Application-Level Caching with Transactional Consistency

by

Dan R. K. Ports

M.Eng., Massachusetts Institute of Technology (2007)
S.B., S.B., Massachusetts Institute of Technology (2005)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2012

Certified by . . . .
Barbara H. Liskov
Institute Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Application-Level Caching with Transactional Consistency

by

Dan R. K. Ports

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 2012, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

ABSTRACT

Distributed in-memory application data caches like *memcached* are a popular solution for scaling database-driven web sites. These systems increase performance significantly by reducing load on both the database and application servers. Unfortunately, such caches present two challenges for application developers. First, they cannot ensure that the application sees a consistent view of the data within a transaction, violating the isolation properties of the underlying database. Second, 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.

This thesis addresses both of these problems in a new cache called TxCache. TxCache is a transactional cache: it ensures that any data seen within a transaction, whether from the cache or the database, reflects a slightly stale but consistent snapshot of the database. TxCache also offers a simple programming model. Application developers simply designate certain functions as cacheable, and the system automatically caches their results and invalidates the cached data as the underlying database changes.

Our experiments found that TxCache can substantially increase the performance of a web application: on the RUBiS benchmark, it increases throughput by up to 5.2× relative to a system without caching. More importantly, on this application, TxCache achieves performance comparable (within 5%) to that of a non-transactional cache, showing that consistency does not have to come at the price of performance.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

ACKNOWLEDGMENTS

This thesis, like the OSDI paper on which it is based, is the result of a collaboration with Austin Clements, Irene Zhang, Sam Madden, and Barbara Liskov. The work described here would not have been possible without their contributions. Austin and Irene implemented major parts of TxCache; I remain amazed by the speed with which they translated my random ideas into a functional system. More than once, Sam's keen acumen for databases provided the solution to a problem that appeared intractable.

My work – and especially this thesis – was substantially improved by working with my advisor, Barbara Liskov. Barbara's principled approach to research and emphasis on the value of providing the right abstractions were a strong influence on my research; I hope that influence shows clearly in this thesis. Barbara's suggestions also improved the clarity of this work. I also thank the other members of my thesis committee, Sam Madden and Frans Kaashoek for their helpful feedback (especially at short notice!).

It has been a pleasure to work with the other members of the Programming Methodology Group: Winnie Cheng, James Cowling, Dorothy Curtis, Evan Jones, Daniel Myers, David Schultz, Liuba Shrira, and Ben Vandiver. In particular, James Cowling and David Schultz provided essential support during the writing of this thesis – both through their positive feedback and encouragement, and by being willing to discuss even the most esoteric details of transaction processing and database implementation.

Before I joined PMG, I finished a master's thesis working with David Karger on peer-to-peer systems. His enthusiasm and insight encouraged me to continue in systems research. As part of that project, I worked closely with Frans Kaashoek, Robert Morris, and various members of the PDOS group. Frans and Robert taught me many valuable lessons about how to do research and how to write a paper.

I spent a few summers as an intern at VMware, where I benefited greatly from working with Carl Waldspurger, Mike Chen, Tal Garfinkel, E. Lewis, Pratap Subrahmanyam, and others.

I also benefited from recent work with Kevin Grittner on the implementation of

Serializable Snapshot Isolation in PostgreSQL. That work influenced the invalidation mechanism in Chapter 7 and the discussion in Appendix B, but more importantly, Kevin provided valuable insight into the real-world benefits of serializable isolation.

I thank my family for their support. My parents, Thomas Ports and Catherine Covey, were a constant source of encouragement. Most of all, I thank my wife, Irene Zhang. It isn't an exaggeration to say that this work wouldn't have been possible without her. She has supported me in many ways: from having helpful discussions about this work, to reviewing drafts of my writing, to providing encouragement when I needed it most.

PREVIOUSLY PUBLISHED MATERIAL

This thesis significantly extends and revises work previously published in the following paper:

> Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010.

Parts of Appendix B are adapted from the following forthcoming publication:

> Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)*, Istanbul, Turkey, August 2012. To appear.

FUNDING

This research was supported by the National Science Foundation under ITR grants CNS-0428107 and CNS-0834239, and by a National Defense Science and Engineering Graduate (NDSEG) Fellowship sponsored by the Office of Naval Research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This thesis addresses the problem of maintaining consistency in a cache of application-computed objects.

Application-level caches are a popular solution for improving the scalability of complex web applications: they are widely deployed by many well-known websites. They are appealing because they can be implemented with a simple, scalable design, and their flexibility allows them to address many bottlenecks, as we discuss below.

However, existing caches place a substantial burden on application developers. They do not preserve the isolation guarantees of the underlying storage layer, introducing potential concurrency bugs. They also place the responsibility for locating and updating data in the cache entirely in the hands of the application, a common source of bugs. Consider, for example, the following report of a major outage at one of the world's most-visited web sites:

> Early today Facebook was down or unreachable for many of you for approximately 2.5 hours. This is the worst outage we've had in over four years, and we wanted to first of all apologize for it. We also wanted to provide much more technical detail on what happened and share one big lesson learned.
>
> The key flaw that caused this outage to be so severe was an unfortunate handling of an error condition. An automated system for verifying configuration values ended up causing much more damage than it fixed.

The intent of the automated system is to check for configuration values that are invalid in the cache and replace them with updated values from the persistent store. This works well for a transient problem with the cache, but it doesn't work when the persistent store is invalid.

Today we made a change to the persistent copy of a configuration value that was interpreted as invalid. This meant that every single client saw the invalid value and attempted to fix it. Because the fix involves making a query to a cluster of databases, that cluster was quickly overwhelmed by hundreds of thousands of queries a second.

...

The way to stop the feedback cycle was quite painful – we had to stop all traffic to this database cluster, which meant turning off the site.

(Robert Johnson, Facebook [50])

This incident highlights both the importance of application-level caches – not a mere optimization, they are a critical part of the site's infrastructure – and the challenges of cache management.

This thesis presents techniques that address these challenges while preserving the flexibility and scalability benefits of existing application-level caches.

1.1 THE CASE FOR APPLICATION-LEVEL CACHING

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 storage layer (often a relational database) for managing persistent state. Either layer can become a bottleneck [6]. Either type of bottleneck can be addressed, but with difficulty. Increasing database capacity is typically a difficult and costly proposition, requiring careful partitioning or complex distributed databases. Application server bottlenecks can be easier to address – simply adding more nodes is usually an option – but no less costly, as these nodes don't come for free.

Not surprisingly, many types of caches have been used to address these problems, ranging from page-level web caches [20, 26, 107, 109] to database replication systems [8, 54, 75] and query/mid-tier caches [39, 58, 100]. But increasingly complex application logic and more personalized web content has made it more useful to cache the result of *application computations*. As shown in Figure 1-1, the cache does not lie in front of the application servers (like a page-level web cache) or between the application server and database (like a query cache). Rather, it allows application to cache arbitrary objects. These objects are typically generated from the results of one or more database queries along with some computation in the application layer.

This flexibility allows an application-level cache to replace existing caches: it can act as database query cache, or it can act as a web cache and cache entire web pages. But application-level caching is more powerful because it can cache *intermediate computations*, which can be even more useful. For example, many web sites have highly-personalized content, rendering whole-page web caches largely useless; application-level caches can separate common content from customized content and cache the common content separately so it can be shared between users. Database-level query caches are also less useful in an environment where processing is increasingly done within the application. Unlike database-level solutions, application-level caches can also address application server load; they can avert costly post-processing of database records, such as converting them to an internal representation, or generating partial HTML output. 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.

1.2 EXISTING CACHES ARE DIFFICULT TO USE

Existing application-level caches typically present a hash table interface to the application, allowing it to GET and PUT arbitrary objects identified by a key. This interface offers two benefits:

- the interface is flexible, in that it allows the application to use the cache to store many different types of objects. As discussed above, it can be used to hold database query results, entire generated web pages, or anything in between.

- the interface lends itself to a simple, scalable implementation. A typical example

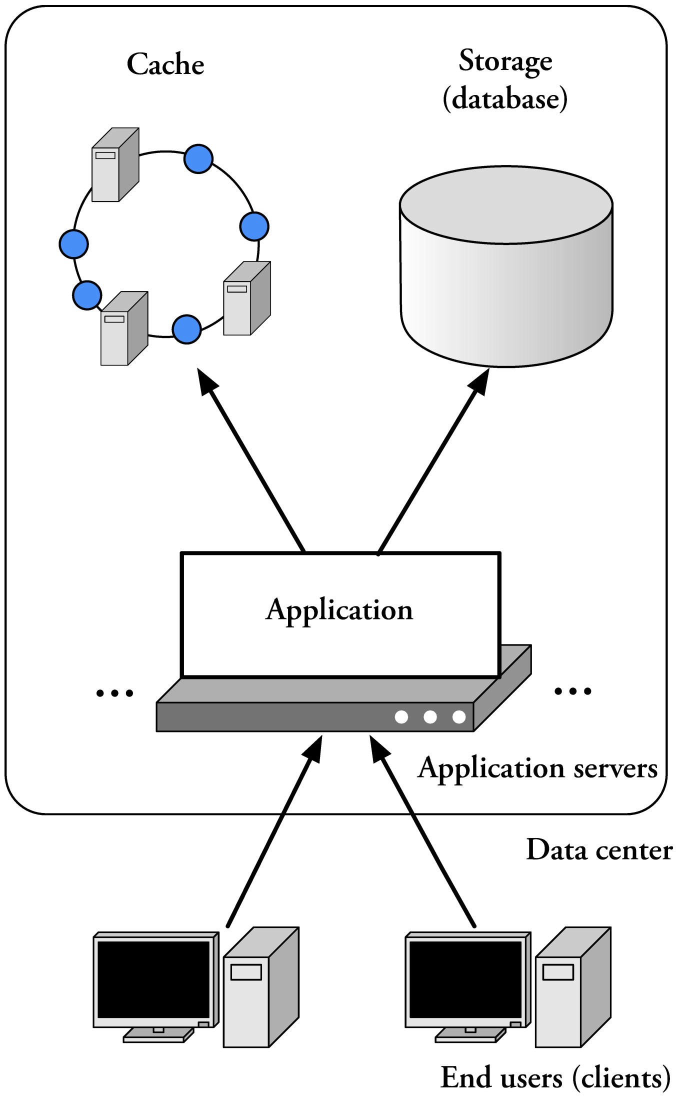


Figure 1-1: Architecture of a system using an application-level cache

is memcached [65], which stores objects on a cluster of nodes using a consistent hashing algorithm [52] – making the cache a simple implementation of a distributed hash table [82,89,96,108]. Importantly, the cache is stored entirely in memory, and does not attempt to do any processing other than returning the object identified by the requested key. This architecture scales well; the largest memcached deployment has thousands of nodes with hundreds of terabytes of in-memory storage [56].

But the GET/PUT interface has a serious drawback: it leaves responsibility for managing the cache entirely with the application. This presents two challenges for developers, which we address in this thesis.

Challenge 1: Transactional Consistency. First, existing caches 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 underlying storage layer 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, these consistency guarantees are violated when data is accessed from the cache..

The anomalies caused by inconsistencies between cached objects can cause incorrect information to be exposed to the user. Consider, for example, an eBay-like auction site. Such an application might wish to cache basic metadata about an auction (its title and current price) separately from more detailed information like the history of bidders. This division would be useful because the basic item metadata is used to construct indexes and search results, whereas both objects are needed when a user views the detailed auction information. Placing a new bid should update both objects. If a subsequent request sees inconsistent versions of the objects, it might, for example, display the latest auction price but not the updated bid history, leaving a user confused about whether his bid has been registered.

Accordingly, memcached has seen use primarily for applications for which the consequences of incorrect data are low (*e.g.*, social networking websites). Applications with stricter isolation requirements between transactions (*e.g.*, online banking or e-commerce) could also benefit from application-level caching: it would improve their performance, reducing their operating costs and allowing them to scale further. However, adding a cache would require them to sacrifice the isolation guarantees provided by the database, a price that many are unwilling to pay.

Even in cases where exposing inconsistent data to users is not a serious problem, transaction support can still be useful to simplify application development. Serializable isolation between transactions allows developers to consider the behavior of transactions in isolation, without worrying about the race conditions that can occur between concurrent transactions. Compensating for a cache that violates these guarantees requires complex application logic because the application must be able to cope with temporarily-violated invariants. For example, transactions allow the system to maintain referential integrity by removing an object and any references to it atomically. With a cache that doesn't support transactions, the application might see (and attempt to follow) a reference to an object that has been deleted.

Challenge 2: Cache Management. The second challenge existing caches present for application developers is that they must explicitly manage the cache. That is, the application is responsible for assigning names to cached values, performing lookups, and keeping the cache up to date. This explicit management places a substantial burden on application developers, and introduces the potential for errors. Indeed, as we discuss in Chapter 3, cache management has been a common source of programming errors in applications that use memcached.

A particular challenge for applications is that they must explicitly *invalidate* data in the cache when updating the backing storage. That is, the cache requires developers to be able to identify every cached application computation whose value might have been affected by a database modification. This analysis is difficult to do, particularly in a complex and evolving application, because it requires global reasoning about the entire application to determine which values may be cached, violating the principles of modularity.

1.3 CONTRIBUTIONS

This thesis addresses both challenges described above in the context of a new transactional, application-level cache which we call TxCache. TxCache aims to preserve the flexibility of existing application-level caches while minimizing the challenges it poses for application developers: adding caching to a new or existing application should be as simple as indicating which data should be cached.

We describe how to build an application-level cache that guarantees *transactional consistency* across the entire system. That is, within a transaction, all data seen by the application reflects a consistent snapshot of the database, regardless of whether the data obtained from the cache or computed directly from database queries. This preserves the isolation guarantees of the underlying storage layer. Typically, the entire system can provide serializable isolation (assuming that the database itself provides serializability). TxCache can also provide snapshot isolation, when used with a database that provides this weaker isolation guarantee.

TxCache provides a consistency model where the system allows read-only transactions to access stale – but still consistent – snapshots. Applications can indicate a freshness requirement, specifying the age of stale data that they are willing to tolerate We argue that this model matches well with the requirements of web applications, and show that it improves cache utilization by allowing cache entries to remain useful for a longer period.

Furthermore, we propose a simple programming model based on *cacheable functions*. In this model, applications do not need to interact explicitly with the cache, avoiding the challenges of cache management described above. Instead, developers simply designate certain existing application functions as cacheable. A cache library then handles inserting the result of the function into the cache and retrieving that result the next time the function is called with the same arguments. Rather than requiring applications to invalidate cache data, we integrate the database, cache, and library to track data dependencies and automatically invalidate cache data.

Towards these ends, this thesis makes the following technical contributions:

- a protocol based on *validity intervals* for ensuring that transactions see only consistent data, even though that data consists of cached objects that were computed at different times

- techniques for modifying an existing multiversion database to generate validity information which can be attached to cache objects

- a *lazy timestamp selection* algorithm that selects which snapshot to use for a read-only transaction based on the application's freshness requirement and the availability of cached data.

- an automatic invalidation system, based on *invalidation tags* for tracking the data dependencies of cache objects, and *invalidation locks*, which use a variant on existing concurrency control mechanisms to detect data dependencies and generate notifications when they are modified.

We have implemented the TxCache system. We evaluated the effectiveness of its programming model in two ways. We ported the RUBiS web application benchmark, which emulates an auction site, to use TxCache. We also examined MediaWiki, a popular web application that uses memcached, and analyzed some bugs related to its use of caching. Both experiences suggest that our programming model makes it easier to integrate caching into an existing application, compared to existing caches.

We evaluated cache performance using the RUBiS benchmark [7]. Compared to a system without caching, our cache improved peak system throughput by 1.5 – 5.2×, depending on the system configuration and cache size. Moreover, we show that the system performance and cache hit rate is only slightly (less than five percent) below that of a non-transactional cache, showing that consistency does not have to come at the expense of performance.

1.4 OUTLINE

This thesis is organized as follows:

We begin with a high-level overview of the system architecture and how it integrates with existing components in a web application, and state our assumptions (Chapter 2). We then describe TxCache's cacheable function programming model, explain how to implement a simplified version of the system that provides this programming model but does not support transactions, and discuss how the programming model can avoid common sources of caching-related application bugs (Chapter 3).

We then turn to the question of how to provide support for whole-system transactional consistency. Chapter 4 describes the semantics that the system provides for transactions. Chapter 5 explains how the system ensures that all data accessed by the application within a transaction reflects a consistent view of the storage state; it introduces the concept of validity intervals and explains how the cache and library use them. The consistency protocol requires some support from the storage layer; Chapter 6 explains how to provide this support. We describe both how to design a new storage system (here, a simple block store) to provide the necessary support, and how to retrofit it into an existing relational database. Chapter 7 discusses how the system tracks data dependencies and uses invalidations to notify the cache of database changes that affect cached objects.

Chapter 8 evaluates the performance of TxCache using the RUBiS benchmark. Chapter 9 surveys the related work, and Chapter 10 concludes.

Appendix A describes an extension to the system that allows read/write transactions to safely use cached data. Appendix B provides an explanation about how TxCache's consistent reads property allows the system to ensure either serializability or snapshot isolation.

2 ARCHITECTURE AND MODEL

2.1 SYSTEM ARCHITECTURE

Our system is designed to be used in an environment like the one depicted in Figure 2-1. We assume an application that consists of one or more application servers that interact with a storage system, typically a database server. Clients (*i.e.*, users) interact with the system by sending requests to application servers; they do not interact directly with the cache or storage. The application servers could be web servers running embedded scripts (*e.g.*, Apache with `mod_php` or `mod_perl`), or dedicated application servers that receive requests from front-end web servers and implement the business logic (as with with Sun's Enterprise Java Beans); both are common deployment options.

The storage layer is typically a relational database, though other types of storage like key-value stores are possible. We assume that the application uses this for storing all of its persistent state. In order for our system to provide transactional guarantees, the storage system itself must of course provide transaction support. We require that it provides either serializable isolation of transactions, or snapshot isolation [10]. Databases that provide serializable isolation must do so using a concurrency control mechanism that ensures that the commit order of transactions matches a serial order; most common concurrency control strategies, including strict two-phase locking [38] and optimistic concurrency control [55], have this property. Appendix B explains

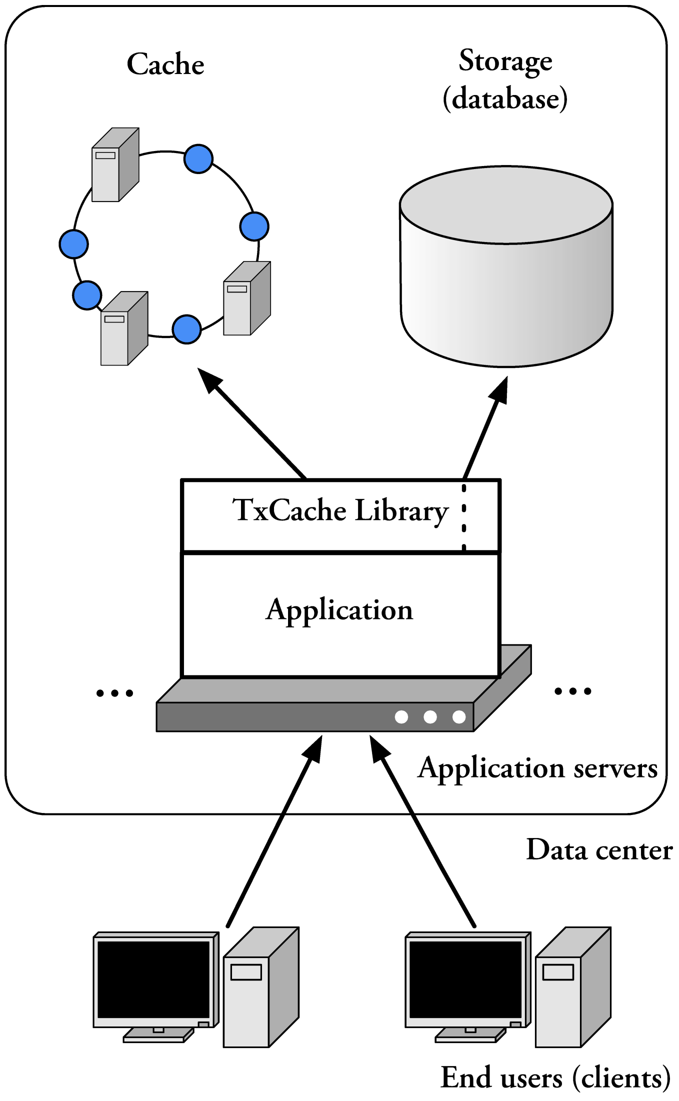


Figure 2-1: System architecture

the rationale for this requirement, and describes how to extend the system to support databases where it does not hold.

TxCache's consistency protocol is based on versioning, and requires some additional support from the storage layer, imposing some additional requirements. It must expose the ability to run read-only transactions on a specific recent snapshot, and must report certain validity information. These requirements are unique to our system; we specify them precisely in Section 5.3 and show how to modify an existing multiversion concurrency control system[1] to support them in Chapter 6. The requirements are not onerous; we added this support to the PostgreSQL DBMS with less than 2000 lines of modifications.

Like the typical application-level cache architecture shown in Figure 1-1, Figure 2-1 shows that our cache is not a part of the database, nor is it an intermediary between the cache and database. However, unlike in previous systems, applications do not interact with the cache explicitly. TxCache introduces a new component: a cache library that runs on each application server and mediates all accesses to the cache. The library implements TxCache's cacheable-function programming model eliminating the need for the application to access the cache directly and allowing application code to remain largely oblivious to the cache. It is also responsible for part of the consistency protocol, ensuring that the application always sees a transactionally-consistent view of the storage.

The cache is partitioned across a set of cache nodes, each running an instance of our cache server. These nodes may be run on dedicated hardware, or they may share resources with other servers. For example, the cache servers may run on the same machines as the application servers, in order to take advantage of memory on these servers that would otherwise go unused.

[1]Though not a strict requirement, these modifications are only likely to be practical for databases that use multiversion concurrency control. This requirement is not particularly restrictive: multiversion concurrency control is a common isolation mechanism implemented by most of today's major database management systems, and is popular for web applications because it offers higher performance for read-only operations.

2.2 ASSUMPTIONS

We assume that all application servers and all cache nodes know the membership of the system, *i.e.*, the IP addresses of every cache server. In particular, we assume that they agree on the *same* view of the system membership. We developed a membership service [31] that provides consistent membership views; other group communication services [4, 14, 86] are also options. We expect the membership to change infrequently, so the cost of maintaining these membership views is negligible. (In smaller deployments, it may even be practical for an administrator to maintain the system membership configuration by hand.)

Cache nodes may fail; we assume that they fail by crashing and lose their state. We discuss how the system responds to failures of cache nodes in Section 3.3.1. Fault tolerance of the storage layer is beyond the scope of this work; we do not discuss how to recover from failures of the storage system beyond simply restarting the cache after a database failure. Application servers may also fail by crashing, but as this has no real impact on the system design we do not discuss it.

All application servers, cache nodes, and the storage layer are fully trusted. We also trust the network that connects them to not modify or corrupt requests. We assume that only trusted application servers are able to send requests to the cache nodes and storage system; this could be enforced using firewall rules or access control policy.

One of our other assumptions is that the application servers and cache nodes within a deployment are on the same local network, *e.g.*, within the same datacenter. This is not a requirement for correctness of the protocol, but a low latency interconnect is important for performance.

3
CACHEABLE FUNCTIONS: A SIMPLE PROGRAMMING MODEL

There are only two hard problems in Computer Science: cache invalidation and naming things.
— Phil Karlton

This thesis aims to improve the usability of application-level caching in two ways. The first, which is addressed in this chapter, is to introduce a new programming model that makes it easy to seamlessly integrate caching into existing systems and frees application developers from the responsibilities of cache management. (The second, providing support for transactions, is the subject of Chapters 4–7.)

The goal of our programming model is to make the process of adding caching to an existing system as simple as deciding what should be cached. In our programming model, developers simply designate certain functions as *cacheable functions*, which automatically and transparently cache their results. These functions behave identically regardless of whether the cache is used, so making a function cacheable is a purely local change: callers of these functions do not need to be modified. Unlike existing systems, our programming model does not require applications to explicitly manage the cache. Applications are not responsible for naming and locating values in the cache, or for invalidating cached values when the database is changed. As we demonstrate, these tasks are challenging – and a frequent source of bugs – because they require global reasoning about the application.

This chapter presents the basic programming model, ignoring issues related to transaction support and concurrency control. Section 3.1 explains the interface presented to application developers. Section 3.2 examines how this interface prevents common problems that currently plague developers who use application-level caching. Section 3.3 then describes how to build a cache that implements this model.

3.1 THE CACHEABLE FUNCTION MODEL

The TxCache programming model is organized around *cacheable functions*. These are actual functions in the program's code, annotated to indicate that their results can be cached. Cacheable functions are essentially memoized [67]: TxCache's library transforms them so that, when called, they first check whether the cache contains the results of a previous call to the same function. If so, that result is returned directly, bypassing the need to execute the function. If not, the function is executed, and its result stored in the cache for future use.

