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, 01-January-2016


Anders Emil Nielsen

# Acknowledgements

I would like to thank my....

# Contents

Chapter 1

# Introduction

Peergrade.io is a platform for facilitating peer-evaluation in university and high school courses. Currently the platforms serves multiple institutions and hundreds of students. A large part of the platform is calculation of various statistics about the courses run on the platform - calculations which take up a large amount of time and are to be recalculated due to small changes.

Peergradeio.io among other web application wants to achieve a great user experience, which require fast presentation of content to the user. One strategy to achieve fast presentation is caching. Current caching systems use a pull based caching strategy, where a value is computed when requested if the value is not present in the cache. These system uses a simple caching system (such as memcached or redis) that supports storing and looking up values by a key. Where most primary databases need to support complex queries and structuring of data, key-value stores such as memcached and redis only need to lookup keys and therefore allows data to be collected and presented fast. Furthermore these systems can be configured to only store data in memory to allow all lookups to be even fast.

Depending on the freshness requirements for a given value, the cache needs to be invalidated. When the freshness requirement is not strict, the cached content can be invalidated after some time period. Another technique is to include some value in the cache key that is known to change when update when the underlying

model changes. Problem Existing caching solutions are based on pull based caching strategy, which is based on a simple model, but it only updates the cache "after-the-fact". This means the first user that requests the value after the cached value has expired will be presented with either stale data or have to wait for the computation to run. This problems presents two challenges, which will be addressed in this thesis.

Caching long running computations. With a pull-based strategy where the content is cached "after-the-fact", the first user requesting a given version of an object has to wait for the computation to finish. In some cases the computations takes a long time to compute leaving the user waiting and affecting user experience. In these cases the value has to be precomputed before the first user requests the content. This leads to challenges on how to ensure the values are recomputed when the source code changes and how to compute the values efficiently without recomputing values unnecessarily and compute the values as soon as possible.

Cache Invalidation. Beside knowing how to precompute the values, it is also necessary to know when the values needs to be precomputed. Most computations rely on underlying data from a storage system and possibly other cached computations. When the underlying data changes, the depending computations becomes stale and they need to be recomputed. The task of manually managing cache invalidation places a burden on the developer and leads to errors.

## 1.1 Architecture and Model

The system designed in this thesis consider the environment as the one illustrated in figure (Insert figure please). We assume the system consists of one or more applications servers that interact with a database system. The web application serves request from the client (i.e. user through browsers and other services using the web application through an API).

In order to achieve a flexible system, the design of the system will mostly be in the application layer through a caching library used in the web application. This will be designed with as few assumptions about the cache and database as possible. This way the system is able to use multiple types of caches and storage systems. The assumptions made about the cache system is that it should work as a key-value store that supports LOOKUP(key) and STORE(key, value). The system will be designed to work with commonly used storage systems used in web applications and implemented using a MongoDB database as storage system.

## 1.2 Requirements

The requirement for the system are based on real-life examples from Peergrade.io, where a lot of them can be related to common web applications. The system must support the architecture described and must be able to persist the cached values (such that they are not swiped on memory-overload etc.). The system will be designed from 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.

**Scalability:** Should be designed for scalability in the sense that the design should still be correct when the amount of underlying data for the web application rises or when the web application is scaled horizontally.

**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.

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

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

## 1.3 Contributions

This thesis addresses challenges described in section (ref. to problem section) in the context of an application-level cache 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 learn about and extend in further research. The implementation will therefore also have a focus on providing well-designed and modular libraries that solves the problem.

The rest of this text should probably be written when we're further in the process and knows more about the result.

Chapter 2

# Caching Approaches

Many caching approaches for improving the throughput of web applications have been proposed. We discuss the approaches closest to the challenges of this thesis. The discussion will consider existing approaches with relation to the requirements for the system we design.

## 2.1 Caching Granularity

One important aspect of caching is the granularity of the cached content. There is no doubt that it is most desirable to be able to cache any granularity, but since the cached content could be anything, the system cannot make any assumptions about cache management to e.g. allow for smarter invalidation. Therefore most existing work are based on a specific caching granularity from the data queried from the database to the HTTP response sent to the client.

### 2.1.1 Database Query Caching

The database can be a bottleneck in the goal of achieving fast rendering of dynamic pages, because it's often a requirement to have structured data at

which you perform complex queries. In both cases the queries can become slow when the application need to scale with relation to data or users. Even though most storage systems allows indexing to optimize specific queries, it can still be difficult for the storage system to optimize in a space efficient way. One solution to this problem is to use query caches.

In [?, ?] this problem is solved using a proxy caching server between the web application and the database. This allows for transparent caching that require almost no changes to the system, but unfortunately it requires a lot of work to maintain the index used for cache validation and parse the queries received from the web application. Furthermore this solution are mostly made for relational databases with SQL language and require a new implementation for it to work on different storage technologies such as document-oriented databases.

