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

Summary (English)

The goal of the thesis is to ...

Summary (Danish)

Målet for denne afhandling er at ...

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 thesis consists of ...

Lyngby, 04-July-2016

Anders Emil Nielsen

Acknowledgements

I would like to thank my....

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 this often not the easiest solution and in some cases the complexity or amount of data used makes it difficult achieve a satisfactory performance. Caching is a popular solution for improving the performance and scalability in these cases since it allows for a simple, scalable and generic way of addressing bottlenecks in the web applications.

Although it sounds like a silver bullet it also places a burden on the developer that must locate and update the cached values while preserving consistency guarantees. This challenge is for example seen in an outage of the whole Facebook system:

> 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 researching the latest caching technique proposed in research and used in practice and contribute with a design and implementation of a caching system in the Python programming language.

1.1 Problem

Most of the existing caching solutions are based on a pull based caching strategy, where the computation runs and the cached value is stored when the client requests the cached value. After the result has been computed and cached, the client will be presented with the cached value until the it is invalidated.

The pull based caching strategy has the advantage that we only have to cache content that is being used, but it also means that the first time a client asks for the value, it has to wait for the computation to finish. This is not optimal with relation to user experience since the user has to wait in order to be presented with the requested content. To solve this problem we have to precompute the cached values such that the user is presented with content as soon as it is requested.

Besides the performance problem on the initial request, existing caching solutions leaves responsibility for the developers to maintain the cache in order ensure consistency and an appropriate level of freshness.

These problems with existing caching solutions, presents two major challenges:

Cache Management.

The first challenge of cache management is faced in any caching system. The developer has to manage the caching system by assigning identifiers to the cached value and keeping it up to date such that the user is not presented with unexpected content.

One particular challenge within cache management is *cache invalidation* since it require the developer to identify every underlying data that affects the given cached value. The developer 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 web application and which computations are 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 and 5.

Data Update Propagation

The second challenge is related to the task of efficiently keeping the cached values up to date and ensure the consistency between the cache and the storage system. At first, if we need to support the web application to be scalable, we need multiple web application processes. This means we run the cache updates in parallel and we therefore need to prevent concurrency bugs and ensure liveness and correctness of the solution.

This challenge will be covered 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 developer that uses the caching system understands how it works from using it and has the ability to extend it. The design of the system should also be flexible to support multiple storage systems and caches.

Adaptability: Should be convenient and easy to adapt into existing systems.

Furthermore the usage of the system should be easy to understand.

Efficiency: Should be efficient with relation to performance such that it does not make existing operations of the systems significantly slower. It should also be efficient with relation to the system load such that it does not use more computational power than necessary to achieve the goal of the system.

Scalability: Should be designed for scalability in the sense that the design should still be efficient for large amount of data and correct when the web application is scaled horizontally.

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

1.3 Context / Running Example

The problem and requirements are based on a running web applications - the Peergrade.io-platform to ensure that the system is also designed, implemented and tested to be used in practice.

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. One of the key parts of the platform is showing various statistics about the performance of the students in a course. The statistics are based on advanced calculations which take up a large amount of time and needs to be recalculated on small changes to the underlying data.

To relate the solution to a practical example, the thesis will use the following code as a running example:

```
1 def course_score(course)
2     participants = ParticipantDB.find_by_course(course)
3     total_score = 0
4     for participant in participants:
5         total_score += participant_score(participant)
6     return total_score / len(participants)
7
8 def time_consuming_participant_score(participant):
9     return numpy.advanced_statistical_method(participant)
```

In this example we have a function `course_score` that computes the average score in a given course by fetching the participants from the primary data store and through iterations of each participant calculate the average score using the function `participant_score` that calculates the score of a single participant using a long running external method.

1.4 Contributions

This thesis addresses challenges described in section 1.1 and requirements in section 1.2 in the context of a caching system. The result will be a design of a cache system solving those challenges based on the requirements. The design will be implemented in Python and made available as open source to allow further research and extensions from the implementation. The implementation will therefore also have a focus on delivering a maintainable and tested library.

1.5 Outline

This thesis is structured 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 present the models used and a set of criteria used to evaluate caching approaches.

The following chapter 3 will describe the existing caching approaches described in literature and used in practice.

Based on the knowledge of existing solution, chapter 4, 5, and 6 describes the solution suggested by this thesis.

The solution is then evaluated and discussed based on test results and the initial requirements in chapter 7.

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 to find the appropriate technique.

Chapter 3 will describe existing caching approaches with relation to caching

evaluation criteria and explain how to choose a caching technique based on these criteria.

Chapter 4 presents the solution suggested by this thesis for the given context and requirements. First the techniques of existing approaches are discussed and followed by a description of the programming model - cachable functions.

Chapter 5 describes in more details how the cachable functions are extended to have automatic invalidations using declared dependencies to underlying data.

Chapter 6 explains the data update propagation algorithm used to extend the cachable functions to have in-place cache updates.

Chapter 7 goes through the test results related to performance, efficiency, and cache memory usages. The solution are then discussed and evaluated based on the test results and the initial requirements.

Chapter 8 finalize the thesis with a conclusion.

Chapter 2

Caching Model

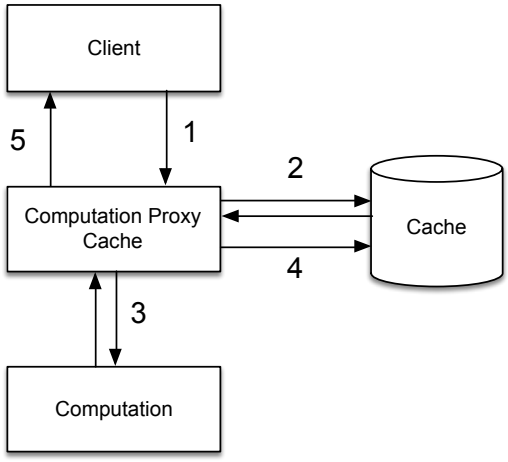
In order to get a common understanding of caching and the terminology related to the topic, we will go through the basics of caching by describing the architecture of a web system using a cache, present a model based on a timeline that introduces the different events involved in caching and last we list criteria for evaluating a caching technique in order to choose an appropriate technique for a given use case.

2.1 Caching Basics

In general caching is about storing the result of a computation at a where you are able to retrieve it fast, such that it is possible to get the result fast instead of recomputing it. This basic algorithm is illustrated on figure 2.1 and can be described as following:

In some cases, where the client is not allowed to wait for the computation to run, step 3-4 are replaced by a step that simply returns an empty value.

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



1. The result v of a computation f is requested.
2. If v is cached and is valid, we go to 5 with $v' = v$. Else we continue.
3. We run the computation f
4. The new result v' of f is stored in the cache.
5. v' is returned.

