Easy Freshness with Pequod Cache Joins

Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula*, Yandong Mao*, Robert Morris*
*Harvard University and *MIT CSAIL*

Abstract

Pequod is a distributed application-level key-value cache that supports declaratively defined, incrementally maintained, dynamic, partially-materialized views. These views, which we call *cache joins*, can simplify application development by shifting the burden of view maintenance onto the cache. Cache joins define relationships among key ranges; using cache joins, Pequod calculates views on demand, incrementally updates them as required, and in many cases improves performance by reducing client communication. To build Pequod, we had to design a view abstraction for volatile, relationless key-value caches and make it work across servers in a distributed system. Pequod performs as well as other in-memory key-value caches and, like those caches, outperforms databases with view support.

1 Introduction

Web developers use application-level key-value caches such as memcached [2] to improve the performance of database-backed sites. Caches can store *base* data, meaning copies of database records; this improves performance by offloading reads from a bottleneck persistent store. More benefit can be gained by caching *computed* data, which derives from base data but is organized more conveniently for readers. Unfortunately, neither approach is easy to program. Twitter and Facebook, for instance, organize their persistent stores by posting user and time [12, 20], but their primary read operations combine and filter many users' data streams into "timelines" based on user subscriptions. A base-data cache for this access pattern would make reads difficult: the application would basically design and execute a query plan on the cache. A computed-data cache that stored the results of complex timeline queries would simplify reads, but complicate writes: developers would have to invalidate or update cached computed results as necessary to maintain freshness. Developers have even responded to these challenges by building application-specific caching systems [12].

Pequod is a general-purpose distributed key-value cache that transparently keeps computed results up to date. Its central idea is the *cache join* abstraction, which compactly expresses the transformations required to turn input cached data into computed results. The developer installs cache joins in advance. Pequod uses cache joins both to compute data requested by clients, and to update cached results in response to updates. Pequod handles the complex task of updating cached computed data with little developer effort.

A cache join is a materialized view [22] implemented in a distributed key-value cache rather than a relational database. It declaratively defines computed data in terms of simple transformations of base data. We show that materialization, where the cache stores computed data and keeps it up to date, is important for performance. Since cached data is by definition partial, Pequod's materialized views must be both partial and dynamic [28, 30]. Cache joins also offer control over data freshness, supporting both periodic snapshots and incremental updates that keep computed data fully up to date [19]. Pequod thus combines many advanced features, supporting in one system distributed, incrementally maintained, both eager and lazy, dynamic, partially-materialized views. Although Pequod builds on implementation strategies from database materialized views, the non-relational, distributed, key-value-cache context required changes both to the cache join abstraction and to its implementation.

Cache joins help Pequod improve application performance. Computing a result often involves reading extra data; executing this computation in servers, rather than clients, reduces network traffic. Pequod also uses dependency records and hints to efficiently maintain its cache in response to updates.

Our work offers several contributions. We observe that simple joins, filters, and aggregations can express relationships among cached data for many applications, and that can be applied to a distributed, ordered key-value cache. We describe the *cache join* abstraction for key-value materialized-view-like queries and query execution plans. We provide efficient policies for computing these cache joins based on application queries. We design a distributed system that supports cache joins using cross-server data subscriptions and update notifications. Finally, we evaluate Pequod on two example applications inspired by popular websites, a Twitter-like microblogging service and a Hacker News-like news aggregator with user karma. We compare our system with existing technologies and find that Pequod preserves key-value cache performance, despite the addition of cache join execution. We show that moving computation into

cache servers can improve overall system performance, and demonstrate that a deployment of Pequod can be scaled to handle Web-class workloads.

2 Design

Pequod is an ordered key-value cache with string keys and values. It supports four basic operations: get(*k*) returns a value; put(*k*, *v*) updates a value; remove(*k*) removes a value; and the ordered scan(*first*, *last*) operation returns a lexicographically-ordered list of those key-value pairs with keys in the given range. Pequod is not a database and, as is usual for key-value caches, it doesn't support multi-key transactions.

To support freshness, base data in a Pequod cache must be kept up to date relative to the persistent backing store (typically a database). A convenient way to do this is to connect Pequod with a database shard, instructing Pequod that some keys can be found in the database and instructing the database that updates to relevant tables should be forwarded to Pequod (e.g., using Postgres's `notify` statement). If a request is made for a database-sourced key, Pequod will query the database and cache the result, and the database will keep Pequod abreast of any changes. Pequod thus acts as a write-around cache: application writes go directly to the database, and applications access Pequod only for reads. We describe the design of Pequod using a write-around deployment, though other deployments are possible (such as write-through or lookaside caching).

We describe Pequod with reference to Twip, an application that models the core of Twitter, but its ideas apply to many applications that use materialized views.

2.1 Caching Twip

A Twip user can *post* tweets, *follow* other users (subscribe to their tweets), and check her *timeline*. This last operation is the most complex: when user `ann` checks her timeline, Twip returns, in a time-sorted list, all recent posts made by any user `ann` follows.

A Twip database store might use two tables, `p` for posts and `s` for subscriptions. `p`'s columns would be `poster` (the user ID of the posting user), `time`, and `tweet`; `s`'s columns would be `user` (the ID of the subscribing user) and `poster` (the ID of the followed user). This query satisfies a timeline check by user `ann` for all interesting tweets posted after time 100:

```
select p.time, p.poster, p.tweet from s, p
   where s.user='ann' and s.poster=p.poster
     and p.time>=100 order by p.time;
```

Timeline checks are frequent, routinely outnumbering new posts by a factor of 100 [20]. They are also expensive. As the query makes clear, a single timeline check must collect information from all a user's subscriptions, and Twitter users average more than 100 subscriptions each [9]. Executing frequent, expensive, latency-sensitive queries on a persistent database is inadvisable. Timeline checks must be cached.

The cache could simply hold base data. For example, the key-value pair `p|bob|100` ↦ `Hi` might represent user `bob`'s tweet of `Hi` at time 100, and the pair `s|ann|bob` ↦ `1` might represent `ann`'s subscription to `bob`. However, a base-data-only cache shifts the query planning burden onto the application. To check `ann`'s timeline, Twip must read her subscriptions with a scan on the `s|ann|` range, and then, for each subscription, scan the corresponding `p|` range for tweets. This involves at least two rounds of RPCs (and, for typical users who follow many others, hundreds of RPCs total).

