Application-Level Caching with Automatic Write-Through Invalidation

Anders Emil Nielsen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Caching is a popular solution for improving the performance and scalability of web applications. When data have been cached the web application does not have to recompute the data until it has been invalidated. Unfortunately, these caching systems presents two major challenges. First, they leave the responsibility of locating the cached values, invalidating correctly, and keep the values up to date, which are tasks that often leads to additional application complexity and programming errors. Secondly, in highly dynamic web applications, where the cached values are invalidated frequently, the cache hit rate can become low such that more users must wait for the cached values to be recomputed. This can be critical for the user experience in cases where the computations are slow.

This thesis address these challenges by introducing a new cache called Smache. Smache uses a programming model that lets the programmer mark a function to be cached by declaring the dependencies to the underlying data. The cached function is then automatically invalidated and afterwards updated when the underlying data changes. Smache uses timestamp invalidation to be able to update all cached values concurrently and thereby allowing to improve the update throughput by adding more processes to update cached values.

This was verified by our experiments that showed Smache is able to improve the throughput of cache updates by adding additional processes. Furthermore they showed that Smache only introduces a constant performance overhead of 2 ms to application operations updating data in the primary storage.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with the design and implementation of a caching system that is optimized to cache long running computations by always serving the results fast and keeping the cached content up to date.

The thesis was made in collaboration with Peergrade IVS and supervised by Nicola Dragoni and David Kofoed Wind.

The thesis consists of the following learning objectives:

- Understand and describe caching
- Compare and evaluate different caching systems
- Design a push-based caching system
- Efficiently implement a push-based caching system
- Evaluate the efficiency of the caching system

Lyngby, 04-July-2016

Anders Emil Nielsen

Acknowledgements

I am thankful to my supervisors Nicola Dragoni and David Kofoed Wind for the freedom to explore on my own while being guided in the right direction.

Another thank you to David Kofoed Wind and Malthe Jørgensen, the CEO and CTO of Peergrade.io, for sharing their vision and help me form a well-defined project. I would also like to say thank you for the time they put into discussing the challenges and solutions with me.

A thank you to Peergrade IVS for letting me use their office space and to the rest of the team behind Peergrade.io, Lasse Nielsen and Simon Lind for sharing their company and pizza.

I would like to say thank you to David Harboe, Mads Oehlendslaeger, and Anne-Sofie Knudsen for the moral support.

And last i would like to say thank you to Sandra Pouplier for her patience and for helping out when most needed.

Contents

Chapter 1

Introduction

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

Web applications are becoming more and more dynamic with more personalized content that often requires complex data queries or computations based on large amounts of data. These computations can become a performance bottleneck in the application, which leads to slow response times and poor user experience for the users.

The performance can often be optimized by profiling and analyzing the code behind the computation, but it is often not the easiest solution and does not guarantee a satisfactory performance in the end. Caching is a popular solution for improving the performance and scalability in these cases, because it allows for a simple, scalable and generic way of addressing bottlenecks in web applications.

Although it sounds like a silver bullet it also places a burden on the programmer, who must locate and update the cached values while preserving consistency. The following description of an outage of the whole Facebook system, indicates the difficulty of cache management:

> The intent of the automated system is to check for configuration

> values that are invalid in the cache and replace them with updated values from the persistent store. This works well for a transient problem with the cache, but it doesn't work when the persistent store is invalid.
>
> Today we made a change to the persistent copy of a configuration value that was interpreted as invalid. This meant that every single client saw the invalid value and attempted to fix it. Because the fix involves making a query to a cluster of databases, that cluster was quickly overwhelmed by hundreds of thousands of queries a second.
>
> Robert Johnson [Joh10]

This example shows how critical the caching system can be and the importance of correctness.

This thesis will address this issue by reviewing the latest caching technique proposed in research and used in practice and contribute with a design and implementation of an open-source caching system in the Python programming language.

1.1 Problem

Most existing caching approaches are based on a pull based caching strategy, where cached values are updated as they are requested by the user. The pull based caching strategy has the advantage that the system only stores cached values that are being requested, but the user requesting a cached value after it has been invalidated must wait for the computation to finish. In some cases, the time taken to compute the cached values are within a reasonable time time interval, but in cases where the computation time exceeds the attention span of the user it becomes critical for the user experience. An alternative approach is to use a push-based strategy, where the system pre-computes the cached values so they are always served to the user directly from the cache.

Furthermore existing caching solutions leaves the responsibility for the programmer to manage the cache, which often leads to programming errors and makes caching difficult for the programmer to maintain.

These problems leads to the following challenges, which will be addressed by the thesis:

Cache Management

The first challenge related to cache management is faced in any caching system, where the programmer has to assign names to cached values and keep them up to date such that the users are not presented with unexpected content.

One particular challenge within cache management is *cache invalidation* that relies on the programmer correctly identifying every underlying data that affects the given cached value. The programmer then has to declare a way for the cached value to be invalidated when any of the underlying data changes. This analysis is difficult since it require global reasoning about how the underlying data changes in the application and which computations are being cached. Furthermore if the computation behind the cached value is altered to depend on new underlying data, the cache invalidation also has to change, making the cache prone to errors if the latter is forgotten.

We discuss this more in chapter 3, 4, and 5.

Data Update Propagation

The second challenge relates to the task of efficiently pre-computing the cached values, which easily becomes an expensive task for the system. To get around this the system must use an efficient technique for identifying when cached values updated and how to update them efficiently such they are ready for the users as soon as possible without using unnecessary CPU-power. This challenge will be addressed in chapter 6.

1.2 Requirements

The final solution addressing the problems described, will be designed with the following non-functional requirements:

Software design: Must be designed to be maintainable such that the programmer that uses the caching system understands how it works from using it and has the ability to extend it.

Adaptability: Should be convenient and easy to adapt into existing systems. This involves a flexible cache that can be extended to support multiple storage systems and cache databases.

Efficiency: Should be efficient with relation to performance and computational power. More specifically existing operations of the application should not become significantly slower, and the throughput of updating cached values should

be fast while not using more computational power than necessary.

Scalability: Should be designed for scalability in the sense that the design should still be efficient for large amount of data and applications deployed to multiple web servers working concurrently.

Fault-Tolerance: Should be designed with considerations on reliability, integrity and maintainability.

1.3 Context

To motivate the development and ensure the system is designed and implemented to be used in practice, the problem and requirements are based on a running web application - the Peergrade.io-platform.

1.3.1 Peergrade.io Platform

Peergrade.io is a platform for facilitating peer-evaluation in university and high school courses. Currently the platforms serves multiple institutions and thousands of students.

Universities and single teachers are able to sign up their courses to use Peergrade.io for facilitating peer-evaluation, where the students are able to grade and/or give each other feedback on their assignments. When a teacher has registered a course to use Peergrade.io, the teacher is able to create assignments. Beside the description of what the assignment is about, the teacher also sets deadlines for handing in and for the grading, and creates structured forms for the students to fill out with grades and text feedback.

When an assignment is open, the student is able to upload their hand-in. After the deadline for handing in has been reached, the Peergrade.io platform will automatically distribute hand-ins for students to answer feedback questions and give grades. When the deadline for grading has been reached, the feedback and grades are made available to the author of the hand-in, and the authors are able to indicate the helpfulness of the feedback by giving a constructive score or mark the feedback as inappropriate. After this step, the assignment is considered completed for the student. Figure 1.1 shows a screenshot of the Peergrade.io platform from the student's perspective.

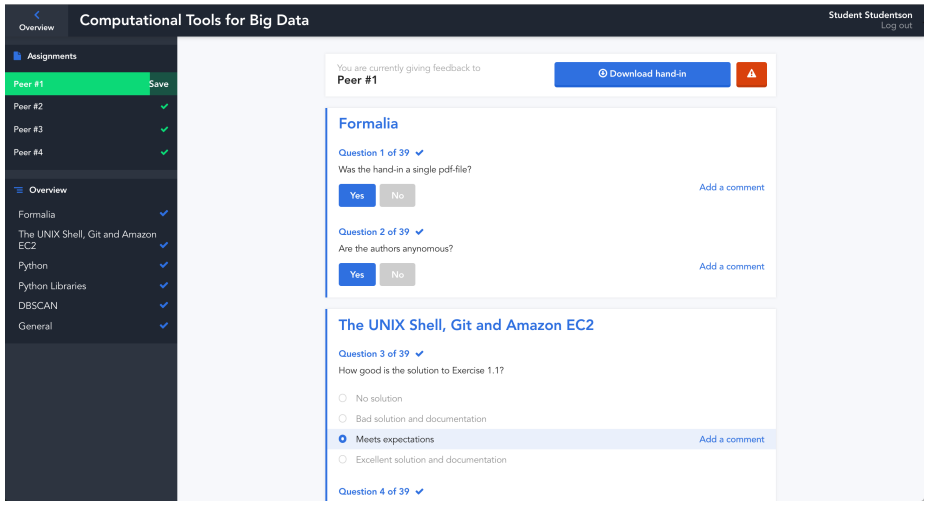


Figure 1.1: A student grades another student's hand-in.

The last part of the platform is a complete overview of the assignment presented to the teacher with hand-ins, feedback, grades, constructive scores, and statistical analyses to help the teachers assign grades. Furthermore the teacher is able to get a "performance summary" of a student to how the student performs compared to the rest of the participants. Figure 1.2 show screenshots of some of the interfaces seen by the teacher.

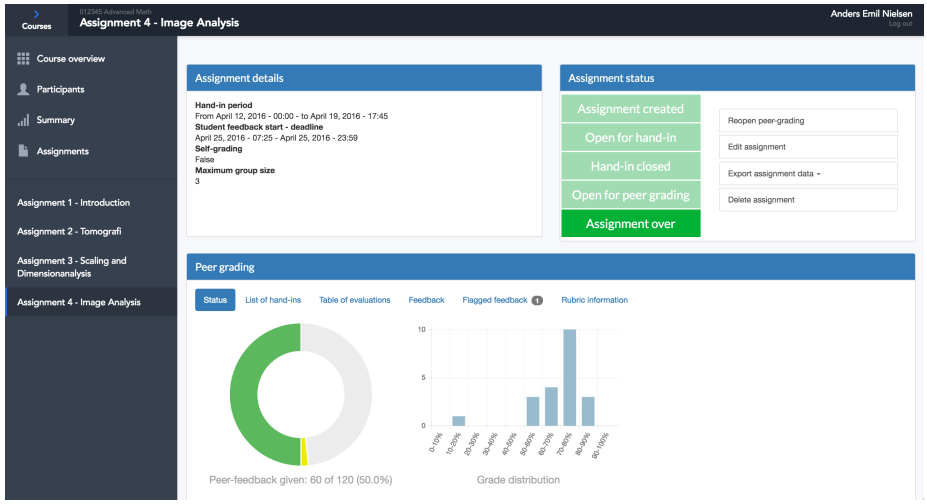


Figure 1.2: A teacher sees the overview of a given assignment.

1.3.2 Running Example

To relate the solution to a practical example, the thesis will use the running example seen in code snippet 1.1.

Code Snippet 1.1: Code with the running example written in Python

```
def course_score(course)
    participants = Database.find_all_participants_in_course(course)
    total_score = 0
    for participant in participants:
        total_score += time_consuming_participant_score(participant)
    return total_score / len(participants)

def time_consuming_participant_score(participant):
    grades = Database.find_all_grades_for_participant(participant)
    return numpy.advanced_statistical_method(participant)
```

In this example we have a function `course_score` that computes the average score in a given course. To find the data for the participants the function makes a call to the primary storage, which returns a list of participant entities. The function iterates over these entities and calculates the participant score for a given participant using the function `participant_score`. The function `participant_score` calls an external function that takes a long time to compute to illustrate a scenario related to the problem description. This example illustrates how the function `course_score` depends on the data for the course, participants, and grades given in the course. The same way we could say `participant_score` depends on the data for the given participant and the grades.

1.4 Contributions

From the goal of finding and implementing a caching system that solves the problem in the context of the requirements described above, the thesis makes the following contributions.

First, we present a set of criteria used to evaluate caching approaches. These criteria are applied to existing approaches used in practical web development and proposed in research. The existing approaches are compared through an overview, which can be used to find a caching approach suitable for a given

use case. By the end of the thesis the overview is extended with the solution proposed by this thesis, Smache.

Secondly, we present the design of Smache, a caching solution that aims to be easy to integrate into existing application and be able to cache the existing functions while invalidating and updating the cached values automatically. From this solution, the thesis makes the following technical contributions:

- a *programming model* that allows programmers to cache existing functions by declaring the dependencies to underlying data, after which the caching system automatically caches the result of the function by naming, localizing and storing the cached value for the function.
- a *description and proof* of how to use *timestamp invalidation* to invalidate and update cached values in a concurrent environment.
- an *automatic invalidation system* based on *timestamp invalidation* and a variant of the *Simple Object Dependence Graph* for representing dependencies between cached values and underlying data.
- a *data update propagation algorithm* based on *timestamp invalidation* that is able to update all cached functions concurrently without additional concurrency control mechanisms.

Thirdly, we also present an implementation of Smache using the programming language Python, MongoDB as primary storage and Redis as the cache database, which is available open-source for future contributions at:

https://github.com/anderslime/smache

We have evaluated the performance and scalability of the data update propagation algorithm and how Smache affects existing operations on the application.

1.5 Outline

The overall structure of the thesis is as following:

With the motivation, problem, and requirements described in this introductory chapter, chapter 2 will give an introduction to the basics of caching and explain the criteria used to evaluate caching approaches. Based on these criteria, existing caching approaches are compared in chapter 3. Chapter 4, 5, and 6

describes the solution suggested by this thesis, which is evaluated in chapter 7 with conclusions in chapter 8.

In more detail, the different chapters covers the following subjects:

Chapter 2 *introduces caching* by presenting the basic caching algorithm, the common architecture of caching system, the models used to describe caching approaches, and *criteria* used to evaluate caching approaches.

Chapter 3 will *describe and compare existing caching approaches* by applying the caching evaluation criteria.

Chapter 4 presents the overall solution of this thesis by describing the *programming model* used to cache functions - cachable functions.

Chapter 5 describes how the programming model is extended with *automatic invalidation* based on the *Object Dependence graph* data structure and *timestamp invalidation*.

Chapter 6 explains how the programming model is extended with write-through invalidation using a data update propagation algorithm also based on *timestamp invalidation*.

Chapter 7 describes experiments performed to evaluate performance and efficiency, followed by an evaluation of Smache based on the results of the experiments and the requirements of the system.

Chapter 8 finalize the thesis with a conclusion and suggestions for future work.

Chapter 2

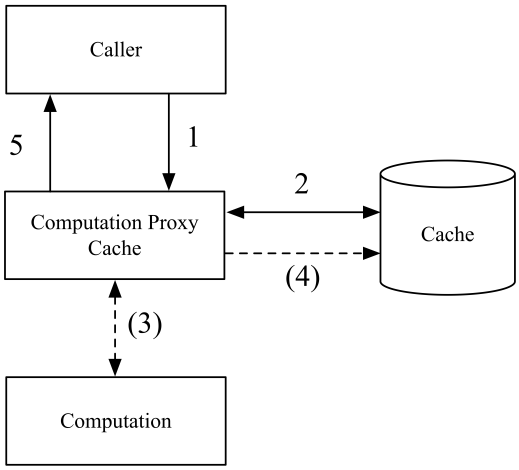
Caching Basics

In order to get a common understanding of caching and the terminology related to the topic, we will give a brief introduction to the basics of caching by describing a basic caching algorithm and the common caching architecture. The chapter also introduces the timeline model used to visualize how a caching approach works, and presents a set of criteria used to evaluate caching approaches.

2.1 Basic Caching Algorithm

In general caching is about storing the result of a computation, such that it for future requests is possible to get the result fast instead of recomputing it. This basic algorithm is described on figure 2.1.

If we look at the cached object from an abstract point of view, we can see it as a *result of a function* given certain *inputs*. Sometimes the inputs are data from a storage system, the result of an API call to some external resource, or maybe a global variable in the code. Some of these inputs are used to identify the cached function, but others change during the lifetime of the cached value. We call these changing input, *underlying data*.



1. The caller requests the value of the computation by calling the proxy
2. The proxy fetches the cached value of the computation through the cache

(3) The proxy calls the computation

(4) The new result of the computation is stored in the cache

3. The value is returned to the caller

Steps annotated with *parenthesis and dashed lines* are only executed in case of cache miss or cache failure.

Figure 2.1: The flow of basic caching

In order for the algorithm to work, we need to be sure that a cached object is uniquely identified by a given name. This presents one of the challenges of cache management, which involves naming and localizing cached values.

Another essential part of the caching algorithm is the invalidation (one of the two hard things in computer science[1]), which in the algorithm is done using a *freshness check* performed when the cached value is requested. How this *check* is performed depends on the specific invalidation technique.

[1] Not scientifically, but at least a favorite saying of Martin Fowler and quote by Phil Karlton

2.2 Caching Architecture

The common caching architecture e.g. used to support the basic algorithm for web applications described above is illustrated on figure 2.2. This architecture consists of a web application that serves HTTP-requests from the client (represented by the user) and interacts with a primary storage database to store and load data. To store and fetch the cached content we introduce a cache database.

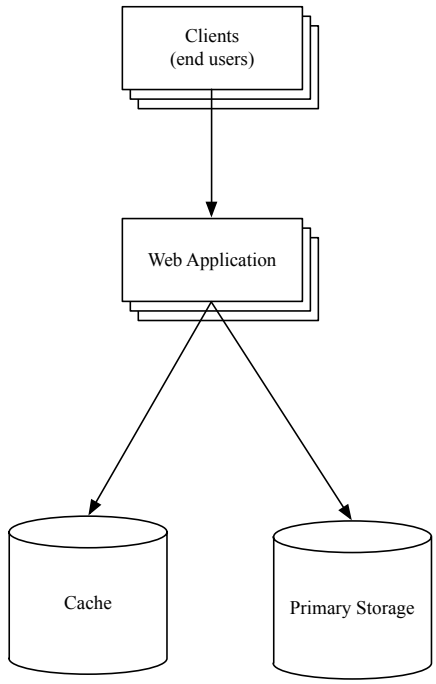


Figure 2.2: The assumed architecture of the system

In most cases the cache database is an in-memory key-value database that supports `LOOKUP` to find cached values and `STORE` to store cached values and behaviour for cleaning up old cached values. A popular choice for cache databases are technologies such as Redis [2] and Memcached [3] since they are simple distributed key-value stores that lives in memory and therefore allows for high-performance operations.

Most modern web applications needs to serve multiple users at the same time,

[2] http://redis.io/
[3] https://memcached.org/

which means the web application must run on multiple processes[4] deployed to one or more servers. We will therefore treat the web application as a concurrent environment.

