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

Not Real

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:

```python
def course_score(course)
        participants = ParticipantDB.find_by_course(course)
        total_score = 0
        for participant in participants:
                total_score += participant_score(participant)
        return total_score / len(participants)

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

In this example we have a function `course_score` that computes the average score in a given course 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

Chapter 2

Caching Model

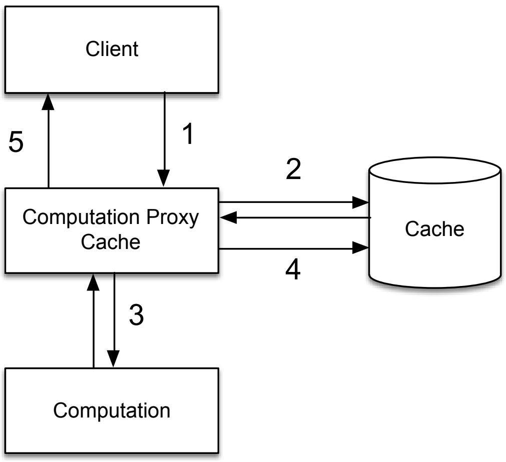
In order to get a common understanding of caching and the terminology related to the topic, we will go through the basics of caching by describing the architecture of a web system using a cache, present a model based on a timeline that introduces the different events involved in caching and last we list 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 value 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 value 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

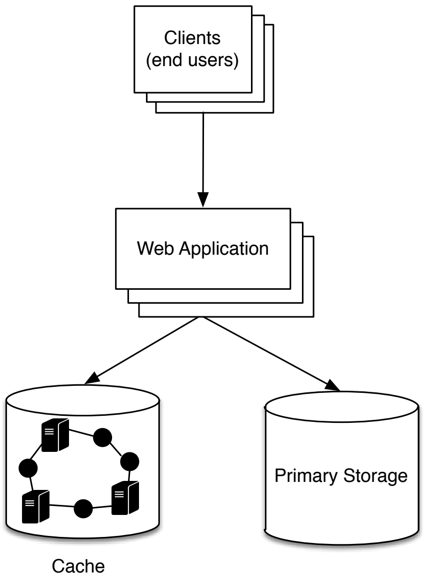


Figure 2.2: The assumed architecture of the system

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

2.2 Evaluating Caching Techniques

[3] http://redis.io/

[4] https://memcached.org/

Chapter 3

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.

3.1 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.

3.1.1 Expiration-based Invalidation

3.1.2 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. [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. 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) [Wan99].

3.1.3 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 [Fow06]. 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. [GZM11] suggests a solution based on the Object Relational Mapper programming technique to capture relevant triggers. [Was11] also suggests using a database wrapper in the application-layer that captures and analyzes database commands and use them as triggers. TxCache suggested in [Por12] 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 [GZM11, Was11], 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.

3.1.4 Automatic Invalidation

In [Por12] 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. [Was11] 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 [Por12] 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 [Por12] 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, [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.

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.

3.2 Updating the Cache

3.3 Caching Approaches in Web Development

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.

3.3.1 HTTP 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.

3.3.2 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 [ASC05, SG12] 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.

[GZM11] 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, [GZM11] 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.

3.3.3 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.

3.3.4 Function Caching

A more flexible technique is to cache functions. [Por12] 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. [Por12] 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.

[Was11] 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 [Por12].

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 [Wan99].

3.4 Choosing the right Caching Technique

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, [Por12] 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 [Was11] 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.

Chapter 4

Cachable Functions

Chapter 5

Automatic Cache 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 => [score1/2, 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

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

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 $s \rightarrow s'$.

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.

Bibliography

[ASC05] Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 264–273, New York, NY, USA, 2005. ACM.

[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.

[doca] Memcached protocol documentation.

[docb] Transactions in redis.

[Fow06] Martin Fowler. Focusing on events, January 2006.

[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.

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

[SG12] Sumita Barahmand Shahram Ghandeharizadeh, Jason Yap. Cosar-cqn: An application transparent approach to cache consistency. *Database Laboratory Technical Report*, june 2012.

[Wan99] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999.

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