To simplify timeline checks, the cache could store computed data. Keys starting with `t|ann|` could store `ann`'s timeline, with copies of tweets by the users `ann` follows. For instance, the `bob` tweet above would correspond to the key-value pair `t|ann|100|bob` ↦ `Hi`. The order of components in this key is semantically important and follows from the lexicographic order of the scan operation. Since each user has their own timeline and tweets are sorted by time, the user and post time must follow the `t|` prefix in that order. We assume multiple followed users might post tweets at the same time, so the key includes the poster to disambiguate them. `Ann`'s timeline check at time 100 would be implemented by a single scan on the half-open range [`t|ann|100|`, `t|ann|`$^+$).[1] This resembles Redis's demonstration Twitter [4] and to some extent the actual Twitter [20], though these services store user timelines as Redis list values (since Redis is not an ordered store) and Twitter stores tweet IDs rather than texts.

Timeline checks are certainly simpler when computed data is cached, but posts and subscriptions are more complicated. Any post must update all relevant timelines. Performance will be good—shifting overhead from read operations to writes is often useful—but implementation remains complex. Furthermore, since the cache might evict a timeline, the application must still contain code to construct timelines from posts.

2.2 Basic cache joins

Pequod aims to help applications avoid this complexity. A Twip application using Pequod can write posts to the database and read timelines directly from the cache without worrying about subscriptions, and with performance as high as that of a typical application cache. The key

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

idea is Pequod's declarative *cache join* specifications, which relate computed data (such as timelines) to base data (such as posts and subscriptions). The Twip **timeline cache join**

```
t|user|time|poster = check s|user|poster
    copy p|poster|time;
```

defines the value of `t|`*user*`|`*time*`|`*poster* as a copy of the value of `p|`*poster*`|`*time* whenever `s|`*user*`|`*poster* exists. Cache join semantics are that of a simple SPJ database query; the syntax is inspired by Datalog [14] "`t`(*user*, *time*, *poster*, *tweet*) :- `s`(*user*, *poster*), `p`(*poster*, *time*, *tweet*)." However, where Datalog and database queries treat all columns alike, cache joins distinguish keys and values. The poster's tweet, which is the *tweet* column in Datalog and SQL, is implicitly referenced in Pequod by the cache join's `copy` operator.

Pequod computes cache joins using strategies effective for application caches. When the timeline cache join is installed, Pequod contains no `t|` keys. Instead, Pequod dynamically materializes the required results in response to scans of the `t|` range. A timeline scan on [`t|ann|100`, `t|ann|`$^+$) would (in a write-around deployment) ensure that the `s|ann` subscriptions and all relevant `p|` posts were cached, then join those keys to produce the relevant timeline. In addition to returning to the client, Pequod caches the computed timeline and installs updaters that keeps it up to date. If `bob` tweets again at time 120, the database will notify Pequod, which will put the new tweet into key `p|bob|120`. This put triggers a process that automatically copies the tweet to key `t|ann|120|bob`. Since Pequod is eagerly and incrementally maintaining `ann`'s timeline, her later timeline checks can execute without additional computation. Pequod also handles subscription changes: modifications to the `s|ann|` range cause incremental recomputation.

2.3 Advanced cache joins

Cache joins can specify query plans and types of maintenance as required by the application. For example, Twitter celebrities can have tens of millions of followers. Copying their posts into so many timelines will use a lot of memory. Since there are relatively few such celebrities and relatively few celebrity posts, it can be more memory-efficient to handle their posts in a different way [26]. For instance:

```
ct|time|poster = copy cp|poster|time;
t|user|time|poster = check s|user|poster
    copy p|poster|time;      // (1) non-celeb, same as above
t|user|time|poster = pull copy ct|time|poster
    check s|user|poster;     // (2) celebrity
```

Here each user's timeline is calculated from *two* cache joins with different execution strategies. The first, for non-celebrities, is eagerly maintained. The second, for celebrities, is not: the `pull` annotation tells Pequod to recompute the join on each request without caching the results. Celebrity users store their posts under `cp|` keys, rather than the usual `p|`; a helper range, `ct|`, combines all celebrity posts in time-primary order. To compute the celebrity portion of a timeline, Pequod checks the `ct` range for celebrity posts, then filters the results through the user's subscriptions. If no celebrities have posted in a time range (a common case), Pequod will complete the celebrity join with a single fast scan. In our tests, celebrity timelines don't offer performance advantages, but they do save memory.

Cache joins can also colocate different classes of values into the same range of keys. This powerful use case doesn't fit easily into a relational model. For instance, consider a Hacker News-like news application with user karma [1]; we call our version Newp. Users can author new articles, comment and vote on articles, and read article pages. An article page shows the article's vote count, its comments, and the "karma" for each user who commented on the article, where a user's karma is the count of votes on the articles that user authored. A cache for Newp might store articles in one key range, comments in another, and votes in another. Karma, which involves an expensive sum over all of a user's votes, should be precalculated and cached as well. Thus, the data necessary to render an article would be spread over many key ranges. But in Pequod, separate cache joins can bring these disparate objects into a single `page|` scan range, as shown in Figure 1. As a result, Newp can issue one scan on [`page|bob|101`, `page|bob|101|`$^+$) to retrieve all of the disparate data needed to render an article page. We call cache joins like this *interleaved* since they interleave results from logically different computations in a single output range.

Newp also shows that Pequod cache joins can aggregate input data. The `karma` join uses a `count` operator to reduce a range of inputs (all keys starting with `vote|`*author*) into a single value, namely the count of the number of inputs. Pequod supports several simple aggregation functions, including `sum`, `count`, `min`, and `max`. Aggregated data is kept up to date just like copied data, and aggregations are easy to add.

2.4 Distributed Pequod

Distributed Pequod supports workloads too large, or too compute-intensive, for a single server to handle. Base data is replicated across servers as necessary to support the maintenance of computed data (cache join outputs). Each base key has a *home server* to which updates are directed (a partition function maps key ranges to home servers). When a base key k is read from a server S other than its home server H, S requests k's value from

```
karma|author = count vote|author|id|voter;
rank|author|id = count vote|author|id|voter;
page|author|id|a = copy article|author|id;
page|author|id|r = copy rank|author|id;
page|author|id|c|cid|commenter =
  copy comment|author|id|cid|commenter;
page|author|id|k|cid|commenter =
  check comment|author|id|cid|commenter
  copy karma|commenter
```

Figure 1: Interleaved cache joins bring the data necessary to render a Newp article into one contiguous range. Key tags like `|a` and `|r` help the application distinguish types of data.

