MTCache: Transparent Mid-Tier Database Caching in SQL Server

Per-Åke Larson
Microsoft
palarson@microsoft.com

Jonathan Goldstein
Microsoft
jongold@microsoft.com

Jingren Zhou
Columbia University
jrzhou@cs.columbia.edu

Abstract

Many applications today run in a multi-tier environment with browser-based clients, mid-tier (application) servers and a backend database server. Mid-tier database caching attempts to improve system throughput and scalability by offloading part of the database workload to intermediate database servers that partially replicate data from the backend server. The fact that some queries are offloaded to an intermediate server should be completely transparent to applications - one of the key distinctions between caching and replication. MTCache is a prototype mid-tier database caching solution for SQL Server that achieves this transparency. It builds on SQL Server’s support for materialized views, distributed queries and replication. This paper describes MTCache and reports experimental results on the TPC-W benchmark. The experiments show that a significant part of the query workload can be offloaded to cache servers, resulting in greatly improved scale-out on the read-dominated workloads of the benchmark. Replication overhead was small with an average replication delay of less than two seconds.

1. Introduction

Many applications today are designed for a multi-tier environment typically consisting of browser-based clients, mid-tier application servers and a backend database server. Application servers do not maintain persistent state and typically run on fairly inexpensive machines. Hence, bottlenecks in the application server tier can be solved easily and cheaply by increasing the number of servers.

All persistent state is maintained by the backend database, which typically runs on a single high-end machine. A user request may cause tens or even hundreds of queries against the backend database. The overall response time seen by a user is often dominated by the aggregate query response time, particularly when the backend system is highly loaded. To improve performance and scalability, one must either reduce the load on the backend server or increase its capacity. The goal of mid-tier database caching is to transfer some of the load from the backend database server to intermediate database servers. An intermediate server has a local database storing a copy of some of the data from the backend database, which allows some queries to be computed locally.

A key requirement of mid-tier database caching that also distinguishes caching from replication is that it must be transparent to applications. That is, adding caching should not require any changes in applications. In particular, applications should not be aware of what is cached and should not be responsible for routing requests to the cache or to the backend server. If applications are required to route requests to the appropriate server, the caching strategy cannot be changed without changing applications.

This paper describes MTCache [21], a mid-tier database cache solution for Microsoft SQL Server that achieves this goal. We describe our design and prototype implementation and report experimental results on the TPC-W benchmark. MTCache is based on the following approach.

• A shadow database is created on the MTCache server that contains the same tables as the backend database and also the same constraints, indexes, views, permissions. All tables are empty but the statistics maintained on tables, indexes and materialized views reflect the data on the backend server.

• What data to cache is defined by creating materialized views on the MTCache server. These materialized views may be selections and projections of tables or materialized views on the backend server.

• The materialized views on the MTCache server are kept up to date by SQL Server replication. When a view is created, a matching replication subscription is automatically created and the view is populated.

• All queries are submitted to the MTCache server whose optimizer decides whether to compute a query locally, remotely or part locally and part remotely. Optimization is entirely cost based.

• All inserts, deletes and updates are submitted to the MTCache server, which then transparently forwards them to the backend server.

MTCache exploits SQL Server’s support for materialized views, distributed queries and transactional replication. The use of replication is not inherent in our design; other mechanisms for propagating updates like two-phase commit could be used. Our approach resembles the ap-
proach taken by DBCache [2][3][14][20]. However, DBCache appears to be limited to caching of complete tables while we also allow caching of horizontal and vertical subsets of tables and materialized views. In addition, DBCache appears to always use the cached version of a table when it is referenced in a query, regardless of the cost. In MTCache this is not always the case: the decision is deeply integrated into the optimization process and is entirely cost-based. For instance, if there is an index on the backend that greatly reduces the cost of the query; it will be executed on the backend database.

MTCache also includes improved optimization of parameterized queries that results in more efficient use of cached data. This is done by producing dynamic plans where the decision whether to use a cached view is made at run time based on the actual parameter values. This is the first implementation of dynamic plans in an industrial-strength database system. Dynamic plans are crucial for the performance of parameterized queries in a caching environment because they exploit the cached data efficiently while avoiding the need for frequent reoptimization.

TimesTen also offers a mid-tier caching solution built on their in-memory database manager [15][16][17]. Their product provides many features but the cache is not transparent to applications.

Caching in other contexts has been studied intensely. Work on caching for data-intensive web sites is covered in several papers [4][5][6][9][12][13]. The focus is on caching web pages, fragments of web pages, or query results typically outside a database system. However, [19] considers caching query results (as materialized views) inside the database system. This type of caching is complementary to mid-tier database caching and one can easily envision a system that uses both types of caching.