2.3 Timeline Model

As with the algorithm on figure 2.1, caching can be described by a series of events. The ordering of events decides whether the cached content or a fresh computation is presented to the client. To describe the different caching techniques, we will use a timeline model with a stream of events. One timeline describes the events occurring in a single process. We can therefore assume that there exists a total ordering of events for a single timeline. To be able to describe the caching techniques explained in this thesis, we will define the following events:

- **Request** is the event occurring when a client requests a given cached object
- **Response** is when the client receives the cached value for the cached object
- **Computation Started** The computation of a new cached object is started
- **Computed and Stored** The computation is finished and the new object is stored in the cache
- **UD Updated** is when the underlying data has been updated and the cached object is considered stale
- **Invalidated** is when the system considers the cached object as invalid

Alongside these events the timeline model illustrates the interval in which the given cached object is considered valid (the cache system will serve the cached value) and when it is actually fresh (consistent with underlying data). The validity interval illustrates when the cache system will respond immediately, which means if the cached object is always considered valid then the approach always serves the latest stored value from the cache. In the time interval, where the fresh and valid interval overlaps, the approach will serve a fresh version of

[4] This is processes as an abstract term used in distributed systems. If we need to be implementation specific this could just as well be threads.

the cached value. If they always overlap then the given approach will guarantee strict freshness.

The timeline model applied to the basic caching algorithm 2.1 is illustrated on figure 2.3.

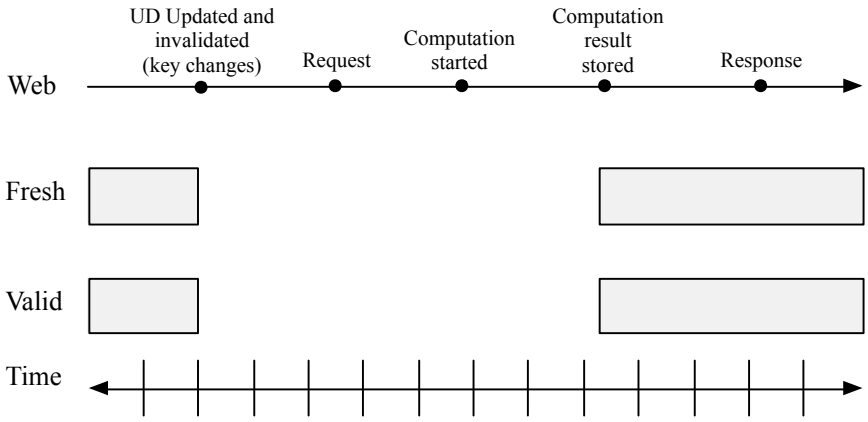


Figure 2.3: The timeline model applied to the basic caching algorithm

2.4 Evaluating Caching Approaches

To choose the correct caching technique for a given use case, we need to decide the criteria for evaluating caching approaches. The overall goal of caching in web development is to achieve a better user experience by responding to the user quickly and to save money by using less CPU power. If we assume that the cache system is able to retrieve cached objects fast, the goals can be achieved by ensuring computations are only executed with purpose and "hitting the cache" as often as possible. We can measure the frequency in which the cache is hit using the metric *cache hit rate*.

To evaluate the caching technique for a given use case, we will see the situation from the perspective of the client (i.e. a user). The client makes a request for some content that is served by the web server. This content is the results of one or more computations that can be cached individually.

We will not make any assumptions about the content send by the server, which means the content could be the result of multiple cached computations. Each

of these computations are based on some underlying data e.g. served from the primary store. In the case where some computations are based on the same data we have to keep the different results consistent. The response might be based on both cached content and content loaded directly from the primary store in which case we have to keep the data consistent across the cache database and the primary storage. This issue leads to the criteria keeping consistency between the data.

There exists multiple levels of consistency, but to keep it relevant and simple, we will evaluate the level of consistency using a binary value: either the caching technique ensures consistency with the data from the primary storage and other cached values using the same technique or else it doesn't.

Another parameter is the freshness of the content returned by the cache. It is most desirable to have content that is as fresh as possible as oppose to having stale data. We therefore use freshness as a binary parameter that we call "Strict Freshness", which evaluates whether a given cached object that is fetched is guaranteed to be based on the newest version of the underlying data.

In theory we want the most recent data, but in reality the goal is to ensure the served content makes sense for the user. If we consider how the Peergrade.io-platform works as described in section 1.3.1, we have two types of users: teachers and students. If a teacher changed the description of an assignment and a student requested and read the description 1 min after, then it would not be unexpected behaviour to show the old description from the students point of view, because the student has no knowledge about the update. On the other hand it would be unexpected for the teacher if the application showed an old description after the update, because the teacher would think that the description hasn't been updated.

While the freshness describes what is expected behaviour with relation to the content, there also exists time limits with relation to keeping the user focused on the task. Miller and Card et. al. [Mil68, CRM91] describes these limits as:

- When the response time is **0.1 second** the user feels that the system is **reacting instantaneously**.
- A response time above **1 second** will **interrupt the user's flow of thought**.
- **10 seconds** is limit related to **keeping the user's attention** on the given dialog.

In the basic caching algorithm described on figure 2.1, the cached value has to

be recomputed if the *freshness check* results in an invalidation. In this algorithm the cached value is updated during the request of the caller, which means the client have to wait for the computation to finish before receiving a response. If the time taken to compute the value exceeds the accepted response time limits described above, it becomes critical for the user experience. Based on this fact we will introduce the binary parameter of whether or not the client has to wait for the computation to finish after invalidation. The importance of this parameter depends on the cache miss rate since only requests with a cache miss are affected.

It is not possible to both serve the cached objects immediately and have strict freshness. This can be proved using the example where a cached object is invalidated just before it is requested. We can only start computing the value at the moment it has been invalidated and given that the time between the invalidation and the request is smaller than the time taken to compute the value, the cached value cannot be ready when it is requested, and we must therefore serve a stale value to have an immediate response.

In cases where we choose immediate response time and we can tolerate serving cached objects that are not strictly fresh, we might want to keep the cached objects as up to date as possible to limit the staleness. To achieve optimal freshness under the circumstances, the system must start updating the cached values as soon as they have been invalidated i.e. when the underlying data is updated. We say that caching approaches that start updating cached values as soon as underlying data changes have "Update on invalidation".

So far we've considered parameters from a user's perspective, but to evaluate the caching technique fully we also need to see it from the perspective of the programmer, because choosing an appropriate approach also involves the complexity added to the application. We want the caching system to be as transparent as possible such that it is easy to add and remove caching declarations from existing computations. Additionally we want the caching system to be robust such that when new code involving a cached computation is added, it should behave as expected without introducing errors. We cannot measure the level of robustness so we will therefore introduce the binary criteria: whether or not the caching approach involves naming, localizing and invalidating cached values. As a simplification we say that these approaches have "No Cache Management".

The last parameter we will introduce is also a requirement of the system: *adaptability*. Since there exists no metric for adaptability, we will introduce a scale of *high*, *medium* and *low*, where high means that is adaptable to most systems and low means that it is adaptable to a few systems. We will define the scale as following:

- **Low**: The technique has assumptions about the primary storage, cache database and/or application, which makes it difficult to change technologies.
- **Medium**: The technique is advanced and requires a lot of implementation effort or external libraries/processes to work, but can easily be implemented for other technologies.
- **Low**: The technique can easily be implemented and applied to existing applications without components other than the one described for basic caching.

The reason behind the *Low* criteria is that when the technique has assumptions about the technology behind the application it means the system is tied to using specific technologies, which makes it difficult to change when the given technology is obsolete or the requirements of the system change. If this is not considered a problem *Low* and *Medium* can be considered the same level of adaptability.

From this discussion, we can sum up the evaluation parameters as follows:

- **Consistency**: The cached object must be consistent with the data from the primary storage.
- **Strict Freshness**: The cached objects must be based on the newest version of it's underlying data.
- **Update On Invalidation**: The cached objects are automatically updated after they become stale.
- **Always Immediate Response**: The cached objects must be served immediately after they have been requested.
- **No Invalidation Management**: The programmer does not have the responsibility for naming, localizing or invalidating.
- **Adaptability**: How adaptable the caching technique is to existing systems.

Chapter 3

Existing Caching Approaches

Since caching is a solution widely used in practice there already exists multiple caching approaches described in literature, articles on the internet and in open source code. In this chapter we will cover some of these existing approaches. The approaches will be analyzed and evaluated based on the criteria described in section 2.4 and requirements from section 1.2. The chapter will end with an overview of the different approaches and a final conclusion on how to choose the right caching technique.

3.1 Expiration-based Invalidation

In some cases it is acceptable to serve stale values in a certain time interval. In these cases, the simplest approach would be to use the expiration-based invalidation technique, which gives up consistency and makes freshness depend on the chosen expiration, but is then able to respond with the cached object immediately independently of how underlying data changes.

The expiration-based invalidation works by assigning a TTL (Time to Live) to the cached object. At some point the TTL expires after which the cached object

is considered invalid. The responsibility of invalidating cached values with TTL is often placed on the cache database by piggybacking the TTL to the cached value when it is stored. The cache database will then invalidate and remove expired values. This is for example supported by the in-memory cache database technologies, Redis and Memcached.

The timeline model on figure 3.1, illustrates how there is a time interval between the underlying data has been updated and the object has expired in which a stale version of the cached object is served.

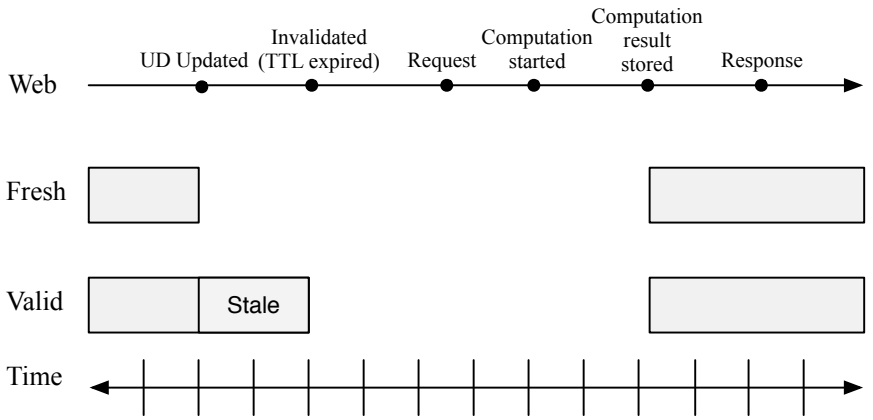


Figure 3.1: The lifecycle of the expiration-based invalidation technique

3.2 Key-based Invalidation

When the cached values are required to be consistent with the primary storage, we must ensure that when underlying data is updated, the depending cached values are updated accordingly. The key-based invalidation gives these guarantees by giving up immediate responses. This means that the key-based invalidation will not be suitable in cases where the computation time exceeds the user's attention span or in cases where the cached values are updated too frequently.

Key-based invalidation works by constructing the cache key from the underlying data such that the key changes in lock-step with the cached content [Han12]. This means there is no specific name identifying some cached content and the cached content for a given key never changes. This simplifies version management from the perspective of cache storage since there is no chance you read

stale values if the key is assumed to be derived from the most recent version of underlying data.

The challenge of this method is to construct the key. To use this technique correctly (i.e. obtain the guarantees promised) the programmer must construct a key that is ensured to change when the cached value is considered stale. Furthermore to obtain a maximum hit rate, the key must not change when the cached value is fresh. Given that the key is optimally constructed, the timeline looks as on figure 3.2. A cached object is considered invalid in the moment after the cache key components are updated (e.g. the *updated_ timestamp* in the code snippet). This means the freshness and validity interval always overlap in the timeline model, and we therefore have consistency.

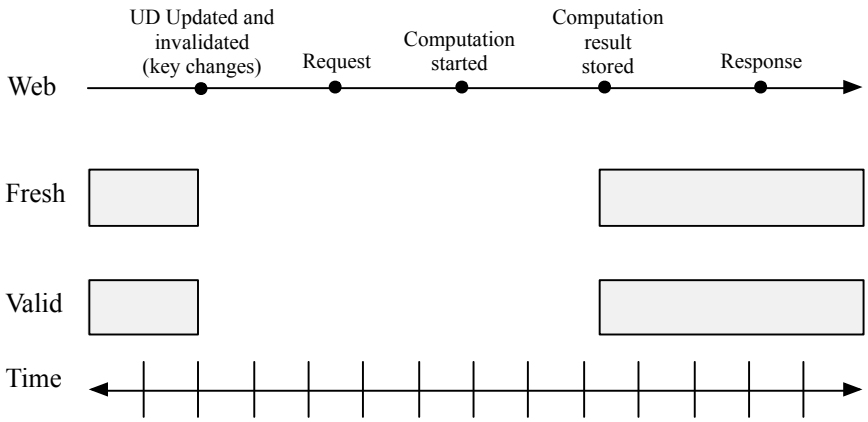


Figure 3.2: The lifecycle of the key-based invalidation technique

In the web application framework, Ruby on Rails, key-based invalidation is implemented by construction the key from the timestamp of the last time the underlying data was updated. The next time the underlying data is updated, the timestamp also changes, and the cached content have thereby been invalidated. An example of this technique can be seen in code snippet 3.1. In this code snippet we use the function name, entity type, entity id and update timestamp as parts of the key. This means that the cached value is invalidated when any of these values changes. The intuition behind these components is that we want a unique cached object for each entity and function and we want the value to be recomputed when the entity is updated i.e. the update timestamp changes.

A caveat of this method method is that it generates cache garbage since new version of cached objects does not overwrite the content for existing keys but stored with a new key. To avoid the complexity of keeping track of the rela-

Code Snippet 3.1: Example of the key-based invalidation technique

```
def time_consuming_participant_score(participant):
    grades = Database.find_all_grades_for_participant(participant)
    return numpy.advanced_statistical_method(grades)

def cache_key_for_participant_score(participant):
    cache_key_components = [
        'cached_participant_score',
        participant.type,
        participant.id,
        participant.update_timestamp
    ]
    return '/'.join(cache_key_components)

def cached_participant_score(participant):
    cache_key = cached_participant_score(participant)
    if is_fresh_in_cache(cache_key):
        return fetch_from_cache(cache_key)
    else:
        result = time_consuming_participant_score(participant)
        set_cached_value(cache_key, result)
        return result

# Load the participant from the primary storage
participant = ParticipantDB.load_one_from_database

# Call the cached version of the time_consuming_participant_score
print cached_participant_score(participant)

# This second time it is called, the return value for
# the time_consuming_participant_score cached
print cached_participant_score(participant)
```

tions between the different cached objects, the responsibility for cleaning up old cached objects is moved to the cache database. Fortunately cache databases (such as Redis and Memcached) implements different cleanup algorithms that detects obsolete values based on some policy. One such policy called *Least Recently Used (LRU)* removes the objects that are least recently used. Another policy called *Least Frequently Used (LFU)* removes the lease frequently accessed objects.

3.3 Trigger-based Invalidation

Instead of invalidating the cached object when requested, the cached values can be invalidated based on certain events triggered when the underlying data is updated. Given that the invalidation triggers are located at all the places where the underlying data is updated, we can achieve the same guarantees of consistency and freshness as with key-based invalidation. As seen on figure 3.3, the timeline is similar to the key-based invalidation and thereby gives the same guarantees of freshness and consistency.

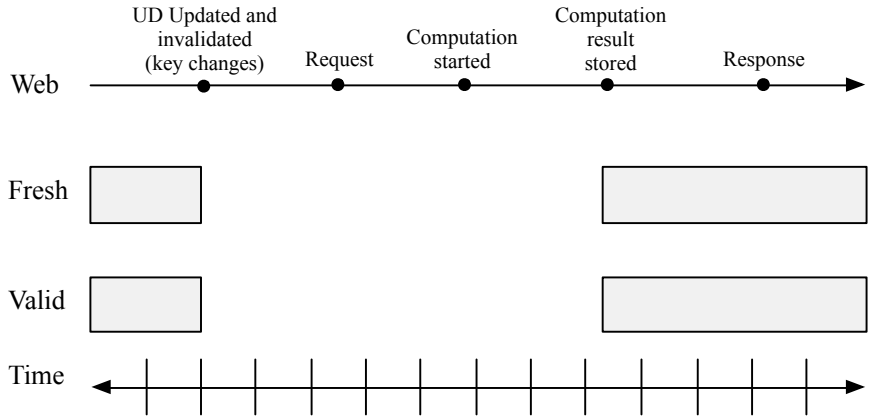


Figure 3.3: The lifecycle of the trigger-based invalidation technique

In key-based invalidation the key is dynamic and used as an invalidation mechanism, but in trigger-based invalidation the key is static i.e. it does not change in the lifetime of a cached object. We will therefore call it the *name* of the cached object. The actual key becomes simpler since it is only responsible for uniquely identifying the cached value, but the localization is still a challenge since the key needs to be shared between the triggers and the places where the cached objects are accessed.

Having a static key also has the advantage that we can locate the old/stale value after invalidation as oppose to key-based invalidation where there is no relation between the versions of cached values. Using this information we can extend the technique to be more fault-tolerant by serving a stale value in the case where a computation fails[1]. Trigger-based invalidation is also a technique

[1]Here we assume that the application provides more value to the client by serving a stale value compared to serving an error or nothing

that comes with a lot of flexibility and is easy to extend as done in section 3.4 and section 3.5.

The simplest type of triggers are manually defined code that invalidates a given key. A code snippet for a naive implementation of manual triggers can be seen in code snippet 3.2. In practice the manual code triggers are often placed right after updates to the underlying data. Although this method is simple it often requires a lot of effort from the programmer and is prone to errors since it requires global reasoning of the application to identify the places where underlying data is updated.

Having a static key also introduces a challenge related to concurrency since we assume that there are multiple application servers. The challenge origins from the problem illustrated on figure 3.4, where an update to the underlying happens during the computation of a cached value. When the computation has finished it will incorrectly mark the cached value as fresh even though it is based on an old version of the underlying data.

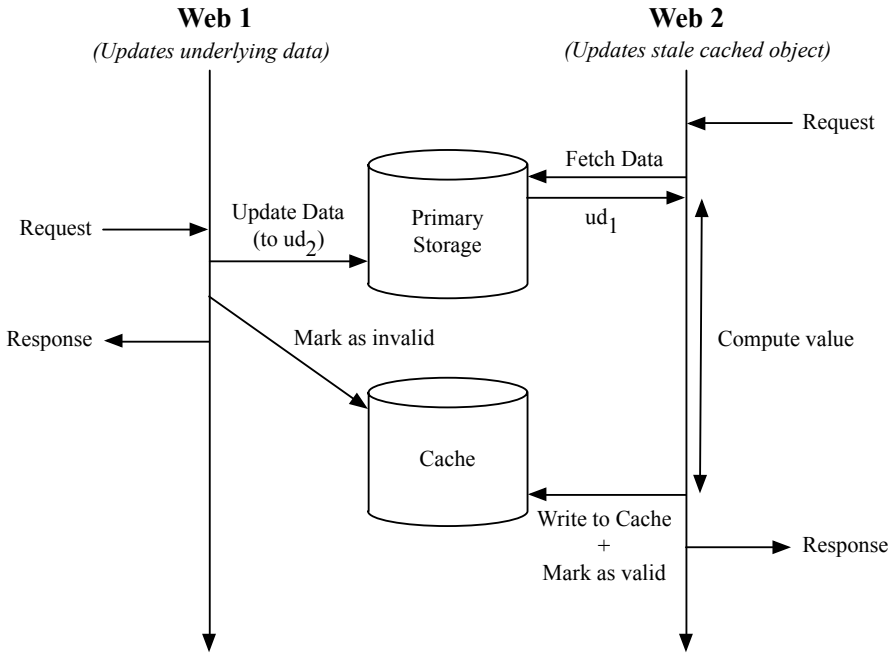


Figure 3.4: A scenario of the trigger-based invalidation that results in a race condition, where the cached value are being incorrectly marked as valid even though it is storing a stale value.