H. In addition to returning the value, H installs a *subscription* for S to k. When H receives an update to k's value, it will send the new value to S. Pequod thus maintains eventually-consistent replicas of base data. Computed data is distributed across servers according to client requests. To compute a cache join, a server first fetches all relevant base data into memory (possibly accessing home servers or the backing store, and possibly working through intermediate cache joins), then runs without further communication.

Since cache joins can execute anywhere, adding servers to a Pequod deployment increases its computational capacity. Base data subscriptions also make replication-based load balancing possible: directing reads for popular data ranges to multiple Pequod servers establishes incrementally-maintained replicas that can distribute query load. Adding more servers to a deployment also increases the system's storage capacity, but due to the data duplication and subscription overhead inherent in our design, a Pequod cache's storage capacity does not necessarily rise linearly with the number of servers. Data duplication is reduced when clients send their requests to appropriate servers. Twip clients avoid redundant timeline storage by sending all timeline checks for user u to a specific server $S(u)$. This is especially important since timelines make up the large majority of system data (each tweet is stored once in the poster's `p|` range and many, many times in followers' `t|` ranges). But some duplication is unavoidable—for instance, a popular user's tweets are copied to all servers, inducing network overhead—and other applications might be unpartitionable.

Our initial design goal for distributed Pequod has been simplicity rather than completeness, and we do not focus on consistency or resilience to failure. Pequod is eventually consistent: every update to base data eventually becomes visible to all interested servers, but since update propagation is asynchronous, different servers 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). Many Web applications are tolerant of this kind of inconsistency. For some applications, the current Pequod prototype 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, clients must read from and write to a single Pequod cache server (base data are written directly to Pequod to avoid the asynchrony of database notification).

2.5 Eviction

Pequod can evict data under memory pressure. Three types of data can be evicted: data computed by a cache join, remote data copied from another Pequod server via subscription, and cached base data, which in our expected deployment is loaded on demand from a database. Eviction requires modifying the store and invalidating all dependent computed data (which can have transitive effects). At present, an overloaded Pequod server simply evicts the least recently used data ranges. This could be improved by considering the expected costs of reloading a range (the latency of fetching base data from the database, the CPU cost of recomputing dependent ranges, and so forth).

2.6 Discussion

Pequod simplifies application design by adding intelligence to the caching layer. But databases already support that intelligence; why not just use one? Applications couldn't afford to use the main persistent database, but perhaps, as in DBProxy [7, 8], another relational database could be used as a cache. This kind of deployment would be an excellent choice if it performed well. Since Web applications already rely on key-value cache performance, and since some of that performance derives from the key-value model, we chose instead to add intelligence to a key-value cache.

3 Cache joins

A cache join declares how to calculate some output key-value pairs from input key-value pairs, which we call *sources*. Since cache joins are expressed in terms of key-value pairs, which have a single meaningful index ordering (namely, lexicographic key ordering), cache joins also expose meaningful server performance properties, making them resemble both database views and query plans.

A cache join specification has four parts. (1) An *output pattern*, such as `t|`*user*`|`*time*`|`*poster*, defines the format of output keys. (2) One or more *source patterns*, such as `copy p|`*poster*`|`*time*, select keys whose values are used to compute results, and define the operators applied to these keys. (3) Optional *performance annotations* (including the order of source patterns) guide query exe-

```
<cachejoin> ::= <key> "=" ["push" | "pull" |
                "snapshot <T>"] <sources>;
<sources>   ::= <source> | <sources> <source>;
<source>    ::= <operator> <key>;
<operator>  ::= "copy" | "min" | "max" | "count"
                | "sum" | "check";
```

Figure 2: Cache join grammar.

cution; see §3.4. (4) *Slot definitions* tell Pequod how to unpack a key into its component slots—for example, by looking for vertical bars, or by taking fixed numbers of bytes. We don't explain slot definitions further.

Pequod supports several source operators. `Copy` tells Pequod to copy the corresponding source's value to the output key. `Check` marks sources whose values aren't interesting. (For example, in the timeline join, subscriptions like `s|`*user*`|`*poster* are used only for the contents of their keys.) There are also several aggregate functions; like SQL's aggregate functions, they combine many source values into a single output.

Unlike a database query, a cache join exposes the performance properties of key ordering (in relational databases index structure is specified elsewhere), and exposes more performance properties through annotations, but has less flexibility. For instance, cache joins must not be recursive (a cache join's output cannot be used as one of its sources), and relationships between "tables" must be expressed entirely through keys.

Users define cache joins in textual form (Figure 2 summarizes the grammar) and install them using an "add-join" RPC. Multiple cache joins may be installed over the same range of keys. One cache join can act as a source for another; this can be useful, for example to permute keys into a more convenient order, but risks cascading invalidations. Performance annotations such as `snapshot` *T* (§3.4) can mitigate this problem somewhat. Users should not install circular cache joins.

Pequod checks for errors (such as recursive queries) at cache join installation time, but some errors are difficult to identify in advance. For instance, consider the timeline join variant `t|`*user*`|`*time* `= check s|`*user*`|`*poster* `copy p|`*poster*`|`*time*. Since this lacks the "`|`*poster*" suffix for timeline keys, it produces undefined results when two or more posters post tweets 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 doesn't know how to combine multiple values into a single string.) But it's not necessarily appropriate to reject such joins out of hand; perhaps the application ensures that no two tweets have the same time. Thus Pequod's users are left responsible for avoiding ambiguous cache joins, either by preventing output collisions or by using aggregate functions with well-defined behavior.

We currently impose additional technical requirements on cache joins. For instance, in a join with n sources, exactly $n-1$ of the operators must equal `check`, and we constrain the use of `count` and `sum` operators to simple cases.

3.1 Cache join query execution

When Pequod receives a scan(*first*, *last*) or get(*key*) request that overlaps with one or more cache joins, it must execute the queries they represent. This section describes the semantics and implementation of cache join query execution. We focus on forward query execution, which starts from base data. The algorithm used by Pequod is a key-value variant of the classical nested-loop implementation for database join queries. The next section describes how incremental maintenance works.