Previous work on caching for heterogeneous systems [1][7] and client-server database systems [8][11] is also relevant to mid-tier database caching.

The rest of the paper is organized as follows. Section 2 provides a brief overview of SQL Server’s support for distributed queries and replication. Section 3 explains the overall architecture of MTCache. Section 4 outlines the steps needed to set up an MTCache server and define what data to cache. Section 5 deals with query processing and optimization and describes required optimizer extensions. Section 6 covers experimental results obtained on the TPC-W benchmark. Section 7 contains our conclusions and outlines potential future work.

2. Background on SQL Server

As our current implementation for mid-tier database caching relies on SQL Server’s support for distributed queries and replication, we briefly review these aspects of SQL Server before describing the details of our solution. Note that our approach is not tied to replication – it works equally well with any other (synchronous or asynchronous) mechanism for update propagation.

2.1 Distributed queries and updates

SQL Server’s support for distributed queries builds on the notion of linked servers. A linked server may be another instance of SQL Server, an Oracle Server, a DB2 server, or any data source that can be reached through OLE DB and has at least minimal SQL support. A query can access tables on one or more linked servers and the local server. Currently, query optimization is largely heuristic: the optimizer attempts to push down the largest possible subquery to a linked server. Any remaining processing required to combine results from different servers is performed locally. Tables on linked servers can be updated in the same way as local tables.

Suppose we have registered another instance of SQL Server as a linked server called “PartServer”. We can then write queries that combine data from the local database and data from PartServer.

```sql
SELECT ol.id, ps.name, ol.qty
FROM orderline ol,
     PartServer.catdb.dbo.part ps
WHERE ol.id = ps.id
  AND ol.qty > 500
  AND ps.type = 'Tire'
```

The query above joins the local table Orderline with the remote table Part on PartServer. If the linked server supports selection, the optimizer would push the query “select id, name from catdb.dbo.part where type = ‘Tire’” to PartServer.

Distributed transactions are supported provided that the participating linked servers support two-phase commit, which SQL Server does. This is done through Microsoft Distributed Transaction Coordinator (DTC). An application can, for example, update some local data and data on several linked servers all within a single (distributed) transaction.

2.2 Replication

This brief overview covers SQL Server transactional replication, which is the form of replication used by our prototype. Other forms of replication supported by SQL Server are not discussed.

SQL Server replication is based on a publish-subscribe paradigm. A publisher (source) makes data available for replication through a distributor (middleman) that propagates changes to subscribers (targets). A publisher defines what data it allows to be replicated by creating one or more publications consisting of a set of articles. An article is defined by a select-project expression over a table or a materialized view. In other words, an article may contain only a subset of the columns and rows of the
underlying table or materialized view. A subscriber specifies what data it wishes to receive by subscribing to the desired publications from one or more publishers.

Changes to a published table or view are collected by log sniffing. A log reader process collects applicable changes from the log and inserts them into a distribution database on the distributor.

The distributor is responsible for propagating the changes found in its distribution database to the affected subscribers. Once changes have been propagated to all subscribers, they are deleted from the distribution database.

Propagation of changes to a particular subscriber can be initiated either by the subscriber (a pull subscription) or by the distributor (a push subscription). The propagation is performed by a separate agent process that wakes up periodically, checks for changes and, if there are any, applies them. The difference is whether the agent runs on the subscriber machine or the distribution machine.

Change propagation is done one complete (committed) transaction at a time in commit order. This ensures that a subscriber always sees a transactionally consistent state but it may not be the absolutely latest state.

3. System architecture

We assume an initial configuration consisting of one or more application servers running IIS (Microsoft’s web and application server) and a backend database server running SQL Server. Application servers connect to the backend server through ODBC connections. Suppose the backend database server has become overloaded, resulting in poor response times. The goal is to switch some of the load to smaller and cheaper intermediate cache servers (running SQL Server) without having to modify applications in any way.

Figure 1 illustrates a system with two MTCache servers. As shown in the figure, an MTCache server can run either on the same machine as the application server or on a separate machine, possibly handling requests from more than one application server.

A caching database is a “shadow” of the corresponding database on the backend server. The shadow database contains exactly the same tables, views, indexes, constraints, and permissions as the backend database but all the shadow tables, indexes and materialized views are empty. However, all statistics on the shadow tables, indexes and materialized views reflect their state on the backend database.

Shadowing the backend catalog information on the caching server makes it possible to locally parse queries, perform view substitution and check permissions. The shadowed statistics are needed for query optimization, as explained in more detail later.