Figure 2.1: The flow of basic caching

from a storage system, sometimes it's the result of an API call to some external resource, sometimes it's global variables in the code. These *inputs* we from now on be referred to as *underlying data*.

In order for the algorithm to work, we need to be sure that when we store the cached object it has to be uniquely identified by some key such that when we lookup the value as in step 2 of the algorithm, we always locate v. This presents one of the challenges of cache management, which is in many cases closely related to cache invalidation (one of the two hard things in computer science[1]).

In the algorithm cache invalidation is simply described as a *check*, but in reality this is the hard challenge of caching. The *check* could be a precomputed indicator from earlier triggers, it could be based on a key derived from the underlying

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

data or some timestamp. These cache invalidation approaches will be described more in section 3.1.

2.1.1 Architecture

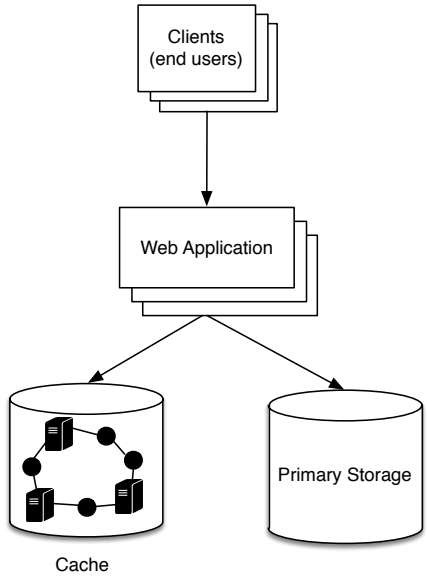


Figure 2.2: The assumed architecture of the system

The architecture of the web application in which the cache is used is important to how the cache system. We assume that the architecture of the system is a common web application architecture as illustrated in figure 2.2 consisting 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. This could easily be the same unit as the primary storage database, but we separate them for better clarification.

Most modern web applications needs to serve multiple users at the same time, which means the web application must run on multiple processes[2] either on a single or multiple machines. We will therefore treat the web application as a distributed system.

[2]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.

A popular choice for cache databases are key-value stores such as Redis [3] and Memcached [4] since they are simple distributed key-value stores that lives in memory and therefore allows for scalability and high-performance operations. In order to support most practical web applications, we will therefore make the assumption that the cache database has the same functionality: it should be possible to store arbitrary content for a given key. Furthermore the final solution of this thesis require the cache database to support atomic transactions for a given key, which are supported in Memcached using the CAS command [doca] and in Redis using the `WATCH`-command [docb]. The reasons behind this assumption will be explained more in chapter 6.

2.1.2 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:

- **Requested** is the event occurring when a client requests a given cached object
- **Computation Started** is when the computations is started
- **Computation Finished** is when the computation is finished and a result is returned
- **Stored** is happening when a given cached object is stored in the cache database
- **UD Updated** is when the underlying data has been updated
- **Invalidated** is when the system has knowledge that the cached object is invalidated

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

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

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 supports immediate responses. In the time interval, where the fresh and valid interval overlaps, the approach will serve a fresh version of 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.

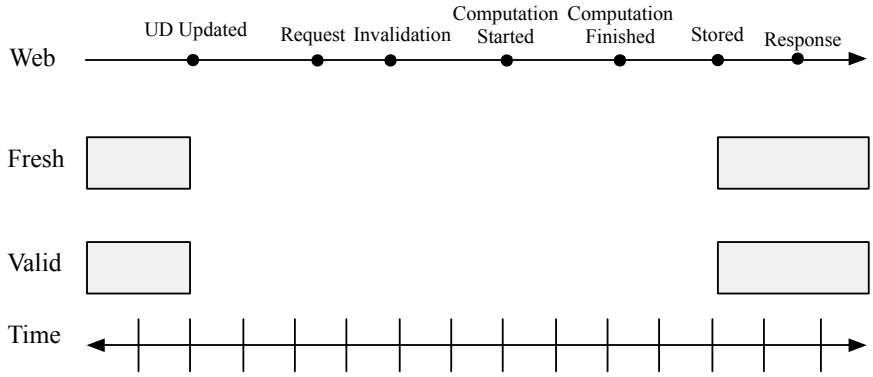


Figure 2.3: The timeline model applied to the basic caching algorithm

2.2 Evaluating Caching Techniques

To choose the correct caching technique for a given use case, we need to know the criteria for finally picking the best suited. The overall goal of caching in web development is to achieve a better user experience by getting a better performance 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 "hitting the cache" as often as possible. We can measure this using the metric *cache hit rate*, which refers to the rate at which the cache is hit among the total number of requests for the cached object.

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 contains 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. Furthermore 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 of the level of consistency.

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 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, but in the end the goal is to make the served content make sense for the user. Consider example 2.1 of the Peergrade.io platform, where there are two types of users. If the 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. On the other hand if the teacher requested the description 1 min after, it would be unexpected behaviour to show the teacher an old version, since the teacher would think the description hasn't been updated. We will represent freshness as a binary parameter that we call "Strict Freshness", which evaluates whether a given cached object that is fetched is guaranteed to be fresh.

Example 2.1 *At Peergrade.io the web application has a teacher and a student interface. Teachers create assignments and sees statistics about the grades students have given to each other through their feedback. The students see the assignments created by their teachers, are able to hand in their assignments, and grade other students' hand in.*

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 computation has to run after it has been invalidated. If the caching technique runs this computation in the same process as the request from the client, the client has to wait for the computation to finish in order to show the content to the user. If the time taken to compute the value exceeds the accepted response time limits described above, it could become 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 weight of this parameter is proportional to the cache miss rate since it's only requests, which results in a cache miss that are affected by the slow response time.

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, it is not possible.

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. This means that we start computing the cached objects as soon as they have been invalidated. In order to achieve this we need some mechanism that invalidates depending cached objects when underlying data are updated. We will also represent "Automatic Invalidation" as a binary parameter.

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 developer since a big part of cache management is manually tracking dependencies between the cached content and the underlying data. We want the caching system to be transparent such that it is easy to add and remove caching 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: does the caching technique involve maintenance with relation to invalidation (shortened as invalidation maintenance)?

The last parameter we will introduce is also a requirement of the system: *adaptability*. Since adaptability cannot be evaluated objectively 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 data store, cache database and/or application that means it cannot be applied to other common technologies.
- **Medium**: The technique is advanced and requires a lot of implementation effort or external libraries/processes to work.
- **Low**: The technique can easily be implemented and applied to existing applications.