3.4 Trigger-based Invalidation with Asynchronous Update

We can extend the trigger-based invalidation by always serving the newest value from the cache and afterwards update the value asynchronously if it is stale. This way we always get an immediate response by giving up strict freshness and consistency.

The timeline of this extension is as on figure 3.5. A naive implementation of this technique would be as the trigger-based seen in code snippet 3.2 with the modification that the system updates the value asynchronously instead of synchronously if no fresh value is found in the cache. An implementation example for this technique can be found in appendix A.

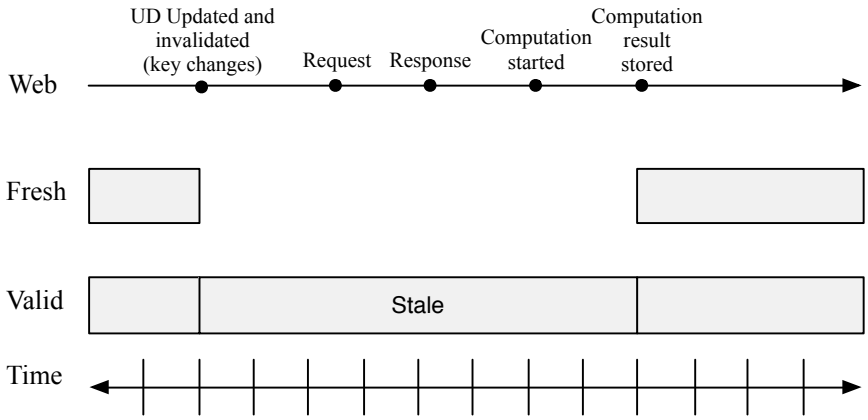


Figure 3.5: The lifecycle of the *trigger-based invalidation* technique where the value is updated in the asynchronous

3.5 Write-Through Invalidation

Write-through invalidation is also an extension to the trigger-based invalidation approach, but instead of updating the cached values when the value is requested, the value is updated in the moment after it has been invalidated. This way we invalidate by writing through the existing version in the cache. The timeline model of this technique seen on figure 3.6 is similar to the asynchronous update

extension, but the time interval in which it serves a stale value is only as long as the time taken to compute the value.

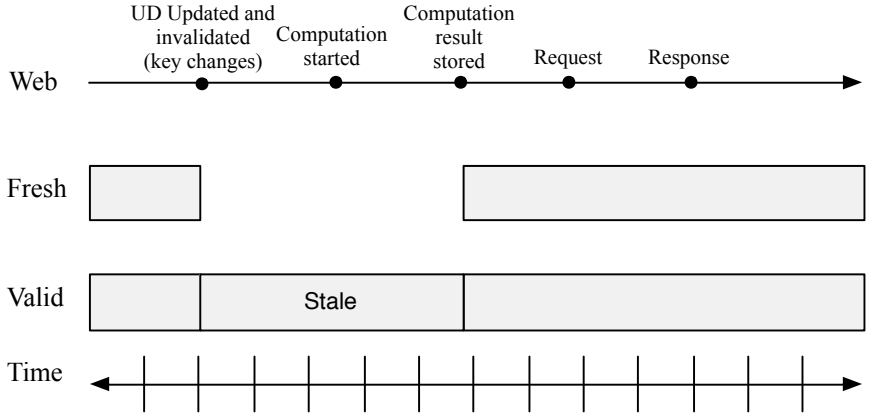


Figure 3.6: The lifecycle of the *write-through invalidation* technique

3.6 Automatic Invalidation

The trigger-based invalidation is an attractive technique since it can provide guarantees with relation to freshness and consistency while allowing for flexibility by extending it. But in practice the overhead for the programmer of managing the triggers and keeping integrity becomes a burden that makes it hard to maintain. A lot of research have therefore been done in making it easier to use trigger-based invalidation by automating cache invalidation and management.

The cached objects are based on underlying data from the primary storage system, which means that changes to the underlying data also origins from the primary store. A lot of work have been put into using the database as the source of the triggers that invalidates the cached objects.

Cache-Genie [GZM11] caches complex database queries by letting the programmer declare predefined queries, which are then cached and updated automatically. It uses an Object-Relational Mapper to detect and trigger changes to underlying data, which will invalidate and update the affected cached queries.

The deploy-time model suggested in [Was11] also uses the database wrapper to

trigger changes to underlying data, but instead of just caching database queries, the deploy-time model is able to cache the results of functions. The programmer is able to declare a function as cached in the source code with declaration of dependencies to underlying data after which the cache system will invalidate the cached functions automatically. The deploy-time model uses static analysis to detect dependencies between different cached functions and to underlying data.

TxCache suggested by Dan Ports et.al. [Por12] also caches the results of functions, but it uses a fully transparent programming model that allows a programmer to declare a function to be cached without any additional information. The TxCache patches the primary storage system to support "invalidation streams" and transactions to detect queries made during a call to a cached function. The primary storage distributes the "invalidation stream" to the cache databases, which uses the stream to invalidate affected cached objects. Using this approach, it is possible to achieve transactional consistency across the cache and primary storage without enforcing full freshness by serving slightly stale data. That is, the TxCache improves the performance of requests to the users while ensuring that all the data served to the user is consistent for the given request. This technique is able to give a very strong guarantee, but it also comes with complexity in the architecture, since it requires an external process that manages snapshots of the databases and it requires the primary storage database to support the invalidation stream technique.

On the other end of the granularity scale, [CLT06] suggests a system that caches HTTP responses. It uses a proxy that that detects the queries made to the primary storage during a HTTP request. Through the information captured by the proxy, the system builds a table that maps a given HTTP resource to the queries made. Furthermore it uses a proxy between the application and database to detect changes to underlying data and invalidate affected cached HTTP responses. This method is interesting since it allows to cache without changing the code of the web application, but it is only described at the granularity of HTTP responses since it uses the communication between the web application, storage system and cache to achieve automatic invalidation. A similar approach is suggested in [CLL+01].

In a series of papers, IBM describes the Content Management System (CMS) they developed for the Olympic Games in 2000 [CDI98, CID99, CIW+00]. In these paper they introduce the Object Dependence Graph (ODG) data structure (described in more details in section 5.1) that is optimized for querying the content affected by changes to underlying data. In this system the dependencies between the different content and the underlying data are declared by the content-writers. That is, the dependencies are not declared in the code base, but they are declared directly in the data. Experiments made to the final system achieves a 100% cache hit rate and the papers describes interesting ideas with

relation to automatic invalidation, but the final system is designed to be used for a CMS-systems.

To be able to evaluate the techniques used in the automatic invalidation approaches described, we will divide the process into different sub problems. Automatic invalidation are based on a caching system that reacts and invalidates when changes happens to underlying data. Automatic invalidation are therefore mainly working around requests, where the client updates underlying data in the primary store. If we consider figure 3.7 that illustrates this flow, the invalidation mechanism is responsible for step 3, 4 and 5. This involves the tasks of: *triggering cache invalidation* and *managing dependencies between underlying data and cached objects*. The following sections will consider these tasks and compare existing techniques.

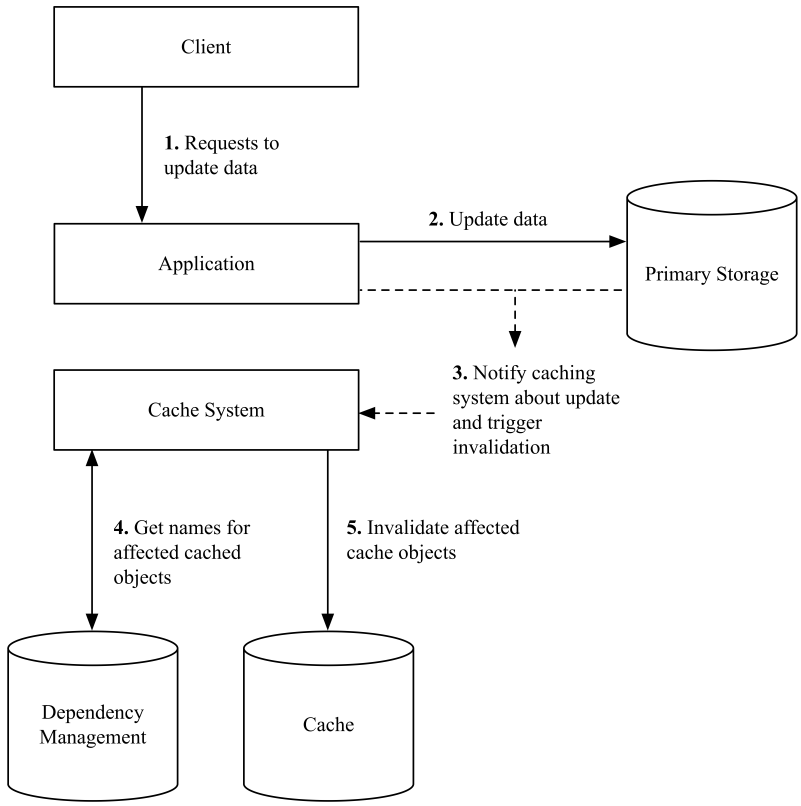


Figure 3.7: The control flow of automatic invalidation when a client requests to update underlying data

3.6.1 Triggering Cache Invalidation

By using the database wrapper as in [GZM11, Was11], the application becomes responsible for detecting changes to underlying data such that the cache system does not need to introduce external processes listening to changes or patch the database. One advantage of using a database wrapper is that in some cases, the wrapper is able to support multiple database technologies such that it e.g. is possible to switch from one SQL technology to another without changing much code. On the other hand changes made to the database that are not made through the given database wrapper are not detected as a trigger, which leaves the responsibility of ensuring all changes are made through the database wrapper. By using a database wrapper we therefore get better adaptability by simplifying the architecture, but require discipline with relation to how changes are made to the primary storage.

Other solutions such as [Por12, CDI98, CID99] assumes that the database is able to send notifications when changes are made to the database. The information from the notifications are intercepted by the caching system and converted to invalidations. [Por12] uses an "invalidation stream" that is replicated directly across all cache nodes, which means the cache nodes has the responsibility of invalidating the correct cached objects. In [CDI98, CID99] this responsibility is extracted into a *cache manager* that transforms the change notifications to invalidations. A similar approach is used in [CLT06] uses a proxy between the application and the primary storage to "sniff" the database traffic and relate it to the HTTP-request of the client.

A comparison of different invalidation trigger techniques is shown on figure 3.1.

	Advantages	**Disadvantages**
Directly from Database	• Any change is captured	• Require database to support triggers
Database Sniffer	• Any change is captured	• Require a database sniffer for the database technology used • Requires an extra running process
Database Wrapper	• Supports all database technologies supported by the database wrapper or the API used by the database wrapper	• Changes made to the database around the database wrapper are not detected

Table 3.1: Comparison of triggers for automatic invalidation

3.6.2 Dependency Management

After the invalidation has been triggered, the invalidation system needs to locate the cache objects that must be invalidated. This involves the task of identifying and declaring dependencies such that the triggers will invalidate the affected cached objects efficiently.

Since dependency management is a burden for the programmer and affects the correctness of the cache implementation, it would be most desirable to have fully automatic dependency management, which is achieved in [Por12, CLT06]. [CLT06] use the technique of proxies between the different servers to sniff the traffic and thereby automatically derive dependencies between HTTP-responses and queries made during the request. [Por12] runs a transaction with the database while the cached object is computed and uses information from the queries made during the transactions to derive dependencies between the underlying data and the cached object. These techniques removes the burden of cache management by having fully transparent caching, but it also means the programmer has less flexibility. Furthermore they are tightly coupled to the technologies used and makes it difficult to port the solutions to other technologies.

Other solutions such as [GZM11, Was11] relies on the programmer declaring dependencies from the cached objects to underlying data. Since the trigger can include information about which underlying data are changed, the caching system can use the declared dependencies to invalidate the corresponding cached objects. [GZM11] supports declarations through function calls that are stored in the memory of the application. The deploy-time model suggested in [Was11] uses static analysis of comments to allow the functions to be executed without the cache database. These techniques does not remove the burden of cache management completely, but they allow the programmer to specify the dependencies in a more declarative and robust way compared to using manual invalidation triggers.

The solution suggested by Jim Challenger et.al. [CDI98, CID99] uses dependencies declared in the content to construct an advanced dependency graph. When updates are made to content or underlying data, the affected cached objects are derived using the dependency graph. This solution is developed for a content management system, where the users can declare dependencies between the fragments of the content i.e. the dependencies are declared in each entity. This makes it unfeasible to use in application with slightly advanced data models, but it solves the problem well in the given case.

An overview of this discussion can be found in figure 3.2.

	Advantages	Disadvantages
Declared in code ([GZM11])	• Easy to reason about dependencies	• Relies on the developer declaring dependencies
Declared in content ([CDI98, CID99])	• Allow different data source for different entities • The developer does not have to declare dependencies	• Burden for users to define dependencies on each entity
Code Generation From Static Analysis ([Was11])	• Semi transparent (requires definition of database relations)	• Does not work for dynamic programming languages • Difficult to detect relational dependencies • Requires knowledge about static analysis to implement
Database Transactions with Invalidation Tags ([Por12])	• Fully transparent	• Require the database to implement the transactions

Table 3.2: Comparison of dependency management techniques for automatic invalidation

3.7 Choosing the Right Caching Technique

In general there is not a best or correct caching solution - it's a matter of choosing the solution best suited in the given context depending on the architecture of the web application and the specific use case. To get closer to the solution suited for our context (section 1.3) we will compare the existing approaches based on the criteria described in section 2.4. An overview of the comparison on table 3.3 shows how the different caching approaches relates to the evaluation criteria.

	Consistency	Strict Freshness	Update On Invalidation	Always Immediate Response	No Cache Management	Adaptability
Arbitrary Content						
Expiration-based	-	-	-	Yes	Yes	High
Key-based	Yes	Yes	-	-	-	High
Manual Trigger-based	Yes	Yes	-	-	-	High
Async. Update	-	-	-	Yes	-	High
Write-Trough	-	-	Yes	Yes	-	Medium
Chris Wasik [Was11]	Yes	Yes	-	Yes	Yes*	Medium
TxCache [Por12]	Yes	Yes**	-	Yes**	Yes	Low
Declared Content						
IBM [CDI98, CID99]	-	-	Yes	Yes	Yes	Low
HTTP-Response						
Chang et.al. [CLT06]	-	-	Yes	Yes	Yes	Low
DB-Queries						
Cache-Genie [GZM11]	Yes	-	Yes	Yes	Yes	Medium
Materialized Views	-	-	-	Yes	Yes	Medium

*) Dependencies to underlying data must be declared, **) Does not give these guarantees fully, but approximately.

Table 3.3: Comparison of caching approaches for different types of content

Code Snippet 3.2: Example of how trigger based invalidation works with manual code invalidation

```
def time_consuming_participant_score(participant):
    grades = Database.find_all_grades_for_participant(participant)
    return numpy.advanced_statistical_method(grades)

def cache_key_for_participant_score(participant):
    cache_key_components = [
        'cached_participant_score',
        participant.type,
        participant.id
    ]
    return '/'.join(cache_key_components)

def cached_time_consuming_function(participant):
    cache_key = cache_key_for_participant_score(participant)
    if is_fresh_in_cache(cache_key):
        return fetch_from_cache(cache_key)
    else:
        result = time_consuming_participant_score(participant)
        set_cached_value(cache_key, result)
        return result

# Load the participant from the primary storage
participant = ParticipantDB.load_one_from_database

# Call the cached version of the time_consuming_participant_score
print cached_participant_score(participant)

# This second time it is called, the return value for
# the time_consuming_participant_score is cached, which means it
# returns the value from the cache
print cached_participant_score(participant)

# Now we invalidate the cached value
cache_key = cache_key_for_participant_score(participant)
invalidate_cached_value(cache_key)

# Since the value has been invalidated
# the cached_time_consuming_function will recalculate
# the result when called at this point
print cached_participant_score(participant)
```

Chapter 4

Smache: Cachable Functions

The goal of this thesis is to present a caching solution that is able to handle long running computations by presenting content to the user fast while addressing the challenges of cache management and efficient update propagation. This chapter will present the overall solution suggested by the thesis.

In this chapter we will start by arguing why the existing caching approaches are not sufficient (section 4.1). Followed by these arguments, we present the programming model that is the basis of the solution (section 4.2), how we implemented this model in Python (section 4.3), and a final discussion on the limitations of the design and implementation of the model (section 4.4).

4.1 Why Existing Approaches are Not Sufficient

In the primary use case of the context in this thesis (described in section 1.3), the platform presents statistical information based on some advanced computation that takes a long time to compute ($>$ 10 sec.). If we want to keep the teacher's attention to the platform, we need to find a caching technique, where the response time does not depend on the computation time. Furthermore we

want to keep the information as fresh as possible.

If we consider the overview on figure 3.3, we can already leave out caches that optimize DB-queries and declared content since they do not solve the problem of caching advanced statistical computations. The solution proposed by IBM is made for declared HTML-content and not on computations based on data from a storage system. This leaves the set of approaches that are able to cache arbitrary content and HTTP-responses, but given caching arbitrary content has less assumptions, they are more attractive.

From the approaches caching arbitrary content it is desirable to have an approach with high adaptability, which is one of the requirements of the system. From the approaches with high adaptability, which has a response time that does not depend on the time of computation, are *expiration-based* and *manual trigger-based invalidation with asynchronous updates.*

From these approaches the expiration-based invalidation technique has the advantage that it doesn't require much invalidation management, but it has the limitation that all cached values of the same kind are invalidated and updated at the same frequency.

If we consider use case example 4.1, the statistical information of current assignments would be updated at the same frequency as the closed assignments, and if we want to keep the cached values for the current information up to date, we also need to update all non current information. This is not desirable with relation to efficiency since the CPU will be busy in time proportional to the taken to compute the statistical information for the assignment and the total number of assignments on the platform. In other words the number of CPU's we need to occupy for this job can be calculated by $\frac{\Delta t_c \cdot n_c \cdot u_c}{t_d}$, where Δt_c is the time taken to compute a single assignment, n_c is the number of assignments, u_c is the number of updates per 24 hours, and t_d is the number of seconds per 24 hours. Considering the example - if we have 500 assignments on the platform and we want to update the information every 10 minutes with a computation time of 30 seconds each, then we get $\frac{30 \cdot 500 \cdot \frac{60}{10} \cdot 24}{60 \cdot 60 \cdot 24}$ occupied CPU's $= 25$ occupied CPU's. If the amount of computational power isn't a problem, the expiration-based approach would be the best solution, but that is not the case of this thesis, where efficiency is a requirement. $\frac{30 \cdot 500 \cdot \frac{60}{10} \cdot 24}{60 \cdot 60 \cdot 24}$ occupied CPU's $= 25$ occupied CPU's. If the amount of computational power isn't a problem, the expiration-based approach would be the best solution, but that is not the case of this thesis, where efficiency is a requirement.

