HARVARD UNIVERSITY
Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

School of Engineering and Applied Sciences

have examined a dissertation entitled:

"Easy Freshness with Pequod Cache Joins"

presented by : Bryan Nathan Kate

candidate for the degree of Doctor of Philosophy and here by certify that it is worthy of acceptance.

Signature _____

Typed name: Professor E. Kohler

Signature _____

Typed name: Professor M. Seltzer

Signature _____

Typed name: Professor G. Morrisett

Signature _____

Typed name: Professor R. Morris

Date: September 12, 2014

Easy Freshness with Pequod Cache Joins

A dissertation presented
by
Bryan Nathan Kate
to
The School of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts

September 2014

Dissertation Advisor: Professor Eddie Kohler Bryan Nathan Kate

Easy Freshness with Pequod Cache Joins

Abstract

This thesis presents the design of Pequod, a distributed, application-level Web cache. Web developers store data in application-level caches to avoid expensive operations on persistent storage. While useful for reducing the latency of data access, an application-level cache adds complexity to the application. The developer is responsible for keeping the cached data consistent with persistent storage. This consistency task can be difficult and costly, especially when the cached data represent the *derived* output of a computation.

Pequod improves on the state-of-the-art by introducing an abstraction, the *cache join*, that caches derived data without requiring extensive consistency-related application maintenance. Cache joins provide a mechanism for filtering, joining, and aggregating cached data. Pequod assumes the responsibility for maintaining cache *freshness* by automatically applying updates to derived data as inputs change over time.

This thesis describes how cache joins are defined using a declarative syntax to overlay a relational data model on a key-value store, how cache data are generated on demand and kept fresh with a combination of eager and lazy incremental updates, how Pequod uses the memory and computational resources of multiple machines to grow the cache, and how the correctness of derived data is maintained in the face of eviction.

We show through experimentation that cache joins can be used to improve the performance of Web applications that cache derived data. We find that moving computation and maintenance tasks into the cache, where they can often be performed more efficiently, accounts for the majority of the improvement.

Contents

Listing of figures

Acknowledgments

I am enormously grateful to my advisor, Eddie Kohler, for being a mentor, collaborator, and friend. With much wisdom and patience he transformed my approach to investigating hard problems and restored my self-confidence. I am a better programmer and researcher for the experience.

I also thank my committee, Eddie Kohler, Margo Seltzer, Greg Morrisett, and Robert Morris, for their guidance and feedback. In particular, many thanks are owed to Margo and Greg, for encouraging me from the outset and lending a sympathetic ear along the way. In addition, I want to thank Rob Wood, Matt Welsh, and Radhika Nagpal for giving me the opportunity to work on the tremendously cool Robobees project. Lastly, I want to acknowledge H.T. Kung for supporting me early on, and Jim Waldo for our many informative and entertaining conversations.

I enjoyed my time as a graduate student, in part due to the terrific company of my peers (in SYRAH and beyond). I greatly appreciate the time spent discussing problems, exploring new cities, and blowing off steam. Specifically, I want to thank Neha Narula, Yandong Mao, and Mike Kester, for contributing considerable time and effort to the design and implementation of Pequod, and Karthik Dantu, Jason Waterman, and Peter Bailis for making the Robobees years so enjoyable and fruitful.

To my family and friends, near and far, old and new: thank you for providing an escape from the grind, and understanding when I could not get away. I cannot thank my parents enough for their constant encouragement and support (especially the weekly child care). I also want to thank Liz, super nanny and beloved Auntie, for brightening Henry's life and making mine so much easier.

Finally, I must thank my wife, Katie, for her endless patience and unwavering confidence in this endeavor; I could not have done this without her. And, of course, my dear son Henry, whose arrival prolonged this experience. He fills my life with stress, laughs, and cuddles, and I would not have it any other way.

This thesis is an extension of "Easy Freshness with Pequod Cache Joins," published in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2014. It is joint work done with Eddie Kohler, Mike Kester, Neha Narula, Yandong Mao, and Robert Morris, and was supported by the National Science Foundation, Microsoft Research, and Quanta Computer.

1

Introduction

Scaling a Web application to handle millions of users is not straightforward. Some of the largest Web companies have struggled—sometimes publicly [26]—to find the right combination of technologies to satisfy their needs. This thesis focuses mostly on issues related to data storage, which are plentiful. Relational databases are notoriously difficult to scale; the overhead required to maintain ACID guarantees can inhibit performance [23]. Strategies for scaling persistent storage, such as sharding or deploying a non-relational storage system [13, 18, 33], can improve application performance, but do not solve the scalability problem entirely. At some point, persistent storage becomes a bottleneck.

System performance is often improved by the introduction of an *application-level cache*: a component that stores a partial copy of application data in memory. The application-level cache is a storage layer situated between the application servers and persistent storage. The cache offloads many of the read operations that would otherwise be handled by the persistent store. With correct provisioning and a cache-friendly workload, this architecture leads to a favorable separation of concerns; the cache handles a large portion of read operations while the persistent store handles writes and cache misses.

However, application-level caches also tend to make applications more complex. It is much easier to code an application when all its storage requests produce authoritative results—when all reads return the most recently written data, and all writes immediately

become persistent. But application-level caches make this kind of semantics much harder to provide, because copies of the data are stored in more than one component (rather than, for instance, an authoritative database). Application-level caches are by definition designed by application developers, who must specify what data are cached and how they are kept fresh (i.e., consistent with the persistent store). Unfortunately, the most useful application-level cache designs often make it difficult to keep data fresh.

Twitter

These issues become clear in the context of the Twitter application [40], a popular microblogging service. As we will see, maintaining freshness in Twitter's application-level cache requires a great deal of application effort. At its core, Twitter comprises three operations: users can "tweet" new messages, subscribe to the tweets of other users by "following" them, and check their timeline (a summary of the recent messages tweeted by those they follow). Tweeting and following are both write operations that change the state of the application: at a minimum, tweets and subscription lists must be stored persistently. Timeline checks are pure read operations. Fortunately, timeline checks outnumber new tweets and subscription changes in the Twitter workload by a factor of 100x and 10x, respectively [25]. This read-heavy workload suggests that introducing an application-level cache might boost application performance.

Unfortunately, a user's timeline is not first-order data; by definition, it must be *derived* from *base* data (the user's subscription list and the list of recent tweets by each user he follows) with a bit of computation. But where in the application workflow should timelines be computed? If computed upon request, a cache could simply store copies of the base data (recent tweets and subscription lists). In this case, maintaining cache freshness is

trivial—every change to the persistent store corresponds directly to a change in the cache. However, this is not how the engineers at Twitter use the application-level cache to serve timelines. Computing timelines on demand, even from cached base data, adds an unacceptable amount of latency to the timeline check operation. Rather, the cache stores *precomputed* timelines in ready-to-read form [25]. When a user first logs into the service, her timeline is computed on demand and stored in the cache. After this point, her timeline is kept fresh with *incremental updates* that occur when new data are written (e.g., a followed user posts a new tweet). Subsequent timeline checks return the pre-computed timeline without any additional computation. Since timeline checks dominate the workload, the Twitter developers decided to optimize the system for that operation. Unfortunately, this makes cache maintenance more complex; identifying and updating the derived cache entries for each write to the base data is nontrivial. For example, a new tweet by a celebrity may be appended to millions of follower's cached timelines. The Twitter developers accept this trade-off to provide faster reads.

Materialized views

The Twitter timeline computation is split into two parts, a one-time bootstrap computation and an ongoing series of incremental updates. This construction is known as a *materialized view* in the relational database community [28]. A materialized view is the result of a database query that is stored as a separate table and kept consistent with the input tables used in its construction. Developers can use materialized views to avoid costly query execution on each read. In trade, the database uses more disk space to store the materialized table and takes more time to process writes to the base tables (this extra work ensures that the materialized views are updated accordingly).

Perhaps the biggest advantage of materialized views in modern databases is that the complexities of view creation and maintenance are abstracted away. A developer can define and query a materialized view with a few lines of SQL. The code for tracking dependencies between base and derived tables and performing incremental updates is completely encapsulated in the database. Unfortunately, this convenient abstraction is trapped in the database, the component that will eventually become a bottleneck. To scale the application, developers resort to reimplementing this database feature using the cache. As a result, the algorithms for computing and updating derived data are embedded in application code. Maintaining the correctness of this code grows more difficult as the application adds new features, or more developers join the team.

This thesis presents a new design for software used to cache derived data in large-scale Web applications. Our system provides an abstraction, the *cache join*, that allows developers to perform in-cache computations, including view materialization. Cache joins improve overall system performance by eliminating communication between the application server and the cache and simplify application code by moving the responsibility for computation and maintenance into the cache.

Description and goals

Our system, Pequod, is an application-level cache. Like other application-level caches, Pequod can be deployed into a data center to hasten the retrieval of information by application routines. At its core, Pequod is a key-value cache that supports the usual key-value operations (GET and PUT). But Pequod's strength and novelty lie in features that facilitate the development of modern, dynamically constructed Web applications.

We are motivated, in large part, by a desire to improve the *programmability* of Web application-level caches. We recognize that developers are implementing sophisticated caching solutions with primitive tools, a process that is both cumbersome and error prone. We address this problem by extending the familiar key-value interface to include more powerful data manipulation primitives. Pequod is designed to provide the following features:

- **In-cache computation.** A fair number of Web applications cache derived data: aggregate values (e.g., average rating), filtered sets (e.g., top ten stories), and lists (e.g., article comments). Each of these cache entries represents the result of a computation performed in an application routine. To generate or update a derived value, the developer must fetch the input data (ideally from the cache), perform the computation, and store it into the cache. In Pequod, developers can describe basic computations that execute within the cache, operating over a set of cache entries and producing new entries. In-cache computations avoid the data transfer overhead to and from the application server and eliminate boilerplate code.
- **Automatic cache management.** Cache maintenance complicates application code. Pequod removes this burden from the application, maintaining freshness internally by tracking dependencies between cache entries and responding to modifications. When in-cache computations are used to produce derived data, the input and output entries are added to an acyclic dependency graph. A traversal of this graph enumerates the cache entries that need to be invalidated or updated when an input is modified. Implementing this type of maintenance in application code with existing caches is possible, but tedious and clumsy.
- **Updating views.** Computing entries and tracking dependencies in the cache simplifies cache management, but it also enables another feature: updates. For many

in-cache computations, the value of a derived cache entry can be incrementally updated, allowing it to remain in the cache. Pequod exposes this functionality to the developer in the form of materialized views—data that are computed and maintained by the cache in a ready-to-read state. Using a cache to store materialized views is not new; Pequod merely incorporates this design pattern as a first-class primitive, making this complex mechanism more accessible.

Pequod exposes these features to application developers with a single abstraction: the *cache join*. A cache join is a declarative statement that defines a computation to be performed by the cache, the cache entries to use as inputs, the keys into which the results will be stored, and the desired maintenance policy for the derived output (one-time computation or incrementally updated). Developers can use cache joins to filter, join, and aggregate cache data with relative ease.

The overarching goal of this work is to improve Web application programmability by transferring complexity into the cache. But improved programmability is of little value if it comes with a significant performance penalty. Pequod is designed to be competitive with existing application caches, offering performance that is comparable to solutions with less expressive interfaces. Experimental results show that in-cache materialization performs no worse than application-managed materialization; in fact, Pequod offers a substantial improvement over existing caches.

Thesis overview

The remainder of the thesis presents the design, implementation, and evaluation of Pequod. Chapter 2 provides a review of application-level caches, the key-value data model, the relational data model, and database materialized views. The remainder of the chapter

categorizes existing techniques for caching dynamic Web applications and compares Pequod with related database and caching systems.

Chapter 3 motivates the need for Pequod with two example Web applications. These applications demonstrate the utility of materialized views for Web applications and the difficulty of implementing this feature using existing key-value caches. One application, a feed-based social network, uses materialization to append the latest information to each user's feed. Pequod essentially prefetches user feeds for quick retrieval. A second application uses materialization to compute and maintain aggregate values that are repeatedly queried. Incremental updates ensure that the aggregates remain fresh.

In Chapter 4 we describe two extensions to the traditional key-value data model, range scans and relational overlays, that are used as building blocks for cache joins. With these extensions, developers can group key-value pairs into data ranges and decompose keys into named components. We also discuss how Pequod can be deployed into an existing Web architecture and how the communication pattern of the application is modified to make use of the cache.

Chapter 5 focuses on data materialization in the cache. We present and discuss the language used to define cache joins. The language allows developers to specify the portion of the keyspace to be computed, the sources of data used as input, and the operator used in the transformation. Finally, we present algorithms for generating new cache entries and maintaining materialized ranges using incremental updates and partial re-evaluation.

Pequod is a distributed system that can be scaled to meet the demands of a growing application. Chapter 6 describes how cache entries are stored and retrieved in a distributed deployment. By design, each cache join invocation is restricted to a single Pequod server. Data that is required to complete the join execution but is not memory-resident (e.g., is

stored on another Pequod server) is copied to the executing server and kept fresh with a subscription mechanism. Likewise, the algorithm for incremental maintenance of derived ranges is updated to support non-local dependencies.

Chapter 7 presents the challenges associated with evicting entries from the cache. Eviction in Pequod is complicated by cache joins. In a typical key-value cache, each key is independent, and can be evicted based on a simple policy (e.g., LRU). However, cache entries in Pequod are neither independent nor equal with respect to eviction cost. Some entries correspond to the derived output of materialized views and have an additional recomputation cost if they are requested post-eviction. Likewise, the eviction of base entries (those used as inputs to one or more cache joins) may result in a cascade of dependent evictions. We discuss and evaluate several eviction policies and an approach to mitigate the impact of cascading evictions in Chapters 7 and 9.

We implemented a prototype system to explore design alternatives and evaluate the choices made. Chapter 8 describes the prototype's construction—specifically, how data is structured and messages are passed—and several optimizations that make it performance-competitive with commercial key-value caches. Chapter 9 presents performance measurements and design analysis. Experimentation shows that Pequod performs comparably with existing key-value caches for most of the workloads tested. Moving materialized views into the cache improves overall performance. On one benchmark, Pequod cache joins outperformed application-managed materialized views by a factor of 1.33x and database materialized views by a factor of 9.55x. Further, we demonstrate that the computational and storage capacities of Pequod scale to accommodate a Web class workload.

Finally, Chapter 10 concludes.

Summary of contributions

View materialization is not novel; the topic is well covered in the relational database literature. The core contributions of this thesis are derived from the application of materialized views in a new context—a key-value application-level cache. Pequod's design addresses several challenges that arise: expressing data transformations in an intuitive way using the key-value abstraction, computing and maintaining derived data from base data partitioned amongst multiple servers, and maintaining cache correctness and performance in the wake of eviction. Further, Pequod supports *partial* materialization: only portions of each view's derived output may be resident in the cache (and maintained) at any time. Pequod is capable of materializing the whole view, but avoids doing so until it is requested. The subset of the view that is materialized is selected *dynamically* based on the access pattern of the application. Though some of these techniques are described in other research systems (§2), their combination and application in the context of an application-level Web cache make for a unique system design. The contributions of this thesis include:

- the cache join abstraction for expressing in-cache computations;
- algorithms for aggregating, joining, and filtering data in a key-value cache;
- techniques for efficiently tracking dependencies between cache entries;
- a distributed system design that supports cache join execution and maintenance;
- a range-based, view-aware cache eviction strategy;
- a performance analysis of cache joins on real-world workloads;
- and the Pequod software itself.

We evaluate the ideas presented in this thesis using a prototype implementation of Pequod. The software is available for download at `http://github.com/bryankate/pequod`.

2

Background

2.1 Existing key-value application-level caches

Read operations dominate many Web application workloads. When persistent storage becomes a bottleneck, developers often deploy a cache to absorb application reads. Application-level caches are incredibly popular with Web developers, and power some of the world's largest services: at Facebook [20] and Twitter [40], application-level caches process billions of requests per second and store trillions of items [5, 34].

The popularity of these systems is due not only to performance, but also to the simple, flexible interface presented to developers. Both memcached [30] and Redis [37], two of the most widely deployed systems, are *key-value caches*. A key-value cache stores $\langle key \mapsto value \rangle$ pairs, mapping uniquely named keys to arbitrary values. These systems provide a core interface:

- **PUT**(k, v). Inserts a key-value pair into the cache.
- **GET**(k). Retrieves the value stored under the key k, if it exists.
- **REMOVE**(k). Removes the key-value pair stored under the key k, if it exists.

Using this basic interface, it is easy for developers to store essential application data—for example, images, session information, or database query results—and retrieve them quickly. Of course, caches are not limited to the key-value interface: more complex interfaces exist, and are useful for improving edge cases and caching structured data [34, 41].

But deploying an application-level cache adds complexity to the application code; the developer is responsible for maintaining cache freshness as data are modified. For most applications, cached data should reflect, as closely as possible, the state of persistent storage. Developers use *expiration* and *invalidation* to keep the cache fresh. Key-value pairs with an explicitly defined expiration time are automatically removed from the cache. Expiration is useful for applications that can tolerate some staleness, such as a periodically updated news site. For applications that cannot tolerate staleness, cache entries should be made invalid along with the precipitating database write (or soon thereafter). The developer is tasked with identifying which entries are affected and modifying the cache by removing the entry or by *updating* it with a new value. We focus on applications that require timely cache maintenance through updates and invalidation.

Cache maintenance is more difficult when entries represent *derived* data, the results of some computation over *base* data. For example, if a developer caches the result of the SQL database query

```
SELECT department, AVG(salary) AS avgpay
  FROM employees
  GROUP BY department;
```

that computes the average salary of employees by department, she would have to invalidate or update the cached result when any of the base data are changed. In this case, any change to the `employees` table—adding or removing a row (e.g., an employee is hired or quits), changing an employee's salary, or switching his group—would require action by the application. At the very least, the developer could invalidate the cached result with every modification to the source table. This approach would ensure correctness, but require a re-computation on subsequent reads. The problem is harder when the scope of the query is narrowed:

```
SELECT department, AVG(salary) AS avgpay
  FROM employees
  WHERE salary > 100000
  GROUP BY department;
```
(2.1)

Invalidating the cached result for every change to the `employees` table is correct, but inefficient. A better approach would recognize that the result is conditionally dependent on a subset of the `employees` table and only invalidate the cached result for changes to those rows. Unfortunately, it is difficult to capture the knowledge of these data dependencies in application code. In addition, it is often more efficient to implement the control logic for invalidating and updating cache entries within the cache than to distribute the same responsibilities amongst multiple independent clients.

2.2 Automatic cache maintenance

Several research systems are designed to reduce the burden of cache maintenance by assuming more responsibility for producing and applying cache invalidations.

TxCache [35] is an application-level cache that tracks dependencies between cached objects and tables in a relational database. The dependencies are used to automatically invalidate cached objects when the database is modified. TxCache improves application programmability by making caching transparent. Clients write queries against the database, and query results are automatically cached. Future queries may return data from the cache, the database, or a combination of the two. The system ensures that mixed results are transactionally consistent (Pequod is designed to be eventually consistent). TxCache achieves one of Pequod's goals, automatic cache maintenance, but it only supports invalidation; cache entries cannot be updated. Without updates, the system will not support cached materialized views, a key design pattern used by many applications.

TAO [41] is a storage system and cache developed and deployed at Facebook. It is specially designed to manage graph-structured data. Clients make changes to the graph structure by adding or removing nodes and edges and by modifying metadata associated with each. Like TxCache, caching is transparent to the client; application code has no direct control over cache usage, and as such, has no responsibility for maintenance. The system mediates access to the database, routing both reads and writes through a caching layer. As a write-through cache, TAO can guarantee that cached data are consistent with the database by performing synchronous writes. However, TAO is a replicated cache, and replicas should reflect the state of the master cache server. During a write, the TAO master cache produces a changeset that can be applied to cache replicas. Thus, cache replicas are updated, not invalidated. Though the cache is updated automatically, TAO is specially constructed for graph operations and does not allow for user-defined materialized views.

DBProxy [3, 4] is an edge cache that proxies database queries and caches results. The system inspects incoming queries and determines if the query can be handled locally from cached data. If not, the query is sent to a master database for execution and the results are cached locally for future use. If the cache is populated with all the data needed to satisfy a query, it is executed without contacting the master. The system rewrites the queries that it sends to the master for execution, widening the scope of constraints in an effort to prefetch base data for the local cache. The master database sends updates to the cache servers for insertions, deletions, and updates to cached base data. DBProxy operates as a transparent cache. The system selects the data to cache and ensures its integrity. The system is designed to handle repeat and overlapping queries efficiently, but is ill suited for Twitter-like workloads that constantly request new data. In such workloads, each request would contact the master database.

Challenger et al. [11] describe an algorithm, Data Update Propagation (DUP), for tracking dependencies between cached objects (HTML pages or fragments) and underlying database tables. The algorithm was deployed in a system that cached the official website for the 1998 Winter Olympic Games. The algorithm works by constructing a directed graph in which nodes represent objects (base data or derived content) and edges represent dependencies. When new data are written, the system traverses the graph to determine the set of objects that require invalidation. The system supports pseudo-updates in the form of prefetching: rather than waiting to generate a new object on the next access, some invalidated objects are re-generated by a background application routine. This system eases the burden of cache maintenance by automatically identifying and invalidating transitively derived data. Unlike TxCache and TAO, the mechanism is not transparent: the developer must provide the data dependencies to the system for tracking. Pequod also tracks dependencies between cache objects, but infers the relationships from materialized view definitions. In addition, Pequod supports incremental maintenance of cached objects, avoiding full re-computation in the common case (§5.5).

2.3 Materialized views

In a relational database, a *view* holds the results of a stored query. For example, a developer could define a view for query (2.1) as:

```
CREATE VIEW highpay AS
  SELECT department, AVG(salary) AS avgpay
    FROM employees
    WHERE salary > 100000
    GROUP BY department;
```

The view definition defines the structure of `highpay` and captures the constraints of the stored query (salary greater than $100,000). To find the average salary of highly paid em-

ployees in the Engineering department, the client could issue the query:

```
SELECT avgpay
  FROM highpay
  WHERE department = 'Engineering';
```

This abstraction nominally breaks query (2.1) into two parts, one to compute the results and one to fetch them. However, database views are not generally executed in two phases. Though a view may be stored temporarily by the database to improve query performance, there is no guarantee that pre-computed results will be available at query time. Thus, non-materialized views provide a useful abstraction but offer no significant improvement over equivalent on-demand queries. When a database view is *materialized* [10], its contents are stored persistently as a table. Reading results from an up-to-date materialized view requires no additional computation (of the stored query).

2.4 View maintenance

There are many uses of database materialized views, each with its own consistency requirements. As a result, database systems have adopted several strategies for maintaining materialized views. Gupta and Mumick [21] and Chirkova and Yang [14] survey and summarize techniques for generating and updating materialized views. We are most interested in strategies that keep the view as fresh as possible while minimizing the cost of reading results. These goals are consistent with applications like Twitter that rely on pre-computation to handle timeline checks at scale. Thus, we concentrate on *incremental view maintenance* methods [7, 22] that update a view to reflect individual changes rather than recomputing the whole view or applying changes as a batch.

Materialized views can be maintained *eagerly* or *lazily*. An eager view maintenance strategy updates derived results synchronously with the precipitating change to the underlying

data. Eager maintenance trades more complex writes (to base data) for faster reads (of derived data). In contrast, a lazy view maintenance strategy [42] defers maintenance operations until the next access to derived data or until the system is lightly loaded and can process updates using idle cycles. When base data are modified, the system computes a minimal changeset of affected tuples along with other contextual information (e.g., the state of the base relations prior to the modification) and logs it for later application. Pequod allows developers to apply either approach according to application needs.

2.5 View selection

In the realm of relational databases, *view selection* refers to the problem of determining which views should be materialized to optimize application performance. This is typically an offline process undertaken by developers, though automated approaches [2] have been proposed in the literature and implemented in commercial databases. Similar tools could be applied to view selection in an application-level cache, perhaps with more weight given to space and expected query frequency. However, we assume that this problem is solved by the developer, who explicitly defines cached views.

Luo proposed partial materialized views (PMV), caching portions of the results of long running queries to quickly generate partial results in future queries [29]. For example, if a query is expected to yield 5,000 results in 30 seconds, PMVs can be used to return the first 100 rows as soon as the query begins. This approach presumes that the requester gains some advantage from inspecting the early, partial results while the remainder are computed. The cached PMVs are invalidated when changes are made to the base tables, but no incremental updates are performed. PMVs are not particularly applicable in a cache setting, where on demand computation is avoided.

Zhou et al. [43, 44] describe dynamic materialized views (DMV), a method for computing and caching portions of a materialized view (in a relational database) based on application access patterns. Developers install materialized view definitions and policies that guide dynamic materialization. DMV is motivated by the desire to save space. Rather than materializing the view in full (as is standard for database materialized views), DMV begins answering queries with on demand computation. Once partial results are generated, the installed policy—for example, one that admits partial results based on frequency of access—determines if they should be stored. Stored results are kept fresh through incremental maintenance.

The design of partial, dynamic materialization in DMV is extremely relevant to the key-value cache context, in which there may be only enough space to store a portion of the defined views at any given time. Pequod takes a similar approach to view generation. However, there are some differences that distinguish the two systems. For example, Pequod materializes portions of the view in contiguous ranges. In DMV, output tuples can be generated individually based on the contents of a special control table. The rows in the control table correspond to input values that have been approved for materialization (by an installed policy). A predicate is evaluated against the control table to determine if a portion of the view should be materialized. In this way, DMV could generate non-contiguous portions. However, this approach places some limitations on the types of inputs that can be used in the predicate (e.g., aggregates). This limitation prohibits some desirable composition, for example, a view that filters the output of an aggregate view. Pequod does not have this limitation, but may incur more overhead to produce non-contiguous output.

2.6 Distributed materialized views

Agrawal et al. [1] added materialized views to PNUTS [16], a distributed, persistent store that exposes a relational data model to users. Internally, tables are implemented with column families [13], a variant of the key-value model. Like Pequod, views are implemented as partitioned tables, are eventually consistent, and are maintained with asynchronous, incremental computation. However, this work did not support partial materialization or some of Pequod's performance annotations. Interestingly, the authors use a different execution strategy for aggregate joins, which use a distributed query to reduce data movement. Pequod might benefit from a similar strategy.

2.7 Summary

Pequod is not the first system to automate cache maintenance by tracking dependencies, materialize views partially and dynamically, or update views incrementally using both eager and lazy strategies. However, Pequod is the only system to combine these features in the context of a distributed key-value application-level cache. The end result is a system with the convenience of materialized views and the performance of a key-value cache.

3

Motivation

We use two example Web applications to demonstrate in-cache view materialization with Pequod. These applications are Twip, a Twitter-like microblogging service, and Newp, a Reddit-like news aggregator [36]. Both are introduced here and referenced throughout the remainder of the thesis.

