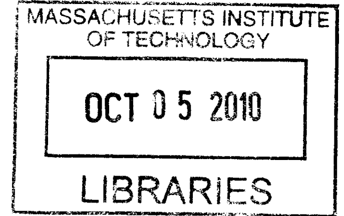
Providing Caching Abstractions for Web Applications

by

Priya Gupta

B.Tech., Computer Science and Engineering
Indian Institute of Technology, Delhi (2008)



ARCHIVES

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

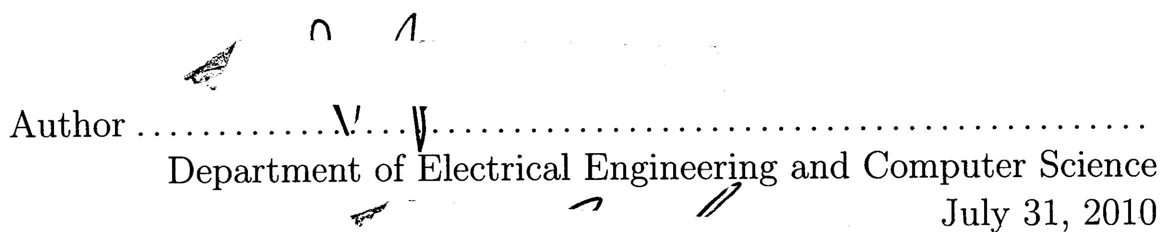
Master of Science in Electrical Engineering and Computer Science

at the

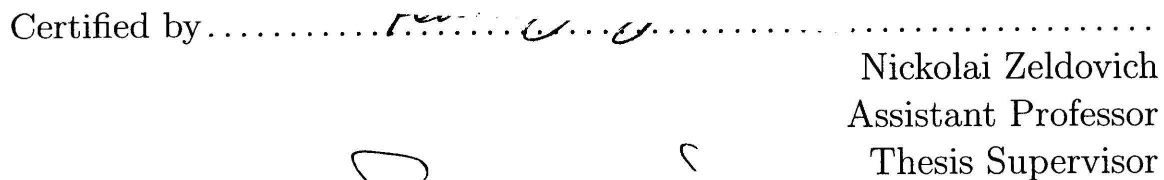
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

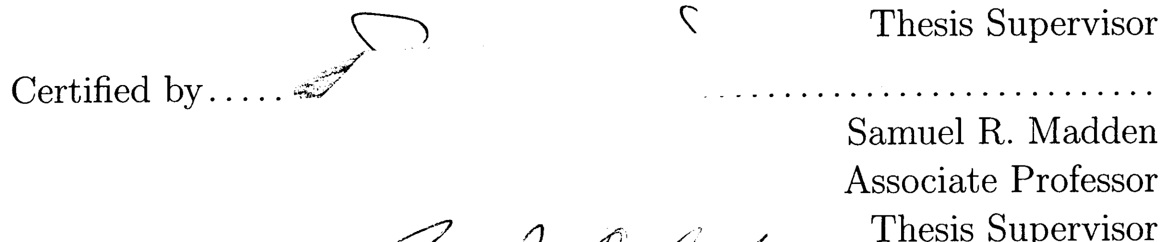
© Massachusetts Institute of Technology 2010. All rights reserved.



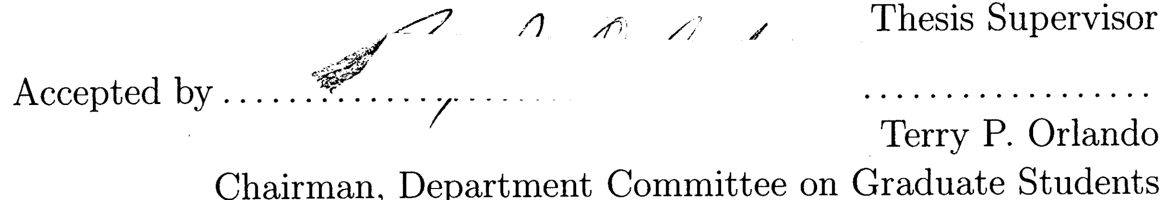
Author ....
Department of Electrical Engineering and Computer Science
July 31, 2010



Certified by ....
Nickolai Zeldovich
Assistant Professor
Thesis Supervisor



Certified by ....
Samuel R. Madden
Associate Professor
Thesis Supervisor



Accepted by ....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Providing Caching Abstractions for Web Applications

by

Priya Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on July 31, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Web-based applications are used by millions of users daily, and as a result a key challenge facing web application designers is scaling their applications to handle this load. A crucial component of this challenge is scaling the data storage layer, especially for the newer class of social networking applications that have huge amounts of shared data.

Caching is an important scaling technique and is a critical part of the storage layer for such 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 system which aims to make it easy for web application developers to build caching mechanisms in their applications. It achieves this by proposing high-level caching abstractions for frequently observed query patterns in web applications. These abstractions take the form of declarative query objects, and once the developer defines them, she does not have to worry about managing the cache (i.e., insertion and deletion) or maintaining consistency (e.g., invalidation or updates) when writing application code.

We designed and implemented CacheGenie in the popular Django web application framework, with PostgreSQL as the database backend and memcached as the caching layer. We use triggers inside the database to automatically invalidate or keep the cache synchronized, as desired by the developer. We have not made any modifications to PostgreSQL or memcached. To evaluate our prototype, we ported several Pinax web applications to use our caching abstractions and performed several experiments. Our results show that it takes little effort for application developers to use CacheGenie, and that caching provides a throughput improvement by a factor of 2-2.5 for read-mostly workloads.

Thesis Supervisor: Nickolai Zeldovich
Title: Assistant Professor

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor

Acknowledgments

My deepest gratitude goes to my advisor Nickolai Zeldovich for his invaluable guidance and support. Throughout my academic journey at MIT, Nickolai has been a source of constant encouragement, impeccable advice and brilliant ideas.

I am also deeply indebted to my co-advisor, Sam Madden, who has guided this work splendidly by providing us with a *database-y* point of view. I thank him for his amazing support, sound direction and wisdom throughout this thesis.

Special thanks to Neha Narula for joining our numerous discussions, helping out with implementation and for always being there to bounce off ideas and offer feedback; and Eugene Wu for working with me during the initial stages of the project. I would also like to thank other fellow grad students at the lab, especially Lenin, Taesoo, Yang, James, Adam and Alvin, who made these past two years on the *ninth* floor an enjoyable experience.

I want to thank all my friends at MIT for their wonderful company; especially Karthik, Neha, Varun and Jairaj, who have always looked out for me. My roommate Andrea has been both my friend and family, and a constant source of fun and advice at the same time. Special thanks goes to my lab mate and friend Ramesh who helped me immensely in the final stages of this thesis, whether it be brainstorming ideas, setting up experiments or proof-reading the thesis.

This thesis would not have been possible without the support and sacrifice of my family. I'd like to thank them for their love and patience—Papa and Mumma for always believing in me, Manish for silently being there, and Richa for her constant guidance and loving care. I dedicate this work to them.

Contents

1 **Introduction** **13**

2 **Design** **17**

2.1 Caching Abstractions 19

2.1.1 Common Query Patterns 20

2.1.2 Cache Class 25

2.2 Cache Consistency Mechanisms 27

2.2.1 Current Approaches 28

2.2.2 Database Triggers 30

2.2.3 Consistency Guarantees 36

3 **Implementation** **41**

3.1 Django 42

3.1.1 Overview of Django Models 42

3.1.2 Caching in Django 44

3.2 CacheGenie in Django 45

3.2.1 Cache Class Implementation 45

3.2.2 `cacheable` Function 47

3.2.3 An Example 47

3.3 Memcached 49

3.4 PostgreSQL 51

4 **Evaluation** **53**

4.1 Experimental Application: Pinax 53
4.1.1 Background Information on Pinax 54
4.1.2 Porting Pinax Applications 55
4.2 Programmer Effort 59
4.3 Performance 61
4.3.1 Experimental Setup 62
4.3.2 Microbenchmarks 66
4.3.3 Social Networking Workload 71
4.4 Conclusions 80

5 Related Work **83**

6 Future Work **89**

7 Conclusion **93**

A Pinax Database Schema **95**

Bibliography **98**

List of Figures

2-1 Various Caching Schemes for Web Applications 18

2-2 Sample Database Schema 21

2-3 Query Patterns 22

4-1 Pinax Social Networking Website 56

4-2 Microbenchmarks: Database vs Cache Performance 68

4-3 Microbenchmarks: Effect of Varying Data Size 69

4-4 Experiment 1—Performance with Varying Clients 73

4-5 Experiment 2—Performance with Varying Workload 76

4-6 Experiment 3—Performance with Varying User Distribution 78

4-7 Experiment 4—Performance with Varying Cache Size 79

A-1 Schema of database tables in Pinax 95

List of Tables

4.1 Various Database Configurations used in Microbenchmarks 66

4.2 Trigger Overhead on INSERT 71

4.3 Average Latency by Page Type 75

4.4 Average Latency by Query Type 75

Chapter 1

Introduction

With the tremendous increase in number of Internet users, web developers face a huge challenge in scaling web applications. In web applications today, databases are often the least scalable components. While it is easy to replicate stateless servers to scale up performance, database servers cannot be scaled linearly very easily. This is mainly because of two reasons: (i) queries spanning multiple database servers are slow and (ii) writes need to be applied consistently across all replicas. Hence, for good performance over large amounts of data it quickly becomes insufficient to rely on native database performance, and merely adding more database servers will not restore performance.

Web application developers typically solve this problem by adding a caching layer above the database to cache the results of time consuming queries such as ones which span multiple servers. Caching is also useful for queries which might not span multiple servers but are sufficiently complicated and/or frequent. Thus, caching forms an important part of storage systems of many web applications today; for example, many websites use `memcached` [10] as a distributed memory caching system.

However, there are various issues involved in using any caching system, the most important of them being cache maintenance. In cases where stale results are acceptable, the application developer sets an expiry time on the cached result based on various heuristics. For example, a user's friend's Facebook [7] profile page might be cached for, say, one minute, as it rarely changes. For other queries, stale data is

not acceptable and hence the developer has to invalidate the cached result whenever data used in computing that result changes. The data on a user's Facebook wall falls into this category. These decisions are based on a trade-off between the tolerance for stale data and amount of extra work required for invalidation and recomputation. An alternate approach that is less commonly used is a *write-through* cache, where the cached result is modified in place whenever data used in computing that result changes, and hence does not need expiry or invalidation. If a data item is used to compute multiple cached results, it is the application's responsibility to update all the cached results.

To summarize, developers today need to manually implement caching mechanisms, and have to manage cache consistency themselves. 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. 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 thesis aims to address these issues with CacheGenie, a system which provides higher level caching abstractions for automatic cache management in web applications, while making no modifications to the underlying database. 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. Specifically, CacheGenie does three things. First, it derives the queries corresponding to the developer's specifications, which are used to compute the cached object from the underlying database. Second, it transparently uses the cached object whenever required by the application, instead of executing the query on the database. Finally, whenever underlying data used in computation of the cached object is modified, it transparently invalidates or updates the cached object. This is done by executing database triggers when the underlying data changes. One of the important goals of CacheGenie is to not make any modifications to the database, and use existing primitives present in modern databases to ensure cache consistency.

Another important point to note here is that we do not provide transactional consistency over the cached data. This means that updates/deletes applied to the cache as part of one database transaction may be visible to other transactions. Once all the updates have propagated, the cache converges to a consistent state.

We observed that typical read queries in web applications can be classified into a few categories, and we aim to provide abstractions for several of these categories. For example, a common but slow read query in web applications is to lookup top-K items from the database according to some order. Looking up the latest 20 status message updates of a user's friends falls into this category. To cache this type of query, we introduce a caching abstraction called *top-K list*. More of these caching abstractions are described in the next chapter.

It is imperative that in addition to making the developer's life easier, using CacheGenie should not lead to any significant decrease in performance as compared to existing solutions. It is a secondary goal of this work to, in fact, improve performance in certain cases. For example, updating the cache in-place as opposed to invalidation or expiry means the result will always be incrementally updated and never recomputed from scratch. Although this results in extra work during each update operation, it leads to overall performance improvement in a read-heavy workload. This is because the cost-benefit of incrementally updating as opposed to calculating from scratch is more if the data item is being queried repeatedly.

We implemented a prototype of CacheGenie by modifying a popular web application framework called Django. Further, we ported a subset of applications from Pinax, which is a collection of usable Django applications, and show that the abstractions we provide are easy to use and do not require many changes to existing applications. For Pinax, we had to add only about 20 lines of code. Also, CacheGenie automatically generates about 1720 lines of trigger code, and we argue that without an automatic cache management scheme, the programmer will have to manually write about the same amount of code to manage the cache for these applications. Our prototype uses memcached as the cache and PostgreSQL for the database. Both these work in our system without any modifications.

We did experiments to evaluate the performance of our prototype with a bookmarks application from Pinax. From the experiments, we observed that using the caching abstractions leads to a 2-2.5 factor of improvement in throughput as compared to a system with no caching. Further, updating cached data in place is 25% faster than invalidating it. We also measured the throughput variation with varying workload and user distribution. As expected, caching wins by a higher margin in a read-heavy workload, and if there are repeated users accessing the application. We also performed microbenchmarks to measure the overhead of triggers, which can be between 3% and 400%, depending on the operations being performed by the trigger.

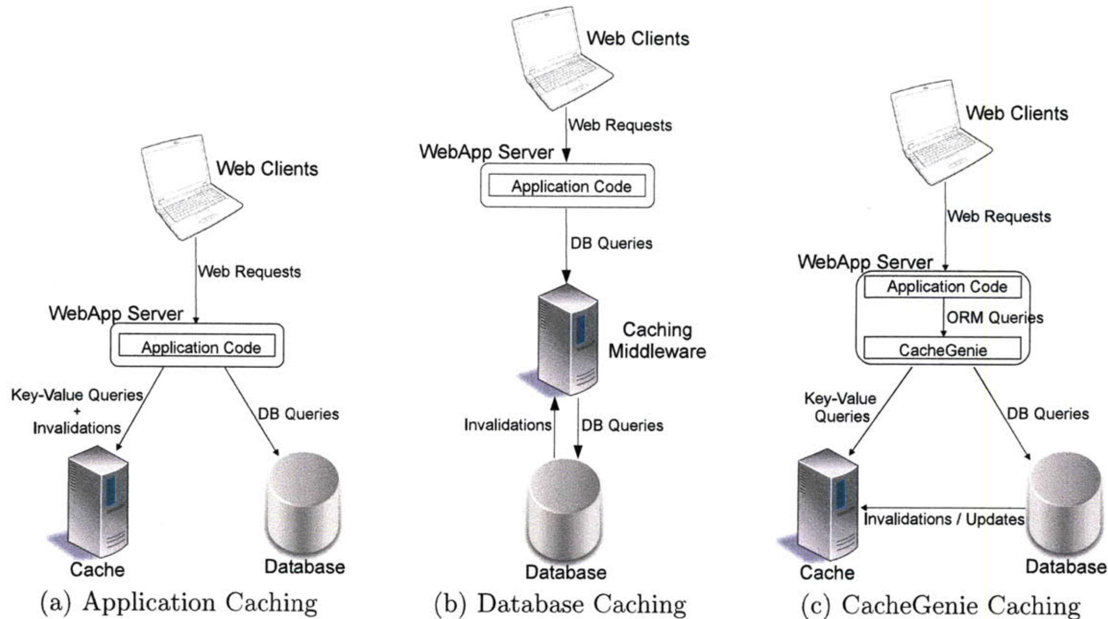
The rest of this thesis is organized as follows: Chapter 2 discusses the system's design. Chapter 3 describes our implementation in detail. Chapter 4 describes the evaluation strategy and results. Chapter 5 discusses some important related work in this area. Chapter 6 talks about limitations of the current system and possible extensions of this work. Finally, Chapter 7 concludes.

Chapter 2

Design

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 2-1a. 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. Caching entire HTML pages or fragments does not work very well for highly dynamic websites such as social networking websites, and so application programmers instead cache data at a more granular level. For example, cached objects maybe a list of a user's friends or a user's profile, rather than individual rows in the users or friends database tables. To guarantee consistency of the cached data with the underlying database, applications implement logic to either put expiration times on the cached content or invalidate them whenever underlying data changes. In this model, the cache and the underlying database are not aware of each other and cache management is the application developer's burden. In summary, 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



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

Figure 2-1: Various Caching Schemes for Web Applications

implement cache management themselves.