Cacheable functions can make requests to the system's underlying storage layer, *e.g.*, they can perform database queries. Cacheable functions can also be nested, *i.e.*, they can make calls to other cacheable functions. Nested cacheable functions allow applications to cache data at different granularities, which is an important benefit of application-level caching. For example, a web application might choose to make the outermost function that generates a webpage cacheable, allowing the entire page to be cached, but can also cache the functions generating individual elements of that page, which might also be used on different pages.

3.1.1 RESTRICTIONS ON CACHEABLE FUNCTIONS

There are some restrictions on which functions can be cacheable. To be suitable for caching, functions must be *pure*, *i.e.*, they must be deterministic, not have side effects, and depend only on their arguments and the storage layer state. (This definition of "pure" differs slightly from the classic one, in that we allow the function to depend on the storage state, treating it as an implicit argument to the function.)

These requirements are straightforward and correspond to an intuitive notion of what types of computations ought to be cached. It would not make sense to cache

- 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 3-1: TxCache API for designating cacheable functions

a function with side-effects, *e.g.*, one that modifies a file, as this side-effect would then take place only on a cache miss. Similarly, it would not make sense to cache a function that is nondeterministic, such as one that returns a random number or the current time.

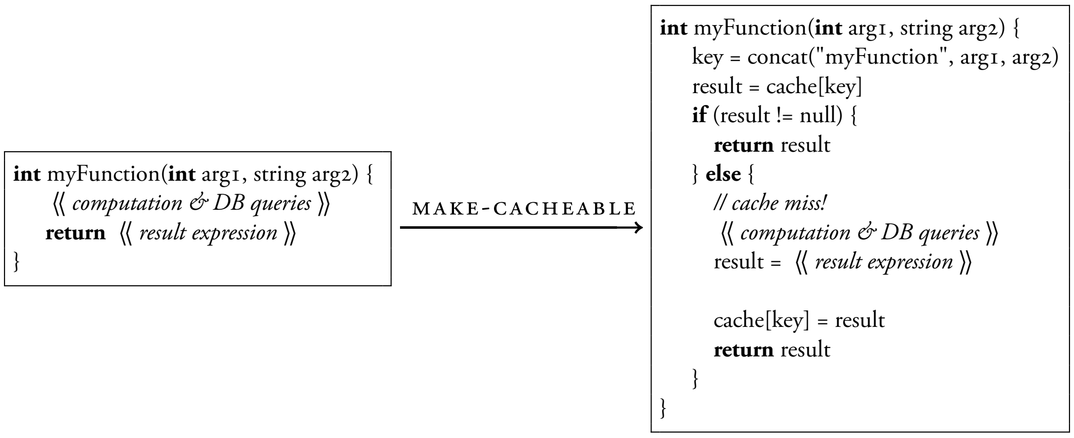
Another aspect of this requirement is that if a cacheable function invokes storage layer operations, they must also be deterministic. This is not always the case for SQL database queries; Vandiver discusses some techniques for finding and eliminating nondeterminism in SQL queries in the context of a database replication system [103, 104].

TxCache currently relies upon programmers to ensure that they only cache suitable functions. However, this requirement could be enforced using static analysis to identify pure functions [91]. It could also be enforced by using dynamic instrumentation to detect and refuse to cache functions that modify state or invoke non-deterministic functions [41]. As a sanity check, our implementation can detect certain non-deterministic functions at runtime and issue a warning; this identified a bug in the RUBiS benchmark we discuss in Section 8.2, but the check is not intended as a comprehensive solution.

3.1.2 MAKING FUNCTIONS CACHEABLE

Figure 3-1 shows the API that TxCache presents to applications. It is presented here in an abstract form using higher-order procedures; this is similar to the API that we provide in Python. In other languages, the details may differ slightly; for example, our PHP implementation takes slightly different form because PHP does not support higher-order procedures.

As the figure shows, the API is minimal: it consists of a single operation, MAKE-CACHEABLE. (This is only a slight simplification; the transactional version of the



```
int myFunction(int arg1, string arg2) {
    ⟨⟨ computation & DB queries ⟩⟩
    return ⟨⟨ result expression ⟩⟩
}
```

```
int myFunction(int arg1, string arg2) {
    key = concat("myFunction", arg1, arg2)
    result = cache[key]
    if (result != null) {
        return result
    } else {
        // cache miss!
        ⟨⟨ computation & DB queries ⟩⟩
        result = ⟨⟨ result expression ⟩⟩

        cache[key] = result
        return result
    }
}
```

Figure 3-2: The MAKE-CACHEABLE operation transforms a normal function into a cached function

system adds a few mostly-standard transaction management functions like COMMIT and ABORT, but no more.) The MAKE-CACHEABLE operation transforms a function into a cacheable function, as shown in Figure 3-2.

Assuming that the functions provided to MAKE-CACHEABLE are indeed pure functions, the resulting cacheable functions behave identically to the original functions. From the perspective of a user of the function, the only difference between the case where the result is available in the cache and the case where it has to be computed anew is the speed of execution. This unchanged behavior is important for modular software development: marking a function as cacheable doesn't require making changes to other parts of the system that depend on it.

3.1.3 AUTOMATIC CACHE MANAGEMENT

What is mainly noteworthy about the API in Figure 3-1 is what it *does not* include. Application developers are only required to indicate what functions they would like to cache. Everything else is handled by the TxCache system; applications are not required to access the cache directly. As a result, applications are not required to track which data is in the cache, where it is stored, or whether it is up to date. In particular, applications are not required to assign names to cached objects, and are not required to notify the cache when cached values change. TxCache instead provides an automatic invalidation mechanism, described in Chapter 7, which tracks data dependencies and notifies the cache when the underlying data in the database changes.

3.2 DISCUSSION

TxCache's cacheable-function programming model offers significant usability advantages over standard application-level caches. The standard interfaces for these caches is a hash table interface: applications GET and PUT arbitrary objects in the cache, identified by application-defined keys. This interface is used by many well-known application-level caches, including memcached [65], Redis [84], JBoss Cache [49], AppFabric [92], and others.

The hash table interface provided by other caches presents two specific challenges to application developers: they are responsible for naming and locating items in the cache, and they are responsible for invalidating cached objects when an update to the database affects them. Both are challenging in large systems and are a common source of caching errors. In TxCache's programming model, these are no longer the responsibility of the application, eliminating this source of bugs.

We examine the problems with naming and invalidations in existing caches, and how TxCache avoids them, in the context of the MediaWiki application [62]. MediaWiki is an application for serving collaboratively-editable "wiki" sites. It is used to power the Wikipedia encyclopedia, one of today's ten most popular websites, as well as several other high-traffic websites. To support these heavy workloads – while providing a rich set of features and dynamic content – MediaWiki uses memcached extensively for caching. Among the objects it caches are the rendered text of articles, the parse trees from which they are generated, metadata about articles and users, and translations of user interface messages. Our analysis of MediaWiki consisted of examining memcached-related bug reports from the MediaWiki bug database, and adapting MediaWiki to use TxCache to cache some of the computations that it currently uses memcached for.

3.2.1 Naming

In a key-value cache interface like the one memcached provides, applications need to explicitly store and look up data in the cache. To do so, application developers must identify the data they wish to access using a cache key. These cache keys must uniquely identify objects in the cache: the same key must be used everywhere to refer to a single object, and no two distinct objects can share a cache key. This requirement seems obvious, but 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 [63]. One representative example (MediaWiki bug #7541), concerned caching of a user's watchlist, which displays recent changes to pages the user has flagged as being of interest. Each user's watchlist was cached using a cache key derived from the user's ID. However, MediaWiki also allows the user to specify the number of changes to display in the watchlist, and this was not reflected in the cache key. As a result, the same results were returned even when the user requested to display a different number of days worth of changes.

It is easy to see how these types of errors might arise in a large, complex application: the code that implements this feature underwent numerous changes in the years since caching was added to it. Any modifications that parameterize the output must also modify the cache key, but this requirement is easy to overlook. In particular, the changes to this module were made by several different developers, all of whom would need to be aware of how caching is being used to avoid errors.

An error like this could not occur with TxCache. TxCache uses cacheable functions as its unit of caching, and uses the function and its arguments to identify the cached value. Because cacheable functions are pure, the arguments uniquely identify the result. Adding a new customization to the watchlist page would require adding an additional argument to the cacheable function, and that argument would be used to distinguish between cached values.

3.2.2 INVALIDATIONS

An even more challenging problem with existing caches is that they require applications to explicitly update or invalidate cached results when modifying the database. These explicit invalidations require global reasoning about the application, violating modularity. To add caching for an object, an application developer must be able to identify every place in the application that might change the value of the object, and add an invalidation. Similarly, an application developer who adds code that modifies data in the database must know which cached values might depend on it, and invalidate them. The cached values and the updates that must invalidate them may be located in different modules and may be written by different developers, making it difficult to keep track of when invalidations are necessary.

As a result, applications that use memcached have cache invalidation code scattered through many functions in many modules. Developers typically rely on ad hoc coordination methods: MediaWiki maintains a text file (`memcached.txt`) that lists the various objects stored in the cache, the cache keys used to identify them, and the situations in which they need to be invalidated. But this hardly guarantees that the resulting code will be correct; indeed, the text file itself contains a disclaimer that it is "incomplete, out of date".

Several MediaWiki bugs resulted from missing or incorrect invalidations [64]. Consider, for example, the process of editing an article on Wikipedia: what cached objects must be invalidated once the update is committed to the database? Some are easy to identify: the cached objects that contain the article text are clearly no longer valid. However, other objects have less obvious dependencies on this change, and must also be invalidated. One example is the USER object corresponding to the user who made the change, which includes various metadata including the user's edit count (which is used to restrict access to certain features to experienced users only). 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 (MediaWiki bug #8391), 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.

Such an error could not happen with TxCache, as it does not require applications to explicitly invalidate cached objects. Rather, it incorporates a dependency tracking mechanism which would have identified that editing a page modified some data that the USER object depended on, and invalidated that object.

3.2.3 LIMITATIONS

By organizing cached data around cacheable functions rather than providing a hash table interface, TxCache imposes more structure on cached data. This structure allows it to simplify cache management by avoiding naming and invalidation issues. However, it does provide a small decrease in flexibility: there are some use cases that can be supported by memcached and other caches, but not by TxCache.

Soft State

TxCache makes the assumption that all cached objects are derived from persistent state in the database (or other storage system). This corresponds to the most common use case of application-level caching. However, some users of memcached also use it as a simple storage system for soft state that is never stored in the backing store. For example, it is sometimes used to track the number of requests from a particular IP address to implement rate limiting as protection against denial of service attacks. This information is needed only in the short term, and does not require durability or consistency, so storing it in the cache is a viable option.

We do not attempt to support this use case (short-term unreliable storage); we target only caching data derived from a persistent backing store. Of course, applications can still use memcached or another existing cache for this purpose.

Invalidations vs. Updates

TxCache includes an automatic invalidation system that notifies the cache when a cached object is no longer valid because of a database change. The next time the application requests the invalidated object, it must recompute it. In most existing systems, applications are responsible for keeping the cache up to date. When the application makes a change to the database, it must identify the cached objects that are affected, which imposes a significant burden on the programmer. However, once they have done so, they can choose either to invalidate these objects, or *update* them in place: they can determine the new value and replace the cache entry directly, making it immediately accessible for future lookups. TxCache does not support this mode of operation.

Gupta showed that using in-place updates instead of invalidations can improve the performance of a memcached-using web application by 0–25% [44, 45]. This result is in the context of an object-relational mapping system that only caches objects corresponding to fixed set of well-defined query patterns. In such a model, it is immediately apparent which changes need to be made to update cached objects. In contrast, TxCache's programming model supports caching of functions including arbitrary computation, so updating a cached object requires recomputing it. In particular, TxCache supports caching complex objects (such as the entire content of web pages that involve many database queries) which would be considerably more expensive to recompute, making the benefit of updates over invalidations significantly smaller.

3.3 IMPLEMENTING THE PROGRAMMING MODEL

The TxCache system implements the programming model described above. This section describes a basic implementation of the system that does not provide support for transactions. In subsequent chapters, we extend this system with support for transactions (Chapters 4–6) and for an automatic invalidation system (Chapter 7).

Recall the system architecture from Figure 2-1. End users interact with application servers (*i.e.*, web servers) which process their requests. In doing so, they interact with a storage layer (*e.g.*, a database). TxCache introduces two new components: a cache, comprised of multiple cache servers, and a library that runs on each application server and translates cacheable function calls to cache accesses.

3.3.1 CACHE SERVERS

TxCache stores the results of application computations in its cache, which is distributed across a set of cache servers. The cache presents a hash table interface: it maps keys to associated values. Only the TxCache library uses this interface; applications do not interact with the cache directly.

The cache is implemented as a simple distributed hash table. It partitions data by key among multiple hosts using consistent hashing [52]. We assume that a membership service provides every node in the system with a consistent view of the system membership (we discuss how membership changes are handled later). Therefore, any participant can immediately determine which cache node is responsible for storing a particular hash key. The cache nodes, therefore, can operate completely independently; they do not need to communicate with each other. Each cache node simply accepts LOOKUP and STORE requests for the keys it is responsible for, using a simple RPC protocol.

The interface and partitioning of our cache are similar to peer-to-peer distributed hash tables, such as Chord [96], Pastry [89], and many others. However, our system lacks many of their more demanding requirements, and so avoids most of their complexity. In particular, multi-hop routing is unnecessary at the scale of perhaps hundreds of nodes in a data center; at this scale, it is feasible for each node to maintain the complete system membership.

Each cache node's data is stored entirely in memory, ensuring that it can always give an immediate response (without blocking on disk I/O, for example). This allows the cache server to be implemented as a single-threaded application which processes one request at a time, avoiding the need for locking within the cache sever. Multiprocessor systems can simply run multiple independent instances of the cache server, relying on consistent hashing to achieve load balancing among them.

The cache stores data in a hash table indexed by key. 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. The cache evicts entries using a least-recently-used replacement policy.

Reconfiguration: Cache Server Failures and Membership Changes

The set of active cache servers can change over time. Cache nodes may fail, or may be removed from the system. New nodes can be added to the system, perhaps to increase the overall capacity of the cache. For the most part, the system handles these membership changes acceptably without any special protocols: we do not use replication to ensure availability. Because the cache can evict entries arbitrarily, the system must already be prepared to handle the absence of data in the cache. If a cache node fails, attempts to access it can simply be treated as cache misses until the node is repaired or removed from the membership list. Similarly, when a new node is added, it does not have any state; it simply indicates a cache miss on all requests until it receives the appropriate state.

As an optimization, newly joining cache nodes can transfer state from the node that previously stored the cached values it will be responsible for. Furthermore, replication can optionally be integrated into the system if losing a cache node's worth of data during a failure is a serious performance concern. However, replication has a clear downside: it reduces the effective storage capacity of the cache (or increases the cost of the cache hardware) by a factor of at least two. A useful property for implementing either a state transfer or replication protocol is that the TxCache server (though not the simplified version shown here) stores versioned data. Because each version is immutable, it is possible to use DHT-like synchronization protocols [23, 30, 95] rather than needing more expensive state machine replication protocols [70, 93] to ensure consistency.

3.3.2 TXCACHE LIBRARY

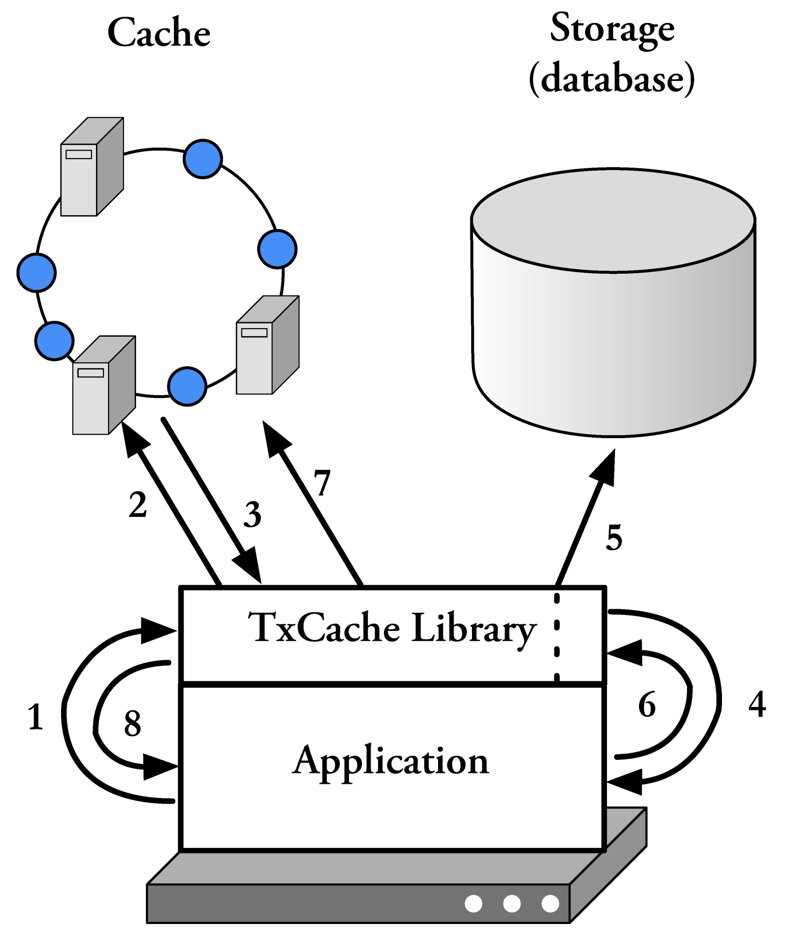
The TxCache library runs on each application server in the system. It is responsible for implementing the cacheable-function programming model: as the application invokes cacheable functions, the library checks for their values in the cache, and stores computed values in the cache as necessary.

The library's MAKE-CACHEABLE function takes a function as input and produces a wrapper function. This wrapper function is a memoized version of the original function. Figure 3-3 shows what happens when the application invokes one of these wrapper functions. The function first constructs a cache key from the function name and the arguments to the function, serializing them in the appropriate (language-specific) way.[1] The library then hashes the cache key to determine the cache server responsible for storing it (using consistent hashing), and sends a LOOKUP request.

If the cache server has the appropriate value available, it returns it to the TxCache library. The TxCache library then returns that directly to the application, avoiding the need to execute the function.

If the TxCache library is unable to obtain the cached result from the cache server, it must recompute it. This might happen if the result has been invalidated, evicted from the cache, or was never computed, or if the cache node is inaccessible. The TxCache library then makes an upcall to the application, requesting that it invoke the cacheable function's implementation (the function originally provided to MAKE-CACHEABLE). As that function executes, it may make queries to the storage layer; the TxCache library monitors those queries to track dependencies for invalidation purposes, as we describe in Chapter 7. When the execution of the function completes, it returns the result to the wrapper function created by the TxCache library. Before passing the return value back to the application, the TxCache library's wrapper function serializes the value and sends a STORE request to the appropriate cache server, inserting the value into the cache so that it can be used for later calls.

[1]Using only the function name and arguments as the cache key assumes that the function will not change. In environments with a dynamic code base, one might want to also incorporate a version number or a hash of the code itself.



	1	application calls wrapper function
	2	TxCache library sends LOOKUP request to cache
	3	cache returns value or indicates cache miss
(cache miss only)	**4**	TxCache library makes upcall to application's implementation function
(cache miss only)	**5**	application executes function, making queries to database as necessary
(cache miss only)	**6**	application returns result to TxCache library
(cache miss only)	**7**	TxCache library adds value to cache
	8	TxCache library returns value to caller

Figure 3-3: Control flow during a call to a cacheable function

4 CONSISTENCY MODEL

TxCache is designed to make application-level caching easy to integrate with existing systems. The cacheable function model described in the previous chapter supports this goal by making caching transparent: designating a function as cacheable should not produce any visible difference to callers of that function. However, there is one way in which the cache we have described so far violates transparency: like other existing caches, it does not provide support for transactions. Without the cache, the application accesses the storage layer directly and can rely on it to ensure isolation between transactions (*e.g.*, serializability). If the application uses data from the cache, however, there is no guarantee that cached values will be consistent with each other or with data from the database.

The second goal of this work, therefore, is to provide transaction support in an application-level cache. Our model of consistency ensures isolation between transactions but explicitly trades off freshness, allowing applications to see slightly stale states of data in the cache. Nevertheless, the data is still consistent: the whole system can guarantee either serializability or snapshot isolation, depending on which the storage layer provides. This chapter defines these properties and argues that the relaxed freshness guarantee is appropriate for applications.

4.1 TRANSACTIONAL CONSISTENCY

TxCache provides transactional consistency for applications that use the cache. Our goal here is to preserve the isolation guarantees of the underlying storage layer. Most existing caches fall short at this goal: while they are typically used with databases capable of ensuring strong isolation between transactions (*e.g.*, serializability or snapshot isolation), the caches do not ensure consistency between data accessed from the cache and data accessed from the database. In contrast, TxCache ensures system-wide isolation guarantees.

The strongest guarantee the system can provide is serializability:

- **serializable isolation**: the execution of concurrent transactions produces the same effect as an execution where transactions executed sequentially in some order

This property is the standard correctness criterion for concurrent execution of transactions. Most databases provide serializable isolation as an option (as required by the ANSI SQL standard [3]). When used with such a database, TxCache provides whole-system serializability: it ensures serializable isolation of transactions regardless of whether the data they access comes from the cache or directly from the database.

We focus on serializable isolation because it is the strongest standard isolation level. Its strong semantics fit well with our goals because they simplify application development: developers can be certain that if their transactions do the right thing when run alone, they will continue to do so in any mix of concurrent transactions. TxCache ensures that this continues to be true even when the application accesses cached data.

TxCache also supports storage systems that do not provide serializability, but instead offer the weaker guarantee of *snapshot isolation*:

- **snapshot isolation**: each transaction reads data from a snapshot of the committed data as of the time the transaction started, and concurrent transactions are prevented from modifying the same data object

Snapshot isolation is known to be a weaker property than serializability, allowing race conditions between concurrent transactions to cause anomalies that would not occur in a serial execution [10]. Nevertheless, it is a popular option for storage systems to provide (both relational databases and other systems [73]) because it allows more concurrency than serializability, while still preventing many of the most common anomalies. When used with a database that provides snapshot isolation, TxCache provides snapshot isolation for the whole system: it ensures that every transaction sees a consistent snapshot of the data regardless of whether it comes from the cache or the database.

Snapshot isolation the weakest isolation level that our system supports. Even weaker isolation levels are also common – for example, most databases provide a READ COMMITTED mode that ensures only that transactions do not see uncommitted data – but it is not clear that it would be reasonable to use these with our system. In these cases, our cache would provide a *stronger* isolation guarantee than the database, which is unlikely to be useful.

4.2 RELAXING FRESHNESS

Our system guarantees transactional consistency (either serializability or snapshot isolation) but does not attempt to guarantee *freshness*: the data in the cache may not reflect the latest data in the storage layer. TxCache relaxes this property in order to provide better performance for applications that can tolerate some stale data.

Typical caches strive to keep the data in the cache as up to date as possible with respect to the backing store. Despite this, keeping the cache completely up to date is not practical. Most caches return slightly stale data simply because modified data does not reach the cache immediately. To ensure that the cache always reflects the latest changes made to its backing store would require using locking and an atomic commitment protocol between the cache and storage (*e.g.*, two-phase commit). This expensive protocol would negate much of the performance benefit of the cache. Furthermore, even with such an implementation, clients still do not have any guarantee that the values returned from the cache are up to date *by the time they are received by the client*.

Trading off freshness for consistency is a common strategy in concurrency control. For example, in snapshot isolation, all data read by a transaction reflects a snapshot of the database taken at the time the snapshot began [10]. This allows read-only transactions to run without blocking read/write transactions, at the expense of freshness: the read-only transaction might see older values of objects that were modified while it ran.

Freshness is explicitly not a goal of our system. Instead, we allow read-only transactions to see somewhat stale, but still consistent views. That is, read-only transactions effectively read data from a snapshot taken up to t seconds *before* the transaction started. (Read/write transactions, on the other hand, always see the latest data, for reasons we discuss in Section 4.3.) This feature is motivated by the observation that many applications can tolerate a certain amount of staleness [53], and using stale cached data can improve the cache's hit rate [61]. The staleness limit t can be specified by the application, reflecting that different applications tolerate different levels of stale data. It can even be specified on a per-transaction basis, because different operations may have significantly different requirements.

This means that there is a total order of the states visible to each read-only transaction, but that this order need not match the order in which transactions start or commit. That is, the system can provide serializability of these transactions, but does not provide linearizability [47], a stronger condition that requires that operations take effect at some point between their invocation and response. (This comparison is somewhat imprecise, in that linearizability is defined in terms of individual operations, whereas our guarantees are in terms of multi-operation transactions.)

The increased flexibility of allowing transactions to see older states improves the cache hit rate by allowing access to cached data even when it is not the most current version available. If multiple transactions access the same data within a short interval, they can use the same cached value, even if that value was updated in the meantime. As our experiments in Section 8.5 show, this is an important benefit for frequently-updated data.