The Twip application focuses on the core functionality of Twitter, constructing user *timelines*. Twip supports three end user operations: a user can *post* tweets, *follow* other users (subscribe to their tweets), and *check* his timeline. This last operation is the most complex: when a user, called the requester, checks his timeline, Twip returns, in a time-sorted list, all recent posts made by any other user that the requester follows. Other aspects of the Twitter service (e.g., search and trending topics) are not represented. Twip uses materialized views to compute and maintain user timelines.

Newp is a simplified news aggregator; links to articles are submitted by end users and are displayed in some ranked order for others to read. The application supports four operations. A user can *post* a new article, *comment* on an existing article, place a *vote* for an article that may affect its ranking on the site, and *browse* an article. Other features of real applications like Reddit, such as voting on comments and constructing a top-N listing of popular articles, are not modeled in Newp.

When a Newp article is browsed, its contents are displayed to the end user. These include the link associated with the article, the article's current vote count, and the attached comment threads, where each comment is accompanied by the *karma* of its author. An author's karma is computed as the sum of the vote counts for all of the articles that she has posted. Newp uses materialized views to maintain frequently read aggregate values (article vote counts and user karma) and organize article data into a form optimized for browsing.

The following sections present exercises in caching the Twip and Newp applications using existing technologies. This exposition outlines the challenges inherent to each application and sets the stage for Pequod. For concreteness, the persistent storage layer is assumed to be a relational database with materialized view support.

A note on terminology: The term *client* is used throughout this document in reference to an application routine that executes within the data center and issues requests to the cache. Humans visiting a Web site are referred to as *end users*.

3.1 Caching Twip

The Twip application stores two types of data into persistent storage, subscription lists and posts. In SQL, these tables might be defined this way:

```
CREATE TABLE sub
  (user varchar(32), follows varchar(32));

CREATE TABLE post
  (poster varchar(32), time int, content varchar(140));
```
(3.1)

The above table definitions are used to illustrate the base relations used by the application for timeline construction; supporting tables, optimizations, and constraints are omitted for clarity. If one user subscribes to another, an entry is added to the `sub` table mapping the subscriber's user ID with that of the user being followed. When a user makes a new post,

it is inserted into the post table with a timestamp.

To refresh the user ann's timeline to contain posts on or after time 100, the application could issue the SQL query:

```
SELECT post.time, post.poster, post.content
  FROM sub, post
  WHERE sub.user = 'ann'
    AND sub.follows = post.poster
    AND post.time >= 100
  ORDER BY post.time;
```
(3.2)

While straightforward, this query is potentially costly to execute. Disk access and concurrency control mechanisms (i.e., locking) can cause long delays. In addition, the sub and post tables are joined and filtered on each query. Executing frequent, expensive, latency-sensitive queries on a persistent database is inadvisable. Timeline checks must be cached.

One option is to store copies of base data (rows from the sub and post tables) in the application-level cache. To construct ann's timeline from the cache, the application first retrieves and parses her subscription list, represented by the key-value pair $\langle$`sub|ann` $\mapsto$ `bob|jay|sue`$\rangle$. Next, it fetches and parses the posts of users bob, jay, and sue. For example, the key-value pair $\langle$`post|bob` $\mapsto$ `100:Hi|102:Bye`$\rangle$, represents user bob's posts of Hi at time 100 and Bye at time 102. Finally, the application assembles the timeline, returns it to the end user, and stores it into the cache as `tline|ann`.

This process is identical to the above SQL query, it is just shifted into the application. The base data cache eliminates some overheads (assuming cache hits) but introduces others. For example, two rounds of communication (and, for typical users who follow many others, hundreds of requests in total) are required to transfer the base data to the back-end server running the application code. And if ann refreshes her timeline one second later, the same data are transferred to the application and joined, effort that is superfluous in the likely scenario that nothing has changed.

An alternative caching strategy for Twip mimics the materialized view approach taken by the Twitter engineers. Each user has a pre-computed timeline stored in the cache (for example, `tline|ann`). To check a user's timeline, the application fetches this key-value pair from the cache. If present, the value is used as-is. If not, the application computes the timeline from base data and stores it into the cache for future use, as above. After the initial computation, the application keeps the cached timelines fresh through incremental maintenance. For example, when a user makes a new post, the application adds the post to the previously cached timeline of every user that follows the poster. Thus, inherent in this application-managed materialized view is a performance trade-off. The application avoids on demand computation in the common case, reducing the latency of timeline checks. In trade, writes are more complex. The application must identify all cache entries that depend on the value written and update each accordingly. Each update uses system resources (memory, CPU, network bandwidth) and represents a potential overhead: there is no guarantee that the update will ever be read, thus making some updates superfluous.

Materializing timelines is an acceptable performance trade-off given the highly timeline-skewed workload of Twitter [25], but what is the impact on application programmability? Maintaining the cached timelines is not necessarily a trivial task. What happens if two users post concurrently and have followers in common? Is it the application's job to lock keys to safely read-then-update or does the cache have an atomic append command? What happens to the cached timeline if a user un-follows another user or deletes a post? These questions are answerable by the application developer, but the solutions certainly add unwanted complexity.

The above caching strategy represents an application-managed materialized view. The `tline` keys are just a rearrangement of base data that facilitates timeline checks. Appending

new posts to the appropriate `tline` keys is a form of view maintenance. These features exist in relational databases. For example, the actions taken by the application are roughly equivalent to the SQL command:[1]

```
CREATE MATERIALIZED VIEW tline AS
  SELECT sub.user, post.poster, post.content, post.time
    FROM sub, post
    WHERE sub.follows = post.poster;
```
(3.3)

The statement establishes a new table, `tline`, that stores the pre-computed timelines of all users. The inputs to this view are the `sub` and `post` tables, and the operator that transforms the inputs is a natural join. We assume the database applies a view maintenance policy that will update the `tline` table whenever a transaction commits that modifies either `sub` or `post`.

This expression is not too complex, but its real benefit is how simple it makes the expression of timeline queries. Query (3.2) can now be expressed like this:

```
SELECT * FROM tline
  WHERE user='ann'
    AND time >= 100;
```
(3.4)

Thus, important application queries get simpler, thanks to a materialized view definition that is specified only once.

Database-managed materialized views are appealing. View definition is succinct, creation and maintenance is contained within the database, and application operations map directly to simple database queries. The downside is performance in a caching role, which is significantly limited with respect to key-value application-level caches. To be fair, the database should not be expected to handle the timeline check workload in addition to its primary responsibility, reliably persisting application data. Disk latency and concurrency

[1] Extensions to the SQL language vary by product. If explicit materialized view definitions are not supported, they can often be approximated using other database features, such as table triggers.

management overheads are unavoidable in the persistent store. Existing application-level caches can be deployed to augment the database and absorb application read requests (such as timeline checks), but lack the convenience offered by database materialized views. Pequod provides the power and simplicity of materialized views with the performance of a key-value cache.

3.2 Caching Newp

The Newp application comprises three types of base data: article links, comments, and votes. To store this information in a relational database, the application developer might choose tables like:

```
CREATE TABLE article
  (aid int, author varchar(32), link varchar(255));

CREATE TABLE comment
  (cid int, aid int, user varchar(32), msg varchar(255));

CREATE TABLE vote
  (voter varchar(32), aid int);
```
(3.5)

Of course, these tables are simplified for illustrative purposes and omit details required of a real service like Reddit. The `article` table holds basic information about the articles submitted by users, including a unique identifier. Likewise, the `comment` and `vote` tables record the relevant article metadata.

When a user selects an article to browse, the application must assemble the relevant data for the response. To browse article 77 without the assistance of a cache, the application might issue a SQL query like:

```
SELECT article.aid, article.author, article.link,
       comment.user, comment.msg,
       karma.val, COUNT(vote.aid) AS vcount
  FROM article
  LEFT JOIN comment ON article.aid = comment.aid
  LEFT JOIN
    (SELECT article.author, COUNT(*) AS val
       FROM article, vote
       WHERE article.aid = vote.aid
       GROUP BY article.author) AS karma
    ON comment.user = karma.author
  JOIN vote ON article.aid = vote.aid
  WHERE article.aid = 77
  GROUP BY article.aid, comment.cid, karma.val;
```
(3.6)

The above query retrieves all of the information necessary to render an article page. However, it contains two computations, a simple count of votes for the article requested and a more complicated karma computation for *each* user that has commented on the article. Computing karma on every article access is wasteful and not sustainable in practice. The application developer could replace the inline vote count and karma computations with materialized views:

```
CREATE MATERIALIZED VIEW vcount AS
  SELECT aid, COUNT(*) AS val
    FROM vote
    GROUP BY aid

CREATE MATERIALIZED VIEW karma AS
  SELECT article.author, COUNT(*) AS val
    FROM article, vote
    WHERE article.aid = vote.aid
    GROUP BY article.author
```
(3.7)

Like the materialized view in Twip, these views rearrange base data into forms that are more convenient to access. Newp stores *aggregate* values in the `vcount` and `karma` tables, avoiding the costly scan-filter-count operations it had performed for each execution of query (3.6). With the materialized views in place, the query to fetch an article is simplified to:

```
SELECT article.aid, article.author, article.link,
       comment.user, comment.msg, karma.val, vcount.val
  FROM article
  LEFT JOIN comment ON article.aid = comment.aid
  LEFT JOIN karma ON comment.user = karma.author
  JOIN vcount ON article.aid = vcount.aid
  WHERE article.aid = 77
  GROUP BY article.aid, comment.cid, karma.val;
```
(3.8)

Materializing aggregate data makes sense in this context. Like Twip, the Newp workload is skewed in favor of read operations; doing more work to maintain views during write operations is justified. Unfortunately, a database-only architecture cannot provide the performance required of this application. The real Reddit service makes extensive use of caching and simple storage. In addition, the application uses a background queue to precompute and cache data, including user karma, in response to new data. Interestingly, 80% of Reddit's traffic is generated by unregistered users ("lurkers") who always see cached content. These requests are handled by static caches and a content distribution network (CDN) [19]. Nonetheless, maintaining vote counts, karma, and comment lists is of the utmost importance to the success of the site. Its registered users, who generate and curate the content, rely on data that are accurate and fresh. The Newp developer could use application-managed materialized views to keep cached copies of these data. Pequod can make some of these tasks simpler by moving the burden of cache maintenance from the application into the cache itself.

4

Usage

In this chapter we describe how Web application developers deploy and interact with Pequod. We introduce some basic extensions to the traditional key-value interface and lay the groundwork for cache joins.

4.1 Data ranges

Pequod presents a familiar key-value abstraction to application developers. Keys and values in Pequod are arbitrary strings, and the cache interface includes the usual PUT, GET, and REMOVE operations. However, Pequod is internally organized as an *ordered* store. By storing keys in sorted order, Pequod incorporates a useful primitive, the *data range*. An additional operation, SCAN(*first*, *last*), returns a list of key-value pairs, lexicographically ordered by key, that fall within the half-open range [*first*, *last*).

By manipulating the *format* of the cache keys, an application developer can form data ranges that are meaningfully grouped for quick retrieval. A common prefix can be used to create a set of key-value pairs; for example, the key format `img`|*userid*|*imgid* would allow all of the images associated with a particular user (identified by *userid*) to be fetched with a single SCAN. Adding a timestamp to the key (`img`|*userid*|*time*|*imgid*) format turns this range into a sorted set, allowing the application to query specific time spans for a given user. Note that Pequod always applies a canonical (lexicographic) ordering to keys—it is up to the developer to exploit this ordering in a domain-specific way.

In many situations Pequod manipulates data in entire ranges rather than by individual key. Data ranges are used to define materialized views, (§5.1) track dependencies of derived data (§5.5), manage data movement between cache nodes (§6.2), and evict data from the cache (§7.1). However, this primitive comes at a cost; Pequod must store the key-value pairs in sorted order, and thus cannot execute most cache operations in $O(1)$ time. Applications that do not take advantage of data ranges will certainly perform better with another caching system. However, experimentation shows that Pequod can outperform other caches for workloads that incorporate materialized views—a consequence of in-cache computation and clever optimizations.

4.2 Relational overlays

From a programmability perspective, database materialized views are appealing because they succinctly define how derived data are constructed from available inputs. A view's definition is declarative: developers state the desired output, the sources of data, and the constraints that must be satisfied before output is generated. These declarative statements are constructed using the relational data model [15], in which tuples of data are arranged into relations (in SQL terms, rows into tables). The elements of a tuple (a column in SQL) are referable by name, as are the relations.

Ideally, Pequod would borrow this approach, allowing developers to make similar view definitions in the application-level cache. But Pequod is not relational, so creating a view definition requires some extension to the basic key-value data model (which treats keys and values as opaque strings). This is accomplished by embedding information *within* the keys; developers treat keys as tuples, adding "columns" of data that can be referenced by name. Keys are grouped into "tables" by adding a common prefix to the key. For example,

the key format

`img`|*userid*|*imgid*

defines a relation, `img`, consisting of tuples with two named elements, *userid* and *imgid*. In terms of the key-value model, this relation is just a set of key-value pairs with a common prefix. As in any key-value cache, the keys are still used holistically to uniquely identify values. However, if the developer informs Pequod of the key formats, the system can interpret the keys as tuples, essentially layering a relational model on top of the key-value model. We refer to this mapping as a key's *relational overlay*.

Relational overlays allow developers to define cache joins using a declarative syntax. Cache joins use concepts from relational algebra—joining, projection, filtering, and aggregation—to produce new arrangements of cache data.

An important distinction between database materialized views and Pequod cache joins is the visibility of the execution strategy. In a database, the relational model is used to declare *what* data should be returned by a query; *how* the data are generated (i.e., the order in which data are scanned, joined, and filtered) is generally left to the database query planner. Pequod also uses a query planner, but takes additional guidance from the developer; a cache join both declaratively defines the output and specifies the exact execution strategy. The details of cache join execution are discussed in §5.3.

4.3 Deployment

Pequod is deployed in the communication path between the back-end servers executing application routines and the persistent storage service. Once deployed, the communication pattern of the application must be changed so that the cache is used to absorb reads that were formerly directed at the persistent store.

Pequod supports several common cache configurations, which are depicted in Figure 4.1:

- **Look-through.** The application first attempts to read from the cache. In the event of a cache miss, the application issues another read request to the persistent store, the results of which may then be stored in the cache for future access. The application writes data to both the cache and the persistent store.
- **Write-through.** The cache is inserted as a layer between the application and the persistent storage service. All reads and writes pass through the cache, which may block client requests while the persistent store is accessed.
- **Write-around.** The application directs all read requests to the cache, which may defer to persistent storage in the event of a miss. The application sends writes only to persistent storage. The cache monitors persistent storage for changes, allowing application writes to indirectly influence cache entries.

Pequod is largely agnostic to the deployment configuration chosen by the developer; cache joins execute identically in each, transforming data that are available in the cache into derived output and keeping that derived data fresh as changes occur. The supported configurations differ in the mechanisms used to populate the cache—that is, how data from the persistent store make their way into Pequod. These mechanical details are not pertinent to the core cache join algorithm, and we only mention the configuration options for context. We do not advocate a specific deployment configuration, and the prototype implementation supports each option—at least to the extent needed for proof-of-concept.[1]

[1]At present, the prototype implementation is missing a feature—the ability to distinguish between an empty data range and a missing data range—that is needed to support a true look-through deployment.

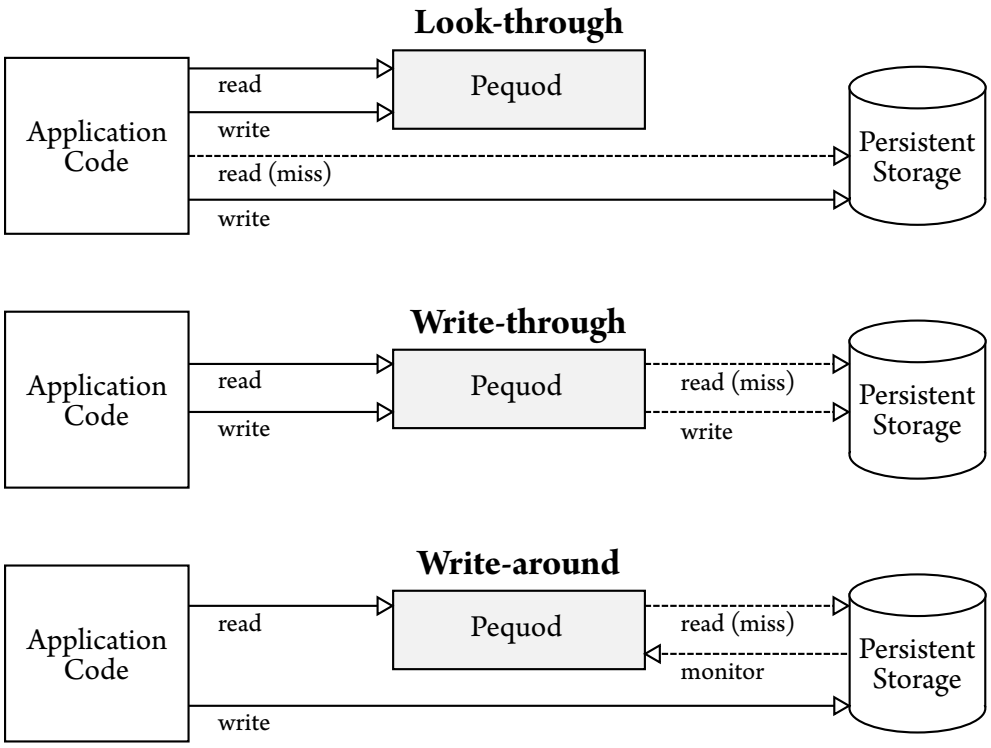

Figure 4.1: Three cache access patterns supported by Pequod. In all three cases, Pequod is made aware of every base data write for the purpose of keeping materialized data fresh.

5

Cache joins

This chapter covers the details of Pequod cache joins: how they are specified and used to generate derived data, how updates are made to derived ranges, how they are composed to create complex behaviors, and how they are parameterized to provide alternate semantics.

5.1 Specification

A cache join is a declarative statement that relates a range of output keys to a set of input key ranges, defining how output values are calculated from input values. Application developers can install new cache joins with the ADDJOIN command. The cache join can be used immediately; a subsequent SCAN of the output range will cause the queried range to be materialized on demand. By default, Pequod will maintain this range until it is evicted or removed. Thus, subsequent SCAN operations will return fresh data quickly.

The *join specification* defines the relational structure of the keys and how output key-value pairs are calculated from input key-value pairs. It has four parts:

1. An *output overlay* defines the format of output keys;
2. One or more *source overlays* select keys whose values are used to compute results and define the operators applied to these keys;
3. Optional *performance annotations* guide query execution (see §5.6);
4. and *slot definitions* tell Pequod how to unpack a key into component overlay values—for example, by looking for vertical bars, or by taking fixed numbers of bytes.

```
<cachejoin> ::= <key> “=” <sources> “with” <slotdefs>;
<sources>   ::= <source> | <sources> <source>;
<source>    ::= [<maintain>] <operator> [<update>] <key>;
<maintain>  ::= “push” | “pull” | “snapshot” <number>;
<operator>  ::= “check” | <boundedop> | “min” | “max” | “sum”;
<boundedop> ::= “copy” [<bounds>] | “count” [<bounds>];
<bounds>    ::= [“lbound” <number>] [“ubound” <number>];
<update>    ::= “lazy” | “eager”;
<key>       ::= <text> <keybody>;
<keybody>   ::= <text> | <slotname> | <keybody> <keybody>;
<slotname>  ::= “<” <text> “>”;
<slotdefs>  ::= <slotdef> | <slotdef> “,” <slotdefs>;
<slotdef>   ::= <text> “:” <number>;
```

Figure 5.1: The cache join grammar. This grammar closely resembles the grammar used in the prototype implementation; some keywords have been updated in this text for clarity. The definitions for text literals and numbers are omitted for brevity.

Figure 5.1 summarizes the grammar. For clarity, cache join examples throughout this text omit the slot definition portion of the specification in favor of a more readable delimited form (using ‘|’ and italicized slot names). The prototype implementation relies on explicit, fixed-length slot definitions to properly parse output and source key formats.

From the cache join specification, Pequod determines the data ranges involved in the view’s construction, the relational overlays to apply to the keys, how the view should be constructed and maintained, and how output values are produced. Pequod uses this information in forward execution of a cache join (§5.3) and in subsequent incremental maintenance tasks (§5.5).

For example, consider a simple movie database that stores information about movies (e.g., title and release date) and actors. We can define a cache join that produces some derived information, the number of movies an actor made in each year, by installing

```
filmcount|actor|year =
  check appearedin|actor|title
  count released|title|year;
```
(5.1)

The cache join produces output keys prefixed with `filmcount`. The format of the output key embeds other information—the values for named slots *actor* and *year*—that is gathered by performing a natural join of two source ranges. An output key is produced if the join's constraints are satisfied; specifically, if the common slots (just *title* in this case) are equivalent in the sources being joined. This join will produce a unique key-value pair for every actor-year combination. For comparison, the equivalent materialized view definition in SQL is:

```
CREATE MATERIALIZED VIEW filmcount AS
  SELECT appearedin.actor, released.year, COUNT(*)
    FROM appearedin, released
    WHERE appearedin.title = released.title
    GROUP BY appeardin.actor, released.year;
```
(5.2)

There are parallels between the two declarative statements. Each specifies the output content (the actor, year, and count of movies), the criteria for joining the two sources (equivalent title), and the computation that occurs (group inputs by actor-year and produce an aggregate value). The SQL definition is more verbose because the language is more complex, supporting many features from the relational algebra. The cache join language supports only one type of join from the relational algebra—the natural join, in which the intersected columns must be equivalent to produce an output tuple—so its syntax is much simpler. The join constraints are expressed by the overlapping slot definitions between sources. Grouping is accomplished by naming slots that must appear in the join's output overlay. In this case, a count is stored in `filmcount` keys with a unique combination of *actor* and *year*.

Given the cache join (5.1) and the cache contents

⟨appearedin|Tom Hanks|Apollo 13 ↦ ∅⟩
⟨appearedin|Tom Hanks|Forrest Gump ↦ ∅⟩
⟨appearedin|Tom Hanks|Philadelphia ↦ ∅⟩
⟨appearedin|Tom Hanks|Sleepless in Seattle ↦ ∅⟩
⟨appearedin|Tom Hanks|Toy Story ↦ ∅⟩

⟨released|Apollo 13|1995 ↦ ∅⟩
⟨released|Forrest Gump|1994 ↦ ∅⟩
⟨released|Philadelphia|1993 ↦ ∅⟩
⟨released|Sleepless in Seattle|1993 ↦ ∅⟩
⟨released|Toy Story|1995 ↦ ∅⟩

Pequod will produce the output

⟨filmcount|Tom Hanks|1993 ↦ 2⟩
⟨filmcount|Tom Hanks|1994 ↦ 1⟩
⟨filmcount|Tom Hanks|1995 ↦ 2⟩

Note that the values of the base data entries (`appearedin` and `released`) have no bearing on the output produced. In fact, they are empty. This is a function of the operators used in the join specification, `check` and `count`. The `check` operator indicates to Pequod that only a source's keys are relevant to the join; the values are ignored. The `count` operator produces an output value—in this case, the number of movies associated with a specific actor in a specific year—by counting key-value pairs. Thus, no source values are needed to produce the desired output.

The Pequod cache join grammar defines a small set of built-in operators. Aggregate operators, such as `count`, behave like SQL's aggregate functions, combining many sources into a single output. The aforementioned `check` operator tells Pequod to ignore source values, and the `copy` operator indicates that a source's value should be copied verbatim as an output value. The exact set of provided operators is not fundamental to the concept of cache joins; those mentioned in Figure 5.1 represent core functions that are currently built into the Pequod prototype implementation.

5.2 Twip and Newp revisited

The caching solutions for Twip and Newp (§3.1, §3.2) demonstrate that materialization is a powerful design pattern that comes at considerable implementation cost. By deploying Pequod, the applications could be written using straightforward commands. This section presents an updated approach to caching both applications based on Pequod cache joins.

As before, the Twip application will cache a combination of base and derived data. For example, the key-value pair ⟨`sub|ann|bob` ↦ `1`⟩ indicates that user `ann` is subscribed to user `bob`. (The values associated with `sub` keys are ignored in Twip, and are therefore irrelevant). In addition, the pair ⟨`post|bob|100` ↦ `Hi`⟩ would cache user `bob`'s tweet of `Hi` at time 100. The Twip developer would like to cache a pre-computed timeline for each user. The Twip *timeline cache join*

```
tline|user|time|poster =
  check sub|user|poster
  copy post|poster|time;
```
(5.3)

defines the value of cache key `tline|`*user*`|`*time*`|`*poster* as a copy of the value of key `post|`*poster*`|`*time* whenever `sub|`*user*`|`*poster* exists. In other words, Pequod will execute a natural join of `sub` and `post` keys and create a new `tline` when the poster is in the user's subscription list.

Note that the cache join specification is generic—it is not specific to any user. Pequod uses contextual information from the client request to materialize concrete ranges.

For example, given the cache contents

⟨sub|ann|bob ↦ 1⟩
⟨sub|ann|ken ↦ 1⟩

⟨post|bob|100 ↦ Hi⟩
⟨post|bob|110 ↦ Bye⟩
⟨post|jay|105 ↦ I don't like Ann!⟩
⟨post|ken|100 ↦ My cat is grumpy.⟩

the timeline cache join can derive the *set* of keys that constitute user ann's timeline

⟨tline|ann|100|bob ↦ Hi⟩
⟨tline|ann|100|ken ↦ My cat is grumpy.⟩
⟨tline|ann|110|bob ↦ Bye⟩

The application client can request the derived timeline range by issuing a SCAN command. For example, to request the portion of user ann's timeline since time 100, the client would issue the command

SCAN(tline|ann|100, tline|ann⁺) (5.4)

This request instructs Pequod to return key-value pairs in the interval [tline|ann|100, tline|ann⁺).[1] When this command is received by Pequod, the system checks whether the derived range is already present; if so, its contents are returned, if not, Pequod ensures that the inputs are cache resident, executes the cache join (restricting the derived range based on the context of the request), and starts tracking the inputs so that future cache modifications are reflected in the derived range on the next request.

The parameters of the client SCAN query limit the scope of automatic updates. For example, query (5.4) issues a request for an open-ended range (it has no specific upper bound). As a result, if any of the users that ann follows makes a new post after time 100, her cached

[1]The notation tline|ann⁺ represents the upper bound of the tline|ann range: [tline|ann, tline|ann⁺) contains exactly those keys starting with tline|ann. In the Pequod prototype implementation, this upper bound is implemented by the unsightly string tline|ann}.

timeline will be updated automatically. If instead the client had issued a more specific request, say for the interval $[$`tline|ann|100`, `tline|ann|110`$)$, Pequod would still update the derived timeline, but only for posts between time 100 and 110. Twip developers use open-ended intervals to ensure that user timelines are kept fresh as time moves forward. The details of cache join scoping and incremental maintenance are presented in §5.3 and §5.5.