The second category, database caching, is illustrated in Figure 2-1b. 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. The cached data can be partial rows returned from the database against which further queries are executed ([22], [24]), or it could be results of exact queries stored in the cache that are returned immediately if future queries match old queries exactly ([31]). Database caching systems typically use query template-based invalidation schemes to maintain cache consistency. The middleware layer is responsible for deciding what queries to cache, how to satisfy the application requests based on what is in the cache, and maintaining cache consistency with the underlying data. In some systems the programmer can provide hints to the middleware as to what data to cache; in others, the system itself adapts to the workload ([24]). 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. For example, this results in re-computation of join queries leading to slower cache performance. Some database

caching systems avoid re-computation by caching results of entire queries; however, that requires the programmer to get involved in managing cache consistency, as described in Section 2.2.1, thereby increasing the burden on the programmer. So, to summarize, while database caching systems aim to reduce programmer burden related to cache management, they have the disadvantage that the mechanisms they employ can result in sub-optimal cache performance.

In essence, neither of the above approaches fully solves the problem of caching in web applications. In this work we aim to combine the best parts of these two approaches into a system that is most beneficial for the programmer. We do this by providing high-level caching abstractions that programmers can use while writing web applications. These abstractions result in the following benefits: They directly cache query results and automatically store and update those results, as opposed to providing a simple key-value store that the programmer must manually manage. Our techniques integrate closely with the database, using triggers to manage cache consistency via updates and invalidations. The caching abstractions decide the granularity of caching, and automatically translate between the data in the cached objects and the data stored in the underlying database. The choice of whether to invalidate or update can be made by the programmer while using these abstractions. The high-level architecture of CacheGenie is illustrated in Figure 2-1c. CacheGenie works as an integral part of the application server, handling the queries for the existing application code. The rest of this chapter describes our design in more detail.

2.1 Caching Abstractions

Rather than trying to provide a generic query caching interface, our goal is to cache common query patterns that emerge in web applications, particularly those written against object-relational mapping (ORM) frameworks like Django. These ORM based applications generate a constrained subset of SQL, and we plan to provide caching for these common queries. For example, in social networking applications, a common query pattern is to follow links in a social graph, such as when looking up a person's

friends. In an ORM, this appears as a sequence of *key – foreign-key* joins. In this work we aim to provide abstractions for such commonly observed types of queries. This section discusses the common types of queries seen in our target web application workloads, and the mechanisms we provide for the programmers to be easily able to cache those queries.

2.1.1 Common Query Patterns

Database query languages, such as SQL, are general purpose and provide a very rich query language, suitable for many applications; however, only a subset of the language features are used in a specific domain. For example, for warehouse type workloads, aggregate queries are most commonly used. We use this observation to define high-level caching abstractions for web applications based on ORMs.

For web applications, we observe that most of the read workload can be classified into the few categories described below. Each category represents a common query pattern observed in web application read workloads. The workloads may also have infrequent queries that lie outside these patterns, but to improve performance, it typically suffices to improve these commonly occurring queries. Hence, in this work we concentrate on these common patterns. Moreover, it is easy for a programmer to identify these patterns in an ORM layer. Working within these frameworks also ensures that the programmer does not have to change the current programming model she is using.

Following are the most common type of queries we observed in web applications. Figure 2-2 gives a partial database schema from a typical social networking application and Figure 2-3 gives sample SQL queries from the application representing these categories.

1. **Feature Query.** This is the simplest type of query which 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. This query does not involve traversing any one-

```
TABLE 1: users      (id              integer
                     username        character varying(30)
                     first_name      character varying(30)
                     last_name       character varying(30)
                     email           character varying(75)
                     password        character varying(128)
                     last_login      timestamp with time zone
                     date_joined     timestamp with time zone);

TABLE 2: profiles   (id              integer
                     user_id         integer
                     name            character varying(50)
                     about           text
                     location        character varying(40)
                     website         character varying(200));

TABLE 3: friendship (
                     id              integer
                     from_user_id    integer
                     to_user_id      integer);

TABLE 4: friendshipinvitation (
                     id              integer
                     from_user_id    integer
                     to_user_id      integer
                     message         text
                     sent            date
                     status          character varying(1));

TABLE 5: groups     (id              integer
                     name            character varying(50)
                     date_creation   timestamp with time zone
                     description     text);

TABLE 6: groups_membership (
                     id              integer
                     user_id         integer
                     group_id        integer
                     added           date);

TABLE 7: status     (id              integer
                     user_id         integer
                     statustext      character varying(140)
                     date_posted     timestamp with time zone);
```

Figure 2-2: Partial database schema from a social networking application

Query 1: Feature Query

```
SELECT    profiles.id, profiles.user_id,
          profiles.name, profiles.about,
          profiles.location, profiles.website
FROM      profiles
WHERE     profiles_profile.user_id = 42
```

Query 2: Link Query

```
SELECT    groups.id, groups.name,
          groups.date_creation, groups.description
FROM      groups, groups_membership
WHERE     groups_membership.user_id = 42 AND
          groups.id = groups_membership.group_id
```

Query 3: Count Query

```
SELECT    COUNT(*)
FROM      friendshipinvitation
WHERE     friendshipinvitation.to_user_id = 42  AND
          friendshipinvitation.status = 2
```

Query 4: Top-K Query

```
SELECT    status.id, status.user_id,
          status.statustext, status.date_posted
FROM      friendship, status
WHERE     friendship.from_user_id = 42 AND
          friendship.to_user_id = status.user_id
ORDER BY  status.date_posted DESC
LIMIT     20
```

Figure 2-3: Common query patterns in the social networking application

to-many or many-to-many relationships and hence is limited to one database table. For example, in a simple 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. Query 1 in Figure 2-3 represents this query in SQL. A Feature Query is fast if the database is indexed by the columns in the `WHERE` clauses, and can be satisfied in one or two disk seeks depending on whether the index page is in the database cache. Since these queries make up a large percentage of the workload, caching them is highly beneficial. Moreover, it is quite simple to determine when a cached Feature Query result should be invalidated.

2. **Link Query.** A query which involves traversing various relationships between entities is a Link Query. For example, a query to find all the people in a user's groups is a Link Query. In relational database terms, these queries involve traversing foreign-key relationships between different tables. These queries span more than one table and involve calculating join based on one or more clauses. The speed of such queries depends on several factors such as whether there are indexes on the join columns, whether the indexes are clustered, and the number of rows involved in the join. Link Queries are in general much slower than Feature Queries, and hence often observed Link Queries should be cached to avoid repeated join computation in the database. Moreover, if the join needs to be computed across multiple databases (either in the application or in a distributed database), it becomes all the more important to cache the Link Query result, as the join can be very slow. An example of a frequent join query for Facebook 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. Query 2 in Figure 2-3 illustrates this query.

3. **Aggregate Queries.** The following are the most common aggregate queries:

   - **Count Query.** 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, pending friend requests, etc., Count queries can be expensive if they involve joins across multiple tables. Count queries on a single table can sometimes be made faster by having a clustered index on the column in the `WHERE` clause. However, this can only work for a few queries since a table can be clustered on only one index. Hence, count queries, in general, are good candidates for caching, as they take up very little memory and the performance benefits can be significant. Query 3 in Figure 2-3 gives an example of a query to get the count of unaccepted friend invitations for a user ("status = 2" in `friendshipinvitation` table means the invitation hasn't been accepted yet.)

- **Top-K Query.** Another common and expensive query is to lookup the list of top-K elements satisfying certain criteria. The latest status updates of a user's friends on her Facebook homepage is an example of a Top-K Query. Another example is the list of top-selling items on Amazon. In applications such as these, Top-K Queries are very common as they appear in frequently accessed pages. In database terms, a top-K query involves sorting the table (which could be a join result), and returning K elements from the top. Hence, Top-K Queries are typically expensive, and their results should be cached whenever possible. Some applications, such as Facebook, build custom solutions to improve performance of Top-K Queries, as described in Chapter 5. 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 re-calculated from scratch—we exploit this property in CacheGenie. Query 4 in 2-3 represents an example Top-K query fetching latest 20 status updates of a user's friends.

We observe that the above types of queries are the most common in web applications. Specific applications may have additional types of query patterns; however, optimizing the query patterns listed above should deliver significant performance ben-

efits to most web applications.

Next we describe the abstraction mechanisms we provide for the above query patterns.

2.1.2 Cache Class

Web applications based on ORMs generate database queries using objects (ORM representation of data in the database) and functions (that filter query results based on the certain clauses). The programmer only has to represent the desired query using the correct object code and the ORM framework converts it into the right database query. In CacheGenie we provide caching abstractions for the query patterns generated by the ORM. The programmer still writes the object code, and we exploit this code to identify the query patterns.

We represent each query pattern observed in the workload by a *Cache Class* abstraction. For instance, there is a Cache Class called `FeatureQuery` representing the Feature Query pattern. To cache data pertaining to different entities, the programmer adds multiple instances of a single Cache Class, and each instance is called a *Cached Object*. For example, consider an application where the programmer is interested in caching profile information of various users, represented by Query 1 in Figure 2-3. In this example, since the programmer is interested in caching a user's profile information, she creates an instance of the `FeatureQuery` class, say `UserProfile`. Once this cached object has been created, the programmer can simply use their existing object code, and CacheGenie will take care of fetching the right data from the cache.

The programmer creates a cached object by specifying parameters to the Cache Class. Some of these parameters are required, and others are optional. For instance, to create `UserProfile`, the programmer is required to specify the table name (`profiles`) and column in the `WHERE` clause (`user_id`); she can optionally specify the whether the cached data should be updated or invalidated on update of the underlying `profiles` table. These parameters are discussed in detail later.

Following are the Cache Classes corresponding to the common query patterns discussed in the previous section.

1. **FeatureQuery Class.** To create an instance of the `FeatureQuery` class, the required parameters are the data entity of interest, and the columns by which to index the cached object. So for instance, as explained in the above example, `profiles` is the entity of interest and `user_id` is the column by which the cached object is indexed. There can be optional parameters such as a list of columns to be fetched for that entity in case the programmer is not interested in fetching all the columns.

2. **LinkQuery Class.** To create an instance of the `LinkQuery` class, the required parameter is the chain of relationships to be followed. For example, the cached object for a user's interest groups (Query 2 in Figure 2-3) is created by specifying the link `group_membership` between users and groups.

3. **CountQuery Class.** `CountQuery` has one of `FeatureQuery` or `LinkQuery` as its base class, and the parameters to create an instance of `CountQuery` class are the parameters of its base class. So for instance, to get the count of a user's pending friend invitations (Query 3 in Figure 2-3), the base class is `FeatureQuery` with parameters `friendshipinvitations` as entity of interest, and `status` and `to_user_id` as columns in `WHERE` clause.

4. **TopKQuery Class.** The `TopKQuery` class also has `FeatureQuery` or `LinkQuery` as its base class. It also takes as additional parameters the column on which to sort, the order of sorting, and K. For instance, to get latest 20 status messages of a user's friends (Query 3 in Figure 2-3), the parameters are: ordered on column `date_posted` of table `status`, descending order and 20, in addition to the parameters to specify the underlying `LinkQuery` to fetch all status messages of a user's friends.

In addition to class specific parameters, the programmer can also specify optional parameters that determine how the cached objects are maintained. The following optional parameters are available in CacheGenie:

1. Consistency mechanism. This parameter specifies the cache consistency update

strategy, i.e., what the cache should do when the underlying data for a cached object changes. The programmer can specify one of two options: (i) invalidate the cached object, (ii) update it in-place. In addition, the programmer can also specify an expiry interval beyond which the cached data should be invalidated. The programmer can choose one of these options according to the application requirements. These mechanisms are discussed in more detail in the next section. The default value for this parameter is update-in-place, and the default value of expiry interval is 0, which means it never expires.

2. Policies based on popularity. If the application has a mechanism to determine popularity of objects, the programmer has the option of adapting caching policies using that information. Note that these features have not yet been implemented in the current prototype. Examples of this are:

    - Prefetching. If the programmer knows that certain items are hot, she can specify that they be prefetched to improve performance. For instance, information about popular groups can be prefetched instead of being demand fetched on first access, thereby improving overall performance during peak-times. This is turned off by default.
    - Variable update strategies. The programmer can choose to vary update strategies within the same cached object according to popularity. So for instance, it makes more sense to use an update policy for highly-active users and an invalidate policy for non-active users. This is because the cost-benefit of incrementally updating as opposed to calculating from scratch is more if the data item is being queried repeatedly. The default case is update-in-place for all objects.

2.2 Cache Consistency Mechanisms

In this section we discuss cache consistency mechanisms in detail. First, we describe current cache consistency approaches and why they are insufficient. Next we describe

our proposal to solve the problem in a way that is most beneficial to the programmer.

2.2.1 Current Approaches

As discussed earlier, current caching approaches in web applications are broadly of two types: application caching and database caching. In the application caching model, developers have three main options to maintain cache consistency:

1. Expiration Interval. Most web caching systems use the technique of letting the data in cache expire after a certain interval of time. There are various heuristics to determine what this interval should be, depending on the application's requirements, and the data item under consideration. This is perhaps the easiest mechanism from the programmer's point of view. However, for a highly dynamic workload (such as that of social networks), this approach is insufficient, as unless the expiration intervals are short the cache would return stale data. A very short expiration interval does not work either since it results in a poor cache hit ratio.

2. Manual Invalidation. In manual invalidation, the programmer writes code to invalidate the cached data whenever there is a write query to the underlying store that could possibly conflict with this data. This means the programmer has to keep track of all possible writes to the underlying data store and determine which updates could affect what data. This can be cumbersome as well as error prone. One important goal of CacheGenie is to make this easier for the programmer and provide mechanisms which automatically take care of invalidation.

3. Write-through update. The third option, which is less frequently used is a write-through approach. Here again, the programmer manually writes code to update the data in cache whenever the application makes a conflicting write query. Since the data in the cache is not invalidated but updated in place, this leads to a better cache-hit ratio. However, sometimes the application might not

have enough information to determine which entries from the cache should be updated, and how. In the worst case, it might mean making more queries to the backend to get the required information. In that case, doing updates via the application can be slow resulting in increased latency of write queries.

Unlike in application caching, in the database caching model the programmer typically does not have to worry about cache consistency, as the caching middleware is responsible for it. Database caching systems use template-based mechanisms to ensure cache consistency. Write queries are executed at the central database server, and when an edge server caches a query, it subscribes to receive invalidation of conflicting query templates. There are two limitations with this model. First, the programmer is expected to specify a priori which query template conflicts with which update template, resulting in undue burden on the programmer. Second, in template based invalidation, if one update can potentially affect another query, all query results belonging to the query template are invalidated. This can cause a poor cache hit ratio, leading to increased origin server load, and therefore increased client latency.

CacheGenie solves the problems with both application and database caching, and maintains cache consistency automatically and transparently. The programmer only needs to specify the update strategy for the cached objects and the system transparently takes care of maintaining the cache consistent according to that strategy. The two options CacheGenie offers are automatic invalidation of cached data and incremental updates to the cached data. The option of letting the cached data expire after a certain interval is provided for the sake of completeness, since it can be useful for infrequently changing data that is tolerant to staleness. Our approach solves the two key problems with previous approaches: First, it relieves the programmer's burden of having to manually implement cache invalidation and update with code sprinkled across the application; the programmer also does not have to determine dependencies between write queries to the database and the cached data. Second, unlike a template-based system, CacheGenie only invalidates cached data that is affected by writes to the database. This leads to fewer invalidations and higher cache hit ratios.

We use database triggers to implement automatic synchronization of the cached data with the underlying database. In the next subsection, we describe how we use database triggers to implement the different update strategies available to the programmer.

2.2.2 Database Triggers

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. Triggers can be defined to execute on a `INSERT`, `UPDATE`, or `DELETE` operation, either once per modified row, or once per SQL statement. Triggers can also be set to fire `before` or `after` the operation. They can be written in many languages (as supported by the particular database) and are typically used for maintaining integrity of data in the database.

We use database triggers to keep the cached data consistent with the database. When a cached object is created, appropriate triggers responsible for that cached object are created in the database. There are multiple triggers associated with each cached object. These correspond to insertion, deletion and updation of rows 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 them, or specify a priori which cached objects might be affected by which write queries. Once all cached objects have been defined, the underlying tables potentially have multiple triggers corresponding to various cached objects.

When a `INSERT`, `UPDATE`, or `DELETE` occurs on the underlying table, all the triggers on that table associated with that event are executed `after` the statement, once for every affected row. The trigger code gets the old and new content of the concerned row as input, and determines which cached entries, if any, can be affected by this row. It then modifies or invalidates these entries appropriately.