Cache join execution logically enumerates all tuples of source keys and selects those that match the join's constraints. The selection step ensures that there's one key per source, that the key formats match the source patterns, and that slots common to multiple source keys have consistent values. Pequod then combines the selected values according to the join's operators and installs the results.

For the timeline join `t|`*user*`|`*time*`|`*poster* `= check s|`*user*`|`*poster* `copy p|`*poster*`|`*time*, the keys ⟨`s|ann|bob`,`p|bob|100`⟩` match the source patterns: the first key matches the first source, the second key matches the second source, and the shared *poster* slot has a consistent value in both keys (`bob`). An output key can be derived from the output pattern and the source keys; here, that key would be `t|ann|100|bob`. The join's operators (`check` for the `s` source and `copy` for the `p` source) indicate that `p|bob|100`'s value should be copied to `t|ann|100|bob`.

The main optimization strategy for nested-loop queries moves selection operators as early as possible, since this avoids enumerating irrelevant tuples. In relational databases, selection operators are functions on columns. In Pequod, selection operators are functions on *ranges*, especially as they map to the "columns" defined by pattern slots. For example, when given a scan query over the timeline range [`t|ann|100`,`t|ann|200`), Pequod can limit its examination of subscriptions to the range [`s|ann|`,`s|ann|`$^+$); and for each resulting subscription `s|ann|`*poster*, it can examine the limited post range [`p|`*poster*`|100`,`p|`*poster*`|200`).

Figure 3 outlines our query execution algorithm; though our implementation is generic, the pseudocode uses the timeline join for concreteness. Two related concepts, slot sets and containing ranges, help move selection early. A *slot set* is a set of slot assignments derived from a cache join and a key or key range. For example, in

compute timeline(*first*, *last*):
  ss := *timelinejoin*.slotset(t, *first*, *last*)
  $[ks^-, ks^+)$:= *ss*.containingrange(s, *first*, *last*)
  for each $ks \mapsto vs$ where $ks \in [ks^-, ks^+)$
      and *ks* matches s|*user*|*poster*:
    ss' := *ss*.addslots(s, *ks*)
    $[kp^-, kp^+)$:= ss'.containingrange(p, *first*, *last*)
    for each $kp \mapsto vp$ where $kp \in [kp^-, kp^+)$
        and *kp* matches p|*poster*|*time*:
      yield t|*user*|*time*|*poster* $\mapsto$ *vp*

Figure 3: Query execution for the timeline cache join.

the timeline join, the key s|ann|bob corresponds to the slot set {*user* $\mapsto$ ann, *poster* $\mapsto$ bob}. Query execution begins by deriving a slot set from the requested output key range; for example, given scan(t|ann|100, t|ann|+), Pequod creates the slot set {*user* $\mapsto$ ann}. Slot sets are augmented with additional slot assignments as Pequod works through source keys. A *containing range* is effectively the inverse of a slot set. Given a slot set, a source pattern, 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 the scan request above and the slot set {*user* $\mapsto$ ann, *poster* $\mapsto$ bob}, the minimal containing range for the p source would be [p|bob|100, p|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. But even with containing ranges, the algorithm must compare the source range keys with their patterns. As a schema-free key-value store, Pequod might have keys in the range that don't match the source patterns. Figure 4 shows a sample execution of this algorithm for the timeline join.

Since Pequod should support any application and provide general key-value cache semantics, we took care to handle any range query. Thus, for example, we correctly implement queries like [t|ann|100, t|bob|200) and [t|a, t|b) that cross multiple timelines. Correct and minimal containing ranges are generated in each case.

3.2 Incremental maintenance

Pequod can keep computed data up to date as sources change. Eager incremental maintenance transfers work from read queries to writes and saves computation on exact re-requests. The view selection problem [22] is easy in Pequod: since all queries are range scans, precomputed ranges naturally benefit queries that partially or completely overlap.

Pequod implements incremental maintenance through two auxiliary data structures. A *join status range* indicates whether a range of keys is up to date with respect to the cache joins whose outputs overlap that range. Join

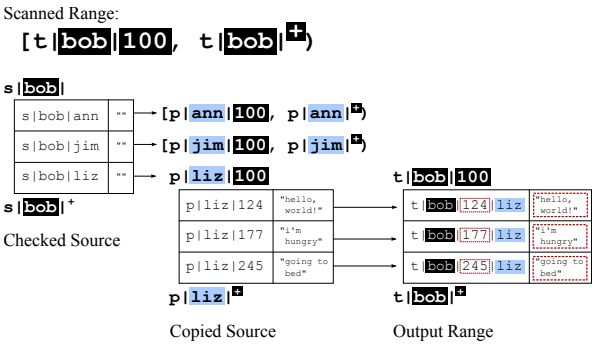

Figure 4: Example query execution of the timeline join. The scanned range provides context used when scanning source ranges. The keys and values in the output key range comprise elements of the original scan range and both source ranges.

compute timeline(*first*, *last*):
  for each subrange $[x^-, x^+) \subset [first, last)$ where
      joinstatus($[x^-, x^+)$) $\neq$ VALID:
    ss := *timelinejoin*.slotset(t, x^-, x^+)
    $[ks^-, ks^+)$:= *ss*.containingrange(s, x^-, x^+)
    js := new join status range for $[x^-, x^+)$
    add updater from $[ks^-, ks^+)$ to *js*
    for each $ks \mapsto vs$ where $ks \in [ks^-, ks^+)$
        and *ks* matches s|*user*|*poster*:
      ss' := *ss*.addslots(s, *ks*)
      $[kp^-, kp^+)$:= ss'.containingrange(p, x^-, x^+)
      add updater from $[kp^-, kp^+)$ to *js*
      for each $kp \mapsto vp$ where $kp \in [kp^-, kp^+)$
          and *kp* matches p|*poster*|*time*:
        yield t|*user*|*time*|*poster* $\mapsto$ *vp*
    install *js* as VALID

Figure 5: Query execution for the timeline cache join, including installation of data structures for later incremental maintenance. Changes from Figure 3 are in black.

status ranges are logically attached to output ranges and form a disjoint cover of key space (every key is associated with exactly one join status range). *Updaters*, in contrast, logically attach to source ranges. An updater links a range of source keys with a *context*—a cache join, a slot set, and a join status range. The context provides the information required to maintain its join status range. For instance, a range of posts [p|bob|100, p|bob|+) might have an updater for the timeline join with slot set {*user* $\mapsto$ ann}; this *user* value lets Pequod map source key p|bob|200 to output key t|ann|200|bob. Many updaters can apply to a given key, so we store updaters in an interval tree. Whenever Pequod modifies its store, it finds all updaters applicable to the modified key and runs the indicated incremental maintenance for each. Figure 5 outlines how join status ranges and updaters are installed during forward query execution.