Pequod executes cache joins using only input data that are cache resident. Further, Pequod can only maintain freshness of derived data if it is aware of changes that are made to its inputs post-execution. It is up to the application developer to ensure that Pequod is populated (and updated) with the data needed to produce meaningful results. Several deployment options for integrating Pequod with persistent storage are discussed in §6.5. Until then, we assume that Pequod contains the data needed to produce the desired cache join output.

With Pequod, the Twip application is simply constructed. New posts and subscriptions are written to the persistent store and cached in Pequod. Timelines are fetched from the cache by scanning `tline|`*user* ranges. The timeline cache join is used to derive these ranges on demand. Once produced, Pequod keeps the derived timelines fresh by monitoring the base data for changes and applying the appropriate updates.

The same strategy can be used for Newp. New articles are written to the persistent store, as are comments and votes. These data are also cached for fast access.

For example, Pequod might contain cached Newp articles, comments, and votes:

```
⟨article|bob|01 ↦ http://xkcd.com/327/⟩
⟨article|bob|02 ↦ http://en.wikipedia.org/wiki/Eddie_Kohler⟩
⟨article|ann|01 ↦ http://github.com/bryankate/pequod⟩

⟨comment|bob|01|100|bob ↦ This is the best site.⟩
⟨comment|bob|01|103|ken ↦ LOL⟩
⟨comment|bob|01|305|ken ↦ I like cats.⟩
⟨comment|ann|01|110|ken ↦ First post.⟩
⟨comment|ann|01|200|ann ↦ Serious replies only!⟩

⟨vote|bob|01|ken ↦ ∅⟩
⟨vote|bob|01|bob ↦ ∅⟩
⟨vote|bob|01|ann ↦ ∅⟩
⟨vote|bob|02|ann ↦ ∅⟩
⟨vote|ann|01|ken ↦ ∅⟩
⟨vote|ann|01|bob ↦ ∅⟩
```

In this scheme, an article is uniquely identified by both the user that posted the article and a sequence number. Together, they form the article identifier (*aid*), but are used separately in cache join specifications (as seen below).

To browse the contents of the article with *aid* `bob|01`, the Newp client would issue two commands in parallel:

```
GET(article|bob|01)
SCAN(comment|bob|01, comment|bob|01+)
```

But how should the application obtain the vote count for the article and the karma for each user? The client could fetch the `vote` data from the cache, but retrieving the base data and computing these values on each access is wasteful. Rather, Newp uses the *vote count cache join* and the *karma cache join*:

```
vcount|aid = count vote|aid|voter;

karma|author = count vote|author|seq|voter;
```
(5.5)

to compute and cache these values. Note that the same `vote` key-value pairs are used in both joins, but are assigned different overlays. In the karma join, the article identifier is

parsed as two slots, the author of the article and a sequence number. The vote count join interprets the same bytes as a single slot. The above cache joins filter the `vote` keys to produce aggregate values:

```
⟨vcount|ann|01 ↦ 2⟩
⟨vcount|bob|01 ↦ 3⟩
⟨vcount|bob|02 ↦ 1⟩

⟨karma|ann ↦ 2⟩
⟨karma|bob ↦ 4⟩
⟨karma|ken ↦ 0⟩
```

Thus in addition to fetching the article details and comments, the Newp application also fetches the article vote count with GET(`vcount|bob|01`). User karma cannot be fetched in parallel; the set of users that commented on the article is not known until the first round of queries is returned. At that point, the karma for all commenters may be fetched in parallel with GET(`karma|`*user*) queries. The vote count and karma cache joins simplify the Newp application relative to a strategy that computes the same values on demand. An extension to this approach that further simplifies the application code is discussed in §5.7.

Twip and Newp are just two examples of how Pequod cache joins can be used to remove complexity from Web applications. In general, the class of applications for which the cache join is useful is equivalent to that for which database-managed materialized views are advantageous. With Pequod, the Twip developer can achieve the same effect as the Twitter developers, but with less effort on the part of the application. A single declarative expression specifies how timelines are generated from scratch, a process we call *forward execution*, and how derived ranges are kept fresh through incremental maintenance. The end result is application code that looks and behaves like a database materialized view—compare (3.3) and (3.4) with (5.3) and (5.4)—but performs like an application-level cache.

5.3 Forward execution

This section describes the semantics and implementation of cache join execution. The focus is on forward query execution, which starts from base data. As we will shortly discover, generating a join's output from scratch involves potentially costly base data scans and metadata queries. We prefer, if at all possible, to perform most of the computation in the cache with incremental updates (§5.5). Nonetheless, we begin by describing forward execution precisely because it is used to bootstrap a join's output and lay the groundwork for incremental maintenance.

Once a cache join specification is installed, the application can start requesting data in the output range. A SCAN of this range will prompt Pequod to check if there exists up-to-date output that can be returned immediately. If not, the output is generated by a forward execution of the installed cache joins that cover the requested range. To illustrate the basics of cache join execution, we assume in this section that every cache join is executed from scratch, ignoring the results of prior executions. We also assume (until §6.3) a single thread of execution.

Pequod uses a nested loop strategy to execute join queries, iteratively joining the source ranges listed in the cache join specification. The goal of the algorithm is to generate key-value pairs that conform to the join's output overlay. To emit output key-value pairs, Pequod must construct a key, by filling each slot in the output range's relational overlay, and compute a value. The former is accomplished by establishing a *slot set* that is filled as the algorithm advances. To begin, the slot set is populated with information from the SCAN request. The request itself offers some context (establishes the initial assignments for some slots) and narrows the scope of the join execution (i.e., to a specific user). The slot set is progressively filled by scanning the source ranges and using the relational overlays of

the source keys to fill in the missing context. Ideally, the scope of the source range scan narrows with each nested loop iteration (because more of the slot set is defined). When Pequod scans the *primary source* (the last source listed in the join specification), the slot set is checked for completion (that all slots are assigned); if complete, an output key is constructed from the set and the value is computed using the operator.

For example, consider a forward execution of the Twip timeline cache join for user `ann`'s timeline check (reproduced below):

```
tline|user|time|poster =
  check sub|user|poster
  copy post|poster|time;

SCAN(tline|ann|100, tline|ann+)
```

To populate the initial slot set, Pequod will parse the keys that define the range bounds of the SCAN request according to the output range's format (in this case, `tline|ann|100` and `tline|ann+` are parsed using the overlay `tline|`*user*`|`*time*`|`*poster*). Any slot that is commonly defined in *both* keys can be added to the slot set. In this case, the initial slot set is $\{user \mapsto \texttt{ann}, time \mapsto \varnothing, poster \mapsto \varnothing\}$. It is correct to omit an assignment of *time* in the initial slot set even though a time was given in the SCAN request. The slot set represents values that must be common of *all* keys selected by the join (that contain that slot). If it were filled with the value 100, the join would only emit posts at time 100. Thus, the slot is initially left blank, and is assigned a value as the `post` source is processed.

Though not used as a join constraint, the time is used to define *containing ranges* for each scan of the key-value store. A containing range is effectively the inverse of a slot set: given a slot set, a source overlay, and the requested output key range, Pequod can calculate a minimal range of source keys that might affect the scan's results. For example, given `ann`'s

```
1  compute_cache_join(first, last, join):
2    ss := join.slotset(first, last)
3    process_sources(first, last, join, ss, 0)
4
5  process_sources(first, last, join, ss, srcidx):
6    src := join.source(srcidx)
7    [key⁻, key⁺) := ss.containingrange(src, first, last)
8    for each ⟨key ↦ val⟩ where key ∈ [key⁻, key⁺) and key matches src.overlay(ss):
9      ss' := ss.fillslots(key)
10     if not join.isprimary(srcidx):
11       process_sources(first, last, join, ss', srcidx + 1)
12     else:
13       emit ⟨join.outputkey(ss') ↦ src.operator(val)⟩
```

Figure 5.2: Pseudo-code for the forward cache join execution algorithm, presented in a recursive form. The list of sources is processed in order, iterating over the keys in that source's containing range. The slot set is updated to reflect the information in each key as it is processed. When the primary source is reached, the output key-value pairs are constructed and stored in the cache.

timeline request and the slot set $\{user \mapsto \texttt{ann}, time \mapsto \varnothing, poster \mapsto \texttt{bob}\}$, the minimal containing range for the `post` source would be $[\texttt{post|bob|100}, \texttt{post|bob}^{+})$. Any post outside that containing range would either not match the required *poster*, or not map to an output key in the requested output key range. Since Pequod should support any application and provide general key-value cache semantics, some care is taken to handle each query correctly. For example, Pequod correctly implements queries like SCAN(`tline|ann|100`, `tline|bob|200`) and SCAN(`tline|a`, `tline|b`) that cross multiple timelines. Correct and minimal containing ranges are generated in each case. But even with containing ranges, the algorithm must compare the source range keys with the relational overlay. As a schema-free key-value store, Pequod might have keys in the range that do not match the current source overlay.

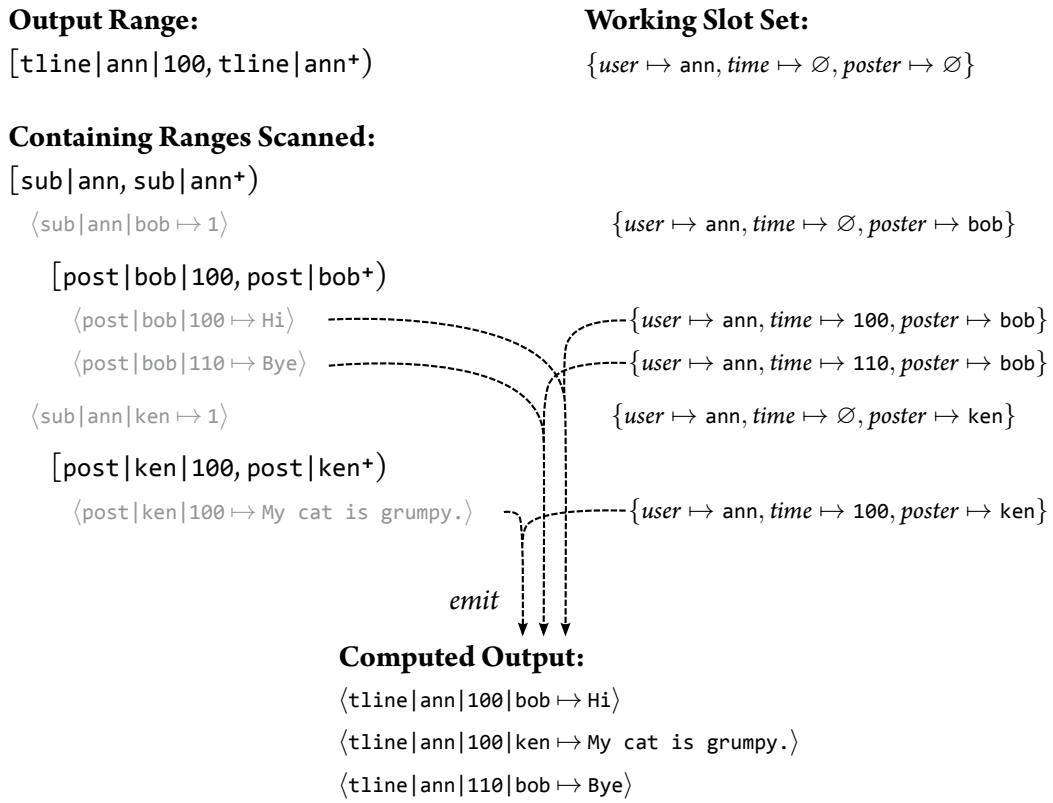

Figure 5.3: An example of forward execution of the Twip timeline cache join. As the algorithm progresses through the containing ranges, the slot set is filled by matching the scanned keys. Output key-value pairs are emitted when the slot set is complete.

Figure 5.2 outlines the forward execution algorithm in pseudo-code. The steps that the algorithm takes are described below and the contextual state (the slot set) at each step are depicted in Figure 5.3. The algorithm begins by scanning the first source (ann's subscription list) using the containing range [sub|ann, sub|ann+). Keys from this range are selected and the algorithm descends into a nested loop for each subscription. For each invocation in the nested loop, the slot set is modified to reflect the definition of *poster* in the subscription key. With this slot assigned, each nested loop selects posts from a single poster by iterating over keys in the containing range [post|*poster*|100, post|*poster*+). As each post is selected, the slot set is updated with the appropriate definition of *time*. Being

the primary source, as each key is selected the slot set is checked for completion and used to emit an output key in the correct format. The value associated with each post is copied as the output value, completing the key-value pair.

5.4 Partial, dynamic materialization

In conventional databases, materialized views are constructed in full; that is, the entire output relation is generated and stored. In terms of Twip, this would be equivalent to materializing the timeline of every registered user for all time. Caching that amount of data is not only technically challenging, it is not useful; a Twip user rarely requests archival data, and not every user is actively checking his timeline.

Pequod is designed to support *partial* materialization. Like a view definition in SQL, a cache join specification declares the entire output relation. However, resource constraints necessitate that Pequod keep only a portion of the output keyspace resident in the cache at any given time. But how are portions selected for materialization? As described in the forward execution example above, portions of the output range are materialized *dynamically* based on client requests. Pequod employs a simple strategy to guide dynamic materialization: output is generated on demand using the parameters of the client request and then maintained until it is evicted by the system or removed by the client. Unless otherwise specified (§5.6), all Pequod cache joins are materialized partially and dynamically according to this simple policy.

5.5 Incremental maintenance

Once generated, derived data ranges are kept consistent with their associated source ranges with incremental maintenance operations. To support incremental maintenance,

Pequod establishes *dependencies* between output and source ranges in the cache. When a source range is modified, Pequod takes the appropriate action to update the derived ranges.

Pequod uses metadata, referred to as a *join status range*, to track the state of generated output. A join status range describes the range bounds covered by the join execution and tracks its validity (i.e., whether the range has expired or has been invalidated). If a request arrives for a range that is partially covered by previously generated output (as determined by querying the installed join status ranges), Pequod will use forward execution only to fill the gaps.

Tracking dependencies is relatively straightforward; metadata describing the generated output (join status ranges) are linked with metadata (called *updaters*) associated with the exact containing ranges used to produce the output. Updaters store *update contexts*, the minimal set of information needed to perform an incremental update in response to a source modification. Each context contains a slot set and links to a cache join specification and join status range. When a modification to a source range occurs, the updaters that overlap with the modified range are invoked. When an updater is invoked, it is passed information about the change that occurred: the affected key, its old value, its new value (if any), and the operation type (PUT or REMOVE). Using the change information and the stored contexts, the updater can update any previously generated outputs by inserting, removing, or updating key-value pairs. The exact updates performed are defined by the operator attached to the modified source range in the cache join specification (e.g., `copy`, `sum`).

For example, if the request SCAN(`tline|ann|100`, `tline|ann+`) is used to fetch `ann`'s Twip timeline, Pequod would install an updater that covers the source range for her subscription list, [`sub|ann`, `sub|ann+`), and an updater for *each* `post` containing range that was

scanned (one for each user that ann follows). When bob inserts a new post

⟨post|bob|500 ↦ My cat's breath smells like cat food.⟩

Pequod should generate a new key for ann's timeline:

⟨tline|ann|500|bob ↦ My cat's breath smells like cat food.⟩

While processing the PUT operation for the new post, Pequod looks for installed updaters. An updater with update context

join status range: [tline|ann|100, tline|ann⁺)
slot set: {*user* ↦ ann, *time* ↦ ∅, *poster* ↦ bob}
operator: copy

is located and invoked. With this update context, Pequod has all the information it needs to update the output. The join status range indicates that the output key is in the tline table, the slot set identifies the *user* as ann and the poster as bob, and new post key provides the *time* as 500. The copy operator is used to define the value of the output (by copying the value of the input).

Figure 5.4 outlines a cache join execution algorithm that is updated to reflect:

- **Partial materialization.** The validity of previously generated output is tracked with join status ranges. The cache join is only executed for subranges of the requested output that are missing or invalid.
- **Incremental maintenance.** Pequod installs updaters into the source ranges used to generate the output. Updaters are garbage collected when dependencies are broken (e.g. by eviction or explicit removal of the source or output range).
- **Nested joins.** The output of one cache join can be used as the input to another. As such, Pequod must check that all source ranges are valid before iterating over the computed containing range. Nested joins are discussed further in §5.7.

```
1  compute_cache_join(first, last, join):
2    for each subrange [x⁻, x⁺) ⊂ [first, last) where join status for [x⁻, x⁺) ≠ VALID:
3      js := new join status for [x⁻, x⁺)
4      ss := join.slotset(x⁻, x⁺)
5      process_sources(x⁻, x⁺, join, ss, js, 0)
6      install js as VALID
7
8  process_sources(first, last, join, ss, js, srcidx):
9    src := join.source(srcidx)
10   [key⁻, key⁺) := ss.containingrange(src, first, last)
11   srcjoin := look up join for [key⁻, key⁺)
12   if srcjoin ≠ ∅:
13     compute_cache_join(key⁻, key⁺, srcjoin)
14   install updater with context {src, js, ss} into [key⁻, key⁺)
15   for each ⟨key ↦ val⟩ where key ∈ [key⁻, key⁺) and key matches src.overlay(ss):
16     ss' := ss.fillslots(key)
17     if not join.isprimary(srcidx):
18       process_sources(first, last, join, ss', js, srcidx + 1)
19     else:
20       emit ⟨join.outputkey(ss') ↦ src.operator(val)⟩
```

Figure 5.4: Pseudo-code for the cache join execution algorithm, modified to query and install metadata as necessary to support partial materialization, incremental maintenance, and chained joins. Important updates to the algorithm in Figure 5.2 are shown in black.

Tracking dependencies with updaters allows Pequod to make incremental modifications to derived data. That is, the metadata determines *what* should be updated. But *when* should the update take place? By default, Pequod uses an *eager* maintenance policy in which updates are applied to derived ranges as soon as possible after the source range is modified. The eager policy optimizes read operations; maintenance tasks are subsumed by write operations, so reads return fresh data without any additional effort. This strategy works well for read-heavy workloads (like Twip and Newp), where the trade-off of diminished write performance makes sense.

However, there are some cases for which eager maintenance is not desirable, even in cachable workloads like Twip. For example, how should a timeline be updated to reflect a new subscription? Under the eager policy, Pequod would back-populate the subscriber's cached timeline with the posts of the user named in the new subscription. As a result, any request that falls within the generated output range would return up-to-date results that reflect the new subscription. But what if historical timeline requests are infrequent? Under the assumption that the vast majority of timeline requests are used for refreshing (fetching posts since the last check), back-populating cached timelines to reflect subscription changes is unnecessary and wasteful.

As a result, Pequod also employs a *lazy* update policy for some updates. Under this policy, a source modification will not generate an immediate update to derived output. Rather, the output ranges are marked to indicate that an update is pending and the context of the modification is recorded. The update is then lazily applied to the output range according to the scope of subsequent requests. For example, consider the lazy update strategy as applied to Twip subscription changes. If user `ann` has a cached timeline that covers the range $[\texttt{tline|ann|100}, \texttt{tline|ann}^{+})$ and she adds a new subscription to `bob`, Pequod will attach the context of the update (the key-value pair inserted) to her join status range. If she then checks her timeline at time 5000, the update is applied only for the range $[\texttt{tline|ann|5000}, \texttt{tline|ann}^{+})$; the update context remains in place for the range $[\texttt{tline|ann|100}, \texttt{tline|ann|5000})$ and will be applied if queried.

The eager and lazy maintenance policies used by Pequod have competing performance goals and costs. The eager approach optimizes for speed of read operations at the expense of slower writes and potentially wasted space (to store derived data that are never read). The lazy approach saves space and reduces computation on writes, but complicates reads

with partial updates. Application developers can consider this trade-off and choose policies to suit the needs of the application. The next section describes how a developer can influence the behavior of cache join execution and maintenance by manipulating the join specification.

5.6 Tuning

In Pequod, the cache join specification defines *what* data is produced in a computation and *how* the join is executed and maintained. In addition to the inputs, output, and operator, a developer can provide performance annotations that change the behavior of the execution algorithm. In this section we describe how developers can change the performance characteristics of Pequod through careful cache join construction and annotation.

Selectivity

The first opportunity for optimization comes from the design of the relational overlays for the data ranges used as inputs to a cache join. The overlay not only defines the slot decomposition of a key, but the order of the slots within the key. Since key-value pairs in Pequod are stored and accessed according to a canonical (lexicographic) order, a thoughtful developer can maximize performance by adjusting slot order.

It is important to reduce the scope of each source range scan during cache join execution. The exact ranges scanned are determined by the containing ranges computed from the working context (the slot set and original request). To narrow the scope of a scan, the developer should order the slots in a relational overlay such that the filled slots appear first.

Consider the two Twip timeline cache join specifications depicted in Figure 5.5. Both joins produce output in the same format, but differ in their use of base data; subscription

(a)

```
tline|user|time|poster =
    check sub|user|poster
    copy post|poster|time

SCAN(sub|ann, sub|ann+)
SCAN(post|bob|100, post|bob+)
SCAN(post|ken|100, post|ken+)
```

(b)

```
tline|user|time|poster =
    check follow|poster|user
    copy post|poster|time

SCAN(follow, follow+)
SCAN(post|bob|100, post|bob+)
SCAN(post|ken|100, post|ken+)
```

Figure 5.5: Two versions of the Twip timeline cache join and the resulting source range scans issued during join execution. Version (a) is more selective, with the scope of its first scan narrowed to a single user. Version (b) requires a full table scan because the *poster* slot is unassigned when the scan is issued.

lists are used to express relationships between users in (a), whereas (b) uses follower lists. Aside from the key prefix, the only difference between these two ranges is the order of the slots in their relational overlays. Subscription lists are ordered by subscriber, and follower lists by subscribee. However, the cost to execute these joins is quite different. Below the join specifications in Figure 5.5 are the containing ranges that Pequod scans to produce output for user ann's timeline check (5.4). The only difference is in the scope of the first scan (the one used to identify ann's subscriptions). In (a), the scope is restricted to a relatively small subrange of the sub table. But in (b), the entire follower table is scanned; the few follower|*poster*|ann keys are discovered at an enormous cost. For context, if ann were a typical Twitter user, the subscription list scan in (a) would evaluate hundreds of keys. In contrast, the follower list scan in (b) would evaluate billions of keys (every relationship in the social graph) [27].

This example demonstrates how slot arrangement in a source range's relational overlay can affect the *selectivity* of range scans: that is, the number of key-value pairs evaluated while scanning. We use an extreme case, that of an unassigned slot preceding an assigned slot, to

accentuate the effect. The developer should not only take care to avoid this arrangement, he should consider the relative selectivity of assigned slots and arrange them to optimize scan performance, if possible.

In addition to optimizing for selectivity within source overlays, the developer should also consider selectivity when determining the order in which source ranges are processed. Scanning the smallest ranges first minimizes the total number of pairwise key-value comparisons required to execute the cache join. This is a well known strategy for set intersection and is used in relational database query optimizers. Some database query optimizers use estimates of selectivity, based on table statistics, to reorder operations within a query at execution time. Pequod does not have an online query optimizer; the system executes a join according to its specification (which is statically defined). Thus, when joining multiple sources, a developer can order those sources in a meaningful way using domain knowledge.

We illustrate this point with an example from Twip. Celebrities that join Twip are put into the difficult position of wanting to connect with fans by following them but not wanting to be inundated with their mundane tweets. Thus, the Twip developer might want to implement a special feature for celebrity users: an exclusive, celebrity only timeline that is free of plebeian content. He might implement it with a new join

```
exclusive_tline|user|time|poster =
  check celeb|poster
  check sub|user|poster
  copy post|poster|time;
```
(5.6)

that generates the exclusive timeline. This join augments the original timeline join with another source (a list of celebrities). To optimize the join execution, the developer would want to put the most selective sources first. A scan of the `sub` range will likely yield thousands of key-value pairs for a celebrity user, but there are likely only hundreds of celebrities

in the service. It would be better to push the `celeb` scan to the beginning of the join so that fewer `sub` keys match and, in turn, fewer `post` ranges are scanned. It is not guaranteed that the earlier sources will be more selective when queried, but it is often possible to make assumptions about the relative selectivity of sources using domain knowledge.

Performance annotations

Pequod offers several performance annotations in the join specification grammar. *Maintenance annotations* determine how a cache join's output will be maintained by Pequod after it is computed. A single maintenance annotation is allowed in each cache join. There are three options:

- **push.** A `push` annotation signifies that the cache join output will be maintained with incremental updates by Pequod until it is removed from the system (e.g., by eviction). Push joins are the functional equivalent of materialized views in a relational database. This is the default maintenance behavior for cache joins, and is applied in the absence of an overriding annotation.
- **pull.** The `pull` annotation disables join maintenance. Output is generated on demand and is valid for a single response. Pull joins are the functional equivalent of queries and non-materialized views in a relational database. Due to the high cost of generating output anew for each query, pull joins should be used sparingly. In a typical caching scenario where data is requested multiple times, push (view maintenance) typically has lower computation costs than pull. We thus expect most Pequod cache joins to use push. Pull joins can be useful, however, if a developer knows that data is unlikely to be re-requested, so the computation and space required for continued maintenance would be wasted.

– **snapshot.** A cache join annotated with `snapshot` behaves like pull join with the exception that generated output is valid for more than one response. Specifically, the output is valid for a finite amount of time, after which it expires and will need to be regenerated. Snapshot joins allow the cache to serve data with a bounded staleness. They can be used as an alternative to (unnecessary) incremental maintenance for data that do not have strict freshness requirements—for example, the listing of articles on the front page of Newp.

A second type of annotation allows a developer to choose the incremental update policy that is applied to the join's sources. This annotation is meaningful only for push joins, as pull and snapshot joins do not require maintenance after execution. The two annotation options are `lazy` and `eager`, corresponding to the two update policies outlined in §5.5. When Pequod processes a command that could trigger an incremental update (PUT or REMOVE), the relevant updaters are located in the range metadata. For eager push joins, Pequod invokes the updaters immediately, potentially modifying the output range. For lazy push joins, Pequod forgoes updater invocation and instead logs the key-value pair that triggered the update. The logged update is applied when the client queries the output range.

A lazily updated push join is a bit of a contradiction. Push joins exist to optimize query latency by pre-computing outputs. Logging updates and applying them at query time partially negates this optimization. Pequod provides the `lazy` annotation as a storage optimization; the application can potentially save space—for example, by avoiding excessive backfilling of a user's timeline after a subscription change (§5.5)—but accepts higher query latencies in trade. As such, only secondary sources may be marked with the `lazy` and `eager` annotations. Primary sources are always updated eagerly. No space savings are possible when the lazy update policy is applied to the primary source. Each key-value pair that trig-

gers a potential update must be logged into each updater for future application. This requires, to first order, the same amount of space as generating the output eagerly.