The last alternative with high adaptability is to use manual trigger-based invalidation with asynchronous updates after request as described in section 3.4. This

approach has the advantage that the values are only invalidated (and thereby recomputed) when a trigger is invoked, which means the programmer has the opportunity to optimize the approach to only invalidate when it is relevant to the user e.g. when the cached value are supposed to change. Although this can be seen as an opportunity it can also be seen as a burden for the programmer to maintain these manual triggers. Furthermore by having asynchronous invalidations, the user is always presented with a stale value on the first request after invalidation. If we consider example 4.1 it is not unlikely that the teacher looks at an old assignment some time after it has been closed, which might just be a single request. In this case, the cache system would serve a stale value and update the value without purpose.

Example 4.1 *On the Peergrade.io platform the teacher is presented with statistical information about the grades given by the students for a given assignment. These assignments are often current at specific time interval after which the assignment becomes "closed" and the statistical information are not updated.*

The approaches suggested by Chris Wasik and Dan Ports focus on automatic invalidation and could probably be extended to do asynchronous updates on request to allow immediate responses by giving up the option of freshness, which would give the same guarantees as the trigger-based invalidation with asynchronous updates, but it would also have the same challenges.

The deploy-time model [Was11] and TxCache [Por12] chooses consistency over ensuring immediate response times and are therefore not suitable for long running computations. TxCache shows how it achieves a higher cache hit rate by limiting the number of recomputations in each request, but it does not reach 100% hit rate. The deploy-time model could be extended to have immediate response time by using write-through invalidation, but it uses static invalidation, which is highly coupled to the specific technology, and makes it difficult to change technologies.

The approach best fit for the context of this thesis is to use write-through invalidation that guarantees immediate response time as well as updating the content in-place such that the content cannot become older than the time taken to compute the value. Although this is the best fit, it still leaves a burden for the programmer to declare the triggers that invokes the write through updates as well as naming the cached objects. Furthermore it leaves the challenge of ensuring the integrity of the cached values and invalidation marks in concurrent environments as described on figure 3.4.

The goal of the thesis is to design a solution that is easy for existing applications to adapt and support caching long running computations. To support long

running computations the cache must be able to serve slightly stale values to serve the result immediately, while keeping the result up to date in an efficient way. As we can see on figure 3.3, there exists no solutions fulfilling those criteria, and we will therefore suggest a new solution, which we call "Smache".

In the rest of this thesis we will describe the design and implementation of "Smache".

4.2 The Cachable Function Model

Smache uses an extended version of the programming model described by Dan Ports [Por12]. The model is organized around *cachable functions*, which essentially are normal functions, where the result are *memoized*[1]. The cachable functions are allowed to make request to the primary storage to fetch the underlying data as well as calling other cached functions. This allows caching at different granularities that gives the benefit of optimizing the invalidation for different types of content.

4.2.1 Restricted to Pure Functions

As mentioned by Dan Ports [Por12] not all functions are cachable. To be able to cache the functions they need to be *pure*, which Dan Ports defines as: *"[...] they [functions] must be deterministic, not have side effects, and depend only on their arguments and the storage layer state."*. Smache does not detect these properties in a function and therefore relies on the programmer to ensure only pure functions are cached.

4.2.2 Making Functions Cachable

Smache has automatic invalidation, but it is only semi-transparent, since that relies on the programmer declaring dependencies to underlying data. The API for making functions cached is defined as following:

`MAKE-CACHABLE(fn, input-types, relation-deps)` → *cached-fn*: Makes a function cacheable. *cached-fn* is a new function that first checks the cache for

[1] In our case: storing the result of the function and return the cached value when the function is called again with the same input

the result of another call with the same arguments. If not found, it executes *fn* and stores its result in the cache.

The added arguments for the procedure does not alter the behaviour of the *cached-fn*, but it requires additional arguments used by the application for naming and invalidation. The *input-types* are the types of the arguments (e.g. the type of entity or type of raw content) given to the original *fn* and the *relation-deps* are declared dependencies to data sources, which cannot be derived from the input of *fn*. The *input-types* are used to indicate how the arguments should be interpreted with relation to naming (described in section 4.2.3) and used together with the *relation-deps* to achieve automatic invalidation (described in chapter 5).

4.2.3 Cache Object Localization

When the application fetches and stores cached objects, the developer must derive a name that is used as the key in the cache store and for identifying the cached object. To localize a cached object correctly the name *must be unique* such that no other cached function is able to have the same name. To make it easier for the programmer to cache functions, Smache automatically generates the name based on the input given to `MAKE-CACHABLE`. The localization uses the `SERIALIZABLE-FUN-NAME` procedure that is defined as following:

`SERIALIZABLE-FUN-NAME(fn, input)` → *fun-name*: Returns the name that uniquely identifies the cached object instance representing the call to the function *fn* with *input*. The name is constructed by concatenating a unique serialized version of *fn* and a serialization of the *input*. The serialization of *input* is done by concatenating a serialized version of each argument.

This procedure makes it possible to represent the cached object in the cache to locate and store the result of a given cached function in the cache database.

To support asynchronous write-through updates that allows the cache system to automatically update the cached objects as they are invalidated, `SERIALIZABLE-FUN-NAME` must also comply with the following requirement:

- The serializations of *fn* and *input* must be done in such a way that there exists another procedure `DESERIALIZE-FUN` that deserializes *fun-name* such that it is possible to locate and call *fn* with *input* in another process.

The `DESERIALIZED-FUN` is defined as following:

`DESERIALIZED-FUN(fun-name)` → *fun*, *input*: Returns a pointer to *fun* as well as the original *input*. It must be possible to call *fun* with *input* as arguments such that the result is the same as before `SERIALIZABLE-FUN-NAME` was applied.

4.2.4 Automatic Cache Invalidation

Dan Ports' TxCache [Por12] handles all parts of cache management transparently such that the programmer only have to define which functions needs to be cached and not define how it should be invalidated. This is a great advantage since it avoids potential bugs related to cache invalidation, but it is a trade-off for flexibility as well as adaptability since this approach moves the complexity to the database in the form of assumptions about the primary storage, cache database, and requires an external daemon process for managing snapshots [Por12].

Smache does not remove the burden of cache management, but it will handle naming (or localization) and invalidation by only relying on the programmer declaring dependencies to underlying data. This will improve the usability of cache management since the cache invalidation only changes when there are changes to the underlying data as described more in chapter 5.

4.2.5 Write-Through Invalidation

Smache also offers the possibility to perform write-through invalidation using a data update propagation system inspired by the solution IBM used to achieve a 100% cache hit rate on the content management system for the Olympic Games in 2000 [CDI98, CID99].

Having write-through invalidation allows the programmer to achieve immediate response times while ensuring that the values returned are updated as soon as they are invalidated, such that the value is not more stale than the time taken to compute it. This will be explained more in chapter 6.

4.3 Implementing Cachable Functions in Python

Smache implements the cachable function by exposing an implementation of the `MAKE-CACHABLE` to the programmer. To apply the procedure the Smache library must be loaded in the application and applied. The `MAKE-CACHABLE`

converts the original function to a cached function that intercepts the method invocation and directs it to a wrapper function in the Smache library. When a client makes a request involving a cached function, the application will call the function wrapper that tries to find the value in the cache first and if it is not present in the cache, the original function will be called after which the result is cached and returned to the caller. This control flow of a cached function is illustrated on figure 4.1.

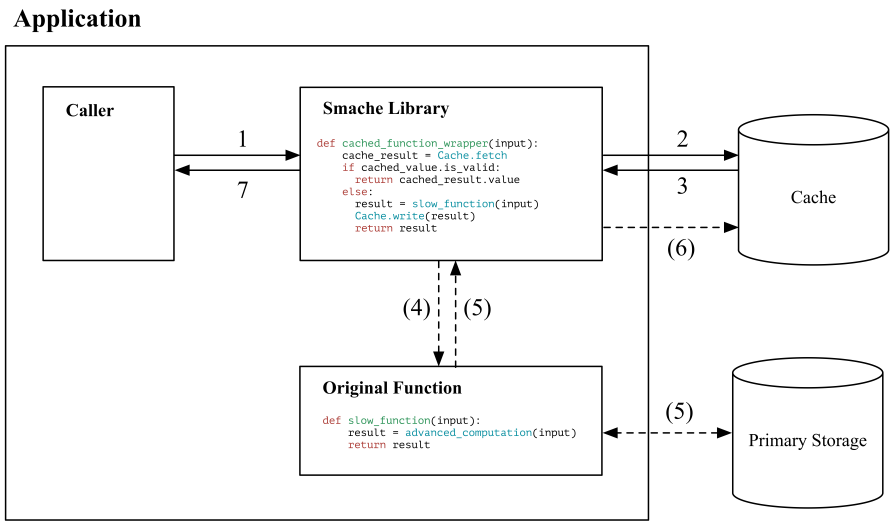
The procedure for making a function cached is implemented using Python-decorators [2], which is an annotation used to modify the behaviour of existing functions while being able to keep a reference to the original function.

Code snippet 4.1 shows how the decorators are applied to the running example. The only addition we have made is a call to a function called `computed` on a module `smache` with a single argument and a `@` as prefix that indicate it is a decorator. Additionally there is a decorator called `relation` in which we indicate the entity type as well as a function that describes how the given relation entity relates to an entity indicated through the input. This call corresponds to the implementation of `MAKE-CACHABLE`, where the arguments given to `computed` are the `input-types` and the arguments given to `relation` are the `relation-deps`.

Smache require the programmer to annotate the types of "underlying data", which are passed into the original function. In this case we have annotated using the `Course`-class that corresponds to an object for an Object-Relational Mapper (ORM) that represents the collection *course*. The "underlying data types" supported are ORM-objects as well as the "Raw"-type that represents primitive values (such as strings, numbers etc.). The number of type annotations has to be the same number of arguments given to the original function. The type annotations are used to derive direct dependencies as explained more in section 5 and to implement automatic naming.

The Smache library implements both the `SERIALIZED-FUN-NAME` and `DESERIALIZED-FUN` to support automatic write-through invalidation. The implementation works by having a *seperator-token* that is used in between the concatenations. The input is deserialized and serialized using the *JavaScript Object Notation*-protocol (JSON) [Doca]. Using the python implementation [Docb] Smache encodes primitive Python values into JSON and decodes them back from JSON to Python values. The implementation of the encoding and decoding can be seen in appendix B. Besides decoding and encoding the cached object instances, the full implementation also involves storing and loading references from/to the computed functions. To give an example, we will demonstrate the usage of the localization procedures with the default seperator token `~~~`. We can serialize

[2] https://www.python.org/dev/peps/pep-0318/



1. The caller invokes the cached function through the cached wrapper in Smache
2. Smache sends a `Lookup` request to the Cache Database
3. The cache database returns a cached object instance

(4) Smache calls the original function through the application

(5) The application executes the original function, makes the necessary calls to the primary storage and returns the result

(6) Smache sends a `Store` request to the Cache Database with the result from the application

7 Smache returns the value to the caller from the application

Steps annotated with *parenthesis and dashed lines* are only executed in case of cache miss or cache failure

Figure 4.1: The control flow during a call to a function cached through Smache

the function `course_score` from the running example in code 1.1. When the running example has been made cachable as in code 4.1 and we use a `Course`-instance from the `ORM` we can serialize and deserialize as seen in code 4.2.

Code Snippet 4.1: How to cache the functions from the running example using Smache.

```
@smache.relations(
    (Participant, lambda participant: participant.courses),
    (Grade, lambda grade: grade.assignment_course)
)
@smache.computed(Course)
def course_score(course)
    participants = Database.find_all_participants_in_course(course)
    total_score = 0
    for participant in participants:
        total_score += time_consuming_participant_score(participant)
    return total_score / len(participants)

@smache.relations(
    (Grade, lambda grade: grade.graded_participant)
)
@smache.computed(Participant)
def time_consuming_participant_score(participant):
    grades = Database.find_all_grades_for_participant(participant)
    return numpy.advanced_statistical_method(participant)
```

4.4 Limitations of Cachable Functions

The design of the cachable functions require the programming language to support serialization and deserialization as described. In Python it is possible to localize modules and functions dynamically based on a string representation, but in other languages this could be a challenge or might even require a different solution, which could be a limitation of both the design and implementation. For example statically typed languages that does not allow dynamic notations, might need a different solution, where the representations are mapped to a data structured holding references to the given functions. Furthermore the solution does not consider caching methods for objects in object oriented languages. To cache methods of objects, the solution would have to be extended to consider the state of the object as "input" for the cached function.

Another limitation of the interface of cachable functions described so far is that it is only possible to use it for trigger-based invalidation and the only control for the programmer is to define the dependencies such that the cached object is invalidated appropriately. There could be cases, where the cached object were

Code Snippet 4.2: Example of how function serialization and deserialization can be done in Smache

```
# Fictive course
course = Course(id='56123123jkasd')

# Assumed output
print course_score(course)
# => 5.52

# Serialization of cached object for course_score called with course
computed_key = smache.computed_key(course_score, course)

print computed_key
# => 'module_name.course_score~~~56123123jkasd'

# Deserialization
computed_fun, arguments = smache.deserialized_fun(computed_key)

print computed_fun(*arguments)
# => 5.52
```

to be invalidated based on some data sources, but also be updated once every day using expiration-based invalidation. Fortunately, the interface can easily be extended using decorators to include optional parameters that would allow hybrid approaches such as these.

4.5 Summary

In this chapter we argued why existing approaches does not solve the problem addressed in this thesis and presented the programming model for the suggested solution. We introduces the interface for *cachable functions* that allows a programmer to declare a function to be cached. When a cached function is called the second time with the same input, the result will be served from the cache. The naming and identification of the related cached object instance is handled transparently by the caching system. The interface also includes declaration that allows cachable functions to be extended with automatic invalidation as described in the next chapter (chapter 5) and write-through invalidation (chapter 6). The programming model has been implemented in Python, where the

declaration for caching a function is implemented using decorators. The serialization and deserialization is implemented by encoding the function and input into JSON and decoded back into a pointer to the function and Python values. This procedure might be a limitation for dynamic languages such as Python.

Chapter 5

Automatic Cache Invalidation

In the model for cachable functions described in the previous chapter 4 we've only described how the system can make a function cachable such that Smache automatically stores and locates the cached object returned by the function. While this removes the burden of naming and localizing cached objects, it does not solve the hard problem of invalidation.

To make cache invalidation easier with Smache, we will extend this solution with a mechanism that automatically invalidates the cached objects based on dependencies declared by the programmer.

This chapter describes how the suggested system uses the declared dependencies to track updates to underlying data and invalidate cached objects correctly. The design of the solution for automatic invalidation is described in three parts. First, we introduce the data structures used to achieve fast and efficient invalidation (section 5.1). Secondly, we describe how the cached objects are registered into the ODG (section 5.3). Finally, we describe how Smache invalidates affected cached objects when changes are made to the underlying data (section 5.4). After the design of the solution, we explain how we implemented automatic invalidation in Python and ends the chapter with a brief summary.

5.1 Simple Object Dependence Graph

The Object Dependence Graph (ODG) was first described in a series of papers by IBM [CDI98, CID99, CIW+00] and designed for the content management system used for the Olympic Games in 2000. In this solution, the final content served to the user is build from HTML-fragments and HTML-pages editable by the users as well as underlying data periodically changing. To be able to serve the final documents fast, the system pre-generates the HTML pages such that the system doesn't have to generate the pages on each request. To do this efficiently, the system maintains dependencies between the different kinds of objects and updates depending objects when underlying data changes. The cache object pre-generation can also be described in terms of caching, where the system uses an write-through invalidation approach.

Jim Challenger et.al. [CID99] presents a simple and a generalized version of the ODG. The generalized version is described for a content management system and includes a number of enhancements to support the requirements of the CMS. These enhancements are not necessary to solve the problem of this thesis, and will therefore use the simple ODG that is described as following:

ODG can be represented using a directed graph, where a vertex either represents underlying data or an object that is a transformation of underlying data. An edge from a vertex representing underlying data u to a vertex representing an object o denoted (u, o) indicates that a change to u also affects o. [CID99] gives the following constraints for the simple ODG:

- Each vertex representing underlying data does not have an incoming edge
- Each vertex representing an object does not have an outgoing edge
- All Vertices in the graph correspond to underlying data (nodes with incoming edges) or objects (nodes with an outgoing edge)
- None of the edges have weights associated with them

Figure 5.1 illustrates an instance of a simple ODG with four vertices of underlying data (u_1, u_2, u_3 and u_4), two vertices representing objects (o_1 and o_2), and the dependencies between them. We can see that when underlying data u_2 changes, the system must update both the cached object of o_1 and o_2 and when u_4 changes, it only needs to update o_2.

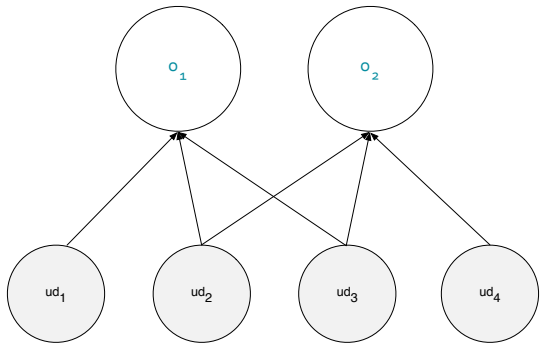


Figure 5.1: An example instance of a simple ODG

5.2 Dependency Data Structures

If we consider the techniques proposed for dependency management in existing solution, which we compared on figure 3.2, the database transaction technique allows for the most transparent dependency declaration and require no effort from the programmer, but it makes assumptions about how the database technology works, which makes it more difficult to make adaptable. Declaring dependencies in the content is not appropriate for caching functions and the technique with code generation requires lots of effort to implement the static analysis, which will only work for a specific programming language and database query language. Smache therefore uses a modification the last technique, where dependencies are declared in code as suggested by Gupta et.al. [GZM11] using variants of the Simple ODG.

There exists two representations of dependencies for cachable functions. The first one is the static representation, which can be derived directly from the declarations in the source code. This representation only describes dependencies between the declarations and not the cached objects. We will define this representation as the *Declaration Dependence Graph (DDG)*. The other representation is dynamic and describes the dependencies between the entities of the underlying data and the cached objects. We define this as the *Instance Dependence Graph (IDG)*.