4.2.1 DEALING WITH STALENESS: AVOIDING ANOMALIES

Because TxCache allows read-only transactions to run in the past, where they may see stale data, one potential concern is that using stale data might introduce new anomalies, or be incompatible with some applications. We argue here that these consistency semantics do in fact align well with application requirements and user expectations, as long as some restrictions are imposed on how far in the past transactions can run. TxCache makes it possible for applications to express these restrictions as freshness requirements.

The main freshness requirement applications typically have is *causality*: after a user has seen some data from a particular time, later actions on behalf of that user should not see older data. This requirement can easily be enforced: the application keeps track of the latest timestamp the user has seen, and specifies that TxCache should require a read-only transaction to run no earlier than that timestamp. TxCache leaves it to the application to track this and specify the appropriate freshness constraint, rather than attempting to implement causality tracking in the TxCache library, because causality requirements are application-specific. For example, a web service might consist of multiple application servers, each processing requests from many users; each user's requests should see a later state than that user's previous requests, even if they were processed by a different server. Furthermore, the data seen by one user should not impose restrictions on a transaction run by a *different* user that happens to execute on the same server. In this case, the application can store the timestamp in each user's per-session state, *e.g.*, by storing it in a HTTP cookie. More generally, in distributed applications where multiple application servers interact with each other, causal interactions can be tracked using Lamport clocks [57], as used in systems like Thor [60] and ISIS [15].

The reason that it is safe to run transactions in the past is that, within a short time period, applications must already be tolerant of transactions being reordered. If, for example, a read-write transaction W and a read-only transaction R are sent at nearly the same time, the database might process them in either order, and so W's change might or might not be visible to R. Either result must be acceptable to the client executing R, unless it knows that W took place from some external source. In other words, unless a causal dependency between the two clients exists, their transactions can be considered concurrent (in the sense of Lamport's *happens-before* relation [57]) and hence either ordering is acceptable.

This type of consistency also fits well with the model of a web service. Users expect that the results they see reflect a recent, consistent state of the system, but there is no guarantee that it is the *latest* state. By the time the result is returned to the user, a concurrent update might have made it stale. Indeed, if executed on a snapshot isolation database, there is no guarantee that the results are even current at the time the database returns an answer; new committed results may have arrived during query execution. If a transparent web cache is present, these effects are exacerbated. Even without a cache, to ensure that a result remains correct until the user acts on it (*e.g.*, an airline ticket reservation is held until paid for), some kind of lease or long-duration lock is necessary; TxCache does not interfere with these techniques, because it does not affect read-write transactions.

Of course, causal interactions external to the system cannot be tracked. For example, nothing can be done about a user who sees a result on one computer, then moves to another computer and loads the same page. To limit the impact of this problem, applications can request a minimum freshness, *e.g.*, they might never be willing to accept data more than 30 seconds old. The value of this minimum freshness may vary dramatically depending on application requirements; some applications are able to tolerate significantly stale data.

4.3 READ/WRITE TRANSACTIONS

Our consistency model allows *read-only* transactions to see stale data. We do not, however, allow read/write transactions to see stale data. We forbid this because it can allow certain non-serializable anomalies that would make it more difficult for application developers to ensure correct behavior. An application might read some cached objects, seeing an earlier state of the system, and use these to make an update. Consider the example of an auction website, which we discuss in Chapter 8. Here, a small amount of staleness is useful for read-only queries: it is acceptable if the list of auctions is a few seconds out of date. But it is important to use the latest data for updates: if a user requests to place a bid, and the system sees stale data when checking if the bid is higher than the previous one, it might miss an even higher bid placed in the interim.

These types of concurrency bugs are similar to the anomalies that can occur when using a snapshot isolation database [10] – but worse, because our relaxed-freshness model would allow applications to see a snapshot of the system from *before* the transaction start. (This model has been formalized as Generalized Snapshot Isolation [36].) Our system could easily be adapted to work in this way, but we prefer the simpler approach of not allowing read/write transactions to see stale data, on usability grounds. It has been our experience that even the anomalies resulting from snapshot isolation are poorly understood by developers, and the analysis required to detect these anomalies is difficult to do [80]. The fact that snapshot isolation anomalies have been discovered in deployed systems supports this conclusion [51].

A simple way to ensure that read/write transactions do not see stale data is to prevent them from using the cache entirely. In our implementation, we use the cache only for read-only transactions and bypass it for read/write transactions; read/write transactions access the storage layer directly, using its normal concurrency control mechanisms. Our experiments (Chapter 8) show that this approach is effective for workloads with a high fraction of read-only transactions, a common pattern for many web applications.

We describe a more sophisticated approach in Appendix A that allows read/write transactions to use cached objects. It uses an approach based on optimistic concurrency: when executing a read/write transaction, clients only access the latest versions of data from the cache, and validate on commit that the data they read remains current. An additional complication, which we also address in Appendix A, is that read/write transactions need to be able to see the effects of modifications made earlier in the transaction, but these effects should not become visible to other concurrent transactions until the transaction commits.

4.4 TRANSACTIONAL CACHE API

The TxCache API is summarized in Figure 4-1. The MAKE-CACHEABLE function remains as described in Section 3.1. Now, the API also allows programmers to group 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 guarantees that all data seen by the application is

- BEGIN-RO(*freshness-requirement*) : Begin a read-only transaction. The transaction sees a consistent view of the system state that satisfies the freshness requirement.
- 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 4-1: TxCache transactional API

consistent with a single state of the storage layer.

The BEGIN-RO operation takes the application's freshness requirement as an argument. This requirement can be expressed either in real-time terms (*e.g.*, the application is only willing to accept states from within the last 30 seconds) or logical terms (*e.g.*, the application must see a state at least as recent as a particular transaction). The latter option is intended to support causal consistency requirements such as ensuring that each user sees the effects of their latest change. Here, transactions are identified by *timestamps*, as discussed in Chapter 5. The COMMIT operation returns the timestamp at which a transaction ran; the application can subsequently use that timestamp as an argument to BEGIN-RO to start a new transaction that must see the effects of the previous one.

5
CONSISTENCY PROTOCOL

This chapter describes the protocol that our system uses to ensure transactional consistency. The protocol is based on a notion of *validity intervals*, which indicate the range of times at which a version of an object was current. These reflect the fact that cached objects are valid not just at the instant they were computed but at a range of times, allowing the system to combine cached objects that were computed at different times as long as they are consistent with each other.

To ensure transactional consistency when using cached objects, TxCache uses a versioned cache indexed by validity interval. We ensure that the storage layer can compute an associated validity interval for each query result, describing the range of time over which its result was valid. 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 transaction, it only retrieves values from the cache and database that were valid at the same time.

5.1 VALIDITY INTERVALS

The basic observation that makes this protocol possible is that every object is valid not just at a single point in time – the time at which it was generated – but at a *range* of times, representing the time range during which it was the most recent

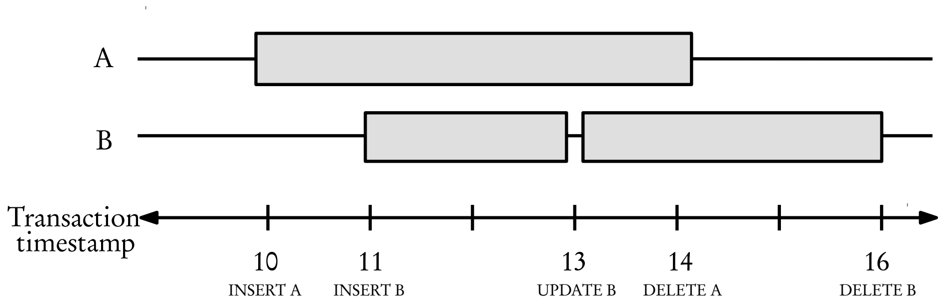


Figure 5-1: Example of validity intervals for two objects

version. This observation makes it possible to combine two cached objects that were computed at different times as long as the data that they rely on has not changed.

We consider each read/write transaction to have a **timestamp** that reflects the ordering in which that transaction's changes became visible to other transactions. These timestamps must have the property that, if two transactions have timestamps t_1 and t_2, with $t_1 < t_2$, then any transaction that sees the effects of transaction t_2 must also see the effects of transaction t_1. For example, the wall-clock time at which a transaction commits can be used as a transaction timestamp, as can a logical timestamp that simply indicates the order in which transactions commit.

The **validity interval** of an object (more precisely, of a *version* of an object) is a pair of timestamps defining the range of time at which that version was current. The lower bound of this interval is the time at which that object became valid, *i.e.*, the commit time of the transaction that created it. The upper bound is a time after which the value may no longer be current, *i.e.*, the commit time of the first subsequent transaction to change the result.

For example, Figure 5-1 shows two objects and their validity intervals. Object A was created by the transaction that executed at timestamp 10 and deleted by the transaction that committed at timestamp 14. We write its validity interval as $[10, 14)$. Object B has two versions, with validity intervals $[11, 13)$ and $[13, 16)$. The transaction at timestamp 13 updates B's value, creating a second version.

Every object's validity interval has a lower bound – the time at which that object became valid. For some objects, the upper bound – the time at which the object is no longer valid – is known. However, for other objects the upper bound may lie in the future; this is the case for objects that are currently valid. We refer to validity intervals with a known upper bound as *bounded* and those where the upper bound is unknown as *unbounded*.

To simplify the explanation of the protocol, we assume in this chapter that all validity intervals are bounded. That is, we assume the system has perfect foresight and knows the timestamp at which an object becomes invalid, even when that timestamp lies in the future. In Chapter 7, we show how to correctly account for objects that are still valid by using *invalidations*: notifications from the database that certain values are no longer current.

5.1.1 PROPERTIES

Validity intervals have the following properties, which our protocol makes use of.

- **intersection**: if a new result is computed by reading two or more objects, the validity interval of the resulting object is the intersection of their validity intervals. We use this property to determine the validity interval of cached objects that depend on multiple data objects from the storage layer.

- **subset**: if an object is valid over an interval $[a, b)$, it is also valid over any subinterval, *i.e.*, any interval $[c, d)$ where $c \geq a$ and $d \leq b$. This seemingly-trivial property is useful because it means that it is acceptable to use conservative estimates of validity intervals when they cannot be accurately computed.

- **uniqueness**: there is exactly one version of any object at any given time, *i.e.*, if two versions of the object have different values, they must have non-intersecting validity intervals. The converse is not true; it is acceptable to have multiple versions, valid at different times, that have the same value.

Most importantly, validity intervals are useful because they allow us to ensure that the system provides transactional consistency. In particular, they allow us to ensure the following property:

- **consistent reads**: if all objects read by a read-only transaction have validity intervals that contain timestamp t, then the transaction sees the same data it would if its reads were executed on the database at time t.

The TxCache library uses this property to ensure that all objects read from either the cache or the database reflect a consistent view of the system state. As we discuss in Appendix B, this property is sufficient to ensure that the cache provides serializable behavior when used with a database that provides serializability, or snapshot isolation when used with a database that provides that.

A generalization of this property is that if the intersection of the validity intervals of all objects read by a transaction has a non-empty interval, then that transaction is consistent with the data at *some* point.

5.2 A VERSIONED CACHE

TxCache stores the results of application computations in its cache, distributed across the set of cache servers. Section 3.3 described the basic architecture of the cache, which presents a hash table interface to the TxCache library. In order to provide support for transactions, we extend our cache to be *versioned*. In addition to its key, each entry in the cache is tagged with its validity interval, as shown in Figure 5-2. 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. The cache stores this information in a hash table, indexed by key, that points to a tree of versions indexed by their validity interval.

Figure 5-3 shows the interface that the cache provides to the TxCache library. The TxCache library can insert objects into the cache using the STORE command. Now, in addition to providing the key-to-value mapping it is adding, it must also provide the associated validity interval. This creates a new entry in the cache. If the cache already contains an existing entry for the same key, it must either have a disjoint validity interval (*i.e.*, the new entry is a distinct version) or the data of both versions must match (*i.e.*, the new entry is a duplicate of the existing entry). Our cache server rejects any request to store a new version of an existing object with different data but an overlapping validity interval, it rejects the request and issues

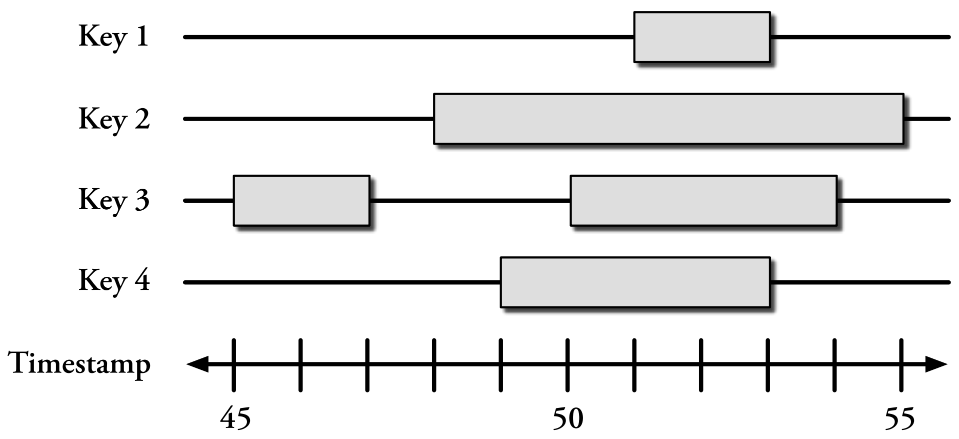


Figure 5-2: An example of versioned data in the cache. Each rectangle is a version of a data item. For example, the data for key 1 became valid with commit 51 and invalid with commit 53, and the data for key 2 became valid with commit 48 and is still valid as of commit 55. There are two versions of the data with key 3; the gap between their validity intervals suggests that there is at least one more version that is not in the cache.

- STORE(*key, value, validity-interval, basis*) : Add a new entry to the cache, indexed by the specified key and validity interval. The *basis* argument is used to track data dependencies for the invalidation mechanism in Chapter 7.
- LOOKUP(*key, timestamp*) → *value, interval* : Returns the value corresponding to *key* whose validity interval contains the specified timestamp, if it is resident in the cache.
- LOOKUP(*key, interval*) → *value, interval* : Returns the value corresponding to *key* whose validity interval intersects the specified interval, if it is resident in the cache.

Figure 5-3: Cache interface. This API is used by the TxCache library to interact with cache servers.

a warning; this indicates an error, typically caused by the application attempting to cache a non-deterministic function. (This warning exposed a bug in the RUBiS benchmark, as we discuss in Section 8.2.)

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

Note, however, that cache LOOKUP operations may fail even if the data was previously stored in the cache. The cache is always free to discard entries if it runs out of space, or during failures. It is not a problem if data is missing: the cache contents are soft state, and can always be recomputed. In our versioned cache, 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.

5.3 STORAGE REQUIREMENTS

Validity intervals give us a way to manage versioned data in the cache. But where do these validity intervals come from? Cached objects are derived from data obtained from the storage layer (*e.g.*, the database), so their validity intervals must also ultimately derive from information provided from the storage layer. Furthermore, the TxCache library must be able to integrate with the storage layer's concurrency mechanism in order to ensure that cached data is consistent with any information accessed directly from the storage layer. TxCache, therefore, imposes two requirements on the storage layer:

- whenever the storage layer responds to a query, it must also indicate the result's validity interval

- BEGIN-RO(*timestamp*) : Start a new read-only transaction running at the specified timestamp.
- BEGIN-RW() : Start a new read/write transaction. This does not require a timestamp; all read/write transactions operate on the latest state of the data.
- COMMIT() → *timestamp*: End the current transaction
- ABORT() : End the current transaction

Figure 5-4: Concurrency control API requirements for the storage layer

- the storage layer must allow the TxCache library to control which timestamp is used to run read-only transactions

The first requirement is needed so that the library can compute the validity intervals of cached objects. The second requirement is needed because a transaction might initially access some cached data, but then miss in the cache and need to obtain data from the database. To ensure that the data obtained from the database is consistent with the cached data previously read, the TxCache library must control which snapshot is used on the database. It does not need this control for read/write transactions, however, as these transactions always bypass the cache and operate on the latest state of the database.

This section specifies the interface that that TxCache library expects from the storage layer. Chapter 6 shows how to modify an existing system to support these requirements.

Figure 5-4 shows the concurrency control API that the TxCache library requires from the storage layer. It requires the usual functions for grouping operations into transactions – BEGIN, COMMIT, and ABORT functions – with a few additions. When starting a transaction, the library will indicate whether that transaction is read-only or read/write. For read-only transactions, the storage layer must allow the library to specify the timestamp at which the transaction runs. TxCache doesn't specify a particular interface for how the application queries the storage layer (for example, a block store and a relational database have very different interfaces) but does require that every query also return an accompanying validity interval.

The storage layer API requirements are similar to TxCache's cache server API (Figure 5-3): both expose to the TxCache library control over query timestamps, and return validity intervals on queries. However, there are some notable asymmetries. First, the cache takes a timestamp as an argument to its LOOKUP function, whereas the storage layer has an explicit BEGIN-RO function that starts a read-only transaction. The active transaction is tracked as connection state, and subsequent queries on the same connection are therefore performed relative to the specified timestamp. We use this interface because it is the standard transaction interface for relational databases; it also allows us to avoid modifying the syntax for query operations, which could be complex. The second difference is that cache LOOKUP operations can specify a range of acceptable timestamps, but here transactions are restricted to run on a specific timestamp. Again, this is partially a pragmatic choice: it matches the way databases are implemented. But there is also a more fundamental reason: some data may not be available in the cache at a given timestamp, so it is useful to specify a range of acceptable timestamps; the storage layer, however, always has access to *all* data for a given timestamp.

5.3.1 PINNED SNAPSHOTS

Ideally, the storage layer would provide the ability to run a read-only transaction at any arbitrary timestamp that the TxCache library specified in the BEGIN-RO call. This requirement can be met by some multiversion storage systems; for example, we describe the design of a block store that provides this interface in Section 6.1.

In practice, however, it isn't always possible to modify an existing storage system, such as a relational database, to support this interface. For reasons we discuss in Section 6.2.1, providing access to a particular version may require the database to record and retain additional metadata; the associated cost can render it infeasible to provide access to *all* previous versions. As a concession to practicality, therefore, we allow the storage layer to provide a more restricted interface that can be easier to implement.

In this more restricted interface, the storage layer allows the TxCache library to start transactions only on certain timestamps in the past, not arbitrary ones. We refer to these timestamps as *pinned snapshots*. Only these timestamps can be used as arguments to BEGIN-RO. If the storage layer provides this interface, it must provide an interface that creates a pinned snapshot from the current database state. TxCache includes a module (the "pincushion", described in Section 6.2.3) that creates pinned snapshots on a regular basis (*e.g.*, every second) and ensures that all application servers know the timestamps of all pinned snapshots. The TxCache library on an application server can also request that a new pinned snapshot be created at the latest timestamp, reflecting the current database state.

5.4 MAINTAINING CONSISTENCY WITH VALIDITY INTERVALS

Validity intervals and timestamps make it possible for the system to ensure serializability. The consistent reads property of validity intervals (Section 5.1.1) indicates that if all objects read by the application during a transaction have validity intervals that contain timestamp t, then the transaction is serializable at time t. The TxCache library uses this property to ensure transactional consistency of all data accessed from the cache or the database. It does so using the following protocol. For clarity, we begin with a simplified version where timestamps are chosen when a transaction begins (Section 5.4.1). In Section 5.4.2, we describe a technique for choosing timestamps lazily to take better advantage of cached data.

5.4.1 BASIC OPERATION

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; it does not use the cache for the duration of this transaction. At the beginning of a read-only transaction, the library selects a timestamp to run the transaction at. If the storage layer uses the pinned snapshot interface, the selected timestamp must correspond to a pinned snapshot. If necessary (*i.e.*, if no sufficiently recent snapshots exist), the library may request a new snapshot on the database and select its timestamp.

The library can delay beginning a read-only transaction on the database (*i.e.*, sending a BEGIN-RO operation) 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 it does start a transaction on the database, the TxCache library specifies the timestamp it selected, ensuring that any results obtained from the database are valid at that timestamp.

When the application invokes a cacheable function, the library checks whether its result is in the cache. To do so, it identifies the responsible cache server using consistent hashing, and sends it a LOOKUP request that 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 application.

If the requested data is not in the cache – or if it is available in the cache but not for the specified timestamp – then the application must recompute the cached value. Doing so requires making an upcall to the application to execute the cacheable function's implementation. As the cacheable function issues queries to the database, the TxCache library accumulates the validity intervals 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 marshals its result and inserts it into the cache, tagged with the accumulated validity interval.

5.4.2 LAZY TIMESTAMP SELECTION

Which timestamp is selected for a read-only transaction can have a significant impact on performance. Figure 5-5 illustrates: if a transaction is run at the latest available timestamp 54, one of the four objects in the cache can be used. If, however, the transaction were run at the earlier timestamp 51 – which might still be consistent with the application's freshness requirement – all four of the objects depicted could be used. A poor choice of timestamp could also be even more limiting: if timestamp 47 were chosen, no cached objects could be used.

But how should the TxCache library choose the timestamp for a read-only transaction? Above, we assumed that the library chooses the transaction's timestamp

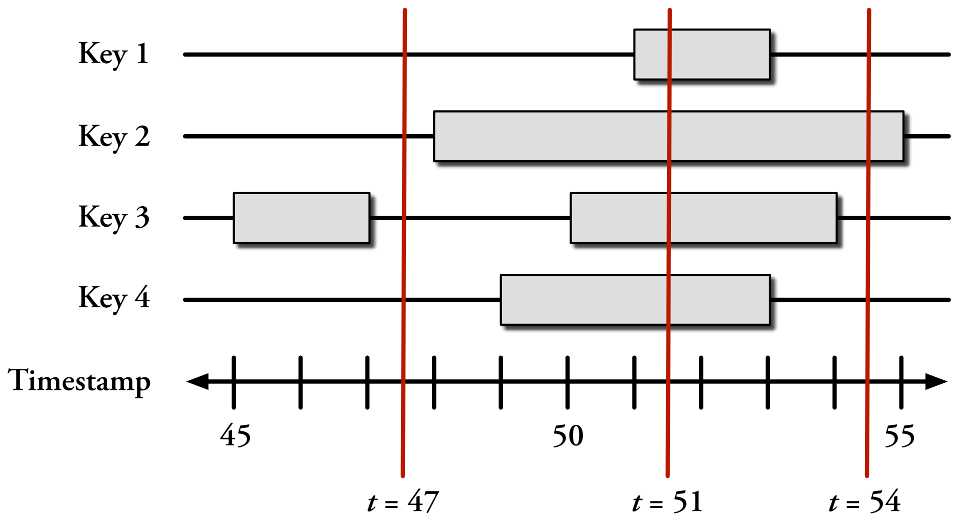
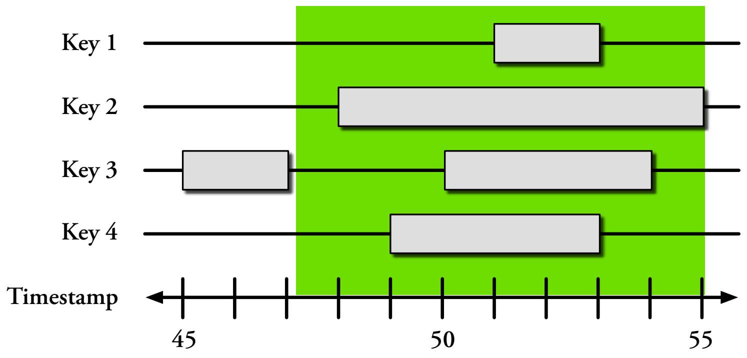


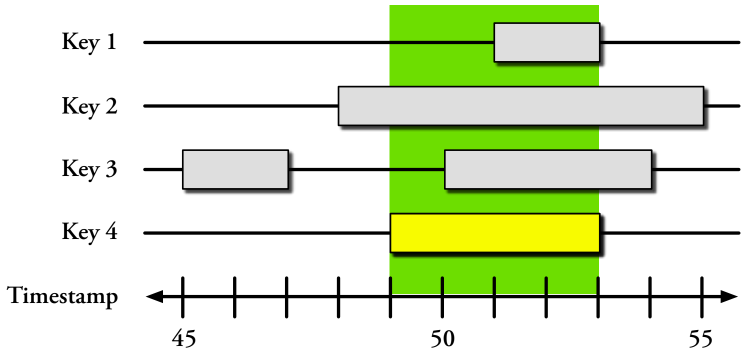
Figure 5-5: An illustration of the importance of timestamp selection. All four objects in the cache are available to transactions running at timestamp 51, but a transaction running at timestamp 54 can use only one of them, and a transaction running at timestamp 47 can use none.

when the transaction starts. Although conceptually straightforward, this approach falls short because the library does not have enough information to make a good decision. It does not know what data is in the cache or what its validity intervals are, as this information is available only on the cache nodes. Furthermore, in our model, the library also does not even know *a priori* what data the transaction will access. Lacking this knowledge, it is not at all clear what policy would provide the best hit rate.