We describe one final annotation that allows a developer to influence the order in which sources are processed. In general, Pequod processes the secondary sources in specification order before processing the primary source and executing the operator. The *filter* annotation, which can be applied to any secondary source, indicates that the source should be processed as late in the join execution as possible. For example, consider an alternate form of the Twip timeline cache join

```
tline|user|time|poster =
  check filter sub|user|poster
  pull copy allposts|time|poster;
```
(5.7)

that uses a global list of all tweets, `allposts`, rather than the individual user `post` ranges. Normally, the `sub` containing range would be scanned first, and for each key enumerated by the scan, the `allposts` range would be scanned (having defined the *poster*) slot. However, the `allposts` overlay is not optimized for this type of scan. The slot *time* is listed first, making these scans extremely inefficient.

It is more efficient to first perform a single scan of `allposts` and then select only the posts that are relevant to the user by checking for a `sub` key that matches the *poster* slot of each enumerated key. The `check` is still applied before emitting output keys with `copy`, but the order of the containing range scans is altered to best match the relational overlay. Using this cache join for normal Twip operation would be a disaster: the `allposts` range would contain thousands of tweets per second, and the majority of the `checks` performed on the `sub` range (each requiring its own tree lookup) would not yield any results. However, this strategy is useful when the initial scan produces few (or zero) results. This is the case for a Twip optimization that deals with celebrity users, presented in §5.7.

5.7 Composition

Pequod allows developers to *compose* cache joins to create compound in-cache computations. In this section we explore two techniques for composing joins, chaining and interleaving, that developers can employ to handle more complex transformations and optimize application performance.

Chaining

One benefit of using data ranges in the cache join specification is that the joins are agnostic to the provenance of the key-value pairs they process. At the level of the cache join abstraction, it makes no difference if a source range consists of data provided by the application client or computed by another cache join. Thus, cache joins can be *chained* together, with the output of one join used as the input to another.

Consider an example from Newp: if the developer wants to display a list of users with karma scores over a certain threshold, she could add a chained join that uses the computed karma as a source. The `elite` join

```
// compute karma from votes, same as above
karma|author = count vote|author|seq|voter;

// use computed karma to produce a list of elite users
elite|author = copy lbound 100000 karma|author;
```
(5.8)

creates a data range that identifies this subset of users with karma scores greater than or equal to 100,000. This join filters the output of the karma cache join (by applying a lower bound to the copied results) and arranges it for quick retrieval with a single SCAN(`elite`, `elite+`) request.

No special syntax is required to chain joins; the developer simply uses the same data range in the specification of two joins (as the output of one join and the input to another).

During join execution, Pequod will check if the current join's inputs are produced by another join. If so, it will ensure that the input range is valid, executing the associated join if necessary.[2] In this way, Pequod will work backward to ensure that the correct output is generated at the end of the chain. Pequod will also work forward to ensure that changes to the base data at the beginning of the chain are propagated throughout.

Pequod does not limit the length of a cache join chain. However, developers need to be mindful of the implications of chaining. Changes to base data will propagate through the chain, causing cascading invalidations and updates that affect write performance. Additionally, developers need to be aware of performance annotations that are defined in predecessor links. For example, adding a `pull` or `snapshot` annotation to the karma cache join will affect how the `elite` range is created and maintained.

Interleaving

Pequod allows developers to *interleave* cache joins, with the output of many joins producing keys in the same range. Interleaved cache joins can be categorized into two groups: *split joins* produce key-value pairs that are semantically indistinguishable, and *mixed joins* produce key-value pairs that are semantically different. For example, the cache joins

```
allimg|name|size|date = copy albumpic|album|name|date|size
allimg|name|size|date = copy attachment|jpg|date|size|name
```
(5.9)

split the production of `allimg` key-value pairs between two joins, each handling image data from different parts of an application. The result is a single range, populated with identically

[2]It is an error to install a cache join chain that contains a cycle. Though the prototype implementation does not presently check for this condition, it could be implemented with a straightforward graph analysis.

formatted key-value pairs. In contrast, the joins

```
sample|time|old|x|y = copy reading|user|time|x|y
sample|time|new|x|y|loc = copy sensor|user|time|loc|x|y
```
(5.10)

mix semantically different keys into a single range (`sample`). In this case, the developer can SCAN the cache for a range of times and retrieve sample points in two formats (perhaps from different versions of the same application).

When might a developer choose to split a single cache join into multiple joins with the same output format? Consider the Twip timeline cache join (5.3): on average, new tweets will be propagated to hundreds of users [6]. However, what happens when a celebrity tweets? That post could be delivered to millions of followers. Executing the updates to millions of timelines is not necessarily prohibitively expensive, especially if updates are applied asynchronously and in parallel (§6.3). But storing the same tweet in millions of timelines is a resource drain—even if the value is shared by all key-value pairs, millions of unique keys must be stored. In addition, storing all of these key-value pairs makes reads slower. Pequod is an ordered store, so the complexity of a lookup is $O(\log n)$, where n is the number of key-value pairs. Thus, there is a potential problem when every celebrity tweet grows n by millions of entries.

To address this problem, the developer can split the timeline computation by installing multiple cache joins that produce output in the same range. This solution exploits the dichotomy that exists in the user base by handling celebrity tweets in a different way: the tweets of regular users are pushed to their followers' timelines while those of celebrities are merged in when a timeline is queried. The updated set of joins illustrates how this works:

```
// non-celebrity, same as above
tline|user|time|poster =
  check sub|user|poster
  copy post|poster|time;

// celebrity
tline|user|time|poster =
  check filter sub|user|poster
  pull copy ctline|time|poster;

ctline|time|poster = copy cpost|poster|time;
```
(5.11)

The biggest difference between these joins is the maintenance policy: the non-celebrity join is materialized into the store and kept fresh (push) while the celebrity join is executed on demand (pull). When a user queries his timeline, both joins are executed before any output is returned. On average, the bulk of the response is pre-generated by the push join and is ready for reading. The pull join will be executed on each query, but it is expected to finish quickly because there are relatively few celebrities. Thus, the developer has traded slightly slower timeline queries for a reduction in store size.

There are a few things to note about the handling of celebrity data. First, celebrity tweets are stored in a special `cpost` range so that they are not copied into a user's timeline when the push join is executed. Second, a special `ctline` range holds all of the celebrity tweets, sorted by time. This rearrangement exists so that the pull join can be optimized. Since there are relatively few celebrities and timeline queries cover a small time interval, this global celebrity tweet range is scanned first and then filtered by the user's subscriptions. In most cases, scanning the `ctline` containing range will return few results, if any. This minimizes the number of lookups in the `sub` range, often eliminating them entirely.

Pequod is not the only system capable of combining execution strategies to produce results. A similar timeline construction can be implemented in SQL by requesting the `UNION` of a materialized view that holds the pre-computed non-celebrity portion and an on de-

mand query that generates celebrity portion. In addition, Silberstein et al. [38] suggest a push-pull strategy for optimizing feed-based applications. Their online algorithm makes this decision individually for every user in the system based on observed usage patterns. At query time, the feed for a user is constructed by combining materialized and freshly generated content.

Mixing is best demonstrated with an example from Newp. To read a Newp article, a client must fetch (1) the article itself, (2) the article's vote count, (3) all comments on that article, and (4) the user karma for the author of each such comment. This might involve many commands; for example, to fetch article `bob|01`:

(1) GET(`article|bob|01`)
(2) GET(`vcount|bob|01`)
(3) SCAN(`comment|bob|01`, `comment|bob|01+`)
(4) for each returned comment, GET(`karma|<commenter>`)

Although the first three commands can proceed in parallel, the last requires commenter IDs known only after step (3). By mixing joins, this interaction can be reduced to a single command. The revised Newp cache joins

```
// vote count and karma, same as above
vcount|aid = count vote|aid|voter;
karma|author = count vote|author|seq|voter;

// mixed joins
page|aid|a = copy article|aid;                                                    (5.12)
page|aid|v = copy vcount|aid;
page|aid|c|cid|commenter = copy comment|aid|cid|commenter;
page|aid|k|commenter =
  check comment|aid|cid|commenter
  copy karma|commenter;
```

allow the application to fetch the article from Pequod in its entirety with the command

SCAN(`page|bob|01`, `page|bob|01+`)

request. The results are grouped with the same common prefix, `page|bob|01`, but have

different relational overlays beyond that prefix and different value types. For example, with cache contents

```
⟨article|bob|01 ↦ http://xkcd.com/327/⟩

⟨comment|bob|01|100|bob ↦ This is the best site.⟩
⟨comment|bob|01|103|ken ↦ LOL⟩
⟨comment|bob|01|305|ken ↦ I like cats.⟩

⟨vcount|bob|01 ↦ 3⟩

⟨karma|bob ↦ 4⟩
⟨karma|ken ↦ 0⟩
```

the mixed page joins will produce the output

```
⟨page|bob|01|a ↦ http://xkcd.com/327/⟩
⟨page|bob|01|c|100|bob ↦ This is the best site.⟩
⟨page|bob|01|c|103|ken ↦ LOL⟩
⟨page|bob|01|c|305|ken ↦ I like cats.⟩
⟨page|bob|01|k|bob ↦ 4⟩
⟨page|bob|01|k|ken ↦ 0⟩
⟨page|bob|01|v ↦ 3⟩
```

Each key-value pair in the response is essentially a tagged union that can be parsed into component data by the application using the delimiters that appear in the keys (a, c, k, v).

The mixed joins are relatively simple, a straightforward application of chaining and copying. Of course, this type of composition trades space for improved performance. In this case the trade-off is warranted; a typical article has hundreds of comments but may be read hundreds of thousands of times (especially if it is linked on the site's front page).

This feature of cache joins has no direct analog in relational database materialized views. The relational model requires that each row in a table consist of the same set of columns. Pequod is flexible in that it can apply a schema (in the form of a relational overlay) when needed to perform a computation, while supporting schema-free queries. This feature is not unique to Pequod; in theory, other systems with schema-free data models, such as

those based on column families, could produce mixed output. However, Pequod requires that the columns in each tuple appear in a fixed order (defined by the relational overlay).

5.8 Discussion and limitations

In this section we discuss the limitations of our design choices and highlight the merits of alternative designs.

Ambiguous joins

Pequod trusts the developer to install error-free joins that are meaningful to the application. There are joins that are technically correct (they can be executed by Pequod) but produce incorrect or ambiguous results. For example, the Twip timeline cache join variant

```
tline|user|time =
  check sub|user|poster
  copy post|poster|time
```
(5.13)

lacks the *poster* slot in the output overlay. This join produces undefined results when there are two or more posters that tweet at the same time. A corresponding database query would produce one tuple per relevant tweet. But Pequod values are strings, not tuples, and the `copy` operator is not capable of combining multiple values into a single string.

It is not necessarily appropriate to reject such joins out of hand; perhaps the application ensures that the *time* slot is unique. Thus, Pequod's users are left responsible for avoiding ambiguous cache joins, either by preventing output collisions or by creating operators with well-defined behavior. Of course, there is also room for improvement in Pequod; perhaps the system could could provide debugging annotations that guide the runtime behavior of the system, producing an error when a conflict occurs.

Enhanced overlays

The cache join grammar in Figure 5.1 is sufficiently expressive for the purposes of Twip and Newp, but some computations are difficult or impossible to define. One limitation is that values are not used as first-class entities in join specifications; for example, a value cannot be named and used as a slot in an output key, nor can an output value be constructed from slot definitions. We illustrate these use cases and explore possible grammar extensions with two quick examples from Newp.

First, consider an update to the `elite` karma join (5.8). If the developer wants to maintain a *sorted* list of elite users, he could install a slightly modified join

```
elite|kval|author =
  copy lbound 100000 karma|author as kval;
```
(5.14)

that uses the *value* of each scanned key-value pair, labeled *kval*, to fill a slot in the output keys.

Next, consider a developer who wants to keep track of the user with the highest karma. The `karma` range is not sorted according to karma score, so it would be necessary to SCAN the entire range to find the highest score. Instead, he might install a cache join

```
maxkarma : author|maxval =
  max karma|author as maxval;
```
(5.15)

in which the output of the operator is explicitly named, *maxval*, and is used to construct a *compound value* for the output key. With this syntax there is always a single, well-defined key-value pair associated with this computation. However, this computation is ambiguous; the value of `maxkarma` is undefined if two or more users have the same karma score. In our prototype implementation, the value is overwritten with the latest computed value or update.

Naturally, if we allow the developer to name values as a whole and construct compound values as output, we should also extend the relational overlay to input values. For example, if the max karma join were used in a chain, the chained join would likely need to interpret the compound value produced in the previous step.

Operators

Pequod should be made extensible with support for user-defined operators; currently, only a handful of built-in operators are available. For example, users could add operators that apply a mathematical function to, categorize, filter, or sample cache data. However, there are some restrictions on operator functionality imposed by Pequod:

1. Operators make local decisions; they are passed a single key-value input pair and the existing key-value output pair (if it exists) and are expected to produce an output based on this information alone.
2. Operators are stateless; they may be invoked as part of a forward execution or in an incremental update. No state information can be stored or retrieved when the operator is invoked.
3. Operators are deterministic; built-in operators make no random decisions, though this is not strictly forbidden.

With user-defined operators and joins composed as an acyclic graph, Pequod begins to resemble streaming query [12] and dataflow batch processing systems [24]. While similar, these tools are designed for different purposes; streaming queries transform live streams of information in a producer-consumer setting and batch processing systems are designed to process data in phases.

View selection

In Pequod, the *view selection* problem, determining which ranges should be materialized to improve performance, is solved by a combination of human input and runtime policies. The developer installs cache joins to define the superset of materialized ranges. However, as a cache with limited space, Pequod will dynamically materialize only a portion of the installed joins' output range. Pequod use a policy to determine which portions of the full view should be materialized and kept fresh (for speedier retrieval) and which should be generated on demand (to save space).

The default policy—which materializes portions as they are requested and keeps them fresh as space allows—treats all requests equally and assumes there will be enough space after eviction to store the output. This policy is simple, but it could lead to excessive churn when the cache is operating at capacity. There are, of course other policies; the cache could materialize and maintain the most frequently requested ranges. Or, perhaps there is an application-specific policy that could be communicated to Pequod. For now, the simple policy works well enough to leave such extensions as future work.

6

Distribution

Pequod operates as a distributed system, with many *cache nodes* cooperating to perform cache functions. In this chapter we extend the cache join abstraction to function in the context of a distributed system. Of particular importance are: how data ranges are stored across many cache nodes and accessed by clients, how cache joins are executed in this distributed setting, how derived data are kept fresh, how data are kept consistent across the system, and how additional resources are used to scale the system.

6.1 Partitioning

Pequod is designed to handle datasets that are larger than the available memory of any single machine. Pequod pools the available memory of many machines and manages their resources on behalf of the application. The interface presented to clients is that of a single store, concealing the actual location of key-value pairs.

A traditional key-value cache, which presents a hash table abstraction, will scale with ease; there is no requirement for sequential range scans, so the cache need only support random access. The key-value pairs are independent of one another and can be stored on any node. However, Pequod is an ordered store, so the key-value pairs must be logically grouped. That is, an application should be able to request all keys in a specified range without contacting too many machines in the cluster.

A distributed application-level cache uses a partitioning scheme to store and retrieve key-value pairs across many cache nodes. A good partitioning both provides a well-known location for each pair and spreads the system load across all available cache nodes. A well-known location is critical for reducing the overhead of cache operations; targeted communication avoids flooding cache nodes with superfluous requests. Load balancing improves system performance by minimizing hot-spots and by raising the effective working set size.

The partitioning scheme in Pequod needs to support sequential scans of arbitrarily sized ranges. As such, partitioning in Pequod is accomplished by *segmenting*: breaking a single range into multiple smaller ranges and storing each on a different cache node. Data segmenting is transparent to the application client; Pequod handles requests that cross segment boundaries internally, merging segments and providing a single response to the client. But Pequod is not capable of creating the partitioning itself, at least not a very efficient one. The developer, with knowledge of application access patterns, must provide this mapping (or an algorithm that can produce one). Fortunately, relational overlays can be used to make this task easier. Rather than considering the whole key as a unit, a Pequod partitioning scheme can operate on one or more slots in a key's overlay. For example, the Twip developer can exploit the user identifier that is embedded in each `sub` and `post` key to evenly distribute users across nodes.

While conceptually simple, an efficient partitioning function may be deceptively difficult to construct. Consider a Twip partitioner that segments base data by user identifier, as above. The function provided by the developer needs to produce a partitioning of the `sub` and `post` keyspaces. The function can interpret the user identifier slot of the keys as a numerical value and evenly distribute keys across cache nodes. Perhaps the users are par-

titioned in a round-robin fashion—for example, 10 users across 3 nodes as:

Cache Node 1	**Cache Node 2**	**Cache Node 3**
[post|001, post|001⁺)	[post|002, post|002⁺)	[post|003, post|003⁺)
[post|004, post|004⁺)	[post|005, post|005⁺)	[post|006, post|006⁺)
[post|007, post|007⁺)	[post|008, post|008⁺)	[post|009, post|009⁺)
[post|010, post|010⁺)		
[sub|001, sub|001⁺)	[sub|002, sub|002⁺)	[sub|003, sub|003⁺)
[sub|004, sub|004⁺)	[sub|005, sub|005⁺)	[sub|006, sub|006⁺)
[sub|007, sub|007⁺)	[sub|008, sub|008⁺)	[sub|009, sub|009⁺)
[sub|010, sub|010⁺)		

This approach easily handles the case where user identifiers are generated sequentially, and offers quick lookup for a single user's data. However, it does not efficiently support scanning across user boundaries—for example, if the application wanted to collect the posts of its oldest users. The system can enumerate all the cache operations needed to piece together this scan, but it will require many small scan requests to every cache node. An alternative is to divide the identifier space by the number of cache nodes and store a range of users on each, as in:

Cache Node 1	**Cache Node 2**	**Cache Node 3**
[post|001, post|004⁺)	[post|005, post|007⁺)	[post|008, post|010⁺)
[sub|001, sub|004⁺)	[sub|005, sub|007⁺)	[sub|008, sub|010⁺)

This makes cross-user queries more efficient, but some consideration must be given to the method of assigning (or correcting) user identifiers so that system load is balanced across cache nodes. In either case, it is the responsibility of the developer to consider these tradeoffs in the context of the application's workload.

Data ranges that are not explicitly mapped by the partitioning function can still be requested by an application client. For example, the `tline` table is not mapped in either partitioning above. If there is a cache join that covers the unmapped range, it will be executed to fill the range. In this way, a cache join can be executed on any cache node. Alternatively,

if the `tline` range is explicitly partitioned, requests will be routed to the appropriate cache node. Requests for unmapped data ranges that do not correspond to cache join outputs return the empty set.

In summary, Pequod relies on an application-defined partitioning function to provide a deterministic mapping of keys to cache nodes. In addition to individual key lookup, this function must enumerate the segment-node pairs that cover a range-based lookup in the keyspace. This mapping is known to all cache nodes and to the application clients.

6.2 Subscriptions

All data ranges that are mapped by the partitioning function have a designated *home node* that stores the authoritative version of a cache entry and makes it accessible to clients and other cache nodes. Cache entries can be copied from the home node to other cache nodes in a distributed deployment.

Fetching *remote ranges* and maintaining their freshness is handled in Pequod with *subscriptions*. A subscription establishes a coupling between two cache nodes: a requester node specifies a range of data to be transferred, and the home node for the requested range complies. In addition, the home node promises to inform the requester of any updates that occur to the subscribed range. The updates continue until the requester explicitly terminates the subscription or the range is evicted from the home node. Cache nodes establish subscriptions using a set of inter-node commands. A summary of the commands:

- **subscribe**(***first, last***). Establishes a subscription for the range [*first*, *last*). All existing cache entries are returned and the subscription is recorded in the home server.
- **unsubscribe**(***first, last***). Cancels an existing subscription for the range [*first*, *last*). The home node removes its record of the subscription and stops sending updates.

- **MODIFY(*key, val, op*)**. Informs the subscriber of a modification to a subscribed range. The *key* argument identifies the modified entry while *op* indicates the type of modification (INSERT, DELETE, or UPDATE).
- **INVALIDATE(*first, last*)**. Informs the subscriber that the subscribed range [*first*, *last*) has been invalidated or evicted at the home node. As a result, the subscription is broken, and the subscriber must remove its record. The subscriber can establish a new subscription if the data are required to handle future requests.

It is important that the subscriber and home node agree on the set of active subscriptions; otherwise cache nodes may produce incorrect join output or serve stale data. For example, if a cache node *C* has a record of a subscription that covers a remote range that is needed to compute a cache join, it will not send another subscription to the home node *H*. Node *C* assumes by the presence of a subscription record that the data are already up-to-date. If, however, the home node *H* did not hold a corresponding subscription, the copied range in *C* may be stale with respect to *H* (because *H* had not been sending MODIFY commands for that range). Assuming a correct implementation of the subscription command set, this property can be maintained if there is a single, ordered, reliable communication channel between each pair of cache nodes. If each cache node processes messages from this channel in order (mixing incoming requests with responses to outgoing requests) then there can be no erroneously held subscriptions. This is not to say that the state of these subscriptions or the corresponding data ranges are exactly synced between cache nodes at all times; there could be messages in flight that must be processed before the subscription state agrees. However, if the system were allowed to quiesce, the subscription state would eventually converge. Consistency is further discussed in §6.6.

Pequod supports proxied requests; for example, if a client sends a request for a data

range to cache node *C* but the home node for the range is actually node *H*. Rather than returning an error, *C* makes a request to *H* for the data and returns the results to the waiting client. In addition, *C* caches a copy of the range in its own store so that it can be used to serve future requests without contacting the home server. Proxy requests offer the developer some flexibility in routing cache operations. For example, the client could avoid the home node intentionally; if many nodes proxy and cache popular data—like the tweets of a celebrity—then the load of serving this data can be effectively balanced. In an another example, a client may submit a request for a data range that is partitioned across many nodes. The primary node (the one communicating with the client) is responsible for splitting the request into multiple parts using the partitioning function, requesting data from other nodes, and constructing a single response to the client.

6.3 Cache join execution

Executing cache joins in a distributed deployment is relatively straightforward; a join is executed on a single machine and the subscription mechanism is used to ensure its inputs are cache resident. Figure 6.1 depicts the inter-node communication patterns typically used in Twip. When the client issues a timeline check for user `ann` (1), Pequod invokes the cache join (2), fetches the non-resident data from a peer node (3), caches it locally, and installs metadata to handle future operations (4). Once the data are local, the cache join continues, producing results for the client. Subsequently, when user `ken` makes a new post at time 500, handled at `ken`'s home node (5), Pequod forwards the information to subscribed nodes to keep their subscriptions up-to-date (6). This modification invokes the installed updater for `ann`'s timeline (7), ensuring that her timeline is fresh for the next check.

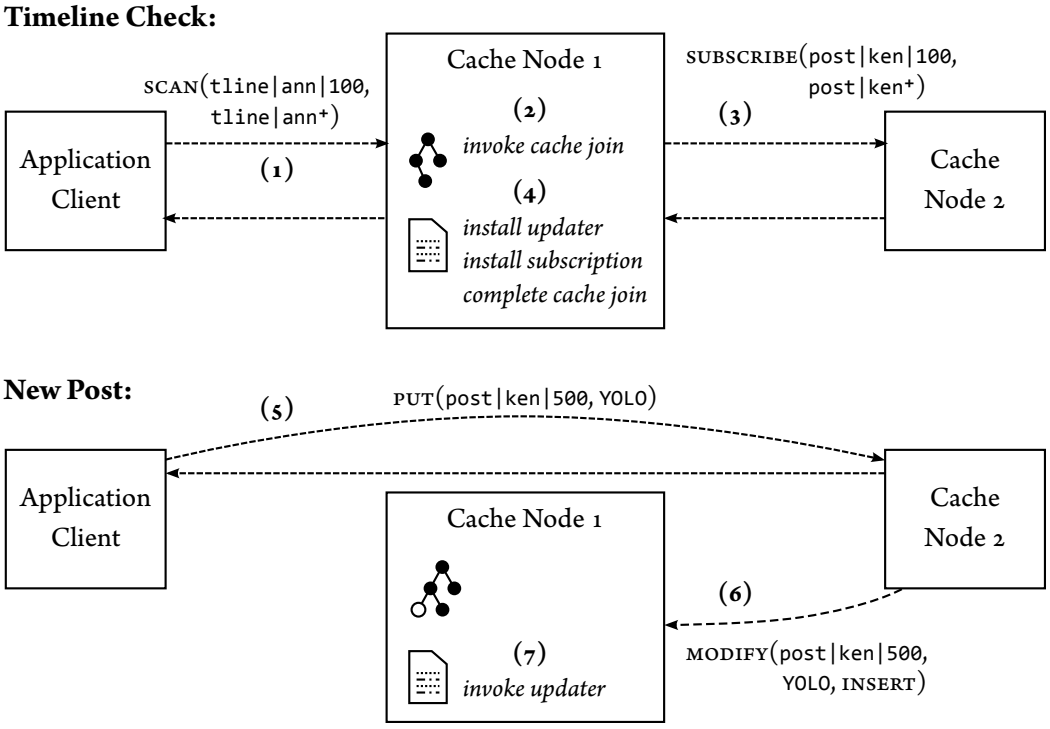

Figure 6.1: A Twip timeline check (1) is executed in a distributed Pequod deployment (2). Remote data needed to support the cache join are fetched and cached (3), and metadata are installed to handle future updates (4). A subsequent post (5) is forwarded to the subscribed node (6) to keep the derived data fresh (7).

Figure 6.2 highlights the changes to the cache join execution algorithm. Prior to scanning a source containing range, Pequod consults the partitioning function to determine if the range is mapped to one or more remote cache nodes. If the range is entirely local, the algorithm proceeds as before. Otherwise, the subscription records are queried to determine if a copy of each required remote range is already present. If not, a new subscription is made by communicating with the appropriate home node. Execution is blocked while the subscription is established.[1] When the response to the subscription is received, a new subscription record is installed and the algorithm can continue.

[1]The prototype implementation minimizes the time spent in a blocked state by making as many subscriptions as possible in parallel during each phase of the join execution.

```
1  process_sources(first, last, join, ss, js, srcidx):
2    src := join.source(srcidx)
3    [key⁻, key⁺) := ss.containingrange(src, first, last)
4    remotes := look up remote partitions of [key⁻, key⁺)
5    for each ⟨remrange ↦ node⟩ ∈ remotes:
6      sub := look up subscription record for remrange
7      if sub = ∅:
8        subscribe(remrange, node)   // blocks
9        install subscription record for remrange
10   srcjoin := look up join for [key⁻, key⁺)
11   if srcjoin ≠ ∅:
12     compute_cache_join(key⁻, key⁺, srcjoin)
13   install updater with context {src, js, ss} into [key⁻, key⁺)
14   for each ⟨key ↦ val⟩ where key ∈ [key⁻, key⁺) and key matches src.overlay(ss):
15     ss' := ss.fillslots(key)
16     if not join.isprimary(srcidx):
17       process_sources(first, last, join, ss', js, srcidx + 1)
18     else:
19       emit ⟨join.outputkey(ss') ↦ src.operator(val)⟩
```

Figure 6.2: Pseudo-code for the cache join execution algorithm, modified to fetch remote data ranges as necessary prior to scanning source containing ranges. Important updates to the algorithm in Figure 5.4 are shown in black. The (unchanged) code for COMPUTE_CACHE_JOIN is omitted.