5.2.1 Declaration Dependence Graph

When the cachable functions are defined as in section 4.2.2, the dependencies are declared using the entity types. The DDG is an extension of the Simple Object Dependence Graph, where the cached functions corresponds to the object vertices and the entity types corresponds to the underlying data. Where the simple ODG only includes direct dependencies, the DDG has two kind of dependencies. An edge from u to o denoted $(u, o)_d$ represents a *direct dependency*, which means that the dependencies identifies the given o. The direct dependencies includes an index indicating a position such that there is an order of direct dependencies for a given cached object. An edge from u to o denoted $(u, o)_l$ represents a *lazy dependency*, which are used for the dependencies that changes over the lifetime of the cached object. The lazy dependencies are declared using procedures relating to direct dependencies and are evaluated every time there is an update to underlying data. Alternatively we could manage these dynamic dependencies in the data structure, which could make the evaluation more efficient, but to reduce the complexity by limiting the amount of state, we evaluate the declarations every time the related underlying data is updated.

Figure 5.2 shows the DDG for the running example from code snippet 1.1. We see that the cached object for the `participant_score`-function depends directly on the participant and it depends lazily on the grades. This is because there exists a participant for each score, but the grades for a given participant changes in the lifetime of the cached score and is therefore lazily evaluated.

The DDG graph is represented using two data structures. The lazy dependencies are represented using an outgoing adjacency list from the underlying data nodes. The direct dependencies are represented using an incoming adjacency list from the object nodes ordered by the position index of the dependency. To access a given adjacency list fast we use hash tables indexed by entity type for the outgoing adjacency list and by the object ID for the incoming adjacency lists as depicted on figure 5.3.

The advantage of the DDG is that it does not change in the lifetime of the application, which means it can be preprocessed when the application starts and it does not have to support update queries.

The DDG is build by iterating through all the cached functions with their respective dependencies. The lazy dependencies are added with the entity type, the cached function and the procedure to evaluate it. The direct dependencies are added using with the cached function and the entity type. When a new dependency is added to the data structure, we lookup the indexed value in the hash table and if there exists an adjacency list, we add the dependency

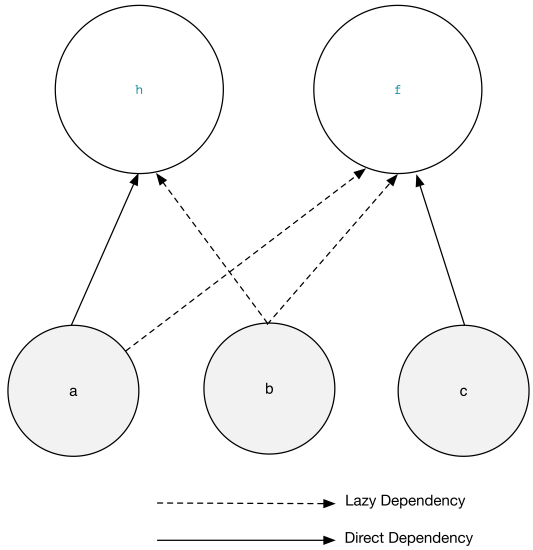


Figure 5.2: The Declaration Dependence Graph of the running example

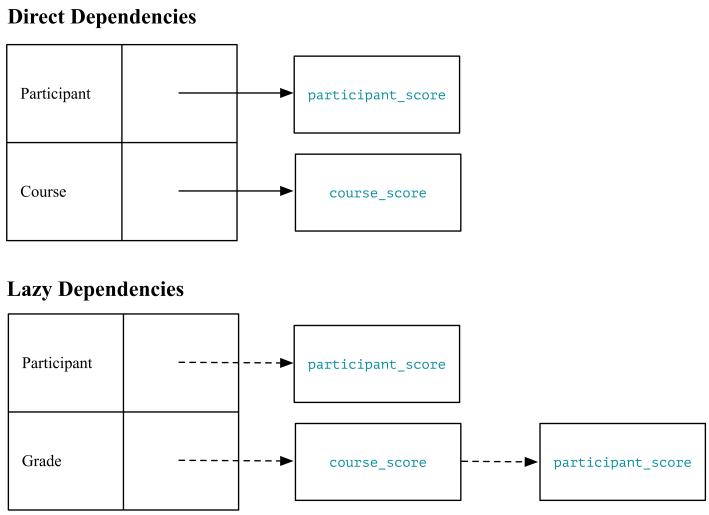


Figure 5.3: An illustration of the data structure representing the DDG on figure 5.2

to the list. If there are no list, we create a new adjacency list with the given

dependency as the only value.

The queries needed to find dependencies for automatic invalidation are the following:

- `lookup_lazy_dependencies(entity_type)`: Finds all lazy dependencies from a given entity type
- `lookup_direct_dependencies(fun_id)`: Finds all direct dependencies from a given function id

Using these data structures we can access a pointer to all lazy or direct dependencies using $O(1)$ expected time if we use a hash table with perfect hashing. [FKS84]. In worst case the space used is the maximum number of edges $O(|u| \cdot |o|)$, where $|u|$ denotes the number of nodes representing underlying data and $|o|$ represents the number of nodes representing cached objects. In our case the number different underlying data is the number of different entity types and the number of different cached objects are the number of different cached function definitions.

5.2.2 Instance Dependence Graph

The Instance Dependence Graph (IDG) is also an extension of the simple Object Dependence Graph (ODG), where the data entities corresponds to underlying data and the cached object instance generated from the cachable functions. An edge from underlying data u to an object o denoted $(u, o)_i$ indicates that if u is changed then o must be updated.

The instance dependence graph for the running example is illustrated on figure 5.4. In the given example, this graph is quite primitive, but it is worth noting that the edges in the IDG are instance specific representations of the direct dependencies of the DDG seen on figure 5.2.

The representation of the IDG is similar to the representation of lazy dependencies for the DDG, where the dependencies are represented using an outgoing adjacency list indexed by a hash table as illustrated on figure 5.5.

New dependencies can be added to the IDG using, the following update query:

- `add_dependency(entity_id, o_id)`: Adds a dependency between the

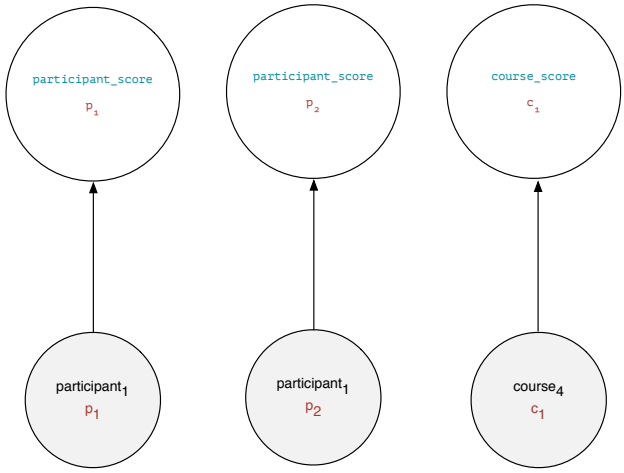


Figure 5.4: An example of an Instance Dependence Graph based on the running example, where we have a single course entity that have two participant entities.

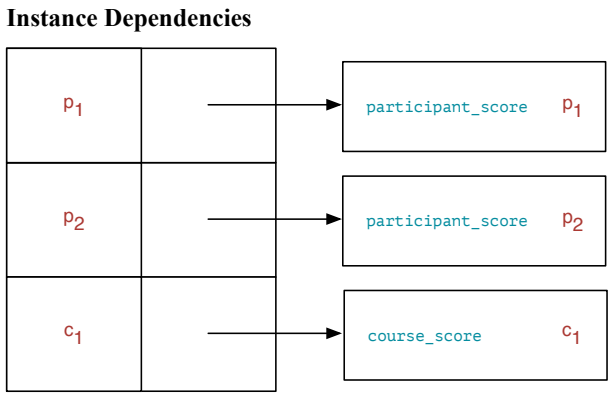


Figure 5.5: An illustration of the data structure representing the IDG on figure 5.4

entity id `entity_id` used as index for the hash table and the cached object id `o_id` used as element in the adjacency list.

When `add_dependency` is used we check if there already exists an adjacency list in the hash table using `entity_id`. If there exists one, we add `o_id` to the given

adjacency list, or else we add a new adjacency list only including `o_id` indexed by `entity_id`.

The queries needed for automatic invalidation are:

- `lookup_cached_objects(entity_id)`: Finds all cached objects depending on the `entity_id`.

If we use perfect hashing as with the DDG we will also be able to lookup instance dependencies using $O(1)$ expected time. If we denote the total number of direct dependency declarations for all the cached functions as $|d|$ then the IDG uses $O(|d|)$ space in worst case.

5.3 Dependency Registration

For the application to know what and when to invalidate, the application must register the dependency declarations and names of cached objects. The dependency declarations that defines the lazy and direct dependencies are part of the source code and therefore available when the application is started as illustrated on figure 5.6. This registration flow collects all the cached functions registered through the cachable function procedure and adds them as dependencies in the DDG.

The registration of cached object instances happens when a cached object is accessed for the first time as illustrated on figure 5.7. In this flow we start by looking up the given cached object in the IDG, and if it has not been registered before, we add a dependency to the IDG.

To generate the name of the cached objects we use the ordered direct dependencies from the DDG combined with the input from the request. We derive dependencies to underlying data from the input that represents entities and add dependencies between the given entities and the cached object through the IDG.

5.4 Invalidation Propagation

The purpose of invalidation is to be able to evaluate the freshness of a given cached object i.e. know which cached objects are fresh and which are stale and

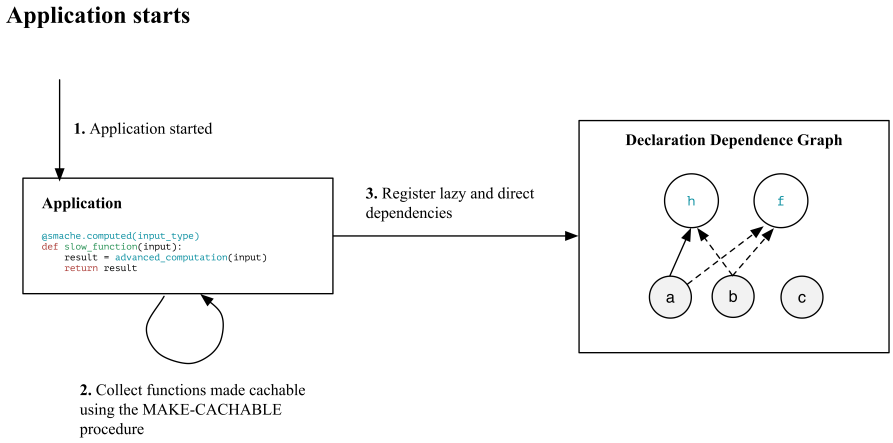


Figure 5.6: The flow in which lazy and direct dependencies are registered from the declarations

need to be updated. A cached object is considered stale when its underlying data has been updated, which happens during the request from a client that involves updating, inserting or deleting data in the primary storage. To be able to react to these events, Smache subscribes to callbacks from the database wrapper. When the database transaction succeeds the database wrapper will notify Smache with the id and type [1] of the manipulated entity. This notification will trigger cache invalidation through Smache. Smache receives the type and id of the updated entity, and runs the algorithm described on figure 5.8.

Since the process of finding keys for depending objects involves multiple network calls to the primary storage, the invalidation propagation must be asynchronous to avoid a performance degrade for requests that involves updating underlying data.

5.4.1 Timestamp Invalidation

To invalidate correctly, the technique for invalidating cached objects require the following liveness and safety property:

[1] Type is another word for the relation in a relational database and a collection in a document-oriented database

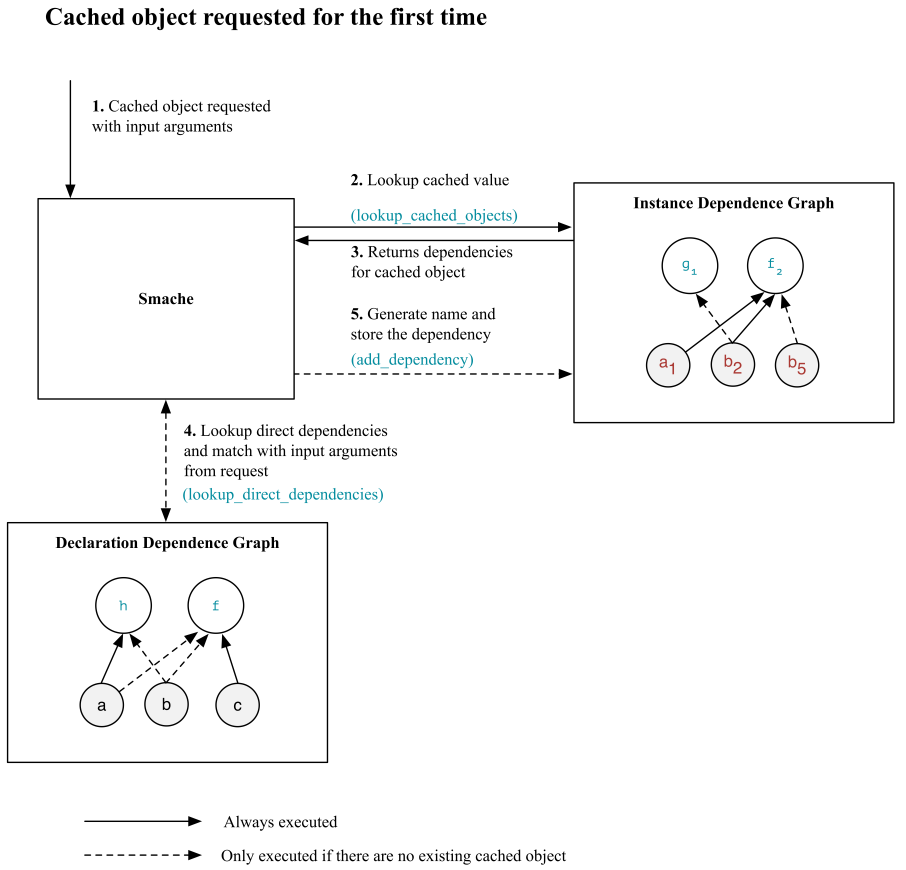
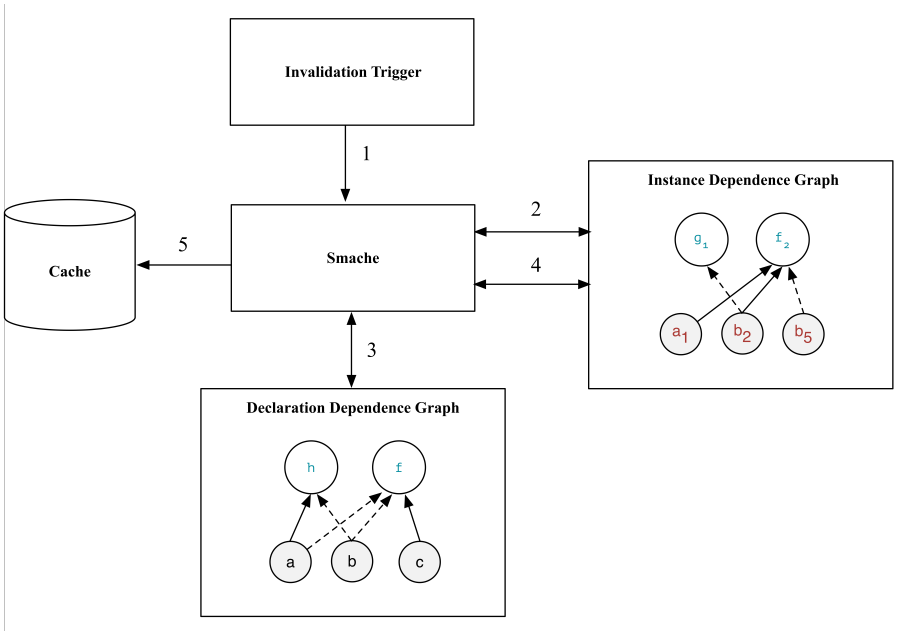


Figure 5.7: The flow in which cached object instances are accessed when they are accessed the first time

- **Safety**: *The value representing a fresh cached object must be based on the newest version of the underlying data*
- **Liveness**: *A cached object with a stale value must eventually be invalidated*

In trigger-based invalidation where the name of a cached object is considered static we need additional information to evaluate whether a given cached object is fresh. In a naive invalidation technique, we can use a boolean value that indicates whether or not the cached value is fresh, which would be correct in an



1. The invalidation trigger notifies Smache about an update to underlying data with type and id of entity.
2. Smache queries the IDG using `lookup_cached_object` with the updated entity id to find instance dependencies>
3. Smache queries the DDG using `lookup_lazy_dependencies` with the update entity, and evaluates the procedures for the returned lazy dependencies.
4. Smache queries the IDG using `lookup_cached_object` with the set of entities returned from the lazy dependency procedures.
5. The cached objects from instance and lazy dependencies are invalidated using *timestamp invalidation*

Figure 5.8: The invalidation propagation algorithm for Smache.

environment where invalidations are processed sequentially. When an invalidation is triggered the value is set to `false` and when the cached object has been updated it would be set to `true`. The problem here is that in an environment with multiple application servers, the technique would be prone to race conditions as illustrated on figure 3.4. Since the given cached object would be marked

as fresh even though the value of the object in reality is stale, the solution would contradict the safety requirement.

One solution to solve this problem is to lock the invalidation mechanism such that the invalidation indication cannot be changed during an update. By using a lock the technique gets atomic updates across invalidation and updates and thereby achieves safety, but it also means that when another process wants to invalidate, it has to wait until the lock has been released. If the waiting process is a web server serving a user, the user would have to wait for the computation to finish.

To accommodate the liveness property and avoid user's having to wait for invalidation, we use a solution suggested in [CID99] that uses a form of logical timestamps to represent the version of a cached object. In this technique the cache system keeps track of the number of times a given cached object has been invalidated, which we denote $num\_of\_updates$. The number starts with 0 and is incremented every time a cached object's underlying data changes. We also keep a timestamp representing the number of invalidations for the given cached object before the computation that wrote the last value, which we denote $last\_update\_timestamp$.

When the cached object is recomputed, the $num\_of\_updates$ is fetched and set as the timestamp of the current computation denoted $current\_computation\_timestamp$. When the computation returns the new value for the cached object, we write update the cached object unless it has been updated during the computation. We can do this using the following algorithm:

1. Fetch the value for $num\_of\_updates$ again denoted $latest\_num\_of\_updates$.
2. If $latest\_num\_of\_updates < current\_computation\_timestamp$ then we update $last\_update\_timestamp$ to be equal to $current\_computation\_timestamp$ and update the cached value. Else we do nothing.

We are able to evaluate the freshness of a cached object as following:

- **A cached object is fresh** when $num\_of\_updates = last\_update\_timestamp$

To avoid race conditions the execution of this algorithm need to be executed in a transaction such that if $num\_of\_updates$ is incremented between step 1 and 2 then the algorithm is aborted. We assume that the execution of a

computation at a given timestamp always yields the same result, which means we can retry the update algorithm with the same result. In other words the algorithm is considered idempotent, and we can therefore retry the update algorithm until it has finished successfully. Alternatively we could lock all the values while executing the algorithm, but this means we will block the incremental of $num\_of\_updates$ that is responsible for invalidating, which we would like to be fast. Furthermore the algorithm require the incremental of $num\_of\_updates$ to be atomic.