[?] also describes a query caching solution, where the cache management is placed entirely in the application-layer. It is based on a common technology used in web application called Object Relational Mapper (ORM), where the data model is mapped to objects in the application and often the queries are made using methods on the object. Using the ORM in the Python framework Django, [?] implements an extension that allows common database queries to be cached and automatically updated. Compared to having a middleware caching layer, this solution has the advantage of being able to integrate with both different database technologies (within the capabilities of the ORM) and caching systems. On the other hand, it does not capture database updates made without using the ORM. This means if updates are made manually or another application uses the same database, the cached queries are not updated.

### 2.1.2 Materialized Views

Where the query caches described so far are either middle-tier or on the application-layer, caching using materialized views occurs on the database layer. Materialized views are "virtual tables" generated from other data in the database. It works by storing queries explicitly declared by the developer. The virtual tables can be explicitly refreshed or update when the dependent data changes. Materialized views is a good solution for optimizing database queries, but since the computation occurs on the database level, the computation capabilities are limited by the database technology.

Caching database queries and materialized views allows for easy and transparent caching, but it does not allow computations on the web application, which limits the applicability.

### 2.1.3 Function Caching

A more flexible technique is to cache functions. [?] describes a programming model for cacheable functions that essentially is functions annotated as cacheable. Although this seems attractive, it has limitations with respect to the procedures executed in the function. [?] describes the requirement of cachable functions of their programming model: "To be suitable for caching, functions must be pure, i.e. they must be deterministic, not have side effects, and depend only on their arguments and the storage layer state." By this definition they explain that the storage layer state are treated as implicit arguments and thereby reach the classical definition of a pure function that is a transformation that always gives the same output from the same input.

[?] suggests a similar programming model, where the relationships of the underlying data has to be explicitly annotated, but where the rest of the caching system is much simpler than the one described in [?].

### 2.1.4 HTTP Response Caching

On the other end of the granularity scale, the developer could choose to cache the entire HTTP response send to the user. This could be the HTML documents served to the user as the website, but it could also be the JSON or XML response from an API. The HTTP protocol is the standard among web browsers to display web content and it's widely used to communicate between web services. Since the HTTP protocol also include caching methods, which will be explained in the HTTP section, it is a very attractive caching technique among web applications.

The HTTP protocol includes multiple mechanisms for controlling cache consistency that allows the web server to implement both key-based and expiration-based cache invalidation. These mechanisms are controlled using HTTP headers. The expiration-based cache invalidation information about the cache date and age is specified using the Cache-Control header. The client is then able to derive if a given resource is valid at a given time or if the resource has to be refreshed. To use key-based invalidation, the web server can attach a tag that uniquely identifies a given version of a resources (e.g. using a hash of the content). When the client sends a new request, it attaches the ETag of the last version received, and the web server can now respond with a 304 Not Modified with no content. This tells the client it can safely use the last version.

Caching HTTP responses is a great technique when the same response are served to the multiple clients, but in situations where the content is updated often

or personalized to each user, it becomes a less efficient technique since large documents are recomputed often. In the case where a small fragment of the content is personalized, it would be more desirable to only update that given fragment instead of recomputing the full document.

## 2.2 Cache Invalidation Techniques

One of the challenges of cache management is to maintain proper consistency between the underlying physical data and the cached data. If a cache system maintains strong consistency we know that when a cached value is read, it is a transformation of the most recent version of the underlying data it depends on. In the less strict model, weak consistency, it is only guaranteed that a cached value eventually becomes consistent with the most recent version of the underlying data.

### 2.2.1 Key-based Invalidation

One method to achieve strong consistency is to use key-based invalidation when the cached value is requested. 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. [**?**]. 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. In order for this caching method to work correctly, the developer must derive a cache key function f(x) such that the result of f(x) must be the same at any given time when given the same input x. 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. Although it simplifies the cache storage, the method also generates cache garbage, since old versions of a cached value are not removed. This means that the system relies on a cache database that is able to replace cache values using a proper policy such as replacing the Least Recently Used (LRU) or Least Frequently Used (LFU) [**?**].

### 2.2.2 Trigger-based Invalidation

Instead of invalidating the cached value when requested, the cached values can be invalidated based on certain triggers, which also guarantees strong consistency. This will make the code for requesting the cached value simpler, since the key used for storing the cached value does not have to update in lock-step with the underlying data.

The simplest triggers are write-through updates, which are manual triggers inserted by the developer at places where the cached values should be invalidated. This require all developers to keep track of all places where underlying data changes, and furthermore be sure that the manual triggers are inserted when new code is introduced.

In some architectures the changes to the system are based on triggers or events. One such architecture called Event Sourcing works by using Domain Events to describe the changes of the system instead of using database commands [**?**]. Since these events are a natural part of the application, they can be used as invalidation triggers without further additions.