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 other data
- **Strict Freshness**: The cached object must be as fresh as the state of the primary store when the value was requested
- **Automatic Update**: The cached object is automatically updated after it has been invalidated
- **Always Immediate Response**: The cached object must be served immediately after it has been requested
- **Invalidation Management**: Whether the developer has the responsibility of maintaining the invalidation of the cached objects
- **Adaptability**: How adaptable the caching technique is to existing systems

Chapter 3

Caching Approaches

Since caching is an approach widely used in practice there already exists multiple caching approaches described in literature, articles on the internet and in open source code. To discover existing solutions for the problems and challenges faced in the thesis, we will research and evaluate existing caching approaches. The approaches will be analyzed and evaluated based on the criteria described in section 2.2 and requirements from section 1.2.

As already the described the most difficult part of caching is the invalidation. We will therefore start by describing existing invalidation techniques followed by how the cached values are updated and related problems. We will also describe some specific caching approaches related to web development and finish the chapter by giving an approach to picking an appropriate caching approach for a use case based on the caching criteria.

3.1 Invalidation Techniques

When the underlying data for a cached value is updated we consider the cached value as invalid. Although this sounds as a trivial part, the mechanism for invalidating the cached value can become complex depending on the requirements for

the use case. To be able to navigate the existing solutions we will cover the general invalidation techniques as well as the more advanced techniques described in literature.

3.1.1 Expiration-based Invalidation

In cases where it is accepted to keep the cached values stale up to a given time interval, we can use the expiration-based invalidation technique, which is the simplest invalidation mechanism since the invalidation only depends on time and not updates of underlying data. Using the expiration-based invalidation we give up consistency and the level of freshness depends on the time interval set by the developer, but we are able to achieve immediate responses.

The expiration-based invalidation works by assigning a TTL (Time to Live) to the cached value. At some point when the TTL has expired, the cached value is invalidated. The responsibility of invalidation cached values with TTL is often placed on the cached database by piggybacking the TTL to the cached value when it is stored. This is for example supported by Redis and Memcached.

To give a sense of how the relation between when a cached value is considered valid (it is served from the cache) and when it is actually fresh, the timeline model has been applied to the expiration-based invalidation on figure 3.1.

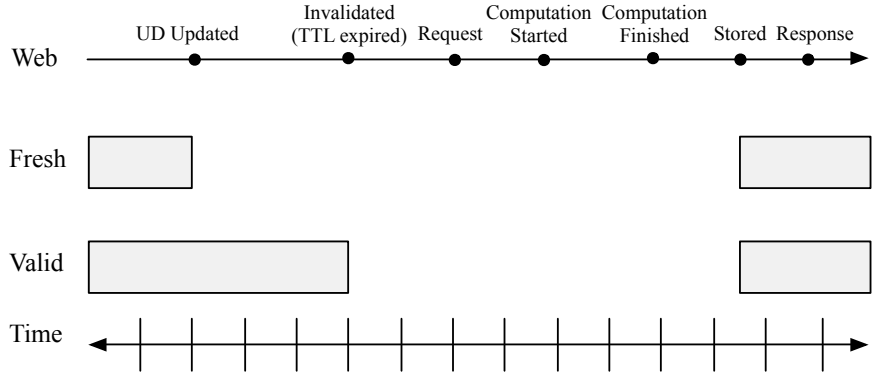


Figure 3.1: The lifecycle of the expiration-based invalidation technique

3.1.2 Key-based invalidation

When the cached values are required to be up to date with the primary storage, we must ensure that when underlying data is updated, the depending cached values must be recomputed before the cached value can be served. The key-based invalidation gives these guarantees by giving up immediate responses since the users have to wait for computations to finish if they request an invalidated cached value. As mentioned in section 2.2 the impact of this trade-off is proportional to the cache miss rate. This means that the key-based invalidation will not be suitable in cases where the computation time is too long or in cases where the cached values are updated too frequently.

Key-based invalidation works by constructing the cache key from parts of the underlying data such that the when the cached object should change, then the key also changes. [Han12]. The cached content is considered immutable and only have to be written once. 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 developer 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.

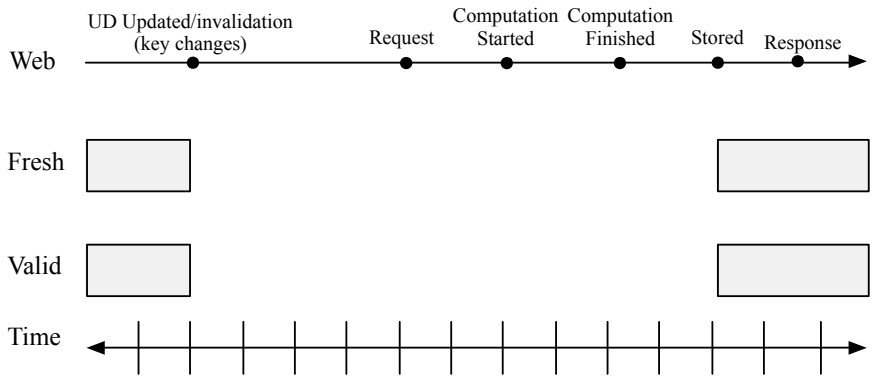


Figure 3.2: The lifecycle of the key-based invalidation technique

In the web application framework, Ruby on Rails, the key construction is simplified by using a key that includes the timestamps of the last update on some

underlying data. An example of this technique can be seen in code example 3.3. In this example we use the type, id and update timestamp as parts of the key. This means that the cached value is considered invalid if it is called with an entity that has a different type, id or update timestamp. The intuition behind these components is that we want a unique cache value for each entity and we want the value to be recomputed when the entity is updated.

The key-based invalidation performs invalidation at the moment, the cached value is requested, but it is considered invalid at the moment after the cache key components are updated (e.g. the *updated_ timestamp* in the code example). This means we get interval the value is considered value and fresh are always overlapping, which means we have consistency.

A caveat of this method method is that it generates cache garbage since old versions of a cached value are not removed. To avoid the complexity of keeping track of the relations between the different, the responsibility for cleaning up is moved to the cache database. Fortunately cache databases (such as Redis and Memcached) implements different such cleanup algorithms that detects obsolete values based on some policy. One such policy called *Least Recently Used (LRU)* removes the cached values that are least recently used by keeping a timestamp of when the cached values where accessed last. Another policy called *Least Recently Used (LRU)* keeps track of the frequency in which the different values are accessed and removes the ones that are least frequently accessed.

3.1.3 Trigger-based Invalidation

Instead of invalidating the cached value when requested, the cached values can be invalidated based on certain events triggered when the underlying data is updated. Given that 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. The timeline model of the trigger-based invalidation on figure 3.4 therefore looks similar to the key-based invalidation.

