Transactional Consistency and Automatic Management in an Application Data Cache

Dan R. K. Ports Austin T. Clements Irene Zhang Samuel Madden Barbara Liskov

MIT CSAIL

txcache@csail.mit.edu

Abstract

Distributed in-memory application data caches like *memcached* are a popular solution for scaling database-driven web sites. These systems are easy to add to existing deployments, and increase performance significantly by reducing load on both the database and application servers. Unfortunately, such caches do not integrate well with the database or the application. They cannot maintain transactional consistency across the entire system, violating the isolation properties of the underlying database. They leave the application responsible for locating data in the cache and keeping it up to date, a frequent source of application complexity and programming errors.

Addressing both of these problems, we introduce a transactional cache, *TxCache*, with a simple programming model. TxCache ensures that any data seen within a transaction, whether it comes from the cache or the database, reflects a slightly stale but consistent snapshot of the database. TxCache makes it easy to add caching to an application by simply designating functions as cacheable; it automatically caches their results, and invalidates the cached data as the underlying database changes. Our experiments found that adding TxCache increased the throughput of a web application by up to $5.2\times$, only slightly less than a non-transactional cache, showing that consistency does not have to come at the price of performance.

1 Overview

Today's web applications are used by millions of users and demand implementations that scale accordingly. A typical system includes application logic (often implemented in web servers) and an underlying database that stores persistent state, either of which can become a bottleneck [1]. Increasing database capacity is typically a difficult and costly proposition, requiring careful partitioning or the use of distributed databases. Application server bottlenecks can be easier to address by adding more nodes, but this also quickly becomes expensive.

Application-level data caches, such as memcached [24], Velocity/AppFabric [34] and NCache [25], are a popular solution to server and database bottlenecks. They are deployed extensively by well-known web applications like LiveJournal, Facebook, and MediaWiki. These caches store arbitrary application-generated data in a lightweight, distributed in-memory cache. This flexibility allows an application-level cache to act as a database query cache, or to act as a web cache and cache entire web pages. But increasingly complex application logic and more personalized web content has made it more useful to cache the result of *application computations* that depend on database queries. Such caching is useful because it averts costly post-processing of database records, such as converting them to an internal representation, or generating partial HTML output. It also allows common content to be cached separately from customized content, so that it can be shared between users. For example, MediaWiki uses memcached to store items ranging from translations of interface messages to parse trees of wiki pages to the generated HTML for the site's sidebar.

Existing caches like memcached present two challenges for developers, which we address in this paper. First, they do not ensure transactional consistency with the rest of the system state. That is, there is no way to ensure that accesses to the cache and the database return values that reflect a view of the entire system at a single point in time. While the backing database goes to great length to ensure that all queries performed in a transaction reflect a consistent view of the database, *i.e.* it can ensure serializable isolation, it is nearly impossible to maintain these consistency guarantees while using a cache that operates on application objects and has no notion of database transactions. The resulting anomalies can cause incorrect information to be exposed to the user, or require more complex application logic because the application must be able to cope with violated invariants.

Second, they offer only a GET/PUT interface, placing full responsibility for explicitly managing the cache with the application. Applications must assign names to cached values, perform lookups, and keep the cache up to date. This has been a common source of programming errors in applications that use memcached. In particular, applications must explicitly *invalidate* cached data when the database changes. This is often difficult; identifying every cached application computation whose value may

have been changed requires global reasoning about the application.

We address both problems in our transactional cache, *TxCache*. TxCache provides the following features:

- transactional consistency: all data seen by the application reflects a consistent snapshot of the database, whether the data comes from cached application-level objects or directly from database queries.
- access to slightly stale but nevertheless consistent snapshots for applications that can tolerate stale data, improving cache utilization.
- a simple programming model, where applications simply designate functions as cacheable. The TxCache library then handles inserting the result of the function into the cache, retrieving that result the next time the function is called with the same arguments, and invalidating cached results when they change.

To achieve these goals, TxCache introduces the following noteworthy mechanisms:

- a protocol for ensuring that transactions see only consistent cached data, using minor database modifications to compute the validity times of database queries, and attaching them to cache objects.
- a lazy timestamp selection algorithm that assigns a transaction to a timestamp in the recent past based on the availability of cached data.
- an automatic invalidation system that tracks each object's database dependencies using dual-granularity invalidation tags, and produces notifications if they change.

We ported the RUBiS auction website prototype and MediaWiki, a popular web application, to use TxCache, and evaluated it using the RUBiS benchmark [2]. Our cache improved peak throughput by $1.5 - 5.2\times$ depending on the cache size and staleness limit, an improvement oonly slightly below that of a non-transactional cache.

The next section presents the programming model and consistency semantics. Section 3 sketches the structure of the system, and Sections 4–6 describe each component in detail. Section 7 describes our experiences porting applications to TxCache, Section 8 presents a performance evaluation, and Section 9 reviews the related work.

2 System and Programming Model

TxCache is designed for systems consisting of one or more application servers that interact with a database server. These application servers could be web servers running embedded scripts (*e.g.* with `mod_php`), or dedicated application servers, as with Sun's Enterprise Java Beans. The database server is a standard relational database; for simplicity, we assume the application uses a single database to store all of its persistent state.

TxCache introduces two new components, as shown in

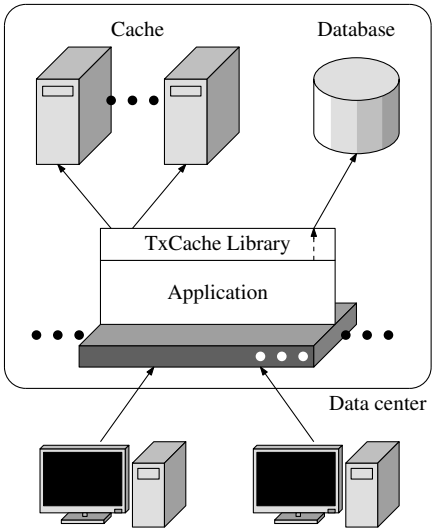


Figure 1: Key components in a TxCache deployment. The system consists of a single database, a set of cache nodes, and a set of application servers. TxCache also introduces an application library, which handles all interactions with the cache server.

Figure 1: a cache and an application-side cache library, as well as some minor modifications to the database server. The cache is partitioned across a set of cache nodes, which may run on dedicated hardware or share it with other servers. The application never interacts with the cache servers; the TxCache library transparently translates an application's cacheable functions into cache accesses.

2.1 Programming Model

Our goal is to make it easy to incorporate caching into a new or existing application. Towards this end, TxCache provides an application library with a simple programming model, shown in Figure 2, based on *cacheable functions*. Applications developers can cache computations simply by designating functions to be cached.

Programs group their operations into transactions. TxCache requires applications to specify whether their transactions are read-only or read/write by using either the BEGIN-RO or BEGIN-RW function. Transactions are ended by calling COMMIT or ABORT. Within a transaction block, TxCache ensures that, regardless of whether the application gets its data from the database or the cache, it sees a view consistent with the state of the database at a single point in time.

