CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Cachematic

Automatic Invalidation in Application-Level Caching Systems

Master's thesis in Algorithms, Languages & Logic

VIKTOR HOLMQVIST & JONATHAN NILSFORS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Cachematic

Automatic Invalidation in Application-Level Caching Systems

Viktor Holmqvist & Jonathan Nilsfors

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Cachematic
Automatic Invalidation in Application-Level Caching Systems
Viktor Holmqvist & Jonathan Nilsfors

Supervisor: Philipp Leitner, CSE
Advisor: Waldron Faulkner, Bison
Examiner: John Hughes, CSE

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Cachematic
Automatic Invalidation in Application-Level Caching Systems
Viktor Holmqvist & Jonathan Nilsfors
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Caching is a common method for improving the performance of modern web applications. Due to the varying architecture of web applications, and the lack of a standardized approach to cache management, ad-hoc solutions are common. These solutions tend to be hard to maintain as a code base grows, and are a common source of bugs.

In this thesis we present *Cachematic*, a general purpose application-level caching system with an automatic cache management strategy. *Cachematic* provides a simple programming model, allowing developers to explicitly denote a function cacheable. The result of a cacheable function will transparently be cached without the developer having to worry about cache management. The core component of the system is a dependency graph containing relations between database entries and cached content. The dependency graph is constructed by having the system listen to queries executed in a database. When a select query is detected within the scope of a cacheable function, the query is parsed and used to derive the dependency graph. When inserts, updates and deletes are detected, the dependency graph is utilized to determine which cached entries are affected by the modification. To evaluate *Cachematic*, a reference implementation was developed in the python programming language.

Our experiments showed that the deployment of *Cachematic* decreased response time for read requests, compared to a manual cache management strategy. We also found that, compared to the manual strategy, the cache hit rate was increased with a factor of around 1.64x. On the contrary, a significant increase in response time for write requests was observed from the experiments.

Keywords: computer science, web applications, caching, cache invalidation, cache management, application-level caching

Acknowledgements

First of all, we would like to thank our supervisor Philipp Leitner for his guidance and valuable feedback throughout the entire project. Additionally, we would like to thank our examiner John Hughes for his feedback.

Second, we would like to thank our industrial advisor Waldron Faulkner and the entire team at Bison for their help and support.

Finally, we would like to express our gratitude to families and friends for their encouragement and genuine interest in the project.

Viktor Holmqvist and Jonathan Nilsfors, Gothenburg, 2018

Contents

List of Figures

List of Tables

1

Introduction

During the past two decades, the web has evolved from serving static interlinked documents to serving and executing dynamic web applications, with complexity traditionally reserved for desktop applications [1, 2, 3].

Additionally, modern web applications are often utilized by millions of users and the growth of the internet shows no signs of slowing down. Consequently, an ever increasing amount of data needs to be processed and served [4]. As web applications have become more and more complex over time, the need for processing data in an efficient way within web applications has become of great importance. A general approach for improving performance in computer systems is the concept of caching. Caching can be employed on multiple levels, for example in a network [5, 6], in a computer or on a single CPU [7]. The purpose of a cache is to temporarily store data in a place that makes it accessible faster compared to if it was fetched from its original source [8].

In context of the internet, a common approach is to cache entire documents or web pages. Static documents can be cached in the browser using *If-Modified-Since* or similar concepts, whereas dynamic documents are usually cached on the serving side, to retain direct control of validity. When a dynamic page is requested in the browser, the server generates the page and the page is then cached. If the same requests is being sent to the server again, the page can be served from the cache [9]. In modern web applications, it is common to use *Javascript* to fetch data from the server and then create parts of the page or the entire page on the client side of the application [10, 11]. The client side rendering approach allows for not having to reload the entire page when for instance a user presses a button that should display some small amount of information. Instead of re-rendering the whole page, a single request can fetch the data needed and then only the affected fragment on the page needs to be re-rendered with the newly fetched data. In a setting such as client side rendering, caching entire web pages might be inefficient.