Up to this point we have made no mention of concurrency with respect to the cache join execution algorithm; for simplicity of explanation we assumed a single threaded, single node deployment. We now assume that Pequod is implemented to allow concurrent operations at each node and is deployed across many nodes. This prompts the question: what is the expected output of a cache join that executes concurrently with other cache operations?

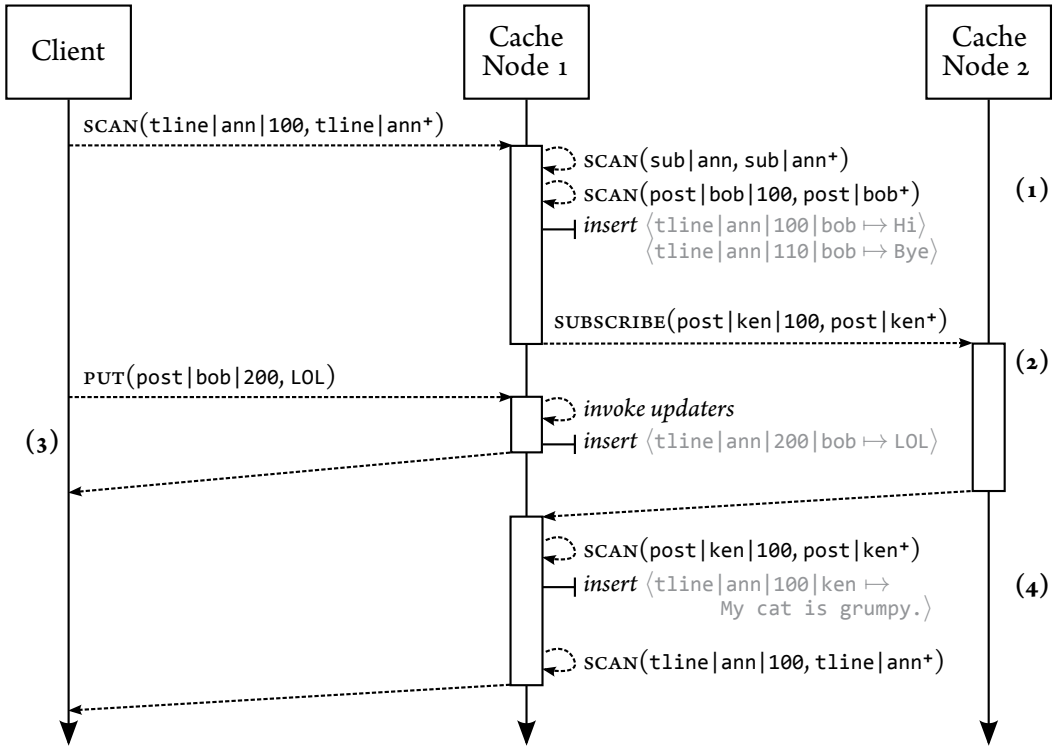

Figure 6.3: An example of concurrent cache operations. A cache join execution is blocked by a subscription request and a new insertion to an already scanned input causes a concurrent update. The response to the client reflects this update.

Consider an example execution, depicted in Figure 6.3, of the Twip timeline join in a two-node deployment. The client issues a SCAN for user ann's timeline and Pequod scans sub|ann to fetch her subscription list. This range is local to C_1. Next, the tweets of user bob, also cache resident, are scanned and some tline|ann key-value pairs are emitted (1). However, the tweets of user ken are not mapped to C_1 and are not cache resident. A subscription request is sent to C_2 for the data and the join execution is blocked (2). In the meantime, C_1 handles the insertion of a new tweet from bob, which emits another tline|ann key thanks to the previously installed updater (3). Eventually, the subscription request is fulfilled and the join execution continues (4). User ken's posts are used to emit the final tline|ann key.

For this example, the response to the client will include `bob`'s post at time 200. When the cache join algorithm emits a key-value pair, it is inserted directly into the store. Likewise, when the concurrent PUT is processed, the installed updater inserts the new output into the store. To gather a response for the client, the requested range, [`tline|ann|100`, `tline|ann+`), is scanned *after* all the sources are processed. Thus, the response to the client will represent the cache join as computed over an inconsistent, node-local view of the source data, which may include modifications to the source ranges that occurred after the start of the join execution.

6.4 Scaling

A distributed application-level cache is *scalable* if the addition of resources to the system has a positive effect on performance. Under this definition, if the cache is operating at capacity, adding more cache nodes should improve overall performance by spreading the cache load over more servers. Likewise, if the application experiences an increasing workload—for example, a steady influx of new users—a developer can add more cache nodes to offset this growth and maintain a performance target.

Like other key-value application-level caches, adding more cache nodes to Pequod will increase the storage capacity of a deployment. But in Pequod the addition of cache nodes also grows the *computational capacity* of the system—that is, the ability to execute cache joins. Parallelism is achieved by executing many cache joins concurrently on different nodes. Thus, when a new cache node is added, the system's computational capacity is increased.

6.5 Deployment

Deploying Pequod as a distributed system is simple. The developer instantiates several Pequod instances and supplies them each with a deployment configuration. The configuration includes the partitioning function, used to segment data and route cache requests, and a network topology, used to establish communication channels between cache nodes. At present, Pequod does not support reconfiguration.

The simplest deployment configuration defines a single, homogeneous cache pool, in which application base data are partitioned across the pool and communication is allowed between all nodes. However, the developer could improve the efficiency of the system by tailoring the configuration to more closely match the application workload. For example, consider a configuration for the Twip application that divides the cache node pool into two groups, *base* nodes that store `sub` and `post` data, and *compute* nodes that execute the timeline cache join. Each cache node is *specialized* by its placement into one of the groups. The partitioning function ensures that the base data are stored only on base nodes, and the network topology arranges the groups into two tiers. Figure 6.4 depicts the two-tiered Twip deployment. Application clients request timelines from compute nodes, which in turn fetch data from base nodes. Clients can write base data through the compute nodes, or in slightly modified topology, directly to the base nodes. Cache nodes within each tier do not intercommunicate.

A tiered deployment of Twip has several potential advantages over the single cache pool. Most importantly, it allows the developer to make more informed decisions regarding provisioning. Through a separation of concerns, she can scale the computational and storage capacities of the system independently. Further, she can make separate provisioning deci-

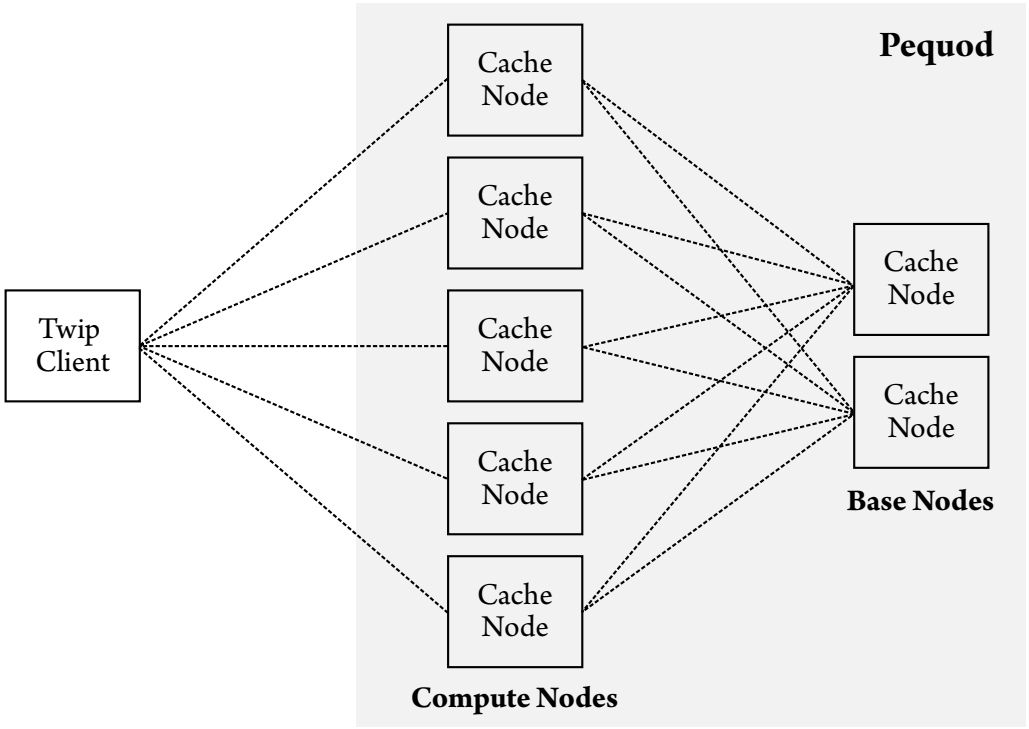

Figure 6.4: A two-tier deployment configuration for the Twip application. Application clients communicate with a tier of nodes that compute cache joins. A second tier stores the base data, providing access to the first tier via range subscriptions. Role specialization allows the developer to make more informed provisioning decisions and reduces overhead.

sions for different categories of data—for example, ensuring that there is enough capacity for a month's worth of tweets to be held in the cache. The distinction between tiers might carry through into the choice of hardware, with inexpensive commodity hardware in the first tier and more expensive (and possibly more reliable) hardware in the second. In addition to provisioning, the specialization of nodes in a two-tiered deployment might facilitate performance and error diagnosis.

It is up to the application developer to determine how Pequod can be deployed to achieve the application's performance goals. The partitioning function and network topology give the developer the tools they need to execute their desired deployment strategy.

6.6 Consistency

Pequod is *eventually consistent*: every update to base data eventually becomes visible to all interested nodes, but since update propagation is asynchronous, different nodes might see updates at different times. The maximum update delay depends on network properties, and is relatively low for our expected deployments (several servers within a single data center). Applications that formerly interacted only with persistent storage may be accustomed to stronger consistency guarantees (e.g., external consistency). Accepting a weaker consistency model is a prerequisite for using an application-level cache like Pequod. Many Web applications are tolerant of this kind of inconsistency.

For some applications, the Pequod prototype implementation can also support "read-your-own-writes" consistency, where writes made by a client are always visible to later reads by the same client. To achieve this level of consistency, each client must read from and write to a single Pequod cache node, and base data must be written directly to Pequod (using a look-through or write-through deployment) to avoid the asynchrony of database notification.

6.7 Discussion and limitations

In this section we discuss the limitations of our design and explore alternate choices.

Distributed cache joins

We established that Pequod's design allows for a cache join to be executed on any cache node and that the join's execution is restricted to exactly one node. The data necessary for executing the join will be copied as needed to the executing node.

Pequod could support an alternate approach: splitting the join execution, evaluating

portions of the join on remote nodes, and combining the results into a single response. This approach distributes the computation itself across several nodes. One advantage of this approach is that the base data used in the partial computation need not be transferred to the executing node. This is particularly useful for aggregate computations that operate over a potentially large amount of data and emit a single value.

Pequod could incorporate this strategy to compute new values, but how are these data maintained (if at all)? Is each node that computes a portion of the result responsible for maintaining that portion locally? Should the original executing node then hold a subscription to each portion so that it can keep a combined value fresh? Without such a subscription, each query of this value would require a gathering step that increases the latency of the operation.

We explore the possibility of distributed cache join execution under the assumption that data replication is an unwanted overhead that can be avoided. However, there are some advantages to replicating the base data and executing the join on a single node. Should space allow, the replicated data can be stored at the executing node and used to construct responses to future requests (and other cache joins). Of course, it might be wise to explore distributed join execution if the system is operating near capacity and the replicated data are evicted and re-fetched repeatedly. There is no universal solution—ideally Pequod would support both execution schemes and switch between them when appropriate.

Replication

The subscription mechanism allows cache joins to be executed on any cache node; the input data is copied to the site of the computation as needed. With this design we trade off storage capacity for computational capacity. But if used carelessly, data subscriptions

could severely limit the storage capacity of the system. Each copied cache entry consumes storage resources that could otherwise be used to cache a unique entry. At the pathological extreme—where every cache entry is copied to every node—the total working set size is reduced to $O(\frac{1}{N})$, where N is the number of cache nodes in the deployment.

Developers can take steps to prevent excessive replication. First, the application's cache joins and workload can be assessed to determine the expected replication factor of the input data. For example, it is likely that a Twip celebrity's tweets will be replicated to all cache nodes. In theory, this is a problem; but there are not that many celebrities, and a great number of user timelines will benefit from a co-located copy of the tweets. The data of regular users will also be replicated, though to a much lower degree. A developer must recognize that replication is unavoidable in the Twip workload. He can avoid excessive replication by carefully choosing where cache joins are executed—for example, repeated requests for a single user's timeline should be handled by the same cache node, if possible.

A similar problem exists in Newp: the base vote data is needed to compute the karma scores of users that comment on an article. If the developer partitions all data ranges across the cache nodes (say, by article identifier) and fetches article data from the corresponding home server, the karma cache join (5.5) will be invoked on every cache node. In the worst case, every commenter has commented on at least one article stored in every node. Because Pequod replicates data for cache join execution, the vote data will be cached on every node.

For example (using numeric identifiers for articles and users):

Cache Node 1	**Cache Node 2**	**Cache Node 3**
Partitioned	*Paritioned*	*Partitioned*
[article|000, article|333⁺)	[article|334, article|666⁺)	[article|667, article|999⁺)
[comment|000, comment|333⁺)	[comment|334, comment|666⁺)	[comment|667, comment|999⁺)
[vote|000, vote|333⁺)	[vote|334, vote|666⁺)	[vote|667, vote|999⁺)
Replicated	*Replicated*	*Replicated*
[vote|334, vote|999⁺)	[vote|000, vote|333⁺)	[vote|000, vote|666⁺)
	[vote|667, vote|999⁺)	
Computed	*Computed*	*Computed*
[karma|000, vote|999⁺)	[karma|000, vote|999⁺)	[karma|000, vote|999⁺)

Realizing that this scheme may cause excessive overhead, the developer could implement an alternate karma strategy. By manipulating the partitioning function, he could designate one cache node to compute all karma scores from co-located vote data:

Cache Node 1	**Cache Node 2**	**Cache Node 3**
Partitioned	*Paritioned*	*Partitioned*
[article|000, article|333⁺)	[article|334, article|666⁺)	[article|667, article|999⁺)
[comment|000, comment|333⁺)	[comment|334, comment|666⁺)	[comment|667, comment|999⁺)
[vote|000, vote|999⁺)		
Computed	*Replicated*	*Replicated*
[karma|000, vote|999⁺)	[karma|000, vote|999⁺)	[karma|000, vote|999⁺)

Then, when the scores are needed by other cache nodes, the output of the join is fetched by subscription. This effectively compresses the data that is transferred between nodes by introducing a specialization. The developer must correctly provision the system to handle the intentional load imbalance.

Though excessive replication could potentially cripple a Pequod deployment, a reasonable partitioning of application data and client requests can avoid this outcome. Excessive replication is not observed in our evaluation of Twip, as we show in §9.4.

Failure handling

We make no attempt to handle failure in the design or prototype implementation of Pequod. However, we recognize that some aspects of Pequod's design cause complications in the presence of failure. This is especially true when compared to contemporary caching systems, like memcached, in which cache servers do not inter-communicate. A failure in memcached is localized—a server that no longer responds to client requests can be replaced without the involvement of the other servers. By contrast, failure handling in Pequod is more complicated: each cache node may hold subscriptions to remote data, and further, may have promised to notify other nodes of changes to its local store. Thus, the disaster recovery and fault tolerance strategies in Pequod must account for these connections and act accordingly. We do not explore fault tolerance in this work, but are confident that existing techniques (e.g., leases, background replication) could be applied to keep the system available in spite of a node failure. Disaster recovery (recovering without maintaining system availability) is less problematic, as the contents of a cache node (or the entire cache deployment) can be rebuilt by loading data from the persistent store and re-executing cache joins. However, this failure mode is also less useful, as the cache is meant to be available, even when individual nodes are offline. The necessary design and implementation effort to validate these assumptions is left as future work.

Dynamic partitioning

The partitioning function in Pequod is statically defined—it is shared between application clients and cache nodes. Using the static function, clients can directly access the cache node that is most appropriate for a given request. The Pequod prototype implementation does not allow the partitioning function to change throughout the life of a deployment. In

a more robust implementation, the function could be adjusted to account for a change in system deployment (e.g., a failure). This scheme works well as long as all participants agree on the function in use.

An alternate design removes the application clients from the equation by introducing *coordinator nodes* that act as a gateway between clients and cache nodes. Once partitioning is hidden, Pequod could implement a number of dynamic partitioning schemes. For example, a Twip coordinator could co-locate the data of users in a social network to minimize replication costs. It could also monitor the computational load and available storage space of individual cache nodes to ensure proper balancing. Some experimentation would be necessary to determine if proxying requests through a coordination layer within the cache is worth the inevitable complexity and latency trade-offs.

7
Eviction

Eviction is a challenging problem in any cache; under memory pressure, the system must decide which data should be removed to make space for new cache entries. At a minimum, the cache's eviction strategy must free enough space to complete any pending operations without significant delay. In many key-value caches, key-value pairs are independent: there are no semantically meaningful relationships among keys. This simplifies several maintenance tasks, such as eviction, since keys can be treated independently. But it does not hold for Pequod. A key-value pair in Pequod is linked to other key-value pairs in a range by its application-defined relational overlay, and possibly to many other key-value pairs in ancestor (or dependent) ranges if it was computed by (or is used as as a source in) a cache join. This interdependence complicates eviction; the implementation must take care to preserve the integrity of derived key-value pairs (e.g., a sum), or perform the necessary co-evictions.

In this chapter we describe how eviction in Pequod differs from other systems. We do not seek an optimal eviction strategy for Pequod; an optimal policy is likely application-specific. Rather, we seek a generalized strategy for eviction that addresses the challenges caused by interdependence and performs adequately in the presence of cache joins.

7.1 Range-based eviction

Pequod applies range-based eviction to the cache; multiple key-value pairs linked by a relational overlay are evicted simultaneously. Range-based eviction ensures the integrity

of data ranges. If we evicted individual keys from a range we could no longer derive correct results from future cache join executions without additional effort to reconstruct the range. Thus, we avoid fragmenting ranges, opting to evict them in their entirety. This strategy aligns with Pequod's treatment of data ranges as a first-class primitive.

We assume that the system is capable of dividing the keyspace into disjoint ranges for the purpose of performing range-based eviction. Further, we require that each key-value pair belong to exactly one eviction range. Since cache joins can be interleaved, producing key-value pairs with semantically equivalent keys, a simple partitioning based on lexicographic order is not sufficient. Our prototype implementation uses the bounds of SCAN operations to define the eviction ranges of base data. Derived ranges are defined by the bounds of their initial cache join execution, and are associated with the join specification that produced the output, guaranteeing exclusive membership of the contained key-value pairs.

In considering whole data ranges as a single unit, Pequod can apply simple eviction policies, such as evicting the least recently used (LRU) items. As a result, a single eviction decision can cause the removal of many key-value pairs. In theory, the evicted range can be reloaded relatively easily, by fetching data from the persistent store or executing a cache join. Further, cache misses caused by the eviction are also localized, potentially restricting the scope of the ill effects in the application domain (e.g., to a single Twip user).

Interconnected cache entries cause another problem: *cascading evictions*. The cache join mechanism creates dependencies between data ranges. With the default maintenance policy, derived data are kept fresh as base data are modified. If a base range is evicted (along with the installed updaters), we can no longer guarantee that the derived data are fresh. Thus, upon eviction of base data, all transitively derived ranges are also evicted. Unsurprisingly, cascading eviction can cause significant performance problems. For example, if Pe-

quod evicted the tweets of a Twip celebrity, millions of user timelines might also be evicted. And the cascade is not limited to a single cache node; when base data are evicted from a node, all subscriptions to that data are canceled. The cascade will continue on through the subscribing nodes.

Cascading eviction is a serious problem for Pequod. The remainder of this chapter presents mechanisms for mitigating its effects on system performance.

7.2 Policies

Each eviction decision in Pequod involves a range of data. But not all ranges have equal eviction cost. There are two costs to consider, the initial cost of performing the eviction and the subsequent cost to reload the evicted data on a future cache miss. Every key-value pair stored in a cache node belongs to a data range, which is categorized in one of three ways: as a base range (caching data from the persistent store), a remote range (data copied from another cache node via subscription), or a derived range (computed by a cache join). The costs of evicting a base range and a remote range are similar—in addition to removing the key-value pairs in the range, Pequod evicts any derived ranges that use the evicted range as an input. The cost of reloading these ranges is added latency (though that latency is likely higher for base ranges). If we consider the computational cost of re-executing any derived ranges, the total cost of evicting a base or remote range could be quite high. By contrast, the cost of evicting a derived range is relatively low. Unless the derived range is used in a chained join, we need consider only the computational cost of re-executing the join on a subsequent SCAN. And for some applications, the re-execution will have a much narrower scope—for example, if we evicted a Twip timeline that covered the last month but only re-execute the join for the past hour.

By understanding the costs associated with evicting ranges in Pequod, we can begin to devise more sophisticated eviction policies. The Pequod prototype implementation supports three eviction policies:

- **Random.** A range is selected at random for eviction. There is no distinction made between base, remote, and derived ranges; all are equally likely to be selected. This is by far the simplest eviction policy implemented in Pequod. It is appealing due to its simplicity, but is unlikely to perform well for applications that have skewed workloads. For these applications it is likely advantageous to evict ranges based on frequency of access.
- **Least recently used (LRU).** The last access time of every range is recorded and stored in a sorted metadata structure. The range that was least recently accessed is selected for eviction, regardless of type. This policy operates under the assumption that recency of access correlates positively with frequency of access; that is, if a range was recently accessed it will be requested again in the near future. This is true for some applications—Twip, for example, has a skewed workload with some users checking their timelines more frequently than others. The last access time is not transitively updated; accessing a base range will not modify a dependent's position in the LRU list. Likewise, accessing a fresh derived range will not affect its inputs' positions. However, if an input range is accessed during join execution—for example, in support of forward execution or a lazy update—the last access time of the input range will be updated accordingly.
- **Category LRU.** The ranges are categorized by type—base, remote, and derived—and are added to category-specific LRU lists. The system ranks the categories and evicts the least recently used range from the lowest priority list first. The lists are

processed in priority order; when a lower priority list is exhausted, the next higher list is used. Thus, given the ranking *derived* $<$ *remote* $<$ *base*, the system will evict all derived data prior to evicting any base data.

Other eviction policies are possible and may be worth pursuing as future work. For example, Pequod might combine the category LRU policy with a notion of connectedness—evicting the base and remote ranges with the fewest dependent ranges first (to limit the cascade effect). Or, the system could compute an expected cost of eviction that includes the reloading cost and the number of anticipated cache misses. This type of calculation, which is similar to the cost analysis performed by a database query optimizer, could use metrics collected at runtime in addition to any static information about the ranges (such as the join structure). Clearly, this is a complex solution to the eviction problem that may or may not outperform simpler strategies. These concepts are not currently implemented in Pequod, and their evaluation is outside the scope of this thesis.

7.3 Tombstones

Cascading evictions pose a significant problem for Pequod; cache joins are useful, but their utility is irrelevant if the system performs poorly in the face of eviction. Pequod avoids cascades by installing an eviction policy that favors uncoupled ranges, mitigating their ill effects with *lazy invalidation*. Lazy invalidation defers eviction cascades by limiting the scope of the immediate eviction to the selected range. The idea is simple: if Pequod can correctly maintain the freshness of a derived range in the absence of its base data, then it is safe to remove the base data and keep the derived data in place. For example, a cache node for Twip could evict the tweets of a user without evicting his followers' timelines.

To guarantee that derived data are kept fresh, Pequod must process all future updates to the evicted base range as if it were cache resident. This means that any installed updaters need to be triggered when the base range is modified. Pequod tracks evicted base data ranges with a piece of metadata called a *tombstone*. Tombstones provide a context to Pequod for the purpose of applying updates. A tombstone stores the bounds of the evicted range and inherits the installed updaters. The use of tombstones changes the update logic slightly; when a PUT or REMOVE operation is performed on a cache node, Pequod first looks for a tombstone that covers the given key. If found, the tombstone's updaters are invoked and the operation does not modify the store. Tombstones are only used when an evicted range has dependencies that Pequod would like to maintain.

Tombstones are not a permanent solution to the cascading eviction problem; they are used only to maintain the freshness of existing derived ranges. Pequod will eventually need to invalidate and evict the derived ranges for one of the following reasons:

1. The evicted range must be scanned to fulfill a client request or as part of a new cache join execution.
2. An update is made that cannot be handled by the tombstone context alone (more on this below).

In both cases, the actual key-value pairs of the evicted range are needed to complete a pending client request. At this point, Pequod will remove the tombstone for the evicted range and evict any transitively derived ranges. Thus, the cascade is not completely avoided, it is just deferred. Lazy invalidation allows the system to continue operating as long as the evicted data are not needed.