Within a transaction, operations can be grouped into *cacheable functions*. These are actual functions in the program's code, annotated to indicate that their results can be cached. A cacheable function can consist of database queries and computation, and can also make calls to other cacheable functions. To be suitable for caching, functions

- BEGIN-RO(*staleness*) : Begin a read-only transaction. The transaction sees a consistent snapshot from within the past *staleness* seconds.
- BEGIN-RW() : Begin a read/write transaction.
- COMMIT() → *timestamp* : Commit a transaction and return the timestamp at which it ran
- ABORT() : Abort a transaction

- MAKE-CACHEABLE(*fn*) → *cached-fn* : Makes a function cacheable. *cached-fn* is a new function that first checks the cache for the result of another call with the same arguments. If not found, it executes *fn* and stores its result in the cache.

Figure 2: TxCache library API

must be pure, *i.e.* they must be deterministic, not have side effects, and depend only on their arguments and the database state. For example, it would not make sense to cache a function that returns the current time. TxCache currently relies upon programmers to ensure that they only cache suitable functions, but this requirement could also be enforced using static or dynamic analysis [14, 33].

Cacheable functions are essentially memoized. TxCache's library provides a MAKE-CACHEABLE function that takes an implementation of a cacheable function and returns a wrapper function that can be called to take advantage of the cache. When called, the wrapper function checks if the cache contains the result of a previous call to the function with the same arguments that is consistent with the current transaction's snapshot. If so, it returns it. Otherwise, it invokes the implementation function and stores the returned value in the cache. With proper linguistic support (*e.g.* Python decorators), marking a function cacheable can be as simple as adding a tag to its existing definition.

Our cacheable function interface is easier to use than the GET/PUT interface provided by existing caches like memcached. It does not require programmers to manually assign keys to cached values and keep them up to date. Although seemingly straightforward, this is nevertheless a source of errors because selecting keys requires reasoning about the entire application and how the application might evolve. Examining MediaWiki bug reports, we found that several memcached-related MediaWiki bugs stemmed from choosing insufficiently descriptive keys, causing two different objects to overwrite each other [22]. In one case, a user's watchlist page was always cached under the same key, causing the same results to be returned even if the user requested to display a different number of days worth of changes.

TxCache's programming model has another crucial benefit: it does not require applications to explicitly update or invalidate cached results when modifying the database. Adding explicit invalidations requires global reasoning about the application, hindering modularity: adding caching for an object requires knowing every place it could possibly change. This, too, has been a source of bugs in MediaWiki [23]. For example, editing a wiki page clearly requires invalidating any cached copies of that page. But other, less obvious objects must be invalidated too. Once MediaWiki began storing each user's edit count in their cached USER object, it became necessary to invalidate this object after an edit. This was initially forgotten, indicating that identifying all cached objects needing invalidation is not straightforward, especially in applications so complex that no single developer is aware of the whole of the application.

2.2 Consistency Model

TxCache provides *transactional consistency*: all requests within a transaction see a consistent view of the system as of a specific timestamp. That is, requests see only the effects of other transactions that committed prior to that timestamp. For read/write transactions, TxCache supports this guarantee by running them directly on the database, bypassing the cache entirely. Read-only transactions use objects in the cache, and TxCache ensures that nevertheless they view a consistent state.

Most caches return slightly stale data simply because modified data does not reach the cache immediately. TxCache goes further by allowing applications to specify an explicit *staleness limit* to BEGIN-RO, indicating that that the transaction can see a view of data from that time or later. However, regardless of the age of the snapshot, each transaction always sees a consistent view. This feature is motivated by the observation that many applications can tolerate a certain amount of staleness [18], and using stale cached data can improve the cache's hit rate [21].

Applications can specify their staleness limit on a per-transaction basis. Additionally, when a transaction commits, TxCache provides the user with the timestamp at which it ran. Together, these can be used to avoid anomalies. For example, an application can store the timestamp of a user's last transaction in its session state, and use that as a staleness bound so that the user never observes time moving backwards. More generally, these timestamps can be used to ensure a causal ordering between related transactions [20].

We chose to have read/write transactions bypass the cache entirely so that TxCache does not introduce new anomalies. The application can expect the same guarantees (and anomalies) of the underlying database. For example, if the underlying database uses snapshot isolation, the system will still have the same anomalies as snapshot isolation, but TxCache will never introduce snapshot isolation anomalies into the read/write transactions of a system that does not use snapshot isolation. Our model

could be extended to allow read/write transactions to read information from the cache, if applications are willing to accept the risk of anomalies. One particular challenge is that read/write transactions typically expect to see the effects of their own updates, while these cannot be made visible to other transactions until the commit point.

3 System Architecture

In order to present an easy-to-use interface to application developers, TxCache needs to store cached data, keep it up to date, and ensure that data seen by an application is transactionally consistent. This section and the following ones describe how it achieves this using cache servers, modifications to the database, and an application-side library. None of this complexity, however, is visible to the application, which sees only cachable functions.

An application running with TxCache accesses information from the cache whenever possible, and from the database on a cache miss. To ensure it sees a consistent view, TxCache uses versioning. Each database query has an associated *validity interval*, describing the range of time over which its result was valid, which is computed automatically by the database. The TxCache library tracks the queries that a cached value depends on, and uses them to tag the cache entry with a validity interval. Then, the library provides consistency by ensuring that, within each read-only transaction, it only retrieves values from the cache and database that were valid at the same time. Thus, each transaction effectively sees a snapshot of the database taken at a particular time, even as it accesses data from the cache.

Section 4 describes how the cache is structured, and defines how a cached object's validity interval and database dependencies are represented. Section 5 describes how the database is modified to track query validity intervals and provide invalidation notifications when a query's result changes. Section 6 describes how the library tracks dependencies for application objects, and selects consistent values from the cache and database.

4 Cache Design

TxCache stores cached data in RAM on a number of cache servers. The cache presents a hash table interface: it maps keys to associated values. Applications do not interact with the cache directly; the TxCache library translates the name and arguments of a function call into a hash key, and checks and updates the cache itself.

Data is partitioned among cache nodes using a consistent hashing approach [17], as in peer-to-peer distributed hash tables [31, 35]. Unlike DHTs, we assume that the system is small enough that every application node can maintain a complete list of cache servers, allowing it to immediately map a key to the responsible node. This list could be maintained by hand in small systems, or using a group membership service [10] in larger or more dynamic environments.

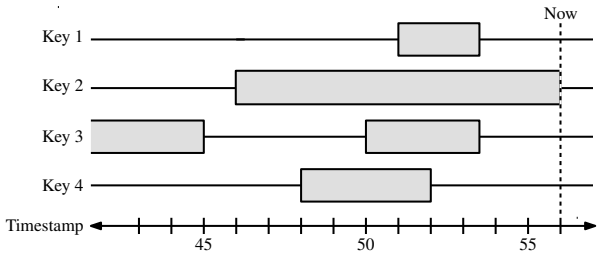


Figure 3: An example of versioned data in the cache at one point in time. Each rectangle is a version of a data item. For example, the data for key 1 became valid with commit 51 and invalid with commit 53, and the data for key 2 became valid with commit 46 and is still valid.

