Transactional Caching of Application Data using Recent Snapshots

Dan R. K. Ports[*] Austin T. Clements[†] Irene Zhang[†] Samuel Madden Barbara Liskov
MIT CSAIL

1 Overview

Many of today's well-known websites use application data caches to reduce the bottleneck load on the database, as well as the computational load on the application servers. Distributed in-memory shared caches, exemplified by *memcached*, are one popular approach. These caches typically provide a get/put interface, akin to a distributed hash table; the application chooses what data to keep in the cache and keeps it up to date. By storing the cache entirely in memory and horizontally partitioning among nodes, in-memory caches provide quick response times and ease of scaling.

However, existing caches have no notion of transactional consistency: there is no way to ensure that two accesses to the cache reflect a view of the database at the same point in time. While the backing database goes to great lengths to ensure this property (serializable isolation), the caching layer violates these guarantees. The resulting inconsistencies can have unpleasant consequences if exposed to the user (*e.g.*, attributing the latest bid to the wrong user on an auction site), or add complexity to application code by forcing it to cope with temporarily violated invariants.

We argue that transactional semantics are not incompatible with cache performance and scalability. We introduce a transactional cache, *TxCache*, which guarantees that all values retrieved from the cache or database during a transaction reflect a consistent snapshot of the database.

TxCache also strives to simplify application design by helping manage the cache. Instead of requiring applications to manually insert and check for values in the cache, TxCache provides a library with which programmers simply designate functions as cacheable, and the library checks the cache for previous calls with the same arguments. In particular, and unlike memcached, TxCache does not require applications to explicitly invalidate cached values; correctly identifying the values to invalidate is difficult because it requires global reasoning about the application.

2 Running Transactions in the Past

TxCache provides a different consistency guarantee than most caches. Typical caches strive to guarantee *freshness*: the cache always reflects the latest state of what it is caching. TxCache relaxes its freshness guarantee slightly to provide *transactional consistency*: within a transaction block, the application sees a view consistent with the state of the database at a particular timestamp.

Providing both freshness and consistency requires conflicting transactions to block or abort, which is impractical in a high-throughput system. Snapshot isolation ensures that a transaction sees a consistent snapshot of the database taken when the transaction begins. TxCache takes this approach further, allowing read-only transactions to be run on a slightly earlier snapshot (up to a user-specified limit). In addition to avoiding conflicts, this increases the hit rate by allowing the use of slightly stale data.

Using stale but consistent cached data is safe because of the inherent asynchrony in distributed systems. Even without a cache, query results are already not guaranteed to be fresh unless leases or locks are used, because concurrent updates might make them stale while in flight. Applications can use read/write transactions when freshness is required. TxCache does not use cached data for read/write transactions, so it introduces no new anomalies.

[*]student; will present [†] student

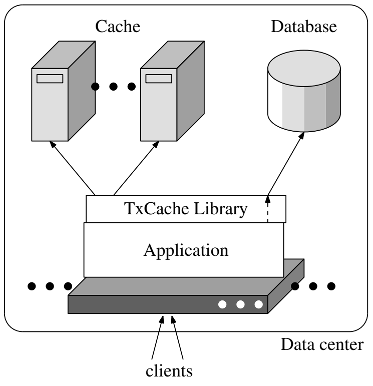

Figure 1: Anatomy of a TxCache deployment.

3 TxCache Design

TxCache is designed for systems where several application servers (*e.g.*, web servers) interact with a database server. TxCache introduces two new components, shown in Figure 1: a cache and an application-side cache interface library. It also requires minor modifications to the database server.

Cache servers. TxCache stores data in RAM using a simple key-value distributed hash table to partition data across many cache server nodes. Applications do not access the DHT directly: TxCache's application-side library assigns keys to, retrieves, and updates cached values as cacheable functions are called.

Cached data is *versioned*. In addition to its key, each entry in the cache is tagged with its *validity interval*, the range of time at which the cached value was current. It is bounded below by the commit time of the transaction that made it valid, and above by that of the first subsequent transaction to change the result.

Database server. TxCache uses a standard relational database with a few modifications. Specifically, it requires the ability to run queries on slightly stale snapshots, and to generate validity intervals for each query result returned. These validity intervals are used to tag the cache entries. We leverage existing concurrency control mechanisms to add the necessary support to an existing DBMS, PostgreSQL, with minimal modifications (under 1,000 lines of code changed) and no observable performance impact.

Library. Applications interact with TxCache through its application-side library. The library assigns a timestamp to the transaction and retrieves from the cache only data valid at that timestamp; the timestamp is assigned lazily based on what data is available in the cache. The library provides language bindings that wrap a cacheable function; when called, it checks for available cached values. If none are available, it invokes the function's implementation, collects the validity intervals from any database queries it makes, and stores the result in the cache.

4 Results

Preliminary results using the RUBiS auction website benchmark are encouraging. Even restricting stale data to at most 30 seconds old, TxCache is able to increase peak throughput by a factor of over $2.5\times$ without sacrificing transactional consistency.