Figure 7.1 depicts how tombstones are used to defer eviction cascades. User ann checks her timeline, which establishes a subscription for ken's posts (1). The remote range is then

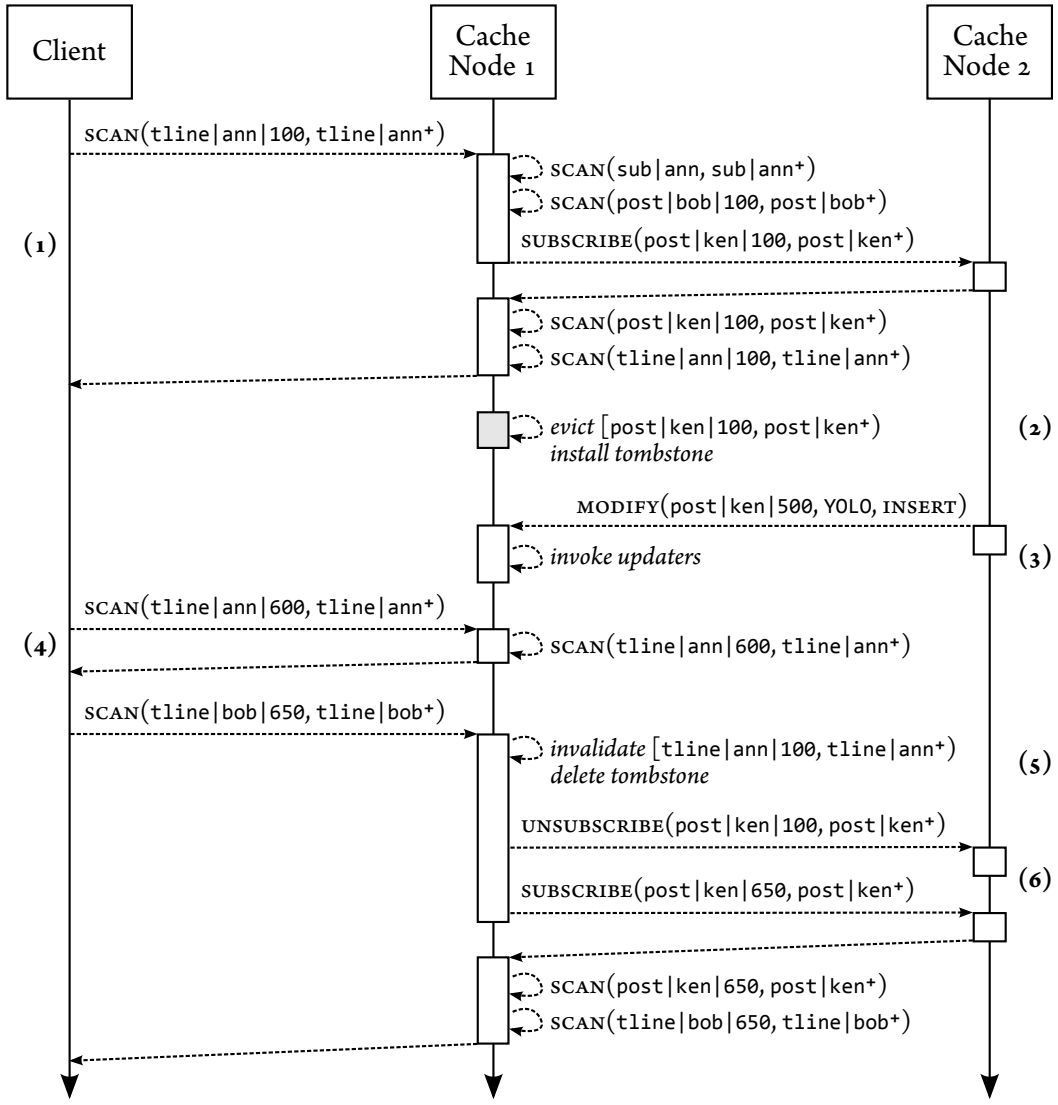

Figure 7.1: A tombstone defers an eviction cascade. The posts of user ken are evicted after user ann's timeline is generated. A tombstone marking the eviction is installed (2) and the timeline remains VALID. A subsequent post by ken (3) and a timeline check by ann (4) are processed as normal. When user bob logs in, his timeline construction requires the data from the evicted range, triggering the cascade and invalidating ann's timeline (5). The subscription to ken's posts is broken, and a new subscription is made to complete bob's timeline (6).

evicted by the cache and a tombstone marking the range [post|ken|100, post|ken+) is installed (2). A subsequent modification to the subscribed range, a new post by ken, is processed by the tombstone, triggering an update to ann's cached timeline (3). Another timeline check by ann (4) requires no additional computation. Eventually, a timeline check by bob, who is subscribed to ken and has no cached timeline, triggers the eviction cascade and invalidates ann's timeline. The cache node breaks the old subscription to ken's posts (to stop notifications) and makes a new subscription that covers bob's timeline request (6). Pequod could use the updaters attached to the defunct tombstone to retain portions of derived timelines—for example, that of ann—by transferring them to the new base range subscription. Our prototype implementation instead reconstructs the derived data when they are next requested.

The ability to apply an update using the context provided by a tombstone is determined by the operator used in the updater and the cache operation being performed. Some operators, such as copy, can always be applied. For example, if we evict a Twip user's tweets from a cache node and retain their followers' timelines, that user could post a new tweet without triggering an eviction cascade. The updater would insert a new key into each of his followers' timelines. In fact, he could even remove an old tweet without incident—Pequod will determine the timeline key for each follower and remove it, if it exists. There is no need to know if that key actually existed in the evicted range. The only context needed by the copy operator is provided by the key-value pair in update operation.

However, some updaters require more information than can be provided by the modifying operation. For example, if Pequod evicts a range that is used by a count operator, how should it handle future PUT and REMOVE operations on that range? Without knowing if a key is already counted in the aggregate, Pequod cannot guarantee the accuracy of the

derived data. To solve this dilemma, Pequod attaches another piece of information to some tombstones: a Bloom filter [8]. A Bloom filter is a space-efficient data structure that can be used to test set membership. When a range is evicted, each key in the range is recorded in the Bloom filter. The filter is then used to determine how operators such as `count` should behave on an update; for PUT on a `count` range, Pequod will only increment the derived value if the key did not previously exist in the evicted range.

Unfortunately, Bloom filters cannot be used to test set membership exactly; a Bloom filter is a probabilistic data structure, and there is some probability of false positives. For example, if we wanted to REMOVE a key from the counted range and the Bloom filter indicates that the key was present in the evicted range, there is a chance that the Bloom filter is wrong. Decrementing the count in this case might lead to incorrect values. However, a Bloom filter can indicate with certainty that an element is not a member of set. For this reason, Pequod uses the Bloom filter only to test for negative membership. This allows the system to perform some updates (such as incrementing the count for new keys) and forestall the eviction cascade.

A Bloom filter is just one piece of contextual information that can be associated with a tombstone. In the Pequod prototype, only updaters that apply `count` and `sum` use Bloom filters. Updaters that apply `min` and `max` use the store itself, inspecting the existing derived output before taking action. Figure 7.2 summarizes how tombstone metadata is used to determine if an update can be applied to an evicted range.

It is inevitable that some updates will require the evicted range to be reloaded. Even if we could efficiently and accurately test for set membership, the `sum` operator still needs access to evicted values to correctly handle overwrites. In addition, operators such as `min` (or `max`) need to scan the entire range to produce a new value should the key that is associated with

Operator	PUT		REMOVE	
copy	yes		yes	
	Bloom match?			
	True	*False*	*True*	*False*
count	no	yes	no	yes
sum	no	yes	no	yes
	Current min/max key?			
	True	*False*	*True*	*False*
min	⋆	yes	no	yes
max	⋆	yes	no	yes

⋆ - yes if the new value is greater/less than the current max/min, no otherwise.

Figure 7.2: A table for determining the applicability of updates to a previously evicted range. The metadata in the tombstone, existing derived output, and the key-value pair of the update operation are used to determine if an update can be applied or if the evicted range must be reloaded.

the current minimum (or maximum) be removed. However, the developer may be able to avoid these cases by carefully constructing cache joins and application update semantics. For example, Twip uses only the `copy` operator, which allows any update to an evicted range. Newp uses the `count` aggregator to produce user karma from votes, but its workload fits well with the Bloom filter. If we tune the filter to have a sufficiently low false positive rate and votes are rarely rescinded, the vast majority of updates will be new insertions that pass the filter. These updates are easily handled by the `count` operator.

In summary, Pequod can defer the ill effects of cascading evictions by storing a small amount of information about the evicted range and keeping derived data in place. Until the evicted data are truly needed, the derived data are maintained and used to respond to client requests. Depending on the application workload, lazy invalidation can reduce cache churn and improve the hit rate.

7.4 Discussion and limitations

The ideas presented in this chapter make eviction overhead more manageable in Pequod. However, there are some issues with this design that need addressing.

Metadata

Tombstones are a useful mechanism for delaying cascading evictions. Unfortunately, the tombstones themselves take up space; range bounds, updaters, Bloom filters and other metadata, though likely smaller in size than the evicted range data they represent, consume memory that could otherwise be used to store cache entries. Eventually, the amount of memory used for tombstones will be nontrivial. The eviction mechanism, which focuses on evicting data ranges, must also take metadata usage into account and weigh the cost of evicting the tombstones (causing cascades) versus the need for more space in the store (which can improve the hit rate). We do not explicitly evict metadata in the Pequod prototype implementation; tombstones are removed when evicted data are reloaded or there are no longer any dependent ranges.

Range splitting

The use of ranges as the unit of eviction allows Pequod to apply simple eviction strategies to an ordered store. The eviction mechanism does not recognize subranges or individual keys within a range. It makes sense for some ranges to be handled as a unit—for example, a Twip user's subscription list is processed in full, never in part. But other ranges, such as the tweets or timeline of a Twip user, have subranges that are more valuable to the application than others. For example, Twip users rarely request historical timelines. The recent timeline data (and transitively, the recent posts of the users that make up the timeline) are

more likely to be accessed and should remain cache resident, if possible. As a result, it may be beneficial to just evict the older portion of these ranges and retain the newer subrange.

To avoid evicting useful data, it might be advantageous for Pequod to recognize some basic patterns in data ranges. Using this information, the system could preferentially select subranges for removal or preservation. For example, Pequod might exploit the fact that Twip `post` and `tline` ranges grow in one direction and choose to evict older data first. But how would Pequod recognize these patterns? We could add an annotation to the join specification that informs the system of common patterns, such as growth in a single direction. Using this information, Pequod could automatically split the range metadata (e.g., join status ranges, updaters) used internally to track the validity cache join inputs and outputs.

Supposing that Pequod can recognize range patterns and subdivide ranges, how would the system automate this process? That is, how does Pequod know where to make the split? One solution is to have the developer use domain knowledge to set some threshold—for example, the number of keys or amount of time that defines a "recent" timeline. However, this introduces yet another tuning parameter to the system. Instead, we propose a mechanism for automated range splitting that uses the SCAN requests themselves to define the partition point. It works in this way: for each access, with some small probability the range being scanned is split at the lower bound of the SCAN request. The two subranges can now be considered individually by the eviction mechanism. The lower range will not likely be accessed again and will work its way toward the head of the LRU. The latter portion will continue to be accessed (and possibly split again) and remain toward the tail of the LRU.

Range splitting is just one optimization that can be made to the range-based eviction mechanism in Pequod. We do not implement this feature in the prototype; we discuss it here to highlight one shortcoming (and subsequent solution) of using data ranges as

units. The above example that automates the splitting process works for ranges that grow in a single direction. Alternate approaches would need to be developed for ranges that are modified in other ways.

8

Implementation

We describe our prototype implementation of Pequod, a proof-of-concept system that serves as a platform for evaluating our design choices. We do not describe the entire system here; rather, we cover the data structures used to store cache entries and metadata and several optimizations that allow Pequod to achieve its performance goals. The code for Pequod is available online for download at `http://github.com/bryankate/pequod`.

The Pequod prototype is implemented in C++ as a single-threaded, event-driven program. We use Tamer [39], a C++ language extension that makes event-driven programming more manageable. As a single-threaded program, commands issued to Pequod are executed to completion—in isolation and with unrestricted access to the program's state—or until blocked by I/O.

A distributed deployment of Pequod is composed of multiple instances of this program communicating via RPC. The prototype uses a custom RPC layer that establishes reliable, asynchronous messaging channels between cache nodes and with clients. Pequod uses JSON structured messages [17], formatted for the wire with MessagePack [32], to invoke cache commands and return responses.

8.1 Ordered data storage

Pequod stores key-value pairs in red-black trees. A traditional key-value cache typically stores cache entries in a hash table, which is an appropriate structure in the absence of multi-key operations. We cannot use a hash table as the primary data structure in Pequod because we expose an ordered store abstraction to the application developer. Though tree-based stores support range scans (which, in turn, enable cache joins), they introduce overhead (relative to a hash table implementation) for every access to the store.

Naturally, we are concerned that $O(\log n)$ time complexity for store lookups will hinder Pequod's ability to remain competitive with other caches. One way to minimize this overhead is to keep the value of n as small as possible. Rather than artificially limiting the total number of cache entries, we partition the store into logical *tables*. Each table has a separate red-black tree that holds keys with the same prefix—for example, those that begin with `sub`, `post`, or `tline`. Pequod uses the relational overlays of the installed cache join specifications to automatically define the table prefixes (the portion of a key's overlay prior to the first slot definition). By partitioning the store in this way, we effectively reduce the value of n for any given lookup to n_{prefix}, the number of keys that share that prefix. Separating concerns for different ranges with this design sped up Pequod significantly relative to a single store (§9.8).

An auxiliary structure, the *hash index*, associates the table prefix with the root of that table's tree. The hash index is used to jump directly to the appropriate table when the store is accessed. For example, if a Twip user generates a new tweet, Pequod begins its traversal of the store (to find the correct insertion point) by jumping directly to the `post` table. Despite being partitioned in this way, the cache can be accessed as a single ordered store; Pequod ensures that cross-table queries are handled correctly.

Pequod also uses red-black trees to store range-based metadata, such as the join status ranges, updaters, active subscriptions, and tombstones. When possible, these metadata are stored in the appropriate table structure so that fewer intervals are considered on each lookup. The nodes that represent the metadata ranges in the interval trees are also used to track ranges for eviction. Pequod uses Boost intrusive data structures [9] that allow a metadata tree node to double as an entry in eviction data structures (e.g., node in a LRU list). The intrusive data structures save space in the cache by sharing a single range object and avoiding superfluous lookups in many cases.

8.2 Optimizations

In this section we focus on implementation details that allow Pequod to approach (and in some cases surpass) the performance of contemporary key-value caches. Optimizations are described here and evaluated in §9.8.

Subtables

The combination of red-black trees and a hash index is effective at reducing the overhead of tree lookups. The mechanism is not reserved for top-level tables; Pequod can further divide the store into *subtables* to provide even faster access to specific ranges of data. The developer can specify a slot in the relational overlay that is used to partition a top-level table. For example, the Twip developer could indicate that the `sub`, `post`, and `tline` tables are all divisible by the *user* slot. As a result, Pequod will create a separate table structure, with its own red-black tree and range metadata, for each unique user. The subtable of a specific user can be accessed in $O(1)$ time by looking up its root in a hash index. Subtables provide a significant performance improvement for applications that have well-defined data

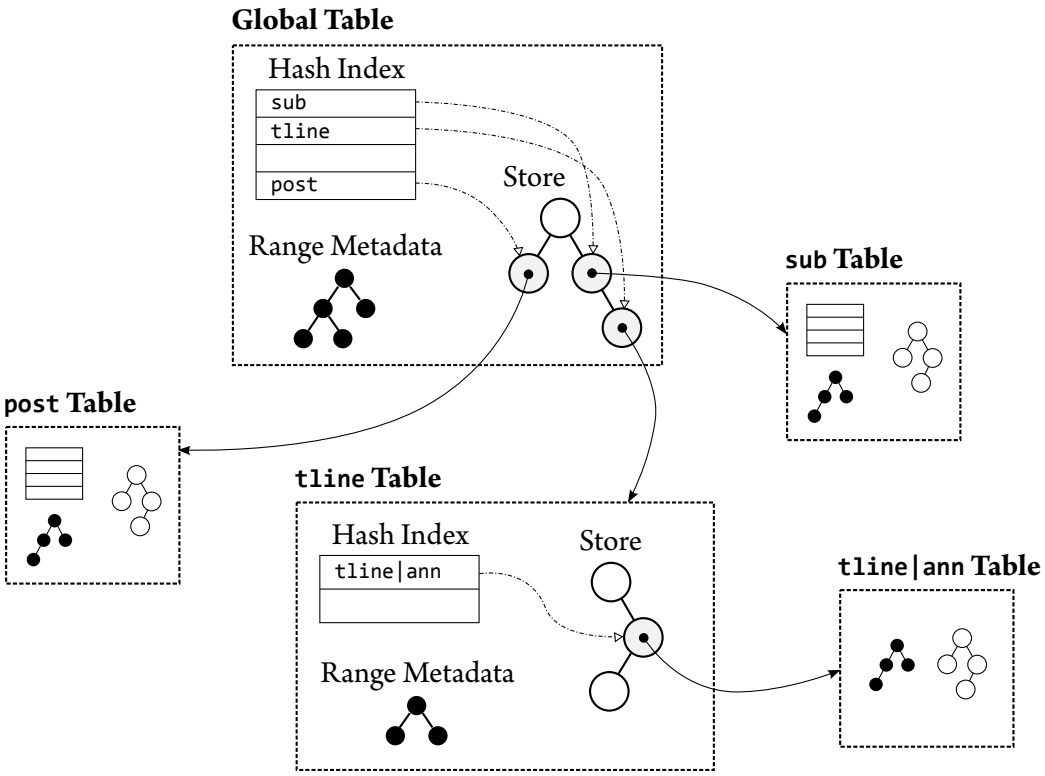

Figure 8.1: The structure of the data store, using Twip as an example. The keyspace is divided into three top-level tables, which are further divided into per-user subtables. Access to an individual user's data is expedited by querying the hash index at each level.

boundaries and rarely request data across these boundaries (such as Twip and Newp).

The layered table abstraction is implemented in Pequod using a tree-of-trees approach. A global table contains a red-black tree, range metadata (e.g., join status ranges and tombstones), and a hash index. Each node in the tree is either a key-value pair or a pointer to a subtable, which has its own set of data structures. In theory, this nesting of tables could continue indefinitely. However, the Pequod prototype limits the number of layers to two. Figure 8.1 depicts the structure of the two-level store for Twip. Using this structure, to access the root of user ann's timeline, Pequod would first look up the `tline` table in the global hash index, then the table for ann in the `tline` table's index. This is much faster than travers-

ing the entire store from the root. However, such a traversal is possible; Pequod can handle any data request with a traversal from the root, descending into subtables as necessary.

Output hints

In many of our applications, each update to a derived range either modifies the same key as the previous update (as is common for the `count` operator) or inserts a new key immediately after the previous update (as when inserting a fresh post into a Twip timeline). Both types of modification can be performed in $O(1)$ amortized time given a pointer to the last-updated key. Each join status range therefore maintains a pointer to its last-updated key. We call this pointer the *output hint*. A reference counting scheme ensures that the hint stays valid even if the underlying key-value pair is removed from the store. This optimization is designed to avoid a costly tree lookup for every update to a base range.

Value sharing

The `copy` operator often requires Pequod to install multiple copies of a value into different output ranges. For example, Twip inserts a copy of a user's tweet into each of his followers' timelines. To reduce memory overhead, Pequod allows multiple output ranges to share the source's value—that is, to point to the same memory. A reference counting scheme ensures that the value remains available even when the original key-value pair is removed from the store.

This optimization fits in naturally with in-cache materialized views; when Pequod generates the output it can recognize that the two values are meant to reference the same content. Alternatively, if the application performed materialization outside of the cache and merely inserted key-value pairs for each derived output, Pequod would be unable to ap-

ply this optimization—it treats each client-provided value as unique. Value sharing is only useful for joins that use the `copy` operator, but it introduces no overhead on other joins.

9 Evaluation

We evaluate Pequod's performance as an application-level cache using workloads that feature materialized views. We show that Pequod outperforms traditional key-value caches on these workloads and that the system can scale to handle realistic load. We investigate the potential bottlenecks in our design, demonstrate the utility of eviction tombstones, and explore the cache join composition and performance annotations. We conclude that our design is sound: Pequod cache joins are a viable solution to the cache freshness problem.

9.1 Experiment setup

We evaluate Pequod using two hardware configurations, a multiprocessor and a cluster of Amazon EC2 virtual machines (VMs). The multiprocessor is an Amazon EC2 instance with 32 cores and 244 gigabytes of RAM running Ubuntu Linux 13.04. The Amazon EC2 cluster, used to evaluate scalability, is composed of many VM instances connected by a 10 gigabit network. Each VM has 32 cores, 60–244 gigabytes of RAM, and runs Amazon Linux 2013.09.2. All Amazon EC2 instances run with hardware-assisted virtualization (HVM) enabled.

Application clients communicate with Pequod nodes using RPC. Experiments on the multicore machine use TCP over the loopback interface for RPC invocation. Clients are event-driven processes that keep many RPCs outstanding. We run enough clients to saturate the Pequod nodes, when applicable.

We do not evaluate database interaction; Pequod is deployed as a look-through cache—applications send it updates directly. Notification bottlenecks in the database made the performance of our write-around deployment uninteresting. Although we enable eviction, it never triggers in our experiments (with the exception of the experiments that measure eviction policy directly, §9.5).

We ran most experiments several times and observed little to no performance variability. Unless otherwise specified, our results report the mean value of three experiment runs.

Twip workload

In most of our experiments, Pequod is configured to run Twip. The underlying data are derived from a Twitter social graph obtained in 2009 [27]. The full graph, which contains approximately 40,000,000 users and 1,400,000,000 relationships, is used in scalability experiments. All other Twip experiments use a sampled subgraph containing approximately 1,800,000 users and 72,000,000 relationships. We sample the original graph by selecting approximately 4.5% of users uniformly at random, preserving the existing relationships between chosen users. Figure 9.1 depicts the distribution of followers per user for both social graphs. The majority of users have 10 or fewer followers, and only 1% have 1000 or more. The most popular user has nearly 3,000,000 followers.

Our clients model the actions of individual Twip users. Each modeled user:

1. "logs in," obtaining a list of recent tweets in their personalized timeline;
2. repeatedly checks for new tweets, subscribes to other users, and posts new tweets;
3. and logs out (though they may log in again later).

The incremental timeline updates in step (2) return many fewer tweets than the initial checks at login time. These events occur in the rough ratio: 5% initial timeline checks, 9%

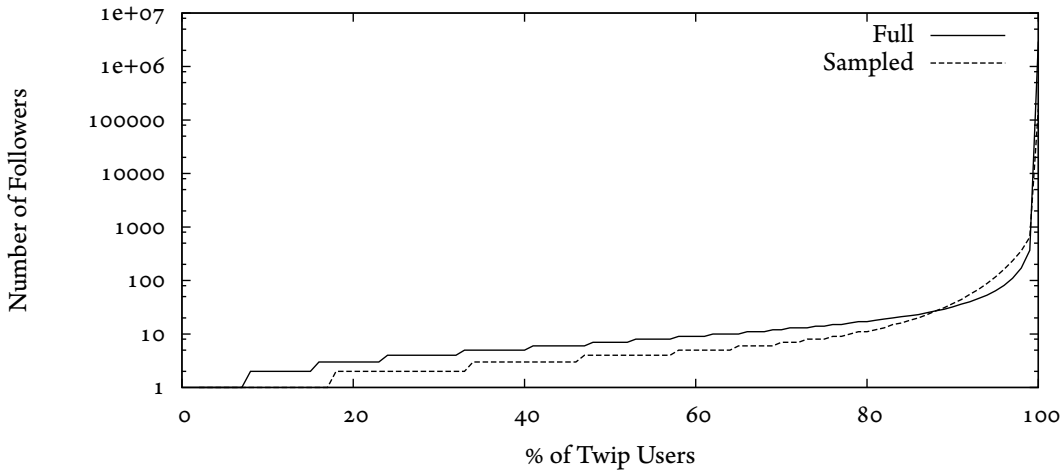

Figure 9.1: The Twip follower distributions for the full 2009 Twitter social graph and the sampled subgraph used in our experiments. In the full graph, 62% of users have 10 or fewer followers, 92% have 40 or fewer, and 1% have over 1000. The most popular user has almost 3,000,000. Note the log axis.

new subscriptions, 85% incremental timeline updates, and 1% posts. This breakdown of events is derived from information on the real Twitter [25] workload. Users post with different likelihoods. The probability that a user posts a message is proportional to the log of their follower count, so more popular users tweet more often.

We define a benchmark, Twip-small, by combining this workload with the sampled Twitter subgraph. In this benchmark, 70% of users are active (the remainder never check their timelines) and each user checks her timeline 50 times. The result is a fixed workload comprising approximately 62,000,000 timeline checks, 6,200,000 new relationships, and 620,000 new posts. This benchmark is used in the majority of experiments. A second benchmark, Twip-large, is used in scalability experiments. The same workload is applied to the full 2009 Twitter graph with 70% of users active. This benchmark comprises approximately 1,400,000,000 timeline checks, 140,000,000 new relationships, and 14,000,000 new posts.

9.2 System comparison

In this section we compare the performance of Pequod against other systems: memcached 1.4.16, Redis 2.8.5, and PostgreSQL 9.1. Both memcached and Redis represent the state-of-the-art in distributed key-value caches. These systems are used extensively in Web application deployments, including the construction and maintenance of cached user timelines at Twitter [25]. Neither product supports in-cache materialization—all computation is performed in client routines. We use PostgreSQL as an example of a system that does support server-side computation and materialization. We compare against PostgreSQL using an atypical configuration; the database is deployed in-memory as a cache.

Key-value workload

We begin our evaluation of Pequod with a basic key-value cache comparison. We want to analyze the potential performance overhead caused by our tree-based implementation. To quantify this effect, we use a third-party benchmarking tool, Memtier Benchmark [31], to execute a pure GET workload on Pequod, memcached, and Redis. Given that Pequod is optimized for cache join execution and maintenance, we do not expect to outperform the comparison systems.

The benchmark is run against a single server instance of each cache type. The benchmark client uses 4 threads with 50 active connections each to saturate the cache with GET requests. The cache is pre-populated with 12 byte keys and 512 byte values. The keys consist of a common prefix, "m|", and a 10 digit, zero-padded number.

The experiment is designed so that every request results in a cache hit. The benchmark is executed for 30 seconds and metrics are collected by the benchmark client. Both Redis and memcached are configured to optimize performance for this experiment; disk checkpoint-

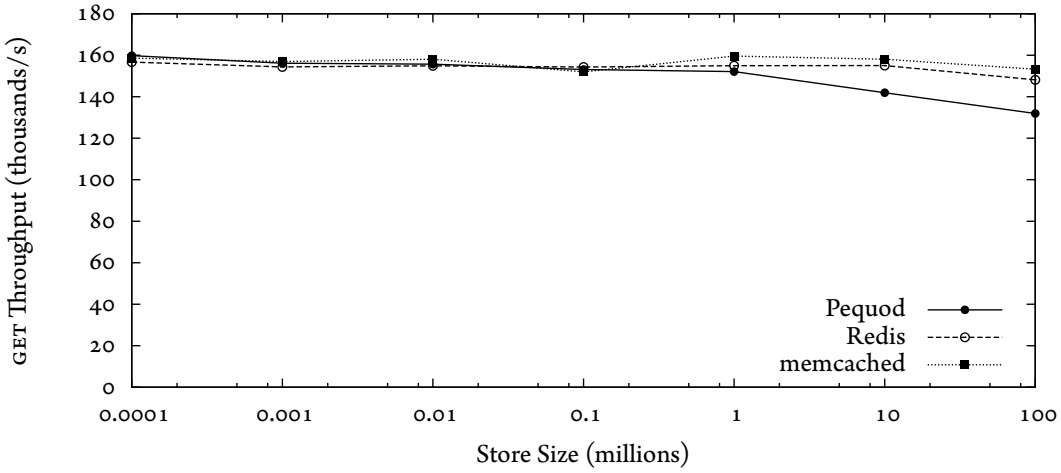

Figure 9.2: A comparison of the throughput achieved by Pequod, memcached, and Redis using a pure GET workload and a single cache server. Overhead caused by tree lookups reduces average throughput by a factor of 1.14x as the store grows from 100,000 to 100,000,000 key-value pairs. Redis and memcached performance remains relatively stable due to implementations based on hash tables. Note the log scale axis.

ing and eviction are disabled and the most efficient binary protocols are used for RPCs.