As mentioned in the previous section, CacheGenie provides two main strategies for maintaining consistency of cached data:

1. **Invalidate.** If the programmer chooses to invalidate cached objects when the

underlying data changes, the trigger code invalidates *only* those entries of the cached object which are affected by this change. Building on the `UserProfile` example of Section 2.1.2, imagine that the profile information of users with `user_id` 42 and 43 is currently in the cache. If an `UPDATE` query updates the profile information of user 42, only the cached entry for user 42 is invalidated, and the cached entry for user 43 remains unchanged. Note that this is different from template based cache consistency mechanisms discussed earlier, which invalidate both the user profiles since they both match the same template. When the application requests profile information for user 42, CacheGenie fetches it from the database and re-inserts it into the cache.

2. **Incremental update.** Simple invalidation makes the trigger code short and fast to execute; however, invalidating frequently used items can lead to a poor cache-hit ratio. In other words, invalidation works well in read-mostly workloads. But for workloads with significant fraction of writes, this leads to poor performance. A better solution in that case is to update the cached data in place. In this approach, the trigger code first determines which entries in the cache could be affected by the data change in the table. Next it figures out how to update the cached data incrementally, without recomputing it from scratch. And finally, it updates the relevant cached objects. Continuing with the previous example, if the programmer chooses incremental updates and 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. This approach significantly reduces the number of cache misses in workloads with a higher write-read ratio.

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 significantly hard to solve for a general view. But because we limit ourselves to a few types of query patterns, it becomes simpler as well as computationally less intensive compared to solving it for a general

query.

An important point to note is that in CacheGenie inserting objects into the cache does not clash with updates. This is because only reading of a data object results in it being inserted into the cache; updates due to triggers never insert an object into the cache. Triggers always modify (or delete) existing cached objects.

In the remainder of this section, we describe how triggers for different Cache Classes are automatically generated. Here we only describe triggers that incrementally update cached objects; similar concepts apply to invalidation of cached objects.

1. **FeatureQuery.** A `FeatureQuery` object depends on data from only one table. For instance, `UserProfile` caches data from only the `profiles` table. There are three triggers associated with a `FeatureQuery` cached object, one each for `INSERT`, `UPDATE`, and `DELETE` on the table. When a cached object is invoked with the parameters associated with that object, it uses these parameters to construct the keys under which that particular data item is cached. For example, `UserProfile` of user with `id` 42 is stored in the cache with a key identified by '42'. To illustrate with a more detailed example, consider the following SQL query corresponding to a Feature Query:

```
SELECT   profiles.id, profiles.user_id,
         profiles.name, profiles.about,
         profiles.location,profiles.website
FROM     profiles
WHERE    profiles_profile.location = 'Boston'
```

   Say, the cached object corresponding to this query class is `UserProfileByLocation`, and it stores a list of profiles of users who are in a particular location. These objects are indexed by a key based on location; for example, the list of users in Boston are indexed by a key identified by Boston, say `UserProfileByLocation:Boston`. On an `INSERT` to the `profiles` table, the key corresponding to the value of `location` in the new row is a candidate for update. So if this location is Boston, the profile of the user corresponding to this new row is added to the

cached object keyed by `UserProfileByLocation:Boston`. Similarly if a row is deleted from the table, the key identified by the `location` field of that row is updated by deleting the profile of the user corresponding to the deleted row. On an update, there are two cases. One, if the location of a row is updated, then the keys corresponding to the old location and new location are updated in cache. Two, if a column other than the location is updated, the key corresponding to the location is updated with the new value of the column. Note that the trigger only updates keys already present in the cache, and does not add any key if it is not already present in the cache. This is to avoid filling the cache with data that might not be used by the application.

2. **LinkQuery.** A `LinkQuery` object depends on data from many tables, and involves joins on foreign-key relationships. Let us call the cached object for Query 2 in Figure 2-3 as `GroupsOfUser`. This query involves a join between the `groups` and `groups_membership` tables, on the foreign-key `group_id` of the `groups_membership` table. For each `LinkQuery`, CacheGenie caches a list of chains formed by these foreign-key relations. So, for `GroupsOfUser` objects, it caches a list of ⟨`group_id`⟩. As in `FeatureQuery` objects, `LinkQuery` objects are also indexed; in this case, `GroupsOfUser` objects are indexed by `user_id`. For example, all groups of user with `id` 42 are cached as a list of ⟨`group_id`⟩ with key `GroupsOfUser:42`. When the application requests this cached `LinkQuery` object, the system first gets the list of groups from the cache and then gets the actual group information. The actual group information is also cached using a `FeatureQuery` object, and this ensures that changes to group information which do not affect the Link Query join do not invalidate or update the cached `LinkQuery` objects.

   To handle consistency of `LinkQuery` objects, we create triggers on all the tables involved in the joins, one each for `INSERT`, `UPDATE`, and `DELETE`. On an `INSERT` into the `groups_membership` table, the key corresponding to the user who added the group is a candidate for update. So if in the new row, `user_id` is 42 and

group_id is 1000, the key GroupsOfUser:42 is updated with the group_id 1000. Similarly if a row is deleted from or updated in the table, the key identified by the user_id field of that row is updated.

For LinkQuery objects with multiple links, the updating mechanism is a bit more involved. For instance, consider the following query to get all the groups of user 42's friends.

```
SELECT  groups.id, groups.name,
        groups.date_creation, groups.description,
FROM    groups, groups_membership, friendship
WHERE   groups.id = groups_membership.group_id AND
        groups_membership.user_id = friendship.to_user_id
           AND
        friendship.from_user_id = 42
```

For the cached object representing this SQL query (say, GroupsOfFriendsOfUser), the system caches a list of ⟨to_user_id, group_id⟩ indexed by the from_user_id. So the groups of friends of user with id 42 are cached as a list of ⟨to_user_id, group_id⟩, with the key as GroupsOfFriendsOfUser:42. For this cached object, there are triggers on the friendship and groups_membership tables. On an INSERT into the friendship table, the key corresponding to the user with id from_user_id is updated with the groups of to_user_id. So for instance, if in the new row, from_user_id is 42 and to_user_id is 24, the system first gets a list of groups of user 24 by querying the database (within the trigger). Say this list is [1000,2000,3000]. The key corresponding to user 42, GroupsOfFriendsOfUser:42, is then updated with the list [⟨24,1000⟩,⟨24,2000⟩, ⟨24,3000⟩]. Note that there are no triggers on the groups table for this cached object since adding a new group will not change any user's existing groups unless a row gets added to groups_membership table.

Now consider an INSERT into the groups_membership table, say with user_id = 42 and group_id = 1000. In this case, the system first gets a list of all users

who are friends of user 42. Lets say the resulting list is [1,2,3]. And then it updates the keys `GroupsOfFriendsOfUser:1`, `GroupsOfFriendsOfUser:2`, and `GroupsOfFriendsOfUser:3` with the tuple $\langle 42{,}1000 \rangle$.

To summarize, to update a `LinkQuery` object, first the system calculates all the keys in the cache that need to be updated and then calculates the new data to be added/removed from those keys.

3. **CountQuery.** As mentioned earlier, a `CountQuery` object can have a `FeatureQuery` or `LinkQuery` as its base class. `CountQuery` objects based on an underlying `FeatureQuery` are cached as a count of rows satisfying the `FeatureQuery`. On an `INSERT` to the underlying table, the key corresponding to the inserted row is determined (exactly as is done for the `FeatureQuery`) and the cached count for that key is incremented. `DELETE` and `UPDATE` on the table are handled in a similar manner, with the cached count being decremented or incremented as required.

   `CountQuery` objects based on `LinkQuery` are implemented in a manner similar to `LinkQuery`, i.e. to cache such a query, CacheGenie caches the list of chains formed by the foreign key relationships. When the application requests the count, it fetches this list from the cache and returns the appropriate count based on the list. Since the data stored in the cache is the same as the underlying `LinkQuery`, the triggers corresponding to such `CountQuery` objects are the same as that for the `LinkQuery`.

4. **TopKQuery.** Like `CountQuery`, a `TopKQuery` object can also be based on an underlying `FeatureQuery` or `LinkQuery`. To cache a `TopKQuery` object, CacheGenie caches an ordered list of results for the underlying query along with the values of the column using which entries in the list are ordered. The size of the list is limited to K elements, as specified by the programmer while defining the cached object.

   Consider for instance, Query 4 in 2-3. Assume this is cached by the name `LatestTwentyStatusOfFriendsOfUser`. For this query, triggers are created on

the `friendship` table as well as the `status` table, which are the tables involved in the underlying join. On an `INSERT` into one of these tables, the keys corresponding to the inserted row are determined, much in the same way as described above for `LinkQuery`. So for instance if user 42 adds a new status message, the trigger determines the friends of user 42, and all the keys corresponding to these users are updated. Since K is 20 and ordered by `date_posted`, the new status is inserted into the list at the correct position, sorted by `date_posted`, and the oldest status is kicked out. The new status is not added to the list if it is older than all 20 statuses already in the list. This can happen when there are frequent status updates and by the time an update is propagated to the cache, 20 other triggers have propagated newer updates.

We believe it is easy to extend the above concepts to other types of queries to generate automatic triggers for managing cache consistency. Next, we discuss the consistency guarantees offered by our system and contrast it with those provided by existing caching systems.

2.2.3 Consistency Guarantees

We have already described the basic mechanisms we provide in CacheGenie to enable cache consistency. In this section we discuss the consistency guarantees we provide with these mechanisms. Following are the various consistency properties of CacheGenie:

1. **Atomic Cache Modification.** CacheGenie ensures individual operations on the cache are atomic. This includes invalidation of a key in cache and updates to cached keys. For example, consider the scenario where two triggers update a cached `TopKQuery` object at the same time. CacheGenie makes sure that the updates are atomic and are applied one after another. This ensures that the cache does not have wrong results.

2. **Immediate visibility of own updates.** The invalidations and updates propagated from the triggers are synchronous. This means that all keys updated

due to an `INSERT`, `DELETE` or `UPDATE` are updated as a part of that statement. The net result is that the user sees the effects of her own update immediately after the update is executed. This is a highly desirable property even for web applications since users expect to see their own updates immediately; delaying these updates leads to a very unsatisfactory user experience.

3. **Commutative Operations on cache.** Another important feature of our Cache Classes is that the updates on the cached objects are commutative. This means that even if the individual updates from two different transactions arrive in different order to the cache, the cache will eventually be in a consistent state. For instance, the result of adding two new status messages to a list of top K status messages of a user is the same, whichever order the new messages get added to the list.

4. **Lag Consistency.** CacheGenie does not currently extend database transactions to the caching layer. This means that updates/deletes applied to the cache as part of one database transaction may be visible to other transactions even before the first transaction completes. We do not provide any sort of read isolation over the cached data. So while one transaction cannot read inconsistent data from the database, it could potentially read inconsistent data from the cache for the time period that the other transaction is executing. Once all the updates have propagated, the cache converges to a consistent state.

5. **Optional Strict Consistency.** We also provide a few mechanisms for the programmer to opt for a strict consistency on a case-by-case basis, if she so desires. In the default case, CacheGenie returns a cached value of a Cache Class query, if available. As described above, this might be stale due to various reasons. 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.

We believe that most of our target web applications, such as social networking applications, do not need strict transactional consistency and that the model we offer suffices for their consistency requirements. It is possible to build caching models with strict transactional consistency, but the overhead of such models may negate the advantages of caching in the first place; we plan to investigate the cost of such transactional models in future work.

Let us compare consistency guarantees of CacheGenie to existing mechanisms that we discussed in Section 2.2.1. An application using expiry intervals on cached objects has to be very tolerant of stale data. Also, choosing the right expiry interval is hard; if the programmer chooses too small an expiry interval, the application incurs too many cache misses whereas if the programmer choose a high value, the application gets more stale data.

Programmers implementing manual invalidation or write-through to cached data, usually provide similar guarantees as CacheGenie. The application is typically structured so that effects of its own writes are immediately visible in the cache. This is done by implementing invalidation as part of the write in the application. Most application caching systems we know of do not implement a strict transactional model over the cache, and are tolerant of a weak consistency model like ours.

None of the database caching approaches we are aware of implements full transactional consistency. DBProxy [22] caches partial tables and computes queries over these tables. It applies changes corresponding to writes in the database in transaction commit order, while ensuring lag consistency and immediate visibility of updates. Similarly, DBCache [24] applies changes in the backend to the cache through a cache daemon; however, it does not ensure that transactions can see their own updates immediately.

GlobeCBC [31] caches query results and propagates invalidations on basis of conflicting templates. Again it does not provide strict consistency with the cache. However it supports lazy invalidation propagation and N-ignorant transactions to decrease traffic at cost of weaker consistency. Ferdinand [25] relaxes consistency for

multi-statement transactions but ensures full consistency when only single-statement transactions are used. This is slightly stronger than the consistency we offer because our system might do multiple updates as part of a single statement and since we do not have read isolation, this may result in transactions reading dirty data of other transactions.

Chapter 3

Implementation

A popular method of developing dynamic web applications today is to use web application frameworks. These frameworks simplify web application development by providing libraries for database access, templating frameworks and session management, and promoting code reuse. Examples of such frameworks are JavaEE (Servlets), WebObjects, Ruby on Rails [14], Django [6], and Zend Framework [21]. These frameworks typically also provide support for using caches to speed up application performance. The mechanisms available here are similar to what we discussed in Chapter 2. They provide support for page-level and fragment-level caching, or for simple key-value pair caching, and it is up to the programmer to deal with cache consistency.

Since the goal of our work is to make it easy for web application programmers to incorporate caching in their applications, a web application framework is the right place to implement our caching abstractions. We implemented a prototype of CacheGenie by extending a popular web application framework called Django. We provide high-level caching abstractions as special primitives in Django—a programmer can easily use these primitives to cache frequently accessed queries that fit these abstractions, and CacheGenie takes care of keeping these query results up-to-date in cache.

Another advantage of using Django is that there are several open-source web applications implemented on top of Django, which we can use to test the performance and usability of our system. One such suite of reusable Django applications geared towards online social networking is Pinax, which is what we use in our evaluation.

Note that even though we picked Django for our prototype implementation, it should be relatively straightforward to apply the techniques developed in this work to other web application frameworks.

We use memcached [10], a popular high-performance, distributed memory object caching system as our caching system. For the backend persistent storage we use Postgres [13], an advanced open-source database. Again, even though we picked these specific systems for our prototype, we believe the concepts are general enough to be applied to most database and caching systems.

3.1 Django

Django is a open-source web application framework based on Python. It provides reusable and pluggable components for common web development activities. It is based on model-view-controller design pattern and is geared towards rapid development of dynamic database-driven web applications.

The core Django framework consists of (i) an object-relational mapper (ORM) which mediates between data models (defined as Python classes) and a relational database, (ii) a regular-expression-based URL dispatcher, (iii) a view system for processing requests, and (iv) a HTML templating system. `Models` in Django are Python classes, which describe a database table. Using models, one can create, retrieve, update and delete records in the database using simple Python code rather than writing repetitive SQL statements. `Views` are Python functions which contain the logic for a webpage. The URL dispatcher specifies which view is invoked for a given URL pattern. And finally, the HTML templates describe the design of the page. Django provides a template language with basic logic statements.

3.1.1 Overview of Django Models

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. All model classes inherit from the base class called `Model`, which contains all the machinery necessary to make these objects capable of interacting with a database. For instance, the model class definition for Table 1 in Figure 2-2 looks like:

```
class User(models.Model):
    username    = models.CharField(max_length=30, unique=True)
    first_name  = models.CharField(max_length=30)
    last_name   = models.CharField(max_length=30)
    email       = models.EmailField()
    password    = models.CharField(max_length=128)
    last_login  = models.DateTimeField()
    date_joined = models.DateTimeField()
```

Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., `CharField`) corresponds to the database column type (e.g., `varchar`). Django automatically gives every model an auto-incrementing integer primary key field called `id`.