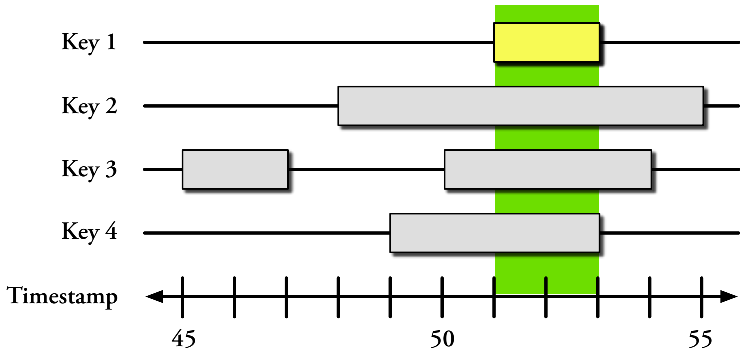
However, the timestamp does not need to be chosen immediately. Instead, it can be chosen lazily based on which cached results are available. This takes advantage of the fact that each cached value is valid over a range of timestamps: its validity interval. For example, consider a transaction that has observed a single cached result x. This transaction can still be serialized at *any* timestamp in x's validity interval. When the transaction next accesses the cache, 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



(a) Initially, the application is willing to accept any data with timestamps that satisfy the application's freshness requirement. Here, that is the interval $[47, 55)$, as indicated by the green region.



(b) After accessing an object in the cache (key 4), the transaction is then constrained to access only objects that were valid at the same time as that object.



(c) As the transaction accesses other objects, the set of acceptable timestamps progressively narrows.

Figure 5-6: Example of lazy timestamp selection

database query result. Figure 5-6 illustrates this process.

Algorithm

Specifically, the algorithm proceeds as follows. When a transaction begins, the TxCache library identifies the set of all timestamps that satisfy the application's freshness requirement. It stores this set as the *pin set*. If the storage layer can run transactions on arbitrary timestamps, this set is simply the set of all timestamps between the application's staleness limit and the current time. If the storage layer uses the more restricted pinned snapshot interface, then this is the set of pinned snapshots in that range. If there are no pinned snapshots in this range, the library requests the creation of a new one and initializes the pin set with it.

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 it contains). 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.

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 forced to select a specific timestamp from the pin set and BEGIN a read-only transaction on the database at the chosen timestamp. This ensures that the application's database queries are performed with respect to that timestamp. Our implementation chooses the latest timestamp in the pin set for this purpose, biasing transactions toward running recent data, though other policies are possible.

The TxCache library may run multiple database queries within the same transaction at different timestamps. This can be necessary if the transaction's pin set changes as a result of cached data it later reads. In this case, the library instructs the database to end its current transaction and start a new one with the newly-chosen timestamp. Thus, from the perspective of the database, the queries are in separate transactions;

however, the TxCache library, by managing validity intervals, ensures that the query results are still consistent with each other.

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.

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 database queries outside a cacheable function.

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 are 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., *there is always at least one timestamp at which the transaction can be serialized.*

The pin set is initially non-empty. It contains the timestamps of all sufficiently-fresh pinned snapshots; if necessary, the library creates a new pinned snapshot. 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. Thus, at least that one timestamp will remain in the pin set after intersecting it with the query's validity interval.

5.4.3 HANDLING NESTED CALLS

Cacheable functions can be nested, *i.e.*, they can call other cacheable functions. This allows applications to cache data at multiple granularities. Supporting nested calls does not require any fundamental changes to the approach above. However, we must keep track of a separate cumulative validity interval 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 the function that calls it, 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).

6
STORAGE LAYER SUPPORT

The protocol that TxCache uses for ensuring whole-system consistency requires some support from the storage layer. Specifically, it requires that the storage layer provide the following two features:

1. It must allow the TxCache library to specify which timestamp a read-only transaction will run at. If the storage layer is not able to provide access to all previous timestamps, it must ensure that application servers are aware of which ones are available.

2. It must report the validity interval associated with every query result it provides

The first requirement is needed to ensure that data obtained from the storage layer (in the event of a cache miss) is consistent with data read from the cache. The second requirement is needed to determine the validity intervals for cached data.

This chapter discusses how to meet these requirements. We discuss how to provide them in two systems. First, we present a simple block store that uses multiversion concurrency (Section 6.1); it gives an example of how to meet these requirements in a simple system designed from the ground up to support them. Next, we discuss how to provide support for relational databases (Section 6.2), where integrating with existing concurrency control mechanisms and complex query processing presents more challenges.

An important concern is that these requirements can be met easily by existing systems. This matters because a cache that operates only with a radically different backing store is of limited value. Although our cache support does require some modifications to the database, we argue that they are not onerous. In particular, we show that existing multiversion databases – which include most of the database systems in common use today – can easily be modified to support these requirements.

6.1 A TRANSACTIONAL BLOCK STORE

The first storage system we describe is a simple transactional block store. We present this system mainly as an instructive example of how one might design a storage system from the ground up to meet our cache's requirements. We note, however, that many practical storage systems do provide a similar interface – either as an end in itself, as in the case of many "NoSQL" key-value stores [32, 37, 66, 84] – or as a primitive for building larger distributed systems [59].

Our block store uses a trivial data model: it maintains a set of *blocks*, identified by a 64-bit integer ID, which store opaque data. The only data operations it provides are to GET the value of a block or replace it with a new value (PUT).

The system guarantees serializable isolation for transactions, even when multiple data blocks are read or modified within a transaction. As shown in Figure 6-1, the block store provides functions to begin, commit, and abort a transaction. Note that operations like GET and PUT do not need to explicitly identify which transaction they are a part of. Rather, the active transaction ID is kept as session state: after a BEGIN operation starts a transaction, subsequent operations sent on the same connection are treated as part of that transaction. We use this connection-oriented interface because it matches the interface provided by relational databases, and we wanted the same TxCache library to be compatible with both.

An important note about the interface provided by this block store is that it allows a read-only transaction to be started at *any* timestamp. That is, it has no notion of pinned snapshots. In order to allow transactions to run at any timestamp, the block store retains old versions of data indefinitely; we do not discuss garbage-collection of old data versions.

- BEGIN-RO(*timestamp*) : Start a read-only transaction running at the specified timestamp. If no timestamp is specified, uses the latest timestamp.
- BEGIN-RW() : Start a read/write transaction; always uses the latest timestamp.
- COMMIT() → *timestamp* : End a transaction, returning the timestamp at which it ran.
- ABORT() : End a transaction, discarding any changes it made.
- PUT(*id, data*) : Store the specified *data* in the block identified by *id*. This replaces the existing data in the block if it exists, or creates it if it does not.
- GET(*id*) → *data, validity* : Returns the data stored in block *id* and its associated validity interval.

Figure 6-1: Block store API

6.1.1 IMPLEMENTATION

The block store server is built using a versioned storage system similar in design to the TxCache cache server. Like the cache, it stores a chain of versions for each block ID. Each version of a block contains the associated data, and is tagged with an associated validity interval; the validity interval's end is null if the version is still current. Unlike the cache, data is stored persistently on disk, and all versions are present.

Figure 6-2 gives a sketch of the implementation of the block store.[1] Starting a transaction assigns a read timestamp (`read_ts`) for the transaction; for a read-only transaction this is the timestamp specified by the TxCache library, whereas for a read/write transaction it is always the latest timestamp. All GET operations in a transaction are performed with respect to this timestamp; they see only data that is valid at this timestamp. A GET operation, therefore, locates and returns the version of

[1]The implementation described in Figure 6-2 is necessarily simplified. Among other things, it does not discuss issues related to on-disk storage, or describe the locking required to allow concurrent processing of requests. Further details about the block store implementation are available in a separate report [79].

the block (if one exists) that was created before the transaction's read timestamp, and was not replaced before the transaction's read timestamp. In a read/write transaction, creating or updating a block with the PUT command creates a new version with the appropriate data, but does not yet fill it in its validity interval or install it into the chain of versions for that block ID.

Our block store ensures serializability using optimistic concurrency control [55]. (This isn't a fundamental choice; it would be equally reasonable to use strict two-phase locking [38] to prevent concurrent updates.) Read-only transactions do not require any concurrency control checks on commit. Committing a read/write transaction first performs a validation phase. If any of the blocks read or written by that transaction have been updated since the transaction started, the transaction is aborted to prevent violating serializability. Otherwise, the transaction is permitted to commit and assigned a timestamp. The server then installs the transaction's changes: it sets the start of the validity interval on the new version of all modified blocks to the transaction's timestamp, and links the version into the chain of previous versions.

Note that this design meets both of TxCache's storage layer requirements. First, it allows the TxCache library to control the timestamp of read-only transactions (as an argument to BEGIN-RO), by using multiversioned storage. Second, it can report the validity interval associated with each query. Because the query model is simple, it is trivial for the system to compute the validity interval of a GET query: it is simply the validity interval of the block version accessed. If, however, that block version is currently valid (*i.e.*, the upper bound of its validity interval is null), the block store returns a validity interval that ends at the latest timestamp, as shown in Figure 6-2. This truncated interval is a conservative one; Chapter 7 describes how to use invalidations to give a better answer.

A final note about the implementation is that, for read-only transactions, the BEGIN-RO and COMMIT operations effectively do nothing at all. They could be eliminated in favor of having clients simply indicate a timestamp as an additional argument to GET operations. However, we use the interface described here for compatibility with the standard transaction interface used by relational databases.

```
// per-session state
timestamp read_ts
set<block> readset
set<block> writeset


begin-ro(timestamp) {
  read_ts = timestamp
}

begin-rw() {
  read_ts = latest_ts
}

get(id) {
  // add to readset
  readset += id

  // walk through the list of versions
  block = latest_version[id]
  while block != null {
    // does this version's validity interval
    // include the timestamp?
    if (read_ts >= block.validity.start &&
        (read_ts < block.validity.end ||
         block.validity.end == null)) {

      if (block.validity.end != null) {
          // use block's validity
          return (block.data, block.validity)
      } else {
          // block is still valid, so bound its
          // validity interval at the latest time
          return (block.data,
                  [block.validity.start, latest_ts])
      }
    }
    block = block.prev_version
  }
  // no acceptable version found
  return not-found
}
```

Figure 6-2: Sketch of the block store implementation

```
put(id, data) {
  block = allocate_new_block()
  block.id = id
  block.data = data
  // will fill in block's validity interval later
  writeset += block
}

commit() {
  // don't need a validation phase
  // for read-only transactions
  if transaction is read-only
    return read_ts

  // OCC validation phase
  for b in (readset + writeset)
     if b modified since transaction start
        abort()

  // assign timestamp to transaction
  ts = increment(latest_ts)

  // install new versions
  for b in writeset
     b.validity.start = ts
     b.validity.end = null
     b.prev_version = latest_version[b.id]
     b.prev_version.validity.end = ts
     latest_version[b.id] = b

  return ts
}
```

Figure 6-2: Sketch of the block store implementation (continued)

6.2 SUPPORTING RELATIONAL DATABASES

A typical deployment of our system will likely use a relational database management system (RDBMS) as its storage layer, as this is the most common storage architecture for web applications. As with the block store, we must ensure that it satisfies TxCache's two storage layer requirements: it must allow control over which timestamp is used to run read-only transactions, and it must report the validity intervals for each query result. However, relational databases have a number of significant differences from the simple block store we previously examined, which present new challenges for meeting these requirements. We list four differences below:

1. Databases typically do not have explicit support for running transactions on any state other than the current one, and providing this requires integrating with existing concurrency control mechanisms

This makes it more challenging to allow the TxCache library control over which timestamp is used to execute read-only queries, compared to the block store, as the database was not designed from the ground up to explicitly provide this functionality.

2. Queries do not return a single object. Instead, they access subsets of relations, specified declaratively – for example, all auctions whose current price is below $10.
3. Queries involve significant processing in the database: data might be obtained using a variety of index access methods and subsequently filtered, transformed, or combined with other records.

These two differences make it more complex to compute the validity intervals for query results. In the block store, which accessed only a single object per query, this task was straightforward; in a database, a single query performs a computation over many different data objects (rows), some of which might not ultimately contribute to the query result.

4. Finally, databases are large, complex, pre-existing pieces of software, unlike the block store, which was designed from scratch

This final difference doesn't present any specific problems but guides our approach: a practical solution to adding caching support must not involve redesigning major components of the RDBMS.

Note that although we discuss these requirements in the context of a relational database, they are not specific to such databases. Some of the challenges described above also arise in real key-value stores, which are typically more complex and full-featured than the simple block store we described previously. For example, many support some form of scanning or searching operation [29, 37].

Although standard databases do not provide the features we need for cache support, we show they can be implemented by reusing the same mechanisms that are used to implement widely-used multiversion concurrency control techniques like snapshot isolation. In this section, we describe how we modified an existing DBMS, PostgreSQL [81], to provide the necessary support. The modifications required are not extensive: our implementation added or modified less than 1000 lines of code. Moreover, they are not PostgreSQL-specific; the approach can be applied to other databases that use multiversion concurrency.

6.2.1 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.

One solution could be to use a *temporal database* [72], which tracks the history of its data and allows "time travel" – running queries against previous database state. Although such databases exist, they are not commonly used, because retaining historical data introduces substantial storage and indexing cost.[2] Therefore, we do

[2]The POSTGRES research database, on which PostgreSQL is based, was one such system [97]. However, time travel support was removed early in transition to an open-source production database, precisely because it was considered too expensive.

not consider this a practical solution.

Furthermore, these systems are overkill: they are designed 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 [10], so many databases already have the infrastructure to support them. Our modifications, then, are intended to expose control over these internal mechanisms to the TxCache library.

Background: Multiversion Concurrency Control in PostgreSQL

Our techniques for adding support for TxCache's requirements to existing databases take advantage of their existing multiversion concurrency control (MVCC) mechanisms. Accordingly, we begin with a primer on how MVCC techniques are implemented. For concreteness, we describe the implementation of MVCC in PostgreSQL, the database we use in our implementation. We emphasize, however, that the techniques we describe are not PostgreSQL-specific but can be applied to other databases that use MVCC. Much of what we describe in this section is true of other MVCC implementations; we indicate some important differences.

Databases using multiversion concurrency control store one or more versions of each tuple. Internally to the storage manager, each transaction has an ID that is used to identify which changes it made. PostgreSQL tags each tuple version with the ID of the transaction that created that version (xmin) and the ID of the transaction that deleted it or replaced it with a new version, if one exists (xmax). When a transaction modifies a tuple (an UPDATE statement), it is treated logically as two operations: deleting the previous tuple version, and inserting the new version. The tag on each tuple version is similar to our use of validity intervals, with one important exception: PostgreSQL's transaction IDs are opaque identifiers rather than timestamps, *i.e.*, their numeric ordering does not correspond to the commit order of transactions. (This is because transaction IDs are assigned when transactions *start*, not when they commit.)

Each query is performed with respect to a *snapshot*, which is represented as a set of transaction IDs whose changes should be visible to the query. When a transaction starts, PostgreSQL *acquires a snapshot* by computing the set of transactions that have already committed, and uses that snapshot for the transaction's queries. During the execution of a transaction, the PostgreSQL storage manager discards any tuples that fail a *visibility check* for that transaction's snapshot, *e.g.*, those that were deleted by a transaction that committed before the snapshot, or created by a transaction not yet committed.

A "vacuum cleaner" process periodically scans the database and removes old tuple versions that are no longer visible to any running transaction.

Pinned Snapshots

TxCache needs the database to be able to run a read-only transaction on a recent state of the data. PostgreSQL has the ability to run transactions with respect to snapshots, so it seems well suited to provide this feature. However, there are two challenges to providing access to past snapshots. First, we need to ensure that the old tuple versions are still present; if they are sufficiently old, the vacuum cleaner process might have removed them. Second, we need to preserve the *metadata* required to reconstruct that state of the database, *i.e.*, the set of transactions that need to appear in the snapshot. PostgreSQL is able to compute the *current* snapshot, *i.e.*, the set of currently committed transactions. But it does not know the *order* in which those transactions committed, so it does not know which transactions should be visible in previous snapshots.

We provided an interface that allows the current snapshot to be saved so that it can be used by subsequent read-only transactions.[3] We added a PIN command that acquires and saves the current snapshot. The PIN command returns an identifier that can be provided as an argument to a later BEGIN SNAPSHOTID command, starting another transaction that sees the same view of the database. The database state for that snapshot remains available at least until it is released by the UNPIN command. A pinned snapshot is identified by the number of read/write transactions

[3]This feature has other uses besides supporting a transactional cache. A "synchronized snapshots" feature that provides essentially the same functionality was independently developed and is scheduled to appear in an upcoming release of PostgreSQL. The semantics are similar, though not exactly the same, to the ones we describe here.

that committed before it, allowing it to be easily ordered with respect to update transactions and other snapshots.

One way to think of this interface is that the PIN command starts a new read-only transaction, but does not associate it with the active connection (as typically happens when starting a transaction). Rather, it gives it a name that allows later read-only queries to be run with the same snapshot, effectively making them part of the same transaction.

Figure 6-3 demonstrates the use of this API. The example consists of five transactions. Transactions 1 and 2 set up a table that contains four values. After transaction 2, the current snapshot is pinned; it is identified by the latest visible transaction's timestamp (2). Transactions 3–5 delete two values from the table and insert one new one. Finally, we run a read-only transaction and specify that it should run at timestamp 2. This transaction sees the state of the table at the time the snapshot was pinned, *i.e.*, it does not see the effects of transactions 3–5.

In PostgreSQL, we implement pinned snapshots by saving a representation of the snapshot (the set of transaction IDs visible to it) in a table, and preventing PostgreSQL's vacuum cleaner process from removing the tuple versions in the snapshot. PostgreSQL is especially well-suited to this modification because of its no-overwrite storage manager [99]. Because stale data is already removed asynchronously, the fact that we keep data around slightly longer has little impact on performance. However, our technique is not PostgreSQL-specific; any database that implements snapshot isolation must keep a similar history of recent database states. For example, the Oracle DBMS stores previous versions of recently-changed data in a separate "rollback segment". The similarity between the requirements for creating a new pinned snapshot and those for starting a new transaction (which runs on a snapshot of the database) suggests that this interface should be straightforward to implement regardless of how the database's storage manager is built.

6.2.2 TRACKING QUERY VALIDITY

TxCache's second requirement for the storage layer is that it must indicate the validity interval for every query result it returns. The TxCache library needs this information to ensure transactional consistency of cached objects. Recall that this interval is

```
-- setup
test=# CREATE TABLE test (x int);
CREATE TABLE

-- timestamp 2
test=# INSERT INTO test VALUES ('1'), ('2'), ('3'), ('4');
INSERT 0 4

-- Pin the current snapshot
-- The snapshot is identified by the timestamp of the latest visible transaction: 2
-- The numbers after '2' are a representation of the database's wall-clock time.
test=# PIN;
PIN 2 1334270999 97317

-- timestamp 3
test=# DELETE FROM test WHERE x=4;
DELETE 1

-- timestamp 4
test=# DELETE FROM test WHERE x=3;
DELETE 1

-- timestamp 5
test=# INSERT INTO test VALUES ('5');
INSERT 0 1

-- This read-only transaction is explicitly specified to run at
-- timestamp 2; therefore, it does not see the changes of transactions 3-5
test=# BEGIN READ ONLY SNAPSHOTID 2;
BEGIN
test=# SELECT * FROM test;
 x
---
 1
 2
 3
 4
(4 rows)
SELECT VALIDITY 2 3

test=# COMMIT;
```

Figure 6-3: Demonstration of pinned snapshots in our modified PostgreSQL

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 of a query always includes the timestamp associated with the transaction in which it ran.

As previously noted, each tuple version in PostgreSQL, and in other multiversion concurrency control databases, is tagged with the ID of the transaction that created it, and the transaction that deleted it or replaced it with a newer version (if any). This effectively serves as a validity interval for that tuple version, although we must translate from PostgreSQL's unordered transaction IDs to timestamps that reflect the commit order of transactions. To accomplish this translation, we modified PostgreSQL to maintain a table in memory that maps transaction IDs to logical timestamps, for transactions that committed after the earliest pinned snapshot.

In the block store, each GET query accessed exactly one block, so the validity interval of the query result is simply the validity interval of the block. In a relational database, however, a SELECT query may access many tuples. We might attempt to compute the validity interval of a query by taking the intersection of the validity intervals of all the tuples accessed during query processing. This approach is overly conservative: query processing often accesses many tuples that do not affect the final result (consider the case of a sequential scan, which reads every tuple in a relation, only to discard the ones that do not match a predicate).

Instead, we compute the validity interval of the tuples that are ultimately returned to the application. We define the *result tuple validity* to be the intersection of the validity times of the tuples returned by the query. To compute it, we propagate the validity intervals throughout query execution, and take the intersection of the validity intervals of the returned tuples. 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. (Aggregate operators require special handling; we discuss them separately later.)

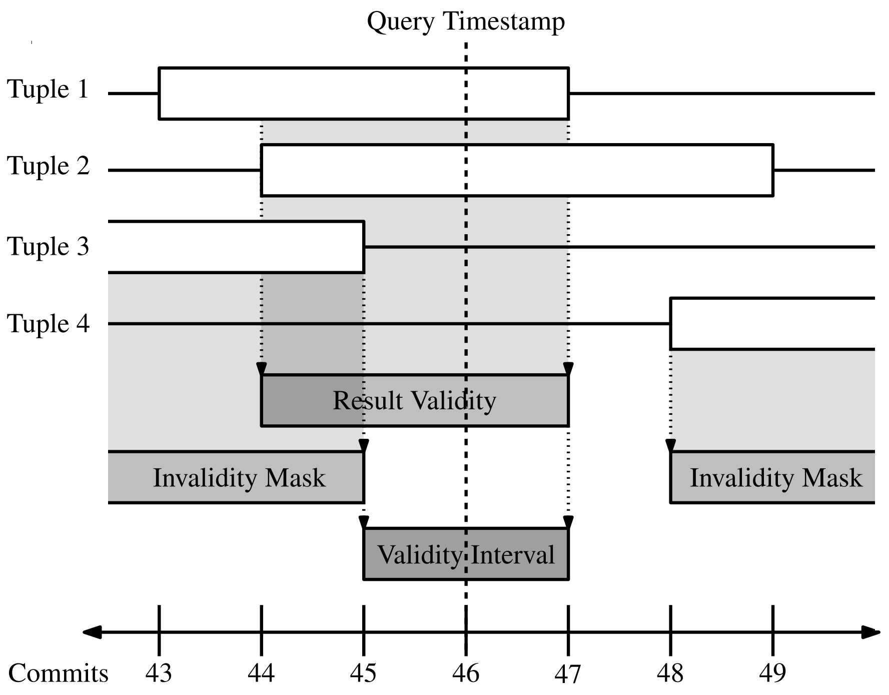


Figure 6-4: Example of tracking the validity interval for a read-only query. All four tuples match the query predicate. Tuples 1 and 2 match the timestamp, so their intervals intersect to form the result validity. Tuples 3 and 4 fail the visibility test, so their intervals join to form the invalidity mask. The final validity interval is the difference between the result validity and the invalidity mask.

Dealing with Phantoms

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, in Figure 6-4, tuples 1 and 2 are returned in the query result; the result tuple validity, therefore, is $[44, 47)$, the intersection of their validity intervals. However, this is not the correct validity interval for the query result. Timestamp 44 should not be included in the interval, because the query results would be different if the query were run at this timestamp. Tuple 3 does not appear in the results because it was deleted before the query timestamp, but would have been visible 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. In Figure 6-4, that includes tuples 3 and 4.

The invalidity mask is a conservative measure. Visibility checks are performed as early as possible in the query plan to avoid processing unnecessary tuples. Some of these tuples might ultimately have been discarded anyway, if they failed the query conditions later in the query plan (perhaps after joining with another table). In these cases, including these tuples in the invalidity mask is unnecessary. While being conservative preserves the correctness of the cached results, it might unnecessarily constrain the validity intervals of cached items, reducing the hit rate. In theory, this problem could be avoided entirely by delaying the visibility check until just before returning the query result, after processing is complete; however, this would be unacceptably expensive because it might involve subjecting tuples to expensive processing (*e.g.*, multiple joins or complex function processing) only to discard the results in the end.

To ameliorate 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 PostgreSQL's usual behavior with respect to sequential scans, where it evaluates the cheaper visibility check first. Delaying the visibility checks improves the quality of the invalidity mask, particularly for sequential scans which must examine every tuple in a relation. It does increase the processing cost when executing queries with complex predicates – which is why standard PostgreSQL performs the validity check first – but the overhead is low for simple predicates, which are most common. In our experiments (Chapter 8), we found that this change did not cause any perceptible decrease in throughput for a web application.

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. An example of this reporting can be seen at the end of Figure 6-3, where the database indicates that the validity interval of the SELECT query is $[2,3)$ by following the query result with the notice "SELECT VALIDITY 2 3".

Aggregates

The approach described above for tracking validity intervals works correctly with all standard classes of query operators except for one: aggregates, such as the COUNT and SUM functions. Aggregates are fundamentally different than other query operators in that they do not commute with the visibility check. That is, we discussed earlier the possibility of delaying the visibility check until after query processing is complete, but this cannot be done with aggregates: it wouldn't be correct to execute a COUNT query by gathering the set of all tuple versions, ignoring validity, counting them, and *then* trying to perform a validity check.