4.1 Versioning

Unlike a simple hash table, our cache is *versioned*. In addition to its key, each entry in the cache is tagged with its *validity interval*, as shown in Figure 3. This interval is the range of time at which the cached value was current. Its lower bound is the commit time of the transaction that caused it to become valid, and its upper bound is the commit time of the first subsequent transaction to change the result, making the cache entry invalid. The cache can store multiple cache entries with the same key; they will have disjoint validity intervals because only one is valid at any time. Whenever the TxCache library puts the result of a cacheable function call into the cache, it includes the validity interval of that result (derived using information obtained from the database).

To look up a result in the cache, the TxCache library sends both the key it is interested in and a timestamp or range of acceptable timestamps. The cache server returns a value consistent with the library's request, *i.e.* one whose validity interval intersects the given range of acceptable timestamps, if any exists. The server also returns the value's associated validity interval. If multiple such values exist, the cache server returns the most recent one.

When a cache node runs out of memory, it evicts old cached values to free up space for new ones. Cache entries are never pinned and can always be discarded; if one is later needed, it is simply a cache miss. A cache eviction policy can take into account both the time since an entry was accessed, and its staleness. Our cache server uses a least-recently-used replacement policy, but also eagerly removes any data too stale to be useful.

4.2 Invalidation Tags and Streams

When an object is inserted into the cache, it can be flagged as *still-valid* if it reflects the latest state of the database, like Key 2 in Figure 3. For such objects, the database

provides *invalidation* notifications when they change.

Every still-valid object has an associated set of *invalidation tags* that describe which parts of the database it depends on. Each invalidation tag has two parts: a table name and an optional index key description. The database identifies the invalidation tags for a query based on the access methods used to access the database. A query that uses an index equality lookup receives a two-part tag, *e.g.* a search for users with name Alice would receive tag USERS:NAME=ALICE. A query that performs a sequential scan or index range scan has a wildcard for the second part of the tag, *e.g.* USERS:⋆. Wildcard invalidations are expected to be very rare because applications typically try to perform only index lookups; they exist primarily for completeness. Queries that access multiple tables or multiple keys in a table receive multiple tags. The object's final tag set will have one or more tags for each query that the object depends on.

The database distributes invalidations to the cache as an *invalidation stream*. This is an ordered sequence of messages, one for each update transaction, containing the transaction's timestamp and all invalidation tags that it affected. Each message is delivered to all cache nodes by a reliable application-level multicast mechanism [10], or by link-level broadcast if possible. The cache servers process the messages in order, truncating the validity interval for any affected object at the transaction's timestamp.

Using the same transaction timestamps to order cache entries and invalidations eliminates race conditions that could occur if an invalidation reaches the cache server before an item is inserted with the old value. These race conditions are a real concern: MediaWiki does not cache failed article lookups, because a negative result might never be removed from the cache if the report of failure is stale but arrived after its corresponding invalidation.

For cache lookup purposes, items that are still valid are treated as though they have an upper validity bound equal to the timestamp of the last invalidation received prior to the lookup. This ensures that there is no race condition between an item being changed on the database and invalidated in the cache, and that multiple items modified by the same transaction are invalidated atomically.

5 Database Support

The validity intervals that TxCache uses in its cache are derived from validity information generated by the database. To make this possible, TxCache uses a modified DBMS that has similar versioning properties to the cache. Specifically, it can run queries on slightly stale snapshots, and it computes validity intervals for each query result it returns. It also assigns invalidation tags to queries, and produces the invalidation stream described in Section 4.2.

Though standard databases do not provide these features, we show they can be implemented by reusing the same mechanisms that are used to implement multiversion concurrency control techniques like snapshot isolation. In this section, we describe how we modified an existing DBMS, PostgreSQL [29], to provide the necessary support. The modifications are not extensive (under 2000 lines of code in our implementation). Moreover, they are not Postgres-specific; the approach can be applied to other databases that use multiversion concurrency.

5.1 Exposing Multiversion Concurrency

Because our cache allows read-only transactions to run slightly in the past, the database must be able to perform queries against a past snapshot of a database. This situation arises when a read-only transaction is assigned a timestamp in the past and reads some cached data, and then a later operation in the same transaction results in a cache miss, requiring the application to query the database. The database query must return results consistent with the cached values already seen, so the query must execute at the same timestamp in the past.

Temporal databases, which track the history of their data and allow "time travel," solve this problem but impose substantial storage and indexing cost to support complex queries over the entire history of the database. What we require is much simpler: we only need to run a transaction on a stale but recent snapshot. Our insight is that these requirements are essentially identical to those for supporting snapshot isolation [5], so many databases already have the infrastructure to support them.

We modified Postgres to expose the multiversion storage it uses internally to provide snapshot isolation. We added a PIN command that assigns an ID to a read-only transaction's snapshot. When starting a new transaction, the TxCache library can specify this ID using the new BEGIN SNAPSHOTID syntax, creating a new transaction that sees the same view of the database as the erstwhile read-only transaction. The database state for that snapshot will be retained at least until it is released by the UNPIN command. A pinned snapshot is identified by the commit time of the last committed transaction visible to it, allowing it to be easily ordered with respect to update transactions and other snapshots.

Postgres is especially well-suited to this modification because of its "no-overwrite" storage manager [36], which already retains recent versions of data. Because stale data is only removed periodically by an asynchronous "vacuum cleaner" process, the fact that we keep data around slightly longer has little impact on performance. However, our technique is not Postgres-specific; any database that implements snapshot isolation must have a way to keep a similar history of recent database states, such as Oracle's rollback segments.

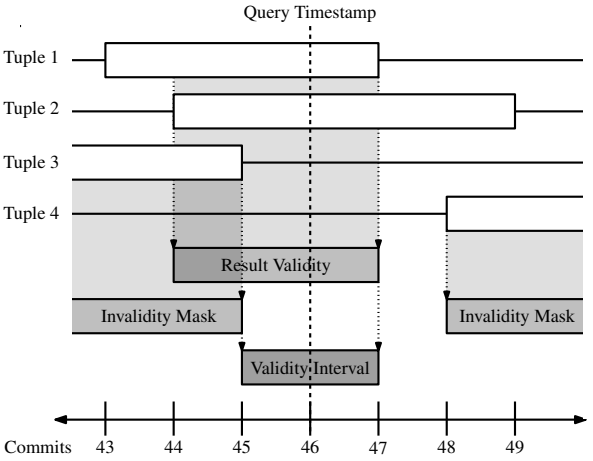


Figure 4: Example of tracking the validity interval for a read-only query. All four tuples match the query predicate. Tuples 1 and 2 match the timestamp, so their intervals intersect to form the result validity. Tuples 3 and 4 fail the visibility test, so their intervals join to form the invalidity mask. The final validity interval is the difference between the result validity and the invalidity mask.

5.2 Tracking Result Validity

TxCache needs the database server to provide the validity interval for every query result in order to ensure transactional consistency of cached objects. Recall that this is defined as the range of timestamps for which the query would give the same results. Its lower bound is the commit time of the most recent transaction that added, deleted, or modified any tuple in the result set. It may have an upper bound if a subsequent transaction changed the result, or it may be unbounded if the result is still current.