We run this benchmark multiple times on the multiprocessor, varying the store size of the cache in each iteration. The store size varies from 100 to 100,000,000 key-value pairs over the course of the experiment. We expect the performance of memcached and Redis, both implemented with hash tables, to remain roughly unchanged, but we expect the performance of Pequod to drop somewhat because of its $O(\log n)$ lookup cost.

Figure 9.2 shows the experiment results. Pequod is performance competitive with memcached and Redis up to 100,000 key-value pairs. As the store grows larger, each lookup incurs more overhead; the total throughput drops by a factor of 1.14x at 100,000,000 key-value pairs. This result is not necessarily problematic—Pequod is designed for in-cache materialization, not pure key-value workloads. In cache join scenarios, optimizations like hash indexes and subtables (§8.2) are expected to mitigate $O(\log n)$ overheads. The devel-

oper also has the option of partitioning a large store across many Pequod instances, each with smaller stores. Regardless, this experiment demonstrates that our prototype performs nearly as well as mature key-value caches on an unfavorable (to Pequod) workload.

As an aside, this experiment also highlights a storage overhead; Pequod adds approximately 297 bytes of overhead per key-value pair. Both memcached and Redis store key-value pairs with less overhead—approximately 104 and 115 bytes, respectively. Some of this overhead is attributed to our choice of data structure, but a good deal comes from implementation optimizations meant to speed cache join execution (e.g., flags, pointers to range metadata). Reducing this overhead is important to improve Pequod's resource utilization, but is left for future study.

Materialization workload

We next compare Pequod with other systems on a workload that includes materialization. We use the Twip-small benchmark to model a Twip deployment. The goal of this comparison is to determine if in-cache materialization, in addition to improving programmability, will improve system performance. Ideally, Pequod would pay no penalty for this convenience—it should perform no worse than its contemporaries.

Only one of our comparison systems, PostgreSQL, is capable of in-cache materialization. Both memcached and Redis require a "client-managed" implementation of materialization. A client-managed application executes the same logic as a cache join, but it does so on an application server. Multiple round trips to the cache are required to construct the initial timeline (in the case of a cache miss) and keep cached timelines maintained when new content is generated. Twitter uses a client-managed approach to cache user timelines [25].

Our Twip application executes in two modes: a client-managed mode that performs

joins in the client and a cache-managed mode that relies on in-cache materialization. An adapter is used to interact with the cache—the application is agnostic to the system under test. We compare Pequod to the systems listed above in addition to "client-Pequod": an execution of the Twip application in client-managed mode using Pequod as a key-value cache (with range support); no cache joins are used. Client-Pequod lets us evaluate the performance of in-cache materialization in isolation.

Each system runs the same TWIP-SMALL workload to completion as quickly as possible. Neither Redis nor memcached supports in-cache computation, so as in client-Pequod, their clients actively manage user timelines; Redis stores timelines as sorted sets of tweets, and memcached as a string to which tweets are appended. PostgreSQL, in contrast, does support in-cache computation. Although our test version lacks automatically-updated materialized views, we use triggers to get a similar effect.

Each system is given six cores in our multicore machine. PostgreSQL runs a single process with multiple threads, while the other systems partition the store and use one process per core. The machine's remaining cores run client processes; for each system, we used the number of client processes that produced the best system runtime. We configure all systems so that data is stored in memory and consistency is relaxed as much as possible.[1]

Figure 9.3 shows the results. Pequod, which uses materialized views, runs a factor of 1.64x faster than client-Pequod, which doesn't. The penalty is roughly equally divided between RPC overhead (client-Pequod makes many more RPCs) and insertion overhead (client-Pequod doesn't benefit from output hints or value sharing). Although a more op-

[1]For PostgreSQL, we allocate a shared memory buffer large enough to hold our entire data set, place the data store in an in-memory file system, and tune for performance: we disable *fsync*, *synchronous commit*, and *full page writes* and set *bgwriter lru maxpages* to zero.

System	Runtime (s)
Pequod	197.06 (1.00x)
Redis	262.62 (1.33x)
client-Pequod	323.29 (1.64x)
memcached	784.43 (3.98x)
PostgreSQL	1882.78 (9.55x)

Figure 9.3: Time to process the TWIP-SMALL benchmark to completion using Pequod and related systems. Smaller numbers are better. Pequod outperforms its closest competitor, Redis, by a factor of 1.33x.

timized client-managed caching system could beat Pequod (perhaps by implementing Pequod-like functionality specialized for the application), RPC overhead and program complexity remain as challenges for any client-managed or special-purpose system. Pequod runs a factor of 1.33x faster than Redis: join support does not sacrifice the performance advantages of key-value caches. Redis runs a factor of 1.23x faster than client-Pequod, however. This difference is due to Redis's hash table data structure, which offers $O(1)$ lookups. Though tree optimizations could speed up client-Pequod somewhat, unordered stores offer fundamental performance advantages over ordered stores. Memcached runs a factor of 3x slower than Redis: the Twip workload has more writes than memcached prefers. The traditional database, despite running in memory with relaxed ACID guarantees, is not a suitable replacement for a Web application cache. Pequod outperforms PostgreSQL by nearly an order of magnitude (9.55x). Widely-available databases with true materialized view support were also evaluated; they performed similarly to PostgreSQL.

In summary, we exceed our goal of performing no worse than existing application caches, at least for the Twip materialization workload. For some applications, Pequod cache joins are both simpler and faster.

9.3 Computational variability

Pequod is specifically designed for in-cache computation using cache joins. One concern with Pequod is that in-cache computation will increase the latency of cache operations when computation is performed (and that the latency varies with the amount of computation). This is a valid concern when comparing Pequod to existing key-value application-level caches that offer consistent performance (albeit with restricted functionality). To make informed provisioning decisions, the developer should be aware of the latency Pequod adds to cache queries. Of course, the latencies will be workload dependent; requesting a fresh Twip timeline will require no computation, whereas a stale or missing timeline will prompt a partial or full join execution.

We measure the query latency in Pequod using the Twip-small benchmark. We run Twip in two modes: first, with the timeline cache join installed in the typical way, computing timelines and keeping them fresh incrementally; and second, with a `pull` annotation that forces timelines to be computed anew on each request. Running Twip in these two modes, we generate timeline requests that

- cover a fresh range and require no computation;
- follow a subscription change and require a partial computation;
- and require a full computation (all requests in the `pull` experiment).

We measure the round trip time (RTT) of each timeline check in the client. We isolate the measurements from buffering and load effects by executing the benchmark synchronously: a single client issues one request at a time. The experiment is conducted on the multiprocessor using a single Pequod cache node. The cache is pre-populated with the subscription lists of all users and 1,000,000 historical posts. Thus, all data needed for timeline construction are cache resident. All active users are logged in prior to the experiment, so each user

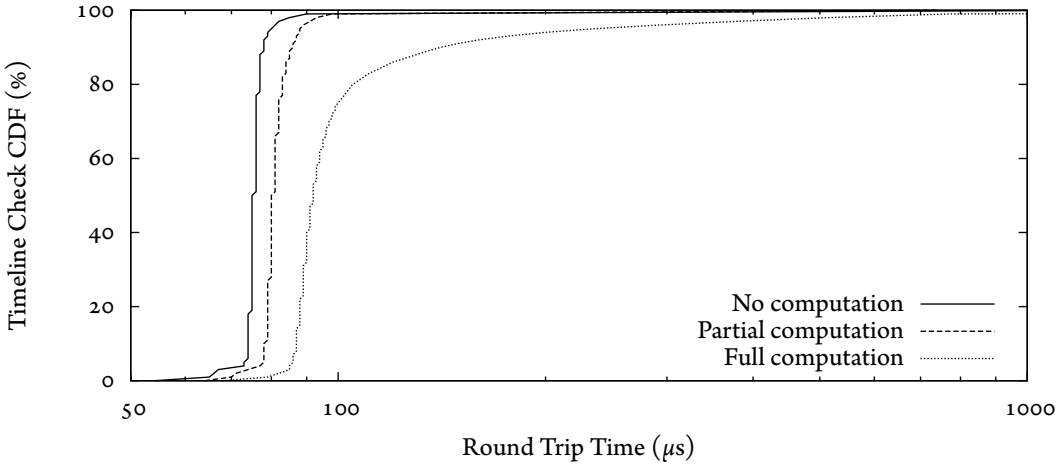

Figure 9.4: Cumulative distribution functions (CDFs) of timeline check RTTs in Twip-small (without network latency). Checks that require little or no computation exhibit low variance. Full timeline computations are significantly more expensive on average and have higher variance. Note the log scale axis.

begins with a valid join status range. This avoids measuring the full computation required for initial timeline construction (which is measured explicitly in the `pull` join mode).

Figure 9.4 shows the results of a single experiment run. The experiment turns out as we expect: requests for fresh timelines have the lowest RTT (mean: 75.98μs, σ: 7.38), followed by those that require partial updates (mean: 81.54μs, σ: 8.68), and those that are computed in full (mean: 130.28μs, σ: 300.25). The high variance of full timeline computation is a reflection of the variable workload: most users subscribe to tens of others, but some subscribe to tens of thousands. In contrast, a partial computation in Twip typically handles a single subscription change and has much lower variance.

The results show the relative difference in computation time of fresh, stale, and missing timelines. This difference is attributed to the amount of work Pequod must do for each join execution (the number of ranges scanned, keys enumerated, and key-value pairs generated). However, we are also interested in the overhead of cache join queries. To measure

query overhead, we compare the timeline checks with a basic key-value operation. Using Memtier Benchmark, we measure the RTT of GET requests issued synchronously to a single cache node. The cache is populated with 1,000,000 key-value pairs, each with a 12 byte key and 1 kilobyte value. Pequod, Redis, and memcached all average 60μs for a basic GET. Timeline checks that do not require computation average 75μs. We attribute some of this difference to cache join overhead—querying range metadata to determine validity—and some to scan overhead—iterating through the store to enumerate key-value pairs. Though discernible in this micro-benchmark comparison, the observed overhead is negligible.

In summary, we verify that join execution will cause variability in request latency. Further, we recognize that full computation can cause significant spikes and should be avoided. This topic is studied further in §9.6. The developer should also determine the computational requirements of their application to properly provision the cache.

9.4 Scalability

Pequod is designed as a distributed service that can be scaled to suit the needs of an application as it grows. As such, we expect the addition of resources to a Pequod deployment to produce an increase in the computational and storage capacities of the system. In this section we conduct experiments to demonstrate scalability and identify any potential limitations. The results shown are from a single experiment run.

Scaling computation

We conduct an experiment using a Twip workload scaled to match the query rates of the real Twitter service [25]. As of 2012, Twitter users generated new content at an average rate of 3,000 tweets per second. During peak hours, the service averaged 6,000 new tweets per

second. In our experiment, we vary the post rate from 1,000–13,000 tweets per second. At these target rates, the Pequod deployment is expected to handle 111,000–1,443,000 client requests per second (100 timeline checks and 10 new subscriptions for every new post). Each run of the experiment is executed in the VM cluster with Pequod deployed in a two-tier configuration (8 base nodes and 18 compute nodes). The cluster is provisioned adequately to prevent a CPU bottleneck at the highest load tested. The Twip clients, 456 in total, collaboratively generate load at the target rate for 5 minutes. The cache is warmed by logging in all active users prior to the experiment.

We run the experiment on a fixed deployment to isolate the varying computation from other factors that could affect join performance as the workload grows, such as a change in the way users are distributed across the nodes. We measure the CPU utilization of each compute node and calculate a metric, W, to quantify the change in computation overhead as the system scales. Abstractly, the metric W represents the amount of resources consumed to perform a fixed unit of work. In a system that scales perfectly, this metric would remain constant as the load increases. We compute W in this experiment as

$$W = \frac{N \times C}{P}$$

where N is the number of compute nodes, $C \in [0, 1]$ is the average CPU utilization of the compute nodes during the experiment, and P is the post rate (in thousands per second). We also record the round trip time of every timeline check, which we expect to remain relatively constant given the static deployment.

The experiment is a success: Pequod scales to handle a realistic Twitter workload. Figure 9.5 depicts the results. The top panel plots our CPU load metric, W, as a function of

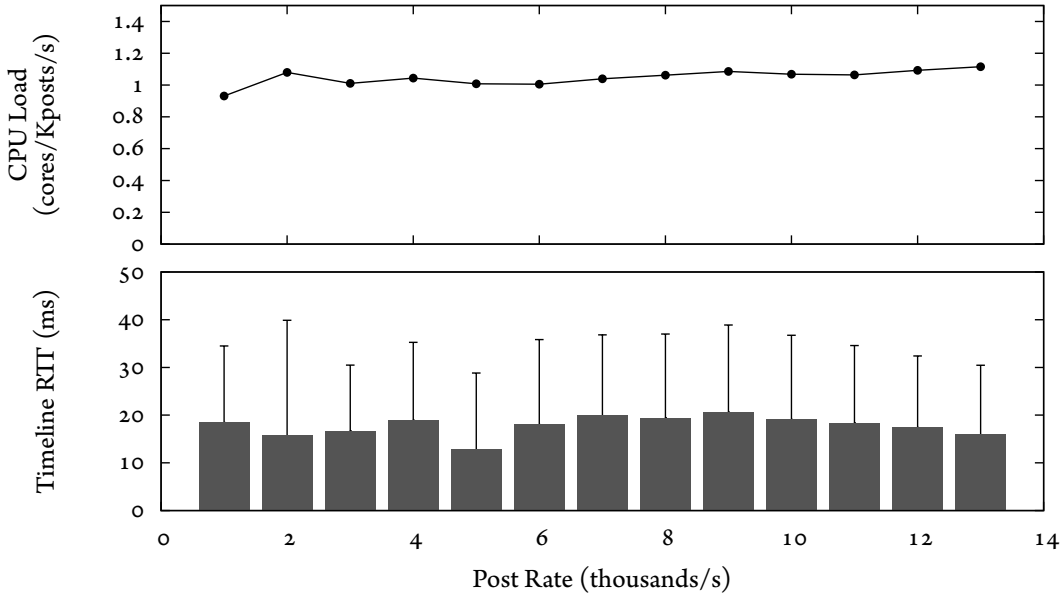

Figure 9.5: An increasing Twip workload is executed using a two-tier Pequod deployment with 18 compute nodes and 8 base nodes. Despite an order of magnitude increase in the workload, the CPU load metric only increases by a factor of 1.43x and the RTT of timeline checks remains relatively stable. The computation overhead of cache joins does not prohibit system scalability for the Twip workload.

the target post rate. It indicates that the cache join computation overhead remains relatively constant with respect to query rate. The application is not perfectly scalable; there is a slight upward trend in the data (a factor of 1.43x from 1,000 to 13,000 posts per second). Given the order of magnitude increase in the workload, this additional overhead is not prohibitive. The increase in W likely relates to the size of the cached timelines as the post rate is scaled. More work is required to construct and transmit a response for each query.

The bottom panel of Figure 9.5 plots the mean and standard deviation of the RTT for timeline checks. These results are not surprising, given that the deployment is fixed (only 18 queries can execute at any time) and the system was provisioned such that the cache nodes are never saturated. When allocated fewer resources, we expect the RTT of timeline checks to suffer. Indeed, when we execute this workload at a rate of 6,000 posts per second

using only 6 compute nodes (not shown), the mean RTT increases to 27.22ms, a factor of 1.51x longer than with 18 compute nodes.

Scaling storage

Pequod is designed to replicate data in support of cache join execution. This design allows the computational capacity of the system to scale with additional cache nodes. In this section we evaluate the effectiveness of this design, demonstrating that data replication does not limit the scalability of the system with respect to storage capacity.

It is possible to construct an application that replicates data to every cache node. In this case, the total storage capacity is asymptotically limited to the capacity of a single node. Twip and Newp do not fall into this category, but they do require some thought to eliminate superfluous replication. If the developer partitions the requests that trigger cache join execution carefully, some data movement can be avoided. For example, if the requests of a Twip user are consistently directed toward a specific cache node, that user's subscription list will be resident only on that node. The posts, however, will be replicated to cache nodes as needed. In the worst case, each of a user's followers will be directed to a different cache node. In practice, multiple followers will share a node, so a single copy of post data will service many derived timelines. We expect celebrity data to propagate to all cache nodes.

We run an experiment to measure the extent of Twip post replication. We use the Twip-large benchmark and a Pequod deployment with 64 compute nodes. The benchmark produces approximately 14,000,000 new posts, which are generated according to the Twip workload (popular users tweet more frequently). Figure 9.6 depicts the results. The bottom panel shows how the posts are distributed across the compute nodes. The top panel is the CDF of the same data. Approximately 50% of posts are replicated to 12 or fewer nodes, and

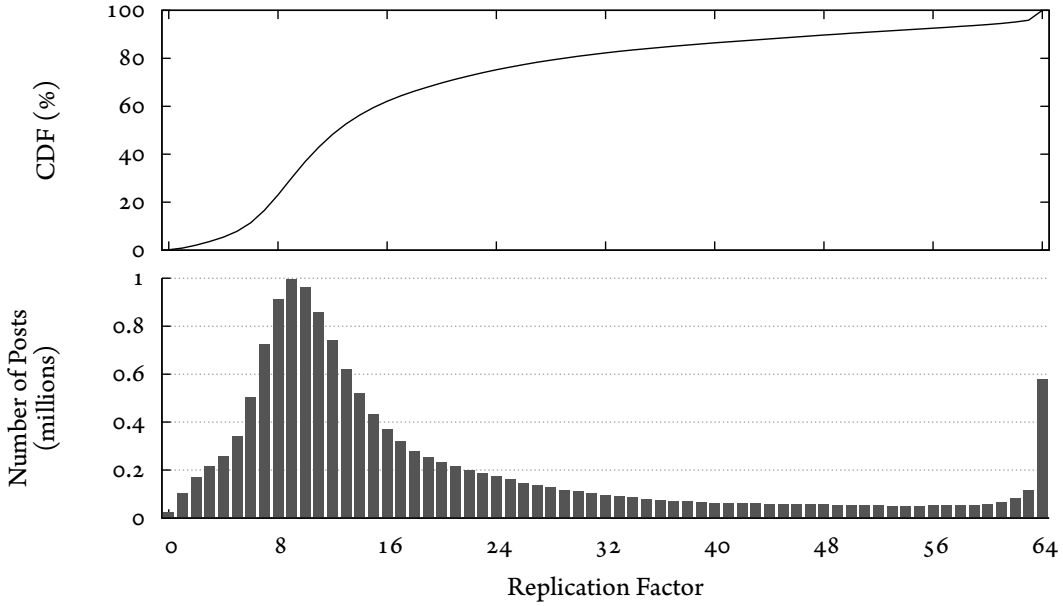

Figure 9.6: The distribution of approximately 14,000,000 Twip user posts over 64 Pequod compute nodes in a Twip-large benchmark execution. Approximately 50% of posts are replicated to 12 or fewer nodes, 80% to 29 or fewer. Celebrity posts, 4% of the total, are replicated to all nodes.

80% to 29 or fewer. Only 4% of the posts—those of celebrities—are copied to all nodes.

The results of the experiment are encouraging. Some overhead is observed, but does not limit scalability. Pequod scales to handle this real-world workload. The overhead is minimized by carefully routing client requests; timeline checks for each user are directed to a specific cache node. This arrangement guarantees that the `sub` and `tline` data for a user is resident on only one compute node. If we had chosen a strategy that is less concerned with data placement, such as a random draw or load balancing, the overhead would grow. The distribution of `post` data across the nodes mimics the follower distribution of our Twip workload (Figure 9.1). Using domain knowledge, a developer can design data partitioning and request routing strategies that prevent excessive replication, and provision the system for the expected replication pattern.

Distribution overhead

Finally, we investigate overheads associated with increasing the number of nodes in a Pequod deployment. A good reason for adding cache nodes is to increase the storage capacity of the system. We have demonstrated that a small number of cache nodes can be sufficient to cover the computational needs of a Web-scale Twip deployment. However, our experiment lasted only five minutes; the real Twitter service stores cached timelines for weeks. The resources of many machines are required to grow the cache to this size.

We conduct an experiment to identify overheads, other than the known replication overhead, that increase with the number of nodes. For example, many Twip users co-located on a compute node can share a single subscription to the posts of a popular user. If the number of compute nodes is doubled, the same data are copied to twice as many nodes. In addition to using more storage resources, the system spends more time generating and processing subscription updates, and more bandwidth transferring the data to subscribers.

We execute the Twip-large benchmark against deployments with 12 and 48 compute nodes. We measure the network bandwidth used for communication between the base and compute nodes and the memory consumed by the cache nodes that is unrelated to storing key-value pairs. The base nodes collectively consume a factor of 1.02x more memory (approximately 7 gigabytes) in the 48 node deployment. A similarly sized overhead is observed at the compute nodes. The increase in consumption is caused by duplicate metadata—mostly subscription records—that arise from spreading formerly co-located users across a larger number of nodes. Likewise, a larger fraction of the consumed network bandwidth is dedicated to inter-node subscription maintenance, increasing from roughly 10% to 16% between 12 and 48 compute nodes. Though these overheads are potential bottlenecks, they do not dominate resource consumption for our Web-scale Twip workload.

9.5 Eviction

Eviction is a critical function in any application-level cache. In general, a cache deployment is typically capable of storing a tiny fraction of the data held in persistent storage. For the cache to be effective, it should hold a working set of data that are currently relevant to the application (resulting in a high cache hit ratio). When the cache is full, data are selected for eviction to make room for new entries. In §7.2 and §7.3 we discuss policies for selecting ranges for eviction and a mechanism (tombstones) for mitigating the ill effects of cascading eviction. In this section we evaluate several policy–tombstone combinations to determine if these strategies are effective in practice.

In this section we run experiments to answer two questions: do type-aware eviction policies perform better than type-agnostic policies, and are eviction tombstones an effective solution for dealing with cascades? We cannot answer these questions in general—eviction performance is workload dependent. Rather, we demonstrate the effectiveness of our design on a Twip workload. We begin with a benchmark based on Twip-small with one modification: there are only two operations, timeline checks and new posts, generated with a ratio of 100:1. This simplified Twip workload allows us to measure the maximum performance impact of tombstones (when cascades are deferred indefinitely).

We execute the simplified benchmark multiple times, varying the eviction policy and toggling tombstone optimization. The policies tested are:

- **No eviction.** Eviction is disabled by setting a memory threshold much higher than what is consumed during the experiment. This is used as a baseline comparison.
- **LRU.** All ranges (base and derived) are represented in a single list of recently used ranges. Items are selected for eviction from the head of the list, and recently accessed items are moved to the back.

Policy	Runtime (s)	Timeline RTT Mean [σ] (ms)	
		Before Eviction	*During Eviction*
No Eviction	370.17 (1.00x)	65.80 [17.19]	–
LRU	496.01 (1.34x)	67.63 [16.38]	105.40 [39.98]
LRU-derived	577.95 (1.56x)	67.51 [26.04]	211.71 [188.44]
LRU-remote	493.96 (1.33x)	66.88 [17.20]	106.09 [40.35]
Tombstone LRU	388.05 (1.05x)	67.71 [16.59]	75.43 [18.81]
Tombstone LRU-derived	593.07 (1.60x)	67.09 [26.35]	222.10 [229.73]
Tombstone LRU-remote	383.34 (1.04x)	66.57 [16.92]	74.85 [18.92]

Figure 9.7: A summary of eviction policy performance on a simplified Twip workload. For a two-tier deployment, preferential eviction of remote ranges results in superior performance. Timeline checks during the eviction period are slower by a factor of 1.58x without eviction tombstones enabled. Enabling tombstones reduces the overhead to a factor of 1.12x.

- **LRU-derived.** Ranges are segregated into two types (derived and remote) and are stored in separate LRU lists. During eviction, the derived list is exhausted before evicting any item in the remote list.
- **LRU-remote.** Identical policy to LRU-derived, with the exception that the remote ranges are evicted preferentially.

We run the benchmark against a two-tier Pequod deployment (§6.5) consisting of a single compute node and a single base node. The nodes are deployed on separate VMs in the Amazon EC2 cluster. The benchmark is executed as quickly as possible, and the total runtime is recorded along with the RTT of every timeline check. We set a memory threshold of 22.5 gigabytes on the compute node. This threshold is selected so that eviction is triggered during the experiment, dividing the execution into pre- and post-eviction segments.

The results of the experiment are summarized in Figure 9.7. From a high-level perspective, the system behaves as expected: with eviction enabled, the store size is limited to

22.5 gigabytes (without eviction it grows to 26.7 gigabytes) and the benchmark runs longer. The overhead introduced by eviction varies with the combination of eviction parameters. We first consider the set of experiments without eviction tombstones: LRU and LRU-remote clearly outperform LRU-derived. This is not surprising, given the cost of full timeline computation (§9.3). The LRU and LRU-remote policies perform similarly because they both primarily evict remote ranges; base data ranges are not accessed (read) after timeline construction in this workload, so they naturally migrate to the head of the LRU. The latency overhead is significant—a factor of 1.58x on average for LRU-remote, the best performing policy. Evicting remote ranges causes eviction cascades, which result in full timeline computations on future checks. Given that both LRU-derived and LRU-remote result in full timeline computations, why is the overhead so much higher (a factor of 1.99x) when derived data are preferentially evicted? If anything, we would expect the overhead of LRU-remote to be higher, because base data must be re-fetched before the timeline computation can begin. The answer lies with one of the performance optimizations, value sharing, implemented in our prototype system (§8.2). Derived timelines have unique keys (which consume additional memory), but they share the memory for their values with the base data key-value pairs. Thus, evicting a derived range computed by the `copy` operator (while the base data remain cache resident) has a limited effect on the total memory consumption: more ranges need to be evicted to reach the eviction goal. The experiment with LRU-derived evicted 5.74x more ranges (2.45x more key-value pairs) than LRU-remote, resulting in many more timeline computations.

We observe from the initial results that evicting remote ranges is preferable for this workload in this deployment configuration. However, there is a significant latency added to timeline checks when eviction is enabled. The overhead is caused by eviction cascades

that invalidate the derived timelines when base data are evicted. We now consider the results from the experiments in which eviction tombstones are enabled. As before, LRU and LRU-remote perform nearly equivalently. As expected, the latency overhead is reduced (but not eliminated) when cascades are deferred. The overhead that remains (a factor of 1.12x), is largely caused by the necessary metadata lookup and validity checking associated with eviction tombstones. From these results, it is clear that eviction tombstones can have a positive effect on system performance.