Further, Django automatically provides a high-level Python API to retrieve objects from the database using the concept of a `QuerySet`. A QuerySet represents a collection of objects from the database. It can have zero or more filters that narrow down the collection based on the specified parameters. In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT. Django also provides the option of executing raw SQL queries if the programmer so desires. Here are some examples of queries over the `User` model :

```
#Create a new user
user1 = User(username='bob007',first_name='Bob' ... )
#Save the object into the database.
user1.save()
#Get all users
user_list = User.objects.all()
```

```
#Get all users who joined in 2007
User.objects.filter(date_joined__year = 2007)
#Count of users
User.objects.count()
#Get email of user1
user1.email
```

Django provides ways to define the three most common types of relationships between database tables: many-to-one, many-to-many and one-to-one. A many-to-one relationship is created by defining a `ForeignKey` field on the related model. So for instance, the `user_id` column in Table 2, `profiles`, is a foreign key reference to `id` column in `user` table. This is represented in Django as follows:

```
class Profiles(models.Model):
    #...
    user = models.ForeignKey(User)
    #...
```

Similarly one can define many-to-many relationships using a `ManyToMany` field, one-to-one relationships using a `OneToOne` field. A separate table is created in the database for a many-to-many relationship.

3.1.2 Caching in Django

Django provides support for caching at different granularity, as well as different caching backends. It provides support for using memcached, which is what we use in our prototype. Django provide caching at various levels, such as site-level caching, view-level caching, template fragment caching and finally, key-value caching. All these options however suffer from the limitations of application-level caching as discussed in Section 2.2.1—the developer has to manually manage the invalidation and expiry of cached objects. In the next section we describe our implementation of CacheGenie with Django, which exposes a better caching interface to the programmer.

3.2 CacheGenie in Django

In Chapter 2, we discussed the various caching abstractions based on frequently observed query patterns in web applications. Now we describe how these high-level abstractions are implemented in Django.

3.2.1 Cache Class Implementation

We implemented Cache Classes corresponding to `FeatureQuery`, `LinkQuery` and `CountQuery` in Django. We have not so far implemented `TopKQuery` class, mainly because the specific Django workload we were working with did not have such queries. But the concepts involved in implementing it are similar to those of the other cache classes and it should be straightforward to implement it based on the design discussed in Chapter 2. Our prototype supports invalidation, update-in-place and expiry interval for consistency management of all these classes; however, it doesn't yet support prefetching or varying update strategies based on popularity of the cached objects.

A Cache Class performs the following functions:

- **Query generation:** It use 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. This is because if the cached data was Python objects, database triggers will have to construct and/or deconstruct these objects to update them in the cache. This results in slow triggers that block the database, making the overall database performance worse. Hence we trade increased computation in client for better database performance. Another point to note here is that the query derived in this step is the query template representing this cached object that is evaluated only at runtime when the correct arguments are supplied. For instance, for a `UserProfile` object, the query template looks like:

```
SELECT   *
FROM     profiles
WHERE    profiles.user_id = %d
```

- **Trigger generation:** This involves determining the database tables and the operations on these tables that need triggers to keep the cached object consistent with the database. It also includes generating the necessary code for the triggers, as described in Section 2.2.2.

- **Query evaluation:** When the application requests the cached object with the correct arguments, the Cache Class fetches the appropriate key from the cache and transforms the returned value into the form required by the Django application (typically model objects). If the key is not present in the cache, it queries the database with the query generated during the generation step, adds the result to cache, and returns 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, reqd_params, opt_params):
        # Checks whether all required parameters
        #  for this cache class are provided
        # Implements Query generation

    def get_trigger_info(self):
        # Implements Trigger generation

    def evaluate(self, *args, **kwargs):
        # Implements Query evaluation

    def make_key(self, *args, **kwargs):
        # Returns the key in cache corresponding
        #  to the provided arguments
```

The `reqd_params` argument provides all the required parameters for this cache class, and similar `opt_params` are the optional parameters.

3.2.2 `cacheable` Function

To create a cached object, the programmer uses a function called `cacheable`:

```
def cacheable(cache_class_type,reqd_params,opt_params):
    #...
    return cached_object
```

This function performs the following tasks:

1. Creates the specified `cache_class_type` object with the given parameters and returns it.

2. Maintains a map of all cached objects created, keyed by the parameters. This is useful for fetching the cached objects automatically, and also ensures that there is only one instance of the cached object with a particular set of parameters.

3. Collects all the triggers information from individual cached objects and installs them on the database. This enables combining of various triggers on the same table to improve performance.

Once the cached object is created, the programmer can either access it directly using the `evaluate` function, or let the system automatically use a cached version whenever available. This is done by intercepting normal Django `QuerySet` queries and using the above created map to figure out if there is a cached version, and returning the cached result transparently.

3.2.3 An Example

We illustrate the use of `cache class` and `cacheable` function in context of an example. We have seen earlier a Django model class representing a user. Similarly, the model class representing the statuses of users (Table 7 in Figure 2-2) is:

```
class Status(models.Model):
    name          = models.CharField(max_length=30, unique=True)
    user          = models.ForeignKey(User,\
```

```
                    related_name = 'posted_status_list')
    statustext   = models.CharField(max_length=140)
    date_posted = models.DateTimeField()
```

Specifying the related_name parameter as posted_status_list in the ForeignKey field user allows accessing a user's statuses through a user model object. Also, we add a field to the User model to represent the friendship relationship:

```
class User(models.Model):
    # ...
    friends = models.ManyToManyField('self', db_table = '
       friendship')
```

Django creates a separate table called friendship in the database for the above many-to-many relationship, similar to Table 3 in Figure 2-2.

In this context, imagine that the developer wants to cache all the status messages of all friends of a user. (In reality, one would cache only the latest status messages, and that corresponds to a TopKQuery, but we use this hypothetical example to illustrate a LinkQuery.) To do that in Django, the developer currently writes the following code:

```
u = User.objects.get(id=42)
my_newsfeed = Status.objects.filter(user__friends = u)
```

This is interpreted as following the foreign-key links from status objects to the user who created that status, and then the friends of that user. To cache this in CacheGenie, the developer first needs to add a cached object definition like so:

```
user_newsfeed = cacheable(
    cache_class_type = 'LinkQuery',
    reqd_params = {
        'main_model' : 'User',
        'related_fields' : ['friends','posted_status_list']
    }
    opt_params = {
```

```
        'expiry_interval' : 0, # means never expire
        'update_strategy' : 'update', # update-in-place
        'use_transparently' : True
    }
)
```

The reqd_params here show how this LinkQuery cached object represents the Django code above. The main_model is User, which means we are interested in following the links starting with a particular user. The related_fields specify the foreign-key links to be followed. In this case, the programmer wants to follow the friends relation first and then get the posted_status_list of those users.

The values in opt_params are the default values of those parameters and did not need to be specified; however, they are mentioned here for the sake of illustration. Once this is done, the system retrieves the cached data when the original code is executed.

Optionally, the programmer can call evaluate on the returned cached object user_newsfeed with the id of the desired user:

```
my_newsfeed = user_newsfeed.evaluate(42)
```

To maintain this cached object, triggers are automatically generated and installed on the tables friendship and status on INSERT, DELETE and UPDATE.

3.3 Memcached

Memcached is an open-source distributed memory object caching system [10]. It is a key-value store which can be scaled over hundreds of machines easily, providing high performance. This is unlike a relational database, which is much harder to scale because it provides support for general query languages like SQL. Memcached is used by several very large, well-known sites including Facebook [7], LiveJournal [9], Wikipedia [19], Flickr [8], Twitter [17], Youtube [20], Digg [5], and Craigslist [4].

There are many client libraries available to interact with memcached servers in most languages. For Django we use a python library called python-libmemcached.

This is a Python wrapper around `libmemcached`, which is a C client library that is significantly faster than the python-based libraries.

In memcached, the servers do not know of each other and do not communicate with each other. The clients have information about all the servers and are responsible for determining which server stores which key. Recent memcached clients provide support for Consistent Hashing across servers. This model allows for a more stable distribution of keys given addition or removal of servers. In a normal hashing algorithm, changing the number of servers can cause many keys to be remapped to different servers, causing many cache misses. Consistent Hashing maps keys to a list of servers, such that adding or removing servers causes a very minimal shift in where keys map to. `python-libmemcached` provides support for consistent hashing.

We have memcached servers running independently of the application servers (running Django). All the application servers talk to the same set of memcached servers and share the cached data. One could potentially have multiple such clusters, each having a layer of application servers and memcached servers. However, if these clusters cache data from a common underlying database, then one has to make sure the data from all these clusters is properly invalidated/updated when the underlying data is modified.

Memcached supports two types of protocols, a text protocol and a binary protocol. Binary protocol is more recent and provides support for many useful features such as compare-and-swap support, append and prepend calls. Compare-and-swap (CAS) functionality is crucial for our system to ensure that two database triggers executing at the same time do not end up overwriting each others modifications to the same key. With CAS, a `gets` call returns a token associated with that key. When calling `set` on the same key, the client can send this token back and memcached ensures the `set` call succeeds only if the current token value of the key matches this token. CAS support is another reason why we chose `python-libmemcached` since this is the only python library which currently provides CAS support.

Memcached uses the Least Recently Used (LRU) policy to evict items when the cache becomes full. When all the memory allocated to it is used up, memcached

reclaims the LRU item. To do this, it searches the last few LRU items for one that has already expired, and is thus free for reuse. If it cannot find an expired item, it evicts one which has not yet expired. This policy serves the purpose of a web application workload well since keys which are not being fetched anymore will slowly fall off. This is the only eviction protocol currently supported by memcached. In our system, apart from the application, triggers also fetch the keys for updating them. This leads to bumping those keys to front of LRU even though they are not really being 'used' by the application. It would be great to have a way to specify when a particular call to memcached should bump the key in LRU. For now, we just use the default option available in memcached.

3.4 PostgreSQL

PostgreSQL (or Postgres) is an advanced open-source relational database. We used Postgres as our underlying persistent data store. Django provides support for Postgres so we did not have to make any modifications to the applications for this.

Django, by default, creates indexes on primary keys (`id`) of the tables. However none of the tables were clustered leading to poor performance for certain queries such as count. We created a few more indexes depending on the workload and also clustered the index appropriately. This improved the performance of count queries significantly.

Another feature of Postgres that we use is triggers. A trigger in Postgres is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. They are implemented using Trigger functions, which are user-defined functions with the return type trigger. Trigger functions can be written in many procedural languages including Perl, Python, pgSQL, C and Tcl. Since our application servers are also in Python, we chose to write the our trigger functions in Python. One can also use C to write trigger functions but writing trigger functions in C involves managing several low-level details, which are already taken care of in higher-level language triggers. This is another reason we

chose to go with Python triggers. Exploring C triggers and whether they improve trigger performance is a piece of future work.

Chapter 4

Evaluation

In this chapter, we evaluate the CacheGenie prototype and present the results. The primary goal of this work is to provide programmers with useful high-level caching abstractions and unburden them from the task of cache management. Thus, an important evaluation metric for our system is an estimate of the reduction in programmer effort. We port a subset of Pinax, a reusable suite of Django applications geared towards online social networking, to use CacheGenie. Using this set of applications, we compare the amount of code a programmer needs to write to manage cache with and without CacheGenie.

The second aspect of our evaluation is performance. We measure the overall performance of CacheGenie under varying parameters, such as different types of workloads, user distribution, and caching strategies. Further, we measure the overhead induced by triggers, and study their impact on the overall performance.

4.1 Experimental Application: Pinax

Pinax is an open-source platform for rapidly developing websites and is built on top of Django. It enables this by providing an extensive set of reusable Django applications. These applications take care of the common tasks involved in building many kinds of sites and lets the developer focus on the more important details. Moreover, the applications already provided in Pinax can be mixed and matched to create various

kinds of sites.

During its initial development, Pinax was used to create a social networking web site, later spun off as Cloud27 [3]. As a result, it provides most of the basic components required to build a social networking site such as profiles, friends, microblogging, bookmarks, messages, and so on. According to Pinax homepage, it is now used by the following websites—CarPosse [2], we20 [18], mftransparency.org [11], SequenceMedical [15] and TuttiVisti [16].

We use Pinax to test both the usability and performance of CacheGenie. We chose Pinax because it provides basic components of real-world social networking type applications, which serve as excellent examples of modern web applications. This allowed us to analyze the query workload and the dependencies between the data that users of a social web application are interested in. It also enabled us to test the applicability of our caching abstractions for these applications and to estimate the amount of change needed in the application to use CacheGenie. And finally, we could evaluate the performance of our system under realistic workloads on a realistic web application.

4.1.1 Background Information on Pinax

The social networking project in Pinax is a collection of various applications associated with a typical social networking site. In Django terminology, a site is made up of many applications—essentially the various independent services on the site. We focused on three applications from the social networking project—profiles, friends and bookmarks. Thus our site consists of users who have a profile each, with users connected to other users by the friendship relationship. Each user makes a list of their favorite bookmarks, either on their own or by marking bookmarks already created by other users. We trimmed out most of the other applications except for certain integral parts of the site such as messages, announcements and avatars, which were not removed. However, these are passive applications and we do not add any data to the tables corresponding to these applications; also the queries corresponding to these applications do not add too much overhead. A list of database table schema

generated by the Django for the site is given in Appendix A.

Figure 4-1 shows a few snapshots from the site. The particular user actions we are interested in are:

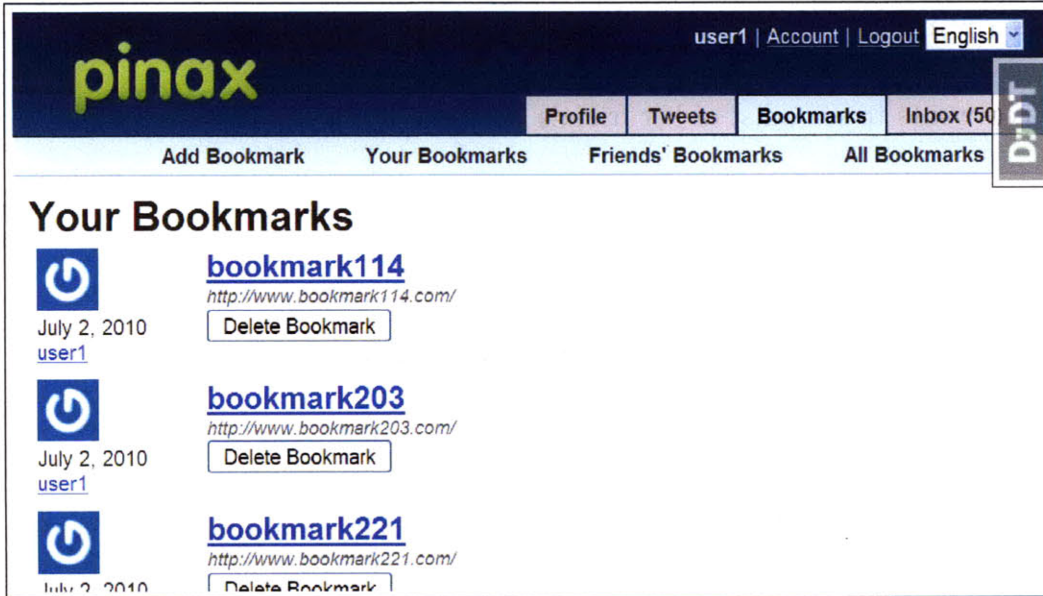
- A user wants to see a list of her own bookmarks (Figure 4-1a).
- A user wants to see a list of her friends.
- A user wants to add a new friend or accept a friend invitation from another user. (Figure 4-1b).
- A user wants to look at a list of bookmarks created by her friends (Figure 4-1c).
- A user wants to add a new bookmark (Figure 4-1d).

Each of these actions has a web page that corresponds to it, as shown in the Figures.

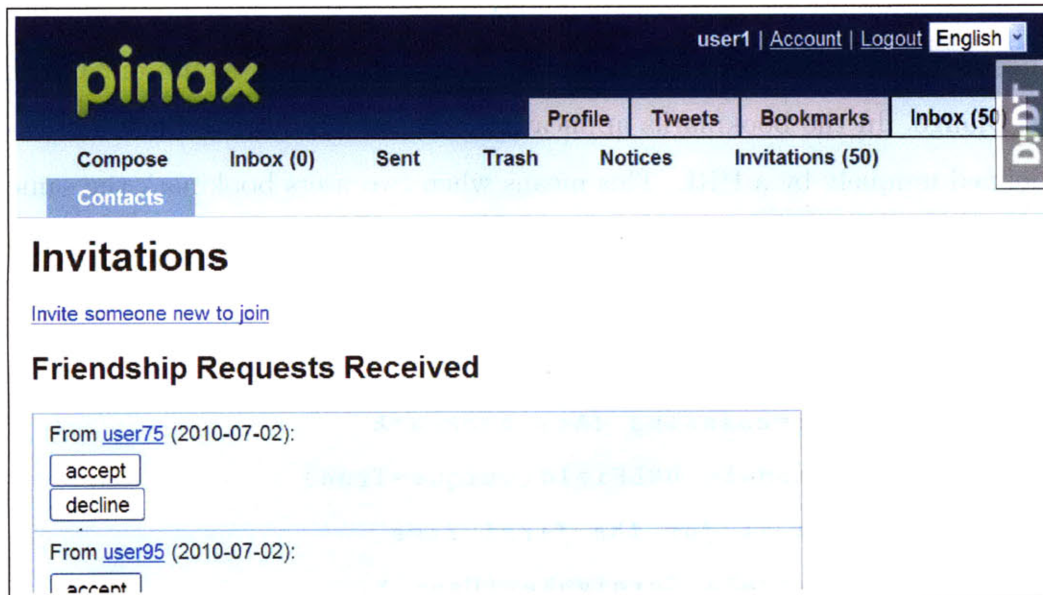
4.1.2 Porting Pinax Applications

Now we describe how we modify the subset of Pinax applications to use our Cache Classes in Django. In the bookmarks application that we discussed earlier, a bookmark is defined uniquely by a URL. This means when two users bookmark the same URL, they create two instances of this bookmark. This is represented by two Django models, `Bookmark` and `BookmarkInstance`:

```
class Bookmark(models.Model):
    # unique URL representing this bookmark
    url          = models.URLField(unique=True)
    # user who added it for the first time
    adder        = models.ForeignKey(User,\
                   related_name='added_bookmarks')
    added        = models.DateTimeField(default=datetime.now)
    description  = models.CharField(max_length=100)
    note         = models.TextField()
```



(a) User's Bookmarks



(b) Friendship Invitations

Figure 4-1: Pinax Social Networking Website

pinax

user1 | Account | Logout English

Profile Tweets Bookmarks Inbox (50)

Add Bookmark Your Bookmarks Friends' Bookmarks All Bookmarks

Friends' Bookmarks

These are bookmarks from your friends:

Sort by: reddit-like hotness or total points or **date added**

July 2, 2010
user386
0 points
bookmark590
http://www.bookmark590.com/
Saved 9 times (save)

July 2, 2010
0 points
bookmark960
http://www.bookmark960.com/
Saved 6 times (save)

(c) User's Friends' Bookmarks

pinax

user1 | Account | Logout English

Profile Tweets Bookmarks Inbox (50)

Add Bookmark Your Bookmarks Friends' Bookmarks All Bookmarks

Add Bookmark

You can drag this 'Add to Pinax Bookmarks' link to your bookmark bar to post the pages you visit!

URL*
www.techcrunch.com

Description*
TechCrunch

Note
Latest technology news..

(d) Add a Bookmark

Figure 4-1: Pinax Social Networking Website

```
class BookmarkInstance(models.Model):
    # Bookmark object of which this is an instance
    bookmark     = models.ForeignKey(Bookmark,\
                   related_name='saved_instances')
    # user who saved this instance
    user         = models.ForeignKey(User,\
                   related_name='saved_bookmarks')
    saved        = models.DateTimeField(default=datetime.now)
    description  = models.CharField(max_length=100)
    note         = models.TextField()
```

We created cached objects for the frequent and/or expensive queries involved in performing the actions listed in the previous section, such as looking up bookmarks and adding bookmarks. A few of them are listed here:

1. **Fetch a user's saved bookmarks.** Existing code to get a user's saved bookmarks from the database is:

   ```
   u = User.objects.get(id=42)
   user_bookmarks = Bookmark.objects.filter(
      saved_instances__user=u)
   ```

   To cache this pattern of queries, namely, bookmarks of a user, we add the following cached object definition to the application.

   ```
   user_bookmarks = cacheable(
       cache_class_type = 'LinkQuery',
       reqd_params = {
           'main_model' : 'User',
           'related_fields' : ['saved_bookmarks','bookmark']
       }
       # If the optional parameters are omitted,
       # default values are used
   )
   ```

This enables automatic caching of queries of the above type, and further, it is used automatically when the existing code is executed. The programmer can also manually access the cached data by calling the `evaluate` method on the `user_bookmarks` object with the right parameters, for instance 42 in this case.

2. **Count of saved instances of a bookmark.** Existing code to get the count of saved instances of a unique bookmark from the database is:

```
b = Bookmark.objects.get(id=1)
BookmarkInstance.objects.filter(bookmark=b).count()
```

To cache this count query, we add the following cached object definition to the application.

```
bookmark_count = cacheable(
    cache_class_type = 'CountQuery',
    reqd_params = {
        'main_model' : 'BookmarkInstance',
        'where_fields' : ['bookmark']
    }
)
```

Again, once this cached object has been defined, the existing code automatically gets the corresponding cached data.

Similarly we create cached objects for a user's friends' bookmarks (`LinkQuery`), a bookmark's data given its id (`FeatureQuery`), user's profile given user's id (`FeatureQuery`) and so on.

4.2 Programmer Effort

In this section we quantify the changes made to Pinax applications to port them to use CacheGenie. Second, we estimate the amount of work a programmer has to do to manually manage the cache for queries that we automatically manage in our system.

As described in Section 4.1.2 with the bookmarks example, the programmer needs to add a cached object definition for each query pattern instance she wants CacheGenie to cache for her. For this purpose, first she needs to identify queries in the application which fall into the Cache Classes we provide. Next, she needs to create a cached object for those queries using the `cacheable` function. To port the subset of Pinax applications described in Section 4.1.1, we added 14 cached objects. 3 of these are `LinkQuery` class, 5 are of `CountQuery` class and 6 are of `FeatureQuery` class. 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 and managed for those queries. Currently, however, we have not implemented this for `LinkQuery` and so we had to write about 6 lines of code to use the 3 `LinkQuery` cached objects. For the rest, the modified Django automatically puts and gets the cached query results from the cache. In the absence of CacheGenie, the programmer has to manually write code to get and put data in cache wherever a query is being made. In our sample application, we counted 22 explicit locations in the code where such modifications are necessary. However, there are many instances where the query is being made implicitly and the programmer would not be able to cache them manually without changing the structure of the code significantly. Also, our application consists only of four types of pages. A realistic application has many more types of pages, and hence we believe that in a real application, the programmer has to manually perform cache operations in many more locations.

To manage the invalidations and updates to the cache, we automatically generate triggers corresponding to the defined cached objects. There are three triggers, one each for insert, delete and update, on each of the tables involved in computing the query. So with our 14 cached objects, we have 48 triggers. These 48 triggers have 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. In frameworks like Django, the code for writes to each table is centralized in one place. Hence the programmer can possibly write one piece of code

for each table and cached object combination, similar to the one set of triggers for each combination in CacheGenie. Hence, we argue that without an automatic cache management scheme, the programmer will have to write about the same lines of code as our generated triggers, i.e. 1720 lines of code. In short, this is the amount of code that the programmer does not have to write to manage the cache for this small application if using CacheGenie. In practice, applications have a lot more cached objects and hence many more lines of code for cache management.

From this experience, we conclude that it requires little effort on the part of programmer to use our abstractions to get automatic cache management.

4.3 Performance

As described in the previous chapters, we provide two strategies to the programmer for cache management—automatic invalidation and automatic updation of cached data. To evaluate the performance benefits of these caching strategies, we compare three systems: (i) No Cache—a system with no caching and all requests 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 use a workload generated from Pinax applications ported to CacheGenie.

Our experiments were designed to answer the following questions:

1. What is the improvement in a web application's performance due to caching?
2. Does updating cached data provide overall better performance than invalidating it? If so, by how much?
3. How does the benefit of caching change as we vary the workload (reads vs writes)? Also, does the advantage of update over invalidate vary with workload?
4. How does benefit of caching depend on the distribution of users in the workload—how much does it improve if there are more repeated users as opposed to more distinct users?

5. What is the overhead of database triggers and automatic cache management? What is the extra cost of a write operation in the cached scenarios?

6. How big a cache do we need in the system in steady state, as compared to the size of the database? How does this vary with the workload?

We conducted various experiments to answer the above questions. The remaining sections discuss these experiments in detail.

4.3.1 Experimental Setup

As illustrated in Figure 2-1c, our experimental setup comprises of three main components: (i) the application layer (web clients + webapp server), (ii) the cache layer and (iii) the database layer. Now we describe each of these components in more detail.

Application 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'. The goal of the application layer (or *app layer* for short) is to simulate a realistic workload for a social-network style web application and generate the corresponding queries to the data backends. It has three functions: (i) simulate the web browser clients and issue GET/POST requests, (ii) act as the web server—serve static pages directly, pass on dynamic page loads to the application server, and serve the response back to web clients, and (iii) act as the application server and process the dynamic requests.

We use a functionality provided by Django called the *test client* to help us implement the app layer. *Test client* is a Python class that acts as a dummy Web browser, allowing one to interact with a Django application programmatically. It simulates GET and POST requests on a URL, processes them in the exact same way as if a web client is requesting them, and returns the HTML response. Using this class, we can simulate all three functions of the app layer in a single Python process—it acts as the web client, the web server and the application server. Note that Django is

single threaded, and so is the test client. This means a single test client instance only issues and process requests sequentially. The term client in the rest of the evaluation will refer to one such instance. The application layer then consists of many such clients running in separate processes and issuing requests, overall simulating a web application workload.

We use the same set of Pinax applications described in Section 4.1.1 for our evaluation. The database is initialized with certain number of users, bookmarks and friendships between users. The experimental workload consists of users logging into the site, performing certain actions according to some distribution and logging out. Specifically, our workload consists of the following actions:

1. Lookup Bookmarks (LookupBM). The user loads a page showing all her bookmarks. The corresponding URL (relative to the main site) is '/bookmarks/your_bookmarks.html'. (Figure 4-1a)

2. Lookup Bookmarks of Friends (LookupFBM). The user loads a page showing all bookmarks of her friends. The corresponding URL is '/bookmarks/friends_bookmarks.html'. (Figure 4-1c)

3. Create Bookmark (CreateBM). The user loads the page at '/bookmarks/add' and submits a form on this page to create a new bookmark. (Figure 4-1d)

4. Accept Friend Invitation (AcceptFR). The user accepts one of the pending friendship invitations, thus creating a new friend. The database is initialized with a number of friendship invitations. The URL for friend invitations is '/invitations'. (Figure 4-1b)

The first two actions are read only, while the second two actions involve write queries. The default ratio of these actions in our workload is ⟨LookupBM : LookupFBM : CreateBM : AcceptFR⟩ = ⟨50 : 30 : 10 : 10⟩. We can also look at it as the ratio of read pages (LookupBM + LookupFBM) to write pages (CreateBM + AcceptFR). The default ratio then is 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. This is supported by the study of user activities by Benevenuto et al [23], where they found that browsing activities (involving no writes) comprised of about 92% of all requests in their 12-day request data for Orkut [12]. 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. So, the actual percentage of write queries measured against total queries in our workload is lower than the percentage of write pages, which is the parameter used in our evaluation.

The set of actions from a user's login till 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. Each session in turn comprises of 10 page loads, in the ratio specified above. Each page load consists of a variable number of queries, on an average 80.

The distribution of users across sessions is according to a *zipf* distribution. Benevenuto et al [23] have studied frequency of users logging into social networking sites over a period of 12 days. According to their work, the majority of users (63%) accessed the social network aggregator's site only once during the 12-day period. From their data, we derived the frequency of user sessions to be zipf-like with the value of zipf parameter being 2.0. We also use [28] and [27] to help us create a realistic social networking workload.

Running the entire Django Python clients for the experiments leads to a high CPU utilization, and to saturate the database in this setting requires a large number of client machines. To avoid this, we first do a trial run with the full Django clients and record all the queries being made to the database and cache by all the clients. We also record the beginning and end of user sessions, as well as beginning and end of page loads. For the actual experiment we only replay these logs. Since a particular user is always handled by only one client, the timing of queries between any two clients does not create any inconsistencies. In this way, we are able to saturate the database

using multiple clients running on a single machine. The client machine is an Intel Core i7 950 with 12GB 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 a gigabit ethernet local network.

Database Layer

The database machine is an Intel Xeon CPU 3.06 GHz with 2GB of RAM, running Debian Squeeze with Postgres 8.3. We changed a few of the configuration settings to enable Postgres to run with high load, including switching off logging, increasing `shared_buffers` and increasing `checkpoint_segments`.

The database is initialized with the following data (Appendix A describes the exact schema) :

- **Users:** The `users` table is initialized with 1 million users. The `profiles` table is also initialized with 1 million profiles, one per user.
- **Bookmarks:** The `bookmarks` table is initialized with 1000 bookmarks. The `bookmarkinstance` table is initialized with a random number of bookmark instances (between 1 and 20) per user.
- **Friends:** The `friendship` table is initialized with random number of friendships, between 1 and 50, for each user. Further the `friendshipinvitation` table is initialized with a random number of pending friendship invitations, between 1 and 100, for each user.

The total database size is about 10 GB. The tables are indexed and clustered as shown in the definitions in Appendix A.

Caching Layer

Our caching layer consists of memcached 1.4.5 running on a Intel Pentium 2.80GHz with 1 GB of RAM. We did not make any changes to the configuration of memcached since it scaled well and was never a bottleneck in any of our experiments. The size

of the cache depended on the experiment, but for most experiments it was 512MB. Note that this is only an upper limit on the amount of memory it *can* use, if needed. As described in the implementation we use the binary protocol to communicate with memcached, both from the clients as well as database triggers.

4.3.2 Microbenchmarks

Although our main evaluation uses Pinax, we also performed a set of microbenchmarks to better understand the performance characteristics of the cache, the database and the database triggers. We describe two such microbenchmarks in this section. The machine configurations for the clients, database and memcached are the same as described above. However, we use a simple one table database and simple queries in these microbenchmarks.

We create a simple database with a single table called `map`, with the following schema:

```
map (key      character varying(13)
     value    character varying(1000));
```

Each `key` is unique and the table has a btree index on the `key` column. We created four configurations of this database by varying the number of rows and the size of `value` column (in bytes). These configurations are enumerated in Table 4.1. The table also shows the approximate size of the database for each configuration.

We measured the performance of the following queries on each of the database configurations:

Database	#Rows	Size of Value	Size of DB
DB 1	10 million	10 B	1 GB
DB 2	50 million	10 B	5 GB
DB 3	1 million	1000 B	1 GB
DB 4	5 million	1000 B	5 GB

Table 4.1: Various Database Configurations used in Microbenchmarks

- DB SELECT (RANDOM)

  ```
  SELECT value FROM map WHERE key = 'key0000000001';
  ```

  We measured 1000 such select queries, choosing a random `key` to lookup each time, and averaged the results.

- DB SELECT (REPEATED)

  ```
  SELECT value FROM map WHERE key = 'key0000000001';
  ```

  We measured 1000 such select queries, with the same `key` each time, and computed average of the results.

- DB INSERT

  ```
  INSERT INTO map VALUES ('key0000000001', '7864382875...');
  ```

  Here, the `value` inserted is a random string of the size specified by that database configuration. We measured 1000 such insert queries, and computed average of the results.

Next, we perform equivalent `get` and `set` queries on the memcached. The memcached queries are:

- MC GET

  ```
  cache.get('key0000000001');
  ```

  We did 100000 such `get` queries, choosing a random `key` to lookup each time and took average over the results. Further, we ensured that all the keys which are looked up are present in the cache beforehand.

- MC SET

