A Trigger-Based Middleware Cache for ORMs

Priya Gupta[†], Nickolai Zeldovich, and Samuel Madden
MIT CSAIL, [†]Google

Abstract. Caching is an important technique in scaling storage for high-traffic web applications. Usually, building caching mechanisms involves significant effort from the application developer to maintain and invalidate data in the cache. In this work we present CacheGenie, a caching middleware which makes it easy for web application developers to use caching mechanisms in their applications. CacheGenie provides high-level caching abstractions for common query patterns in web applications based on Object-Relational Mapping (ORM) frameworks. Using these abstractions, the developer does not have to worry about managing the cache (e.g., insertion and deletion) or maintaining consistency (e.g., invalidation or updates) when writing application code.

We design and implement CacheGenie in the popular Django web application framework, with PostgreSQL as the database backend and `memcached` as the caching layer. To automatically invalidate or update cached data, we use triggers inside the database. CacheGenie requires no modifications to PostgreSQL or `memcached`. To evaluate our prototype, we port several Pinax web applications to use our caching abstractions. Our results show that it takes little effort for application developers to use CacheGenie, and that CacheGenie improves throughput by 2–2.5× for read-mostly workloads in Pinax.

1 Introduction

Developers of popular web applications often struggle with scaling their application to handle many users, even if the application has access to many server machines. For stateless servers (such as HTTP front-ends or application servers), it is easy to spread the overall load across many machines. However, it is more difficult to add database servers, since partitioning the data over multiple machines is non-trivial, and executing queries across multiple machines can be expensive in itself. Database replication solves this problem well in read extensive workloads, but does not work well in write-heavy loads such as social networks, which we concentrate on in this work. Web application developers typically solve this problem by adding caching middleware in front of the database to cache the results of time-consuming queries, such as queries that span multiple servers, complicated aggregate queries, or small but frequent queries. Thus, caching forms an important part of storage systems of many web applications today; for example, many websites use `memcached` [8] as a distributed in-memory caching system.

However, popular caching solutions such as `memcached` offer only a key-value interface, and leave the application developer responsible for explicitly managing the cache. Most importantly, the developer must manually maintain cache consistency by invalidating cached data when the database changes. This has several disadvantages. First, developers have to write a significant amount of code to manage the application's caching layer. Second, this code is typically spread all over the application, making the

application difficult to extend and maintain. It is also a common source of programming errors; for example, a recent outage of Facebook was caused by an error in application code that tried to keep `memcached` and MySQL consistent [13]. Finally, the developers of each application independently build these caching mechanisms and cannot re-use other developers' work, due to the lack of common high-level caching abstractions.

This work aims to address these issues with *CacheGenie*, a system that provides higher level caching abstractions for automatic cache management in web applications. These abstractions provide a declarative way of caching, where the developers only specify what they want to cache and the desired consistency requirements, and the underlying system takes care of maintaining the cache.

The first goal of CacheGenie is to relieve programmers of the burden of cache management. CacheGenie does three things to achieve this. First, it generates database queries based on object specifications from the developer; these queries are used to get the result from the underlying database, which is then cached. Second, whenever possible, CacheGenie transparently uses the cached object instead of executing the query on the database. Finally, whenever underlying data a cached object depends on is changed, it transparently invalidates or updates the cached object. This is done by executing database triggers when the underlying data changes.

The second goal of CacheGenie is to avoid making any modifications to the database or cache, and to use existing primitives present in modern databases (i.e., triggers) in order to ensure cache consistency. This makes it easy for developers to start using CacheGenie and to reuse existing components.

The third and final goal of CacheGenie is to minimize the performance overhead of maintaining a consistent cache. Any database-cache synchronization approach will incur a combination of two overheads: updating/invalidating the cache in response to database writes and sending read requests to the database if lookups in the cache fail. In CacheGenie, we minimize the first overhead by incrementally updating the cache from within the database, since the database has all the information regarding which keys to update and how. The second overhead is minimized through the use of an *update* approach where, once data is fetched into the cache, it is kept fresh by propagating incremental updates from the database (unless evicted due to lack of space). In contrast, in an *invalidation* approach, data is removed from the cache due to writes to database and must be completely re-fetched from the database. In our experiments we show that in CacheGenie, the overhead of maintaining a consistent cache is less than 30% (versus an approach in which no effort is made to synchronize the cache and database).

Our implementation of CacheGenie is done in the context of Object-Relational Mapping (ORM) systems. In recent years, ORM systems like Hibernate [12] (for Java), Django [7] (for Python), Rails [11] (for Ruby) and others have become popular for web application development. ORM systems allow applications to access the database through (persisted) programming language objects, and provide access to common database operations, like key lookups, joins, and aggregates through methods on those objects. ORM systems help programmers avoid both the "impedance mismatch" of translating database results into language objects, and the need for directly writing SQL. Because most accesses to the database in an ORM are through a small number of interfaces, we focused on supporting these interfaces. While CacheGenie does not cache

non-ORM operations, all updates to the database (through ORM and otherwise) are propagated to the cache through database triggers, ensuring cache consistency. Our current implementation of CacheGenie propagates updates to the cache non-transactionally (i.e., readers can see dirty data, but not stale data), and serializes all writes through the database to avoid write-write conflicts.[1] CacheGenie provides caching abstractions for common query patterns generated by the ORM.

We have implemented a prototype of CacheGenie for Django, PostgreSQL, and `memcached`. To evaluate CacheGenie, we ported several applications from Pinax [26] (a suite of social networking applications coded for Django) to use CacheGenie. Our changes required modifying only 20 lines of code; CacheGenie automatically generated 1720 lines of trigger code to manage the cache. Our experiments show that using CacheGenie's caching abstractions leads to a 2–2.5× throughput improvement compared to a system with no cache.

This paper makes four contributions. First, we describe CacheGenie, a novel and practical system for automatic cache management in ORMs, which works with an unmodified database and `memcached`. Second, we introduce new *caching abstractions* that help programmers declare the data they wish to cache. Third, we use a trigger-based approach to keep the cache synchronized with the database. Finally, we evaluate CacheGenie on a real-world web application and show the benefits of incremental updates and invalidations using our approach.

The rest of this paper is organized as follows: §2 gives a background of current caching strategies and discusses some important related work in this area. §3 explains the concept of caching abstractions, how we support them, and discusses the consistency guarantees offered by CacheGenie. §4 describes our implementation using Django, Postgres and `memcached`. §5 discusses our experience with porting Pinax applications to use CacheGenie, and describes our experimental results. Finally, §6 concludes.