Where the key is used as an invalidation mechanism in key-based invalidation, the key is static for trigger-based invalidation i.e. the key for identifying a cached value does not change in its lifetime. 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 value is accessed. The responsibility of invalidating the keys are then moved to the definition of event triggers.

```
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,
        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, which
# means it returns the value from the cache
print cached_participant_score(participant)
```

Figure 3.3: Example of the key-based invalidation technique

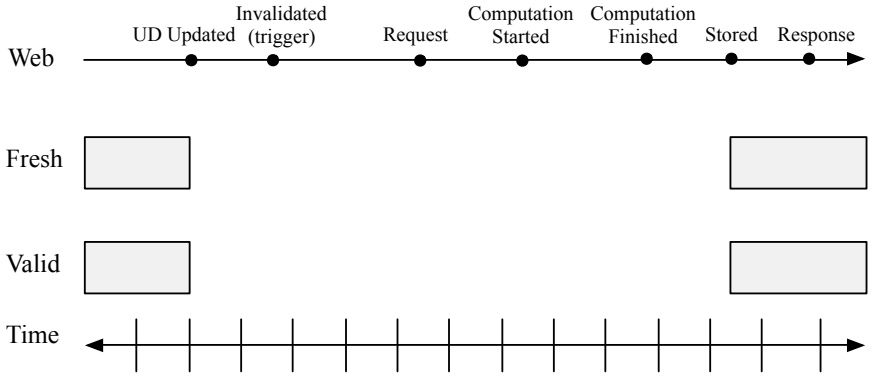


Figure 3.4: The lifecycle of the trigger-based invalidation technique

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]. We can also extend the technique to be more flexible with relation to the properties as explained in section 3.1.4 and section 3.1.5.

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 snippet 3.5. 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 developer 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 3.6, 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, which makes it stale.

[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

```
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:
        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)
```

Figure 3.5: Example of how trigger based invalidation works with manual code invalidation

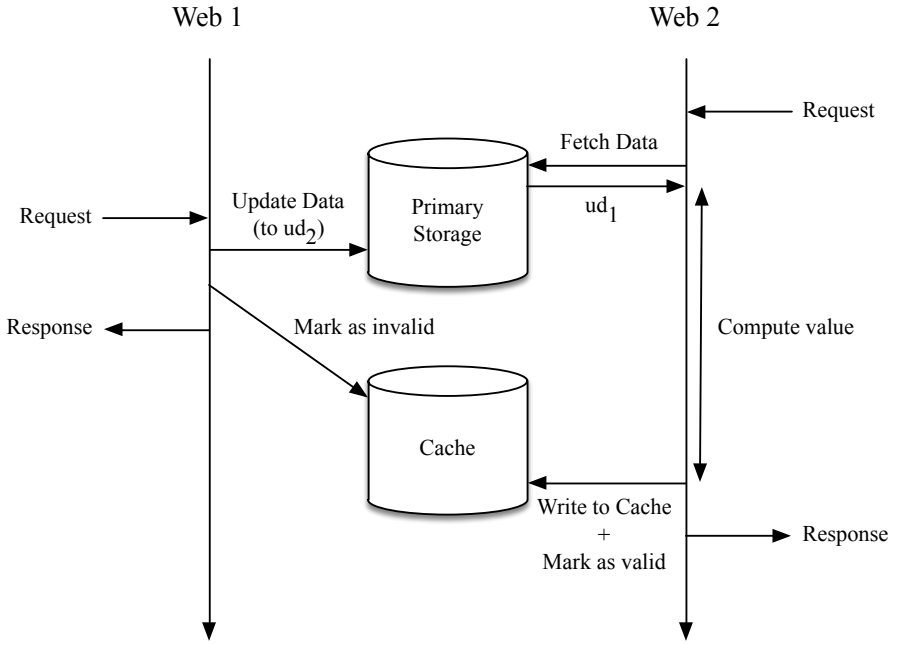


Figure 3.6: A scenario of the trigger-based invalidation that results in a race condition, where the cached value are being incorrectly marked as valid even though it is storing a stale value.

3.1.4 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. To give this guarantee fully the user have to wait for the computation the first time the value is requested if the application haven't ''pre-heated'' the cache i.e. included a build step before deployment that computes all cached values.

The timeline of this extension is as on figure 3.7. A naive implementation of this technique would be as the trigger-based seen in code snippet 3.5 with the modification that the system updates the value asynchronously instead of synchronously if no fresh value is found in the cache. The snippet for this can be found in appendix A.1.

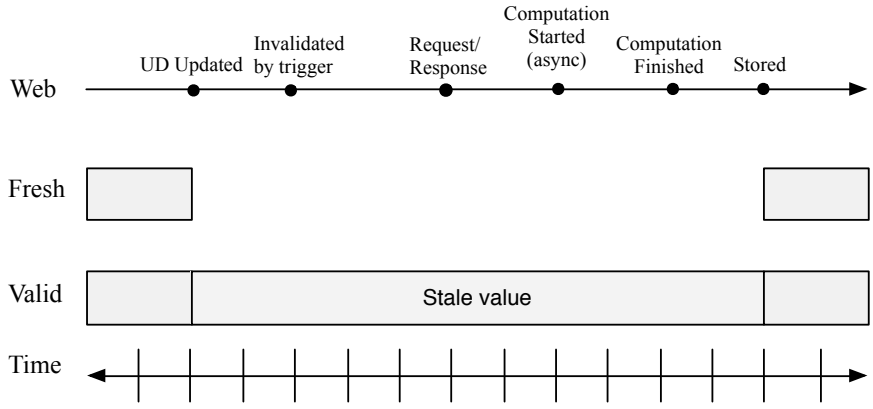


Figure 3.7: The lifecycle of the trigger-based invalidation technique where the value is updated in the asynchronous

3.1.5 Write-Through Invalidation

Write-through invalidation is also an extension to the trigger-based invalidation method, 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 3.8 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.

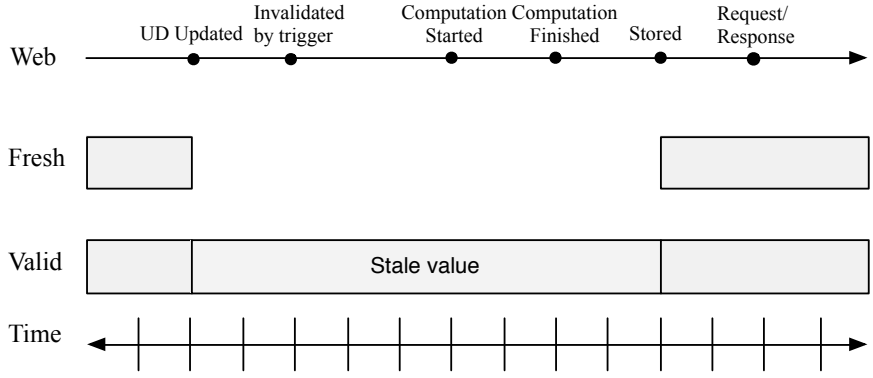


Figure 3.8: The lifecycle of the write-through invalidation technique

3.1.6 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 developer 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.