1.1 Application Level Caching

Application level caching is the concept of caching data internal to an application. The most common use case is caching of database results, in particular for queries that are executed frequently and involve significant overhead [12]. The cache is often implemented as a key-value store or in-memory database that ensures quick access from the application [13, 14]. Ultimately, the advantage of application level caching is no different from any other cache. It improves performance by reducing time needed to access some requested data.

One of the greatest challenges with application level caching, and caching in general, is cache management, i.e. keeping the cache up to date when the underlying data changes, and avoiding stale or inconsistent data being served from the cache. One of many examples illustrating the complexity in cache management is a major outage of Facebook caused by cache management problems [15].

A common method for cache management is cache invalidation. Cache invalidation works by directly replacing or removing stale data in a caching system [16, 17]. Compared to other methods such as time-based expiry or validation of cache entries on each request [18], cache invalidation is particularly well suited for frequently updated dynamic data. It is implemented by explicitly purging or replacing entries in the cache when they are deemed invalid. In order to determine when a cache entry is invalid, the system needs to know what resources, such as database results or data from other external sources, were used to derive the cache entry. Whenever those resources are updated, the entry should be purged or updated.

Another important aspect of cache invalidation is the granularity of the invalidation process, illustrated by the following example. A cache entry consists of a set of tuples R from database relation T. With course-grained invalidation, the cache entry could be invalidated by any update on the table T. In a more granular setting, the cache entry could be invalidated only if the selected tuples R are actually affected by an update. If the cache entries consist of tuples from multiple relations with joins and complex where clauses, the task of determining whether a cache entry should be invalidated becomes more complicated. The desired level of granularity heavily depends on the frequency of updates, and fine-grained invalidation might involve significant overhead. The balance of granularity and overhead has to be considered when invalidating cache entries for optimal performance [19, 20].

In this thesis we present the design and an implementation of *Cachematic*, a general-purpose middle-tier caching library with an automatic invalidation strategy.

1.2 Bison

The company Bison has developed an application-wide caching system for the web API of their business intelligence platform. The web API is implemented in Python and utilizes relational databases as primary storage. The caching system uses a simple dependency table to keep the cache up to date with the database. The system employs a decorator interface, similar to the cacheable function interface proposed in this thesis. The dependency graph is managed manually by the developers, by specifying dependencies as strings returned together with the result to be cached. Whenever the database is updated, the relevant strings are looked up in the dependency graph to identify cache entries to be invalidated. The strings relevant for each update is determined manually by developers on implementation. The initial deployment of the caching solution successfully improved the responsiveness of the system. As the application has grown in size, and the caching solution has been deployed throughout the system, the complexity of the dependency graph has grown to be incomprehensible by any one developer, resulting in sub-optimal and erroneous invalidation of cache entries.

1.3 Thesis Statement

Application level caching has become the ubiquitous solution to scaling web applications [21]. Ad-hoc implementations of cache management are common when deploying application level caching, and often cache management is manually handled by developers. The characteristic of a manually handled cache is that developers have to explicitly cache data when appropriate and invalidate or change cached entries when these entries become outdated. The task of manually managing the cache is tedious and error prone [22]. Especially, this approach does not scale well with a code base that grows in amount of code and complexity [23].

1.3.1 Objectives

The goal of this thesis is to research whether it is possible to devise an algorithm for automatic cache invalidation. We have formulated three research questions which lay ground for the work carried out during this research.

- Is it possible to devise an automatic dependency resolution algorithm for cache management with the goal to improve cache hit rates?
- In the context of the Bison web application, does the algorithm actually improve hit rates, and does this improve end user performance for specific, representative workload scenarios?

- Can the algorithm guarantee that all cache entries that needs to be invalidated actually get invalidated?