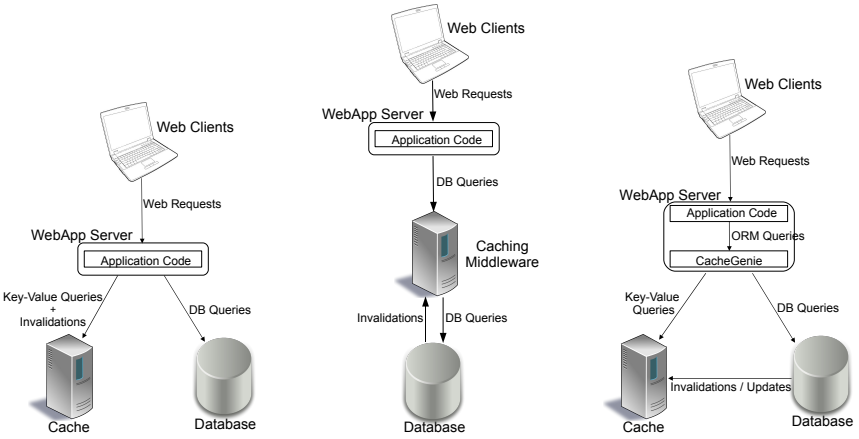
2 Background and Related Work

Web applications employ several caching strategies to improve their performance and reduce the load on the underlying data store. These strategies can be divided into two main categories: *application caching* and *database caching*.

The first category refers to application-level caching of entire HTML pages, page fragments or computed results. This scenario is illustrated by Figure 1a. In this scenario, the web application is responsible for cache management, and typically uses a key-value store, such as `memcached`, as the cache. Cache management includes (i) choosing the granularity of cache objects, (ii) translating between database queries and cache objects, so that they can be stored in a key-value store, and (iii) maintaining cache consistency.

With application-level caching, the cache and the underlying database are not aware of each other, and cache management is the application developer's burden. The advantage of application-level caching is that it allows for caching at a granularity best suited to the application. The disadvantage is that application developers have to manually implement cache management themselves via three possible options. The first option is to **expire** cached data after a certain interval of time. Although this mechanism is easy to use, it is insufficient for highly dynamic websites, and coming up with the *right*

[1] We include a description of how to extend CacheGenie with full serializability in §3.3.



(a) Application caching (b) Database caching (c) CacheGenie

Figure 1: Different approaches to caching in web applications.

expiration time is often difficult. The second option is **manual invalidation**, where the programmer has to keep track of all possible writes to the underlying database and determine which updates could affect what data in the cache. This can be cumbersome and error-prone. The third option is a **write-through** cache. In this case, every time the programmer writes code to update data in the database, she must also write code to update the cache. Since the data in the cache is not invalidated but updated in place, this can increase the number of cache hits. However, sometimes the application might not have enough information to determine which entries from the cache should be updated; this might lead to additional queries to the database, making the updates slower.

The second category, database caching, is illustrated in Figure 1b. In this model, a middleware layer caches partial or full data from the database near the application servers to reduce the load on the database server. In some cases, the cached data can be partial rows returned from the database against which further queries are executed [1, 3, 18, 20]. In this case, the middleware layer is responsible for deciding what to cache, how to satisfy the application requests based on what is in the cache, and maintaining cache consistency with the underlying data. Though this model frees the developer from managing the cache, it can result in sub-optimal caching behavior since cached objects are typically database rows and not application-level objects.

A simple version of database caching is to cache results of exact queries, and return the same results for identical future queries, as in GlobeCBC [25]. To maintain cache consistency, template-based invalidation schemes are typically used (templates are used because the problem of determining whether two queries touch the same data is hard [14]). Update queries are executed at the database server, and when a cache server stores the results of a query, it subscribes to receive invalidations based on conflicting query templates. There are two limitations with this model. First, the programmer must specify *a priori* all pairs of conflicting query and update templates; this can be time-consuming and error-prone. Second, if one update can potentially affect another query,

all cached results belonging to the corresponding query template are invalidated. This can lead to poor cache hit ratios, and thereby increase server load.

None of the above approaches fully solve the problem of caching in web applications. CacheGenie combines the best parts of these approaches into a system that is most beneficial for the programmer. CacheGenie provides high-level caching abstractions that programmers can use without making substantial changes to the application, database system, or caching layer. CacheGenie caches query results and automatically stores and updates those results, as opposed to providing a key-value store that the programmer must manually manage. The caching abstractions determine the granularity of caching, and automate translation between the data in the cached objects and the data stored in the underlying database. Unlike a template-based system, CacheGenie only invalidates cached data that is affected by writes to the database (see §3.2). This leads to fewer invalidations and higher cache hit ratios. The high-level architecture of CacheGenie is illustrated in Figure 1c. CacheGenie operates as a layer underneath the application, modifying the queries issued by the ORM system to the database, redirecting them to the cache when possible.

There are several other systems that provide automatic cache management. Labrinidis et al. [17] present a survey of the state-of-the-art in caching and materialization in web application databases. Cache invalidation as a strategy to maintain strong cache consistency has been explored in numerous prior works [19, 21]. Challenger [5] proposed using a dependency graph between cached objects and underlying data to regenerate or invalidate relevant HTML pages or fragments in the cache. However, the query workload that they consider was mostly reads, with few writes; a higher write fraction in an invalidation based system will lead to poor cache-hit ratio. Degenaro et al. [6] explored a similar approach in the context of generic query result caching. Ferdinand [9] provides a disk-based cache of query results, which uses a publish-subscribe model to achieve consistency in a scalable distributed manner. Using a pub-sub model in CacheGenie could improve scalability of propagating cache updates. A key difference between CacheGenie and these systems is that CacheGenie updates cached data in-place.

Several systems leverage snapshot isolation for caching. TxCache [24] provides a transactional cache, and ensures that any data seen within a transaction, whether it comes from the cache or the database, reflects a slightly stale but consistent database snapshot. TxCache lets programmers designate specific functions as cacheable; it automatically caches their results, and invalidates the cached data as the underlying database changes. Unlike TxCache, CacheGenie performs in-place updates instead of invalidation, but can cache and update only pre-determined functions (caching abstractions) instead of arbitrary functions. CacheGenie also relaxes transactional guarantees to allow the application to access fresh data, as many web applications do not require or use strong transactional guarantees. SI-cache [22, 23] similarly extends snapshot isolation to caching in J2EE applications. The key difference in CacheGenie, in addition to the above, is that CacheGenie maintains a single logical cache across many cache servers. In SI-cache, each application server maintains its own cache, and if many servers cache the same data, the total effective cache capacity is greatly reduced.

TimesTen [27] allows for caching partial tables from the database in their in-memory store, providing a SQL-like syntax to the developer to specify which partial tables to

cache. It allows for updates to flow from backend database to the cache using database triggers. However, the updates flow to the cache only periodically (the period of refresh specified by the application) and hence the application might see stale data. This is unlike CacheGenie, which ensures synchronous propagation of updates. Further, TimesTen caches raw table fragments, and hence requires extra computation at query time. AutoWebCache [4] uses aspect-oriented techniques to implement a caching middleware for dynamic content for J2EE server-side applications. Unlike CacheGenie, AutoWebCache caches entire web pages, and uses template-based invalidation for cache consistency.