The notification provided to an updater includes the modified source key, the new value, the old value, and the type of change (insert new key, update existing key, or remove existing key). The updater reacts by modifying either the attached join status or the cache's value for some key. The form of this modification depends on the relevant operator; for example, the updater for `copy` operators calculates the appropriate output key and inserts the value there.

Pequod also supports *invalidations*, which provide a form of *lazy* view maintenance [29]. Unlike an eager updater, which updates the cache upon notification, an invalidating updater just marks its join status range as INVALID. The invalidity will be detected and corrected when the output range is queried. There are two kinds of invalidation. Complete invalidation removes installed updaters and requires that a range be recomputed from scratch. Partial invalidation instead logs the source modification into an entry on the relevant join status range. The logged modification—or a subset of it—will be applied later, when the output range is queried [29]. Lazy maintenance, and especially partial invalidation, shifts some of the burden of view maintenance back onto read operations from write operations. Our prototype uses lazy maintenance (invalidations) for `check` sources and eager maintenance for all other sources, a choice that performs well for our applications. For example, Twip subscription changes logically shift many tweets into or out of a timeline. Thanks to lazy maintenance, however, Pequod shifts only those tweets strictly required by queries. Since most timeline checks are updates, rather than loads of past tweets, this can perform much less work than eager maintenance would. Our policy would not work equally well for all applications, however, and we would like to offer users more control over maintenance type.

Several important optimizations improved Pequod's performance by large factors. Updaters frequently overlap; for example, a Twip user's posts have one updater per subscriber. It was especially important to combine such updaters whenever possible. If a new updater is installed for the same source range as an existing updater—or for an overlapping range—Pequod reduces memory usage and the size of the updater tree by appending information about the new updater to the existing one. Other important performance improvements were obtained by compressing or eliminating the context information stored with updaters, since in many cases Pequod can derive an output key completely from the source key and the relevant join status range. (Consider a timeline-join updater on source range [`p|ren|200`,`p|ren|`$^+$) associated with join status range [`t|ann|100|`,`t|ann|`$^+$). The join status range uniquely determines the *user* slot, and the source key uniquely determines the *poster* and *time* slots.)

3.3 Resolving missing data

The join query execution algorithm in Figure 5 assumes that all required source data is present in the cache. However, this assumption need not hold. A source range might overlap with the output of another cache join; the source range might exist in the persistent backing store, but have expired from the cache; and in distributed Pequod, the source range might be stored on a different server. Pequod detects these cases and loads the data as required before executing the join query. The first case can be handled with a recursive query execution. In the second and third cases, the data is loaded and metadata is installed to indicate its presence.

Pequod reduces query latency by loading missing base data in parallel. When base data is missing, Pequod initiates an asynchronous fetch request (to the database or to a home server) and attaches a *restart context* to the current join status range. It then continues to execute the query using any cache-resident data. When all required fetches complete, the restart contexts are used to restart the query. The restarted query behaves as if executed from scratch, so every server query produces results consistent with a snapshot of that server's state. However, Pequod doesn't recompute any parts of the query that were already completed and haven't been invalidated since. In general, a query execution is iteratively evaluated until there are no outstanding restart contexts and the join status range is marked VALID; in most cases, this requires at most one round of fetches and a cleanup execution that fills in gaps. The resulting output range is scanned to construct a client response.

We always resolve missing base data by loading it into server memory. This allows us to execute cache joins locally. Other strategies, such as executing cache joins in partitioned fashion and aggregating the results [5], could benefit some applications by reducing data movement.

3.4 Performance annotations

Cache joins contain annotations that affect evaluation performance. First, the *maintenance type* defines whether and how a join is kept up to date. The default `push` type asks for the incremental maintenance described in §3.2. A `pull` join, in contrast, is calculated from scratch on each query using the procedure in §3.1. As we saw for Twitter celebrities (§2.3), this can save memory for some applications and data patterns. Finally, a `snapshot` T join implements deferred view maintenance. Pequod calculates the join from scratch, but caches the result—without further updates—for T seconds. Snapshot joins induce less maintenance overhead than push joins and less computation overhead than pull joins. Second, the order of sources is a performance annotation. Pequod examines the source descriptions in a cache join in order, and different source orders can perform quite differ-

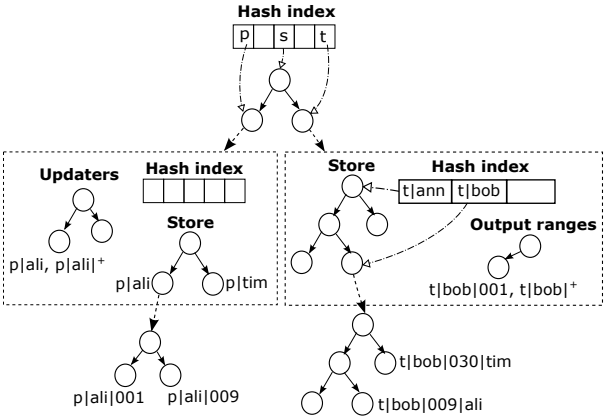

Figure 6: Pequod internal data structure for Twip. The logical store divides into tables (the rectangle layer) and, when appropriate, subtables (the lowest layer).

ently. Joins are generally more efficient when the smallest ranges are examined first, since this reduces the data to be examined later. It's usually best to arrange sources so that slots are encountered in the same order as in the output key, since this leads to small containing ranges. The user may be able choose an even better order using their knowledge of key counts and update frequencies.

4 Implementation

Pequod is a single-threaded, event-driven C++ program. Pequod uses red-black trees to store key-value pairs and bookkeeping information, such as updaters and join status ranges. Several optimizations use auxiliary data structures such as hash tables to reduce tree lookups, improving performance significantly for some workloads. This section describes Pequod's implementation, focusing on these optimizations and when they are useful. Figure 6 shows the overall arrangement of Pequod's internal data structures as configured for Twip.

