MTCache: Mid-Tier Database Caching for SQL Server

Per-Åke Larson   Jonathan Goldstein
Microsoft
{*palarson,jongold*}*@microsoft.com*

Hongfei Guo
University of Wisconsin
guo@cs.wisc.edu

Jingren Zhou
Columbia University
jrzhou@cs.columbia.edu

Abstract

MTCache is a prototype mid-tier database caching solution for SQL Server that transparently offloads part of the query workload from a backend server to front-end servers. The goal is to improve system throughput and scalability but without requiring application changes. This paper outlines the architecture of MTCache and highlights several of its key features: modeling of data as materialized views, integration with query optimization, and support for queries with explicit data currency and consistency requirements.

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. A single user request may generate many 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. The goal of mid-tier database caching is to transfer some of the load from the backend database server to front-end database servers. A front-end server replicates some of the data from the backend database, which allows some queries to be computed locally.

A key requirement of mid-tier database caching is application transparency, that is, applications should not be aware of what is cached and should not be responsible for routing requests to the appropriate server. If they are, the caching strategy cannot be changed without changing applications.

This paper gives a brief overview of MTCache, a mid-tier database cache solution for Microsoft SQL Server that achieves this goal. We outline the architecture of our prototype system and highlight a few specific features:

- Modeling of local data as materialized views, including a new type called partially materialized views.
- Integration into query optimization and improvements for parameterized queries
- Support for explicit currency and consistency requirements

The rest of the paper is organized as follows. Section 2 outlines the overall architecture of MTCache. Section 3 deals with query processing and optimization and describes optimizer extensions for parameterized queries. Section 4 describes the support for explicit currency and consistency requirements and section 5 introduces partially materialized views. More details can be found in references [6] and [7].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

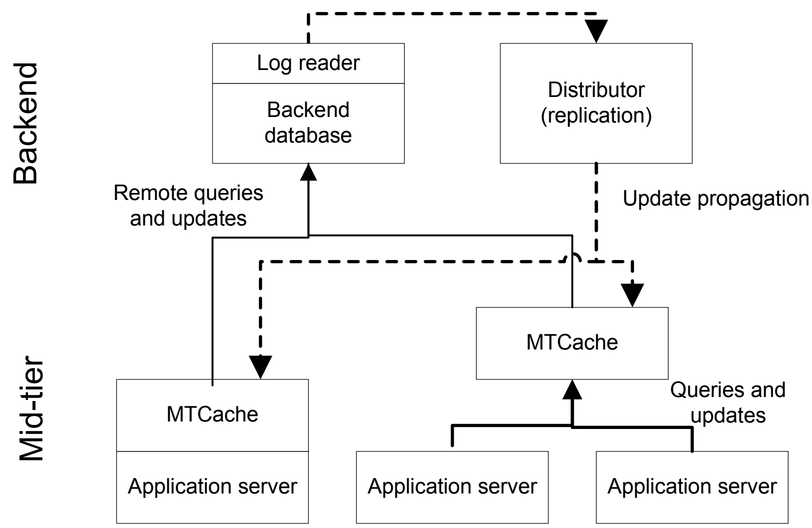

Figure 1: Configuration with two MTCaches and three application servers

2 MTCache architecture

Figure 1 illustrates a configuration with two MTCache servers. An MTCache server can run either on the same machine as an application server or on a separate machine, possibly handling queries from several application servers.

An MTCache front-end server stores a 'shadow' database containing 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 matching and check permissions. The shadowed statistics are needed during query optimization.

The actual data stored in an MTCache database is a subset of the data from the backend database. The subset consists of fully or partially materialized select-project views of tables or materialized views residing on the backend server. (Partially materialized views are explained in section 5.) What data to cache is controlled by the DBA who simply creates 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 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 cached data can be thought of as a collection of distributed materialized views that are transactionally consistent but may be slightly out of date.

An MTCache server may cache data from multiple backend servers. Each shadow database is associated with a particular 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.

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 immediately forwards them to the backend server. Changes are then propagated asynchronously to all affected MTCache servers.

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 front-end server instead of the backend server, we only need to redirect the appropriate ODBC source from the backend server to the front-end server.

3 Query optimization and processing

Queries go through normal optimization on the MTCache server. Cached views are treated as regular materialized views and, if applicable, picked up by the optimizers view matching mechanism [4]. 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 makes a cost-based decision on whether to evaluate a subexpression locally or remotely.