Tracking the validity interval of aggregates requires an extra step, compared to other query operators. With other operators, such as joins or filters, it suffices to set the validity interval of the generated tuple to the intersection of the validity intervals of the tuple or tuple that generated it. For aggregates, we must also add the inverse of that validity interval to the invalidity mask. The reason for this extra step is that an aggregate, such as COUNT, would produce a *different* result tuple at times outside the result's validity interval. (This differs from operators like joins which produce *no* tuple at these other times.) Using the invalidity mask to track this effect is appropriate, as the tuples corresponding to different aggregate results are effectively a different type of phantom.

6.2.3 MANAGING PINNED SNAPSHOTS

The database modifications described in Section 6.2.1 explain how the database implements pinned snapshots. However, the TxCache consistency protocol requires pinned snapshots to be created on a regular basis, and requires application servers to be aware of which snapshots are pinned on the database, and their associated wall-clock timestamps. As described in Section 5.4, the TxCache library needs this to determine which pinned snapshots are within a read-only transaction's application-specified staleness limit.

The block store we described earlier does not require any of this management of pinned snapshots because it provided access to *all* previous versions of data. However, if we wanted the block store to discard sufficiently old data – an important practical concern – it would also need to notify the application servers as to which versions are available.

We employ another module called the *pincushion* to manage the set of pinned snapshots. It notifies the storage layer when to create pinned snapshots, releases them (UNPIN) when they are no longer needed, and notifies the application servers about which snapshots are available. This section describes its operation.

Pincushion

The *pincushion* is a lightweight daemon that tracks and manages the set of pinned snapshots. In theory, this functionality could be incorporated into the DBMS itself. However, we chose to locate it in a separate daemon in order to minimize the changes we had to make to the database, and to reduce the load on the database (which can easily become the bottleneck). This daemon can be run on the database host, on a cache server, or elsewhere; in our experiments, we co-located it with one of the cache servers.

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. It periodically (*e.g.*, every second; the frequency is a configurable system parameter) sends a PIN request to the database server, pinning the current snapshot. It then adds that snapshot, and the corresponding wall-clock time (as reported by the database) to the table.

When the TxCache library on an application node begins processing 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 sends a list of matching snapshots, and flags them as potentially in use, for the duration of the transaction. The TxCache library can also request that the pincushion create a new pinned snapshot; the library would do so if there are no sufficiently fresh pinned snapshots already, *e.g.*, if snapshots are taken every second but the application requests to run a transaction with freshness limit 0.1 seconds.

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.

The pincushion is accessed on every transaction, making it a potential bottleneck for the scalability of the system. However, it performs little computation, making this only an issue in large deployments. For example, in our experiments (Chapter 8), nearly all pincushion requests received a response in under 0.2 ms, approximately the network round-trip time.

Improving Scalability

Given that the pincushion is a centralized service, it could become a bottleneck in large systems. Accordingly, we would like to eliminate the requirement for the pincushion to be accessed on every transaction, in order to improve scalability. We observe that the pincushion serves two separate purposes: it allows transactions to look up which snapshots are pinned, and it tracks which snapshots are in use so that it can unpin old unused snapshots. We address the first goal by disseminating information about pinned snapshots using multicast, and the second using timeouts.

The pincushion continues to be responsible for creating new pinned snapshots periodically. However, rather than having application servers contact the pincushion to learn which snapshots are pinned, we distribute this information using a multicast mechanism. Whenever the pincushion creates a new pinned snapshot, it sends a message containing the snapshot's ID and timestamp to all application servers. Each application server uses this to maintain a local copy of the table of pinned snapshots. Note that this assumes the availability of an application-level multicast system that can distribute messages from the database to all application servers. This is a reasonable assumption as these servers will typically be in the same datacenter; moreover, our invalidation mechanism (Chapter 7) also relies on a multicast mechanism.

Because application servers do not contact the pincushion when they start transactions, the pincushion no longer knows which pinned snapshots are in use. To garbage-collect pinned snapshots, we therefore use a very simple strategy: the pincushion simply removes them after a fixed timeout. When it does, it multicasts a message to the application servers notifying them of this change so they can update their table. Unlike the centralized approach, this timeout strategy makes it possible for a transaction's snapshot to be released before it has finished executing, potentially forcing it to abort because the data is no longer available. However, given a sufficiently long timeout, this possibility is unlikely: most transactions should be short, as long-running transactions are well known to be problematic for many reasons. Long-running transactions can contact the pincushion to request a longer timeout. Note, however, that forcing such transactions to abort may even be desirable in some circumstances, as retaining old versions of data indefinitely is not without cost.

7 AUTOMATIC INVALIDATIONS

Until now, we have assumed that we know the precise upper bound of all objects in the cache. But this assumption is clearly not realistic: many cached objects are still valid, so the upper bound of their validity interval lies in the future and we do not know what it is. Indeed, these cached objects are often the most useful, as they may remain valid well into the future. To handle these objects correctly, we need *invalidations*: notifications from the database to the cache that the data from which a cached object was computed has changed. The cache uses these notifications to keep the validity intervals of cached objects up to date as items change.

Invalidations are challenging. Cached objects can have complex, non-obvious dependencies on multiple data objects from the storage layer. Existing caches like memcached leave the question of which objects to invalidate and when entirely to the application developer. As we argued in Section 3.2, this analysis is challenging for application developers and a frequent source of bugs. We avoid this and support a programming model where application developers simply indicate the functions they would like to cache and the TxCache library handles the rest. Accordingly, our system is designed to provide *automatic invalidations*: it tracks data dependencies and automatically updates the cache when the database is modified.

This chapter describes how the system tracks objects that are currently valid. It introduces three components. First, *invalidation streams* (Section 7.1) give us a way to distribute change notifications to the cache and process them in a way that keeps validity intervals up to date. Second, *invalidation tags* (Section 7.2) allow us to represent the data dependencies of cached objects. Finally, *invalidation locks* (Section 7.3) provide a technique for modifying the storage layer to detect data dependencies and automatically generate invalidations without the application being involved.

7.1 UNBOUNDED INTERVALS AND INVALIDATION STREAMS

How should the cache represent objects that are currently valid? Other objects have *bounded* validity intervals: the upper bound represents a timestamp after which the object may no longer be valid. Currently-valid objects, however, have *unbounded* validity intervals: the precise upper bound is unknown. For these objects, invalidations from the storage layer will allow us to know when they are no longer valid.

When the database executes a query and reports that its validity interval is unbounded, *i.e.*, the query result is still valid, it so indicates by setting a still-valid flag on the output. It also provides a *concrete upper bound*: a timestamp up to which the result is already known to be valid. The timestamp of the latest transaction committed on the database can serve as this bound (more precisely, the latest transaction committed before query execution *started*, as there could be concurrent changes). We write an unbounded validity interval as, for example, $[12, 42+)$. Here, the concrete upper bound of 42 indicates that the object is valid until at least timestamp 42, and the $+$ indicates that it may also be valid beyond that time.

When the application performs multiple database queries that are combined to produce a single cacheable object, the TxCache library combines their validity intervals by taking their intersection. Computing intersections involving unbounded intervals follows the obvious rules: the upper bound of the intersection is the smaller of the concrete upper bounds of the two intervals, and the still-valid flag is cleared unless it was set on both input intervals. Thus, for example,

$$[12,42+) \cap [15,37+) = [15,37+) \qquad \text{and} \qquad [12,42+) \cap [15,48) = [15,42)$$

7.1.1 INVALIDATION STREAMS

When the storage layer indicates that a query result has an unbounded validity interval, it assumes responsibility for providing an invalidation when the result may have changed. It does so by distributing an *invalidation stream* that identifies which cached objects may no longer be valid.

The invalidation stream is an ordered sequence of messages generated by the storage layer. Each message contains a timestamp indicating the most recently committed read/write transaction at the time that message was sent, and a list of identifiers indicating which cached objects are no longer valid. (Sections 7.2 and 7.3 explain precisely *how* invalidations identify the affected objects.) If a long time passes without any read/write transactions committing, the database also sends an empty invalidation message simply to indicate that the latest timestamp is still current. This stream is distributed to all cache nodes using a reliable multicast mechanism.

When a cache node receives an invalidation message, it truncates the validity intervals of any affected cached objects, setting their upper bound to the timestamp included in the invalidation. This reflects the fact that the corresponding change in the storage layer may have invalidated the affected object. The cache treats all objects that are still valid as though the upper bound of their validity interval is the timestamp of the latest invalidation received. Thus, receiving an invalidation message effectively acts to extend the validity intervals of objects that are still valid, while truncating those that are no longer valid. This is demonstrated in Figure 7-1.

Including a timestamp invalidation stream message ensures that invalidations are processed by cache nodes in the correct order. The concrete upper bounds used when adding new objects to the cache further ensure that INSERT operations are ordered with respect to invalidations. This ordering resolves a potential race condition: a cached object with unbounded interval might arrive at the cache node *after* its invalidation. Note that the window for this race condition is wider than one might expect. A cacheable object might depend on significant application-level computation or multiple queries, and the object is not added to the cache until this is complete, whereas the invalidation might happen any time after the first query.

When an invalidation message arrives before the object it is meant to invalidate, however, the concrete upper bound on the inserted object will be earlier than the

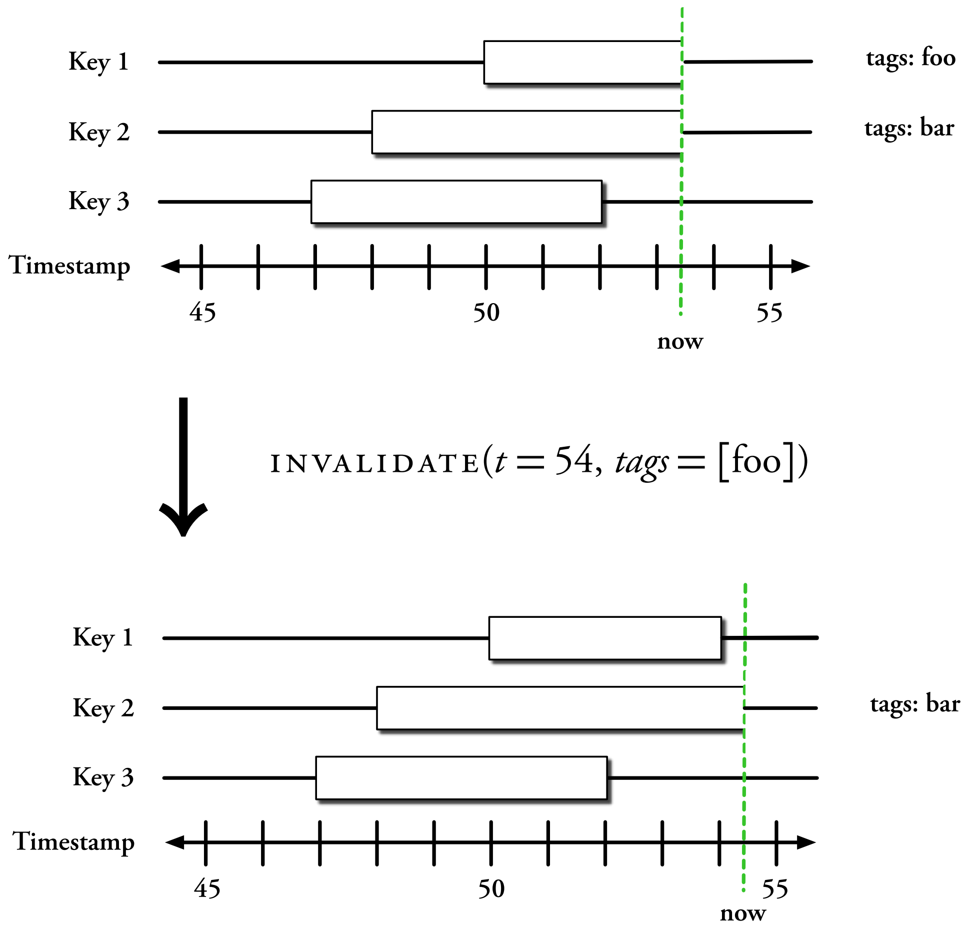


Figure 7-1: Example of cache invalidation processing. Initially, the latest invalidation received by the cache server has timestamp 53, so the two objects in the cache with unbounded intervals are known to be valid at least through time 53. After the cache receives an invalidation for timestamp 54, the validity interval of the object affected by the invalidation has its upper bound set to that timestamp, while the validity interval of the other object is extended as it is now known to remain valid at least through time 54.

timestamp on the latest invalidation message. To handle this case, cache servers keep a small history of recently-received invalidation messages so they can check whether a newly-arrived object has already been invalidated. It isn't a serious problem if this history is insufficient – as might happen if an object took an unusually long time to compute – because every cached object has a guaranteed upper bound: the concrete upper bound. The cache server simply clears the still-valid flag, truncating its validity interval at this bound.

7.1.2 DISCUSSION: EXPLICIT INVALIDATIONS

We have not yet addressed the issue of how to detect and track data dependencies so the system can automatically generate invalidations. We do, however, have enough of a framework in place that we could easily implement explicit invalidations: the application can indicate to the database during each read/write transaction which cached objects are affected, and the database can place that information into the invalidation stream. (It isn't clear how, in our cacheable-function programming model, the application would identify cached objects; this approach would be better suited for a system where applications manage the cache themselves.)

Although this approach doesn't address the major usability issue we identified – the need for explicit invalidations – the ordered invalidation stream does address an important concern for real systems. Specifically, it solves the race condition issue mentioned above, where the invalidation arrives before one of the objects it is supposed to invalidated. This race condition was a concern for MediaWiki developers, who opted not to use memcached to cache negative lookup results – the fact that an article *doesn't* exist. If a user concurrently created an article, and one of these negative results was added to the cache just after the invalidation, the negative result might never be invalidated, causing the article text to effectively be lost. This situation can't arise in our system: the timestamps on the new cached object and the invalidation messages reveal the ordering.

A key design decision here is that in our system, all invalidations come from the database server, which generates the invalidation stream. It can therefore guarantee that the invalidation stream is ordered according to the transaction commit order. In other systems, applications typically send updates directly to the cache servers; this avoids the need for the database to be involved in cache invalidations, but makes it difficult or impossible to achieve reliable ordering of updates and invalidations.

7.2 INVALIDATION TAGS

Invalidation streams alone are enough to allow us to implement explicit invalidations, but this solution isn't sufficient. We want to generate invalidations automatically, which means that we need to be able to represent the data dependencies of a cached object. We track data dependencies using *invalidation tags* which can be attached to objects in the cache. These tags are identifiers representing a logical set of data items. As a simple example, one might use an invalidation tag for each relation in the database. Each currently-valid cached object has a *basis*: a set of invalidation tags representing the data used to compute that object, *e.g.*, the relations accessed when computing the cacheable function.

Having this information for each cached object allows us to send invalidations in terms of invalidation tags, *i.e.*, in terms of which data objects were modified, rather than which cached objects were affected. This is a fundamentally important difference. Identifying the data objects modified by a transaction can be done using *local* analysis. Taking, for example, a system that uses database relation IDs as invalidation tags, it is easy to identify the relations affected by an update transaction: the database can track this information dynamically, or it could be determined using a relatively simple static analysis of queries. Without invalidation tags, determining which cached objects might be affected by an update requires a *global* analysis of the program to determine what data it might be caching.

Every object stored in the cache that has an unbounded validity interval must also have a corresponding basis so that the cache knows when to invalidate it.[1] The TxCache library includes this information as an extra argument in a STORE request, as indicated in Figure 7-2. Invalidation tags are not meaningful for objects that already have bounded validity intervals, *i.e.*, those that are not currently valid. The

[1]The one exception is a cached object that depends on no database state, *i.e.*, a constant computation. Such an object effectively has an infinite validity interval. This rare corner case is not very interesting.

- STORE(*key, value, validity-interval,* ***basis***) : Add a new entry to the cache, indexed by the specified key and validity interval. If the validity interval is unbounded, the basis is a set of invalidation tags reflecting the object's data dependencies. It is not used for objects with bounded validity intervals.
- INVALIDATE(*timestamp, tags*) : Update the validity interval for any items with the specified invalidation tags in their basis, truncating it at the given *timestamp*.

Figure 7-2: Cache interface with invalidation support. This is an extension to the basic interface in Figure 5-3.

TxCache cache server maintains a tree mapping invalidation tags to the objects whose basis contains them. When it receives an invalidation message from the invalidation stream – consisting of a timestamp and a list of the invalidation tags affected – the cache server looks up the affected objects, and updates their validity interval. It also clears their basis, as this information is only needed for objects that are still valid.

7.2.1 TAG HIERARCHY

At what granularity should we track data dependencies? Above, we considered using invalidation tags to track the database *relations* that a cached object may have been derived from. Although it is certainly possible to capture data dependencies at this level, it is clearly overly conservative. In our auction site example, a query that looks up the current price for a particular auction will find itself with a data dependency on the entire AUCTIONS table. As a result, a modification to *any other* item in the same table will cause the cached object to be invalidated, which would have a serious detrimental effect on the cache hit rate.[2]

We might instead choose to represent data dependencies at a fine granularity. Taking this approach to its extreme, we could use individual database *tuples* as the invalidation tags. In comparison to relation-level tags, this approach has different

[2]The perils of overly course granularity in invalidations are well known in other systems. For example, the MySQL database includes a simple query cache. It uses relation-level invalidations, which are sufficiently restrictive that the conventional wisdom is to turn the cache off entirely for tables that see even a moderate number of updates [94].

problems. It prevents the problem described above where an update to a single row unnecessarily invalidates a cached result that depends on a different row. However, some queries may access many tuples. A query that accesses every row in a large table (*e.g.*, to compute aggregate statistics) could easily depend on many thousands of rows; the resulting basis would be too costly to store or transmit. Similarly, a modification that updates every row in a large table (*e.g.*, to add a new column as part of a schema change) could generate invalidations for thousands of rows that would have to be processed separately. A second problem with tuple-granularity invalidation tags is that they fail entirely in the presence of phantoms, which we discuss in Section 7.3.

The tension between overly-coarse relation-granularity tags and overly-fine tuple-granularity tags indicates that a single granularity is unlikely to fit all use cases. We address this by supporting invalidation tags with *multiple granularities*. This provides the benefits of fine-grained dependency tracking for objects with a small set of dependencies, but makes it possible to fall back on coarse-grained (*e.g.*, relation-level) dependency tracking when the costs of doing otherwise would be prohibitive.

To support multiple granularities, we organize invalidation tags into a hierarchy. A coarse-granularity tag is considered a *supertag* of all the finer-granularity *subtags* it subsumes. For example, in a system with a two-level hierarchy consisting of relation and tuple tags, the tag identifying tuple 127 of the USERS table would be a subtag of the tag identifying the entire USERS table. To make the relationship clear, we would write this tag as USERS:127, with the colon denoting the subtag relationship. A more complex system might use more levels in the hierarchy; we discuss one with four levels in Section 7.3. It is always safe (conservative) to *promote* a tag in a cached object's basis, replacing it with its supertag. This can be used to reduce the size of the object's basis by coalescing multiple fine-granularity tags into a coarser-granularity tag, at the cost of a potential loss in precision.

Invalidations can also be issued at multiple granularities. For example, a single update transaction might modify a small set of tuples, or modify an entire relation. Representing the latter as a single invalidation tag reduces the cost both of distributing the invalidation message and of identifying matching tuples.

In this hierarchical invalidation tag scheme, cache servers need to process invalidations according to the following rule:

- an invalidation message containing invalidation tag x affects any cached object whose basis contains x, a supertag of x, *or* a subtag of x

This rule appears somewhat surprising at first glance, but an example makes it clear. An invalidation message for tuple 127 of the USERS table clearly affects any object whose basis contains the exact tag USERS:127. It also affects any object that depends on the entire USERS table, *i.e.*, any supertag of the specified tag. Similarly, if an invalidation message for the relation-level tag USERS arrives, it must also invalidate any object depending on a subtag like USERS:127, as invalidating the entire relation necessarily affects any objects depending on part of the table.

7.3 INVALIDATION LOCKS

The final piece of the invalidation puzzle is how to automatically determine the invalidation tags to assign to a cached object, and which invalidation tags to invalidate as part of a read/write transaction. The use of invalidation tags makes this a question of what data is accessed or modified by the transaction – a question that the database is in a prime position to answer.

A related question is exactly what semantics we ought to ascribe to particular invalidation tags. The hierarchical tag structure described above explains how to process tags with multiple granularities, but it does not specify a particular meaning for each level of the hierarchy. This question is more subtle than it might appear. Above, we considered relation- and tuple-granularity tags. But tuple-granularity tags are, in fact, problematic for a reason more fundamental than their fine granularity: they do not accurately capture read dependencies for relational databases. In the relational model, queries access subsets of the data identified by predicates, *e.g.*, the set of users whose name is Alice. Expressing this subset in terms of a set of tuple-level invalidation tags fails to capture conflicts with newly inserted tuples: creating a new user named Alice ought to cause the previous result to be invalidated, but would not. This is another instance of the "phantom" problem in database concurrency control [38].

Accurately representing the read and write dependencies of operations in a relational database presents quite a challenge. Besides the phantom issue discussed above, an effective solution should be able to handle complex queries implemented using a variety of indexes and query operators: what invalidation tags would we assign, for example, to a cached object containing the list of top ten employees by salary?

Our insight is that tracking these read and write dependencies is precisely the problem solved by the database's concurrency control mechanism. If the database supports serializable isolation of concurrent transactions, it must have a mechanism for tracking which objects are read or modified by a transaction. This is necessary regardless of how the database implements serializability, as determining which transactions conflict with each other is a fundamental requirement [38]. Moreover, the database's mechanism must be capable of dealing with phantoms, and must be accurate even for complex queries, as these are important requirements for the correctness of the database system.

This section presents a general method for adapting existing concurrency control mechanisms in a database (or other storage system) to determine which invalidation tags to use or invalidate.

Invalidation Locks. When a read-only transaction accesses the database, it acquires a new type of lock, an *invalidation lock*. It acquires these locks on the same data items the system would normally acquire read locks[3] on for concurrency control. The presence of one of these locks indicates that some cached object may depend on the locked data, and we may later need to send an invalidation if it changes. Invalidation locks differ from normal read locks in two important respects. First, they are not released when a transaction commits. Second, and most importantly, invalidation locks do not block conflicting write operations; rather, a conflict simply indicates the need to send an invalidation. This makes "lock" somewhat of a misnomer; it might be more accurate to refer to them as flags, but we use the term "lock" to make the connection to the database's lock manager clear.[4]

The set of invalidation locks acquired during execution of a query determines the

[3]For clarity, we describe this technique in terms of a database that uses locking for concurrency control, but this isn't a requirement.

[4]This abuse of nomenclature is not unique to us. Serializable Snapshot Isolation, for example, uses non-blocking "SIREAD locks" to track read dependencies [19].

$$\underbrace{\textsc{rubis}}_{\text{database}} : \underbrace{\textsc{users}}_{\text{relation}} : \underbrace{\textsc{firstname}/\text{page-36}}_{\text{index page}} : \underbrace{\textsc{alice}}_{\text{index entry}}$$

Figure 7-3: A four-level invalidation tag in PostgreSQL's tag hierarchy

set of invalidation tags that must be applied to any cached objects that depend on it. If the database already has a hierarchical mechanism it uses internally for tracking multi-granularity read dependencies (as most common databases do), we use the same hierarchy for our invalidation locks. The canonical example of such a hierarchy would be the lock hierarchy in a system using multi-granularity locking [40]. However, it is not specific to two-phase locking; other concurrency control approaches such as optimistic concurrency control [2, 55] or serialization graph testing [19, 22] frequently also use a multi-granularity mechanism for tracking read dependencies.

For concreteness, consider the tag structure that is used in PostgreSQL, the database on which we base our implementation. (Note, by the way, that PostgreSQL is an example of a database that does not use locking; it uses Serializable Snapshot Isolation [19, 80], a form of serialization graph testing.) It tracks read dependencies using a four-level hierarchy. The coarsest granularity is an entire database; the next coarsest is entire relations within that database. Within a relation, finer-grained dependencies are expressed in terms of *index entries*: the next granularity is pages of a particular index, containing subtags for individual entries on that page. Using index entries makes it possible to avoid problems with phantoms: whereas a tuple-granularity tag could not capture the predicate "users whose name is Alice", a tag representing the NAME=ALICE entry in an appropriate index does. This is a standard technique known as next-key locking [68]. Figure 7-3 shows an example of a four-level tag.

As read/write transactions acquire locks on data they modify, they may encounter conflicts with invalidation locks. Unlike normal read locks, the presence of an invalidation lock does not block the writer. Rather, the lock conflict "breaks" the invalidation lock and issues an invalidation: the invalidation lock is removed, and the corresponding invalidation tag is added to the invalidation stream. More precisely, the invalidation lock is not removed and the invalidation is not issued until the read/write transaction commits; this accounts for the possibility that the transaction might abort.