The validity interval is computed as the intersection of two ranges, the *result tuple validity* and the *invalidity mask*, which we track separately.

The result tuple validity is the intersection of the validity times of the tuples returned by the query. For example, tuple 1 in Figure 4 was deleted at time 47, and tuple 2 was created at time 44; the result would be different before time 44 or after time 47. This interval is easy to compute because multiversion concurrency requires that each tuple in the database be tagged with the ID of its creating transaction and deleting transaction (if any). We simply propagate these tags throughout query execution. If an operator, such as a join, combines multiple tuples to produce a single result, the validity interval of the output tuple is the intersection of its inputs.

The result tuple validity, however, does not completely capture the validity of a query, because of *phantoms*. These are tuples that did *not* appear in the result, but would have if the query were run at a different timestamp. For example, tuple 3 in Figure 4 will not appear in the results because it was deleted before the query timestamp, but the results would be different if the query were run before it was deleted. Similarly, tuple 4 is not visible because it was created afterwards. We capture this effect with the invalidity mask, which is the union of the validity times for all tuples that failed the *visibility check*, *i.e.* were discarded because their timestamps made them invisible to the transaction's snapshot. Throughout query execution, whenever such a tuple is encountered, its validity interval is added to the invalidity mask.

The invalidity mask is conservative because visibility checks are performed as early as possible in the query plan to avoid processing unnecessary tuples. Some of these tuples might have been discarded anyway if they failed the query conditions later in the query plan (perhaps after joining with another table). While being conservative preserves the correctness of the cached results, it might unnecessarily constrain the validity intervals of cached items, reducing the hit rate. To ameloriate this problem, we continue to perform the visibility check as early as possible, but during sequential scans and index lookups, we evaluate the predicate before the visibility check. This differs from regular Postgres with respect to sequential scans, where it evaluates the cheaper visibility check first. Delaying the visibility checks improves the quality of the invalidity mask, and incurs little overhead for simple predicates, which are most common.

Finally, the invalidity mask is subtracted from the result tuple validity to give the query's final validity interval. This interval is reported to the TxCache library, piggybacked on each SELECT query result; the library combines these intervals to obtain validity intervals for objects it stores in the cache.

5.3 Automating Invalidations

When the database executes a query and reports that its validity interval is unbounded, *i.e.* the query result is still valid, it assumes responsibility for providing an invalidation when the result may have changed. At query time, it must assign invalidation tags to indicate the query's dependencies, and at update time, it must notify the cache of invalidation tags for objects that might have changed.

When a query is performed, the database examines the query plan it generates. At the lowest level of the tree are the access methods that obtain the data, *e.g.* a sequential scan of a heap file, or a B-tree index lookup. For index equality lookups, the database assigns an invalidation tag of the form TABLE:KEY. For other types, it assigns a wildcard tag TABLE:$\star$. Each query may have multiple tags; the complete set is returned along with the SELECT query results.

When a read/write transaction modifies some tuples, the database identifies the set of invalidation tags affected.

Each tuple added, deleted, or modified yields one invalidation tag for each index it is listed in. If a transaction modifies most of a table, the database can aggregate multiple tags into a single wildcard tag on TABLE:⋆. Generated invalidation tags are queued until the transaction commits. When it does, the database server passes the set of tags, along with the transaction's timestamp, to the multicast service for distribution to the cache nodes, ensuring that the invalidation stream is properly ordered.

5.4 Pincushion

TxCache needs to keep track of which snapshots are pinned on the database, and which of those are within a read-only transaction's staleness limit. It also must eventually unpin old snapshots, provided that they are not used by running transactions. The DBMS itself could be responsible for tracking this information. However, to simplify implementation, and to reduce the overall load on the database, we placed this functionality instead in a lightweight daemon known as the *pincushion* (so named because it holds the pinned snapshot IDs). It can be run on the database host, on a cache server, or elsewhere.

The pincushion maintains a table of currently pinned snapshots, containing the snapshot's ID, the corresponding wall-clock timestamp, and the number of running transactions that might be using it. When the TxCache library running on an application node begins a read-only transaction, it requests from the pincushion all sufficiently fresh pinned snapshots, *e.g.* those pinned in the last 30 seconds. The pincushion flags these snapshots as possibly in use, for the duration of the transaction. If there are no sufficiently fresh pinned snapshots, the TxCache library starts a read-only transaction on the database, running on the latest snapshot, and pins that snapshot. It then registers the snapshot's ID and the wall-clock time (as reported by the database) with the pincushion. The pincushion also periodically scans its list of pinned snapshots, removing any unused snapshots older than a threshold by sending an UNPIN command to the database.

Though the pincushion is accessed on every transaction, it performs little computation and is unlikely to form a bottleneck. In all of our experiments, nearly all pincushion requests received a response in under 0.2 ms, approximately the network round-trip time. We have also developed a protocol for replicating the pincushion to increase its throughput, but it has yet to become necessary.

6 Cache Library

Applications interact with TxCache through its application-side library, which keeps them blissfully unaware of the details of cache servers, validity intervals, invalidation tags and the like. It is responsible for assigning timestamps to read-only transactions, retrieving values from the cache when cacheable functions are called, storing results in the cache, and computing the validity intervals and invalidation tags for anything it stores in the cache.

In this section, we describe the implementation of the TxCache library. For clarity, we begin with a simplified version where timestamps are chosen when a transaction begins and cacheable functions do not call other cacheable functions. In Section 6.2, we describe a technique for choosing timestamps lazily to take better advantage of cached data. In Section 6.3, we lift the restriction on nested calls.

6.1 Basic Functionality

The TxCache library is divided into a language-independent library that implements the core functionality, and a set of bindings that implement language-specific interfaces. Currently, we have only implemented bindings for PHP, but adding support for other languages should be relatively straightforward.

Recall from Figure 2 that the library's interface is simple: it provides the standard transaction commands (BEGIN, COMMIT, and ABORT), and functions are designated as cacheable using a MAKE-CACHEABLE function that takes a function and returns a wrapped function that first checks for available cached values[1].

When a transaction is started, the application specifies whether it is read/write or read-only, and, if read-only, the staleness limit. For a read/write transaction, the TxCache library simply starts a transaction on the database server, and passes all queries directly to it. At the beginning of a read-only transaction, the library contacts the pincushion to request the list of pinned snapshots within the staleness limit, then chooses one to run the transaction at. If no sufficiently recent snapshots exist, the library starts a new transaction on the database and pins its snapshot.

The library can delay beginning an underlying read-only transaction on the database (*i.e.* sending a BEGIN SQL statement) until it actually needs to issue a query. Thus, transactions whose requests are all satisfied from the cache do not need to connect to the database at all.

When a cacheable function's wrapper is called, the library checks whether its result is in the cache. To do so, it serializes the function's name and arguments into a key (a hash of the function's code could also be used to handle software updates). The library finds the responsible cache server using consistent hashing, and sends it a LOOKUP request. The request includes the transaction's timestamp, which any returned value must satisfy. If the cache returns a matching result, the library returns it directly to the program.