To add this capability to the optimizer, we introduced a new operator, DataTransfer, and a new physical property, DataLocation, for each data source and operator. DataLocation 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. Cached views and their indexes are Local and all other data sources are Remote. A new optimization rule 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 startup cost.

Cost estimation was modified 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 during execution.

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 cid $<=$ 1000 and receive the query

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

where @p1 is a run-time parameter. The view cannot always be used; only when the actual value of @p1 is less than or equal to 1000. Unfortunately, the actual parameter value is only known at run time, not at optimization time. To fully exploit views like Cust1000 also for parameterized queries, the optimizer generates plans with two branches, one local branch and one remote branch, with a SwitchUnion operator on top that selects the appropriate branch at run time. During view matching, it is noticed that Cust1000 contains all required rows if @p1 $<=$ 1000 but not otherwise. The optimizer then creates an alternative with a SwitchUnion operator on top having the switch predicate @p1 $<=$ 1000 and two children, one using the cached view Cust1000 and the other submitting a query to the backend server. During execution, the switch predicate is evaluated first and the local branch is chosen if it evaluates to true, otherwise the remote branch.

4 Currency and consistency requirements

Many applications routinely make use of cached or replicated data that may be somewhat stale, that is, the data does not necessarily reflect the current state of the database. However, current database system do not allow an application to indicate what level of staleness is acceptable. We have extended SQL so queries can include explicit data currency and consistency (C&C) requirements and MTCache guarantees that the query result satisfies the requirements. Explicit C&C requirements give the DBMS freedom to decide on caching and other strategies to improve performance while guaranteeing that applications requirements are still met.

Let us first clarify what we mean by the terms currency and consistency. Currency simply refers to how current or up-to-date a set of rows (a table, a view or a query result) is. Consider a database with two tables, Books and Reviews, as might be used by a small online book store. Suppose that we have a replicated table BooksCopy that is refreshed once every hour. The currency of BooksCopy is simply the elapsed time since the last refresh to the commit time of the latest transaction updating Books on the back-end database.

Suppose we have another replicated table, ReviewsCopy, that is also refreshed once every hour. The state of BooksCopy corresponds to some snapshot of the underlying database and similarly for ReviewsCopy. However, the two replicas do not necessarily reflect exactly the same snapshot. If they do, we say that they are (mutually) consistent.

C&C constraints are expressed through a optional currency clause that occurs last in a Select-From-Where (SFW) block and follows the same scoping rules as the WHERE clause. We will illustrate what can be expressed through a few example queries.

```
Q1: Select ...
     from Books B, Reviews R
     where B.isbn = R.isbn and B.price < 25
     currency bound 10 min on (B,R)
```

The currency clause in Q1 expresses two constraints: a) the inputs cannot be more than 10 min out of date and b) the two inputs must be consistent, that is, be from the same database snapshot. Suppose that we have cached replicas of Books and Reviews, and compute the query from the replicas. To satisfy the C&C constraint of Q1, the result obtained using the replicas must be equivalent to the result that would be obtained if the query were computed against mutually consistent snapshots of Books and Reviews that are no older than 10 min (when execution of the query begins).

```
Q1: Select ...
     from Books B, Reviews R
     where B.isbn = R.isbn and B.price < 25
     currency bound 10 min on B, 30 min on R
```

Query Q2 relaxes the bound on R to 30 min and no longer requires that the inputs be mutually consistent. For a catalog browsing application, for example, this level of currency and consistency may be perfectly acceptable.

Q1 and Q2 required that all input rows be from the same snapshot, which may be stricter than necessary. Sometimes it is acceptable if rows or groups of rows from the same table are from different snapshots, which is illustrated by Q3. The phrase 'R by R.isbn' has the following meaning: if the rows in Reviews are grouped on isbn, rows within the same group must originate from the same snapshot.

```
Q3: Select ...
     from Reviews R
     currency bound 10 min on R by R.isbn
```

MTCache enforces C&C constraints when specified. Consistency constraints are enforced at optimization time while currency constraints are enforced at execution time. MTCache keeps track of which cached views are guaranteed to be mutually consistent and how current their data is. We extended the SQL Server optimizer to select the best plan taking into account the query's C&C constraints and the status of applicable cached views. In contrast with traditional plans, the execution plan includes runtime checking of the currency of each local view used. Depending on the outcome of this check, the plan switches between using the local view or submitting a remote query. The result returned to the user is thus guaranteed to satisfy the query's consistency and currency constraints. More details can be found in reference [7].