There has been a lot of work exploring materialized views in databases and algorithms to incrementally update them. Materialized views are also useful in pre-computing and thus providing fast access to complex query results. The problem of incremental view maintenance is similar to the problem of maintaining up-to-date cached query results. However, unlike materialized views in the database, CacheGenie is geared towards maintaining views in a more distributed scenario. Moreover, CacheGenie scales better because it transfers the load from the database to a distributed cache. CacheGenie employs techniques similar to view maintenance [10] and materialization of entire web pages [16], except that in CacheGenie, "materialization" happens in a third caching layer, and CacheGenie caches more granular data than whole web pages.

A recent system called TAO by Facebook works in a way similar to ours, by letting programmers think in terms of high-level abstractions rather than SQL, and automatically managing the cache for them. An important difference however is that they perform write-through caching, whereas we propagate the updates through the database. CacheMoney [15] is a library for Ruby on Rails [11], which enables write-through and read-through caching to `memcached`; unlike CacheGenie, it does not support joins.

Table 1 summarizes the relation of CacheGenie to several representative systems.

System	Cache granularity	Source code modifications	Stale data	Cache coherence
`memcached`	Arbitrary	Every read	Yes	None
`memcached`	Arbitrary	Every read + write	No	Manual invalidation
TxCache	Functions	None	Yes (SI)	Invalidation / timeout
TimesTen	Partial DB Tables	None	Yes	Incremental update-in-place
GlobeCBC	SQL queries	None	No	Template-based inv.
AutoWebCache	Entire webpage	None	No	Template-based inv.
CacheGenie	Caching abstractions	None	No	Incremental update-in-place

Table 1. Comparison of CacheGenie with representative related systems.

3 Design

In this section, we describe the programmer's interface to CacheGenie's caching abstractions and then describe how they are implemented internally using triggers.

3.1 Caching Abstractions

Rather than trying to provide a generic query caching interface, CacheGenie's goal is to cache common query patterns generated by ORMs like Django. The workloads may also have infrequent queries that are not handled by the patterns CacheGenie supports

(e.g., because Django allows applications to directly write SQL queries). However, to improve performance, it typically suffices to improve these commonly occurring queries, and CacheGenie's approach does not require that all queries be mediated by the caching layer. Working within these frameworks also ensures that the programmer does not have to change the programming model she is using.

ORM-based web applications generate database queries using objects (such as an object representing an entire table, called models in Django) and functions (such as filtering objects based on the certain clauses). The programmer issues queries by calling functions on objects; the ORM framework issues SQL queries to the database in response to these function calls. CacheGenie provides caching abstractions called *cache classes* for common query patterns generated by the ORM. Each query pattern is represented by one cache class. To cache data pertaining to different entities but following the same pattern, the programmer defines multiple instances of the corresponding cache class, and each instance is called a *cached object*. Once a cached object is defined, the programmer can simply use existing object code, and CacheGenie takes care of fetching the right data from the cache.

We explain the concept of cache classes with an example from the social networking domain. Imagine that a developer wants to fetch the profile data of users in the application. To get this data from the database in Django, the developer would write the following code (assuming models for `User`, which contains administrative information about users, and `Profile`, which contains detailed information entered by the user, have already been created. The `Profile` is related to model `User` by the `user_id` field):

```
profile = Profile.objects.get(user_id=42)
```

To cache this data in Django, the developer has to manually put/get the profile data into the cache wherever needed in the application, and manually invalidate the cached data wherever profile of a user is modified, using statements such as the following:

```
from django.core.cache import cache
cache.set('profile:42', user_profile, 30) # Cache user 42's profile, 30s expiry
cache.get('profile:42')                   # Get data from cache
cache.delete('profile:42')                # Invalidate cached copy
```

In CacheGenie, this query falls under the FeatureQuery cache class (described below). To cache this data, the developer only needs to add a cached object definition once:

```
cached_user_profile = cacheable(cache_class_type = 'FeatureQuery',
    main_model = 'Profile',                 # Main Model to cache
    where_fields = ['user_id'],             # Indexing column
    update_strategy = 'update-in-place',    # optional arguments
    use_transparently = True)               # optional arguments
```

Once this definition is made, CacheGenie automatically and transparently manages profile data in cache. The application code to access the cached data remains exactly same as before (i.e., it is the same as getting the object from the database), and the developer does not have to write any additional code to manage the cache. As we do not provide strict transactional consistency in CacheGenie (see §3.3), the programmer can set `use_transparently` to false for objects which might need such guarantees. In that case, CacheGenie will not transparently fetch the cached data. The programmer can manually call `evaluate` on the returned cached object `cached_user_profile` with the `id` of the desired user to get the cached data manually:

```
profile = Profile.objects.get(user_id=42)           # get from cache or db
profile = cached_user_profile.evaluate(user_id=42)  # explicit cache lookup
```

CacheGenie supports the following cache classes:

1. **Feature Query** involves reading some or all features associated with an entity. In relational database terms, it means reading a (partial or full) row from a table satisfying some clause—typically one or more `WHERE` clauses. For example, in a social networking application which stores profile information in a single table, the query to get the profile information of a user, identified by a `user_id`, is a Feature Query. Since these queries make up a large percentage of many workloads, caching them is often beneficial.

2. **Link Query** involves traversing relationships between entities. In relational database terms, these queries involve traversing foreign key relationships between different tables. Since they involve joins, Link Queries are typically slow; caching frequently executed Link Queries is often beneficial. An example of a frequent join query in a social networking app is to look up information about the interest groups to which a user belongs. This query involves a join between the `groups_membership` table and the `groups` table. To create an instance of the `LinkQuery` cache class, the application must specify the chain of relationships to be followed.

3. **Count Query** caches the count of rows matching some predicate. A typical web application's page displays many types of counts, for example, a user's Facebook page displays counts of her friends, messages in the inbox, unread notifications and pending friend requests. Count queries are good candidates for caching, as they take up little memory in cache but can be slow to execute in the database.

4. **Top-K Query** caches the top K elements matching some predicate. In database terms, a top-K query involves sorting a table (or a join result) and returning the top K elements. Top-K queries are often expensive, and their results should be cached when possible. One important property of Top-K queries is that the cached results can be incrementally updated as updates happen to the database, and don't need to be recomputed from scratch, and CacheGenie exploits this property. An example of a Top-K query is fetching latest 20 status updates of a user's friends. Additional parameters for a Top-K query are the sort column, the order of sorting, and the value K.

In addition to class specific parameters, the programmer can also specify the cache consistency strategy, which can be (i) invalidate the cached object, or (ii) update it in-place (default). Since cache objects are defined based on existing queries in the application, it should be possible to derive these definitions automatically from the application, either by looking at SQL queries generated, or analyzing the ORM queries. The developer would still be able to choose the characteristics such as cache consistency requirements. We would like to explore this in future work.