1.3.2 Methodology

To answer the questions stated in previous subsection, a software library (*Cachematic*) was designed, developed and evaluated. The development process was iterative and included three iterations. Each iteration followed a subprocess of development, testing and evaluation. The evaluation was carried out in the context of the Bison platform.

The main goals of the three iterations are listed below

1. Have a library that can cache results of functions and return cached results
2. Have complete parsing of sql queries and updates and naively invalidate cached results on updates
3. Have a fine grained dependency evaluation and invalidation integrated into Bison's platform.

1.4 Thesis Outline

Chapter 2 introduces necessary concepts needed to follow the thesis. We describe key value stores and SQL queries.

Chapter 3 presents previous research carried out on the subject of application level caching. The chapter ends with a comparison table describing how *Cachematic* differentiates from other solutions.

Following, in Chapter 4, is a thorough description of the design of *Cachematic* and the implementation created during this project.

Furthermore, Chapter 5 presents an evaluation of the library. Measures on response time and hit rate are reported as well as the overhead of the implementation. The results of the measurements are shown in relation to two baselines: the currently deployed solution in Bison and a solution without caching.

Lastly, in Chapter 6, we present our conclusions in a discussion. We also suggest potential interesting future research which could extend the research introduced in this thesis.

2

Background

2.1 Existing solution

The cache management solution used at Bison today consists of a simple dependency table, mapping arbitrary strings, called *dependencies*, to cache keys. In each function to be cached, the dependencies have to explicitly be returned at the end of the function. The cache key of the function call is subsequently inserted into the dependency table, once for each dependency.

```
@cache.decorator()
def funds(max_age=10):
    query = sqlalchemy.text("""
        SELECT * FROM fund WHERE age_years > :max_age
    """)

    funds = db.execute(query, max_age=max_age).all()

    # Add scoped primary key of each included fund
    dependencies = ["fund:{}".format(fund.id) for fund in funds]

    # Global dependency to invalidate on new funds
    dependencies.append("funds")

    return funds, dependencies
```

Figure 2.1: Example of a cacheable function utilizing the manual solution

Figure 2.1 shows a cacheable function utilizing the manual solution, and illustrates how the dependency strings are commonly generated. A dependency string is generated for each row returned in the query, consisting of the primary key prefixed with the table name to ensure uniqueness across tables. Additionally, a global dependency for the table is appended to account for edge cases where the cached entry cannot be identified by an existing row.

Dependency	Cache Keys
"fund:1"	"funds,max_age=5" "funds,max_age=10"
"fund:2"	"funds,max_age=10" "funds,max_age=12"
"fund:3"	"funds,max_age=12"
"funds"	"funds,max_age:5" "funds,max_age:10" "funds,max_age:12"

Table 2.1: An example of how the dependency table might look after executing the function in Figure 2.1

Bison has established developer guidelines for dependency generation, to ensure the same patterns are used throughout the application. In many cases, in particular with complex queries, it is still very hard to determine the dependency strings required to cover all edge-cases. An example of how the dependency table might look after executing the function is shown in Table 2.1.

```
def update_fund_age(fund_id, age_years):
    query = sqlalchemy.text("""
        UPDATE fund SET age_years = :new_age WHERE id = :fund_id
    """)

    result = db.execute(query, new_age=age_years, fund_id=fund_id)

    cache.invalidate("fund:{}".format(fund_id))

    return result
```

Figure 2.2: Example function updating the table queried in Figure 2.1

```
def create_fund(name, age_years, commitment):
    query = sqlalchemy.text("""
        INSERT INTO fund (name, age_years, commitment)
        VALUES (:name, :age_years, :commitment)
    """)

    result = db.execute(query, name=name, age_years=age_years,
        commitment=commitment,
    )

    cache.invalidate("funds")

    return result
```