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.

In [GZM11] the database wrapper [2] is used to detect and trigger changes to the underlying data. This paper suggests a caching approach for caching database queries by declaring predefined queries with dependencies to underlying data using an extension to the database wrapper. The database wrapper is then responsible for detecting changes to underlying data and invalidate the affected cached queries. The advantage of using a database wrapper is that the caching system will still work if the database used by the wrapper is changed. 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.

This problem can be solved using a database technology that is able to notify about changes directly from the database as done in the approach from IBM [CDI98, CID99] that uses the IBM DB2 [3] database. In this approach the triggers are intercepted by a cache-manager that is able to invalidate the affected values using a dependency graph. Having a cache-manager introduces a single point of failure and potentially a bottleneck, but it allows capturing dynamic dependencies, which is required for the given application. The details about this approach will be described in section 5.1.

In [Por12] a patch is made to the database such that the database is able to support so called "invalidation streams" that can help the caching system invalidate the affected cached values. The cache nodes stores information about how the different cache objects should be invalidated represented by invalidation tags. The invalidation stream from the storage system are then distributed across all cache nodes and used to invalidate the affected cached objects using the invalidation tags. The primary reason for these invalidation streams is to

[2] In this case in the form of an Object-Relational Mapper
[3] `http://www.ibm.com/analytics/us/en/technology/db2/`

allow transactional consistency such that the primary storage and cache nodes share information that can be used to fetch a consistent set of data across both sources.

The same approach also suggests using transactions between the primary storage and the application during the computation of cached objects to capture dependencies to entities from the primary storage using the queries made in the given transaction. This way the dependencies are captured automatically, but it requires the database to implement these transactions.

[Was11] suggests using static analysis of the code to capture dependencies between the cached functions and the underlying data. In this approach the developer must denote the data relationships, which are analyzed by the static analyzer. The static analyzer will then detect relationships between the cached functions and the underlying data as well as between different cached functions. Based on this information an object in the application process will store the dependencies for each type of dependency and not for each dependency instance. The exact dependencies are then derived using queries to the primary storage when changes are triggered. The given approach also uses a static analyzer to detect the relevant triggers.

The advantages of the deploy-time approach is that it doesn't require additional processes and it does not have to keep track of state in the form of dependencies between the different cache object instances, but these decisions have the cost that it is difficult to implement fault-tolerant measures (such as invalidation retry) as well as a poorer user experience from the performance impact of invalidation. Furthermore the approach has no mechanisms for avoiding the problem of concurrency bugs (describe in section 3.6).

On the other end of the granularity scale, [CLT06] suggests a system that caches HTTP responses. It uses a sniffer process that monitors the lifetime of a HTTP request with the queries made to the storage system. Through the information captured by the sniffer, the system builds a table that maps a given HTTP resource to the queries made. The system then caches the HTTP resource that is invalidated when underlying data related to the given resource changes. 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.

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 requests to update underlying data in the primary store. If we consider figure 3.9 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.

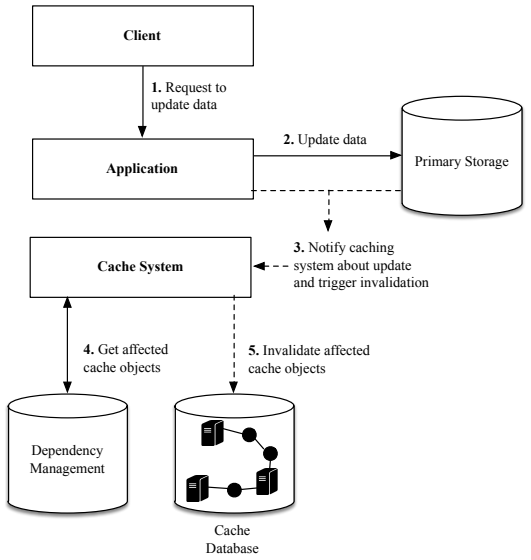


Figure 3.9: The control flow of automatic invalidation when a client requests to update underlying data

3.1.7 Triggering Cache Invalidation

When the client updates underlying data, the application receives a request from the client and sends a request to the primary storage to update the relevant data. To be able to react to this action, triggers have to be implemented during this flow. One technique used by [GZM11, Was11] is to have the triggers in the application such that the application triggers invalidation when the changes from the client has been applied to the primary storage. The given techniques involves a database wrapper (in the form of an ORM) that implements callbacks invoked when it sends commands to the database. If the database wrapper is implemented using the adapter-pattern, it is possible to change the underlying database technology without changing the implementation of the caching system. This means it is easier to implement the caching system for multiple

database technologies as well as change the technology for existing solutions.

Other solutions such as [Por12, CDI98, CID99] assumes that the database is able to send notifications when changes are made to the database. These 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 third process that converts the change notifications to invalidation of affected cache objects. A similar approach is used in [CLT06] that 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.

In the techniques with in-application triggers, changes to the database around the application are not captured. So if the system has the requirement that the primary storage can receive commands from multiple applications, the best solution would be to use triggers directly from the database. But this also means the system is required to use a database technology that supports this. This comparison is also shown on figure 3.10.

	Advantages	Disadvantages
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
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

Figure 3.10: Comparison of triggers for automatic invalidation

3.1.8 Dependency Management

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

Since dependency management is a burden for the developer 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]. In [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 compute 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 developer 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 relies such as [GZM11, Was11] on the developer 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 developer 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 cache 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 3.11.

	Advantages	Disadvantages
Declared in code (Cache-Genie)	- Easy to reason about dependencies	- Relies on the developer correctly identifying and declaring dependencies
Declared in content (IBM)	- Allow different data source for different entities - The developer does not have to declare dependencies	- Burden for users to define dependencies on each entity
Static Analysis and Code Generation (Deploy-Time Model)	- Semi automatic: only require definition of database relations	- Static analysis cannot be implemented correct in dynamic languages - Difficult to detect relational dependencies - Requires knowledge about static analysis to implement
Database Transactions with Invalidation Tags (TxCache)	- Fully automatic	- Require the database to implement the transactions

Figure 3.11: Comparison of dependency management techniques for automatic invalidation

3.1.9 HTTP Caching

3.2 Choosing the Right Caching Technique

In general there is no best or correct solutions - it's a matter of choosing the solution best suited in the given context depending on the web application and specific use case. We will therefore compare the different approaches based on the parameters described in section 2.2. This comparison is illustrated on figure 3.12.