The invalidation lock approach offers several benefits:

- it doesn't require careful thought about exactly how to represent data dependencies; it reuses the mechanisms from database concurrency control

- it can handle all types of queries. In particular, using index page locking, it can represent a range query such as finding all users with ID between 19 and 37. In an earlier version of this work [78], we used a simpler scheme that could not handle range queries without the use of coarse relation-granularity locks.

- it only sends invalidations for data that have invalidation locks set. Therefore, it can avoid sending invalidations for data that is not in the cache, or that has already been invalidated.

7.3.1 STORING INVALIDATION LOCKS

The challenge with invalidation locks is how to maintain the set of invalidation locks that have been acquired. Because there will be a lock on every piece of data used by any cached object, the set of invalidation locks is likely to be large. It isn't unrealistic to imagine that there might be one lock for every row in the database (or potentially even more, as there can be locks for multiple index entries). This means that the usual technique of maintaining an in-memory lock table is unlikely to scale.

Fortunately, invalidation locks differ from regular locks in some ways that can simplify the task of representing them. First, unlike normal locks, it does not matter which transaction placed an invalidation lock on a particular object; all that matters is whether any such lock exists. It is also never necessary to have multiple invalidation locks on the same object at the same time; it is only necessary to send one invalidation for a particular tag no matter how many cached objects might depend on it.

These simplifying assumptions can be used to reduce the footprint required to store invalidation locks. All that is needed is a single bit for each lockable object representing whether that object currently has an invalidation lock set, *i.e.*, a single bit per relation, index page, or index entry. This bit could be represented simply by setting a flag in the row/page header. For a database where the entire data set fits in RAM, there is very little cost to maintaining this information as it does not have to be written to disk. This use case is actually a common one; in-memory databases are increasingly common today [71, 98] and are especially appealing for the sorts of high-performance web applications that our system targets.

For databases that don't fit entirely in memory, we could still use the same approach of storing a flag in the row header to indicate whether an invalidation lock is set. However, this would mean that performing a read-only query might require writing to the disk to set this flag, which would be undesirable. Instead, we use another simplifying assumption about invalidation locks: it is always safe to assume that there is an invalidation lock set on an object, as this would at worst cause a spurious but harmless invalidation to be sent. Accordingly, we track invalidation locks for any objects (pages or index entries) that are currently available in the buffer cache, but do not make any effort to persist the invalidation locks to disk if the pages need to be swapped out to disk; instead, we simply make the conservative assumption when loading a page from disk that all of the entries on it have invalidation locks set. This does mean some unnecessary invalidations will be sent, but the number of these should be low. Most deployments of our system will strive to have as much data stored in the cache as possible. The benefit of suppressing invalidations for data that is not cached comes primarily from objects that are updated multiple times in quick succession, and these will likely be still resident in the buffer cache.

8 EVALUATION

This chapter analyzes the effectiveness of TxCache using the RUBiS web application benchmark. The evaluation seeks to answer the following questions:

- What changes are required to modify an existing web application to use TxCache? (Section 8.2).
- How much does caching improve the application's performance, and how does the cache size affect the benefit? (Section 8.4)
- Does allowing transactions to see slightly stale data improve performance? (Section 8.5)
- How much of a performance penalty does TxCache's guarantee of transactional consistency impose? (Section 8.6)
- How much overhead is caused by TxCache's modifications to the database? (Section 8.7)

8.1 IMPLEMENTATION NOTES

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.

Cache Server. The cache server implements the versioned cache described in Section 5.2. It provides a versioned hash table interface, and maintains an index from invalidation tags to cached objects necessary to process invalidation messages. The server accepts requests from the TxCache library over a simple TCP-based RPC protocol. It is single-threaded, allowing it to avoid locking. This contributes to the simplicity of the server: our implementation only requires about 2500 lines of C code.

Our cache server is not optimized for maximum throughput, as cache throughput was not a bottleneck in any of our experiments. Several performance optimizations have been developed for memcached that could be applied to the TxCache cache server. For example, a custom slab allocator can be used to reduce the cost of allocating cached objects (our implementation uses the standard system `malloc`). Using a UDP-based RPC protocol can also improve performance by reducing memory overhead and avoiding contended locks in the kernel [90].

Database Modifications. We modified PostgreSQL 8.2.11 to provide the necessary support for use with TxCache: pinned snapshots, validity interval computation, and invalidations. This version of PostgreSQL provides snapshot isolation as its highest isolation level rather than serializability, so we use this configuration in our experiments.

Our implementation generates invalidations using a simplified version of the protocol in Chapter 7. It does not attempt to store invalidation locks; it simply generates all appropriate invalidations for every update. Our implementation tracks invalidation tags using two granularities: relation-level tags and index-entry tags. Because this version of PostgreSQL does not incorporate index-range locking, fine-granularity invalidation tags cannot be used for range queries; this limitation was not an issue for the queries used in our benchmark application. Subsequently, we incorporated index-range locking into PostgreSQL as part of a new serializable isolation level [80].

The modifications required to add TxCache support to PostgreSQL were not extensive; we added or modified about a thousand lines of code to implement pinned snapshots and validity interval tracking, and another thousand lines to implement invalidations.

The modified database interacts with the pincushion, which creates new pinned snapshots and notifies application servers about them. We also use a separate daemon to receive invalidation notices from the database and distribute them to cache nodes. It currently sends unicast messages to each cache server rather than using a more sophisticated application-level multicast mechanism.

Client Library. The TxCache library is divided into two parts: a language-independent component that accesses the cache and ensures consistency, and a set of language-specific bindings. The language-independent part uses the lazy timestamp selection protocol of Section 5.4.2 to ensure consistency between objects read from the cache and those obtained from the database. It also mediates interactions between the application and the database: applications must use the TxCache library's interface to send queries to the database. This interface always passes queries directly to the database (it does not try to parse or otherwise interpret them), but the library may also insert commands to start a transaction at an appropriate snapshot, and monitors the validity intervals and invalidation tags associated with each query result.

PHP Bindings. The RUBiS benchmark is implemented in PHP, so we provided a set of PHP language bindings for TxCache. These bindings provide the support for applications to designate cacheable functions, and marshal and unmarshal objects when they are stored in or retrieved from the cache.

We previously presented the TxCache interface in terms of a MAKE-CACHEABLE higher-order procedure. PHP lacks support for higher-order procedures, so the syntax is a bit more awkward. Instead, the PHP library provides a `txcache_call` function that takes as arguments the name of a function to call and the arguments to pass to it. This function checks for an appropriate result in the cache, and, if none is found, invokes the function. Figure 8-1 demonstrates a typical use: users invoke a `getItem` function which is actually a wrapper function that calls `txcache_call`.

```
function getItemImpl($itemId)
{
  $res = sql_query("SELECT * FROM users ...");
   ⟪ some computation on query result ⟫
  $user =  ⟪ result ⟫
  return $user;
}
function getItem($itemId)
{
  global $TX; // this contains the cache configuration
  return txcache_call($TX, 'getItemImpl', $itemId);
}
```

Figure 8-1: Example of a cacheable function in PHP

8.2 PORTING THE RUBIS BENCHMARK

RUBiS [7] 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.

Determinism. Cacheable functions must be deterministic and depend only on database state. Generally, identifying such functions is easy; indeed, nearly every read-only function in the system has this property. However, while designating functions as cacheable, we discovered that one of RUBiS's actions inadvertently made a nondeterministic SQL query. Namely, one of the search results pages divides results into pages using a "SELECT ... LIMIT 20 OFFSET *n*" statement, *i.e.*, returning only the n through $n+20$th results. However, it does not enforce an ordering on the items it returned (*i.e.*, it lacks an ORDER BY clause) so the database is free to return *any* 20 results. This caused search results to be divided into pages in an unpredictable and inconsistent way. This turned out to be a known bug in the PHP implementation, which was not written by the original authors of RUBiS.

TxCache generally leaves determining which functions are cacheable up to the application developer. However, it incorporates a simple sanity check that was sufficient to detect this error. The cache issued a warning when it detected an attempt to insert two versions of the same object that had overlapping validity intervals, but different values. This warning indicated a possible non-determinism.

Optimizations. 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 that performs a search and displays lists of matching items. Originally, this function performed a join in the database, returning the list of matching items and their description. We modified it to instead perform a database query that obtains the list of matching items, then obtain details about each item by calling our `getItem` cacheable function. This allowed the individual item descriptions to be cached separately.

Several other optimizations were not related to caching, but simply eliminated other bottlenecks with the benchmark workload. For example, we 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, to the point where the system spent the majority of this time processing this query. We addressed this by adding a new table and index containing each item's category and region IDs. We also removed a few queries that were simply redundant.

8.3 EXPERIMENTAL SETUP

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. The emulator repeatedly selects the next interaction for each client by following a transition matrix and each client's interactions are separated by a randomly-chosen "think time". We used the standard RUBiS "bidding" workload, a mix of 85% read-only interactions (browsing) and 15% read/write interactions (placing bids). The think time between interactions was chosen from a negative exponential distribution with 7-second mean, the standard for this benchmark (as well as the TPC-W benchmark [102]).

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 (using `mod_fcgi` to invoke the PHP interpreter), 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.

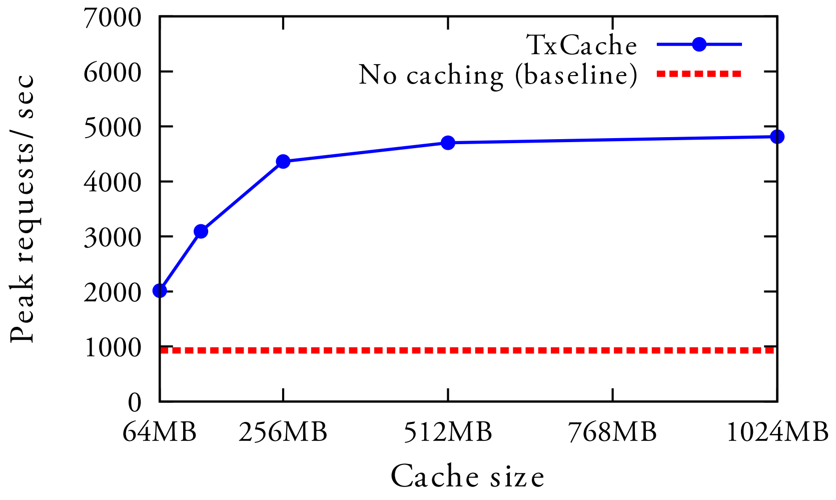
For the in-memory configuration, we used seven hosts as web servers, and two as dedicated cache nodes. For the larger configuration, eight hosts ran both a web server and a cache server, in order to make a larger cache available.

8.4 CACHE SIZES AND PERFORMANCE

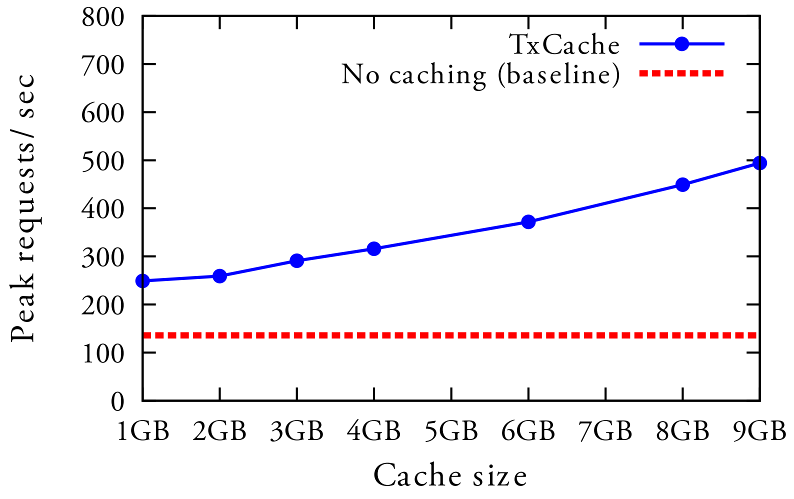
We evaluated RUBiS's performance in terms of the peak throughput achieved (requests handled per second) as we varied the number of emulated clients. Our baseline measurement evaluates RUBiS running directly on an unmodified PostgreSQL database, without TxCache. This achieved a peak throughput of 928 req/s with the in-memory configuration and 136 req/s with the disk-bound configuration.

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 8-2. 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. This throughput improvement comes primarily from reduced load on the database server.

The RUBiS PHP benchmark does not perform significant application-level computation. Even so, we see a 15% reduction in total web server CPU usage. We could expect a greater reduction in application server load on a more realistic workload than this simple benchmark, as it would likely perform more application-level computation, *e.g.*, to do more sophisticated formatting of results. Furthermore, our measurement of web server CPU usage takes into account not just time spent on legitimate application processing but also the time the PHP interpreter spends parsing source code on each request.



(a) In-memory database



(b) Disk-bound database

Figure 8-2: Effect of cache size on peak throughput (30 second staleness limit)

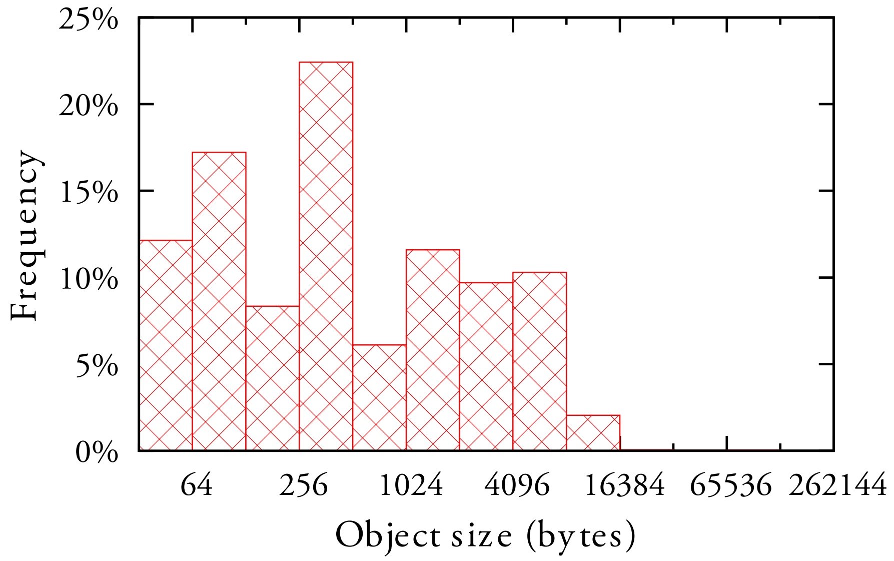
Cache server load is low. Even on the in-memory workload, where the cache load is spread across only two servers, each cache server's load remained below 60% utilization of a single CPU core. Most of this CPU overhead in kernel time, suggesting inefficiencies in the kernel's TCP stack as the cause; switching to a UDP protocol might alleviate some of this overhead [90]. The average cache lookup latency is 0.7 ms, significantly lower than even the in-memory database's average query latency of 5.5 ms (and even before the database reaches its capacity). There are an average of 2.4 invalidation tags per cached object, and the tags themselves average 16 bytes.

8.4.1 CACHE CONTENTS AND HIT RATE

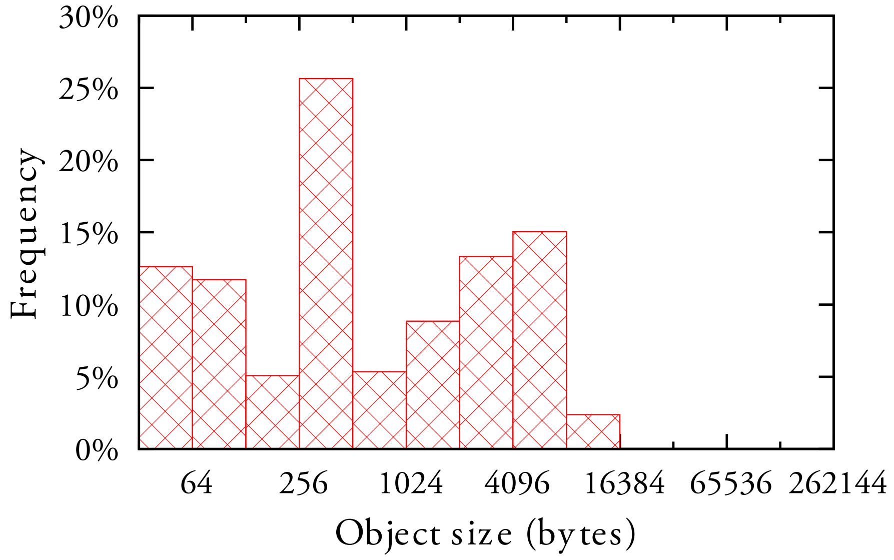
We analyzed the contents of the cache (after the warmup period) for both the in-memory and disk-bound configurations. As shown in Table 8-3, the cache contains many items – over 500,000 for the in-memory configuration and over 4.5 million for the disk-bound configuration. Nearly all of the cache server's memory (over 95%) goes to storing the cached values themselves; another 2% goes to storing the names (keys) for these values. Validity intervals are small (9 bytes per object), so storing them does not require much space. Invalidation tags are slightly larger (16 bytes, on average) and more numerous (an average of 2.4 tags per cache entry). However, the cost of storing invalidation tags is still small: about 2% of memory.

Figure 8-4 shows the distribution of the sizes of cached objects. Half of the cached objects are smaller than 400 bytes; these objects typically represent the results of a single database query, such as looking up the name of a user. Most of the rest of the objects are under 8 KB in size; these objects are typically generated HTML fragments or the results of more complex queries. There are a handful of objects as large as 100 KB, *e.g.*, the detailed bid history of especially popular items, but there are sufficiently few of them that they do not make up a significant fraction of the cache workload.

Figure 8-5(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. On the in-memory workload, the cache hit rate directly translates into an overall performance improvement because each cache hit represents load removed from the database – and a single cache hit might eliminate the need to



(a) In-memory database



(b) Disk-bound database

Figure 8-4: Distribution of cache object sizes. Note the logarithmic x-axis.

	In-memory workload		**Disk-bound workload**	
Cache entries	560,756		4,506,747	
Total cache size	817.9 MB		7.820 GB	
Cached values	778.5 MB	(95.2%)	7.477 GB	(95.6%)
Cache keys	18.1 MB	(2.2%)	145 MB	(1.8%)
Validity intervals	4.9 MB	(0.6%)	40 MB	(0.5%)
Invalidation tags	16.4 MB	(2.0%)	166 MB	(2.1%)

Table 8-3: Cache contents breakdown

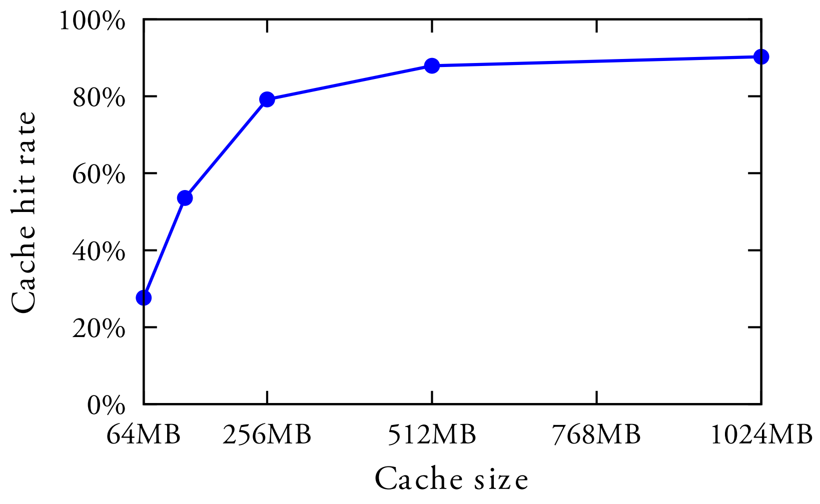
perform several queries.

Interestingly, we always see a high hit rate on the disk-bound database (Figure 8-5(b)), but this hit rate does not always translate into a large performance improvement. This illustrates an important point about application-level caching: the value of each cached object is not equal, so hit rates alone rarely tell the whole story. This workload exhibits some very frequent queries, such as looking up a user's nickname by ID. These can be stored in even a small cache, and contribute heavily to the cache hit rate. However, they are also likely to remain in the database's buffer cache, limiting the benefit of caching them. This workload also has a large number of data items that are individually accessed rarely (*e.g.*, the full bid history for auctions that have already ended). The latter category of objects collectively makes up the bottleneck in this workload, as the database must perform random I/O to load the necessary information. The speedup of the cache is primarily determined by how much of this data it can hold.

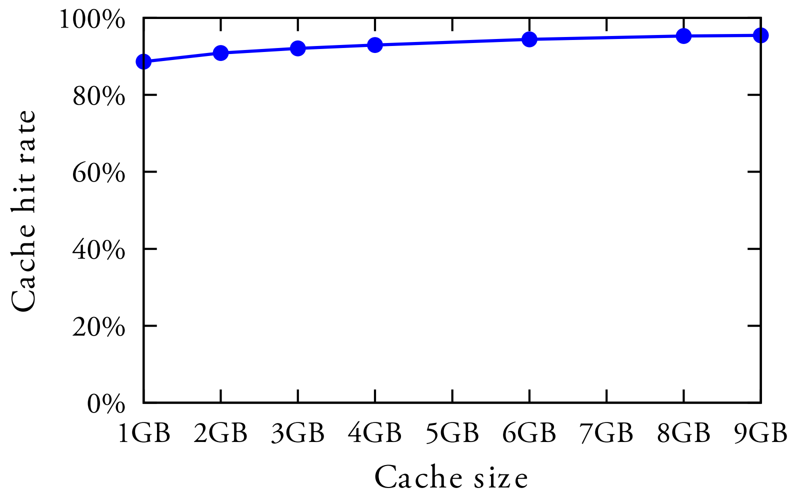
8.5 THE BENEFIT OF STALENESS

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 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 8-6 compares the peak throughput obtained by running transactions with



(a) In-memory database



(b) Disk-bound database

Figure 8-5: Effect of cache size on cache hit rate (30 second staleness limit)

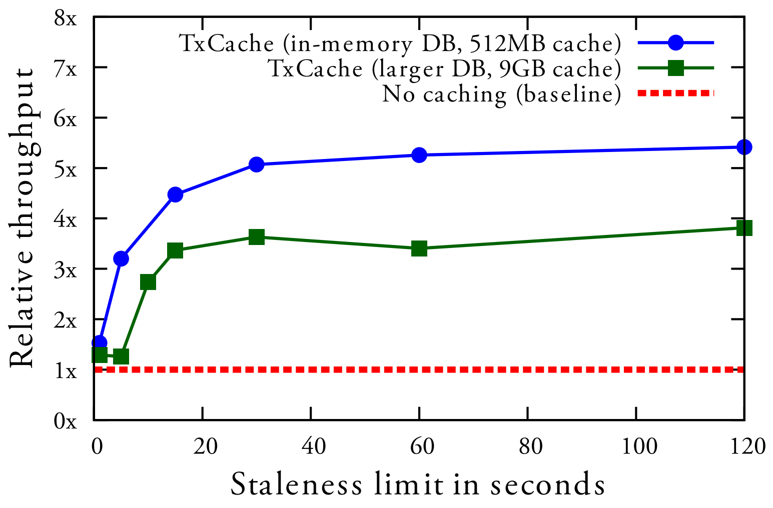


Figure 8-6: Impact of staleness limit on peak throughput

staleness limits from 1 to 120 seconds. Even a small staleness limit of 5-10 seconds provides a significant benefit. The RUBiS workload includes some objects that are expensive to compute and have many data dependencies: for example, indexes of all items in a particular category along with their current prices. These objects are invalidated frequently, but the staleness limit permits them to be used. The benefit begins to diminish at around a staleness limit of 30 seconds. This reflects the fact that the bulk of the data either changes infrequently (such as information about inactive users or auctions), or is modified frequently but also accessed multiple times every 30 seconds (such as the aforementioned index pages).

8.6 THE COST OF CONSISTENCY

A natural question is how TxCache's guarantee of transactional consistency affects its performance. We explore this question in two ways: we examine cache statistics to analyze the causes of cache misses, and we compare TxCache's performance against a non-transactional cache.

8.6.1 CACHE MISS ANALYSIS

We classified cache misses into four types, inspired by the common classification for CPU cache misses [48]:

- *compulsory miss*: the object was never in the cache, *i.e.*, the cacheable function was never previously called with the same arguments
- *staleness miss*: the object has been invalidated, and is sufficiently old that it exceeds the application's staleness limit
- *capacity miss*: the object was previously evicted from the cache
- *consistency miss*: some sufficiently fresh version of the object was available, but it was inconsistent with previous data read by the transaction

The last category, consistency misses, are precisely the category of interest to us, as these are the cache misses that would be caused by TxCache but would not occur in a non-transactional cache.

The TxCache cache server records statistics about the cache hit rate and cache misses of various types. Figure 8-7 shows the breakdown of misses by type for four different configurations of the RUBiS workload. Our cache server cannot distinguish staleness and capacity misses, because it discards the associated validity information when removing an item from the cache either due to staleness or cache eviction.

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 validity intervals that overlap with the transaction's request, either because the items in the cache change rarely or the cache contains multiple versions. Workloads with higher staleness limits experience a higher proportion of consistency misses (but fewer overall misses) because they have more stale data that must be matched 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.