4.1 Structure

Pequod stores data in several layers of tree visible to clients as a single ordered key-value store. The first tree layer separates logical *tables*, such as `p|` and `t|`, into separate subtrees. Each table stores the relevant key-value pairs and bookkeeping structures (an interval tree of updaters and a tree of join status ranges). By separating concerns for different ranges, this design sped up Pequod significantly relative to a one-level store.

Tables can themselves be subdivided. Many applications have natural key boundaries across which scans are rare; for example, Twip scans mostly lie within a timeline range. If developers mark these boundaries, Pequod will use them to break the store into *subtables*. Thanks to a hash table that indexes the subtables, operations that lie entirely within a subtable can jump to that subtable in $O(1)$ time (rather than $O(\log N)$). However, the entire key-value store is still ordered, and operations that cross subtable boundaries will execute as expected. The use of subtables improves the runtime of our Twip benchmark by a factor of 1.55x, but increases memory consumption by a factor of 1.17x, a consequence of additional bookkeeping.

4.2 Output hints

In many of our applications, each update to a join status 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 tree. This optimization avoids tree lookups in our Twip benchmark, and improves its performance by a factor of 1.11x.

4.3 Value sharing

The `copy` operator often requires Pequod to install multiple copies of a value into multiple output ranges. For example, Twip inserts a copy of each tweet into each of the interested followers' timelines. To reduce memory overhead, Pequod allows multiple output ranges to share the source's value. This optimization fits in naturally with server computation and might not work as naturally if sharing was entirely directed by application clients. This optimization reduces memory consumption by a factor of 1.14x on our Twip benchmark. Value sharing is only useful for `copy` joins, but it introduces no overhead on other joins.

5 Evaluation

This section evaluates Pequod's performance. Pequod performs well compared with related systems, its materialization strategy works well on our workloads, and unconventional features of its data model (interleaved cache joins) can benefit real applications. Furthermore, distributed Pequod can scale across multiple servers to handle large workloads.

5.1 Experimental setup

We evaluate Pequod using two hardware configurations, a multiprocessor and a cluster of Amazon EC2 virtual machines. The multiprocessor is an Amazon EC2 `cr1.8xlarge` instance with 32 logical processors and 244GB of RAM running Ubuntu Linux 13.04. The Amazon EC2 cluster, used to evaluate scalability, consists

of `cc2.8xlarge` and `cr1.8xlarge` VM instances connected by a 10Gbps network. Each VM has 32 cores, 60-244GB of RAM, and runs Amazon Linux 2013.09.2.

Application clients communicate with Pequod servers 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 servers.

In most of our experiments, Pequod is configured to run Twip. The underlying data is derived from a Twitter social graph obtained in 2009 [21]. The full graph, which contains 40 million users and 1.4 billion relationships, is used in the scalability experiment (§5.5). All other Twip experiments use a sampled subgraph containing 1.8M users and 72M relationships.

Our clients model the actions of individual Twip users. Each modeled user (1) "logs in," obtaining a list of many recent tweets; (2) repeatedly checks for new tweets, subscribes to other users, and posts tweets of their own; and (3) logs out (though they may log in again later). The incremental timeline updates in step (2) return many fewer tweets than the initial scans at login time. These events occur in the rough ratio 5% initial timeline scans, 9% new subscriptions, 85% incremental timeline updates, and 1% posts, which we derived using information on the real Twitter [20]. 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. In one common workload, 70% of users are active (the remainder never check their timelines) and each active user checks their timeline 50 times. This results in approximately 62M timeline checks, 6.2M new relationships, and 620K new posts over the course of the experiment.

We do not evaluate database interaction or eviction. Pequod is deployed as a look-aside 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.

We ran most experiments several times and observed little to no performance variability. Confidence intervals would not be visible on our graphs.

5.2 System comparison

Pequod aims to improve the programmability of application-level caches by offering developers more interesting semantics. These semantics do not compromise performance: Pequod performs no worse than comparable caches.

We evaluate two Twip implementations. The first, "Pequod," is the Twip application described above; timelines are fetched via the timeline join. In the second, "client Pequod," application clients are responsible for maintaining timelines. There are no cache joins. After making a post, the posting client sends a timeline update for every subscribed user. Client Pequod lets us evaluate the performance impact of server-managed computation in isolation. We also evaluate the Redis (version 2.8.5) and memcached (version 1.4.16) key-value caches and a traditional database, PostgreSQL (version 9.1). Each system runs the same workload to completion as fast as possible. Redis and memcached don't support server-side computation, so as in client Pequod, their clients actively manage user timelines; Redis stores timelines as sorted sets of tweets, memcached as a string to which tweets are appended. PostgreSQL, in contrast, does support server-side 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.[2]

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

Figure 7: Time to process a Twip experiment to completion using Pequod and related systems. Smaller numbers are better.

Figure 7 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 optimized 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-

[2] We disable Redis disk checkpoints and avoid triggering eviction in memcached by configuring the amount of available memory. 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.

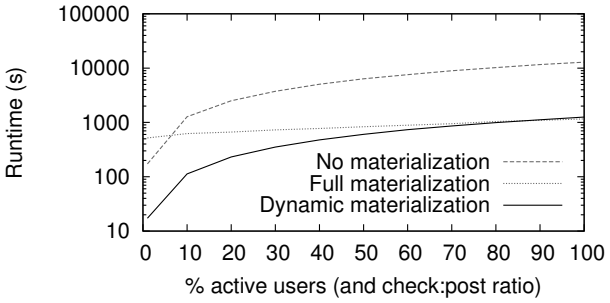

Figure 8: Pequod's dynamically materialized views generally outperform other strategies on the Twip benchmark.

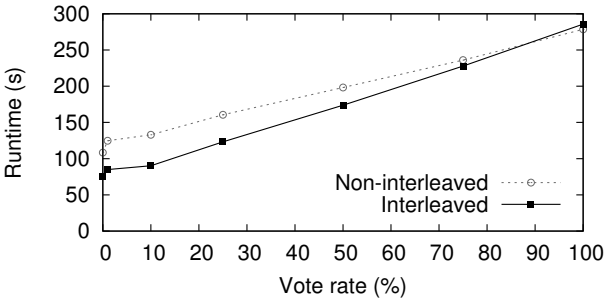

Figure 9: Newp interleaved cache joins perform better than fetching article data in separate RPCs, except when writes are very common.