Although CacheGenie implements cache classes for only a few query patterns, it is easy to extend the same concepts to other types of queries. Note, however, that cache classes correspond to queries common to multiple applications, and hence are reusable. Each developer does not have to write their cache classes, unless they want to cache a query pattern very specific to their application. To give an idea of what it takes to write a new cache class, we list here the functions it must perform:

1. **Query generation** uses the models and fields in the cached object to derive the underlying query template to get that object from the database. Note that we cache the raw results of queries and not Django model objects constructed from them.

2. **Trigger generation** involves determining which database tables and operations need triggers to keep the cached object consistent with the database. It also includes generation of the necessary code for the triggers, as described in §3.2.

3. **Query evaluation** involves fetching the appropriate data from the cache and transforming the returned value into the form required by the Django application, when the application requests it. If the key is not present in the cache, the cache class must query the database with the query generated during the generation step, add the result to cache, and return the appropriate transformed values to the application.

As an example, the definition of LinkQuery Class is as follows, with each function executing one task from the above.

```
class LinkQuery(CacheClass):
    def __init__(self, *args, **kwargs): # Initialize, Implement Step 1.
    def get_trigger_info(self):          # Implements Step 2.
    def evaluate(self, *args, **kwargs): # Implements Step 3.
    def make_key(self, *args, **kwargs): # Returns the corresponding key.
```

3.2 Database Triggers

Writes in CacheGenie are sent directly to the database, where it uses database triggers to automatically sync the cached data with these changes. In CacheGenie, for each cached object, there are three triggers—for insertion, deletion and update—generated on each of the tables underlying the cached object. These triggers are automatically generated from the cached object specifications. The programmer does not need to manually write the triggers, or specify *a priori* the cached objects that may be affected by each write query. When fired, the trigger code determines which cached entries, if any, can be affected by the modified data. It then modifies or invalidates these entries appropriately.

If the programmer chooses to invalidate cached objects, the trigger code invalidates *only* those entries of the cached object which are affected by this change. For example, imagine that the profile information of users with `user_id` 42 and 43 is currently in the cache. A query that updates the profile information of user 42 causes only the cached entry for user 42 to be invalidated, and the cached entry for user 43 remains unchanged. Note that this is different from template-based cache consistency mechanisms, which invalidate both the user profiles since they both match the same template.

Invalidation makes the trigger code simple, but invalidating frequently used items can lead to a poor cache-hit ratio. A better solution may be to update the cached data in response to the update. In this approach, the trigger code determines which entries in the cache could be affected by the data change in the table, figures out how to update the cached data incrementally, and finally updates the relevant cached objects. Continuing with the previous example, if an `UPDATE` query updates the profile information of user 42, the cached entry for user 42 is updated with the new profile information and is available to any future request from the application. The problem of figuring out how to update a cached object is similar to the problem of incrementally updating a materialized view. This problem has been previously studied, and is hard to solve in general. However, because CacheGenie supports a few fixed types of query patterns, it becomes less computationally intensive compared to solving it for a general view. View maintenance

techniques [10] can be applied to incrementally update cached objects other than the ones supported by CacheGenie.

The generated trigger operates in four steps. First, the trigger gets the modified rows from the database as input. Second, based on the modified rows (depending on the cached object for which this trigger is responsible), it figures out which keys in the cache can be affected by these rows. Third, if the cache consistency strategy is to update the cache, it queries the cache for the affected keys and calculates new values for them. Fourth, again depending on the cache strategy, it either deletes the affected keys from the cache or updates them in place with the new computed values.

To illustrate how triggers are generated in more detail, we provide a detailed example of how a trigger for a Top-K Query cache class is generated. To cache a `TopKQuery` object, CacheGenie caches an ordered list of results in `memcached` for the underlying query. The list contains K elements, as specified by the programmer when defining the cached object, plus a few more, to allow for incremental deletes. Consider the example of a Facebook-like social networking application where each user has a `wall` where any friend can post a note for this user. Let the `wall` table schema be:

```
wall (post_id int, user_id int, content text, sender_id int, date_posted date)
```

Suppose the developer wants to create a Top-K cached object for the latest 20 posts on a user's wall. The cached object definition that the developer writes for this would be:

```
latest_wall_posts = cacheable(cache_class_type = 'TopKQuery',
    main_model = 'Wall', where_fields = ['user_id'],
    sort_field = 'date_posted', sort_order = 'descending', k = 20)
```

For this cached object, CacheGenie automatically generates three triggers on the `wall` table (one each for `INSERT`, `DELETE` and `UPDATE`). When a new post gets inserted into the table, the corresponding trigger runs and gets the new post as input. From this inserted row, CacheGenie's trigger finds the `user_id` whose wall the post belongs to (say, 42), and then determines the key in the cache that will be affected, say `LatestWallPostsOfUser:42` (we use this key prefix for the sake of illustration; in practice a system-generated unique prefix is used). Next, assuming the update strategy, CacheGenie's trigger queries `memcached` for this key. If not present, the trigger quits. Otherwise, it finds the correct position of the new post in the cached list of posts according to `date_posted` and modifies the list accordingly. Finally, it puts the key back in the cache with the new modified value. The actual generated Python code for this trigger is:

```
cache       = memcache.Client(['host:port'])
table       = 'wall'
key_column  = 'user_id'
sort_column = 'date_posted'

new_row   = trigger_data['new']
cache_key = 'LatestWallPostsOfUser:' + new_row[key_column]
(cached_rows, cas_token) = cache.gets(cache_key)

if cached_rows is not None: # if present, update
  new_sort_val = new_row[sort_column]
  insert_pos = 0
  for row in cached_rows:
    if new_sort_val > row[sort_column]:
      break
    insert_pos += 1
  if insert_pos < len(cached_rows):  # update
    cached_rows.remove(len(cached_rows) - 1)
```

```
cached_rows.insert(insert_pos, new_row)
cache.cas(cache_key, cached_rows, cas_token)
# not shown is retry when CAS fails
```

The trigger for `DELETE` similarly determines whether the deleted row exists in the list, and if so, deletes it. For top-K queries, CacheGenie fetches a few additional rows beyond the top K to support deletes without immediate re-computation. When this reserve is exhausted, CacheGenie have to recompute the entire list for caching. `UPDATE` triggers simply update the corresponding post if it finds it in the cached list.

More details of how triggers for other cache classes are automatically generated are left out for the sake of brevity. Next, we discuss the consistency guarantees offered by CacheGenie and contrast it with those provided by existing caching systems.

3.3 Consistency Guarantees

We have already described the basic mechanisms provided in CacheGenie to enable cache consistency. In this section we discuss the consistency guarantees CacheGenie provides with these mechanisms. First, CacheGenie performs atomic cache invalidations/ updates for any single database write. Second, CacheGenie provides immediate visibility of a transaction's own updates. All cached keys affected by a write query are updated as a part of that statement, and hence the user sees the effects of her own writes immediately after the query is executed. This is a highly desirable property even for web applications since users expect to see their own updates immediately.

