accesses are highly skewed on a few read-dominant tables, and there was a clear separation between write-dominant tables and read-dominant tables. We demonstrated that web application clones could scale up to heavy loads and the backend database server eventually becomes the performance bottleneck in the system.

We have presented our prototype implementation of a middle-tier database cache. By extending DB2's federated features we turned a DB2 instance into a DBCache without changing user applications. The novelty of this extension is that query plans at the cache may involve both the cache and the remote server based on cost estimation. Through experiments, we showed that the overhead of adding a full-strength DBMS as a middle-tier database cache was insignificant for e-Commerce workloads. Consequently, middle-tier database caching improved users response time significantly when the backend database server was heavily loaded.

Future work includes extending the DBCache prototype to handle special SQL data types, statements, and user defined functions. We are also investigating alternatives for handling updates. Usability enhancements, such as cache performance monitoring, and dynamically identification of candidate tables for caching are important directions for us to pursue.

Acknowledgements

We would like to thank Mehmet Altinel for his helpful suggestions on the paper, Larry Brown for setting up and testing the prototype on NT platforms, and Kiran Mehta and Dan Wolfson for discussions about our architecture. We would also like to thank Satya Dash, Joseph Fung, Jim Kleewein, Peter Schwarz, and David Tolleson for answering our questions on the ECDW benchmark, DB2 federated features, and DPropR.

References

[1] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *SIGMOD Conference 1996*: 137-148.

[2] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. *SIGMOD Conference 2001.*

[3] Jim Challenger, Arun Iyengar, and Paul Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. *IEEE INFOCOM 99*.

[4] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Cache Mechanism for Heterogeneous Web Querying. *Proc. 8th World Wide Web Conference (WWW8),* 1999.

[5] Mike Conner, George Copeland and Greg Flurry. Scaling Up e-Business Applications with Caching. *DeveloperToolbox Technical Magazine*, August 2000. http://service2.boulder.ibm.com/devtools/news0800/art7.htm

[6] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, and Divesh Srivastava, Michael Tan. Data Caching and Replacement. *VLDB 1996*.

[7] Anindaya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Krithi Ramamritham, and Dan Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. *VLDB 2001*.

[8] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching Multidimensional Queries Using Chunks. *SIGMOD Conference 1998*: 259-270.

[9] Ashish Gupta and Inderpal Singh Mumick (Editors). Materialized Views: Techniques, Implementations, and Applications. The MIT Press, 1999.

[10] IBM. DB2 DataJoiner. http://www-4.ibm.com/software/data/datajoiner/

[11] IBM.DB2 DataPropagator. http://www-4.ibm.com/software/data/DPropR/

[12] IBM. DB2 System Monitor Guide and Reference. http://www-4.ibm.com/cgi-bin/db2www/data/db2/udb/winos2unix/support/document.d2w/report?fn=db2v7f0frm3toc.htm

[13] IBM. WebSphere Commerce Suite. http://www-4.ibm.com/software/webservers/commerce/wcs51.html

[14] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal 5(1)*: 35-47 (1996).

[15] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. Cache Investment: Integrating Query Optimization and Distributed Data Placement. *ACM TODS*, December 2000

[16] Alexandros Labrinidis and Nick Roussopoulos. WebView Materialization. *SIGMOD Conference 2000*.

[17] Qiong Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. *VLDB 2001*.

[18] C. Mohan. Caching Technologies for Web Applications. *VLDB 2001*. http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf

[19] Oracle Corporation. Oracle Internet Application Server Documentation Library. http://technet.oracle.com/docs/products/ias/doc_index.htm

[20] Mary Tork Roth, Fatma Ozcan, Laura M. Haas: Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. *VLDB 1999*

[21] Segue Software, Inc. SilkPerformer. http://www.segue.com/html/s_solutions/s_performer/s_performer.htm

[22] TimesTen. TimesTen Front-Tier. http://www.timesten.com/products/fronttier/index.html

[23] Transaction Processing Performance Council. TPC-W Benchmark. http://www.tpc.org/tpcw/default.asp

[24] Khaled Yagoub, Daniela Florescu, Valérie Issarny, and Patrick Valduriez. Building and Customizing Data-Intensive Web Sites Using Weave. *VLDB 2000*.