	Consistency	Strict Freshness	In-place update	Always Immediate Response	No Invalidation Management	Adaptability
Arbitrary Content						
Expiration-based Invalidation						High
Key-based Invalidation						High
Manual Trigger-based Invalidation						High
Key-based Invalidation with Async Update						High
Write-through Invalidation						Medium
Dan Ports: Transactional Consistency						Low
Chris Wasik: Managing Cache Cons...						Medium
Declared HTML-content						
IBM						Low
The other XML ones						Low
HTTP-response						
Y-K. Chang et al.: DB Sniffer						Low
DB-Queries						
Cache-Genie						Medium
Materialized Views						Medium

Figure 3.12: Comparison of caching approaches

From this comparison we see that some decisions have to be made between some of the parameters. At first the approach cannot both give an immediate response and provide strict freshness since if the cached computation takes Δt_c to compute and the invalidation happens just before the request then the response time will be Δt_c.

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. Based on the overview of existing caching solutions introduces in the last chapter, we will present a solution that solves the problem for the context of the this thesis.

4.1 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 that is not depending on the time taken to compute the information. Furthermore we want to keep the information as fresh as possible.

If we consider the overview on figure 3.12, we can already leave out caches that optimizes 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 approaching caching arbitrary content it is desirable to have an approach with high adaptability, which was stated as one of the requirements of the system. From the approaches with high adaptability that 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 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 are 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 time of the computing 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 of the computation, n_c is the number of computations, Δu_c is the number of updates per 24 hours and Δt_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 occupy a CPU in $\frac{60 \cdot 500 \cdot \frac{10 \cdot 60}{60 \cdot 60 \cdot 24}}{60 \cdot 60 \cdot 24}$ occupied CPU's = 50 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 alternative with high adaptability is to use manual trigger-based invalidation with asynchronous updates after request as described in section 3.1.4. This approach has the advantage that the values are only invalidated (and thereby recomputed) when a trigger is invoked, which means the developer 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 developer to maintain these manual triggers. And by having asynchronous invalidations, the user is presented with a stale value, and only if the value is requested again after the update, the user receives the fresh value. 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. In cases such as this, where the value is only fetched once in a while it would not be optimal to have asynchronous updates, since the value will never be close to fresh when it is actually presented.

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 with additional automatic invalidation, but it would also have the same problems.

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 developer to declare the triggers that invokes the write through updates. Furthermore it leaves the challenge of ensuring the integrity of the cached values and invalidation marks in concurrent environments as described on figure 3.6.

The goal of this thesis is to present a caching approach that solves the problem of the thesis for the use case described in section 1.3 as well as fulfill the requirements described in section 1.2. Since the computation time in the given use case can be long, we must ensure that the response time does not depend on the computation time. Since we must respond immediately we need to trade-off strict freshness. To keep the values as up to date as possible we can use in-place updates based on trigger invalidation. As we can see on figure 3.12, there exists no solutions fulfilling those criteria. In the rest of the thesis we will present the design and implementation of the solution that meets these requirements, which we call "Smache".

4.2 The Cachable Function Model

Smache uses the same programming model as the one 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 for the function 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 developer to ensure only pure functions are cached.

4.2.2 Making Functions Cachable

Overall the API for making cached functions are similar to the Dan Ports' approach in which they define a `MAKE-CACHABLE` procedure that converts a pure function to a cached function with the following definition [Por12]:

`MAKE-CACHABLE(fn)` → *cached-fn*: Makes a function cacheable. *cached-fn* is a new function that first checks the cache for the result of another call with the same arguments. If not found, it executes *fn* and stores its result in the cache.

This definition is the basis of making a function cachable, but it will be extended further as automatic invalidation are added in chapter 5 and in-place updates are added in chapter 6.

[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

4.2.3 Automatic Cache Invalidation

In Dan Ports' solution the system handles all parts of cache management such that the developer 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 as well as a 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 developer 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.

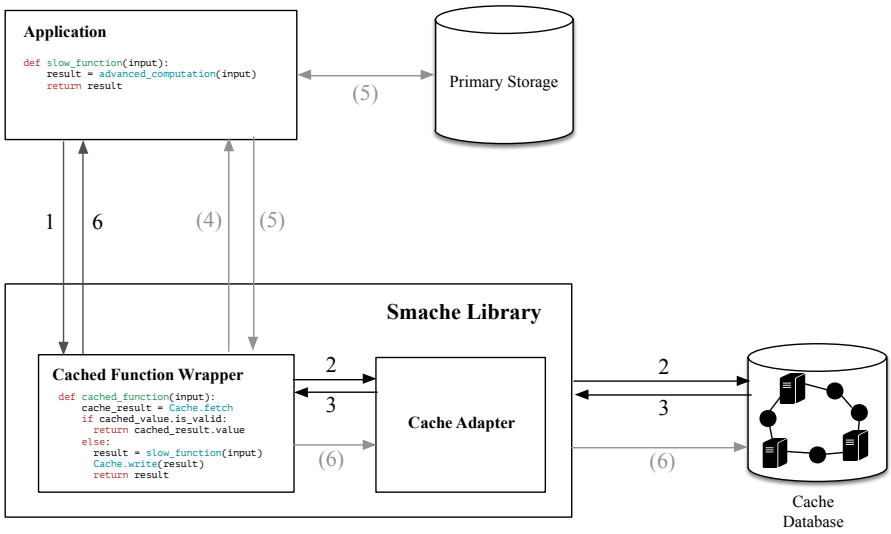
4.2.4 Data Update Propagation

Smache also offers the possibility to perform in-place updates 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(TODO: Include references).

Having in-place updates allows the developer to have immediate response times while ensuring that the values returned are updated as soon as they are invalidation 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

The basic cachable functions as they are described so far can be implemented using the architecture described in section 2.2. Smache implements the cachable function by exposing an implementation of the `MAKE-CACHABLE`. To apply the procedure the Smache library must be loaded in the application and applied. After the `MAKE-CACHABLE` function has been used, the control flow illustrated on figure 4.1 is applied. 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.



1. The application calls Smache through the cached function wrapper
2. Smache sends a `Lookup` request to the Cache Database through the Cache Adapter
3. The cache adapter returns the value with an indication of freshness based on the result from the cache database

(4). Smache calls the original function through the application

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

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

7. Smache returns the value to the caller from the application

steps annotated with (paranthesis) are only executed on **cache miss or cache failure**

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

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

The implementation of the basic caching on the running example is seen on code snippet 4.2. In this code 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.