Currently, CacheGenie does not extend database transactions to the caching layer, because there were few conflicts in our workload (we have omitted the experiment which measures this for lack of space). The implication of this is that a transaction may read the results of an uncommitted update of another transaction from the cache (because writes are still done in the database, write-write conflicts are prevented). Another reason we chose not to add transactional support to the cache is that doing so would require changes to `memcached`, which we tried to avoid. We note that other database caches, like DBCache [3] and DBProxy [1] also provide relaxed transactional semantics; similarly, the application caching systems we know of also provide a weak consistency model.

However, for completeness, we describe a method for supporting full transactional consistency in CacheGenie. The key observation is that the database cannot determine whether a pair of concurrent transactions conflict because it does not see all the read operations. Thus, CacheGenie would track the readers and writers of keys in `memcached` and block transactions from completing their reads/writes to `memcached` according to the rules of two-phase locking.

In our design, when a transaction begins, the application and database decide on a transaction id, *tid*. Whenever a database trigger issues any updates/invalidations to `memcached` as a part of a transaction, it includes *tid*. For each key k, `memcached` would keep track of $readers_k$ (a list of the *tid*s of all uncommitted transactions that read k), and $writer_k$ (the id of the uncommitted writer of k, if any). Note that a database write to a row r can affect multiple keys in the cache, and a given key k may be a cached result corresponding to many rows in the database, so the keys associated with these readers and writers do not have an exact correspondence with rows in the database.

When using invalidations, we need to keep the readers and writers for invalidated keys, even if the keys themselves have been removed from the cache. Similarly, when a

transaction T does a lookup of a key k, we need to add T to $readers_k$, even if k has not yet been added to the cache.

Our modified `memcached` blocks reads and writes if they conflict. Thus, a transaction T reading key k will be blocked if $(writer_k \neq \text{None} \wedge writer_k \neq T)$, and a transaction T writing key k will be blocked if $(writer_k \neq \text{None} \wedge writer_k \neq T \wedge readers_k - \{T\} \neq \{\})$.

When Django is ready to commit a read/write transaction T, it sends a commit message to the database. If T commits at the database, CacheGenie removes T from the readers and writers of all keys in `memcached`, and allows any transaction blocked on one of those keys to resume executing, by adding itself to the appropriate readers and writers (latches must be used to prevent concurrent modifications of a given readers or writers list). If T aborts, CacheGenie has to remove T from the readers list of all keys it read, and remove all keys it wrote from the cache (so that subsequent reads will go to the database). For read-only transactions, Django does not need to contact the database for either aborts or commits, because read-set tracking is done entirely in the cache. Django can issue single-statement (autocommit) transactions for read queries that are not satisfied by the cache; if Django later needs to perform a write as a part of one of these transactions, these initial reads will not affect correctness of the serialization protocol.

Note that deadlocks can occur in the above protocol; because keys can be distributed across several `memcached` servers, we propose using timeout-based deadlock detection (as in most distributed databases). When a deadlock occurs, Django will need to abort one of the transactions on the database server, using the above abort scheme. Note that we cannot abort a transaction that is in the process of committing at the database server. One concern with the performance of this approach is the overhead of tracking each read/write operation by `memcached`. However, we think it would be insignificant as compared to (a) network latency, and (b) actual operation as size of *tid* is much smaller than the actual key/value size. Database overhead is also minimal as it only needs to maintain a list of `memcached` servers it contacted for each transaction.

Although we haven't implemented this full-consistency approach, we do provide a simple mechanism for the programmer to opt for a strict consistency on a case-by-case basis, if she so desires. If the programmer is aware that some cached object needs strict consistency in certain scenarios, she can opt out of automatic fetching from cache for that particular cached object. Then the programmer manually uses the cached object when she requires weak consistency and does not use it in case she requires strict consistency. The query in the latter case goes directly to the database and fetches the fresh results.

4 Implementation

We implemented a prototype of CacheGenie by extending the popular Django web application framework for Python. One advantage of using Django is that there are several open-source web applications implemented on top of Django, which we can use to test CacheGenie's performance and usability. In particular, we use Pinax, which is a suite of reusable Django applications geared towards online social networking.

Applications in Django interact with the database via models. A Django model is a description of the data in the database, represented as Python code. A programmer defines her data schema in the form of models and Django creates corresponding tables in the database. Further, Django automatically provides a high-level Python API to retrieve objects from the database using functions on model objects, such as `filter` and

`limit`. Django also provides ways to define many-to-one, many-to-many and one-to-one relationships between database tables, using model attributes referring to other models.

We implemented the cache classes described in §3.1 as special classes in Django. A programmer can cache frequently accessed queries that fit our abstractions by defining instances (called cached objects) of the appropriate cache class. The cache class performs three functions—(i) it uses the models and fields in the cached object to derive the underlying query template to get that object from the database, (ii) it generates and installs the associated triggers on the database, and (iii) it intercepts regular Django queries to return cached results transparently, if present, and otherwise populates the cache with the desired data from the database. Our prototype supports invalidation, update-in-place, and expiry intervals for cache consistency.

We use unmodified `memcached` for caching. The default least-recently used (LRU) eviction policy works well for a web application workload. However, keys get bumped to the front of LRU when touched by the triggers, even though they are not really being "used" by the application. One can modify `memcached` to support a modified LRU policy where certain actions can be specified not to affect LRU.

For the database, we use unmodified Postgres, which is supported by Django natively. We exploit Postgres triggers (written in Python) to manage the cache on writes to the database. Note that even though we picked Django, `memcached`, and Postgres for our prototype implementation, it should be easy to apply our design to other web application frameworks, caches, and databases.

5 Evaluation

The first aspect of our evaluation is ease of use. To evaluate CacheGenie's ease of use, we ported Pinax, a reusable suite of Django applications geared towards online social networking, to use CacheGenie. Our results show that CacheGenie's abstractions require few changes to existing applications (changing about 20 lines of code for Pinax).

The second aspect of our evaluation is performance. For this, we compare three systems: (i) `NoCache`—a system with no caching, where all requests are being served from the database, (ii) `Invalidate`—CacheGenie prototype in which cache consistency is maintained by invalidating cached data when necessary, and (iii) `Update`—CacheGenie prototype in which consistency is maintained by updating cached data in-place. We evaluate performance using the Pinax applications ported to CacheGenie. Although it would be instructive to compare CacheGenie with other automated cache management approaches, our system is inherently tied to the ORM model, whereas most of the previous systems are not, making a direct comparison difficult.

We ran several experiments to evaluate CacheGenie's performance. The results of these experiments show that using CacheGenie improves request throughput by a factor of 2–2.5 over `NoCache` for a mixed read-write workload. We also see that the `Update` scenario has up to 25% throughput improvement over `Invalidate`. Increasing the percentage of reads in the workload improves caching benefits of CacheGenie: for a read-only workload, CacheGenie improves throughput by a factor of 8.