  ```
  cache.set('key0000000001', '7864382875...');
  ```

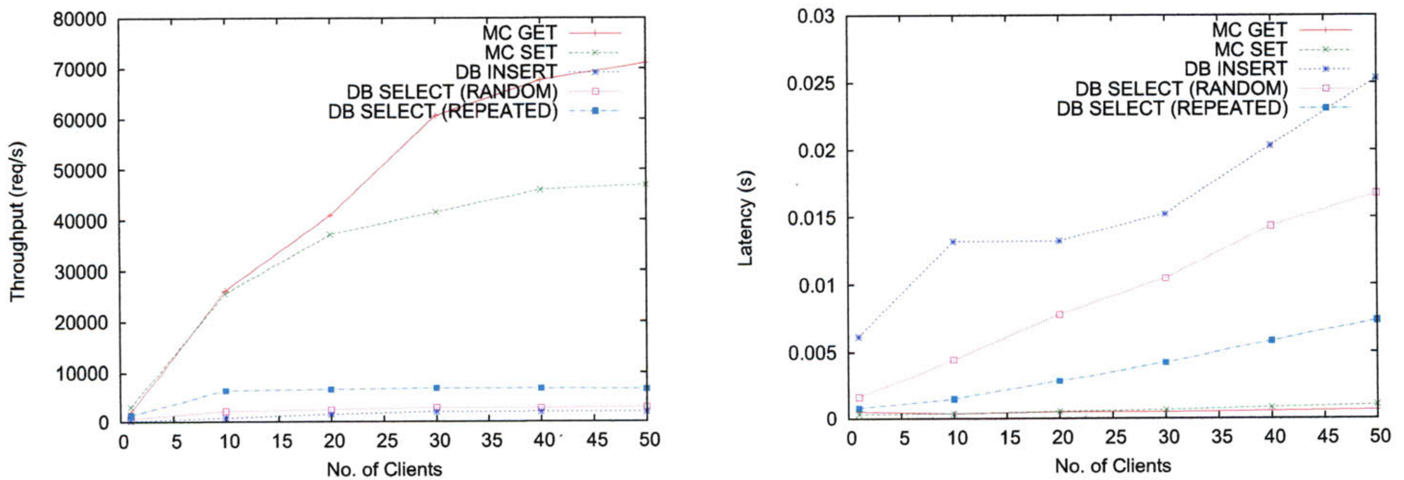
  Here, again, the `value` inserted is a random string of size either 10 or 1000 bytes. We took measurements of 100000 such set queries and computed averages over them.

Database vs Cache performance

In this microbenchmark we wanted to answer the following question:

How does raw performance of a relational database compare with the raw performance of a cache in terms of simple read and write queries?

We chose the database configuration DB 3 from Table 4.1 for this comparison. We ran the database and memcached workload with varying number of clients and measured average throughput and latency of each type of query. The results are shown in Figures 4-2. Figure 4-2a shows how the throughput varies with increasing number of clients, and Figure 4-2b shows the corresponding query latency variation. The size of `value` for the memcached queries is 1000 B.



(a) Query Throughput

(b) Query Latency

Figure 4-2: Microbenchmarks: Database vs Cache Performance

From the first graph, we see that MC GET throughput is about 25 times that of DB SELECT (RANDOM) throughput and about 10 times that of DB SELECT (REPEATED). This shows that even for repeated queries, memcached performs better than a database. We believe this is due to the overhead of query planning in the database. Moreover, MC SET throughput is about 25 times more than that of DB INSERT. The latency graph shows similar improvements over all three database

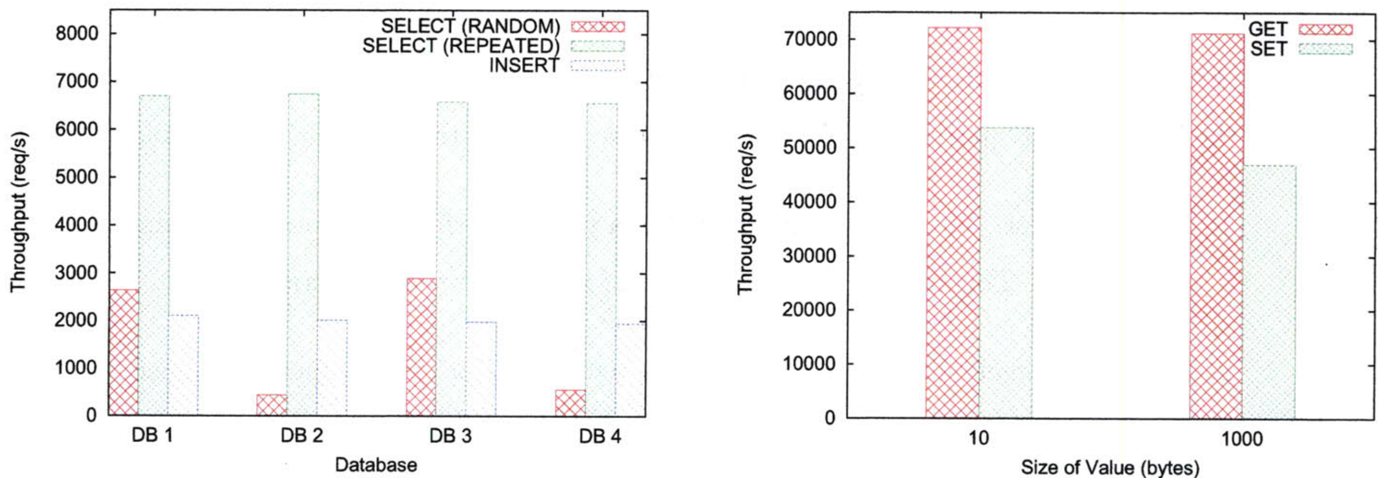
queries.

Effect of Varying Data Size

We aim to answer the following question with this microbenchmark:

How does database and cache performance vary with the size of the database and size of data read or written in each query?

For this, we compare the maximum query throughput achieved by each of the four database configurations in Table 4.1, for the three types of database queries. Also, we observe the variation in MC GET and SET performance as the size of `value` in the query changes. The results for this experiment are shown in Figure 4-3.



(a) Throughput with varying Database size

(b) Cache Throughput with varying Value size

Figure 4-3: Microbenchmarks: Effect of Varying Data Size

From these results, we draw the following conclusions:

1. DB SELECT (RANDOM) throughput drops by 5-6 times as the database size increases from 1 GB (DB 1, DB 3) to 5 GB (DB 2, DB 4). DB SELECT (REPEATED) and DB INSERT, however, do not show a significant performance drop. (Figure 4-3a)

2. Increasing the size of `value` column from 10 bytes (DB 1, DB 2) to 1000 bytes (DB 3, DB 4) does not have a significant effect on the performance of database queries. (Figure 4-3a)

3. Increasing the size of `value` in memcached queries leads to a 15% drop in the performance of MC SET, while it has no effect on MC GET. (Figure 4-3b)

4. The results show that the relative win of memcached over database for read queries increases to a factor of 160 as the database size increases.

Trigger Overhead

With this microbenchmark, we want to answer the following question:

What is the overhead of launching a database trigger? How does this overhead vary when the trigger performs queries to the cache and the database?

For this experiment, we chose DB 4 from the Table 4.1. We tested the performance of the DB INSERT query with following *after insert on row* triggers on the `map` table:

1. No-op Trigger. This trigger just launches a python script that does not do any useful work. This measures the overhead of simply launching a trigger.

2. MC-1op Trigger. This trigger launches a python script, which opens a connection to memcached and does 1 set operation. This measures the overhead of opening a connection to the cache inside a trigger, which happens in our actual triggers for Pinax applications.

3. MC-100op Trigger. This trigger launches a python script, which opens a connection to memcached and does 100 set operations. This measures how expensive it is to perform a memcached operation from the trigger.

4. DB-1op Trigger. This trigger launches a python script, which does a query to the same database. This measures overhead of doing SQL queries to the same database within a trigger. Some of our actual triggers need to do this in order to figure out which cached entries to update.

Trigger Type	Avg INSERT Latency
No Trigger	6.3 ms
No-op	6.5 ms
MC-1op	11.9 ms
MC-100op	30.6 ms
DB-1op	6.5 ms

Table 4.2: Trigger Overhead on INSERT

The results are shown in Table 4.2. From the results, we can see that the overhead of launching a trigger is minimal, as is the overhead of a local SQL query from within the trigger. However, opening a remote memcached connection doubles the INSERT latency. Each memcached operation done from within the trigger takes about 0.2 ms, which is the same amount of time taken by a normal client to perform a memcached operation (as seen from our previous benchmark). From this we conclude that even though launching a trigger does not have significant overhead, doing useful work from the triggers such as accessing memcached does have more than 100% overhead. We revisit this in the next section where we describe the increase in latency of write operations at the cost of improving read operations.

4.3.3 Social Networking Workload

In this section, we describe our performance experiments with the Pinax applications, present their results and discuss our conclusions from those 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, 100, 20% write pages, 2.0, and 512 MB respectively. In each experiment, we measure the throughput and latency values, 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.

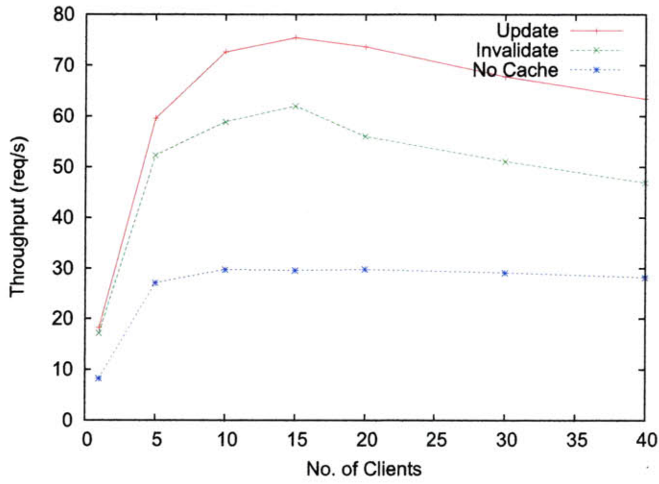
Experiment 1: Throughput and Latency Measurement

In this experiment, we compare the three caching strategies—No-cache, Invalidate and Update (as mentioned in Section 4.3)—in terms of the maximum load they can support. We measure the throughput and latency of these strategies under increasing load (i.e., increasing number of parallel clients).

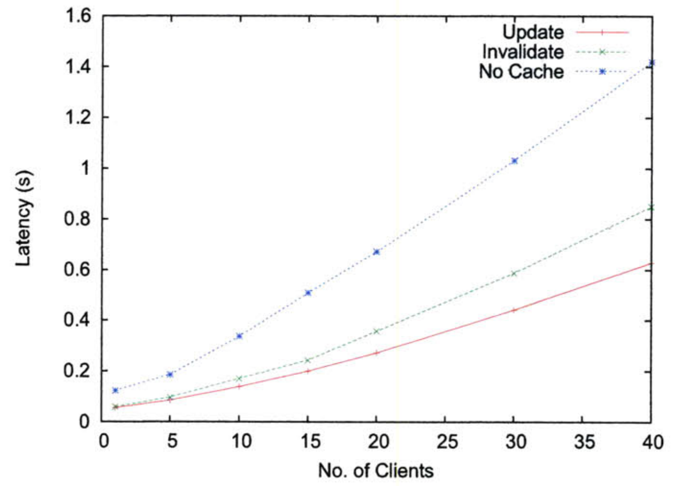
We ran the experiment for 1, 5, 10, 15, 20, 30 and 40 parallel clients. The results are shown in Figure 4-4. Figures 4-4a and 4-4b show the page load throughput and page load latency, respectively, as the number of clients increases. Further, Figures 4-4c and 4-4d depict the corresponding query throughput and latency.

From Figure 4-4a we see that the maximum throughput of the Update system occurs at 15 clients, the value of max throughput being 75 requests per second. Invalidate system also achieves its maximum throughput of 62 requests per second at 15 clients. For the No Cache system, however, the maximum throughput is only 30 requests per second, achieved at 10 clients. Thus, at its peak, the CacheGenie systems provide a 2-2.5 times throughput improvement over the No cache 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 since we have not implemented caching abstractions for all types of queries, there are some queries in the system which are never cached. It is predominantly because of these queries that the throughput advantage we get from our system is only a factor of 2-2.5, which is much less than the throughput benefit memcached can have over the database as we saw in our microbenchmarks.

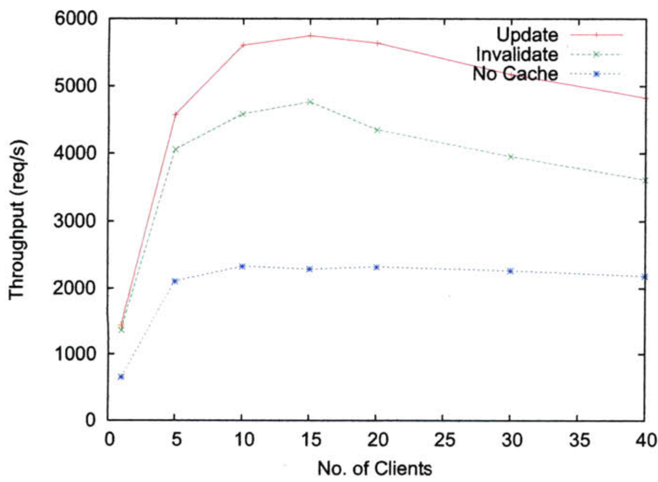
In all three systems, the database is the bottleneck and limits the overall throughput of the system. In the No Cache case, the CPU of the database machine is saturated, while in the two cached cases, the disk I/O is the bottleneck. This is easily explained as the queries hitting the database in No Cache are repeated and hence a lot of time in the database is spent in evaluating the query results for the data already in memory. On the other hand, for the cached cases, bulk of the queries are either non-repeated (since the system caches most of the repeated queries), or writes.



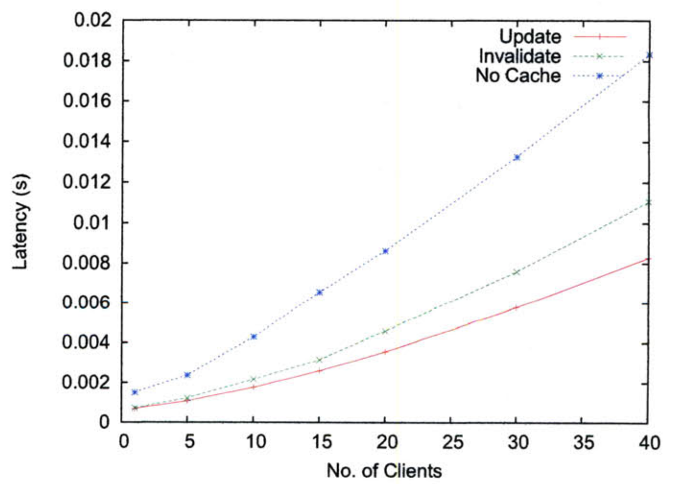
(a) Page Load Throughput



(b) Page Load Latency



(c) Query Throughput



(d) Query Latency

Figure 4-4: Experiment 1—Performance against Varying Clients

Moreover, writes in the cache system are slower because of the extra overhead of triggers. (We'll show that this is the case in a later experiment where the throughput increases dramatically for the cached systems when there are no writes.) Hence the database becomes bottle necked on the disk. One result of this difference is that the throughput starts dropping beyond a certain point for the cached cases since a disk-bounded system leads to thrashing beyond peak throughput. This is not the case for No Cache case, where the throughput stays at the peak value since CPU resources scale better under heavy load.

Another important point to note from this graph is that the throughput is more in case of incremental updates as opposed to invalidate. The trade-off between these two cases is that updating leads to slower writes (because triggers have to do more computation) but faster reads (because there are more cache hits). Figure 4-4a illustrates that the overhead of recomputing from database is more than the overhead of updating the relevant cached entries.

Figure 4-4b shows how the latency of page loads increases as load on the system increases. We see that the Update case has the least latency of 0.2 seconds per page load at peak throughput (15 clients), followed by Invalidate with a latency of 0.24 seconds and No Cache with a latency of 0.5 seconds. 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.

The query throughput and latency graphs (Figure 4-4c and 4-4d) mirror the corresponding page load graphs. Also, from this graph we can calculate the average number of queries per page (for this workload) to be 76. Table 4.3 and 4.4 lists the average latency for various types of page loads and queries, respectively, for the three systems in this experiment. One can note from these numbers that the average latency of a INSERT/DELETE/UPDATE operation in the cached cases is 5-10 times more than that in the no cache case. This is the cost that the write operations have to pay to improve the overall performance of the system.

For all the following experiments, unless otherwise specified, we run 15 parallel clients since that achieves the maximum throughput for all the systems.

Page Type	Update	Invalidate	No Cache
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 4.3: Average Latency by Page Type in Experiment 1 (with 15 clients)

Query Type	Update	Invalidate	No Cache
Query	2.6 ms	3.1 ms	6.5 ms
Query in Database	21.4 ms	25.5 ms	6.5 ms
Query in Database:SELECT	9.5 ms	14.6 ms	6.4 ms
Query in Database:INSERT	114.2 ms	122.2 ms	12.0 ms
Query in Database:DELETE	11.2 ms	15.4 ms	3.2 ms
Query in Database:UPDATE	20.8 ms	16.1 ms	2.1 ms
Query in Cache	0.5 ms	0.5 ms	-
Query in Cache:GET	0.4 ms	0.4 ms	-
Query in Cache:ADD	0.4 ms	0.6 ms	-

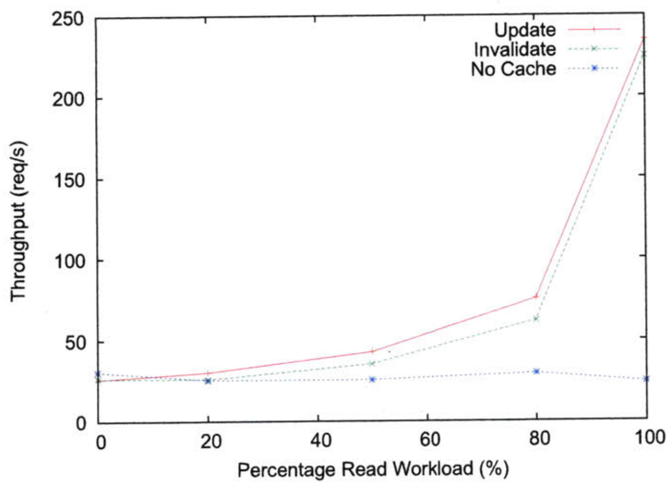
Table 4.4: Average Latency by Query Type in Experiment 1 (with 15 clients)

Experiment 2: Effect of Varying Workload

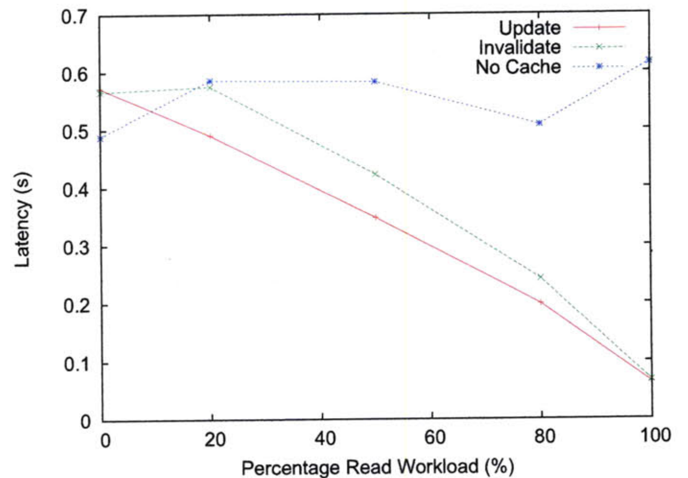
In this experiment, we vary the ratio of read pages to write pages in the workload, and measure how it affects the performance in the three caching strategies. The default workload ratio as mentioned before is 80% read pages and 20% write pages. We perform experiments with the following additional ratios of reads to writes.

- 0% reads, 100% writes (0:0:50:50)
- 20% reads, 80% writes (10:10:40:40)
- 50% reads, 50% writes (25:25:25:25)
- 100% reads, 0% writes (50:50:0:0)

The value in parentheses is the exact breakup between different types of read and write pages, i.e. ⟨LookupBM : LookupFBM : CreateBM : AcceptFR⟩. The results of these experiments are shown in Figure 4-5a and 4-5b respectively.



(a) Page Load Throughput



(b) Page Load Latency

Figure 4-5: Experiment 2—Page Loads Performance with Varying Workload

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 as we saw from Table 4.4, database writes are slower in the cached system due to the overhead of triggers. For 0% read pages, the ratio of database write queries to read queries is high, causing the overall performance of cached cases to be worse.

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 times the throughput of No Cache case. This is because in absence of any writes, database reads become much faster. Again, the throughput bottleneck here comes from the queries which we do not cache at all. Also note that the workload variation does not significantly affect the No Cache case since it is already CPU bound because of reads, which hit the database buffer pool. But it affects the cached cases since they are disk-bound, and disk performance changes as the number of writes

goes down.

We see that the gap in throughput between Update and Invalidate is zero at 0% reads and increases as the number of reads increases. This is because as the read workload increases, the advantage of better cache hit ratio overcomes the disadvantage of slower triggers in Update. However, the gap reduces back 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 :

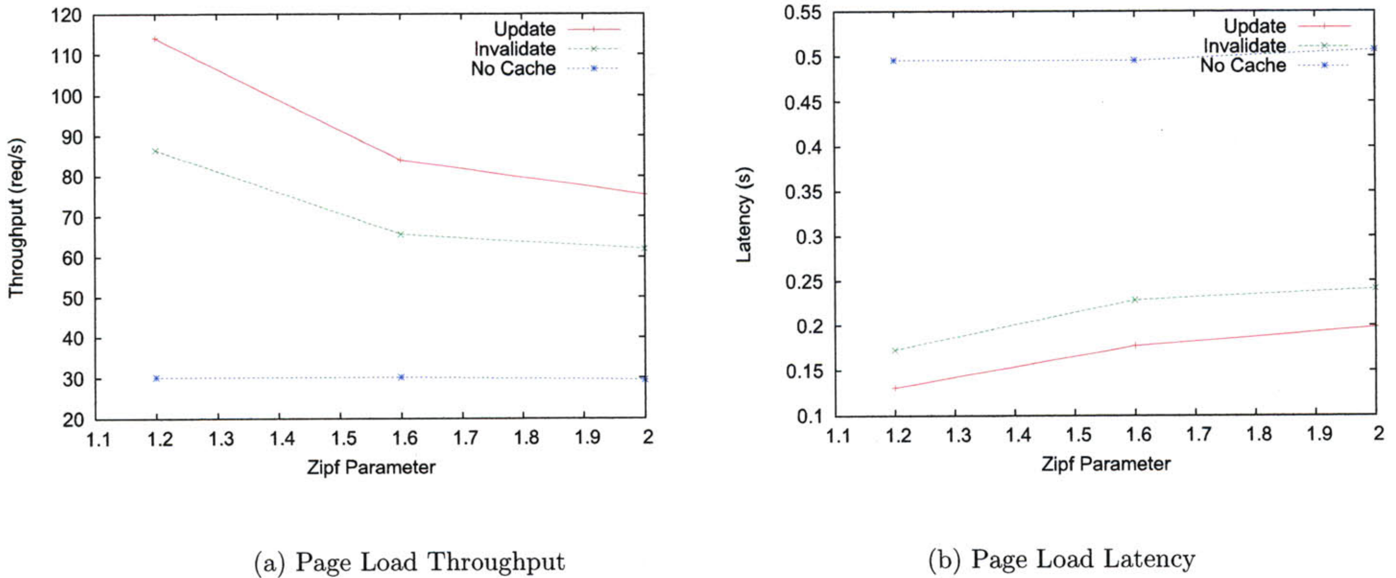
$$p(x) = \frac{x^{-a}}{\zeta(a)} \tag{4.1}$$

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 parameter a means more users login frequently.

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 over a period of time the 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. So the database performance benefits from users who login 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. The default value of the parameter

is 2.0 (which is used in the previous experiments.) We run the experiments with two other values of a: 1.2 and 1.6. Figure 4-6 shows the results from this experiment.



(a) Page Load Throughput

(b) Page Load Latency

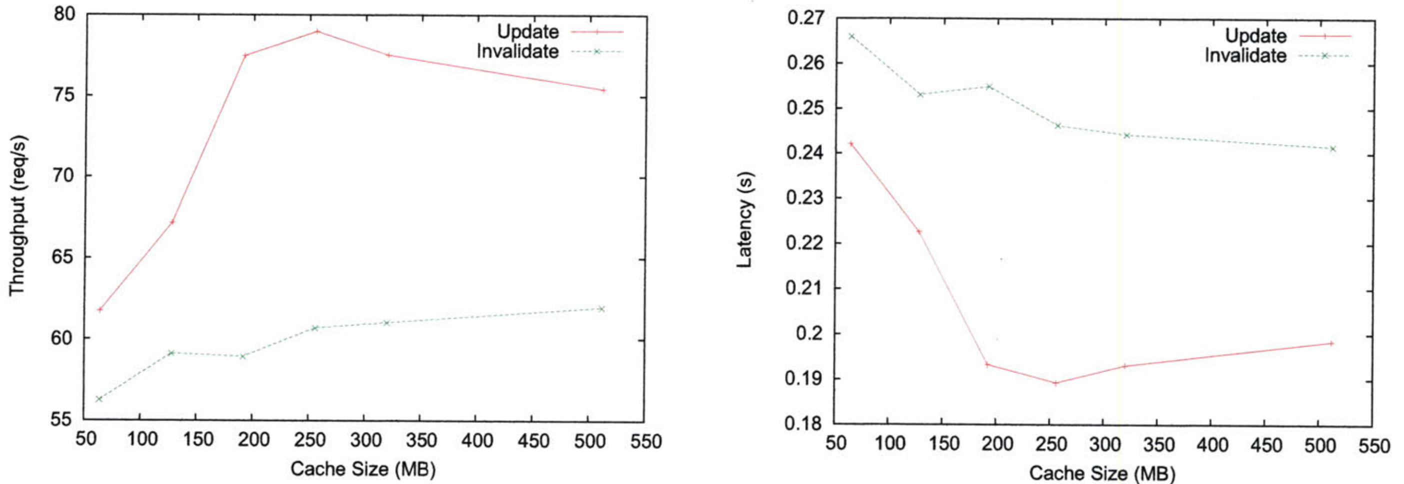
Figure 4-6: Experiment 3—Page Loads Performance with Varying User Distribution

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 No Cache 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 No Cache case is already CPU-bounded, 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 cannot have a cache that is big enough to hold everything that can ever

be cached. The purpose of this experiment is to analyze the effect of evictions due to a smaller cache on the system performance (in the cached cases).



(a) Page Load Throughput

(b) Page Load Latency

Figure 4-7: Experiment 4—Page Loads Performance with Varying Cache Size

We run our experiment with the maximum cache size set to 64 MB, 128 MB, 192 MB, 256 MB and 320 MB. The results from the experiments are in Figure 4-7. From the first graph, we can see that the throughput in the Update case plateaus at about 192 MB, whereas for the invalidate case it does so at about 128 MB. This is because the Update case never invalidates the data in cache, and so needs more space. Although we need more space for achieving maximum throughput, we can see that even with just 64MB of cache space, the Update case throughput is about 61 requests/s and Invalidate is 56 requests/s, which is still twice than the No Cache case. From this we conclude that even in the presence of evictions, caching performs significantly better than No Cache.

The conclusions of this experiment are related to Experiment 3. A lower zipf parameter implies there are more number of frequent users, which in turn means we need less space in the cache to satisfy more number of requests. So, 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 did an experiment by putting memcached on same machine as the database, so that for the cached cases, the database has lesser memory. The throughput of the Update case in this experiment was 64 requests/s (down from 75), and the throughput of the Invalidate case was 48 requests/s (down from 62). This performance is still better than the No Cache case whose throughput was 30 requests/s. These results validate our hypothesis.

4.4 Conclusions

We ported a subset of applications from the Pinax project to use the CacheGenie prototype implemented in Django. This required about 14 lines of code to be added to define the cached objects. Also, our system generates about 1720 lines of trigger code, and we argue that without an automatic cache management scheme, the programmer will have to write about the same lines of code to manage the cache for these applications. From this experience, we conclude that it requires minimal effort on the part of programmer to use CacheGenie's abstractions to get automatic cache management.

From our microbenchmarks, we found 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 an overhead ranging from 3% to 400% depending on the amount and kind of work being done inside the trigger.

With our performance evaluation on Pinax, we tried to answer a few questions (Section 4.3). We will summarize the answers to these questions here:

1. We get a throughput improvement by a factor of 2-2.5 by using CacheGenie, as opposed to no caching. Database performance is the limiting factor in both the cached scenarios.

2. Updating cached data in-place rather than invalidating it provides a throughput

benefit between 0-25%, depending on the workload and user distribution.

3. The advantage of using caching increases as the percentage of reads in the workload increases. A workload with 100% reads gives a throughput improvement by a factor of 8 over the No Cache case. The advantage of update over invalidate is maximum in a medium workload and decreases as the workload becomes read-only or write-mostly.

4. As there are more repeated users in the workload, throughput with caching improves by about 1.5 times, whereas with no caching, it does not change.

5. Synchronous update of cache on writes to database increases the latency of write operations by 5-10 times. This is the penalty for improving overall throughput of the system using automatic cache management.

6. The size of cache needed for good performance is much lesser than the total size of the database. Also, using spare memory as a cache gives better performance than using that memory with the database. The cache size needed also depends on frequency of users and the distribution of workload.

Chapter 5

Related Work

There has been a lot of work done in improving caching to enable scalable storage for web applications. Large scale websites such as Facebook typically employ in-house solutions to solve their scaling problems. There has also been prior academic work in this area on different caching strategies. This chapter discusses this related research and places it in the context of our work.

As discussed in Chapter 2, caching strategies can be divided into two main categories: *application caching* and *database caching*. In the context of database caching, we mentioned three prior systems, namely DBProxy, DBCache and GlobeCBC, which we describe in detail below.

DBProxy [22] is an edge-of-network cache that transparently adapts to changes in workloads. The cache in this system is a stand-alone database engine that maintains partial materialized views of previous query results. Application access patterns determine which materialized views are cached. DBProxy ensures data consistency by subscribing to a stream of updates propagated by the database server. DBCache [24] is a similar system which caches an entire table or a subset of the tables from the backend database server. As in DBProxy, the results of queries are computed at run-time from this cached data, and if not available, the queries are sent to the backend database. Changes at the backend database are propagated to the cache using a data replication tool. An important characteristic of their system is that they maintain dynamic cache tables which are populated at run-time based on the queries issued