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

This call corresponds to the implementation of `MAKE-CACHABLE`.

Smache require the developer to annotate the types of "data source", 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 "data source 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.

Smache uses the type annotations to do smart serialization and deserialization when the name of the cached value have to be inferred. The raw input are represented as they are in a serialized form, but the data source inputs are serialized using their. When the function is then deserialized the Smache library will use the id of the data sources to load a fresh version of the ORM-object from the database. Another advantage is that the Smache now know that this cached function has to be invalidated when the corresponding course entity is updated, which will be explained in chapter 5.

```
1  @smache.computed(Course)
2  def course_score(course)
3      participants = ParticipantDB.find_by_course(course)
4      total_score = 0
5      for participant in participants:
6          total_score += participant_score(participant)
7      return total_score / len(participants)
8
9  def time_consuming_participant_score(participant):
10     return numpy.advanced_statistical_method(participant)
```

Figure 4.2: Implementation of basic caching on the running example

Chapter 5

Automatic Cache Invalidation

5.1 Object Dependence Graph

Jim Challenger et. al. has written multiple papers on the system used for the content management website in the Olympic Games in 1998 and 2000 [CDI98, CID99]. The system is based on content that are all precomputed when served to the user, which resulted in a system that scaled for many users with content served fast since the web server only had to find the appropriate cached article when serving content. In order to allow editors to change articles and fragments, the system introduces the Data Update Propagation (DUP) algorithm. DUP uses an Object Dependence Graph (ODG) that describes the relationship between fragments using a Directed Acyclic Graph (DAG). The ODG describes both the relationships of how the fragments are embedded in each other and relationships describing the hypertext links between articles. To avoid race conditions and hypertext links to missing fragments, the fragments need to be updated in a specific order. More specific when a fragment f1 that embeds another fragment f2, the system need to update f2 before f1. Since the ODG is described DAG there is always a topological order of the nodes, which satisfies the described property for any node. The system runs using a CMS system, where the content is defined using a CMS system and not using functions from the source code. This simplifies the challenge of persisting the cached content

since it does not change when a new version of the source code is deployed. It therefore leaves the challenge of updating cached content, when the definition of the computations changes.

5.2 Existing Solutions For Cache Invalidation

Map out how the method for doing this (sniffing)

* Sniffing (of the database) * Invalidation Streams (Transactional Consistency) * ORM Signals: Using signals from the database wrapper (CacheGenie) * Static code analysis on SQL-queries: However managing cache consistency does it * IBM: dependency graph + cache manager + triggers from database

5.3 Extending Cachable Functions With Automatic Invalidation

- Seperate static and dynamic "state" - Static: only changes when new versions of the code base is deployed - Dynamic: changes after deployment - often initiated by changes to the underlying data - Static state is preferred since it it's easier to test and has not 'state'

- Static data now: - The computed functions: - find functions from its id - find depending relations from other entities

- Dynamic data now: - The dynamic functions: - Dependency graph stuff - The stored cache elements

- How it could work: - When calling the cached function (fun instance registration): - When a given computed fun is executed we will serialize the arguments the following way: - Raw values (that can be json) is just serialized into json - Underlying data (entities) are serialized by their entity id and instance id - We can then cache the given value under a unique identifier given by concatenation - A unique id for the fun (python: module + fun name) - The serialized arguments - From the entity arguments we store a reference from the entity instance to the computed fun instance - This is stored in the "dependency graph" - The next time the function is called with the same argument, it will be fetched from the cache - When underlying data changes: - If the data is corresponding to an (entity) argument of the function - We lookup the dependency graph for all the

computed funs affected by the entity - These computed funs are then marked as stale and/or written throug/updated - TODO: describe relation updates

- Operations needed to be supported: - When the data is fetched: - Lookup cache to see if value is fresh - (Computed and) Store cached value - When underlying data changes: - Find computed functions depending on the given entity instance - Both: - directly (through arguments) - related (through relations) - Set cached value as stale - Update cached value

- Data structures introduces: - Find affected computed fun instances from given relation - Subproblems: - Find underlying data identifier from relation instance - Done using a relation-function defined by the developer. This function is executed every time the query is executed. - Find affected for computed fun instances from underlying data identifier and given computed fun - Using dictionary (redis) with - key: underlying data identtifier - value: computed fun identifier - Record stored when a new function instance is executed - Find affected computed fun instances from direct underlying data - Using dictionary (key-value store/redis) with: - key: underlying data instance id - value: computed fun identifier - Record (also) stored when a new function instance is executed

- DS Example:

– Static dep graph:

prof - score user -/

prof - hello user -/

– Data Structure: - Given instances: prof/1 - score/1/2 user/2 -/

prof/1 - hello/1/5 user/5 -/

- Find all funs affected by underlying data: prof/1 => [score/1/2, hello/1/5] user/2 => [score/1/2] user/5 => [hello/1/5]

- Find all funs of a given kind affected by underlying data (for relations) score.prof/1 => [score/1/2] hello.prof/1 => [hello/1/5] score.user/2 => [score/1/2] hello.user/5 => [hello/1/5]

Chapter 6

Data Update Propagation

6.1 Updating the Cache

6.1.1 Data Update Propagation

6.1.2 Consistent Concurrent Write-Through

At first we would like the caching to be correct. In the case of caching, we will define correctness in terms of liveness and safety: the caching system will update the cache when necessary and it will eventually return the most fresh value computed. That is if we have a computation f that computes the value v_1 at time t_1 and v_2 at time t_2 then the cache store will eventually contain v_2 given that $t_2 > t_1$. Although this could be seen as a prerequisite for the caching system, most implementations ignore this fact to achieve a simpler cache system. Because when we want to keep the integrity between updates we need some kind of ordering for the updates that adds complexity to the system. The problem is also illustrated on figure 6.1.

Since we need to be able to handle multiple web servers, we need to consider the system as a concurrent system.