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

Table 8-7: Breakdown of cache misses by type. Figures are percentage of total misses.

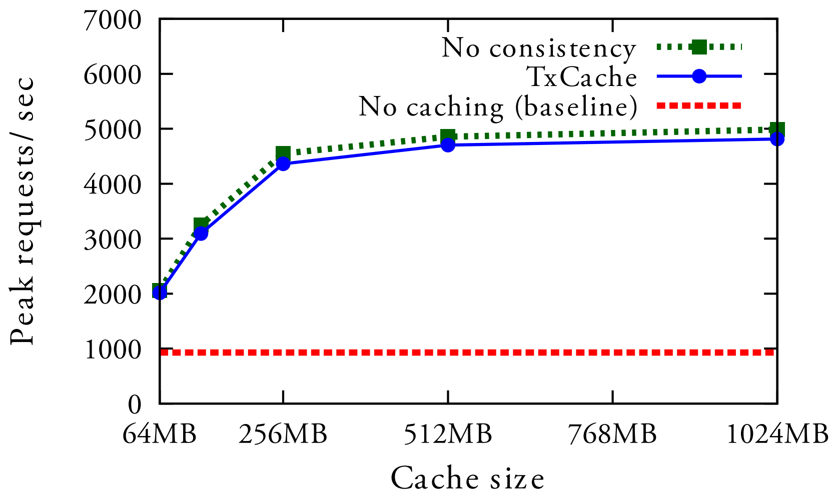


Figure 8-8: Comparison of TxCache and a non-transactional cache for varying cache sizes; in-memory workload

8.6.2 EXPERIMENTAL COMPARISON

The low fraction of consistency misses suggests that providing consistency has little performance cost. We verified this experimentally by comparing against a non-transactional version of our cache. That is, we modified our cache to continue to use our invalidation mechanism, but configured the TxCache library to blithely ignoring consistency: it accepted any data that was valid within the last 30 seconds, rather than the usual protocol that requires the validity intervals to intersect.

The results of this comparison are shown in Figure 8-8, which reproduces the experiment in 8-2(a): it measures RUBiS system throughput for varying cache sizes on the in-memory configuration. Now, however, the non-transactional version of our cache is shown as the "No consistency" line. As predicted, the benefit that this configuration provides over consistency is small. On the disk-bound configuration, the results could not be distinguished within experimental error.

8.7 DATABASE OVERHEAD

The TxCache system requires some modifications to the database, to compute validity intervals, provide access to recent snapshots, and generate invalidations. The experiments described earlier compare the performance of RUBiS running with TxCache to a baseline configuration running directly on a unmodified PostgreSQL database. Not surprisingly, the benefit of caching exceeds the additional overhead incurred by the database modifications. Nevertheless, we would like to know how much overhead these modifications cause.

To answer this question, we first performed the baseline RUBiS performance experiment (without TxCache) with both a stock copy of PostgreSQL and our modified version. We found no observable difference between the two cases: our modifications had 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.

Because our end-to-end web application benchmark saw negligible overhead, we performed a database-level benchmark to more precisely measure the cost of these modifications. We used the `pgbench` benchmark [76], a simple transaction-processing benchmark loosely based on TPC-B [101]. This benchmark runs transactions directly on the database server; there is no cache or application server involved. The modified database, however, still computes validity intervals and generates invalidation messages (which are sent to a "multicast" daemon that simply discards them, since there are no cache servers). A simulated pincushion contacted the database every second to obtain a new pinned snapshot, then released after 1–180 seconds.

We ran these benchmarks on a different system configuration than the RUBiS benchmark. We used a 2.83 GHz Core 2 Quad Q9550 system with 8 GB RAM running Ubuntu 11.10. The most significant difference is that the database was not stored on disk at all; rather, it was stored on an in-memory file system (`tmpfs`). As a result, the workload is purely CPU-bound – even more so than the in-memory RUBiS configurations, where modifications and write-ahead log updates still had to be written to disk. This is a "worst-case" configuration intended to expose the overhead of our modifications. In particular, it stresses the invalidation mechanism, because this in-memory configuration can support an unusually high update rate because updates do not need to be synchronously written to disk.

Even on this worst-case, in-memory configuration, the overhead of TxCache's database modifications is low. Figure 8-9 compares the performance of our modified database to stock PostgreSQL 8.2.11. The modifications impose an overhead of less than 7% for computing validity intervals and generating invalidations. The amount of overhead also depends on the age of the pinned snapshots that are being held, as the system must retain all tuple versions that are newer than the earliest pinned snapshot. This makes query processing slightly more expensive, because the system must examine these versions and decide which ones should be visible to a running transaction. However, the performance impact is not large: the cost of retaining an extra 180 seconds of state is under 3%, even with the unusually high update rate of this benchmark.

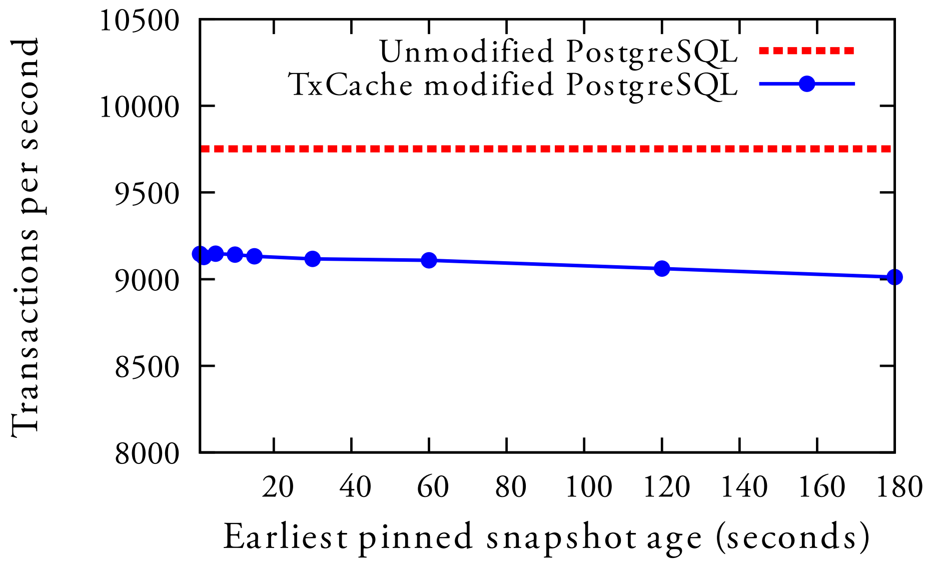


Figure 8-9: `pgbench` transactions per second for modified and unmodified PostgreSQL

9 RELATED WORK

Many different forms of caching and replication have been proposed to improve the throughput of high-performance web applications. Since there are so many such systems (both research and practical) to make a comprehensive list impractical, this section discusses some of the systems most closely related to TxCache.

We discuss two broad classes of related systems: application-level caches and cache management systems for them (Section 9.1), and database-level caches and replication systems (Section 9.2). We also discuss other systems that have proposed using multiversion storage and relaxing freshness guarantees (Section 9.3).

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.

9.1.1 WEB CACHES

Dynamic web caches operate at the highest layer, storing entire web pages produced by the application. Typically, they require cached pages to be regenerated in entirety when any content changes. Such caches have been widely used since the Web became popular (when content was mostly static) and remain in common use today.

When used for dynamic content, caches need to invalidate pages when the underlying data changes. Typically, this is done in one of three ways. First, timeouts can be used to expire cached data periodically. Second, the cache can require the application to explicitly invalidate pages when they are modified [107]. Finally, the system can track data dependencies to identify which objects need to be invalidated when the underlying database is modified. Challenger et al. [25, 26] describe a web cache that models the dependencies between cacheable objects and underlying data objects as a graph; this system requires the application developer to provide this object dependency graph. Cachuma [109] groups dynamic pages into classes by their URL, and requires the developer to specify data dependencies for each class. TxCache also uses invalidations, but integrates with the database to automatically identify dependencies.

As we noted in the introduction, full-page caching is becoming less appealing to application developers as more of the web becomes personalized and dynamic. When a different version of a page is served to each user, caching them has little value.

9.1.2 APPLICATION DATA CACHES

As a consequence of increased personalization and dynamic content, web developers are increasingly turning to application-level data caches. 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. TxCache falls into this category, as do many existing systems.

We have already discussed memcached [65], which provides the abstraction of a distributed hash table into which the application can store arbitrary binary objects using a GET/PUT/DELETE interface. The hash-table interface, where objects are identified by application-specified keys, is a fairly standard one among other caches, though they often differ in terms of what types of objects they can store. Other application-level caches often integrate into a specific language runtime, *e.g.*, caches that cache Java [9, 35, 49] or .NET [69, 92] objects. Redis [84] (which can be used either as a cache or as a persistent store) can cache a variety of data structures such as strings, lists, and sets, and provides operations to manipulate them such as set intersections and unions. In all cases, the hash-table interface requires application developers to choose keys and correctly invalidate objects, which we argued can be a source of unnecessary complexity and application 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 caches do support transactions within the cache. JBoss Cache [49], for example, can use either locking or optimistic concurrency control to allow transactions to update multiple cached objects atomically. Windows Server AppFabric [92] (formerly known as Velocity) provides locking functions that can be used to implement similar functionality. Other caches provide single-object atomic operations; for example, memcached has a compare-and-set operation that acts on a single key. However, even the caches that support transactions differ from TxCache in that they only support atomic operations *within* the cache, whereas TxCache provides transactions that operate across both the cache and storage layer.

Some caches provide support for replicated data [49, 69, 92]. Though replication is a useful technique for ensuring the availability of data despite node failures, this usually isn't the reason that these systems support it. Rather, replication is used for performance: making multiple copies of frequently accessed objects allows multiple servers (possibly located in different data centers) to handle requests for that object. This technique is suitable for objects that are accessed frequently but infrequently modified. Our cache does not use replication, but it could easily be extended to do so, as discussed in Section 3.3.1.

We noted in Section 4.2 that it is impossible to guarantee that the cache always reflects the latest state of the backing store unless using an atomic commitment protocol between the cache and the storage layer, which would be prohibitively expensive. Ehcache [35], a Java object cache, does allow the cache to participate in a two-phase commit protocol, allowing the cache to be updated atomically with the database. However, this is not intended to ensure serializability, as this cache supports only the weaker READ COMMITTED isolation level. It is instead intended for durability (the cache stores data on disk, and therefore can recover from crashes).

The interface of these application-level caches has some similarity to distributed hash tables (DHTs) from peer-to-peer systems research [82, 89, 96, 108]. Indeed, some similar techniques have been used in their implementation, such as the use of consistent hashing [52] for data partitioning. However, peer-to-peer DHTs are designed to scale to millions of nodes and use sophisticated routing protocols to identify the node responsible for storing a particular object without full knowledge of the system membership. Our cache assumes that the system is small enough that each node can maintain the entire system membership, as in the OneHop DHT [42]. DHTs are also typically intended for persistent storage, so must address issues of data durability [30, 95] that our cache can ignore.

9.1.3 CACHE MANAGEMENT

Several others have observed the problems that explicit cache management poses for developers of applications that use application-level caching, and proposed systems to address these challenges. Typically, these systems take the form of a library that mediates interactions with an existing application-layer cache such as memcached.

One approach is to have the application (or a library in the application) update or invalidate the cache. For example, Cache-Money [18] is a library for the Ruby on Rails object-relational mapping system that implements a write-through cache: the library updates the cache before making any changes to the database. This library restricts the application to caching only a specific set of objects, corresponding to database tuples and indexes; it does not allow arbitrary application computations to be cached, or even more sophisticated database queries such as joins. Wasik [105] proposed a system for managing consistency in an application that uses memcached. This system analyzes the application code and rewrites it to insert the appropriate invalidations for each database modification; however, the analysis requires programmers to indicate what data in the database each cached object depends on. Because this class of solutions modifies the application to provide dependencies, it assumes that only one application modifies the database. This can be a serious restriction in environments where multiple applications share a database, as changes made by other applications will not trigger the necessary cache invalidations. In practice, even systems where a database is used primarily to support a single application may still have other tools modifying the database, *e.g.*, to support bulk-loading new data or to allow administrators to manually correct data errors.

A second approach is to involve the database, using triggers to invalidate cached objects after each update. This approach avoids the aforementioned problem with multi-application databases, as *any* modification will invoke the trigger. However, the database must be told which triggers to create and which cached objects they should invalidate. Application developers can indicate these dependencies manually [46], though this approach could be error-prone. CacheGenie [44,45] is a system that automatically creates database triggers for cache invalidation. This system integrates with an object-relational mapping framework (Django) and allows caching application objects corresponding to a fixed set of common query patterns.

Several of these cache management systems rely on object-relational mapping (ORM) frameworks like Ruby on Rails, Django, or Java Entity Beans [18,44,45,74]. These frameworks provide persistent objects for an application (written in an object-oriented language) by mapping the objects to a corresponding schema in a relational database, and automatically generating queries to access it. Caching systems for these frameworks typically cache fixed categories of objects defined by the ORM framework; often, these objects have a one-to-one correspondence with database tuples. TxCache uses a more general programming model, caching the results of calls to cacheable functions. This allows TxCache to cache arbitrary application computations, unlike ORM-based caches. However, because these ORM-based caches have well-defined classes of cached objects, they can update cached objects in place rather than invalidating and recomputing them, as discussed in Section 3.2.3.

Most of the systems described above do not provide transaction isolation, *i.e.*, they do not ensure the consistency of cached data read during a transaction. One exception is SI-Cache [74], which provides snapshot isolation in a cache of Java Entity Beans. CacheGenie [45] also describes an extension that provides serializability by using strict two-phase locking in both the cache and the database.

9.2 DATABASE CACHING AND REPLICATION

Another popular approach to improving web application performance is to deploy a caching or replication system within the database layer, or between the database and the application. These systems address only database performance; unlike application-level caches, they offer no benefit for application server load. However, they require no modifications to the application.

Query caches are one way to address load in the database layer. These add an additional layer in front of the database that caches the results of recent queries. As a simple example, the MySQL database itself includes a query cache that can avoid the need to process queries that are character-for-character identical to previous ones; it invalidates a cache entry whenever one of the relations it depended on is modified. A more sophisticated example is the ABR query cache [33] which uses a dependency graph to determine which values need to be invalidated. Ferdinand [39] is a query cache that distributes invalidations using a publish-subscribe multicast system; such a strategy could be applied to improve the performance of TxCache. None of these query caches support serializablility for transactions that make multiple queries.

Middle-tier caches also sit in front of the database, but perform more complex processing. DBProxy [5], for example, caches the results to previous queries, like a query cache. However, it incorporates a query processor that allows it to also answer other queries that can be computed from a subset of the cached results. DBCache [16] and MTCache [58] both run a full DBMS at the cache node, populated with full or partial tables from the underlying database; the cache determines which queries can be executed locally and which must be executed by the backing database. These systems also do not attempt to guarantee serializability for transactions that make multiple queries.

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 updates it as the base tables change. Most work on materialized views seeks to incrementally update the view rather than recompute it in its entirety [43]. 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 [21].

A different approach to improving database performance is to replicate the database. Of the many replication approaches that exist, we focus on those that provide transactional consistency semantics. (Others offer weaker guarantees, such as eventual consistency [34, 75].) Most replication schemes used in practice take a primary copy approach, where all modifications are processed at a master and shipped to slave replicas. This replication can be performed synchronously to improve durability, but is often performed asynchronously for performance reasons. One way to do implement primary-copy replication is to transfer a copy of the master database's write-ahead log records to the slaves and replay them there; this "log shipping" approach is widely used in commercial products. Another way is to use a middleware component that executes all update transactions on each replica in the same order [24, 103]. In either case, each replica then maintains a complete, if slightly stale, copy of the primary database.

Several systems defer update processing to improve performance for applications that can tolerate limited amounts of staleness. In FAS [87], clients send their queries to a "scheduler" proxy. This scheduler executes read/write transactions on the primary, then lazily propagates them to the other replicas. In the meantime, it keeps track of how far each replica's state lags behind the primary, and routes read-only transactions to an appropriate replica based on an application-specified staleness limit. Ganymed [77] takes a similar approach, but relies on each replica database to provide snapshot isolation so that slave replicas can process read-only transactions from clients concurrently with updates that are being propagated from the master.

9.2.1 CACHING VS. REPLICATION

Caching and replication are closely related subjects. Like TxCache, both FAS and Ganymed use the idea of allowing read-only transactions to see slightly stale data in order to improve their performance. However, TxCache's design differs significantly from these two systems. One might wonder, then, if techniques like those used in FAS and Ganymed could be applied in TxCache's context of application-level caching.

These database replication systems take a *proactive* approach: after an update to the master database, they update the other replicas' state. Such an approach is well-suited for data replication. After processing a read/write transaction, the set of objects that need to be updated on each replica is easy to identify: it is exactly the same set of tuples modified on the master. This approach can be applied to application-level caching where each application object is a representation of a particular database tuple [88]. However, the approach isn't suitable for TxCache's cache of arbitrary application computations. The set of objects that would need to be updated is large (there are many possible cacheable function calls), and recomputing them is expensive.

Therefore, TxCache must take a *reactive* approach: it opportunistically caches the results of previous computations. Because these objects were computed at different times, TxCache uses validity intervals to ensure that the application sees a transactionally-consistent view of the system, even while reading objects that were not generated at the same time. This mechanism isn't necessary in database replication systems that proactively update the replicas, as each replica has a complete, consistent copy of the entire database at any moment.

9.3 RELAXING FRESHNESS

TxCache builds on a long history of previous work on multiversion concurrency control. Many different approaches to using multiple versions to ensure isolation exist [2, 10, 12, 85]. The most prevalent such approach in production systems today is snapshot isolation [10]. In snapshot isolation, all data read by a transaction comes from a snapshot of the database taken at the time the transaction started. Snapshot isolation permits some "write-skew" anomalies that would not occur in serializable executions. TxCache provides the same guarantee as snapshot isolation: within a transaction, applications see state corresponding to a snapshot of the database, even though some of it may come from the cache. However, TxCache does not introduce write skew anomalies into applications, because it does not use the cache for read/write transactions.

Unlike snapshot isolation, where transactions use a snapshot current as of the time they started, TxCache can assign transactions a snapshot slightly earlier than their actual start time. A similar technique, called Generalized Snapshot Isolation [36] was proposed for use in a replicated database system. Liskov and Rodrigues [61] also proposed running transactions on past snapshots to improve the performance of a distributed transactional file system. Our work is inspired by this proposal. The proposed file system uses a backing store similar to the block store we described in Chapter 6. Because our system supports caching application-level objects derived from database queries, rather than file system blocks, it requires new mechanisms such as dependency tracking (invalidation tags) and validity interval computation mechanisms suitable for use in a relational database. The transactional file system also assumes an environment where each client has its own local cache; TxCache has a different model consisting of a single shared distributed cache.

Bernstein et al. defined a notion of relaxed-currency serializability [11] that encompasses TxCache's use of stale data, and relies on a similar notion of validity intervals. They assume a model similar to that of the replicated database systems previously discussed, in which validity intervals are more readily available. Our contributions include a technique for easily generating validity intervals using existing database concurrency control mechanisms, and using them to generate validity information for application-level objects.

As we argued in Section 4.2, assigning transactions an earlier timestamp is safe as long as it does not permit any causality anomalies. Other systems also allow operations to be reordered when there are no causal dependencies. Lamport introduced the happens-before relation, wherein transactions are considered concurrent if they do not depend on each other [57]. This idea was notably used in the ISIS system [15], whose virtual synchrony communication model enforced ordering constraints only on causally-dependent messages.

10
CONCLUSION AND FUTURE WORK

Application-level caching is a powerful technique for improving the performance of web applications. These caches store higher-level objects produced using a combination of database queries and application-level computation. Accordingly, these caches have the flexibility to be used in many different applications, from caching the results of individual database queries to entire generated web pages – and, importantly, various stages of processing in between. Application-level caches can therefore address two types of performance bottlenecks in web applications. They can reduce the load on the database layer – often dramatically, as caching complex objects can eliminate the need for multiple database queries. They can also reduce the load on application servers, which database-level caching and replication systems cannot address.

10.1 CONTRIBUTIONS

In this thesis, we have introduced a new application-level cache, TxCache. TxCache is able to cache the results of arbitrary application computations, retaining the flexibility described above. TxCache improves on existing application-level caches in three ways.

First, TxCache provides a simple programming model: cacheable functions. Rather than requiring application developers to explicitly identify and access data in the cache – the standard interface for most application-level caches – TxCache simply asks application developers to identify functions whose results should be cached. The TxCache system then automatically caches the results of that function. The application developer is no longer responsible for naming or invalidating cached data, both of which we have seen to be a source of application bugs. TxCache implements its programming model using a new cache library. It avoids the need for explicit invalidations by using invalidation tags and invalidation locks, new mechanisms for detecting and tracking data dependencies.

Second, TxCache provides support for transactions. It guarantees that all data seen by the application within a transaction reflects a consistent view of the database state, whether it is accessed directly from the database or through cached objects. This guarantee can prevent exposing anomalous data to the user, allowing application-level caching to be used by applications that require strict isolation between transactions – a class of applications that were previously unable to use application-level caches. Even for other applications, transaction isolation can simplify development. TxCache ensures transactional consistency by tracking the validity interval of every object in the cache and database.

Finally, TxCache provides a consistency model where transactions can see slightly stale data, subject to application-specified freshness requirements. However, it still guarantees that each transaction sees a consistent view of the data. As we have seen, when applications can tolerate even a few seconds of stale data, it improves the effectiveness of the cache significantly by allowing objects that are frequently modified to remain useful. TxCache chooses which snapshot to use for a given transaction using a lazy timestamp selection protocol that takes into account which data versions are available in the cache.

To our knowledge, TxCache is the first system to provide either automatic cache management or whole-system transaction support in a cache capable of storing arbitrary application-computed values.

10.2 FUTURE WORK

Scalability is a major challenge for developers building web sites that handle requests from millions of users. Application-level caches are one of many techniques that are used to address this challenge. Deploying a system like TxCache at large scale – say, one comparable to Facebook's memcached deployment with thousands of cache servers and a database partitioned over thousands of cache nodes – presents new challenges.

The most likely barrier to scalability in our design is the invalidation system. Our invalidation mechanism requires the storage layer to broadcast the invalidation stream to all cache nodes. Each cache node, therefore, receives a list of invalidation tags for every update processed by the storage layer. In a large deployment, the cost of distributing and processing this stream might prove to be excessive: each cache server must process every invalidation message even if only to determine that it has no objects affected by the invalidation.

One possible approach to improving the scalability of the invalidation mechanism is to use a publish/subscribe multicast system. Such a system could allow invalidation messages to be sent only to relevant cache nodes. This approach has previously been used to efficiently distribute invalidations in a scalable query cache system [39]. A challenge to this approach is that cached objects are distributed among cache nodes without any locality. That is, the cache objects that depend on a particular database object are not likely to be stored on the same cache server. Indeed, if many cached objects depend on a particular database object, they might be distributed across most of the cache servers; any invalidations for the associated invalidation tag would need to be distributed to all of them.

Thinking more generally about scalability, developments in scalable storage systems could have an impact on application-level caching. Classically, the backing store for web applications has been a relational database. In database-backed applications, one common use for application-level caches like memcached is to store database tuples, as the lightweight in-memory cache can provide faster responses and greater scalability than the database. A recent trend is that developers are increasingly turning to storage systems such as partitioned in-memory databases [98] or key-value stores [29, 32, 37, 66, 84]. Given that these storage systems are designed to provide similar scalability and latency advantages, will application-level caching continue to be useful? We believe that it will, as the ability to cache intermediate results of application-level computations can substantially reduce the load on application servers. Indeed, application-level caching might prove even more useful in applications that use a key-value store as their persistent storage, as they will need to do more of their processing in the application layer.

10.3 CONCLUSION

TxCache is an application-level cache designed to be easy to integrate into new or existing applications. Towards this end, it provides higher-level abstractions designed to simplify application development: the cacheable-function programming model simplifies cache management, and the cache's transaction support simplifies reasoning about concurrency. In this respect, TxCache differs from typical application-level caches that optimize for a simple cache implementation but require increased complexity in the applications that use the cache. Our experiments with the RUBiS benchmark show that TxCache is effective at improving the performance of a web application, and that transactional consistency and invalidation support impose only a minor performance penalty – suggesting that providing higher-level cache abstractions does not have to come with a performance cost.

A

CACHING FOR READ/WRITE TRANSACTIONS

The system we have presented so far does not use the cache at all for read/write transactions. Instead, they access data only through the database; this approach allows the database's concurrency control mechanism to ensure the serializability of these transactions. This approach is reasonable for application workloads that have a low fraction of read/write transactions. However, it would be desirable to allow read/write transactions to access cached data.

A.1 CHALLENGES

TxCache's consistency protocol for read-only transactions ensures that they see a consistent view of data that reflects the database's state at some time within the application's staleness limit. This guarantee isn't sufficient to ensure the serializability of read/write transactions – even with a staleness limit of zero. The problem is that *write skew* anomalies can occur: two concurrent transactions that update different values might each read the old version of the object that the other is updating.