by the application, and adapt to the workload. Both these systems fall under the category of content-aware caching. CacheGenie differs from both these systems in that we cache the actual query result and so there is no computation involved at query time to determine the result. We have already discussed the cache consistency guarantees provided by these systems in Section 2.2.3, and how they differ from our model.

GlobeCBC [31] is a content-blind query caching middleware for web applications. Content-blind caching systems store the query results in the cache as-is, and return the result when the same query is issued by the application again. As compared to content-aware caching systems, this approach avoids the overhead of computing query results from cached data. Our work is similar to GlobeCBC in that we also cache query results. However, in GlobeCBC, to ensure cache consistency, the developer has to mark conflicting read and write templates manually. The system then uses this information to determine when to invalidate cached content. CacheGenie, on the other hand, does this automatically based on the cached object definition. Other differences in cache consistency guarantees have already been discussed in Section 2.2.3.

There are other systems which try to provide automatic cache management. Challenger [30] proposed using a dependency graph between cached objects and underlying data to update or invalidate relevant HTML pages or fragments in the cache. Whenever underlying data changes, the affected objects in the cache are either invalidated or regenerated. However, the query workload that they consider was mostly reads, with very few writes; for example editing/adding news articles, which is done by very few people (editors). Our system is designed to deal with many more writes (for example, many users of a social network editing their personal data or adding friend connections to other users), although we assume that overall there are more reads than writes.

Ferdinand [25] is a proxy-based distributed database query caching system. Each proxy's cache is a simple disk-based map between each database query and a materialized view of that query's result. It uses a publish-subscribe model to achieve consistency in a scalable distributed manner. This differs from our work in that they

cache direct database query results transparently, whereas CacheGenie lets the developer specify what data should be cached and also provides options for varying update strategies. However, it would be instructive to explore a pub-sub model in our setting to achieve scalability in propagating updates to the cache. Also, on updates to the underlying data, Ferdinand always invalidates the affected cached query results. We however allow updates and demonstrate the advantage of incrementally updating the cached data as opposed to simply invalidating it.

There has been a lot of work exploring materialized views in databases and algorithms to incrementally update them. Materialized views are also useful in precomputing and thus providing fast access to complex query results. The problem of incremental view maintenance is similar to the problem of maintaining query results in the cache up-to-date. [26] gives an overview of the techniques proposed for view maintenance. The ideas from these techniques can be applied to incrementally updating cached objects that are more generic than the ones we explored in this work.

TxCache [29] 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 snapshot of the database. TxCache lets programmers designate specific functions as cacheable; it automatically caches their results, and invalidates the cached data as the underlying database changes. TxCache relaxes the freshness guarantee of cached data slightly to provide transactional consistency. Our work, however, relaxes transactional guarantees in favor of enabling the application to access fresh data in the cache. Moreover, we also provide mechanisms to update the cached object instead of simply discarding them when underlying database changes.

Facebook [7] is a large scale social networking website with about 500 million users. Memcached is a central component of data storage and serving infrastructure at Facebook. As the persistent data layer, Facebook has hundreds of MySQL servers. These servers execute queries locally and all joins are done by the application. Facebook then stores results of these queries in the memcached layer, itself composed of many servers. Till recently, developers had to manually store the data in cache, and manually invalidate it on writes. As we discussed earlier this approach is cumber-

some and error-prone. Recently, Facebook started building a new way to organize their data, called Facebook objects and associations, because most of their data fits into this form. Further, they are developing a system called TAO on top of memcached. TAO is API-aware (i.e. it understands Facebook objects and associations) and supports write-through on updates. This approach has similar vision as ours, in that it relieves the programmer of the burden of managing the cache, and also updates the cached data instead of invalidating. Further, it allows programmers to think in terms of high-level abstractions rather than in terms of SQL, which is what we try to achieve using caching abstractions. An important difference however is that they support write-through cache, whereas we propagate the updates through the database. We are not aware of the consistency guarantees provided by their system.

We briefly stated the various granularities of caching available in Django. Here we will describe them in detail. Site-level caching caches every page that doesn't have GET or POST parameters, and one can choose to cache only non user-specific pages. The programmer needs to specify the number of seconds after which each page should expire. This option works for mostly static websites, but not so well for dynamic applications. View-level caching lets developers cache the output of individual views (that is, the entire page produced by the view). Again, the developer can specify expiry interval for the view. Template fragment caching lets the developer cache fragments of a page from the template. If the fragment depends on a dynamic data item, Django lets developers cache multiple copies of the fragment for different values of that item. Fragment level caches also require developers to specify expiry time for the fragment. Key-Value caching allows developers to cache arbitrary data indexed using keys defined by the developers themselves. It exposes a get-set API, and since the keys are set by the developers, they can delete them as well. This is unlike the previous three options where the key is unknown to the developer and they have to rely on expiry intervals. This is the most flexible option available to Django programmers to cache their query results and other computed data.

Ruby on Rails [14] is another popular web application framework used by developers today. Cache Money [1] is a library which enables write-through and read-through

caching for Ruby on Rails. It is also based on memcached. The library is responsible for populating the cache in case of a cache-miss, and for keeping the cached objects up-to-date in case of writes. This work is similar to ours in the following ways: (i) the developer needs to specify what data they want to cache using indices (similar to specifying cached objects in CacheGenie), (ii) the system takes care of transparently updating the cached data, and (iii) is able to cache only specific classes of queries. They also provide a support for transactions over memcached by enhancing the client library. The writes to the cache are buffered in the client until the transaction is committed. Reads within the transaction are read from the buffer. The client library acquires locks while performing writes; however, reads do not take locks, and hence it is possible to peek inside a partially committed transaction. Rollback is easy in this scenario since the buffered writes can simply be discarded. We currently do not provide support for rollback or locking while writing. Cache Money does not support any kinds of joins, however, whereas we support joins (LinkQuery Cache Class). Also, we provide more flexibility to the programmer to decide whether they want to invalidate the cached objects or update them. Cache management in Cache Money is all done in the application layer, whereas we propagate the updates from the database.

Chapter 6

Future Work

Our work can be extended in several directions; in this chapter we discuss a few of them and outline our plans for future work.

Transactional Consistency

The most important aspect of CacheGenie that we would like to improve is the consistency guarantees it offers over data in the cache. As described in Section 2.2.3, we do not provide transactional guarantees over the cached data. We would like to extend our model to provide some version of transactions over cached data, so that one transaction does not see another transaction's dirty data. At the same time, a transaction should be able to see its own uncommitted changes, and once the transaction commits, its changes need to be visible to all other transactions. There are various challenges in providing this capability, with the major constraint being performance. It is easy to see that memcached performance will drop if one transaction performing a read on a key blocks, waiting for another transaction that wrote to the key to finish.

Here we discuss one possible design to implement transactions in CacheGenie. This design involves building a wrapper around memcached that implements transactions. All keys in the cache are marked to be in one of two states, *uncommitted* or *committed*. When a transaction begins, the application and database decide on a

common transaction id, say *tid*. Whenever the database issues any updates/invalidations to memcached as a part of a transaction, it sends its tid along. The memcached wrapper first acquires a lock on the key, checks whether the key is in the committed state currently, and if yes, updates it with this new value, and changes its state to uncommitted. Further, it keeps a list of all keys modified in each ongoing transaction. At commit time, the application sends a commit message to the wrapper along with the tid. The wrapper then atomically commits to the cache all the keys in that tid's list of modified keys. Similarly, if the application issues an abort, the keys modified in the transaction are simply invalidated.

Any read query to the cache from the database or application is also accompanied by tid of the transaction in which the read is being performed. If the key being asked for has been modified within the same transaction, the modified value is returned. Otherwise, the wrapper returns a special value indicating that the key is currently being modified, and that the query should be issued again in sometime. We take this approach to avoid blocking within memcached and to ensure maximum throughput. Further, to prevent long running transactions from indefinitely blocking other transactions, the wrapper times-out pending transactions after some time and invalidates the keys modified in them. Similar to read queries, updates from other transactions also have to try again if a current transaction is modifying that key.

An important point to note is that we need to implement some kind of locking in the wrapper to atomically perform a set of operations; memcached does not currently provide this functionality. Also, any approach which tries to ensure full transactional consistency can cause a performance slowdown. However, we expect that web applications typically would not have very long running transactions. Specifically, in Django, the common practice is to commit immediately after every write operation. Small transactions means less contention and better performance. Moreover, since we can simply invalidate the data in cache, it is easy to deal with aborts and deadlocks. We plan to implement this approach and measure the performance of the system with transactional consistency.

Multiple Database Support

Clearly, any large scale web application will not use a single database server for its entire persistent state. Currently, web applications either use commercially available distributed databases as their persistent backend, or distribute their queries in the application while using multiple independent single-server databases in the backend. However, in both cases, caching still plays an important role since computing queries over multiple databases is slow. In our current prototype we assume there is a single database backend. We would like to extend our prototype to work with a system having multiple database backends. We believe it should be relatively straightforward to extend our model to support either types of distributed databases.

Supporting other workloads

Another future direction is to analyze other types of web application workloads in addition to the ones we studied in this work, and identify other commonly occurring query patterns. Once we identify more patterns, we can implement Cache Classes for them and thus enable automatic cache management for more queries. Also, we would like to port other kinds of applications to CacheGenie and evaluate its usability and performance with those applications.

From an evaluation point of view, it would be interesting to obtain actual workload traces of real applications, and evaluate our system's performance using those traces. While we tried to model real user behavior in our evaluation, it is far from ideal and it would be instructive to experiment with real data. Another evaluation we plan to do is to implement application-level manual invalidation in Pinax applications (just like developers today would implement), and compare the performance of that implementation with CacheGenie.

There are certain parts of our design which we have not yet implemented in our prototype. Two of these are prefetching cached objects, and providing variable cache update strategies based on popularity of users and associated data (Section 2.1.2).

Also, to use a LinkQuery class, currently the developer has to manually use the cached object instead of the original code. For FeatureQuery and CountQuery however, the system automatically fetches cached objects when the original code is used. We also haven't implemented the Cache Class for TopKQuery pattern. We plan to implement these missing pieces as part of future work.

Performance Optimizations

The performance of write queries can be improved by optimizing the database triggers. Currently, each cached object has three triggers (corresponding to `INSERT`, `DELETE` and `UPDATE`) on each table involved in computing the query result. This means most tables have multiple triggers if there are multiple cached objects that depend on them. Since there is significant overhead in launching a trigger and opening a connection to memcached, combining these multiple triggers into one trigger each on `INSERT`,`DELETE` and `UPDATE` can lead to significant performance improvements. We believe this optimization is straightforward to implement and plan to incorporate this in our system.

A second idea for trigger optimization is to offload the trigger execution to a long running external process on the same or a different server. We expect this will reduce the load on the database server and let writes complete faster. However, getting good performance with this approach can affect consistency. For example, if the trigger simply sends the data required to update the cache to the external process and returns, it is possible that the database write can complete without the relevant cached objects being updated. This can lead to the transaction not seeing its own updates in the cache. But, to avoid this, if the trigger waits for the external process to acknowledge that the execution has finished, it might not afford a performance benefit. We have done some preliminary implementation of this offloading approach, and observed significant performance benefits if we choose the first option, i.e. not wait for the execution to finish. We would like to explore this area more fully and quantify the benefits we can get from it.

Chapter 7

Conclusion

This thesis presents CacheGenie, a system which provides high-level caching abstractions for modern web applications. The main goal of the work in this thesis are to provide automatic cache management for web applications without requiring any changes to the underlying database or cache. A secondary goal is to improve cache performance by providing mechanisms to automatically update cached data as the underlying database changes, instead of the currently prevalent strategy of invalidating it.

The key ideas that help us in achieving these goals are: (i) extracting common query patterns from ORM-based web applications and providing caching abstractions for them, (ii) exploiting database triggers to automatically keep the cached results of these common queries consistent with the database, and (iii) generating automatic triggers using the caching abstractions paradigm to either update the cached data or invalidate it whenever the underlying database is modified.

Our current prototype implements these abstractions by modifying Django, a popular web application framework, and works with unmodified PostgreSQL and memcached. To evaluate CacheGenie, we ported a subset of applications from the Pinax project (which is based on Django) to use our prototype. We had to add only about 20 lines of code for this. From this experience, we conclude that it requires minimal effort on the part of programmer to use CacheGenie to get automatic cache management.

From our microbenchmarks, we found that using memcached instead of a database can improve throughput by a factor of 10 to 150. Further, a database trigger can induce an overhead ranging from 3% to 400% depending on the amount and kind of work being done inside the trigger. We did a series of performance experiments on the modified Pinax applications, and CacheGenie improved the throughput of the applications by a factor of 2-2.5, depending on the cache update strategy. Updating cached data in place is 25% faster than invalidating it. We also measured the variation in performance as (i) the workload changes from more reads to more writes, (ii) as the user distribution changes from more distinct to more repeated and (iii) as the size of the available cache varies. In each case we determined which configurations are best suited to get maximum performance from our system.

There are several directions in which this work can extended and we plan to work on some of them to make CacheGenie more useful to application developers. We hope that ideas from this work will be used by web application framework designers to develop useful and efficient caching frameworks for web applications.

Appendix A

Pinax Database Schema