We also performed microbenchmarks to understand the performance characteristics of CacheGenie. These microbenchmarks show that using `memcached` instead of a database can improve throughput by a factor of 10 to 150 for simple queries. Further, a database trigger can induce overhead ranging from 3% to 400%.

The rest of this section describes these experiments in more detail.

5.1 Experimental Setup

Pinax is an open-source platform for rapidly developing websites and is built on top of Django. We modified Pinax to use CacheGenie, and focused on three applications from the social networking component of Pinax—profiles, friends and bookmarks. We deal with only four particular actions by the user: (i) LookupBM: lookup a list of her own bookmarks, (ii) LookupFBM: lookup a list of bookmarks created by her friends, (iii) CreateBM: add a new bookmark, and (iv) AcceptFR: accept a friend invitation from another user.

We created cached objects for the frequent and/or expensive queries involved in loading the pages corresponding to these actions. For instance, we added cached objects for getting a list of a user's bookmarks, a count of saved instances of a unique bookmark, a list of bookmarks of a user's friends and so on. Once these cached objects have been defined, the application code automatically obtains the corresponding cached data.

For performance evaluation, as illustrated in Figure 1c, our experimental setup comprises of three main components: (i) the application layer (web clients, web server, and application), (ii) the cache layer, and (iii) the database layer. Since the aim of our evaluation is to measure the performance of the cache and database (the data backend), we have combined the web clients, web server and application server into a single entity called the 'application layer', which simulates a realistic social-network style workload and generates the corresponding queries to the data backends.

Our experimental workload consists of users logging into the site, performing the four actions (**Page Load**) described above according to some distribution and logging out. The default ratio of these actions in our workload is $\langle$LookupBM : LookupFBM : CreateBM : AcceptFR$\rangle$ = $\langle 50 : 30 : 10 : 10 \rangle$. We can also look at it as the ratio of read pages (LookupBM + LookupFBM) to write pages (CreateBM + AcceptFR). The default ratio is then 80% reads and 20% writes. We believe that this ratio is a good approximation of the workload of a social networking type application where users read content most of the time and only sometimes create content (Benevenuto et al [2] found that browsing activities comprised of about 92% of all requests). Note that 20% writes does not mean 20% of queries are write queries, but only reflects the percentage of write *pages*. In practice, as in real web application workloads, a write page also has several read queries in addition to write queries.

The set of actions from a user's login until her logout is referred to as one **Session**. We refer to each action/page as a **Page Load**. Further, any request for data issued by the client to either the database or the cache is referred to as a **Query**. For most of the experiments each client runs through 100 sessions. The distribution of users across sessions is according to a *zipf* distribution with the zipf parameter set to 2.0 (as in Benevenuto et al [2]). Each session in turn comprises of 10 page loads, in the ratio specified above. Each page load consists of a variable number of queries, depending on the application's logic, which averages about 80.

For the final measurements, we only replay the queries generated during actual workload runs. As such, we need only one client machine to saturate our database. The client machine is an Intel Core i7 950 with 12 GB of RAM running Ubuntu 9.10. We use Python 2.6, Django 1.2 and Pinax development version 0.9a1. The client machine,

database machine, and the `memcached` machine communicate via Gigabit ethernet. The database machine is an Intel Xeon CPU 3.06 GHz with 2 GB of RAM, running Debian Squeeze with Postgres 8.3. The database is initialized with 1 million users and their profiles, 1000 unique bookmarks with a random number of bookmark instances (between 1 and 20) per user. Further, each user has 1–50 friends to begin with, and 1–100 pending friendship invitations. The total database size is about 10 GB. The tables are indexed and clustered appropriately. The caching layer consists of `memcached` 1.4.5 running on a Intel Pentium 2.80 GHz with 1 GB of RAM. The size of the cache depended on the experiment, but for most experiments it was 512 MB. Note that this is only an upper limit on the amount of memory it *can* use, if needed.

5.2 Programmer Effort

As described in §3.1, the programmer needs to add a cached object definition for each query pattern instance she wants CacheGenie to cache for her. To port the Pinax applications described earlier in this section, we added 14 cached objects. Adding each cached object is just a call to the function `cacheable` with the correct parameters.

Once the cached object has been created, caching is automatically enabled for corresponding queries. In the absence of CacheGenie, the programmer has to manually write code to get and put data in the cache wherever a query is being made. In our sample application, we counted 22 explicit locations in the application code where such modifications are necessary. However, there are many more instances where the query is being made implicitly, such as from the Django framework. In such cases, the programmer will have to modify the internals of Django in order to cache these queries. We believe developers may find this undesirable.

To manage cache invalidations and updates, CacheGenie automatically generates triggers corresponding to the defined cached objects. For the 14 cached objects, CacheGenie generates 48 triggers, comprising of about 1720 lines of Python code. Without CacheGenie, the programmer will have to manually invalidate any cached data that might be affected by any write query to the database, and without an automatic cache management scheme, the programmer will have to write about the same number of lines of code as our generated triggers, i.e. 1720 lines of code. Large applications may have many more cached objects and hence many more lines of code for cache management.

5.3 Microbenchmarks

We used microbenchmarks to quantify performance characteristics of the cache, database and database triggers. We ran a variety of experiments on a small database (fits in RAM) and a similarly sized `memcached` instance to measure the time cost of database vs. cache lookup queries, and the time cost of running a trigger.

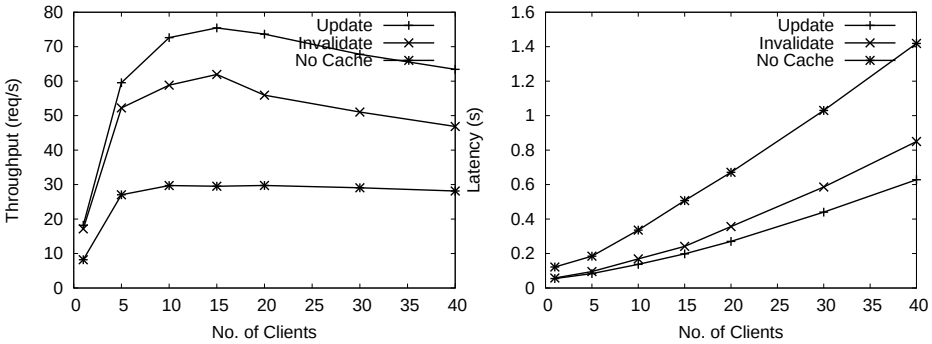
For database vs. cache lookups, we found that simple B+Tree lookup on the database takes 10–25$\times$ longer on the database, suggesting there is significant benefit in caching.