Furthermore, read/write transactions must be able to see the effects of their own uncommitted modifications, even in cached objects. For example, the read/write transaction that places a new bid might update the auction's price, then read the auction's data in order to display the result to the user. In doing so, it should not access a cached version that does not reflect the user's new bid. At the same time, the changes made by a read/write transaction should not be visible to *other* transactions until it commits.

A.2 APPROACH

Here, we describe an approach that allows read/write transactions to use cached data. This approach differs from the one for read-only transactions in three ways:

- A read/write transaction can only see cached data that is still current, avoiding the aforementioned write-skew anomalies.
- A read/write transaction cannot use cached objects that are impacted by the transaction's own modifications, ensuring that the transaction always sees the effects of its own changes
- A read/write transaction cannot add new data to the cache, preventing other transactions from seeing objects that contain uncommitted data

We require read/write transactions to see data that is still valid in order to prevent anomalies that could occur if they saw stale data. The first step towards achieving this is to have the TxCache library request only objects from the cache server that have unbounded validity intervals. However, this only ensures that the objects were valid *at the time of the cache access*; they might subsequently be invalidated while the read/write transaction is executing.

To ensure that the cached data seen by a read/write transaction remains current, we take an optimistic approach. We modify the database to perform a validation phase before committing the transaction. In this validation phase, the database aborts the transaction if any of the data it accessed has been modified. To do so, the database must know what data the application accessed through the cache. Invalidation tags, which track the data dependencies of cached objects, make this possible. The TxCache library keeps track of the invalidation tags of all the cached objects it has accessed during a read/write transaction, and provides these to the database when attempting to commit a transaction. The database keeps a history of recent invalidations. If any of the tags indicated in the commit request have been invalidated by a concurrent transaction, it aborts the read/write transaction,

We must also ensure that read/write transactions see the effects of their own changes. Again, invalidation tags make this possible, as they allow the TxCache library to determine whether a cached object was affected by a read/write transaction's database modifications. During a read/write transaction, the TxCache library tracks the invalidation tracks the transaction's writeset: the set of invalidation tags that will be invalidated by the transaction's changes. (Note that this requires the database to notify the read/write transaction of which tags it is invalidating, whereas previously we only needed to send this list of tags to the caches via the invalidation stream.) When the library retrieves an object from the cache, it also obtains the object's basis – the set of invalidation tags reflecting its data dependencies. If the object's basis contains one of the invalidation tags in the transaction's writeset, then the application must not use this object. In this case, the TxCache library rejects the cache object, treating it as a cache miss and recomputing the cached object. However, unlike a cache miss in a read-only transaction, the TxCache library does not insert the results of the computation into the cache, as these results reflect uncommitted data and should not yet be made visible to other transactions.

B

FROM CONSISTENT READS TO SERIALIZABILITY

The protocol presented in Chapter 5 uses validity intervals to provide a *consistent reads* property: all data seen by a transaction reflects a consistent state of the storage layer, regardless of whether that data was obtained from the cache or from the database. The properties we ultimately want are system-wide transaction isolation guarantees. This appendix explains how the consistent reads property interacts with the storage layer's concurrency control mechanisms to provide either serializability or snapshot isolation, depending on which is supported by the storage layer.

This appendix begins by reviewing the consistent reads property that the consistency protocol provides (Section B.1). We next consider snapshot isolation databases, and show that the consistent reads property ensures snapshot isolation (Section B.2.) Subsequently, we consider serializable databases (Section B.3). We first show that TxCache provides serializability for databases that provide a *commitment ordering* property, namely that the order in which transactions commit matches the apparent serial order of execution. We then consider databases that provide serializability *without* this stronger requirement, and explain how to extend the system to support these databases.

B.1 CONSISTENT READS

TxCache ensures that for every read-only transaction, there is a timestamp t such that the validity intervals of each object read by the transaction contains t. As a result, the consistent reads property of validity intervals (Section 5.1.1) means that the transaction sees the effects of all read/write transactions that committed before time t (and no later transactions), *i.e.*, the transaction sees the same data it would if its reads were executed on the storage layer at time t.

Recall also that read/write transactions do not use the cache, so their reads are sent directly to the database. (We do not consider here the extensions in Appendix A that allow read/write transactions to use the cache.)

B.2 SNAPSHOT ISOLATION DATABASES

The property described above is similar to the snapshot isolation level provided by many multiversion databases, such as Oracle and PostgreSQL. This isolation level is defined as follows:

- **snapshot isolation**: a transaction always reads data from a snapshot of committed data valid as of its *start-timestamp*, which may be any time before the transaction's first read; updates by other transactions active after the start-timestamp are not visible to the transaction. When a transaction is ready to commit, it is assigned a *commit-timestamp* larger than any existing start-timestamp or commit-timestamp, and allowed to commit only if no concurrent transaction modified the same data.

(This definition is based on the one given by Berenson et al. [10]; Adya [1] provides an implementation-independent definition in terms of proscribed phenomena in the serialization graph.)

When used with a snapshot isolation database, TxCache provides snapshot isolation for all transactions. Read/write transactions certainly have snapshot isolation, because they bypass the cache and execute directly on the database using its normal concurrency control mechanisms. Read-only transactions can use cached data, but

TxCache's consistent reads property ensures that transactions are assigned a timestamp and only see the effects of transactions that committed before that timestamp.

TxCache differs from snapshot isolation databases in that it allows read-only transactions to see a consistent state of the database from *before* the transaction started (subject to the application-specified freshness requirement). That is, a read-only transaction's start-timestamp might be lower than the latest commit-timestamp. Our definition of snapshot isolation allows this, though not all definitions do. The original definition [10] is ambiguous; Adya's definition [1] explicitly permits the system to use an earlier start time; Elnikety et al. [36] refer to this property as *generalized* snapshot isolation, distinguishing it from conventional snapshot isolation. Note, however, that TxCache only provides this relaxed freshness guarantee for read-only transactions; read/write transactions still execute with snapshot isolation under either definition.

B.3 SERIALIZABILITY

The standard criterion for correct execution of concurrent transactions is serializability:

- **serializability**: the execution of concurrent transactions produces the same effect as some serial execution of the same transactions, *i.e.*, some execution where only one transaction executes at a time

This definition is implementation-independent; there are many possible ways to implement serializability. For example, two-phase locking [38], optimistic concurrency control [55], timestamp ordering [85], serialization graph testing [22], and several other techniques can provide serializability.

Many (though not all) systems that provide serializability also provide the following stronger property:

- **commitment ordering**[1]: the execution of concurrent transactions produces the same effect as a serial execution of the transactions *in the order they committed*

[1]The *commitment ordering* terminology is due to Raz [83]; the same property has also been referred to as *dynamic atomicity* [106] and *strong recoverability* [17].

If the database provides serializability with this commitment ordering property, then TxCache's validity-interval-based consistency protocol is sufficient to provide serializability (Section B.3.1). If the database does not provide this property, then we require some additional support from the database to provide serializability, which might require read-only transactions to sometimes block (Section B.3.2).

B.3.1 SERIALIZABLE DATABASES WITH THE COMMITMENT ORDERING PROPERTY

Two of the most common concurrency control techniques provide this property. Strict two-phase locking ensures that if two concurrent transactions attempt to make conflicting accesses to the same object, the second access is blocked until the first transaction completes, thereby ensuring that if one transaction precedes another in the serial order it also precedes it in the commit order. Optimistic concurrency control achieves this property by preventing transactions from committing if they conflict with a previously-committed transaction.

When used with a database that provides serializability and the commitment ordering property, TxCache guarantees serializability, even for transactions that use cached data. Read/write transactions bypass the cache, so the database's concurrency control mechanism ensures that there is a corresponding serial order – and that this serial order matches the order in which the transactions committed. TxCache's consistent reads property ensures that each read-only transaction sees the effects of all transactions that committed before that transaction's timestamp. This set of transactions is a prefix of the serial order, so the read-only transaction can be serialized at that point in the serial order.

The idea of using snapshot reads for read-only transactions while using a standard concurrency control mechanisms for read/write transactions is not a new one. The first instance appears to be a database system from Prime Computer that uses two-phase locking for read/write transactions, but uses multiversion storage to ensure that read-only transactions see a consistent snapshot [27, 28]. Bernstein and Goodman later generalized this technique as the "multiversion mixed method" [13]. A similar technique can be achieved with many databases today by running read/write transactions with two-phase locking but running read-only transactions under snapshot isolation.

B.3.2 SERIALIZABLE DATABASES WITHOUT THE COMMITMENT ORDERING PROPERTY

Not all serializability implementations have the commitment ordering property described above. For example, approaches based on serialization graph testing [22] do not. One example is Serializable Snapshot Isolation (SSI) [19], which is relevant because it is used as serializable isolation level in the PostgreSQL DBMS [80] that we use in our implementation of TxCache. This approach guarantees serializability, but the serial order may not match the order in which transactions commit.

Applying TxCache to a database like this requires some additional support from the database. The reason is that it is more difficult is that TxCache's consistent reads property ensures that each read-only transaction sees the effects of all transactions that committed before a certain time. In other words, the read-only transaction sees the effects of the transactions in a prefix of the *commit* order, but this might not be a prefix of the *serial* order.

To see how consistent reads might not be enough to ensure serializability, consider an example of how the commit order might differ from the serial order in PostgreSQL's Serializable Snapshot Isolation. In this example, we simulate a transaction-processing system that maintains two tables.[2] A *receipts* table tracks the day's receipts, with each row tagged with the associated batch number. A separate *control* table simply holds the current batch number. There are three transaction types:

- NEW-RECEIPT: reads the current batch number from the control table, then inserts a new entry in the receipts table tagged with that batch number
- CLOSE-BATCH: increments the current batch number in the control table
- REPORT: reads the current batch number from the control table, then reads all entries from the receipts table with the *previous* batch number (*i.e.*, to display a total of the previous day's receipts)

[2]Kevin Grittner suggested this example.

If the system is serializable, the following useful invariant holds: after a REPORT transaction has shown the total for a particular batch, subsequent transactions cannot change that total. This is because the REPORT shows the previous batch's transactions, so it must follow a CLOSE-BATCH transaction. Every NEW-RECEIPT transaction must either precede both transactions, making it visible to the REPORT, or follow the CLOSE-BATCH transaction, in which case it will be assigned the next batch number.

Consider the interleaving of transactions T_1 and T_2 in the interleaving shown in Figure B-1. (Ignore the read-only transaction T_R for the moment.) In this interleaving, the NEW-RECEIPT transaction T_1 first reads the current batch number, then uses it to insert a new value into the receipts table. While it is executing, however, the CLOSE-BATCH transaction T_2 increments the batch number and commits first – so the batch number that T_1 read is no longer current by the time it commits. This execution would not be permitted by either strict two-phase locking or optimistic concurrency control. However, this execution *is* a valid serializable history – it is equivalent to the serial execution $\langle T_1, T_2 \rangle$ – and PostgreSQL's serializable mode does allow it.

Note that although there is an equivalent serial order $\langle T_1, T_2 \rangle$ for these two transactions, they commit in the opposite order. This means that the database may have a temporarily inconsistent state in the interval between when they commit, which is only a problem if some other transaction observes the inconsistent state. For example, the read-only transaction T_R in Figure B-1 reads both the current batch number and the list of receipts for the previous batch; it sees T_2's incremented batch number but not T_1's new receipt, violating serializability. To prevent serializability violations like this, the database needs to monitor the data read by each transaction, even read-only ones. If executed directly on the database, PostgreSQL would detect the potential serializability violation here, and prevent it by aborting one of the transactions.

This concurrency control approach is problematic for TxCache because the database needs to be aware of the data read by each transaction, even if that transaction is read-only. With TxCache, some of the data accessed by a transaction might come from the cache, and the database would be unaware of this. Validity intervals

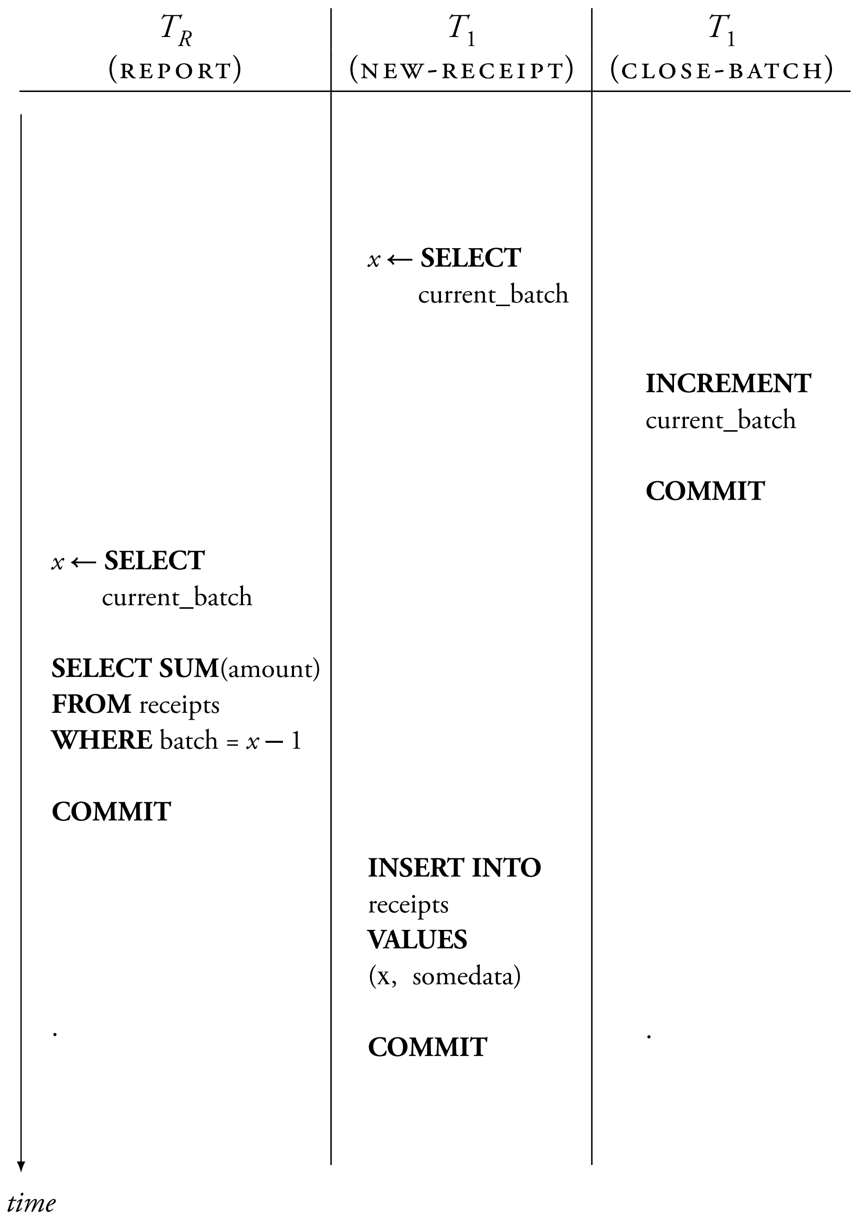

T_R
(REPORT)
T_1
(NEW-RECEIPT)
T_1
(CLOSE-BATCH)
$x \leftarrow$ **SELECT** current_batch
INCREMENT current_batch
COMMIT
$x \leftarrow$ **SELECT** current_batch
SELECT SUM(amount) **FROM** receipts **WHERE** batch = $x - 1$
COMMIT
INSERT INTO receipts **VALUES** (x, somedata)
COMMIT
time

Figure B-1: Example of an anomaly that can occur when the commit order does not match the serial order aren't sufficient to maintain consistency because they reflect the commit order of transactions, not their serial order.

However, it is desirable to support databases such as PostgreSQL that do not have this commitment ordering property. We can support these databases by observing that the consistency protocol in Chapter 5 ensures that the data read by any read-only transaction is consistent with the database state at some timestamp *that corresponds to a pinned snapshot*. Therefore, it is sufficient if pinned snapshots are taken only at points where the commit order matches the equivalent serial order of execution, *i.e.*, points where it is not possible for a subsequent transaction to commit and be assigned an earlier position in the serial order. PostgreSQL already has a way to identify these points, referred to as *safe snapshots*, as they are useful for certain in-database optimizations for read-only transactions [80]. We can ensure that TxCache interoperates correctly with PostgreSQL's serializable mode as long as we ensure that all pinned snapshots are also safe snapshots; this is done by having the PIN command wait until the next safe snapshot is available. One implication of this is that read-only transactions may have to block while they wait for a safe snapshot to become available.

BIBLIOGRAPHY

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, March 1999.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995. ACM.

[3] American National Standards Institute. Database language - SQL. American National Standard for Information Systems X3.135-1992, November 1992.

[4] Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, USA, 1998.

[5] Khalil Amiri and Sanghun Park. DBProxy: A dynamic data cache for web applications. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE '03)*, Bangalore, India, March 2003. IEEE.

[6] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Alan Cox, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. TR02-388, Rice University, 2002.

[7] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenopoel, Emmanuel Cecchet, and Julie Mar-

guerite. Specification and implementation of dynamic web site benchmarks. In *Proc. Workshop on Workload Characterization*. IEEE, November 2002.

[8] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. In *Proceedings of the International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[9] 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.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995. ACM.

[11] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed currency serializability for middle-tier caching and replication. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, June 2006. ACM.

[12] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185 – 221, June 1981.

[13] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4), December 1983.

[14] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. Technical Report 85-694, Cornell University, Ithaca, NY, USA, July 1985.

[15] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, TX, USA, October 1987. ACM.

[16] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with dbcache. *IEEE Data Engineering Bulletin*, 27(2):11–18, 2004.

[17] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[18] Cache-money. `https://github.com/nkallen/cache-money`.

[19] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 2008. ACM.

[20] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, USA, May 2001. ACM.

[21] K. Selçuk Candan, Divyakant Agrawal, Wen-Syan Li, Oliver Po, and Wang-Pin Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, Hong Kong, China, August 2002.

[22] Marco Antonio Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[23] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, May 2003.

[24] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: flexible database clustering middleware. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June 2004. USENIX.

[25] James R. Challenger, Paul Dantzig, Arun Iyegar, Mark. S. Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, 12(2), April 2004.

[26] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM 1999*, New York, NY, USA, March 1999. IEEE.

[27] Arvola Chan, Stephen Fox, Wen-Te K. Lin, Anil Nori, and Daniel R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, Orlando, FL, USA, June 1982. ACM.

[28] Arvola Chan and Robert Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, February 1985.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, November 2006. USENIX.

[30] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, USA, May 2006. USENIX.

[31] James Cowling, Dan R. K. Ports, Barbara Liskov, Raluca Ada Popa, and Abhijeet Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009. USENIX.

[32] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall,

and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, October 2007. ACM.

[33] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *Proceedings of the 17th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '00)*, Hudson River Valley, NY, USA, April 2000. IFIP/ACM.

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

[35] Ehcache. `http://ehcache.org/`.

[36] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS '05)*, Orlando, FL, USA, October 2005. IEEE.

[37] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store for cloud computing. Technical report, Cornell University, Ithaca, NY, USA, December 2011.

[38] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[39] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable query result caching for web applications. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB '08)*, Auckland, New Zealand, September 2008.

[40] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling of Data Base Management Systems*, pages 1–29, 1977.

[41] Philip J. Guo and Dawson Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *Proceedings of the 2nd USENIX Workshop on Theory and Practice of Provenance (TAPP '10)*, San Jose, CA, USA, February 2010. USENIX.

[42] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, USA, March 2004. USENIX.

[43] Ashish Gupta, Inderpal Sing Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, USA, June 1993. ACM.

[44] Priya Gupta. Providing caching abstractions for web applications. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 2010.

[45] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for ORMs. In *Proceedings of the International Middleware Conference*, Lisbon, Portugal, December 2011.

[46] Magnus Hagander. Data-driven cache invalidation. `http://www.hagander.net/talks/Database%20driven%20cache%20invalidation.pdf`, December 2010.

[47] Maurice P. Herlihy and Jeannette M. Wing. Linearizabiliy: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[48] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1626, December 1989.

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

[50] Robert Johnson. More details on today's outage. `http://www.facebook.com/notes/facebook-engineering/`

more-details-on-todays-outage/431441338919, September 2010. Facebook Engineering Note.

[51] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, September 2007.

[52] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC '97)*, El Paso, TX, USA, May 1997. ACM.

[53] Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair Veitch. LazyBase: Freshness vs. performance in information management. In *Proceedings of the 1st Hot Topics in Storage and File Systems (HotStorage '09)*, Big Sky, MT, USA, October 2009. USENIX.

[54] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, San Francisco, CA, August 2000.

[55] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[56] Marc Kwiatkowski. memcache @ facebook. In *QCon 2010*, Beijing, China, April 2010. Available at http://qcontokyo.com/pdf/qcon_MarcKwiatkowski.pdf.

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

[58] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent mid-tier database caching in SQL server. In *Proceedings of the 2003 ACM SIGMOD*

International Conference on Management of Data, Santa Diego, CA, USA, May 2003. ACM.

[59] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge, MA, USA, October 1996. ACM.

[60] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, Portugal, June 1999.

[61] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. ACM.

[62] MediaWiki. `http://www.mediawiki.org/`.

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

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

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

[66] Memcachedb. `http://memcachedb.org/`.

[67] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[68] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)*, Brisbane, Australia, August 1990.

[69] NCache. `http://www.alachisoft.com/ncache/`.

[70] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Ontario, Canada, August 1988. ACM.

[71] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[72] Gultekin Özsoyoğlu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[73] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010. USENIX.

[74] Francisco Perez-Sorrosal, Marta Patiño-Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic SI-Cache: consistent and scalable caching in multi-tier architectures. *The International Journal on Very Large Data Bases*, 20(6):841–865, December 2011.

[75] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997. ACM.

[76] `pgbench`. See `http://www.postgresql.org/docs/9.1/static/pgbench.html`.

[77] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Ontario, Canada, October 2004.

[78] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010. USENIX.

[79] Dan R. K. Ports, Austin T. Clements, and Irene Y. Zhang. Optimizing distributed read-only transactions using multiversion concurrency. Available at `http://drkp.net/drkp/papers/anastore-6830.pdf`, December 2007.

[80] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, August 2012.

[81] PostgreSQL. `http://www.postgresql.org/`.

[82] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, USA, August 2001. ACM.

[83] Yoav Raz. The principle of commitment ordering — or — guaranteeing serializability in a heterogenous environment of multiple autonomous resource managers using atomic committment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*, Vancouver, BC, Canada, August 1992.

[84] Redis. `http://redis.io`.

[85] David P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1978.

[86] Rodrigo Rodrigues, Barbara Liskov, Kathryn Chen, Moses Liskov, and David Schultz. Automatic reconfiguration or large-scale reliable storage systems. *IEEE Transactions on Dependable and Secure Computing*, 9(2):146–158, March 2012.

[87] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS — a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, Hong Kong, China, August 2002.

[88] Uwe Röhm and Sebastian Schmidt. Freshness-aware caching in a cluster of J2EE application servers. In *Proceedings of the 8th International Conference on Web Information Systems Engineering (WISE '07)*, Nancy, France, December 2007.

[89] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001. IFIP/ACM.

[90] Paul Saab. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`, December 2008. Facebook Engineering Note.

[91] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '05)*, Paris, France, January 2005.

[92] Nithya Sampathkumar, Muralidhar Krishnaprasad, and Anil Nori. Introduction to caching with Windows Server AppFabric. Technical report, Microsoft Corporation, November 2009. `http://msdn.microsoft.com/library/cc645013`.

[93] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[94] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL: Optimization, Backups, and Replication*. O'Reilly and Associates, third edition, March 2012.

[95] Emil Sit, Andraes Haeberlen, Frank Dabek, Byun-Gon Chun, Hakim Weatherspoon, Frans Kaashoek, Robert Morris, and John Kubiatowicz. Proactive replication for data durability. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Santa Barbara, CA, USA, February 2006.

[96] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):149–160, February 2003.

[97] Michael Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*, Brighton, United Kingdom, September 1987.

[98] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, September 2007.

[99] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, Washington, DC, USA, June 1986. ACM.

[100] Oracle TimesTen In-Memory Database Cache. `http://www.oracle.com/database/in-memory-database-cache.html`.

[101] Transaction Processing Performance Council. TPC Benchmark B. `http://www.tpc.org/tpcb/`, June 1994.

[102] Transaction Processing Performance Council. TPC Benchmark W. `http://www.tpc.org/tpcw/`, February 2002.

[103] Benjamin Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, October 2007. ACM.

[104] Benjamin Mead Vandiver. *Detecting and Tolerating Byzantine Faults in Database Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2008.

[105] Chris Wasik. Managing cache consistency to scale dynamic web systems. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 2007.

[106] William Edward Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, April 1984. Also available as technical report MIT/LCS/TR-314.

[107] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *Proceedings of ACM SIGCOMM 1999*, Cambridge, MA, USA, August 1999. ACM.

[108] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[109] Huican Zhu and Tao Yang. Class-based cache managmenet for dynamic web content. In *Proceedings of IEEE INFOCOM 2001*, Anchorage, AK, USA, April 2001. IEEE.