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 an application-level cache. Pequod outperforms PostgreSQL by nearly an order of magnitude (9.55x).[3]

To summarize, Pequod performs no worse than widely available key-value caches; for this workload, it even offers a small performance improvement. The additional semantics provided by Pequod simplify application development without compromising performance.

5.3 Materialization strategy

Pequod implements cache joins using a 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. 1 million posts are distributed among all 1.8M users as described above (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 million timeline checks spread uniformly across the $1.8\text{M} \times p/100$ active users, resulting in a check:post ratio between 1:1 and 100:1. We use five clients and one server, run the workload to completion as fast as possible, and measure the elapsed time.

Figure 8 shows the results. As expected, the no-materialization strategy performs relatively well with few active users, but as timeline scans 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 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 when users can be inactive.

[3]Widely-available databases with true materialized view support were also evaluated; they performed similarly to PostgreSQL.

5.4 Cache join choice

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.

We evaluate two versions of the Hacker News-like Newp application that use different joins. The first uses separate ranges for aggregate data (karma and vote counts); constructing an article requires many RPCs in two round trips. The second uses the interleaved cache join from §2.3 to colocate this data. Reading an article requires a single scan, but more server computation and storage overhead is incurred (upon each vote aggregate values are copied into each `page|` range). The shared workload has three types of operation: reading an article, commenting, and voting. The Pequod data store is pre-populated with 100K articles, 50K users, 1M comments, and 2M votes. We simulate 20M user sessions; each user reads a random article; with a varying chance votes on the article; and independently with a 1% chance comments on the article. The experiment is run using a single server and multiple clients. We expect the interleaved approach to perform well when article reads far outnumber votes and comments.

The results, shown in Figure 9, indicate that interleaved cache joins are superior for most vote rates tested.

The non-interleaved implementation issues many gets per article (e.g., for karma), each of which incurs overheads including an $O(\log N)$ lookup. The interleaved join improves overall performance until the cost of precomputation outweighs the cost of processing many gets (90% vote rate). Storing votes and karma in a hash table might shift the crossover to the left, but the interleaved join would preserve its main advantage, namely code simplicity.

5.5 Scalability

Our evaluation of distributed Pequod focuses on a problem inherent in cache joins, namely the CPU overhead of cache join execution. Pequod can do more work per request than would a simpler cache, putting pressure on server CPUs. We show that distributed Pequod scales well enough that adding servers reduces this pressure.

Our experimental setup involves a large backing store, which holds the full 2009 Twitter social graph, and a variable number of Pequod compute servers, which execute the timeline join in response to client timeline checks. Both backing store and compute servers are Pequod processes, but the backing store absorbs all writes while the compute servers absorb all reads. Each experiment runs the same Twip workload: 28M active users issue 1.4B timeline checks, make 140M new subscriptions, and generate 14M new posts. We run the workload as fast as possible to ensure that the bottleneck is within Pequod. Measurements indicate that for the server setups measured, in each case, the bottleneck is Pequod compute server CPU. All of a user's compute requests are directed to the same compute server, minimizing unnecessary data duplication. To warm the cache servers, each active user is logged into the system prior to the experiment, ensuring that a join status range exists, base data are present, updaters are installed, and subscriptions are established between the compute and base servers. We use up to 30 virtual machines on the Amazon EC2 testbed. 6 VMs run 32 Pequod servers for the backing store and up to 48 client processes, while the remaining 24 VMs run between 12 and 48 Pequod compute servers.

Figure 10 shows the result. Throughput increases by 3x (from 1.42M to 4.27M qps) as the number of compute servers increases from 12 to 48. Perfect scalability would increase throughput by 4x; unfortunately, some Pequod overheads (such as base data required per compute server) do not drop linearly with the number of compute servers. Total memory consumption increased from 290GB to 297GB at the base servers, a consequence of storing duplicate subscription information. The compute servers stored more total data thanks to base data duplication; total memory used by all compute servers increased from 1.2TB to 1.5TB. Likewise, a larger fraction of the consumed network bandwidth is dedicated to inter-server subscription maintenance, increasing from roughly 10% to 16% between 12 and 48 compute servers (the rest is client communication). Though these overheads are potential bottlenecks to system scalability, Pequod still performs and scales well overall.

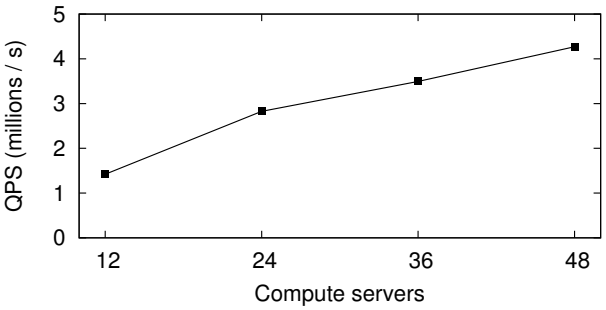

Figure 10: Adding computational capacity results in a speedup for a fixed Twip workload.

6 Related work

Application caches Pequod's key-value design was inspired by existing application-level caches [2, 3].

Pequod is an ordered store. This helped make cache joins simple and useful, but unordered hash table stores, such as Redis and memcached, offer faster $O(1)$ operations. Several of our implementation tricks and optimizations—some of them enabled by the cache join abstraction—reduce the cost of tree walking. To speed up Pequod further, we could replace its binary trees with more cache-efficient structures [24] or, better, investigate cache join variants for unordered stores. (This would probably require structured values à la Redis.)

Previous work has tracked dependencies among cached data in application-level caches [15, 31]. The application developer specifies these dependencies explicitly, either by item or by item class; the systems respond to updates by invalidating dependent objects. In TxCache [25], dependency tracking is automatic, rather than explicit. TxCache provides transactional consistency, which Pequod does not, and can invalidate cached objects that are computed by arbitrary pure functions, rather than SPJ queries. Compared to these systems, Pequod's cache joins define dependencies in a particularly natural way, and Pequod can update dependent objects, which we found faster than invalidating them.