5.4.1.1 Proof

Since the technique has not been proved for correctness in [CID99] and because the correctness of Smache relies on this technique, we will give a proof of correctness for timestamp invalidation by proving the liveness and safety requirements described above. In these proves we assume that the invalidation and update algorithms are executed atomically.

Safety

We will prove *safety* using contradiction by *assuming we have a cached object considered fresh that holds a stale value stored in the cache.*

1. We assume that $num\_of\_updates = t_i$ and since we know that a cached object is considered stale when $num\_of\_updates = last\_update\_timestamp$, we also have that $last\_update\_timestamp = t_i$
2. For this to happen, a computation f_i starting when $num\_of\_updates$ was equal to t_i ended up updating the cached value to be v_i and $last\_update\_timestamp$ to be t_i.
3. Since v_i by definition is a fresh value, because the value of $num\_of\_updates$ has not changed during the execution of f_i, the value of the cached object can only become stale in on of the following cases:
   a) Another computation f_j updated the cached object with a stale value after the computation from step 2 updated the cached object.
   b) The underlying data was updated during the computation of f_i.
4. If the computation f_j resulted in a stale value it must have been started at a point where $num\_of\_updates$ was equal to t_j, where $t_j < t_i$. This would contradict the algorithm since f_j would not update the cached object with its stale value since $num\_of\_updates$ must have been equal to t_i after the update from f_i and since $t_i > t_j$.

5. If the underlying data was updated after the computation was started when $num\_of\_updates$ was equal to t_i then $num\_of\_updates$ must have been updated to be t_k, where $t_k > t_i$ after which we have a contradiction of the original assumption since $last\_update\_timestamp = t_i$ and the cached object is therefore not fresh since $num\_of\_updates \neq last\_update\_timestamp$ since $t_k \neq t_i$.

Liveness

The value of a cached object becomes stale, when the underlying data has been updated. We therefore need to prove that an update to underlying data affecting a cached object considered fresh, eventually results in the cached object becoming stale. We will also prove this by contradiction using the *assumption that an update to underlying data affecting a fresh cached object, resulted in the object having a stale value and still be considered fresh.*

1. We know that the algorithm eventually triggers an invalidation that will increment $num\_of\_updates$ and we know that $last\_update\_timestamp$ cannot be set to a value larger than $num\_of\_updates$.
2. That is we have a moment immediately after $num\_of\_updates$ was incremented, where $num\_of\_updates > last\_update\_timestamp$.
3. The only case in which the cached object can be considered fresh and still have a stale value after this moment of staleness, would be if the execution of a cached object update $last\_update\_timestamp$ being set equal to $num\_of\_updates$ without updating the value of the cached object. This contradicts the algorithm of timestamp invalidation, because we only update $last\_update\_timestamp$ while updating the cached value.

5.4.2 Data Maintained by The Cache

To support timestamp invalidation we will extend the representation of a cached object with the following values:

- *value*: The result of the cached function related to the cached object
- $num\_of\_updates$: The number of updates the given cached object is aware about
- $last\_update\_timestamp$: The timestamp corresponding to the computation of the current *value*

By representing the timestamps in the same object as the value we can more easily implement concurrency control mechanisms such as a lock or transactions as described above. We could represent the timestamps in another database, but this would mean we had to support distributed transactions across the cache database and the database with timestamps. We could also represent the timestamps in seperate objects than the value, but this would make it harder to shard the cached objects across multiple cache servers since we need to ensure that the cached objects are on the same server as the value to avoid distributed locking.

5.4.3 Database Wrapper Triggers

We've already discussed existing approaches for triggering invalidation in section 3.6.1 with a comparison on figure 3.1. The triggers sent directly from the database or through a database sniffer have an advantage of capturing all changes made to the database - also those not sent through the application. The disadvantage of those techniques are that they are coupled to the database technology used. Instead of using a technique that depends on the exact database technology, Smache uses the database wrapper to trigger invalidations. The database wrapper makes it easier to change the implementation and thereby the database technology, which means the caching system becomes more flexible and easier to use in applications using different technologies as well as to change the technologies in the future. Furthermore since the triggers are intercepted through function callbacks in the web server process, the system doesn't need an external process to convert triggers from the database or sniffer into invalidations.

5.5 Implementing Automatic Invalidation

The Smache library implements automatic invalidation as described in the design above except for a few details such as the exact data structures of the dependency graphs, which have been replaced by similar representations to simplify the implementation. In this section we explain how the implementation of cachable functions (section 4.3) is extended to support automatic invalidation. We will start by explaining how the IDG and DDG are implemented (section 5.5.1) followed by a description on how we implement transactions to update cached objects with timestamp invalidation (section 5.5.2). Finally, we explain how we implement asynchronous invalidation to avoid affecting the performance of requests updating underlying data (section 5.5.3).

5.5.1 Dependency Graphs

The dependency graphs are used by the update procedure to derive affected cached objects to be invalidated.

The DDG is static and will not change after the application starts and we can therefore easily duplicate the data across all the web server without implementing any replication. The DDG is therefore represented using objects in the application that can be queried by calling the methods of the DDG repository.

On the other hand, the IDG is dynamic in the lifetime of the application and will therefore be represented in a separate process. Optimally the IDG should be represented using hash tables, but to simplify the architecture, the data structure is implemented using the cache database that is assumed to be a common cache server supporting `LOOKUP` and `STORE` as described in section 2.2. More specifically the current implementation uses *Redis* for storing dependencies. In this implementation the name corresponds to keys in the hash table and the linked lists are represented in the content by a serialized list.

Alternatively we could represent the dependency graphs using a cache manager process, but this would just introduce an additional potential bottleneck as well as a single point of failure. In our solution we limit the number of possible process failures by integrating the dependency graph into existing components of the architecture.

5.5.2 Update Transactions

To implement timestamp invalidation we need to implement concurrency control to ensure that the timestamp invalidation is implemented correctly. In the version of Smache implemented during this thesis, we use *Redis* [2] as the cache database. Redis have support for simple transactions using a combination of the `WATCH`-command and the `MULTI`-command. The `WATCH`-command is executed with a key, which tells Redis that a given transaction should fail if the specificed key is modified during the transaction. The `MULTI`-command allows to specify multiple commands in a single atomic action. Code snippet 5.1 is a simplified version of the code that implements the update procedure with timestamp invalidation. It uses the Python Redis [McC] library to communicate with the Redis database. When it invokes the `WATCH`-command the client will request Redis to start a transaction. Afterwards we build a `MULTI`-command that will update the cached object if it is newer and invokes the `EXECUTE`-command.

[2] http://redis.io/

The Redis-client will send the command and if the transaction fails it will raise a `WatchError`-exception indicating the key has been modified, after which we retry the transaction with the same values.

5.5.3 Asynchronous Invalidation

Recall figure 3.7 illustrating the flow where changes to the data in the primary storage system triggers a notification to the caching system, which invalidates affected cache objects. If the notification in step 3 was invoked synchronously then the performance of step 4 and 5 would affect the response time of the request and thereby affect the user experience of the use case involving the given request. In our case we cannot ensure to retrieve dependencies fast since we have lazy dependencies that must be evaluated with requests to the primary storage. We will therefore implement the trigger-step asynchronously such that requests involving updates to underlying data are not affected by Smache.

If the notification is delivered more than once we will just achieve a worse cache hit rate and affect performance, but it will not affect the integrity of the cache object. We must therefore monitor and ensure At-Least-Once semantics of the notification. With most technologies the simplest solution for implementing asynchronous behaviour is to start a new thread that executes the invalidation. This way the asynchronous behaviour is controlled by the web server and we don't have to introduce new components to the architecture. Although this would be the preferred solution the Python language does not implement real concurrent behaviour, because it implements a Global Interpreter Lock that prevents multiple threads to execute the same bytecode at once. To implement concurrent invalidations in Python we will therefore use the concept of *Background Jobs*, where the notifications are represented by a *job* that is pushed into a queue. To execute the jobs we have multiple *workers*, which are basically processes that also contains a version of the source code for the application. When there are jobs in the queue, a worker will pull the given job from the queue and execute it. Figure 5.10 illustrates how background workers are used to invalidate cache object asynchronously. To implement this behaviour we need to add application workers to the architecture as illustrated on figure 5.9.

5.6 Summary

In this chapter we suggest an extension to cachable functions that provides automatic invalidation. The caching system uses the dependency information

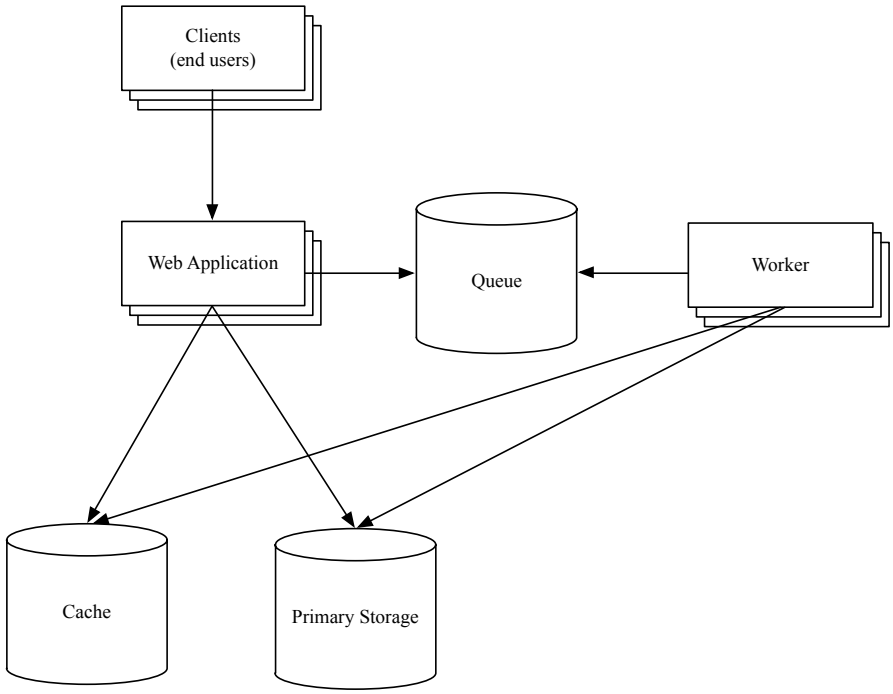


Figure 5.9: The architecture required by a web application that uses Smache with automatic invalidation.

given as arguments, when a function is made cachable, to derive the dependency strategy for how and when to invalidate the related cached object. To provide efficient and fast invalidation, the caching system uses extensions of the Simple Object Dependence Graph data structure called Declaration Dependence Graph (DDG) and Instance Dependence Graph (IDG), to access the dependency information fast. The caching system registers static dependency information stored in the DDG when the application starts and dynamic dependency information in the IDG when a cached function is called. When underlying data are changed, the caching system will query the dependency data structures to find the cached object instances affected by the change, which are then invalidated using timestamp invalidation. To verify correctness of the solution, we proved safety and liveness of the timestamp invalidation technique. The automatic invalidation was implemented in Python and used Redis to represent dependency graphs and to facilitate update transactions required for timestamp invalidation. To avoid affecting the performance of existing update operations in the application, the invalidation are performed asynchronously using the *Background Jobs*.

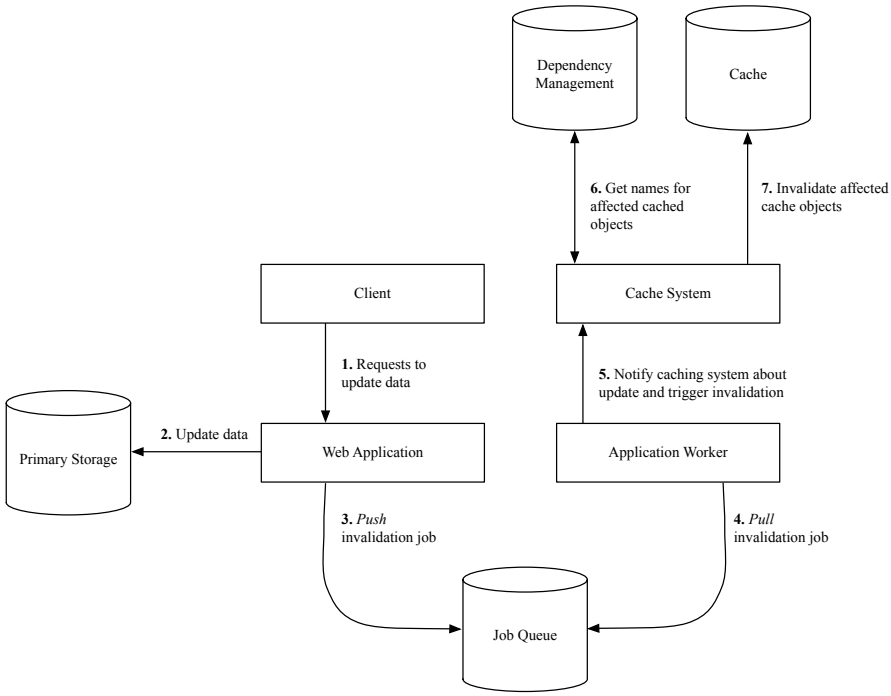


Figure 5.10: How background workers are used do perform invalidation asynchronously.

Code Snippet 5.1: The code for implementing timestamp invalidation in Python. RedisConnection represents an object communicating with the (Redis) cache database. CacheStore represents the object that communicates with the cache database.

```
import time
import random
import json

retries = 25

def store(key, value, computation_timestamp):
    # Use a pipeline to execute more than one command
    pipeline = RedisConnection.pipeline()
    store_retry(key, value, computation_timestamp, pipe, retries)

def store_with_retry(key, value, computation_timestamp, pipe, retries):
    try:
        if retries > 0:
            # Execute WATCH command
            pipe.watch(key)

            # Fetch the current cache object
            current_cache_object = CacheStore.fetch(key)

            # Update the value and last_update_timestamp if it has
            # a newer timestamp than the current_cache_object
            if has_newer_timestamp(cache_object, current_cache_object):
                pipe.multi()
                pipe.hset(key, 'value', json.dumps(value))
                pipe.hset(key, 'last_update_timestamp', computation_timestamp)
                pipe.execute()

    except WatchError:
        # This is executed if the key was modified by another process
        # during the transaction. We wait for a random number of seconds
        # and retry
        time.sleep(random.random())
        store_entry(key, value, computation_timestamp, retries - 1)

def has_newer_timestamp(cache_object, computation_timestamp):
    """
    Checks if the timestamp from the computation is newer than the
    one for the current stored cache object
    """
    return cache_object.num_of_updates is None or \
        computation_timestamp > cache_object.num_of_updates
```

Chapter 6

Data Update Propagation

Until now we've explained how to build a caching system that is able to cache the result of functions, where the result is invalidated automatically when its underlying data changes. This solution makes it easier to manage cache invalidation and locate cached values, but after a result has been invalidated the user has to wait for the value to be recomputed, which can be critical for the user experience in cases where the computation time is too long.

In this chapter we will extend the current solution with write-through invalidation using a data update propagation (DUP) algorithm that schedules updates for invalidated cache objects addressing the second challenge of the problem description 1.1. We will start by covering existing approaches for data update propagation (section 6.1), followed by an analysis of the problems involved with concurrent write-through updates (section 6.2). Based on the knowledge from these sections we will describe the design (section 6.3) and implementation (section 6.4).

6.1 Existing Data Update Propagation Approaches

In the approach suggested by Jim Challenger et.al. [CDI98, CID99, CIW+00], the DUP algorithm is based on invalidation using an Object Dependence Graph as described in section 5.1. When underlying data changes all depending cached objects are scheduled for update. Because some of the cached objects depend on each other, the scheduling have to enforce some ordering to compute the values correctly. If a cached object o_2 that depends on another cached object o_1 were updated first, then it would be based on an old version of o_1, which means it results in a stale value and thereby violates safety. To solve this problem, the system updates all cached objects in a topological order, which ensures that o_2 is ordered after o_1 since it depends on o_1. One limitations of this technique is that it only works if the object dependence graph has no cycles i.e. it is an *Directed Asyclic Graph*. Another limitation is that when there is an order of the jobs then they must be synchronized when executed in parallel.

Labrinidis et.al. [LR01] suggests an algorithm that schedules cache updates prioritized by a Quality of Data measurement, which they define as the probability that a request results in a fresh value from the cache. The algorithm is then designed to maximize the overall Quality of Data by continuously evaluating the measurements and scheduling the updates in a prioritized order.

6.2 Concurrency Analysis of Write-Through Invalidations

Since the cached objects are accessed by multiple processes they become a shared resource, and we therefore have to consider concurrency challenges such as potential race conditions. Figure 6.1 illustrates a potential race condition that can occur, when multiple processes processes tries to update the same cached objects at the same time. In this case a race condition affects the correctness of the system such that a given cached object is marked as fresh even though it's value is stale, and thereby violates the safety requirement.

6.3 Design of the DUP Algorithm

The existing solution described in section 6.1 assumes that a given cached computation is only accessed by one process at the same time. These techniques

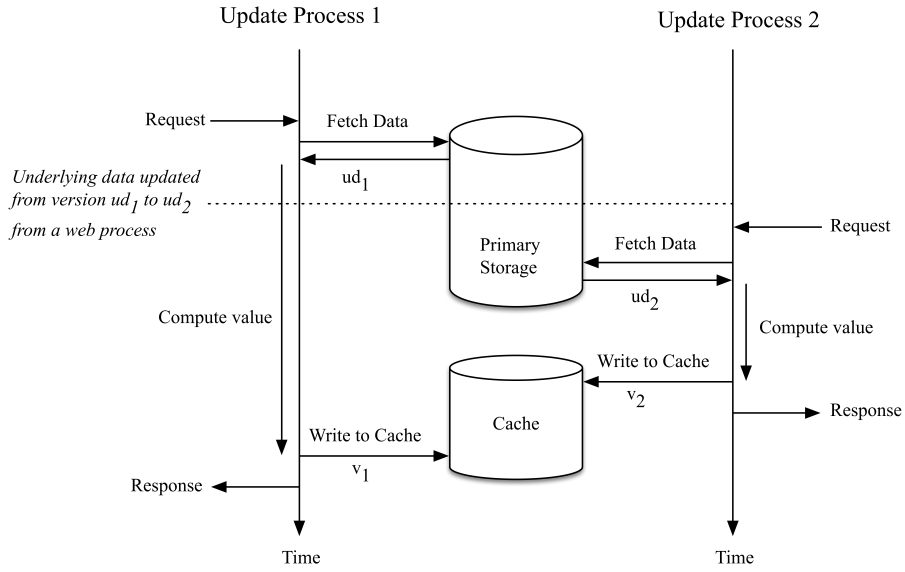


Figure 6.1: Showing how two concurrent caching updates from two different application servers results in an inconsistent state. We see that even though the request from *Update Process 2* are based on data older than *Update Process 1* it gets to write.

have the advantage that they avoid race conditions, but in order to optimize with concurrency, they have to synchronize the update to avoid executing the same computation at the same time. The optimizations are then achieved using scheduling algorithms that evaluates the update jobs and prioritize based on some metric such as the freshness or computation time. These solutions enforce an order of execution, which means some jobs have to wait for others to finish. The only way to optimize the execution of these jobs is to acquire faster CPU's, which is expensive compared to buying more CPU's.