In the event of a cache miss, the library calls the cacheable function's implementation. As the cacheable

[1] In languages such as PHP that lack higher-order functions, the syntax is slightly more complicated, but the concept is the same.

function issues queries to the database, the library accumulates the validity intervals and invalidation tags returned by these queries. The final result of the cacheable function is valid at all times in the intersection of the accumulated validity intervals. When the cacheable function returns, the library serializes its result and inserts it into the cache, tagged with the accumulated validity interval and any invalidation tags.

6.2 Choosing Timestamps Lazily

Above, we assumed that the library chooses a read-only transaction's timestamp when the transaction starts. Although straightforward, this approach requires the library to decide on a timestamp without any knowledge of what data is in the cache or what data will be accessed. Lacking this knowledge, it is not clear what policy would provide the best hit rate.

However, the timestamp need not be chosen immediately. Instead, it can be chosen lazily based on which cached results are available. This takes advantage of the fact that each cached value is valid over a range of timestamps: its validity interval. For example, consider a transaction that has observed a single cached result x. This transaction can still be serialized at *any* timestamp in x's validity interval. On the transaction's next call to a cacheable function, any cached value whose validity interval overlaps x's can be chosen, as this still ensures there is at least one timestamp at which the transaction can be serialized. As the transaction proceeds, the set of possible serialization points narrows each time the transaction reads a cached value or a database query result.

Specifically, the algorithm proceeds as follows. When a transaction begins, the library requests from the pincushion all pinned snapshot IDs that satisfy its freshness requirement. It stores this set as its *pin set*. The pin set represents the set of timestamps at which the current transaction can be serialized; it will be updated as the cache and the database are accessed. The pin set also initially contains a special ID, denoted $\star$, which indicates that the transaction can also be run in the present, on some newly pinned snapshot. The pin set only contains $\star$ until the first cacheable function in the transaction executes.

When the application invokes a cacheable function, the library sends a LOOKUP request for the appropriate key, but instead of indicating a single timestamp, it indicates the *bounds* of the pin set (the lowest and highest timestamp, excluding $\star$). The transaction can use any cached value whose validity interval overlaps these bounds and still remain serializable at one or more timestamps. The library then reduces the transaction's pin set by eliminating all timestamps that do not lie in the returned value's validity interval, since observing a cached value means the transaction can no longer be serialized outside its validity interval. This includes removing $\star$ from the pin set because once the transaction has used cached data, it cannot be run on a new, possibly inconsistent snapshot.

When the cache does not contain any entries that match both the key and the requested interval, a cache miss occurs. In this case, the library calls the cacheable function's implementation, as before. When the transaction makes its first database query, the library is finally forced to select a specific timestamp from the pin set and BEGIN a read-only transaction on the database at the chosen timestamp. If a non-$\star$ timestamp is chosen, the transaction runs on that timestamp's saved snapshot. If $\star$ is chosen, the library starts a new transaction, pinning the latest snapshot and reporting the pin to the pincushion. The pin set is then *reified*: $\star$ is replaced with the newly-created snapshot's timestamp, replacing the abstract concept of "the present time" with a concrete timestamp.

The library needs a policy to choose which pinned snapshot from the pin set it should run at. Simply choosing $\star$ if available, or the most recent timestamp otherwise, biases transactions towards running on recent data, but results in a very large number of pinned snapshots, which can ultimately slow the system down. To avoid the overhead of creating many snapshots, we used the following policy: if the most recent timestamp in the pin set is older than five seconds and $\star$ is available, then the library chooses $\star$ in order to produce a new pinned snapshot; otherwise it chooses the most recent timestamp.

During the execution of a cacheable function, the validity intervals of the queries that the function makes are accumulated, and their intersection defines the validity interval of the cacheable result, just as before. In addition, just like when a transaction observes values from the cache, each time it observes query results from the database, the transaction's pin set is reduced by eliminating all timestamps outside the result's validity interval, as the transaction can no longer be serialized at these points. If the transaction's pin set still contains $\star$, $\star$ is removed.

The validity interval of the cacheable function and pin set of the transaction are two distinct but related notions: the function's validity interval is the set of timestamps at which its result is valid, and the pin set is the set of timestamps at which the enclosing transaction can be serialized. The pin set always lies within the validity interval, but the two may differ when a transaction calls multiple cacheable functions in sequence, or performs "bare" database queries outside a cacheable function.

6.2.1 Correctness

Lazy selection of timestamps is a complex algorithm, and its correctness is not self-evident. The following two properties show that it provides transactional consistency.

Invariant 1. *All data seen by the application during a read-only transaction is consistent with the database*

state at every timestamp in the pin set, i.e. *the transaction can be serialized at any timestamp in the pin set.*

Invariant 1 holds because any timestamps *in*consistent with data the application has seen are removed from the pin set. The application sees two types of data: cached values and database query results. Each is tagged with its validity interval. The library removes from the pin set all timestamps that lie outside either of these intervals.

Invariant 2. *The pin set is never empty,* i.e. *the transaction can always be serialized at some timestamp.*

The pin set is initially non-empty: it contains the timestamps of all sufficiently-fresh pinned snapshots, if any, and always $\star$. So we must ensure that at least one timestamp remains every time the pin set shrinks, *i.e.* when a result is obtained from the cache or database.

When a value is fetched from the cache, its validity interval is guaranteed to intersect the transaction's pin set at at least one timestamp. The cache will only return an entry with a non-empty intersection between its validity interval and the bounds of the transaction's pin set. This intersection contains the timestamp of at least one pinned snapshot: if the result's validity interval lies partially within and partially outside the bounds of the client's pin set, then either the earliest or latest timestamp in the pin set lies in the intersection. If the result's validity interval lies entirely within the bounds of the transaction's pin set, then the pin set contains at least the timestamp of the pinned snapshot from which the cached result was originally generated. Thus, Invariant 2 continues to hold even after removing from the pin set any timestamps that do not lie within the cached result's validity interval.

It is easier to see that when the database returns a query result, the validity interval intersects the pin set at at least one timestamp. The validity interval of the query result must contain the timestamp of the pinned snapshot at which it was executed, by definition. That pinned snapshot was chosen by the TxCache library from the transaction's pin set (or it chose $\star$, obtained a new snapshot, and added it to the pin set). Thus, at least that one timestamp will remain in the pin set after intersecting it with the query's validity interval.

6.3 Handling Nested Calls

In the preceding sections, we assumed that cacheable functions never call other cacheable functions. However, it is useful to be able to nest calls to cacheable functions. For example, a user's home page at an auction site might contain a list of items the user recently bid on. We might want to cache the description and price for each item as a function of the item ID (because they might appear on other user's pages) in addition to the complete content of the user's page (because he might access it again).

Our implementation supports nested calls; this does not require any fundamental changes to the approach above. However, we must keep track of a separate cumulative validity interval and invalidation tag set for each cacheable function in the call stack. When a cached value or database query result is accessed, its validity interval is intersected with that of *each* function currently on the call stack. As a result, a nested call to a cacheable function may have a wider validity interval than its enclosing function, but not vice versa. This makes sense, as the outer function might have seen more data than the functions it calls (*e.g.* if it calls more than one cacheable function). Similarly, any invalidation tags from the database are attached to each function on the call stack, as each now has a dependency on the data.