The data actually stored in an MTCache database is a subset of the data from the backend database. The subset consists of materialized select-project views of tables or materialized views on the backend server. What data to cache is user controlled. An administrator determines a caching strategy and implements it by creating a collection of materialized views on the MTCache server.

The cached views are kept up to date by replication. When a cached view is created, we automatically create a replication subscription matching the view. If no suitable publication exists, one is automatically created first. The cached data can be thought of as a collection of distributed materialized views that are transactionally consistent but may be slightly out of date.

Note that a cache server may store data from multiple backend servers. Each shadow database is associated with a single backend server but nothing prevents different databases on a cache server from being associated with different backend servers. The same applies to replication: different databases may receive data from different distributors.

4. Enabling caching

This section describes the steps required to set up an MTCache server. If replication has not already been enabled, it must be first configured by specifying which machines will act as publishers, distributors, and subscribers. This is done using standard SQL Server tools (Enterprise Manager). This step sets up the necessary infrastructure on the machines involved. For example, it creates the distribution database on the distributor.
Setting up SQL Server to cache data from a target database is a fairly simple process, consisting of creating and executing two SQL scripts and rerouting ODBC sources.

- An automatically generated script that configures the cache server and sets up the shadow database.
- A manually created script that creates the cached materialized views and the corresponding replication subscriptions.
- Rerouting the application’s ODBC sources from the backend server to the cache server.

The first script can be generated automatically by a few clicks using standard SQL Server scripting tools (Enterprise Manager). The initial script contains SQL commands to create a shadow database with tables, views, indexes and permissions matching the target database on the backend server. The script is then slightly augmented by running it through a small application that we wrote.

The augmented version of the script is then run on the target server, which configures the server and creates the shadow database. The MTCache server is now ready for service but not very useful because it contains no data.

The DBA must decide what data to cache, that is, what cached materialized views to create. Cached views can be selections and projections of tables or materialized views residing on the backend server. When a cached view is created on the mid-tier server, we automatically create a replication subscription (and publication if needed) and submit it. Replication then immediately populates the cached view and begins collecting and forwarding applicable changes.

The DBA manually writes a script to create the desired views and runs the script. Additional, identically configured, MTCache servers can then be configured by running the two scripts.

Once the shadow database and the cached views have been created, applications can begin using the MTCache server. In Windows, applications connect to a database server by connecting to an ODBC source. An ODBC source definition maps a logical name to an actual server instance and specifies a number of connection properties. To cause an application to connect to the mid-tier server instead of the backend server, we only need to redirect the appropriate ODBC sources from the backend server to the mid-tier server. This can be done using standard GUI tools. No changes are required to the application.

5. Query optimization and processing

The processing of queries and updates relies on SQL Server’s distributed query capability. All insert, delete and update requests against a shadow table are immediately converted to remote inserts, deletes and updates and forwarded to the backend server.

Queries go through normal optimization on the mid-tier server. Cached views are treated as regular materialized views and, if applicable, picked up by the optimizer’s view matching mechanism [10]. However, even if all the necessary data is available locally, the query is not necessarily evaluated locally. It may be faster to evaluate the query on the backend if, for example, the backend has an index that greatly speeds up processing. The reverse is also true; even if none of the required data is available locally, it may be worthwhile evaluating part of the plan locally. An extreme example is a query that computes the Cartesian product of two tables. It is cheaper to ship the individual tables to the local server and evaluate the join locally than performing the join remotely and shipping the much larger join result. The optimizer needs to make a cost-based decision on whether to evaluate a subexpression locally or remotely and not rely on heuristics.

To make a cost-based decision, the optimizer must be able to find the best plan for evaluating a subexpression on the backend server. This could be done in two ways:

- **Remote optimization**: Send the subexpression to the backend server and have it return the estimated cost and cardinality.
- **Local optimization**: Replicate the required catalog information and statistics on the local server and optimize the subexpression locally.

We rejected remote optimization because of the high expected overhead. The optimizer may have to optimize hundreds of subexpressions to find the best overall plan. The overhead of packing, transferring, and unpacking hundreds of subexpressions would greatly increase optimization time.

Making local optimization work required some modifications of the optimizer. To include the cost of transferring data from the backend server to the local server, we introduced a new operator, DataTransfer, and a new physical property, DataLocation, that can be either Local or Remote. The DataTransfer operation simply changes the DataLocation property from Remote to Local or vice versa. All other operators leave DataLocation unchanged. (The DataTransfer operator is actually implemented by overloading the Project operator.) Cached views and their indexes are Local and all other data sources are Remote. The required DataLocation on the root of the query is Local.