DBProxy [7, 8] is a distributed database cache that can store partial query results in edge cache nodes and service later queries from those caches. The caches are incrementally maintained by a master backend database; new results are produced by re-executing queries (with an exception for some aggregates, which are also stored as exact-match results). Pequod, in contrast, uses eager view maintenance to avoid costly computation on the

read path. DBProxy transparently populates its cache by inspecting queries. Pequod is not transparent: developers decide what data is cached and describe how to maintain that data.

Materialized views Pequod borrows joins and materialized views [11, 16] from relational databases. Many production databases offer materialized views for data warehousing, replication, and storing results of expensive computations. Most views are meant to be transactionally consistent with the underlying data, and are kept up to date either by synchronous updates when base data changes (eager update) or by refreshing the view when it is read (lazy update). We borrow from work on efficiently maintaining views with incremental updates and batch processing [10, 13, 18, 27–29, 32].

Materialized views in Pequod are eventually consistent with respect to base data and are maintained through asynchronous, incremental updates. Views in Pequod can also be partially and dynamically materialized [28, 30], a relatively advanced feature.

Several research systems have applied one or more of these techniques. Agrawal et al. [5] added materialized views to PNUTS [17], a distributed key-value store. 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.

Dynamic materialized views (DMV) [30] can partially materialize a view based on data access patterns. In DMV, the selection of rows to materialize can be specified manually or handled at runtime by a feedback loop with policies for admission and eviction. Pequod's dynamic materialized views borrow from DMV, but implement a simple selection policy based on access time.

DBToaster [6] presents a method for deriving incremental update triggers from relational view queries, but only works on aggregate queries and does not partially materialize views.

Luo's partial materialized views (PMVs) cache portions of frequently-executed queries with the goal of allowing early access to partial query results [23]. PMVs are restricted—for instance, they do not support insertions on base data—but a similar feature might be useful for some Pequod applications.

Applications The real Twitter service actively updates the timelines of logged-in users as tweets arrive [20]; this was one inspiration for Pequod's hybrid pull/push architecture. The load on Twitter's service is high: Twitter has more than 150M active users that generate 300K timeline reads per second on average. On a typical day Twitter handles 4-6K new tweets per second (340M per day), resulting in 300K deliveries per second (2.6B per day) to user timelines [20]. Twitter scales its timeline service by partitioning its users amongst an expandable set of cache servers. Our Twip application uses the same strategy. Twip does not support other Twitter features, such as search; we have not investigated whether these features would benefit from Pequod.

Silberstein et al. [26] describe the importance of balancing "pull" and "push" strategies in social networking services, an insight we borrow for celebrity join. Given a Twitter-like application, their system determines at runtime which tweets should be materialized into followers' timelines and which should not. A more complex join operator could conceivably support their algorithm in Pequod.

7 Conclusion

Pequod is a distributed key-value cache that uses a new abstraction, the cache join, to automatically rearrange and transform cached data in ways useful for applications. By understanding how cached data is computed, Pequod is able to keep cached data fresh and provide performance benefits while presenting a simple API to users. As future work, we hope to improve Pequod further by optimizing its data structure design and exploring options for configuration changes and recovery from server failure.

Acknowledgements

We thank the SOSP and NSDI reviewers and our shepherd, Ali Ghodsi, for many helpful comments. This work was partially supported by a Microsoft Research New Faculty Fellowship, by the National Science Foundation (awards 0834415 and 0915164), and by Quanta Computer.

References

[1] Hacker News FAQ. http://ycombinator.com/newsfaq.html.

[2] memcached. http://memcached.org.

[3] Redis. http://redis.io.

[4] Retwis. http://redis.io/topics/twitter-clone.

[5] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *Proc. SIGMOD 2009, ACM SIGMOD Int'l Conf. on Management of Data*, pages 179–192. ACM, June 2009.

[6] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endowment*, 5(10): 968–979, June 2012.

[7] K. Amiri, S. Park, and R. 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, Nov. 2002.

[8] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: a dynamic data cache for web applications. In *Proc. ICDE 2003, 19rd Int'l Conf. on Data Engineering*, pages 821–831. IEEE Computer Society, Mar. 2003.

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

[10] J. A. Blakeley, P.-Å. Larson, and F. W. 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.

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

[12] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proc. 2013 USENIX Annual Technical Conf.*, pages 49–60. USENIX, June 2013.

[13] M. W. Cain. Creating and using materialized query tables (MQT) in IBM DB2 for i5/OS (version 2.0). Technical report, IBM, Sept. 2006. http://public.dhe.ibm.com/partnerworld/pub/pdf/courses/438a.pdf.

[14] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Engineering*, 1(1): 146–166, Mar. 1989.

[15] 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, Mar. 1999.

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

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endowment*, 1(2):1277–1288, August 2008.

[18] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In A. Gupta and I. S. Mumick, editors, *Materialized Views*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[19] A. Gupta, I. S. 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.

[20] R. Krikorian. Real-time delivery architecture at Twitter. Talk at QCon New York. http://www.infoq.com/presentations/Real-Time-Delivery-Twitter, Oct. 2012.

[21] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. WWW 2010, 19th Int'l World Wide Web Conf.*, pages 591–600. ACM, Apr. 2010.

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

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

[24] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys'12, 7th ACM European Conf. on Computer Systems*, pages 183–196. ACM, 2012.

[25] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. 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, Oct. 2010.

[26] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *Proc. SIGMOD 2010, ACM SIGMOD Int'l Conf. on Management of Data*, pages 831–842. ACM, June 2010.

[27] F. W. Tompa and J. A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, October 1988. URL `http://dx.doi.org/10.1016/0306-4379(88)90005-1`.

[28] J. Zhou, P.-Å. Larson, and J. Goldstein. Partially materialized views. Technical Report MSR-TR-2005-77, Microsoft Research.

[29] J. Zhou, P.-Å. Larson, and H. 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, Sept. 2007.

[30] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *Proc. ICDE'07, 23rd Int'l Conf. on Data Engineering*, pages 526–535. IEEE Computer Society, Apr. 2007.

[31] H. Zhu and T. Yang. Class-based cache management for dynamic Web content. In *Proc. INFOCOM'01, 20th Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pages 1215–1224, 2001.

[32] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. SIGMOD'95, 1995 ACM SIGMOD Int'l Conf. on Management of Data*, pages 316–327. ACM, May 1995.