5 Partially materialized views

In current database systems, a view must be either fully materialized or not materialized at all. Sometimes it would be preferable to materialize only some of the rows, for example, the most frequently accessed rows, and be able to easily and quickly change which rows are materialized. This is possible in MTCache by means of a new view type called a partially materialized view (PMV). Exactly which rows are currently materialized is defined by one or more control tables (tables of content) associated with the view. Changing which rows are materialized is a simple matter of updating the control table.

We will illustrate the idea using the Books and Reviews database mentioned earlier. Suppose we receive many queries looking for paperbacks by some given author and follow-up queries looking for related reviews. Some authors are much more popular than others and which authors are popular changes over time. In other words, the access pattern is highly skewed and changes over time. We would like to answer a large fraction of the queries locally but it is too expensive in storage and maintenance required to cache the complete Books and Reviews tables. Instead, we cache only the paperbacks of the most popular authors and the associated reviews. This scenario can be handled by creating one control table and two partially materialized views as shown below.

```
Create table AuthorList( authorid int)

Create view BooksPmv as
  select isbn, title, price, ...
  from Books
  where type = 'paperback' and authorid in (select authorid from AuthorList)

Create view ReviewsPmv as
  select isbn, reviewer, date, ...
  from Reviews
  where isbn in (select isbn from BooksPmv)
```

AuthorList act as a control table and contains the Ids of the authors whose paperback books are currently included in the view BooksPmv. The contents of BooksPmv is tied to the entries of AuthorList by the subquery predicate in the view definition. BooksPmv in turn acts as a control table for ReviewsPmv because it includes reviews only for books that are found in BooksPmv. In this way the contents of the two views are coordinated.

To add information about a new author to the two views, all that is needed is to add the author's id to AuthorList. Normal incremental view maintenance will then automatically add the author's paperback books to BooksPmv, which in turn causes the related reviews to be added to ReviewsPmv.

View matching has been extended to partially materialized views and also produces dynamic plans. Suppose we receive a query looking for paperbacks by an author with authorid=1234. This query can potentially be answered from BooksPmv but it is not guaranteed because its content may change at any time. To handle this situation, the optimizer produces a dynamic plan with a SwitchUnion on top that selects between using

BooksPmv and a fallback plan that retrieves the data from the backend server. Determining which branch to use consists of checking whether authorid=1234 exists in AuthorList.

This brief description covers only the simplest form of PMVs, namely, PMVs with single discrete-value control tables. Other types of control tables are possible, for example, range control tables and a PMV may have several control tables. Further details will be provided in an upcoming paper.

6 Related Work

Our approach is similar to DBCache [8, 1, 2, 3] in the sense that both systems transparently offload some queries to front-end servers, forward all updates to the backend server and rely on replication to propagate updates. DBCache was originally limited to caching complete tables [8] but more recently has added support for dynamically determined subsets of rows by means of a new table type called Cache Tables [3]. MTCache provides additional flexibility by modeling the cache contents as fully or partially materialized views, thereby allowing caching of horizontal and vertical subsets of tables and materialized views on the backend. 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 fully integrated into the optimization process and is entirely cost-based.

TimesTen also offers a mid-tier caching solution built on their in-memory database manager [9, 10]. Their product provides many features but the cache is not transparent to applications, that is, applications are responsible for query routing.

References

[1] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, L. Brown DB-Cache: Database Caching for Web Application Servers, SIGMOD 2002, 612.

[2] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB 2003, 718-729

[3] C. Bornhovd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald, DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures, SIGMOD 2003, 662.

[4] J. Goldstein, P.-Å. Larson, Optimizing Queries Using Materialized Views: A practical, scalable solution. SIGMOD 2001, 331-342.

[5] P.-Å. Larson, J. Goldstein, J. Zhou, Transparent Mid-Tier Database Caching in SQL Server, SIGMOD 2003, 661.

[6] P.-Å Larson, J. Goldstein, J. Zhou, MTCache: Transparent Mid-Tier Database Caching in SQL Server. ICDE 2004, 177-189

[7] H. Guo, P.-Å Larson, R. Ramakrishnan, J. Goldstein, Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. SIGMOD 2004 (to appear).

[8] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, J. F. Naughton, Middle-Tier Database Caching for e-Business, SIGMOD 2002, 600-611.

[9] The TimesTen Team, High Performance and Scalability through Application-Tier, In-Memory Data Management, VLDB 2000, 677-680.

[10] The TimesTen Team, Mid-Tier Caching: The TimesTen Approach, SIGMOD 2002, 588-593.