A lot of work has been put into using triggers from the database to invalidate cached data. [**?**] suggests a solution based on the Object Relational Mapper programming technique to capture relevant triggers. [**?**] also suggests using a database wrapper in the application-layer that captures and analyzes database commands and use them as triggers. TxCache suggested in [**?**] uses daemon processes to monitor the database for relevant triggers. This method has the advantage that it allows multiple types of applications to manipulate the same database as opposed to [**?**, **?**], where the triggers would not have been captured if the database command was made around the application. On the other hand it introduces complexity of running and monitoring the processes. With relation to database monitoring this introduces a trade-off between the complexity of the system and an assumption that multiple types of applications does not alter the same data.

### 2.2.3 Expiration-based Invalidation

The requirement for strong consistency introduces complexity as seen with the trigger-based and key-based cache invalidation. Some objects can be cached with weak consistency, which allows much simpler caching techniques. One method is to assign a TTL (Time to Live) to the cached value. At some point when the TTL has expired, the cached object is invalidated. The invalidation

can be enforced by the cache database (Redis and Memcached supports this - TODO: include references) or as part of the protocol between the client and server as with HTTP-caching explained in section ? [?].

## 2.3 Automatic Cache Invalidation

In [?] Dan Ports et. al. uses database triggers to achieve transactional consistency for application-level caching, which ensures that any data seen within a transaction, shows a slightly stale but consistent snapshot across the storage and cache system. The database triggers are implemented using two database daemons that monitors a slightly modified version of PostgreSQL. The suggested solution, called TxCache, promises a very strong consistency guarantee, but it also comes with assumptions about the storage system and cache system used, and requires additional running daemons, which makes the full system more complex to run reliably. Furthermore these requirements contradicts flexibility and adaptability since the system assumes specific properties from the storage system and cache system, which makes it more difficult to change these components and adopt the caching system if an existing system does not use the given components.

Another solution proposed by Chris Wasik et. al. [?] uses deploy-time analysis of the code to detect dependencies between the cached functions and the dependent relations. To invalidate the cached functions automatically, the system injects code that invokes relevant invalidation callbacks in places where the underlying data is updated. Where [?] suggests a system that comes with requirements for the architecture and technologies used, the deploy-time model is a simple system that is able to use simple key-value stores for caching and any SQL storage system. But as oppose to [?] it does not result in as strict consistency guarantees. But despite of being a simple method, the deploy-time model is based on a system where the source code changes for different environments, which could cause errors in one environment and not in others. As an example, the code in a development or test environment could work as expected but still result in errors when deployed to a production environment, where the code is injected. Even though the deploy-time solution avoids single points of failures as with a cache manager, it needs additional operations that have to be executed in the existing procedures. In a system with complex dependencies between the procedures and underlying data, the generated source code could decrease performance that cannot be optimized by the developer using the system.

CacheGenie is another cache system described by Priya Gupta et. al. that uses the Object Relational Mapping (ORM) library to detect changes made to the

database. Some ORM libraries already implements these triggers, which makes this approach easier to integrate into web applications that uses ORM libraries, since the caching library does not rely on database monitors. CacheGenie tries to solve the problem of managing cache invalidation when caching database queries, by letting the developer predefine cached queries that are automatically updated in the application. CacheGenie is also based on a simple model, but each the cache definitions are based on assumptions about the specific queries and cannot be used to cache objects of a more coarse granularity.

On the other end of the granularity scale, [**?**] 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.

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 [**?**, **?**]. 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.

## 2.4 Discussion

The system need to cache the result of computations, which means it has to cache objects with a granularity more coarse than database queries. The HTTP protocol includes multiple features for cache management between the client and server, but it also makes the caching inflexible with relation to efficiency. In some situations HTTP responses includes shared fragments that need to be computed for each HTTP endpoint. Since the system expects long running computations, it would be more efficient to cache the result of those computations, meaning the system need to work on a granularity of fragments or functions. Since functions returns an output that could be considered a fragment, we will consider them as the same granularity.

In order to keep the cached values up to date we need automatic cache invalidation. In order to achieve automatic cache invalidation with transactional consistency, [**?**] describes a solution a solution that need assumptions about the storage and cache system. The system designed in this thesis does not need such strict consistency guarantees, and some of the components in the solution is therefore not necessary.

The solution suggested by Jim Challenger et. al. is highly relevant to the problem of this thesis, but since the system is designed for a publishing system, it leaves some challenges that have to be solved to satisfy the requirements of this thesis.

Also relevant is the paper by [**?**] that suggests a solution that identifies dependencies and injects invalidation callbacks into the source code on deployment. This method makes the caching transparent, but it also comes with the cost that the code will be different in development and production. Furthermore the solution described does not perform write-through updates, which would also be inefficient since it would slow down existing operations that involved cache invalidations.

Conclusion from this chapter: solutions exists for a lot of the sub-problems this thesis is facing, but there exists no complete solution to satisfy the requirements.

Appendix A

# Stuff

This appendix is full of stuff ...