Tombstones can be used to defer (but not eliminate) the ill effects of eviction cascades: when the evicted data is next accessed, the cascade will occur. The application workload will determine the limitations of this optimization. For example, a partial timeline computation is required when a Twip user makes a new subscription. As part of this computation, the recent posts of the subscribed-to user are scanned and merged into the existing cached timeline. If those posts were previously evicted from the cache, the computation would stall while the tombstone is removed (causing an eviction cascade) and the data are reloaded. The cascade is problematic because it causes the existing cached timeline (and the timelines of other users that subscribe to the same user) to be invalidated.

We conduct another experiment to evaluate the utility of tombstones in the presence of operations that will trigger eviction cascades. We use the TWIP-SMALL benchmark, which includes subscription changes, in the same two-tier deployment. We test three configurations of Pequod: no eviction, LRU-remote, and LRU-remote with tombstones enabled.

Figure 9.8 depicts the experiment as a time series. We measure the store size and SCAN RPC rate at the compute node, and the rates of SUBSCRIBE and UNSUBSCRIBE RPCs at the base node. Note: these rates refer to the number of remote data range subscriptions that occur in the distributed system, not Twip user subscriptions. The runs are indistinguishable

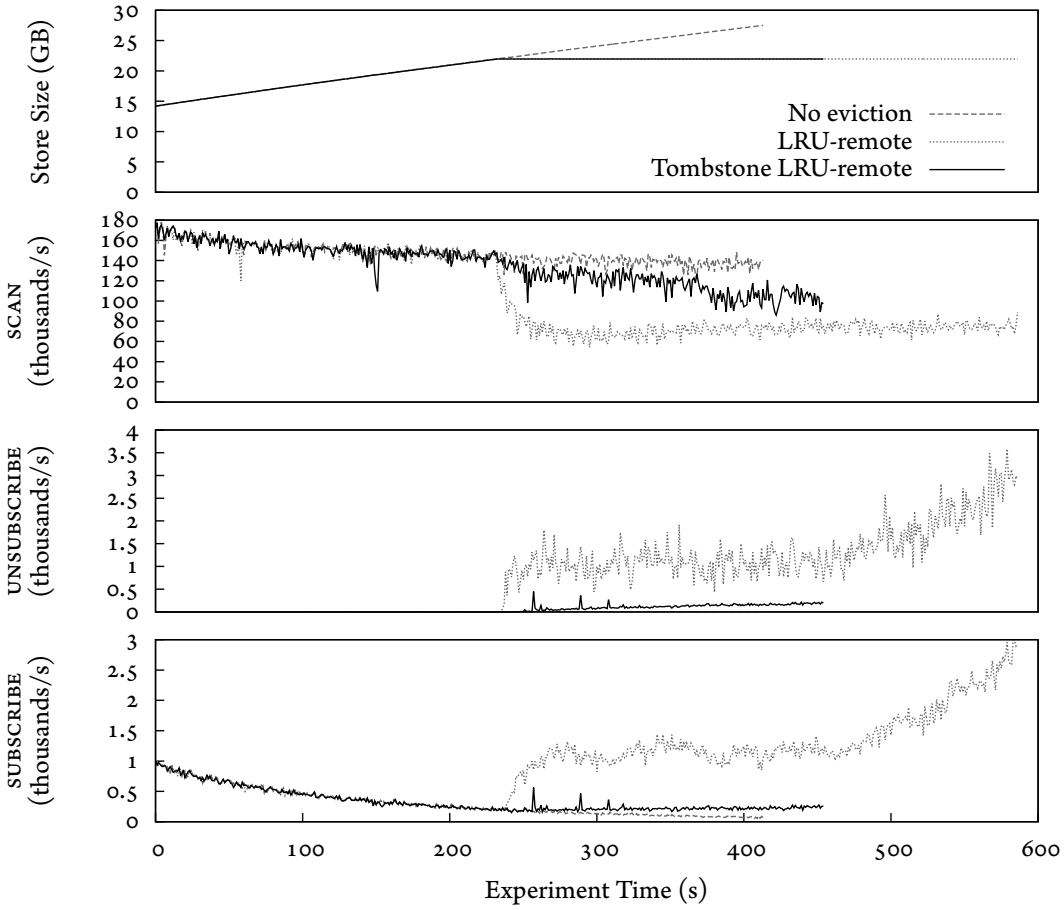

Figure 9.8: A timeseries of the TWIP-SMALL benchmark, executed with varying eviction parameters. The eviction threshold is crossed around 250 seconds. Tombstones mitigate eviction costs (excessive subscription changes between nodes) and improve overall performance (SCAN rate).

prior to crossing the eviction threshold (around 250 seconds into the experiment). After crossing, LRU-remote displays the expected behavior: remote ranges are evicted, subscriptions are canceled, and derived timelines are invalidated. The evicted data are reloaded (as evidenced by the number of new subscriptions made) when the invalidated timelines are recomputed. This churn continues through the end of the experiment, resulting in a significant drop in the SCAN (timeline checks) rate. The experiment with tombstones exhibits similar, albeit more subtle, behavior when eviction is triggered. Tombstones eliminate au-

tomatic subscription cancellations, so the large spike in UNSUBSCRIBE RPCs does not occur. The small number that do occur (nearly an order of magnitude fewer) suggest that tombstones are an effective mechanism for reducing eviction cost.

In summary, we show that range-based eviction is feasible, and that simple policies can be applied to guide range selection. We conclude that, for the Twip workload in a two-tier deployment, preferential eviction of remote ranges is superior to that of derived ranges and that tombstones are effective at mitigating eviction overhead. Though these conclusions are restricted in scope, these techniques show great potential for use in the context of other applications.

9.6 Materialization strategy

Pequod implements cache joins using a partial, dynamic materialization strategy: queries are computed on demand, but recently-accessed ranges are eagerly and incrementally updated. We compare this strategy with the obvious alternatives, namely no materialization (where no ranges are cached) and full materialization (where all ranges are cached and kept up to date).

We create a Twip workload comprising only timeline check and post operations. One million posts are distributed amongst all 1,800,000 users as described in §9.1 (proportionally to the log of the follower count). We then vary p, the percentage of active users, between 1 and 100. Each workload performs $p \times 1{,}000{,}000$ timeline checks spread uniformly across the $1{,}800{,}000 \times \frac{p}{100}$ active users, resulting in a check:post ratio between 1:1 and 100:1. We use five clients and one cache node, run the workload to completion as quickly as possible on the multiprocessor, and measure the elapsed time.

Figure 9.9 shows the results. As expected, the no-materialization strategy performs rel-

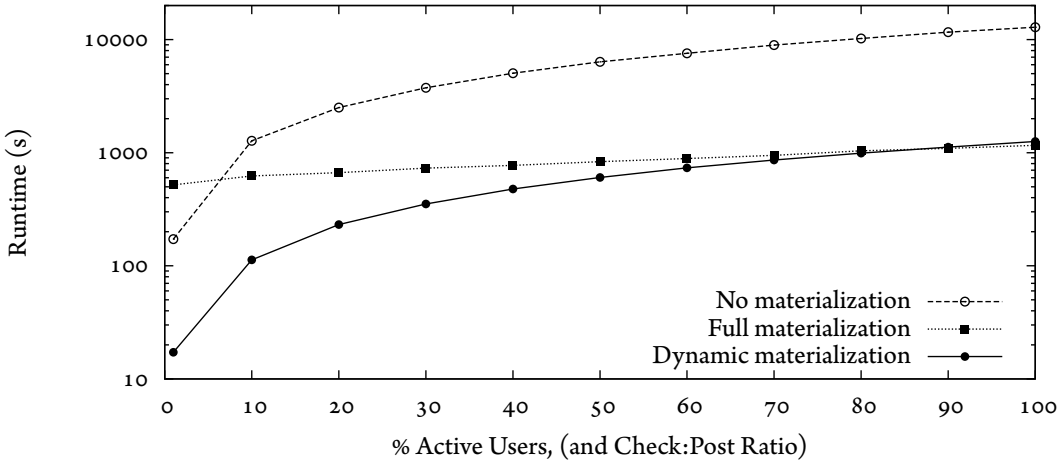

Figure 9.9: The dynamic materialization strategy is evaluated against two alternatives (no materialization and full materialization) using a variable Twip workload of timeline checks and posts. Smaller runtimes are better. Below 90% active users the dynamic policy is best, avoiding unnecessary materialization and conserving memory. Note the log scale axis.

atively well with few active users, but as timeline checks increase, materialization quickly becomes important for performance. Because it avoids materializing data in which no one is interested, Pequod's dynamic materialization outperforms full materialization up to approximately 90% active users. After that, full materialization performs slightly better (a factor of 1.08x better at 100% active users). This performance difference is due to the join computation that dynamic materialization must perform when a user first logs in. Full materialization keeps all timelines up to date at all times; though this avoids login overhead, it inevitably uses more memory to keep fresh the timelines of inactive users. For example, at 50% active users, the full materialization strategy consumes 1.39x more memory than the dynamic strategy in this experiment.

This experiment compares Pequod's dynamic materialization strategy against the logical extremes. While it is evident that this strategy is best amongst the tested alternatives for the Twip workload, it is not necessarily optimal. An application-specific materialization policy

could outperform the one-size-fits-all approach used in Pequod. For example, Silberstein et al. [38] espouse a per-user materialization strategy for feed-based applications like Twitter. Even if an application-specific strategy is not implemented, the developer might choose a different metric—like access frequency—to guide dynamic materialization. Evaluating these strategies is beyond the scope of this thesis.

9.7 Composition and tuning

Pequod leaves view selection and query planning to the application developer; this flexibility, and the design flexibility offered by the key-value context, can improve application performance. In this section we evaluate the effectiveness of interleaved cache joins (§5.7) and eager incremental maintenance (§5.6).

Mixed joins

We begin with an evaluation of mixed cache joins: the co-location of related (but disparately formatted) data into a single range. We use a Newp benchmark in this experiment. The Newp workload has three types of operation: reading an article, commenting, and voting. The Pequod data store is pre-populated with 100,000 articles, 50,000 users, 1,000,000 comments, and 2,000,000 votes. We simulate 20,000,000 user sessions in which each user

1. reads a random article;
2. with a varying chance votes on the article;
3. and independently with a 1% chance comments on the article.

We execute two versions of the Newp application. The first uses separate ranges for aggregate data (user karma and article vote count), requiring many RPCs in two round trips to construct an article. The second uses an additional set of joins to integrate article data

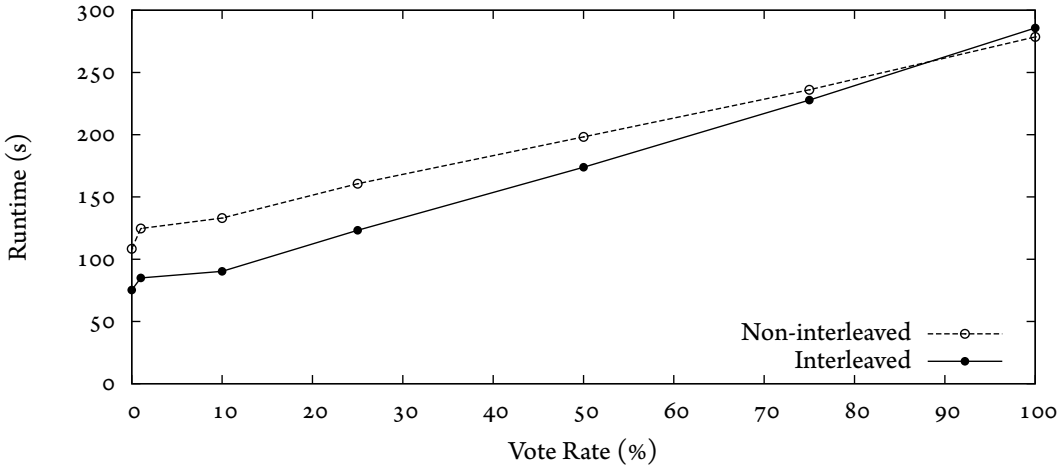

Figure 9.10: A comparison of Newp cache join compositions. Smaller runtimes are better. The version of the application that interleaves article data into a single data range outperforms the non-interleaved version until users vote on 90% of browsed articles.

into a single range. Reading an article requires a single RPC, but more server computation and storage overhead is incurred (upon each vote, aggregate values are copied into each page range). The interleaved Newp joins used in this experiment are discussed in §5.7.

The experiment is run on the multiprocessor using a single cache node. We measure the time required to execute the benchmark to completion, as quickly as possible. We expect the interleaved approach to perform well when article reads far outnumber votes and comments.

The results, shown in Figure 9.10, indicate that interleaved cache joins are superior for most vote rates tested. At 10% vote rate, the interleaved implementation outperforms the non-interleaved version by a factor of 1.47x. The non-interleaved implementation issues many GETs per browsed article, each of which incurs overheads including an $O(\log n)$ lookup. The interleaved join improves overall performance until the cost of pre-computation outweighs the cost of processing many GETs (90% vote rate).

Interleaved Newp cache joins are appealing: the application code is reduced to a single SCAN to retrieve an article and performance improves. However, there are costs associated with interleaving. As the experiment shows, the burden of maintaining these derived ranges can eventually outweigh the improved performance of article reads. Fortunately, writes are infrequent in this domain [19], so this is only a problem in theory.

Split joins

We next evaluate an application of join splitting: replacing the functionality of a single cache join with one or more joins (with differing execution or maintenance strategies). Celebrities are a problem for Twip; their posts are disseminated to a large number of followers' timelines, resulting in computation and storage inefficiencies. To remedy this, the Twip developer might split the timeline cache join so that celebrity posts are not pushed into user timelines, but are collected with a `pull` join at query time and merged into the SCAN results (§5.7).

We compare the performance of the split timeline join (5.11) with the original, single join implementation (5.3). We run the TWIP-SMALL benchmark on the multiprocessor using a single cache node. The benchmark is run to completion and the total execution time is recorded. The version with the split timeline join is slower by a factor of 2.47x.

Though this result is negative, it does not invalidate the concept of join splitting. It only demonstrates that splitting Twip timelines in this manner is not effective for this workload. The `pull` portion of the query adds a computational overhead to every timeline check that includes a celebrity (a fair proportion of users). The results in §9.6 clearly indicate that this overhead is significant. In addition, there is no improvement in memory utilization; the split join fails to reduce the size of the store. This result is likely a side effect of the Pequod

prototype implementation. In theory, the `pull` join need not insert any key-value pairs into the store (the results are only valid for a single request). However, our implementation stores the keys produced by `pull` joins (until the next SCAN), making it difficult to evaluate any potential improvements to storage utilization.

Eager incremental maintenance

Finally, we evaluate the effect of the `eager` performance annotation on the Twip timeline cache join. The annotated cache join

```
tline|user|time|poster =
  check eager sub|user|poster
  copy post|poster|time
```
(9.1)

prompts Pequod to process subscription changes eagerly, triggering incremental updates when a `sub` entry is added or removed. The default policy is to process such updates lazily, deferring updates to the next timeline check. Subscription changes are 10 times more frequent than new posts in our Twip workload. However, these changes affect only one user's timeline. By contrast, a new post will likely trigger updates to hundreds of derived ranges. As the primary source, these updates are always handled eagerly.

In theory, the eager policy will reduce latency for a fraction of timeline checks (those following a subscription change) by shifting the maintenance burden to the write path. This comes at a cost: historical entries may be generated to backfill the existing timeline. Such entries are unlikely to be viewed by the user and represent unnecessary computation and storage overheads.

We measure the effects of the `eager` annotation by executing the Twip-small benchmark against eager and lazy versions of the timeline cache join. The experiment uses a single cache node on the multiprocessor. The benchmark is executed as quickly as possible

and the total time to completion is used to compare the two policies. The results (of a single experiment run) slightly favor eager maintenance; the eager policy outperforms the lazy policy by a factor of 1.07x. As expected, the eager policy eliminates the 5 microsecond overhead on timeline checks after subscription changes (§9.3). However, the anticipated memory overhead is not observed.

In this experiment we observe the positive effects of eager maintenance, but very little of the ill effects. For the reverse side of the eager-lazy trade-off to be evident in the Twip workload, the cost of backfilling a user's timeline must be substantial. Unfortunately this is not the case for the Twip-small benchmark: it is simply too small to notice this effect. However, the trade-off remains; much larger deployments that keep weeks worth of timelines in the cache [25] would likely benefit from the lazy incremental maintenance policy.

9.8 Optimizations

We describe several implementation optimizations in §8.2 that were applied to the Pequod prototype. In this section we assess the performance impact of each.

We conduct a factor analysis to quantify the effects of each optimization individually. We measure the time required to execute the Twip-small benchmark to completion. We begin with an unoptimized version of Pequod and add optimizations one at a time, rerunning the benchmark for each. Five Twip clients are used to saturate a single cache node on the multiprocessor.

The results of the experiment are depicted in Figure 9.11. Insertion hints improve the benchmark runtime by a factor of 1.11x over the base implementation. Enabling value sharing results in negligible speedup, but reduces memory consumption by a factor of 1.14x. The largest improvement comes from enabling support for subtables. The Twip applica-

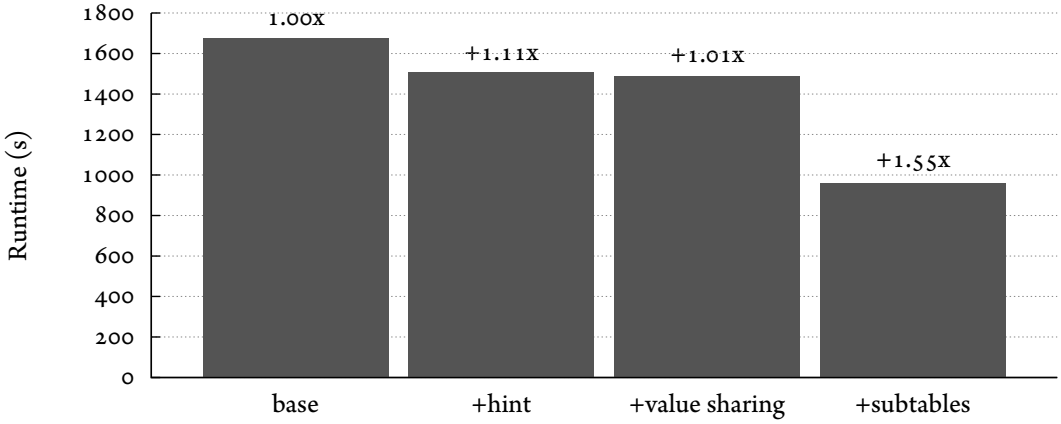

Figure 9.11: A factor analysis of implemented optimizations. Lower runtimes (higher speedups) are better. Insertion hints and subtables mitigate tree-based lookup overheads and improve performance by a factor of 1.11x and 1.55x, respectively. A negligible speedup is obtained from value sharing, but this optimization reduces memory consumption by a factor of 1.14x.

tion is well suited for this optimization because each user can be segregated into a separate subtable. Application operations are performed on a per-user basis, so all store lookups make use of the hash index at each level. Subtables further improve the benchmark runtime by a factor of 1.55x, but increase memory usage by a factor of 1.17x—a consequence of additional bookkeeping.

9.9 Summary

We show that Pequod cache joins can improve the performance of Web applications such as Twip and Newp. In-cache materialization outperforms application-managed materialization by a factor of 1.33x on our Twip workload. Further, Pequod scales to handle real-world Twitter workloads: a Pequod cluster with 18 compute nodes and 8 base nodes can process 13,000 new tweets and serve 1,300,000 timelines per second. The most significant barrier to scalability is replication overhead, which can be mitigated by the developer with careful partitioning and provisioning. Eviction cascades are a significant source

of overhead, if left unchecked. Fortunately, a combination of type-based eviction policies and tombstones are effective at mitigating the effects of cascades. Finally, we demonstrate that composing and tuning cache joins can lead to performance improvements in some applications.

10

Conclusion

This work studies a key component of modern Web application architectures, the application-level cache. We focus on the interface provided by the cache to developers. For popular caches, such as memcached and Redis, this interface is limited to application-directed fetch and store operations, often using a simple key-value data model. We recognize that some applications make extensive use of derived data types, and that computing and maintaining these data using the basic interface is cumbersome and inefficient.

Pequod addresses these deficiencies with cache joins, which introduce in-cache computation in the form of materialized views and provide a richer data model to the developer. With cache joins, developers can transfer complexity from application code to the cache. The only computation supported by Pequod is a natural join, which can be used to filter, join, and aggregate cached data. We observe that this small but powerful set of operations, especially when composed, can be used to express the core computations of some large Web applications. We find that moving these computations into the cache improves overall application performance by eliminating network operations and streamlining updates.

We chose the materialized view abstraction for its familiarity and simplicity. However, there exist fundamental differences in the data model and resource limitations between a relational database and an application-level cache. Many of the contributions of this work stem from the adaptation of this well-known abstraction to the context of a key-value cache.

We show that a relational data model can be layered atop a simple key-value model using relational overlays, and that these overlays can be used to define join computations. We further show that cache joins can be efficiently computed and automatically maintained with incremental updates. We address the issue of resource limitations in the cache by evaluating joins partially and dynamically, extending join execution across multiple servers with data partitions and subscriptions, and evicting data in a way that preserves derived data when possible.

Summary

From our perspective, Pequod is a success. Our prototype implementation outperforms the most popular systems on application benchmarks that make substantial use of derived data. Additionally, we find the cache join abstraction to be intuitive and freeing: a concise join definition installed into the cache eliminates a mess of cache maintenance code in the application. In our opinion, Pequod cache joins improve the programmability of Web applications. We hope that the ideas in this thesis encourage others, as they did us, to produce more usable software systems.

References

[1] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *Proc. SIGMOD'09, ACM SIGMOD Int'l Conf. on Management of Data*, pages 179–192. ACM, June 2009.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. VLDB'00, 26th Int'l Conf. on Very Large Data Bases*, pages 496–505. VLDB Endowment, September 2000.

[3] Khalil Amiri, Sanghyun Park, and Renu Tewari. A self-managing data cache for edge-of-network Web applications. In *Proc. CIKM'02, 11th Int'l Conf. on Information and Knowledge Management*, pages 177–185. ACM, November 2002.

[4] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. DBProxy: a dynamic data cache for Web applications. In *Proc. ICDE'03, 19rd Int'l Conf. on Data Engineering*, pages 821–831. IEEE Computer Society, March 2003.

[5] Chris Aniszczyk. Caching with Twemcache. Blog post. `https://blog.twitter.com/2012/caching-with-twemcache`, July 2012.

[6] Beevolve, Inc. An exhaustive study of Twitter users across the world. `http://www.beevolve.com/twitter-statistics/#b2`, October 2012.

[7] Jose A. Blakeley, Per-Åke Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proc. SIGMOD'86, 1986 ACM SIGMOD Int'l Conf. on Management of Data*, pages 61–71. ACM, May 1986.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[9] Boost Intrusive Data Structures. `http://www.boost.org/doc/libs/1_56_0/doc/html/intrusive.html`.

[10] R. F. Boyce, D. D. Chamberlin, M. M. Hammer, and W. F. King. Specifying queries as relational expressions. In *Proc. SIGPLAN'73, 1973 Meeting on Programming Languages and Information Retrieval*, pages 31–47. ACM, November 1973.

[11] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. INFOCOM'99, 18th Joint Conf. of the IEEE Computer and Communications Societies*, volume 1, pages 294–303. IEEE Computer Society, March 1999.

[12] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. CIDR'03, 1st Biennial Conf. on Innovative Data Systems Research*. VLDB Foundation, January 2003.

[13] 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 *Proc. OSDI'06, 7th USENIX Conf. on Operating Systems Design and Implementation*, pages 205–218. USENIX Association, November 2006.

[14] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[15] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[16] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endowment*, 1(2):1277–1288, August 2008.

[17] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, The Internet Engineering Task Force, July 2006. `http://tools.ietf.org/html/rfc4627`.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP'07, 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220. ACM, October 2007.

[19] Jeremy Edberg. Scaling Reddit from 1 million to 1 billion -– pitfalls and lessons. Talk at RAMP Conf. `http://www.infoq.com/presentations/scaling-reddit`, August 2013.

[20] Facebook. `http://facebook.com`.

[21] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. In Ashish Gupta and Iderpal Singh Mumick, editors, *Materialized Views*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[22] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD'93, 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pages 157–166. ACM, May 1993.

[23] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. SIGMOD'08, ACM SIGMOD Int'l Conf. on Management of Data*, pages 981–992. ACM, June 2008.

[24] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. Eurosys'07, 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pages 59–72. ACM, March 2007.

[25] Raffi Krikorian. Real-time delivery architecture at Twitter. Talk at QCon New York. `http://www.infoq.com/presentations/Real-Time-Delivery-Twitter`, October 2012.

[26] Raffi Krikorian. New tweets per second record, and how! Blog post. `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`, August 2013.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proc. WWW'10, 19th Int'l World Wide Web Conf.*, pages 591–600. ACM, April 2010.

[28] P.-Å. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB'85, 11th Int'l Conf. on Very Large Data Bases*, pages 259–269. VLDB Endowment, August 1985.

[29] Gang Luo. Partial materialized views. In *Proc. ICDE'07, 23rd Int'l Conf. on Data Engineering*, pages 756–765. IEEE Computer Society, April 2007.

[30] memcached. `http://memcached.org`.

[31] Memtier Benchmark. `https://github.com/RedisLabs/memtier_benchmark`.

[32] MessagePack. `http://msgpack.org`.

[33] Domas Mituzas and Mark Konetchy. Facebook Tech Talk: MySQL & Hbase. `http://livestre.am/1aeeW`, December 2011.

[34] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. NSDI'13, 10th USENIX Conf. on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, April 2013.

[35] 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 *Proc. OSDI'10, 9th USENIX Conf. on Operating Systems Design and Implementation*, pages 1–15. USENIX Association, October 2010.

[36] Reddit. `http://reddit.com`.

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

[38] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *Proc. SIGMOD'10, ACM SIGMOD Int'l Conf. on Management of Data*, pages 831–842. ACM, June 2010.

[39] Tamer. `https://github.com/kohler/tamer`.

[40] Twitter. `http://twitter.com`.

[41] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. TAO: How Facebook serves the social graph. In *Proc. SIGMOD'12, ACM SIGMOD Int'l Conf. on Management of Data*, pages 791–792. ACM, May 2012.

[42] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. Lazy maintenance of materialized views. In *Proc. VLDB'07, 33rd Int'l Conf. on Very Large Data Bases*, pages 231–242. VLDB Endowment, September 2007.

[43] Jingren Zhou, Per-Åke Larson, and Jonathan Goldstein. Partially materialized views. Technical Report MSR-TR-2005-77, Microsoft Research, 2005.

[44] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *Proc. ICDE'07, 23rd Int'l Conf. on Data Engineering*, pages 526–535. IEEE Computer Society, April 2007.