We also created a new optimization rule that adds a DataTransfer operation whenever the parent requests a Local result and the input expression is Remote. DataTransfer is a (physical) property enforcer, similar to sorting. The estimated cost of a DataTransfer operation is proportional to the estimated volume of data shipped plus a constant startup cost.
We modified cost estimation to favor local execution over execution on the backend server. All cost estimates of remote operations are multiplied by a small factor (greater than 1.0). The motivation is that, even though the backend server may be powerful, it is likely to be heavily loaded so we will only get a fraction of its capacity.

The final plan produced by the optimizer may be a completely local plan, a completely remote plan or a combination thereof. Subexpressions to be evaluated remotely are easy to find in the plan: simply look for DataTransfer operators. Every subexpression rooted by a DataTransfer operator is converted to a (textual) SQL query and sent to the backend server where it will be parsed and optimized again. Unfortunately, queries can only be shipped as textual SQL at this time. If it were possible to ship execution plans instead, reoptimization on the backend server could be avoided.

5.1 Parameterized queries

Parameterized queries require special care if we want to make maximal use of the cached data. Suppose we have a cached selection view, Cust1000, containing all customers with customer ID less than or equal to 1000 and receive the query

```
select cid, cname, caddress
from customer
where cid < @p1
```

where @p1 is a run-time parameter. (Parameters and local variables must be prefixed by “@” in T-SQL.) Whenever the actual value of @p1 is less than or equal to 1000, we can compute the query locally form the cached view, otherwise we have to compute it remotely.

Unfortunately, the actual parameter value is only known at run time, not at optimization time. We modified the optimizer to exploit views like Cust1000 whenever possible by generating plans with two branches, one local branch and one remote branch, and selecting the appropriate branch at run time. Even though we discuss the optimizer enhancements in the context of cached views, the implementation is general and applies to all materialized views. To the best of our knowledge, this is the first implementation of dynamic plans in a commercial database system.

Figure 2 (a) shows the conceptual plan generated for the example query. During view matching we recognize that the view Cust1000 contains all required rows when @p1 <= 1000 but not otherwise. We then create an alternative with a ChoosePlan operator on top having the guard predicate @p1 <= 1000 and two children, one using the cached view Cust1000 and the other using the (remote) Customer table. During execution, the left branch is chosen if the guard predicate evaluates to true, otherwise the right branch.

```
Figure 2: Logical plans generated for example query with run-time parameters.
```

To avoid having to introduce a completely new operator, the ChoosePlan operator is actually implemented using a UnionAll operator and two Select operators as shown in Figure 2 (b). A UnionAll operator simply concatenates the results produced by its inputs. The guard predicate references only the parameter value and no columns and can therefore be implemented as a Select operator with a startup predicate. A startup predicate is evaluated only once, when the operator is first opened. If it evaluates to false, the operator’s input expression is not opened because every row it might produce would be rejected by the startup predicate. The guard predicate becomes the startup predicate for the left branch and its negation becomes the startup predicate for the right branch. Hence, depending on the current value of @p1, the left branch or the right branch will be executed but not both.

The cost of the combined plan depends on which branch of the plan is used. We estimate the cost of the combined plan as a weighted average of the cost of the

---

1 Later versions do implement a separate operator for choosing among plan branches at run time but the results reported here were all obtained using the UnionAll implementation.
two input plans. Suppose the estimated costs of the input plans are $Cl$ and $Cr$, respectively, and the left plan is used with frequency $Fl$ and the right plan with frequency $1-Fl$. Then the cost of the combined plan is computed as $Fl \cdot Cl + (1-Fl) \cdot Cr$. $Cl$ and $Cr$ are known but we must somehow estimate the frequency $Fl$.

$Fl$ is the probability of the guard predicate evaluating to true, which for our example query is the probability of the actual parameter value being less than or equal to 1000. To estimate $Fl$, we would need the frequency distribution of the actual parameter values. Unfortunately, this distribution is not available. Lacking any better information, we currently estimate $Fl$ under the assumption that $@p1$ is uniformly distributed between the min and max values of the column cid. Another possibility would have been to use the distribution of the column values but there is no particular reason to assume that parameter values follow the same distribution as column values.

5.1.1 Plans producing “mixed” results

Note that our implementation has the property that all rows in the query result will originate either from the view Cust1000 or from the Customer table but never from both, i.e., the plan never produces a “mixed” result. This is not the only option; one could also allow plans to produce “mixed” results where required rows not found in the view are obtained from the Customer table.

Figure 3 shows a plan for the example query that may produce a “mixed” result.