7 Experiences

We implemented all the components of TxCache, including the cache server, database modifications to PostgreSQL to support validity tracking and invalidations, and the cache library with PHP language bindings.

One of TxCache's goals is to make it easier to add caching to a new or existing application. The TxCache library makes it straightforward to designate a function as cacheable. However, ensuring that the program has functions suitable for caching still requires some effort. Below, we describe our experiences adding support for caching to the RUBiS benchmark and to MediaWiki.

7.1 Porting RUBiS

RUBiS [2] is a benchmark that implements an auction website modeled after eBay where users can register items for sale, browse listings, and place bids on items. We ported its PHP implementation to use TxCache. Like many small PHP applications, the PHP implementation of RUBiS consists of 26 separate PHP scripts, written in an unstructured way, which mainly make database queries and format their output. Besides changing code that begins and ends transactions to use TxCache's interfaces, porting RUBiS to TxCache involved identifying and designating cacheable functions. The existing implementation had few functions, so we had to begin by dividing it into functions; this was not difficult and would be unnecessary in a more modular implementation.

We cached objects at two granularities. First, we cached large portions of the generated HTML output (except some headers and footers) for each page. This meant that if two clients viewed the same page with the same arguments, the previous result could be reused. Second, we cached common functions such as authenticating a user's login, or looking up information about a user or item by ID. Even these fine-grained functions were often more complicated than an individual query; for example, looking up an item requires examining both the active

items table and the old items table. These fine-grained cached values can be shared between different pages; for example, if two search results contain the same item, the description and price of that item can be reused.

We made a few modifications to RUBiS that were not strictly necessary but improved its performance. To take better advantage of the cache, we modified the code for display lists of items to obtain details about each item by calling our GET-ITEM cacheable function rather than performing a join on the database. We also observed that one interaction, finding all the items for sale in a particular region and category, required performing a sequential scan over all active auctions, and joining it against the users table. This severely impacted the performance of the benchmark with or without caching. We addressed this by adding a new table and index containing each item's category and region IDs. Finally, we removed a few queries that were simply redundant.

7.2 Porting MediaWiki

We also ported MediaWiki to use TxCache, to better understand the process of adding caching to a more complex, existing system. MediaWiki, which faces significant scaling challenges in its use for Wikipedia, already supports a variety of caches and replication systems. Unlike RUBiS, it has an object-oriented design, making it easier to select cacheable functions.

MediaWiki supports master-slave replication for the database server. Because the slaves cannot process update transactions and lag slightly behind the master, MediaWiki already distinguishes the few transactions that must see the latest state from the majority that can accept the staleness caused by replication lag (typically 1–30 seconds). It also identifies read/write transactions, which must run on the master. Although we used only one database server, we took advantage of this classification of transactions to determine which transactions can be cached and which must execute directly on the database.

Most MediaWiki functions are class member functions. Caching only pure functions requires being sure that functions do not mutate their object. We cached only static functions that do not access or modify global variables (MediaWiki rarely uses global variables). Of the non-static functions, many can be made static by explicitly passing in any member variables that are used, as long as they are only read. For example, almost every function in the TITLE class, which represents article titles, is cacheable because a TITLE object is immutable.

Identifying functions that would be good candidates for caching was more challenging, as MediaWiki is a complex application with myriad features. Developers with previous experience with the MediaWiki codebase would have more insight into which functions were frequently used. We looked for functions that were involved in common requests like rendering an article, and member functions of commonly-used classes. We focused on functions that constructed objects based on data looked up in the database, such as fetching a page revision. These were good candidates for caching because we can avoid the cost of one or more database queries, as well as the cost of post-processing the data from the database to fill the fields of the object. We also adapted existing caches like the localization cache, which stores translations of user interface messages.

8 Evaluation

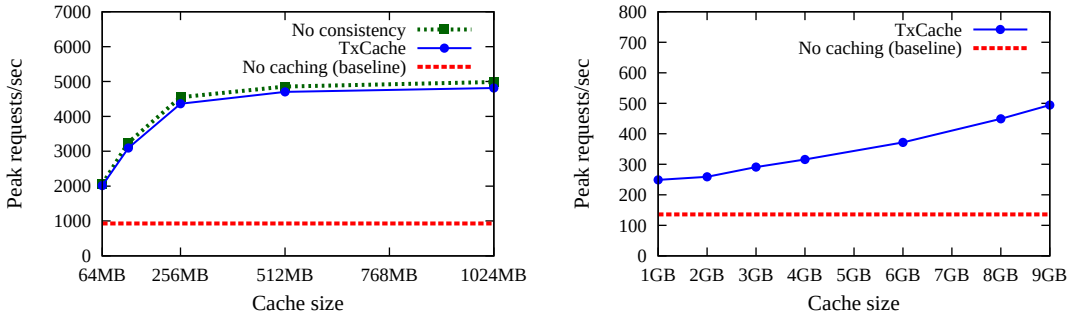
We used RUBiS as a benchmark to explore the performance benefits of caching. In addition to the PHP auction site implementation described above, RUBiS provides a client emulator that simulates many concurrent user sessions: there are 26 possible user interactions (*e.g.* browsing items by category, viewing an item, or placing a bid), each of which corresponds to a transaction. We used the standard RUBiS "bidding" workload, a mix of 85% read-only interactions (browsing) and 15% read/write interactions (placing bids) with a think time with negative exponential distribution and 7-second mean.

We ran our experiments on a cluster of 10 servers, each a Dell PowerEdge SC1420 with two 3.20 GHz Intel Xeon CPUs, 2 GB RAM, and a Seagate ST31500341AS 7200 RPM hard drive. The servers were connected via a gigabit Ethernet switch, with 0.1 ms round-trip latency. One server was dedicated to the database; it ran PostgreSQL 8.2.11 with our modifications. The others acted as front-end web servers running Apache 2.2.12 with PHP 5.2.10, or as cache nodes. Four other machines, connected via the same switch, served as client emulators. Except as otherwise noted, database server load was the bottleneck.

We used two different database configurations. One configuration was chosen so that the dataset would fit easily in the server's buffer cache, representative of applications that strive to fit their working set into the buffer cache for performance. This configuration had about 35,000 active auctions, 50,000 completed auctions, and 160,000 registered users, for a total database size about 850 MB. The larger configuration was disk-bound; it had 225,000 active auctions, 1 million completed auctions, and 1.35 million users, for a total database size of 6 GB.

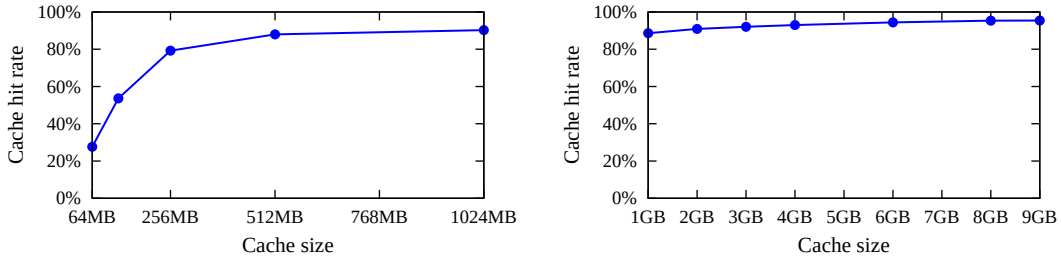
For repeatability, each test ran on an identical copy of the database. We ensured the cache was warm by restoring its contents from a snapshot taken after one hour of continuous processing for the in-memory configuration and one day for the disk-bound configuration.