We also looked at the time to run a database trigger as a part of an INSERT operation, relative to the cost to perform a plain INSERT. We found that a plain INSERT takes about 6.3 ms, while an INSERT with a no-op trigger takes about 6.5 ms. Opening a remote `memcached` connection, however, doubles the INSERT latency to about 11.9 ms. Each `memcached` operation done from within the trigger takes an additional 0.2 ms, which is the same amount of time taken by a normal client to perform a `memcached` operation.

Hence, the main overhead in triggers comes from opening remote connections; if we could reuse the connection for subsequent triggers, it would make the write operations much faster. Exploring this is a part of future work.

5.4 Social Networking Workload

In this section, we describe our performance experiments with the Pinax applications, present these results, and discuss our conclusions from these experiments. Each experiment has the following parameters: number of clients, number of sessions for each client, workload ratio, zipf parameter, and cache size. The default values for these parameters are 15 clients, 100 sessions, 20% write pages, 2.0, and 512 MB respectively. In each experiment, we measure throughput and latency, and compute averages for the time intervals during which all the clients were simultaneously running. We also warm up the system by running 40 parallel clients for 100 sessions before the start of each experiment.



(a) Page load throughput. (b) Page load latency.

Figure 2: Experiment 1—Performance against varying clients.

Experiment 1: Throughput and Latency Measurement In this experiment, we compare the three caching strategies—`NoCache`, `Invalidate` and `Update`—in terms of the maximum load they can support. The results in Figure 2 show the page load throughput and page load latency as the number of parallel clients increases.

From Figure 2a we can see that CacheGenie systems—`Invalidate` and `Update` provide a 2–2.5× throughput improvement over the `NoCache` system. This improvement is due to a significant number of queries being satisfied from the cache, thereby reducing the load on the database. Note that the advantage we get from our system is much less than the throughput benefit that `memcached` has over a database in our microbenchmarks. This is because we do not cache all types of queries; the uncached queries access the database and make it the bottleneck.

In all three systems, the database is the bottleneck and limits the overall throughput of the system. In the `NoCache` case, the CPU of the database machine is saturated, while in the two cached cases, disk I/O is the bottleneck. This is because queries hitting the database in `NoCache` are repeated, and hence a significant amount of time in the database is spent computing query results for in-memory data. For the cached cases, the bulk of the queries are either writes or not repeated (since the system caches most of the repeated queries). Hence, the database becomes bottlenecked by disk. This also explains why the throughput in cached cases drops after a certain point.

Note that the throughput is greater in the `Update` case than in the `Invalidate` case. On one hand, updating leads to slower writes, because triggers have to do more computation. On the other hand, updates lead to faster reads, because there are more cache hits. Figure 2a illustrates that the overhead of recomputing from the database after invalidation is more than the overhead of updating the relevant cached entries.

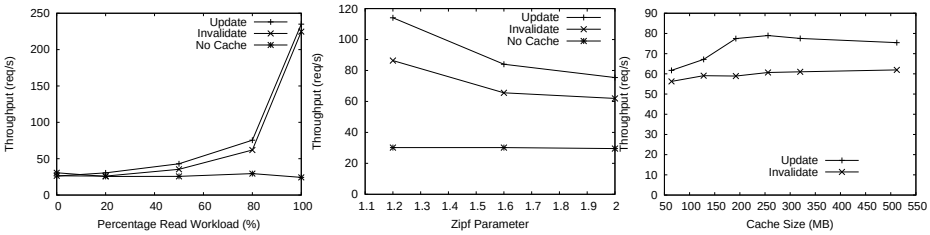
From Figure 2b we see that `Update` has the least latency of all the three scenarios, followed by `Invalidate` and `NoCache`. Also, the latency in all three cases rises more steeply as we increase the number of clients beyond 15, corroborating the fact that throughput drops slightly after this point. Table 2 shows the average latency for various types of page loads for the three systems in this experiment. The increased latency of CreateBM and AcceptFR is due to the overhead of updating the cache during writes to the database. In effect, the update operations become slower in order to ensure that subsequent reads are faster.

Page Type	`Update`	`Inval.`	`NoCache`
Login	0.29 s	0.34 s	0.11 s
Logout	0.10 s	0.11 s	0.05 s
LookupBM	0.05 s	0.05 s	0.22 s
LookupFBM	0.06 s	0.16 s	1.25 s
CreateBM	0.55 s	0.53 s	0.09 s
AcceptFR	1.03 s	1.24 s	1.01 s

Table 2. Average latency by page type in Experiment 1 (with 15 clients).

For all of the following experiments, in order to achieve the maximum throughput for all systems, we run 15 parallel clients, unless otherwise specified.

Experiment 2: Effect of Varying Workload In this experiment, we vary the ratio of read pages to write pages in the workload, and measure how it affects the performance of the three caching strategies. The default ratio, as mentioned before, is 80% read pages and 20% write pages. The results of these experiments are shown in Figure 3a.



(a) Varying workload. (b) Varying user distribution. (c) Varying cache size.

Figure 3: Throughput results for Experiments 2, 3, and 4.

From the figure, we see that for a workload with 0% reads, caching does not provide any benefit. In fact, it makes the performance slightly worse. This is because database writes are slower in the cached system due to the overhead of triggers. As the percentage of reads in the workload increases, however, the performance of cached cases improves. In the extreme case of 100% reads, the cached case throughput is about 8× the throughput of `NoCache`. Also note that the workload variation does not significantly affect `NoCache` since it is already CPU bound because of reads, which hit the database buffer pool. However, workload variation affects the cached cases, since they are disk-bound, and the disk is accessed less as the number of writes goes down.

The gap in throughput between `Update` and `Invalidate` increases as the number of reads increases from 0% because as the fraction of reads increases, the advantage of

better cache hit ratio overcomes the disadvantage of slower triggers in `Update`. The gap reduces when we have 100% reads because nothing in the cache is being invalidated or updated, and so both cases are equivalent. From this experiment, we conclude that caching shows much more benefit in a read-heavy workload than a write heavy one.

Experiment 3: Effect of Varying User Distribution The formula for zipf distribution is $p(x) = \frac{x^{-a}}{\zeta(a)}$ where ζ is the Reimann zeta function and a is the zipfian parameter. In our experiments, $p(x)$ is the probability that a user has x number of sessions, i.e. logs in x number of times. $p(x)$ is high for low values of x and low for high values of x. In other words, most users log in infrequently, and a few users log in frequently. Also, a low value of the zipfian parameter a means the workload is more skewed whereas a high value means that users login with a more uniform distribution.

The value of zipf parameter affects both performance of the database and the cache. In the cache, if there are certain users who login frequently, then the data accessed by them remains fresh in the cache and the infrequent users' data gets evicted. This means that, over a period of time, frequent users will find most of their data in cache, and hence the number of cache hits goes up, improving the system's performance. It also means we need a cache big enough to hold only the frequent user's data, which is much smaller than the total number of users in the system. It matters for the database performance as well, but only within short intervals, since the buffer pool of database gets churned much faster than the cache. Thus, database performance improves when users log in repeatedly in a short time span.