5.1.2 ChoosePlan algebraic properties

We also need to consider how the ChoosePlan operator interacts with other operators during optimization. The following query joining customers and orders illustrates the issue.

```sql
select c.name, o.date, o.total
from customer c, orders o
where c.ckey = @p1
  and c.ckey = o.ckey
```

Figure 4 (a) shows the initial plan obtained after view matching against the expression “select ckey, name from Customer”. Figure 4 (b) shows the plan obtained by pulling the ChoosePlan operator above the join. Is this transformation valid? If it is, what are the benefits and drawbacks of applying the transformation?
It is easy to show that the transformation is valid by considering two cases separately. If \( @p \leq 1000 \), the guard predicate is true and the left branch is chosen in both plans. That is, the active part of the plan is obtained by dropping the right branch and eliminating the ChoosePlan operator. The active parts of the two plans are clearly equivalent for this case, in fact, they are identical. If \( @p > 1000 \), the guard predicate is false and the right branch is chosen in both plans. As in the previous case, the active parts of the two plans are clearly equivalent. Hence, the transformation is valid (in both directions).

Pulling the ChoosePlan operator above the join may produce a better plan because the two branches can now be optimized independently. The optimizer may, for example, choose different join methods for the two branches. More importantly, it expands the expression in the right branch, giving the optimizer the opportunity to push a “larger” query to the backend server. However, the transformation has two drawbacks. It increases optimization time and the final plan may be larger than minimally needed.

So far we have only discussed the interaction between ChoosePlan and equi-join. The proof idea for equi-join can be applied to other operators to show that ChoosePlan can be pulled above any query operator: selection, projection, all types of joins (inner join, outer join, semi-join, anti-semijoin), aggregation, set operations, and even another ChoosePlan operator. This implies that we can always propagate ChoosePlan operators to the top of the plan tree. Doing so, gives the optimizer maximal freedom but may increase optimization time and final plan size.

5.2 Stored procedures

Stored procedures (written in T-SQL) are widely used by applications running on top of SQL Server. A stored procedure may contain significant application logic and contain many parameterized queries. They are, in fact, the primary source of parameterized queries. If a stored procedure is found locally, we evaluate it on the mid-tier server; otherwise the call is transparently forwarded to the backend server.

By default stored procedures are not copied from the backend server to the MTCache server but the DBA can selectively copy over produces that she wishes to run locally. This gives the DBA an additional degree of control over what work is offloaded to the cache server. Note that a stored procedure can be run locally even when some of the data it requires is not available locally.

6. Experimental evaluation

The primary goal of mid-tier database caching is to improve scalability by offloading some of the query workload from the backend server to cache servers. This improves overall throughput and response time, especially when the backend server is heavily loaded. In this section we report on an experimental evaluation of MTCache using the TPC-W benchmark. The experiments were designed to answer three specific questions.

1. How successful is MTCache at offloading work from the backend server and how does that affect system throughput?
2. How high is the replication overhead on the backend server and on the cache servers?
3. What is the latency for update propagation from the backend server to the cache servers in both high-load and low-load scenarios?

We first give a brief overview of the TPC-W benchmark, our implementation of the benchmark and the computing environment used for the experiments. Subsequent sections report on experiments addressing each of the questions raised above.

6.1 TPC-W Implementation

6.1.1 Overview of TPC-W

TPC-W is a transactional web benchmark developed by the Transaction Processing Performance Council (TPC) [18]. The benchmark simulates an online book seller like Amazon and models a set of users using a web browser to browse and order items available from the book seller’s web site. The benchmark stresses a variety of components required in such a system (web servers, database systems, network). The focus is on total system throughput, not the performance of individual components. The throughput metric used is web interactions per second (WIPS).

The benchmark environment consists of a set of load drivers, web servers, image servers, and a database server. The load drivers, which simulate user behavior, generate a series of page requests to the web servers according to rules specified in the benchmark. The web servers process requests, returning dynamically generated HTML pages. The processing of a request may involve several queries and/or updates against the database server. A response page may contain URLs referencing images. The response page is returned to the load driver, which then retrieves the images referenced on the page from the appropriate image servers.

More specifically, a load driver simulates a number of (logical) user sessions. A user session is identified by a session cookie. The session state includes a shopping basket stored on the database server. Each user has at most one outstanding request and waits a fixed length of time, called the user think time, before issuing a new request. The load on the system can be increased by increasing the number of users, or decreasing the think time. The bench-
The benchmark contains specific latency requirements for each request type, stating that 90% of the requests must complete within a certain time limit. The limit varies with request type but is typically three seconds.

There are fourteen request types providing functionality typical for a storefront site: searching the catalog in various ways, retrieving details about items, updating information about an item, adding to or deleting items from a shopping cart, ordering items in the shopping cart, and retrieving information about orders.