For the in-memory configuration, we used seven hosts as web servers, and two as dedicated cache nodes. For the larger configuration, eight hosts ran both a web server and a cache server, in order to make a larger cache available.



(a) In-memory database (b) Disk-bound database

Figure 5: Effect of cache size on peak throughput (30 second staleness limit)



(a) In-memory database (b) Disk-bound database

Figure 6: Effect of cache size on cache hit rate (30 second staleness limit)

8.1 Cache Sizes and Performance

We evaluated RUBiS's performance in terms of the peak throughput achieved (requests handled per second) as we varied the number of emulated clients. Our baseline measurement evaluates RUBiS running directly on the Postgres database, with TxCache disabled. This achieved a peak throughput of 928 req/s with the in-memory configuration and 136 req/s with the disk-bound configuration.

We performed this experiment with both a stock copy of Postgres, and our modified version. We found no observable difference between the two cases, suggesting our modifications have negligible performance impact. Because the system already maintains multiple versions to implement snapshot isolation, keeping a few more versions around adds little cost, and tracking validity intervals and invalidation tags simply adds an additional bookkeeping step during query execution.

We then ran the same experiment with TxCache enabled, using a 30 second staleness limit and various cache sizes. The resulting peak throughput levels are shown in Figure 5. Depending on the cache size, the speedup achieved ranged from 2.2× to 5.2× for the in-memory configuration and from 1.8× to 3.2× for the disk-bound configuration. The RUBiS PHP benchmark does not perform significant application-level computation; even so, we see a 15% reduction in total web server CPU usage. Cache server load is low, with most CPU overhead in kernel time, suggesting inefficiencies in the kernel's TCP stack as the cause. Switching to a UDP protocol might alleviate some of this overhead [32].

Figure 6(a) shows that for the in-memory configuration, the cache hit rate ranged from 27% to 90%, increasing linearly until the working set size is reached, and then growing slowly. Here, the cache hit rate directly translates into a performance improvement because each cache hit represents load (often many queries) removed from the database. Interestingly, we always see a high hit rate on the disk-bound database (Figure 6(b)) but it does not always translate into a large performance improvement. This workload exhibits some very frequent queries (*e.g.* looking up a user's nickname by ID) that can be stored in even a small cache, but are also likely to be in the database's buffer cache. It also has a large number of data items that are each accessed rarely (*e.g.* the full bid history for each item). The latter queries collectively make up the bottleneck, and the speedup is determined by how much of this data is in the cache.

8.2 Varying Staleness Limits

The staleness limit is an important parameter. By raising this value, applications may be exposed to increasingly stale data, but are able to take advantage of more cached

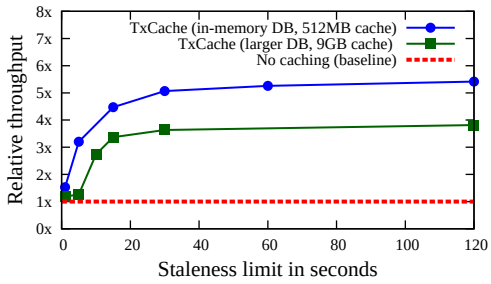


Figure 7: Impact of staleness limit on peak throughput

data. An invalidated cache entry remains useful for the duration of the staleness limit, which is valuable for values that change (and are invalidated) frequently.

Figure 7 compares the peak throughput obtained by running transactions with staleness limits from 1 to 120 seconds. Even a small staleness limit of 5-10 seconds provides a significant benefit. RUBiS has some objects that are expensive to compute and have many data dependencies (indexes of all items in particular regions with their current prices). These objects are invalidated frequently, but the staleness limit permits them to be used. The benefit diminishes at around 30 seconds, suggesting that the bulk of the data either changes infrequently (such as information about inactive users or auctions), or is accessed multiple times every 30 seconds (such as the aforementioned index pages).

8.3 Costs of Consistency

A natural question is how TxCache's guarantee of transactional consistency affects its performance. We explore this question by examining cache statistics and comparing against other approaches.

We classified cache misses into four types, inspired by the common classification for CPU cache misses:

- *compulsory miss*: the object was never in the cache
- *staleness miss*: the object has been invalidated, and its staleness limit has been exceeded
- *capacity miss*: the object was previously evicted
- *consistency miss*: some sufficiently fresh version of the object was available, but it was inconsistent with previous data read by the transaction

Figure 8 shows the breakdown of misses by type for four different configurations. Our cache server unfortunately cannot distinguish staleness and capacity misses. We see that consistency misses are the least common by a large margin. Consistency misses are rare, as items in the cache are likely to have overlapping validity intervals, either because they change rarely or the cache contains multiple versions. Workloads with higher staleness limits experience more consistency misses (but fewer overall misses) because they have more stale data that must be matched

	in-memory DB			disk-bound
	512 MB 30 s stale	**512 MB 15 s stale**	**64 MB 30 s stale**	**9 GB 30 s stale**
Compulsory	33.2%	28.5%	4.3%	63.0%
Stale / Cap.	59.0%	66.1%	95.5%	36.3%
Consistency	7.8%	5.4%	0.2%	0.7%

Figure 8: Breakdown of cache misses by type. Figures are percentage of total misses.

to other items valid at the same time. The 64 MB-sized cache's workload is dominated by capacity misses, because the cache is smaller than the working set. The disk-bound experiment sees more compulsory misses because it has a larger dataset with limited locality, and few consistency misses because the update rate is slower.

The low fraction of consistency misses suggests that providing consistency has little performance cost. We verified this experimentally by modifying our cache to continue to use our invalidation mechanism, but to read any data that was valid within the last 30 seconds, blithely ignoring consistency. The results of this experiment are shown as the "No consistency" line in Figure 5(a). As predicted, the benefit it provides over consistency is small. On the disk-bound configuration, the results could not be distinguished within experimental error.

9 Related Work

High performance web applications use many different techniques to improve their throughput. These range from lightweight application-level caches which typically do not provide transactional consistency, to database replication systems that improve database performance while providing the same consistency guarantees, but do not address application server load.

9.1 Application-Level Caching

Applying caching at the application layer is an appealing option because it can improve performance of both the application servers and the database. Dynamic web caches operate at the highest layer, storing entire web pages produced by the application, requiring them to be regenerated in their entirety when any content changes. These caches need to invalidate pages when the underlying data changes, typically by requiring the application to explicitly invalidate pages [37] or specify data dependencies [9, 38]. TxCache obviates this need by integrating with the database to automatically identify dependencies.

However, full-page caching is becoming less appealing to application developers as more of the web becomes personalized and dynamic. Instead, web developers are increasingly turning to application-level data caches [4, 16, 24, 26, 34] for their flexibility. These caches allow the application to choose what to store, including query results, arbitrary application data (such as Java or .NET

objects), and fragments of or whole web pages.