In this experiment we vary the parameter a of the zipf distribution and see how it affects the performance of the three systems. Figure 3b shows the results from this experiment. From the graph, we can see the cached cases have a $1.5\times$ higher throughput with $a = 1.2$ as compared to $a = 2.0$. The `NoCache` case, however, fails to show any improvement with changing values of a. The performance benefit in the cached cases comes from the database, which is disk-bound. With a lower zipf value, the database is hit with more repeated queries and reduces disk activity, thereby improving the performance for those cases. However, the `NoCache` case is already CPU-bound, and since Postgres does not have a query result cache, it still has to compute the results for the repeated queries from cached pages in the buffer pool.

Experiment 4: Effect of Varying Cache Size In all our experiments so far, the cache was big enough (512 MB) and there were no evictions. This means the only misses in the cache would be for keys which have never been put in the cache in the first place, or which have been invalidated. In a realistic system we may not have a cache that is big enough to hold everything that can ever be cached. The purpose of this experiment is to analyze the performance effect of evictions due to a smaller cache (in the cached cases).

The results from this experiment are shown in Figure 3c. The graph shows that the throughput of `Update` plateaus at about 192 MB, whereas for `Invalidate` it does so at about 128 MB. This is because `Update` never invalidates the data in cache, and thus requires more space. We can see that even with only 64 MB of cache space, `Update` and `Invalidate` have at least twice the throughput than `NoCache`. In practice the cache size needed depends on frequency of users and the distribution of workload.

Another important result of our experiments is that using spare memory as a cache is much more efficient than using that memory in the database. To validate this, we ran

`memcached` on same machine as the database, so that for the cached cases, the database has less memory. The throughput of `Update` in this experiment was 64 requests/s (down from 75), and the throughput of `Invalidate` was 48 requests/s (down from 62). This performance is still better than `NoCache` whose throughput was 30 requests/s.

Experiment 5: Measuring Trigger Overhead In §5.3, we measured the overhead of triggers in a simple database with simple insert statements. That established the lower-bound that triggers impose on an unloaded database. In this experiment, we measure the impact of triggers on the throughput of CacheGenie for the social networking workload.

An *ideal* system will be one in which the cache is updated for "free", incurring no overhead to propagate writes from the database to the cache. In such a system, the cache always has fresh data, making reads fast, and writes are done at the maximum rate the database can support. The throughput of such a system will be the upper bound on what we could possibly achieve in CacheGenie. To estimate the performance of such a system, we re-ran query traces from our social networking workload with the default parameters and with the triggers removed from the tables in the database. In this way, we make the same queries to the database and `memcached` as in experiment 1, but without any cache consistency overhead. Since the triggers are off, the cache is not updated (or invalidated). Even though this means that the read queries to cache will return incorrect data, it gives us a correct estimate of the performance of the *ideal* system.

From this experiment, we measured the throughput for the ideal `Update` system trace to be 104 requests/s (up from 75) and for the ideal `Invalidate` system to be 80 requests/s (up from 62). In other words, adding triggers brings down the throughput by 22–28% for a fully loaded database. We believe that this overhead is reasonable. In the future, we plan to explore various trigger optimizations to minimize this overhead. One is to combine various triggers on a single table into one single trigger to avoid trigger launching overhead. Second, triggers written in C could be potentially faster than Python triggers. Third, we can reuse connections to `memcached` between various triggers.

6 Conclusion

We presented CacheGenie, a system that provides high-level caching abstractions for web applications. CacheGenie's semantic caching abstractions allow it to maintain cache consistency for an unmodified SQL database and cache system, by using auto-generated SQL triggers to update caches. Our prototype of CacheGenie for Django works with Postgres and `memcached`, and improves throughput of Pinax applications by a factor of 2 to 2.5. Modifying Pinax to take advantage of CacheGenie required changing only 20 lines of code, instead of requiring programmers having to manually maintain cache consistency at every database update.

Acknowledgments

This work was partially supported by Quanta.

Bibliography

[1] K. Amiri, S. Park, and R. Tewari. DBProxy: A dynamic data cache for Web applications. In *Proc. of the ICDE*, Bangalore, India, 2003.

[2] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proc. of the IMC*, Chicago, IL, 2009.

[3] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.
[4] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic Web content: Designing and analysing an aspect-oriented solution. In *Proc. of the Middleware*, Melbourne, Australia, 2006.
[5] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic Web data. In *Proc. of the INFOCOM*, New York, NY, 1999.
[6] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Proc. of the Middleware*, New York, NY, 2000.
[7] Django Software Foundation. Django. `http://www.djangoproject.com/`.
[8] B. Fitzpatrick et al. Memcached. `http://memcached.org`.
[9] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for Web applications. In *Proc. of the VLDB*, Auckland, New Zealand, August 2008.
[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18:3–18, 1995.
[11] D. H. Hansson et al. Ruby on Rails. `http://rubyonrails.org/`.
[12] Hibernate Developers. Hibernate. `http://hibernate.org/`.
[13] R. Johnson. Facebook outage. `http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919`.
[14] B. T. Jónsson. *Application-oriented buffering and caching techniques*. PhD thesis, University of Maryland, College Park, MD, 1999.
[15] N. Kallen. Cache Money. `http://github.com/nkallen/cache-money`.
[16] A. Labrinidis and N. Roussopoulos. WebView materialization. In *Proc. of the SIGMOD*, Dallas, TX, 2000.
[17] A. Labrinidis, Q. Luo, J. Xu, and W. Xue. Caching and materialization for Web databases. *Foundations and Trends in Databases*, 2(3):169–266, 2010.
[18] P.-Å. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent mid-tier database caching in SQL server. In *Proc. of the ICDE*, Boston, MA, 2004.
[19] C. Liu and P. Cao. Maintaining strong cache consistency in the World-Wide Web. In *Proc. of the ICDCS*, Baltimore, MD, 1997.
[20] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-Business. In *Proc. of the SIGMOD*, Madison, WI, 2002.
[21] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, 1988.
[22] F. Perez, M. Patiño-Martinez, R. Jimenez-Peris, and B. Kemme. Consistent and scalable cache replication for multi-tier J2EE applications. In *Proc. of the Middleware*, Newport Beach, CA, 2007.
[23] F. Perez-Sorrosal, R. Jimenez-Peris, M. Patiño-Martinez, and B. Kemme. Elastic SI-Cache: Consistent and scalable caching in multi-tier architectures. *VLDB Journal*, 2011.
[24] 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. of the OSDI*, Vancouver, BC, Canada, 2010.
[25] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic Web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, Netherlands, June 2006. `http://www.globule.org/publi/GCBRCDWA_ircs022.html`.
[26] J. Tauber et al. Pinax. `http://pinaxproject.com/`.
[27] The TimesTen Team. Mid-tier caching: The TimesTen approach. In *Proc. of the SIGMOD*, Madison, WI, 2002.