The benchmark specifies three different workloads: Browsing, Shopping, and Ordering. A workload simply specifies the relative frequency of the different request types. The fourteen request types can be divided into two broad activity classes: Browse and Order. The Browse class is dominated by home page requests, search requests and retrieval of item details. The Order class is dominated by requests to modify shopping cart content, customer registration, and order submission. Browse activities generate very little update activity in the database while Order activities are dominated by updates. The table below shows the frequency of the two activity classes in the three workloads.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Browse</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browsing</td>
<td>95%</td>
<td>5%</td>
</tr>
<tr>
<td>Shopping</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>Ordering</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

From a database point of view, the Shopping workload is read-dominated while the Ordering workload is update-dominated. The Shopping workload is somewhere in between the two. It is considered the “main” workload of the benchmark.

The database consists of eight tables containing information about items (books), authors, customers, orders, and credit card transactions. Although not required by the benchmark, shopping cart information is also typically stored in the database (because session state must persist for at least two hours). The shopping cart tables are updated most frequently, followed by tables related to orders, customers, and credit card transactions. Changes to the item table are infrequent but do occur. The author information never changes.

The complete set of queries can be found in the benchmark specification. They vary greatly in terms of cost, ranging from straightforward lookups (get item details given the item id) to queries requiring considerable processing. One of the most expensive and also frequent queries is the bestseller query: among the last 3333 orders, find the 50 most popular items of a given category, ranked by popularity. Title and author search queries (find items containing a given string in the title field or in the author field) are also rather expensive and frequent.

6.1.2 Experimental setup

Our test environment is depicted in Figure 5. We used ten 500MHz single processor machines with 512MB of memory and one machine with dual 500MHz processors and 1GB of memory. Each machine had a 100 MBit Ethernet card, and was connected to a high performance Cisco network switch. The backend database server and the load driver machines ran Windows 2000 Server while all other machines ran Windows .NET Server. The web servers were all IIS 6.0. The backend database ran on Microsoft SQL Server 2000. Of the single processor machines, three were used as load drivers, five as web/cache servers, and two as image servers.

The size of the database depends on two scale factors: number of items and number of emulated users. We set up the database with 10,000 items and 10,000 emulated users. This resulted in a database containing 28.8 M customers, 57.6 M addresses, 25.9 M orders, and 77.8 M order lines.

In our implementation of the TPC-W benchmark, user requests are processed by an ISAPI extension (a DLL) hosted by IIS and all database requests are implemented as SQL Server stored procedures. The implementation is from a Microsoft internal TPC-W Benchmark Kit. No page, fragment or image caching was used, that is, every page returned was generated dynamically by the ISAPI application.
Each web server machine also included a local MTCache, that is, SQL Server 2000 with our modifications for caching. The data cached consisted of projections of four tables: item, author, orders, and orderline. Note that the order and orderline tables are large and updated frequently. This design allowed us to run all search queries locally (title search, search by category, author search, bestseller search) and also a frequent lookup query on items. These queries account for the bulk of the load in the Browse class. Running them locally shifted a significant part of the overall load to the cache server, especially in the Browsing scenario. All indexes on the cache servers were identical to indexes on the backend server, as it would have been unfair to make the backend seem unnecessarily slow as a result of less aggressive indexing.

Of the 29 stored procedures used by the benchmark, we chose to copy 24 to the cache servers. The five that were not copied were update dominated and would not have benefited significantly from running on the middle tier.

6.2 TPC-W Experiments

In this section, we describe a set of experiments designed to answer the three questions posed earlier. As a result, this section is divided into three corresponding subsections. All experiments were performed in the TPC-W environment described above.

Unless explicitly stated otherwise, load was generated for these experiments by fixing the user wait time to one second, and by steadily increasing the number of users per web server until the response latency requirements specified in the benchmark were barely met. This was done as a way of finding a consistent tradeoff between benchmark throughput and response time. All collected information such as page server latency, benchmark throughput, and individual machine load, were collected after the system reached a steady state.

In all experiments, CPUs were the bottleneck. Neither the network nor the disks were saturated in any of the experiments.

6.2.1 Experiment 1 – Scaleout

This set of experiments was designed to answer the first question listed above: How successful is MTCache at offloading work from the backend server and how does that affect system throughput?

To answer this question we first need to establish baseline throughput when all database work is performed by the backend server. We configured all web servers to access the backend database directly. For this case, we measured the throughput shown in the table below for the three workloads. The backend database server was the bottleneck. The CPU load was about 90%; increasing it further increased the response times beyond the limits specified in the benchmark.