Instead of using advanced scheduling algorithms to achieve a high throughput we suggest a solution that allows concurrent updates, which means we can scale our update execution by adding more processes and thereby be able to execute more jobs at the same time.

To allow concurrent updates we use Timestamp Invalidation as already explained in section 5.4.1. Timestamp Invalidation ensures that a given computation does not overwrite the value of a cached object if the computation is based on a newer version of underlying data. Figure 6.2 shows how the timestamp

invalidation handles the concurrency challenge described in section 6.2.

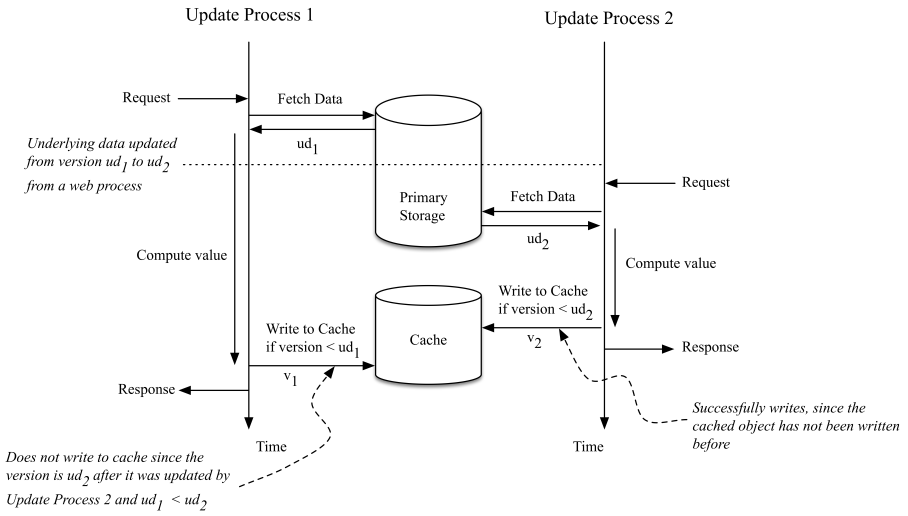


Figure 6.2: How Invalidation Timestamps fixes the concurrency problem described in figure 6.1.

Since invalidation timestamps ensures the integrity of the cached objects we are allowed to scale horizontally and thereby execute as many jobs as there are processes available. Beside updating the cached objects concurrently, the algorithm will only schedule new update jobs if a similar job has not already been scheduled. Furthermore when a scheduled update job is processed it will only execute the computation if the cached object is stale.

If a failure happens during the update of a cached object, the algorithm will retry the job by re-scheduling the job. This can be done without additional control mechanisms since timestamp invalidation only require At-Least-Once semantics, which means it allows executing the same job twice without affecting correctness.

6.4 Implementation of the DUP Algorithm

Smache implements the data update propagation described above as an extension of the automatic invalidation implementation. After a set of cached objects have been invalidated as illustrated on figure 5.10, Smache schedules the same

cached objects to be updated unless an update job for the given cached object already exists.

To be able to work on multiple jobs at the same time, the updates are scheduled asynchronously using the same concurrency model as with concurrent invalidation described in section 5.5.3 - using background jobs. A cached object is scheduled to be updated by pushing an update job with the name of the cached object as seen on figure 5.10. The background workers will pull the update jobs off the queue and update the cached object unless it is already fresh. Smache updates the cached object by deserializing the name of the cached object using an implementation of `DESERIALIZED-FUN` as seen in code snippet 4.2, which results in a reference to the cached function and the arguments identifying the cached object. The cached function is then executed with the given arguments and updated in a transaction using timestamp invalidation as described in section 5.5.2. The implementation of this procedure is seen in code snippet 6.1.

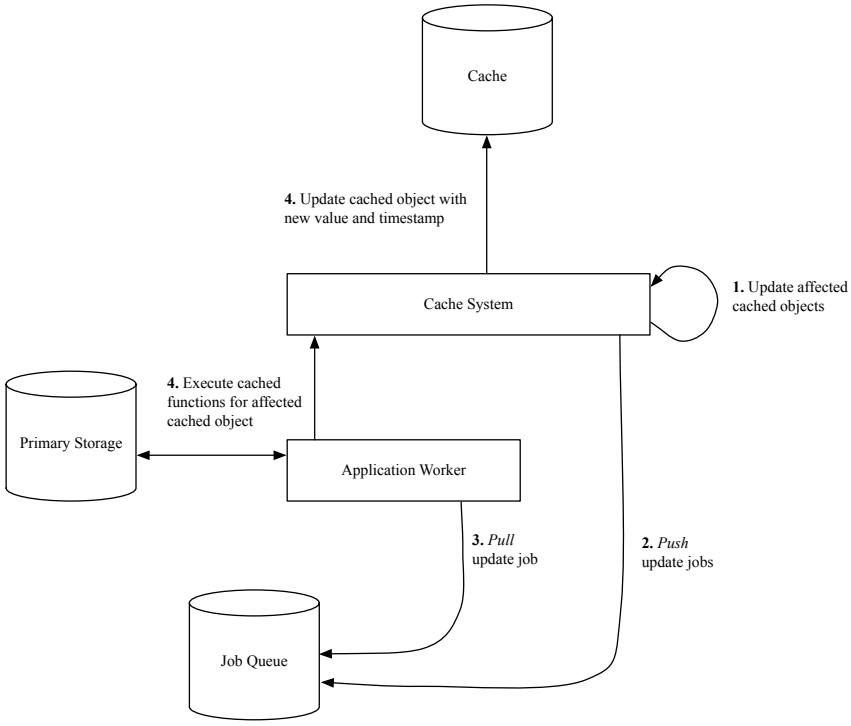


Figure 6.3: How Smache schedules cached objects to be updated using background workers

Since we use the same background workers as with automatic invalidation but

Code Snippet 6.1: The code for updating a cached object identified by the key (name).

```
def update_cached_object(key):
    if not CacheDB.is_cached_object_fresh(key):
        # Deserialize and execute cached function
        cached_function, arguments = deserialized_fun(key)
        computed_value = cached_function(*arguments)

        # Store the new value unless it has been
        # updated by another update process
        computation_timestamp = CacheDB.last_update_timestamp(key)
        CacheDB.store(key, computed_value, computation_timestamp)
```

with different procedures, the DUP algorithm does not require additional components for the architecture seen on figure 5.9.

6.5 Discussion on Fault-Tolerance

In the current architecture described on figure 5.9 we assume the web serves and background workers to be redundant, such that if a process fails in some way (e.g. due to network failures), another process is ready to do the task. We do not make any assumptions about the primary storage, so the level of fault-tolerance depends on the setup. The last two components - the queue and the cache database are not redundant.

The queue could fail if there is a network failure such that the web application cannot connect to the queue server. In this case the caching system would not be able to invalidate or schedule updates, which results in invalid freshness evaluation and no cache updates, leading to two major problems. At first we lose the invalidation and update notifications, such that we are not able to recover and restore the application into a consistent state. Secondly, the web application could potentially serve inconsistent cache values to the clients.

We can solve the first problem by having a local proxy queue in the web application that receives the notifications and forwards them to the distributed queue. If a network failure happened, the local queue would still have stored the messages and when the connection is restored, it can resend its messages. The other problem can only be solved by ensuring that we do not serve any

cached values, since we cannot be sure if they are consistent. This is an appropriate fallback, but if we recomputed every cached value instead of serving stale values, the application could end up crashing due to high load on the servers.

An advantage of having the queue is that the invalidation and update procedures are executed in isolation. This means if an error or failure happens during the invalidation or update process, then it will not affect the web server behaviour and due to the nature of background jobs, we will be able to retry the procedures easily by rescheduling failed procedures. We can do this without implementing any protocol measures, because the invalidation and updates are designed to be *At-Least-Once* semantics i.e. they must be ensured to be send at least once, but if they are received more than once then it will not affect correctness.

The cache server is also not designed to be redundant since timestamp invalidations require transactions for a given key. We could implement distributed transactions using distributed locking as explained in the Redis documentation [Docc], but this solution requires all cache servers to be available all the time to ensure consensus across all the cache servers. Although the cache servers cannot be reduntant, we are still able to scale horizontally by using the sharding technique, where the cached objects are distributed across nodes, which is possible because there is no need for interaction between the cached objects.

If the cache server fails in an setup without redundant cache servers, the application would not be able to evaluate freshness or fetch cached values. This should be handled as with failures to the queue, where we could either make the application fail if we want to ensure consistency or just compute the functions normally as before caching was introduced. The appropriate fallback here depends on the use case, which means it could be a declaration made for each cached function whether or not they should be computed on cache failure.

6.6 Summary

In this chapter automatic invalidation was extended to have write-through invalidation such that the cached objects are updated as they are invalidated instead of updating as they are requested. To do this correctly we introduced a Data Update Propagation (DUP) algorithm that uses timestamp invalidation to allow updates to be executed concurrently instead of using a scheduling algorithm. The algorithm was implemented in Python with background workers, which executes the update procedures concurrently with possibility for retries. The solution does not provide a maximum level of fault-tolerance, but the chapter discusses how the solution can be extended with additional measures to achieve

a higher level of fault-tolerance.

Chapter 7

Tests and Evaluation

Throughout this project Smache has been implemented in Python and integrated into the Peergrade.io-platform. Instructions for where to find and use the library is available in appendix C. The library is also open sourced on GitHub available on the following URL:

https://github.com/anderslime/smache

Besides the Python implementation of Smache, the source code also includes automated tests with a test coverage of 97% and a small benchmark framework to execute and plot the results of performance tests. The automated tests are automatically verified by a continuous integration application, Travis, that runs the full test suite when new versions of the code are merged. The results of the automated tests are available at https://travis-ci.org/anderslime/smache.

The rest of this chapter will lay out results of experiments made to test the assumptions about the solution. The experiments are performed on a MacBook Pro (Retina, 15-inch, Early 2013) with an 2.4 GHz Intel Core i7 (4 cores) processor and 8 GB 1600 MHz DDR3 RAM. The evaluation seeks to answer the following questions:

- What changes are required to modify existing web application to use

Smache? (section 7.1)

- What is the performance impact for procedures updating data in the primary storage, when cachable functions are introduces with Smache? (section 7.2)
- How efficient are Smache updating cached objects? (section 7.3)

The chapter will end with an evaluation of the system based on the test results and the initial requirements 7.4.

7.1 Changes Required to Use Smache

Smache was integrated into the Peergrade.io platform, which is an existing web application with a single code base deployed to multiple web servers. To test the adaptability we applied the library to functions of different granularities such as functions returning a single value, functions returning a data structure and functions returning the HTML served to the user. From this integrations we experienced an easy integration, but some of the functions had to be split up to to cache optimally.

For example some of the functions had statistical calculations that iterated using `for`-loops and made time consuming calculations in each iteration as seen in code snippet 7.1, which were not able to be cached individually. To make this function optimally cachable, we would have to extract it into a method such as in the running example in code snippet 1.1. The running example are then made cachable by adding cachable declaration using Python-decorators as in code snippet 4.1.

The HTTP-responses were not directly cachable, since they work on request-objects specific to the client making the request. In order to cache the responses we extracted the code that generated the HTML-fragment served as the HTTP-body.

In order to use Smache, the system also require additional components added to the architecture. To achieve the guarantees described in this thesis, the system must add the following components to the architecture, also illustrated on figure 7.1:

- Cache database to store the cached values and metadata

- Background Workers with a Queue-system to have asynchronous invalidation and updates

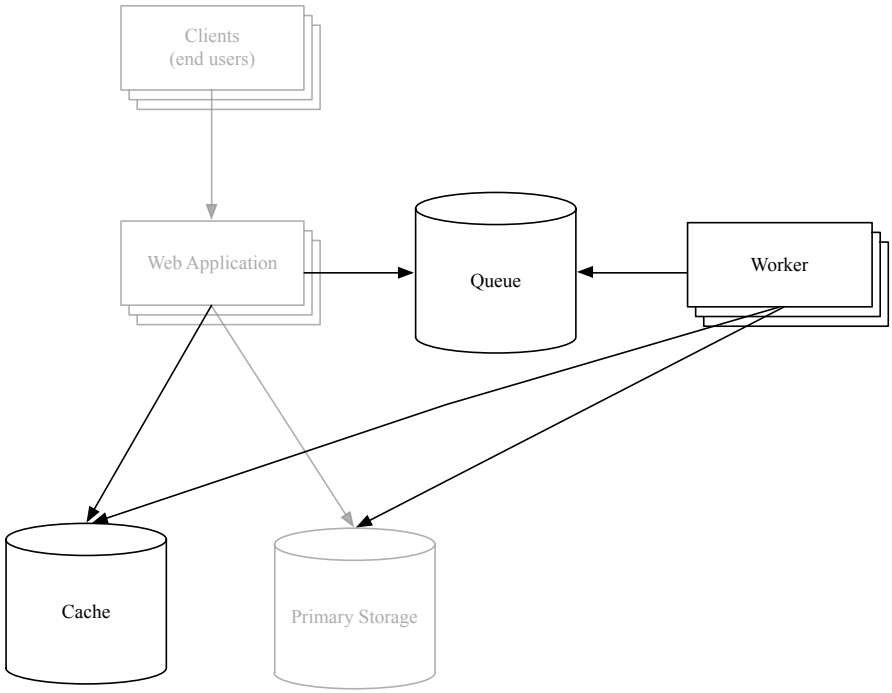


Figure 7.1: Additional architectural components required to use Smache compared to a normal web application including clients, web application servers and a primary storage.

Even though this is optimal, Smache has been implemented such that it is easy leave them out. The cache database can for example be replaced by the primary storage that is already a part of the system. The background workers can also be left out and replaced by synchronous invalidation and updates that are executed in the same process as the web servers, but this would affect the performance of the web applications as seen in the results in section 7.2.

When Smache was introduced into Peergrade.io, the components required were already introduced, which meant Smache could be introduced without additional architectural components. The application already used background workers for optimizing procedures such as sending e-mails. Furthermore it already used a basic caching approach with a cache server.

Code Snippet 7.1: Code where Smache can cache the course score, but not the individual participant scores.

```
def course_score(course)
    participants = Database.find_all_participants_in_course(course)
    total_score = 0
    for participant in participants:
        grades = Database.find_all_grades_for_participant(participant)
        total_score += numpy.advanced_statistical_method(participant)
    return total_score / len(participants)
```

7.2 Performance Impact of Existing Operations

One of the requirements of the system is that the performance of existing operations should not be affected negatively by introducing the caching system. On the requests that changes underlying data, the application will notify and execute invalidation in the same process, which could affect the response time of the update request. The performance of invalidation have been evaluated using a test that uses both *direct dependencies* and *lazy dependencies*.

The results of this test, visualized on figure 7.2, indicates how a synchronous and asynchronous write-through mechanism impacts the time taken to update underlying data. If we consider the asynchronous mechanism used by Smache, it introduces a small constant overhead of around 2 ms as seen on the exact results shown on table 7.1. The synchronous mechanism introduces an overhead that increases with the execution time of the cached function, which can become critical for the user experience with long running computations.

Function duration	Without caching	Smache (async.)	Synchronous
50 ms	6.2 ms (1.0x)	7.9 ms (1.27x)	143.2 ms (23.0x)
100 ms	6.3 ms (1.0x)	7.9 ms (1.25x)	146.9 ms (23.4x)
200 ms	6.4 ms (1.0x)	8.1 ms (1.26x)	168.2 ms (26.4x)
400 ms	6.3 ms (1.0x)	8.0 ms (1.25x)	207.1 ms (31.2x)

Table 7.1: The time taken updating underlying data that affects a single cached object for an application using Smache compared to a system using synchronous write-through invalidation.

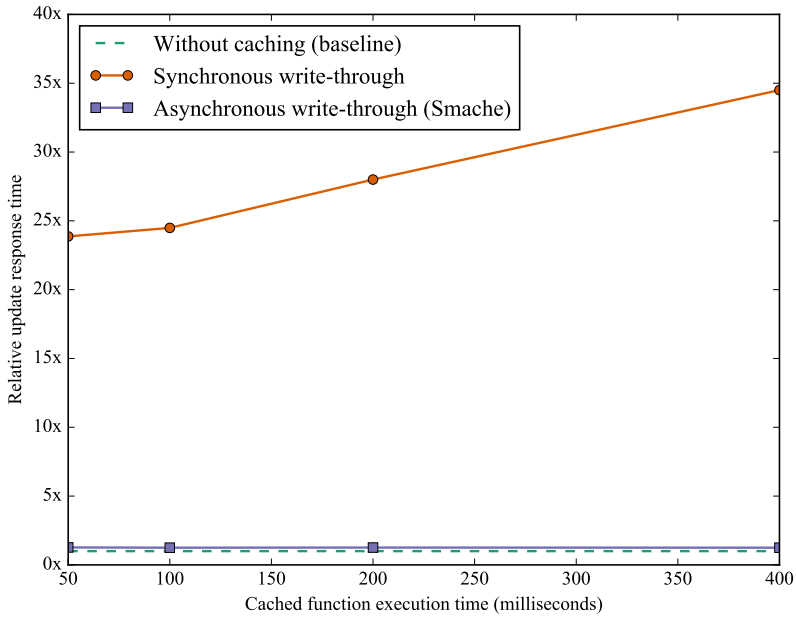


Figure 7.2: Impact of introducing synchronous and asynchronous write-through in a case, where the update affects a single cached object instance with variable execution time.

7.3 Update Throughput

To evaluate the performance and scalability of the concurrency used in the data update propagation, we will measure the update throughput while increasing the number of workers, where the number of workers is the amount updates the system is able to process at the same time. While it would be possible to measure against examples in the context of the Peergrade.io, we have chosen to measure against patterns of cached functions to make the experiments easy to reproduce for future research. The tests will be performed against a use case, where many cached functions depends on a single underlying data resource (section 7.3.1) and afterwards a case where multiple cached functions depends on each other in a nested hierarchy (section 7.3.2).

7.3.1 Many Cached Object Instances

In this use case we will test a use case, where multiple instances of the same cached function depends on the same underlying data such that when the underlying data updates then all the cached objects instances for the cached function must be updated. The use case is illustrated on figure 7.3.