These caches present to applications a GET/PUT/DELETE hash table interface, so the application developer must choose keys and correctly invalidate objects. As we argued in Section 2.1, this can be a source of unnecessary complexity and software bugs. Most application object caches have no notion of transactions, so they cannot ensure even that two accesses to the cache return consistent values. Some support transactions *within* the cache, allowing applications to atomically update objects in the cache [34, 16], but none maintain transactional consistency with the database.

9.2 Database Replication

Another popular alternative is to deploy a caching or replication system within the database layer. These systems replicate the data tuples that comprise the database, and allow replicas to perform queries on them. Accordingly, they can relieve load on the database, but offer no benefit for application server load.

Some replication systems guarantee transactional consistency by using group communication to execute queries [12, 19], which can be difficult to scale to large numbers of replicas [13]. Others offer weaker guarantees (eventual consistency) [11, 27], which can be difficult to reason about and use correctly. Still others require the developer to know the access pattern beforehand [3] or statically partition the data [8].

Most replication schemes used in practice take a primary copy approach, where all modifications are processed at a master and shipped to slave replicas, usually asynchronously for performance reasons. Each replica then maintains a complete, if slightly stale, copy of the database. Several systems defer update processing to improve performance for applications that can tolerate limited amounts of staleness [6, 28, 30]. These protocols assume that each replica is a single, complete snapshot of the database, making them infeasible for use in an application object cache setting where it is not possible to maintain a copy of every object that could be computed. In contrast, TxCache's protocol allows it to ensure consistency even though its cache contains cached objects that were generated at different times.

Materialized views are a form of in-database caching that creates a view table containing the result of a query over one or more base tables, and updating it as the base tables change. Most work on materialized views seeks to incrementally update the view rather than recomputing it in its entirety [15]. This requires placing restrictions on view definitions, *e.g.* requiring them to be expressed in the select-project-join algebra. TxCache's application-level functions, in addition to being computed outside the database, can include arbitrary computation, making incremental updates infeasible. Instead, it uses invalidations, which are easier for the database to compute [7].

10 Conclusion

Application data caches are an efficient way to scale database-driven web applications, but they do not integrate well with databases or web applications. They break the consistency guarantees of the underlying database, making it impossible for the application to see a consistent view of the entire system. They provide a minimal interface that requires the application to provide significant logic for keeping cached values up to date, and often requires application developers to understand the entire system in order to correctly manage the cache.

We provide an alternative with TxCache, an application-level cache that ensures all data seen by an application during a transaction is consistent, regardless of whether it comes from the cache or database. TxCache guarantees consistency by modifying the database server to return validity intervals, tagging data in the cache with these intervals, and then only retrieving values from the cache that were valid at a single point in time. By using validity intervals instead of single timestamps, TxCache can make the best use of cached data by lazily selecting the timestamp for each transaction.

TxCache provides an easier programming model for application developers by allowing them to simply designate cacheable functions, and then have the results of those functions automatically cached. The TxCache library handles all of the complexity of managing the cache and maintaining consistency across the system: it selects keys, finds data in the cache consistent with the current transaction, and automatically detects and invalidates potentially changed objects as the database is updated.

Our experiments with the RUBiS benchmark show that TxCache is effective at improving scalability even when the application tolerates only a small interval of staleness, and that providing transactional consistency imposes only a minor performance penalty.

Acknowledgments

We thank James Cowling, Kevin Grittner, our shepherd Amin Vahdat, and the anonymous reviewers for their helpful feedback. This research was supported by NSF ITR grants CNS-0428107 and CNS-0834239, and by NDSEG and NSF graduate fellowships.

References

[1] C. Amza, E. Cecchet, A. Chanda, S. Elnikety, A. Cox, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. TR02-388, Rice University, 2002.

[2] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site

benchmarks. *Proc. Workshop on Workload Characterization*, Nov. 2002.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. Middleware '03*, Rio de Janeiro, Brazil, June 2003.

[4] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp. WebSphere dynamic cache: Improving J2EE application experience. *IBM Systems Journal*, 43(2), 2004.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD '95*, San Jose, CA, June 1995.

[6] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proc. SIGMOD '06*, Chicago, IL, 2006.

[7] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proc. VLDB '02*, Hong Kong, China, 2002.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: flexible database clustering middleware. In *Proc. USENIX '04*, Boston, MA, June 2004.

[9] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. INFOCOM '99*, Mar 1999.

[10] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *Proc. USENIX '09*, San Diego, CA, June 2009.

[11] A. Downing, I. Greenberg, and J. Peha. OSCAR: a system for weak-consistency replication. In *Proc. Workshop on Management of Replicated Data*, Nov 1990.

[12] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proc. SRDS '05*, Washington, DC, 2005.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. SIGMOD '96*, Montreal, QC, June 1996.

[14] P. J. Guo and D. Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *Proc. TAPP '10*, San Jose, CA, Feb. 2010.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD '93*, Washington, DC, June 1993.

[16] JBoss Cache. http://www.jboss.org/jbosscache/.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC '97*, El Paso, TX, May 1997.

[18] K. Keeton, C. B. Morrey III, C. A. N. Soules, and A. Veitch. LazyBase: Freshness vs. performance in information management. In *Proc. HotStorage '10*, Big Sky, MT, Oct. 2009.

[19] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *Transactions on Database Systems*, 25(3):333–379, 2000.

[20] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[21] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *Proc. ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[22] MediaWiki bugs. http://bugzilla.wikimedia.org/. Bugs #7474, #7541, #7728, #10463.

[23] MediaWiki bugs. http://bugzilla.wikimedia.org/. Bugs #8391, #17636.

[24] memcached: a distributed memory object caching system. http://www.danga.com/memcached.

[25] NCache. http://www.alachisoft.com/ncache/.

[26] OracleAS web cache. http://www.oracle.com/technology/products/ias/web_cache/.

[27] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP '97*, Saint Malo, France, 1997.

[28] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proc. Middleware '05*, Toronto, Canada, Nov. 2004.

[29] PostgreSQL. http://www.postgresql.org/.

[30] U. Röhm, K. Böhm, H. Schek, and H. Schuldt. FAS: a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proc. VLDB '02*, Hong Kong, China, 2002.

[31] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, Heidelberg, Germany, Nov. 2001.

[32] P. Saab. Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919, Dec. 2008.

[33] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Proc. VMCAI '05*, Paris, France, Jan. 2005.

[34] N. Sampathkumar, M. Krishnaprasad, and A. Nori. Introduction to caching with Windows Server AppFabric. Technical report, Microsoft Corporation, Nov 2009.

[35] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Transactions on Networking*, 11(1):149–160, Feb. 2003.

[36] M. Stonebraker. The design of the POSTGRES storage system. In *Proc. VLDB '87*, Brighton, United Kingdom, Sept. 1987.

[37] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. *SIGCOMM Comput. Commun. Rev.*, 29(4):163–174, 1999.

[38] H. Zhu and T. Yang. Class-based cache management for dynamic web content. In *Proc. INFOCOM '01*, 2001.