<table>
<thead>
<tr>
<th>Workload</th>
<th>WIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browsing</td>
<td>50</td>
</tr>
<tr>
<td>Shopping</td>
<td>82</td>
</tr>
<tr>
<td>Ordering</td>
<td>283</td>
</tr>
</tbody>
</table>

Next we measured throughput with caching enabled. For each of the three workloads, we steadily increased the number of web/cache servers and measured the throughput and the load of the backend database server. In this case, the web/cache servers were the bottlenecks and their CPU load had to be limited to 90% to stay within the response time requirements. The results are shown in Figures 6 (a) and 6 (b).
The most important trend to observe in the figures is the linear increase in throughput (WIPS) as we increased the number of web/cache servers coupled with the low load on the backend server. The measured throughput without caching (at 90% backend CPU load) and with five cache servers are summarized in the table below.

<table>
<thead>
<tr>
<th>Workload</th>
<th>No cache</th>
<th>Five web/cache servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIPS</td>
<td>WIPS</td>
<td>Backend load</td>
</tr>
<tr>
<td>Browsing</td>
<td>50</td>
<td>129</td>
</tr>
<tr>
<td>Shopping</td>
<td>82</td>
<td>199</td>
</tr>
<tr>
<td>Ordering</td>
<td>283</td>
<td>271</td>
</tr>
</tbody>
</table>

For the Browsing workload, almost all database work was successfully offloaded to the cache servers. Even with five mid-tier servers running with maximal load (90% CPU load), the backend server was coasting, showing a CPU load of only 7.5%. The throughput increased to 129 WIPS. This is about as high as one can expect with five processors. The backend server with two processors sustained 50 WIPS, i.e. 25 WIPS per processor. If all the work were offloaded to the web/cache servers, there are five processors and we should expect to see 5*25 = 125 WIPS. The actual number is slightly higher.

Suppose that the load on the backend server continues to increase linearly as we add even more web/cache server. If five web/cache servers generate a backend load of 7.5%, then we should be able to add at least 10 times as many web/cache servers without saturating the backend server. A configuration with 50 web/cache servers would be expected to sustain a throughput of around 1250 WIPS. This is admittedly a speculative analysis but it shows the potential.

The Shopping workload generates more updates, which are always processed on the backend server. Hence, a somewhat smaller fraction of the load is offloaded to the cache servers. With five mid-tier servers, the backend server showed a CPU load of 15.9%, still low but higher than for the Shopping workload. Going through the same calculation, we find that one CPU sustained 41 WIPS so with five processors working, we should expect 5*41 = 205 WIPS. The actual number is 199 WIPS, slightly lower.

A similar speculative analysis as above shows it should be possible to use around 25 web/cache servers before saturating the backend server. Such a configuration should be able to sustain a throughput around 1000 WIPS.

In the Ordering workload, 50% of the interactions involve updates so much less of the work can be offloaded to cache servers. With five mid-tier servers, the backend server load was 55.4% and the throughput was 271 WIPS, slightly lower than the 283 WIPS achieved without caching. In other words, adding five cache servers reduced the load on the backend server somewhat but did not increase throughput. We are in the process of obtaining more detailed measurements to determine where the overheads are.

It is clear the Browsing and Shopping workloads could be scaled out considerably further before the backend becomes saturated. These two workloads are more typical of storefront applications than the Ordering workload. While the Ordering workload could not be scaled out too much further, it should be possible to add three or more middle tier machines and achieve WIPS scores of around 450-500 before the backend becomes saturated. This limited scalability for the Orders workload is understandable since that workload is dominated by updates, which must be run on the backend server.

The answer to our question is therefore that scalability depends on the frequency of updates in the workload. In scenarios typical of storefront applications today, which are dominated by browsing, the benefits offered by MTCache style caching are considerable.

6.2.2 Experiment 2 – Replication Overhead

In this experiment, we evaluate the overhead associated with replication on both the backend and the middle tier. The overhead on the backend is caused by the log reader and distributor, which use the log to find and distribute changes in data to middle tier caches. The overhead on the MTCache servers is the work associated with applying the changes pushed by the distributor.

To determine this overhead, we ran an experiment where we saturated the backend server CPUs using two web servers. All five caches were still being updated with new data, but they were not used to answer queries. We then examined the load on one of the middle tier machines that was not being used as a web server to determine the load on middle tier machines associated with maintaining the cache. We used the Orders workload since it contains the highest portion of updates, and therefore had the highest overhead. The result was that the CPU on the middle tier machine not used as a web server was 15% utilized.