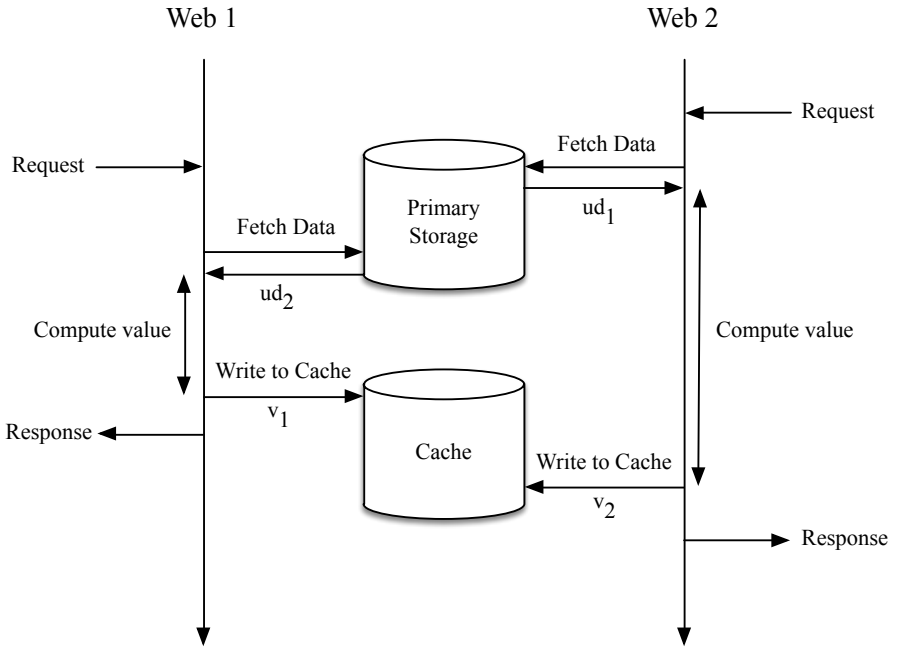


Figure 6.1: Showing how two concurrent caching updates from two different application servers results in an inconsistent state. We see that even though the request from *Web 2* are based on data older than *Web 1* it gets to write

The problem is basically that the key of the cached content is the same, which means that we do not have any accounting of which update are allowed to overwrite others. E.g. if two updates u1 and u2 happens right after each other and executes a write through update with relevant data, where u2 will produce a fresh result and u1 will not. If u2 is fast at computing and u1 is slow then u2 will start by writing the fresh cache value, and u1 will overwrite the fresh one with a stale result.

In this problem we need a mechanism such that u1 does not overwrite u2.

One solution is to use locks such that only one update can be computed per key in a given moment. In the given case u2 would be locked until u1 has finished and written to the cache.

The downside of this method is that the result can in staleness of time t(u1) + t(u2) - 1.

Another solution is to use timestamps. When u1 is scheduled, it fetches a timestamp from resource (redis incremental). This timestamp is then submitted when the cached value is written. In the given example u1 would get one timestamp t and u2 will then get $t+1$. u2 will then submit a cached value with timestamp $t+1$ and afterwards u1 will try to submit a stale cached value with timestamp t, but since a newer version has been written it would be ignored.

NOTE: This method is also mentioned in the IBM paper as amount of times a given value has been written.

The downside of this method is that we need some kind of centralized timestamp generator, which will then become a single point of failure of the system. Theoretically this could be solved in a proper manner using a distributed and replicated timestamp generator (ETCD maybe).

We need to assign timestamp in the execution. If we assigned timestamps before we inserted into the queue and we retried a job, then it would be fair to assume that the retried job had the newest data and therefore has to be able to write to the cache.

We want the properties:

* Liveness: a given function is executed after finite time. In this case it means that if we have any set of updates U that results in the global state s and we have a cached value $C(f, s)$ that is the cached result of function f for state s, then we know that at some point in time after the events in U have occured, we must have stored $C(f, s)$ or $C(f, s')$ where $s \to s'$.

If we used a naive algorithm, where a given update invokes an update execution that then computes the value and updates the cache, we would have potential race conditions and not ensure liveness. This could happen in the simple case

u => f

with event order:

* u1: u is updated => global state s * f is invoked for worker w1 with state s * u2: u is updated => global state s' * f is invoked for worker w2 with state s' * worker w2 finishes and updates f with the cached value C(f, s') * worker w1 finishes and updates f with the cached value C(f, s)

In this case, the resulting value would be C(f, s) which would contract liveness since we have a set of updates $U = u1, u2$ that gives the global state s', where the cached value $C(f, s')$ is never stored.

To solve this we can use timestamps:

Instead of incrementing timestamp as above, we use the notion of state timestamp and value timestamp. There is one state and value timestamp for each cached value. These are defined as:

* state timestamp: the amount of times the underlying data/state has been updated. When some underlying data is updated, this number is updated for all affected computed nodes. * value timestamp: the state timestamp which the last update was based on.

The system should then do as following:

* R1 (value update): Right before the state input is fetched cst_i and the function is executed, the state timestamp for the cached value is taken an called $ft_i = cst_i$. When the function has finished and wants to store the cached value, it need to ensure that the current value timestamp for the cached value cvt_i holds: $cvt_i < ft_i$. If this holds, the cached value is written and $cvt_i := ft_i$. This is done atomically. * R2 (freshness): When a cached value is written, We also set the cached value as fresh ($cf_i = True$) iff the current $cst_i = cvt_i$, else we set $cf_i = False$.

When one value is updated, we need to propagate the update. Since we do not want to calculate an entry more than once, we need to do it in topological order.

Another thing is: - What if a function instance is set to be executed a lot of times? - Here we would like the topmost instance to be executed, because it means that it's siblings will be cached. Alternatively if only the topmost was executed, we could risk that some siblings are not updated, if e.g. there is a condition which removes the relation. Phew... - We also don't want to risk starvation, ie we want to ensure the instance is executed at some (fair) point in the future. If we e.g. removed every instance except the topmost a lot of times, it would result in starvation.

One solution would be to truncate the queue when a new job comes in (or periodically).

This would need some kind of algorithm to control and remove duplicate jobs: One way would be to remove every duplicate job except for the first and last in line. This would avoid starvation since the first job in line will never be removed, and it will ensure that we eventually will calculate the newest entry. This assumes that the queue will be finished in a fair amount of time. For example if 100 jobs with including the same function would be enqueued then it would mean we probably mean that the truncated job would have a 97 jobs

in between.

Chapter 7

Results and Evaluation

To prove efficiency:

- Test that a given function is not made slower after being cache - To test this we want to trigger an underlying data change and ensure it does introduce performance regression - System load: - Test that functions are not executed more times than necessary - Show that caching improves performance: - Examples where caching improves performance - Test the 'warm up'

Maybe not to be tested, but to be argued: - Software design: - Defend the library and how it's used - Adaptability - Discuss that the definition is could be improved, but fits evaluation in the cases we have

Chapter 8

Conclusion

- Pull based more fault tolerant - if execution fails, we just serve an allright value

8.1 Future Work

Appendix A

Stuff

This appendix is full of stuff ...

A.1 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(
          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)
```

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.

[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] Memcached protocol documentation.

[docb] Transactions in redis.

[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, February 2012.

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

[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.