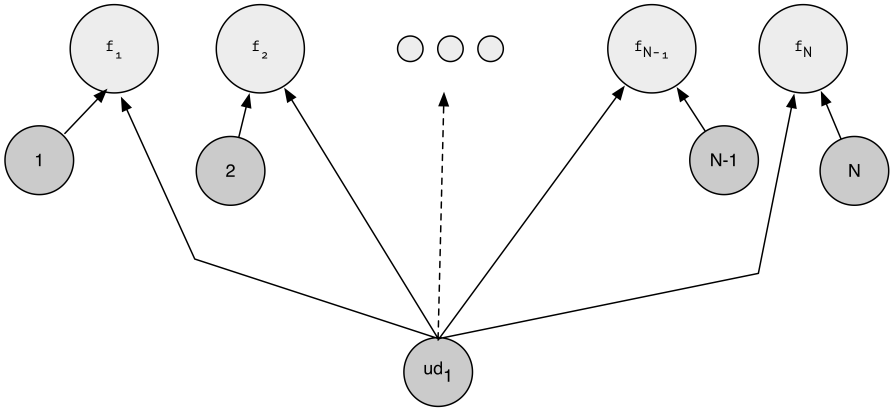


Figure 7.3: Illustration of the dependencies of the cached function for the "Many Cached Functions" test case

To simulate slow functions the tests use the `sleep`-method from Python's `time`-module such that each function uses 400 ms to execute.

The results of the tests are visualized figure 7.4 by a graph showing the number of updates the system can process per second (throughput) for different amount of workers. This graph shows how concurrency affects the throughput of the cache system, where we see a significant increase in throughput every time the number of workers are increased. We also see that the impact of adding workers are becomes higher when there are more cached objects even though it also has an impact for small amounts (around 10 cached objects).

7.3.2 Nested Cached Functions

To evaluate the performance on nested cached functions, we apply the tests on a setup, where multiple cached functions depend on each other as illustrated on figure 7.5.

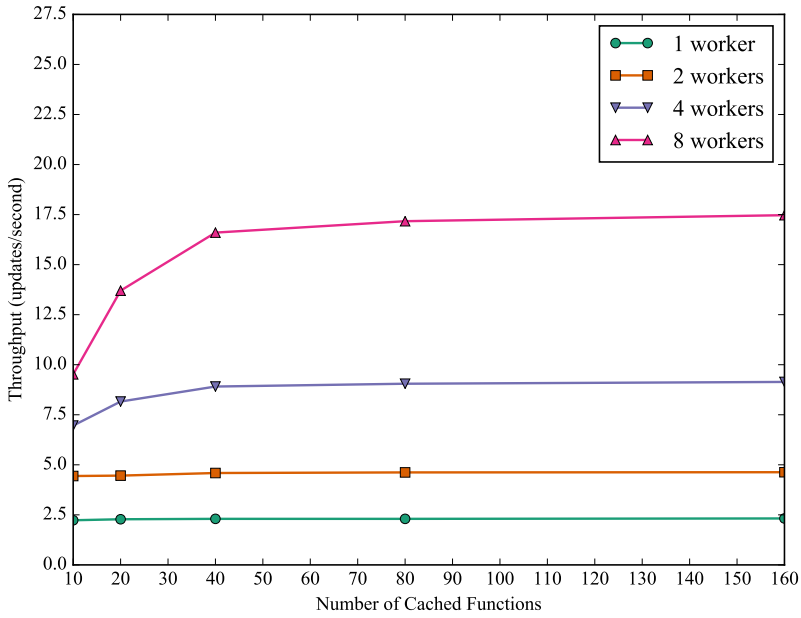


Figure 7.4: How the system scales with many cached functions depending on the same underlying data while the number of workers is increased

Compared to the impact on the use case with many cached functions (section 7.3.1), the results on figure 7.6 shows that the impact of concurrent updates are not as high and that the impact highly depends on the depth of the cached functions.

A reason why the use case with nested cached functions are interesting is because it cannot be parallelized by scheduling algorithms that assumes a given cached function only can be computed once at the same time. These algorithms would process the cached functions in this test case sequentially, which corresponds to the result of using 1 worker. If we compare the throughput of using multiple workers to the throughput of using a single worker as seen on table 7.2, we see that there is still a gain in throughput from processing multiple functions at once.

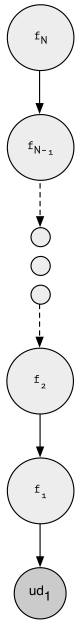


Figure 7.5: Illustration of the dependencies of the cached function for the "Nested Cached Functions" test case

Cached functions	Sequentially	Concurrently	Improvement
2	0.80 updates/sec	1.57 updates/sec	1.97x
4	0.94 updates/sec	1.88 updates/sec	2.01x
8	0.96 updates/sec	2.43 updates/sec	2.55x

Table 7.2: Improvement of processing nested cached functions concurrently (using 8 workers) compared to sequentially (using 1 worker).

7.4 Evaluation

The final solution, Smache, solves the problem described in section 1.1 by introducing a caching system that provides automatic write-through invalidation to ensure that the cached objects are always served instantly. Smache allows the programmer to define whether a given cached function should serve stale values or recompute a value if it is stale. To relate to existing solutions table 7.3 shows how Smache relates to the cache evaluation criteria (see appendix D for the table including existing approaches).

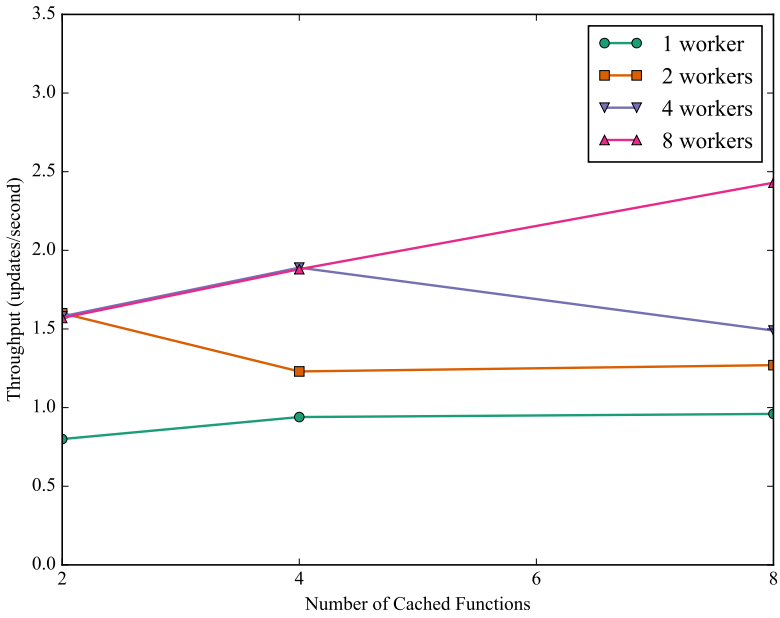


Figure 7.6: How the system scales with nested cached functions while the number of workers is increased

	Consistency	Strict Freshness	Update On Invalidation	Always Immediate Response	No Cache Management	Adaptability
Arbitrary Content						
			...			
Smache	-	-	Yes	Yes	Yes*	Medium
Smache w/o staleness	-	Yes	Yes	-	Yes*	Medium
			...			

*) Dependencies to underlying data must be declared

Table 7.3: How Smache relates to the cache evaluation criteria

In order to make the Smache library adaptable for existing application it has been designed and implemented to be flexible and introduce as few new components as possible. The architectural components are made as extensions such that they easily can be changed or left out if some of the guarantees are not

needed in the given context. To give all guarantees promised during the thesis, Smache requires to web application to support background jobs and have a cache server. The interface for making functions cachable is not fully transparent, but it relies on the developer declaring dependencies. Smache could be extended with a transparent interface, where the dependencies are automatically detected, but this would also affect the flexibility, the control of the programmer and make it more difficult to understand when a given cached function is invalidated. We therefore suggest to use a tool that helps the programmer identify dependencies instead of removing the control or a solution where the programmer is able to declare dependencies optionally. This also relates to the requirement of making the system maintainable and understandable for the developer.

The test results of this chapter indicates that the Smache solution does not introduce performance regressions to the system. The requests that updates underlying data are normally affected when the application uses a synchronous write-through invalidation, but Smache only introduces a small constant overhead of a few milliseconds ($< 3ms$) by invalidating and updating asynchronously. The update propagation algorithm allows concurrent updates that in some cases results in a higher throughput than scheduling algorithms. The algorithm uses the timestamp invalidation technique to ensure correctness such that the updates can be executed concurrently, which means we achieve a simpler system that does not require scheduling algorithms to achieve a high throughput and we are able to scale the throughput by adding more workers. On the downside it also means the algorithm can risk executing the same cached functions more than once at the same time and thereby using more CPU power than necessary. We can therefore say that the system is efficient with relation to the performance of the web application and the update throughput, but not with relation to CPU usage if we compare to some of the existing solutions. This is a trade-off for simple update propagation.

Smache does not control the space usage, which means the presence of the cached values depends on how the cache database controls space. The optimal choice for a cache database is a in-memory database, but since memory is expensive, it could be necessary to change to a disk-based database if the space used becomes too expensive.

The system has been designed with fault-tolerant measures such as isolation and retries of invalidation and updates, but to keep the solution simple, the system does not apply measures to ensure the availability of the cache system. More specifically the design does not implement measures to prevent or recover from faults happening in the queuing system or the cache database, which means they are seen as a single point of failure of the system. To run the application correctly it must be ensure that the queue and cache database are highly available.

Chapter 8

Conclusion and Future Work

Caching is a popular technique for improving the performance and scalability of web applications, but it also relies on the programmer correctly managing the cache and making decisions about the freshness and cache updates. In some cases the cached computations takes a long time to compute, which can be critical or the user experience if the time a user have to wait for the computation exceeds the attention span.

In this thesis we have analyzed the challenges of cache management and presented a set of evaluation criteria to find the appropriate caching technique for a given use case. The evaluation criteria was applied to existing caching solutions used in practical web development and described in research. From the existing solutions we found no technique that solved the problem while meeting the requirements, and the thesis therefore introduces a new application-level caching system, Smache.

Smache presents a programming model that allows the programmer to mark existing functions as cached by declaring the dependencies to underlying data used to compute the function. The cached functions are automatically invalidated and updated when the underlying data changes. Smache uses timestamp invalidation to ensure the integrity of the cached values and represents the dependencies using a variant of the Object Dependence Graph data structure to

find affected cached values efficiently. Smache was implemented in Python and integrated into the Peergrade.io-platform.

Smache makes it easier for the programmer to cache long running computations without affecting the user experience of the end user. It has been designed to be easy to integrate into existing applications with a possibility to cache existing functions without changing the code inside the function and by reusing architectural components used in many web application. Our experiments performed on the Smache library showed that it only introduces a constant overhead for existing operations of the application and the update throughput showed to increase when more updates where processed concurrently. That is, Smache can be scaled horizontally by adding more update workers to increase the update throughput when the number of users rises or when new cached functions are introduces.

To our knowledge, Smache is the first caching system with automatic write-through updates that allows to cache arbitrary computations.

8.1 Future Work

Smache provides a foundation for a flexible and efficient caching library, which can lead to extensions and improvements. Smache is available for open-source and it is therefore possible to contribute to the official Python-implementation of this thesis through GitHub:

https://github.com/anderslime/smache

The following list are our ideas for future projects as extension of this thesis:

- **Transparent Dependency Registration**: The version of Smache described in this thesis relies on the developer identifying the underlying data used for a given computation. Even though it becomes simpler than the traditional invalidation techniques, it would be even easier to cache functions if this task was performed automatically by Smache as in the solution suggested by Dan Ports [Por12].
- **Fault-Tolerant Improvements**: As a trade-off for simplicity, this thesis did not design Smache to be fully fault-tolerant, which means the tolerance of faults are limited in areas of the solution as discussed in section 6.5. In the same section we also suggest improvements to make the solution more

fault-tolerant to e.g. handle failures happening to the queue and failures of the cache database.

- **Hybrid Invalidation**: The focus of this thesis was to cache in the use case with long running computations. Since Smache provides a strong foundation for invalidation and update scheduling, it would be beneficial to extend Smache with other invalidation approaches such as expiration-based invalidation. This could allow for interesting optimizations.

- **Scheduling Optimizations**: Smache uses concurrency to achieve a simple scheduling solution while achieving a high throughput as a trade-off for efficiency. While simplicity is a desirable property, it could be necessary to optimize the scheduling in applications of larger scale.

- **Test on Massive Scale**: Smache was designed for the common web application, which is not considered a large scale compared to e.g. Facebook architectures. It would be interesting to research what is required to use Smache on larger scale.

- **Other Primary Storage Technologies**: Smache has been implemented to support MongoDB, because it is used by Peergrade.io. To make Smache adaptable for more web applications and to test the flexibility of the Smache design, it would therefore be interesting to add adapters for other database technologies such as PostgreSQL and MySQL.

- **Other Programming Languages**: Smache is implemented in Python, also used by Peergrade.io, but to test the flexibility of the design, it would be interesting to implement Smache in other programming languages such as Ruby, Java and C#.

List of Figures

List of Tables

Appendix A

Code Snippet for Trigger-based Invalidation with Asynchronous Update

```
def time_consuming_participant_score(participant):
    return numpy.advanced_statistical_method(participant)

def cache_key_for_participant_score(participant):
    cache_key_components = [
        'cached_participant_score',
        participant.type,
        participant.id
    ]
    return '/'.join(cache_key_components)

def cached_time_consuming_function(participant):
    cache_key = cache_key_for_participant_score(participant)

    if is_fresh_in_cache(cache_key):
        return fetch_from_cache(cache_key)
    else:
        update_cache_async(
```

94 Code Snippet for Trigger-based Invalidation with Asynchronous Update

```
            time_consuming_participant_score,
            participant
          )
          return fetch_from_cache(cache_key)

# Load the participant from the primary storage
participant = ParticipantDB.load_one_from_database

# Call the cached version of the time_consuming_participant_score
# Since there is currently no value in the cache it returns nothing
print cached_participant_score(participant)

# sleep for 5 seconds to wait for the computation to finish
sleep(5)

# This time we get an immediate response since the result is
# cached from the asynchronous update
print cached_participant_score(participant)

# Now we invalidate the cached value
cache_key = cache_key_for_participant_score(participant)
invalidate_cached_value(cache_key)

# This time we serve the same value as when it was called
# previously, but now it is stale
print cached_participant_score(participant)
```

Appendix B

Implementation of Function Serialization and Deserialization

To be able to serialize and deserialize a cached object instance, the Smache library implements the class `FunctionSerializer` with the public method `serialized_fun` and `deserialized_fun`, which respectively serializes a function with input into a string representation and deserializes a string representation into a function name with deserialized arguments.

```
1  import json
2  from functools import reduce
3
4  class FunctionSerializer:
5      seperator_token = '~~~'
6
7      def serialized_fun(self, computed_fun, *args, **kwargs):
8          args = self._serialized_args(args, computed_fun.arg_deps)
9          elements = [json.dumps(computed_fun.id)] + args
10         return reduce(lambda x, y: x + self.seperator_token + y, elements)
11
12     def deserialized_fun(self, fun_key):
```

```
13          elements = fun_key.split(self.seperator_token)
14          return self._fun_name(elements), self._deserialized_args(elements)
15
16      def _fun_name(self, elements):
17          return json.loads(elements[0])
18
19      def _deserialized_args(self, elements):
20          return [json.loads(element) for element in elements[1:]]
21
22      def _serialized_args(self, arguments, arg_types):
23          return [self._serialized_arg(argument, arg_type)
24                  for argument, arg_type in zip(arguments, arg_types)]
25
26      def _serialized_arg(self, argument, arg_type):
27          return json.dumps(arg_type.serialize(argument))
```

To see the function serialize in action, we can inspect the unit test for the function serializer class.

Appendix C

Source Code

The source code is available open-source on the URL:

https://github.com/anderslime/smache

Or available as a downloadable ZIP-file at:

https://github.com/anderslime/smache/raw/master/smache.zip

Python is required to use the library. To run the tests, execute the following command in the terminal:

```
python setup.py test
```

Further instructions on how to use the library is located in the `README.md`-file from code base.

Appendix D

Comparison Of Caching Approaches Including Smache

	Consistency	Strict Freshness	Update On Invalidation	Always Immediate Response	No Cache Management	Adaptability
Arbitrary Content						
Expiration-based	-	-	-	Yes	Yes	High
Key-based	Yes	Yes	-	-	-	High
Manual Trigger-based	Yes	Yes	-	-	-	High
Async. Update	-	-	-	Yes	-	High
Write-Trough	-	-	Yes	Yes	-	Medium
Chris Wasik [Was11]	Yes	Yes	-	Yes	Yes*	Medium
TxCache [Por12]	Yes	Yes**	-	Yes**	Yes	Low
Smache	-	-	Yes	Yes	Yes*	Medium
Smache w/o staleness	-	Yes	Yes	-	Yes*	Medium
Declared Content						
IBM [CDI98, CID99]	-	-	Yes	Yes	Yes	Low
HTTP-Response						
Chang et.al. [CLT06]	-	-	Yes	Yes	Yes	Low
DB-Queries						
Cache-Genie [GZM11]	Yes	-	Yes	Yes	Yes	Medium
Materialized Views	-	-	-	Yes	Yes	Medium

*) Dependencies to underlying data must be declared, **) Does not give these guarantees fully, but approximately.

Table D.1: Comparison of caching approaches for different types of content including Smache

Bibliography

[CDI98] J. Challenger, P. Dantzig, and A. Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, pages 47–47, Nov 1998.

[CID99] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 294–303 vol.1, Mar 1999.

[CIW+00] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 844–853 vol.2, 2000.

[CLL+01] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. *SIGMOD Rec.*, 30(2):532–543, May 2001.

[CLT06] Yeim-Kuan Chang, Yu-Ren Lin, and Yi-Wei Ting. Caching personalized and database-related dynamic web pages. In *2006 International Workshop on Networking, Architecture, and Storages (IWNAS'06)*, pages 5 pp.–, 2006.

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of*

the SIGCHI Conference on Human Factors in Computing Systems, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM.

[Doca] JSON Documentation. Json specification. http://www.json.org/.

[Docb] Python Documentation. Json implementation in python. https://docs.python.org/2/library/json.html.

[Docc] Redis Documentation. Distributed locks with redis. http://redis.io/topics/distlock.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[GZM11] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. *Middleware 2011: ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, chapter A Trigger-Based Middleware Cache for ORMs, pages 329–349. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Han12] David Heinemeier Hansson. How key-based cache expiration works. https://signalvnoise.com/posts/3113-how-key-based-cache-expiration-works, February 2012.

[Joh10] Robert Johnson. More details on today's outage. website, September 2010.

[LR01] Alexandros Labrinidis and Nick Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 391–400, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[McC] Andy McCurdy. Redis client for python. https://pypi.python.org/pypi/redis.

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[Por12] Dan R. K. Ports. *Application-Level Caching with Transactional Consistency*. Ph.D., MIT, Cambridge, MA, USA, June 2012.

[Was11] Chris Wasik. Managing cache consistency to scale dynamic web systems. Master's thesis, University of Waterloo, 2011.