In order to measure the overhead associated with replication on the backend server, we first measured the throughput of the overall system (described in the previous section). We then ran the same experiment with the log reader turned off. This completely eliminates all overhead associated with replication. By comparing the throughput with the log reader turned off to the throughput with the log reader turned on, we can measure the overhead associated with replication on the backend. The result was that when the log read was turned on, we achieved 283 WIPS, while we achieved 311 WIPS when the log reader was turned off, that is, replication caused a 10% reduction in throughput.
In summary, the overhead associated with replication on both the middle tier and the backend servers was small (less than 15%) even for an update-dominated workload.

6.2.3 Experiment 3 – Replication Latency

Another experiment evaluated the latency of update propagation from the commit time on the backend to the commit time on the middle tier under both light loads and heavy loads.

To evaluate the latency under light loads, we ran a single web server with a small number of users following the Ordering scenario, and measured the average latency of updates using the Perfmon performance monitoring tool. The result was that average latency was 0.55 s.

To evaluate the latency under heavy loads, we saturated four of the five web servers using the Ordering scenario and with caching turned on. With the last web server, we saturated the backend by disabling the cache. As a result, four of the five web servers were saturated, and the backend was also saturated. We then measured the average latency of updates. The resulting average latency was 1.67 s, which is still quite acceptable for today’s web driven scenarios.

6.2.4 Conclusions from experiments

In summary, MTCache is an effective way of scaling today’s online storefront scenarios as was demonstrated by the linear TPC-W scalability for the workloads most representative of such applications. For instance, the TPC-W Browsing and Shopping scenarios should achieve scale factors at least in the 10s.

In addition, the overhead (<15%) and latency (<2 s) associated with keeping the middle tier caches up to date are relatively small, and is quite acceptable for today’s web driven scenarios.

7. Future Work

Our initial prototype of MTCache is far from complete and can be improved in several ways. For example, we do not currently refresh the shadowed catalog information. This clearly needs to be done. It would also be desirable to reduce the amount of shadowed catalog information by shadowing only the information relevant to the cached views, the tables they depend on, and associated permissions.

There are currently no tools to help a DBA define a caching strategy by analyzing a workload and providing advice on what cached views to create and where to run stored procedures. Such a design tool would be highly desirable.

People today routinely design application system that, in part, use cached or replicated data that is not entirely fresh. More precisely, the data does not necessarily reflect the current state of the database. However, SQL (the language) and current database systems have no notion of relaxed data freshness requirements. We still cling to the notion that every query must produce a result that reflects the latest state of the database. We would like to see SQL extended so that a query could include an indication of its data freshness requirements. For example, a query might include an optional clause stating that a result up to 30 seconds old is acceptable. Explicitly stating freshness requirements would give the DBMS freedom to decide on caching and other strategies to improve performance while making sure that applications’ freshness requirements are met.

8. Concluding Remarks

The goal of mid-tier database caching is to improve system throughput and response time by transparently offloading some of the query processing load from a backend database server to cheaper cache servers. This paper described MTCache, our initial prototype of a mid-tier cache solution for SQL Server. The design builds on SQL Server’s support for materialized views, distributed queries, and replication. MTCache is transparent to applications, which is one of the key requirements for a mid-tier caching solution. Transparency provides flexibility because the caching strategy can be changed without having to change applications in any way.

The key contributions of this work are:

- Modeling cached data as materialized views that are updated asynchronously.
- Full transparency for queries and updates, including transparent stored procedure calls.
- Full integration into cost-based optimization by shadowing the catalog information from the backend server.
- The use of dynamic plans for parameterized queries to squeeze maximal benefit out of locally cached data without excessive reoptimization.

More specifically, MTCache models the cached data as a collection of materialized views that are maintained by transactional replication. MTCache automatically creates replication subscriptions when materialized views are created. A shadow database on the cache server duplicates catalog information, but not data, from the backend server. The availability of the shadowed catalog information makes fully cost-based query optimization possible. Achieving this goal required some enhancements to the SQL Server optimizer so that it distinguishes between cached (local) and remote data and takes into account the cost of returning results across the network.

The optimizer was also enhanced to better exploit cached data for parameterized queries by producing dy-
namic plans where which part of the plan is executed is depends on run-time parameter values.

Another contribution of this work is the series of experiments we ran using the TPC-W benchmark to measure the performance of MTCache. TPC-W emulates a storefront application modeled after a book seller. On the read-dominated Browsing and Shopping workloads we measured very significant improvements in system throughput because a substantial part of the query processing could be offloaded to MTCache servers. On the Shopping workload, it appears possible to increase throughput by a factor of 25 before saturating the backend server and on the Shopping workload by a factor of more than 10.

9. References