```
TABLE 1: auth_user (id              integer
                    username        character varying(30)
                    first_name      character varying(30)
                    last_name       character varying(30)
                    email           character varying(75)
                    password        character varying(128)
                    is_staff        boolean
                    is_active       boolean
                    is_superuser    boolean
                    last_login      timestamp with time zone
                    date_joined     timestamp with time zone );

Indexes:
    "auth_user_pkey" PRIMARY KEY, btree (id)
    "auth_user_username_key" UNIQUE, btree (username)
```

```
TABLE 2: profiles_profile (
                    id          integer
                    user_id     integer
                    name        character varying(50)
                    about       text
                    location    character varying(40)
                    website     character varying(200) );

Indexes:
    "profiles_profile_pkey" PRIMARY KEY, btree (id)
    "profiles_profile_user_id_key" UNIQUE, btree (user_id)

Foreign-key constraints:
    "profiles_profile_user_id_fkey" FOREIGN KEY (user_id) REFERENCES
        auth_user(id) DEFERRABLE INITIALLY DEFERRED
```

Figure A-1: Schema of database tables in Pinax

```
TABLE 3: bookmarks_bookmark (
                    id                  integer
                    url                 character varying(200)
                    description         character varying(100)
                    note                text
                    has_favicon         boolean
                    favicon_checked     timestamp with time zone
                    adder_id            integer
                    added               timestamp with time zone );

Indexes:
    "bookmarks_bookmark_pkey" PRIMARY KEY, btree (id)
    "bookmarks_bookmark_url_key" UNIQUE, btree (url)
    "bookmarks_bookmark_adder_id" btree (adder_id)

Foreign-key constraints:
    "bookmarks_bookmark_adder_id_fkey" FOREIGN KEY (adder_id)
       REFERENCES auth_user(id) DEFERRABLE INITIALLY DEFERRED
```

```
TABLE 4: bookmarks_bookmarkinstance (
                    id              integer
                    bookmark_id     integer
                    user_id         integer
                    saved           timestamp with time zone
                    description     character varying(100)
                    note            text
                    tags            character varying(255) );

Indexes:
    "bookmarks_bookmarkinstance_pkey" PRIMARY KEY, btree (id)
    "bookmarks_bookmarkinstance_bookmark_id" btree (bookmark_id)
       CLUSTER
    "bookmarks_bookmarkinstance_user_id" btree (user_id)

Foreign-key constraints:
    "bookmarks_bookmarkinstance_bookmark_id_fkey" FOREIGN KEY (
       bookmark_id) REFERENCES bookmarks_bookmark(id) DEFERRABLE
       INITIALLY DEFERRED
    "bookmarks_bookmarkinstance_user_id_fkey" FOREIGN KEY (user_id)
       REFERENCES auth_user(id) DEFERRABLE INITIALLY DEFERRED
```

Figure A-1: Schema of database tables in Pinax

```
TABLE 5: friends_friendship (
                    id              integer
                    to_user_id      integer
                    from_user_id    integer
                    added           date );

Indexes:
    "friends_friendship_pkey" PRIMARY KEY, btree (id)
    "friends_friendship_to_user_id_key" UNIQUE, btree (to_user_id,
       from_user_id)
    "friends_friendship_from_user_id" btree (from_user_id)
    "friends_friendship_to_user_id" btree (to_user_id) CLUSTER

Foreign-key constraints:
    "friends_friendship_from_user_id_fkey" FOREIGN KEY (from_user_id
       ) REFERENCES auth_user(id) DEFERRABLE INITIALLY DEFERRED
    "friends_friendship_to_user_id_fkey" FOREIGN KEY (to_user_id)
       REFERENCES auth_user(id) DEFERRABLE INITIALLY DEFERRED
```

```
TABLE 6: friends_friendshipinvitation (
                    id              integer
                    from_user_id    integer
                    to_user_id      integer
                    message         text
                    sent            date
                    status          character varying(1) );

Indexes:
    "friends_friendshipinvitation_pkey" PRIMARY KEY, btree (id)
    "friends_friendshipinvitation_from_user_id" btree (from_user_id)
    "friends_friendshipinvitation_to_user_id" btree (to_user_id)
       CLUSTER

Foreign-key constraints:
    "friends_friendshipinvitation_from_user_id_fkey" FOREIGN KEY (
       from_user_id) REFERENCES auth_user(id) DEFERRABLE INITIALLY
       DEFERRED
    "friends_friendshipinvitation_to_user_id_fkey" FOREIGN KEY (
       to_user_id) REFERENCES auth_user(id) DEFERRABLE INITIALLY
       DEFERRED
```

Figure A-1: Schema of database tables in Pinax

Bibliography

[1] Cache Money. http://github.com/nkallen/cache-money.

[2] Car Posse. http://carposse.com/.

[3] Cloud27. http://cloud27.com/.

[4] Craigslist. http://www.craigslist.org/.

[5] Digg. http://digg.com.

[6] Django. http://www.djangoproject.com/.

[7] Facebook. http://www.facebook.com.

[8] Flickr. http://www.flickr.com/.

[9] Live Journal. http://www.livejournal.com/.

[10] Memcached. http://memcached.org.

[11] mftransparency.org. http://mftransparency.org/.

[12] Orkut. http://www.orkut.com/.

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

[14] Ruby on Rails. http://rubyonrails.org/.

[15] Sequence Medical. http://sequencemed.com/.

[16] TuttiVisti. http://tuttivisti.com/.

[17] Twitter. http://twitter.com/.

[18] we20. http://we20.org/.

[19] Wikipedia. http://www.wikipedia.org/.

[20] YouTube. http://www.youtube.com/.

[21] Zend Framework. http://framework.zend.com/.

[22] Khalil Amiri, Sanghyun Park, and Renu Tewari. DBProxy: A dynamic data cache for Web applications. In *In Proc. ICDE*, pages 821–831, 2003.

[23] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing User Behavior in Online Social Networks. In *IMC '09: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pages 49–62, New York, NY, USA, 2009. ACM.

[24] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Adaptive Database Caching with DBCache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.

[25] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable Query Result Caching for Web Applications. *Proc. VLDB Endow.*, 1(1):550–561, 2008.

[26] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18:3–18, 1995.

[27] Marcelo Maia, Jussara Almeida, and Virgílio Almeida. Identifying User Behavior in Online Social Networks. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 1–6, New York, NY, USA, 2008. ACM.

[28] Atif Nazir, Saqib Raza, and Chen-Nee Chuah. Unveiling Facebook: A Measurement Study of Social Network Based Applications. In *IMC '08: Proceedings*

of the 8th ACM SIGCOMM conference on Internet Measurement, pages 43–56, New York, NY, USA, 2008. ACM.

[29] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI '10: 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[30] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for Web applications. *IEEE Internet Computing*, 11(1):60–66, 2007.

[31] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo 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`.