Figure 2.3: Example function inserting into the table queried in Figure 2.1

Figure 2.2 and Figure 2.3 illustrates the manual invalidation process, explicitly invalidating hard coded strings as determined by the developers on implementation. The

function in Figure 2.3 invalidates the global dependency string `"funds"`, since no fund-specific dependency can exist for a newly created fund. This specific problem is a common edge-case and has caused numerous bugs in the Bison web application. In addition to causing bugs, utilizing table-scoped dependency strings will result in over-invalidation. Reducing over-invalidation by adding more granularity

2.2 Key-value stores

A key-value store is a database designed to hold a data structure mapping keys to values, similar to a hash map or dictionary. Each value in the database is uniquely identified and indexed by its key. This enables retrieving, modifying and deleting values from the database in constant time. To further improve performance of these operations, key-value stores commonly store most data in memory [24, 25]. The key-value architecture combined with efficient read and write operations make in-memory key-value stores a good storage solution for caching systems [26]. Examples of key-value stores are *Redis*[1], *Memcached*[2] and *Amazon DynamoDB* [27]. In the library implemented in this thesis, *Redis* was used as the cache backend and for storage of the dependency graph.

Redis

Redis is a distributed in-memory key-value store. It supports multiple data types such as lists, sets, strings and hashes. To utilize Redis for more complex data structures, such as objects in high level programming languages, the objects have to be serialized and stored as strings. Furthermore, Redis supports simple operations on supported data types. For instance, it is possible to push entries to a list and perform set operations on sets stored in Redis. Redis can be configured with different eviction policies such as LRU (least recently used), TTL (time to live) and random eviction. The eviction policy determines what happens when a Redis instance runs out of memory. By default, Redis uses LRU eviction, meaning that the entry least recently used will be evicted to leave space for new entries. Research has shown that an eviction policy optimized for data access patterns and other application specific parameters can be beneficial , but it was considered out of scope for this thesis, and Redis was configured to use LRU eviction in the evaluation [28].

[1] https://redis.io/
[2] https://memcached.org/

2.3 SQL

SQL is an abbreviation of Structured Query Language. It is a programming language designed to manage data in relational database management systems. Mainly there are four different types of queries that are used to read and write data to and from a database. The four types are described in some detail below. For a more thorough explanation of SQL we refer to the W3Schools tutorial on SQL[3].

2.3.1 Insert query

Insert queries are used to insert a new row into a table. The insert queries specify what table the row should be inserted into and what values the row should contain. There exists a few different variants on the syntax of inserting one or more rows. An example is shown below.

```
INSERT INTO table_name (column_1, column_2, column3, ...)
VALUES (value_1, value_2, value_3, ...)
```

2.3.2 Select query

Select queries are used to read data from the database. A select query always consists of the columns(the syntactic sugar * can be used to to represent all columns) to be selected and one or more tables from where the data is supposed to be retrieved.

```
SELECT column_1, column_2, column_3
FROM table_name
```

Optionally, select queries can contain a where clause, explained in detail in 2.3.6, to specify filters on what data rows to fetch. A number of aggregate functions can also be utilized to retrieve meta information. An example of an aggregate function is count which counts the number of rows selected by the query. Select queries also support ordering of rows and limiting the number of results returned. It is also possible separate the result of a select query in different groups. Furthermore, select queries support joining which essentially means connecting different tables on some condition. Joining is explained further in Section 2.3.5.

```
SELECT table_1.column_1, table_2.column_2, column3, ...)
FROM table_1
JOIN table_2 ON table_1.column_1 = table_2.column_1
WHERE table_2.column_4 < 10 AND table_1.column_5 IS NOT NULL
ORDER BY table_1.column_1
LIMIT 5
```

[3]https://www.w3schools.com/sql/

2.3.3 Delete query

Delete queries are used to delete rows in a table. Usually delete queries are defined with some condition, similar to select queries. The condition specifies which rows should be deleted. If a condition is excluded from the delete query, all rows in the specified table are deleted.

```
DELETE FROM table_1
WHERE table_1.column_1 = 1
```

2.3.4 Update query

Update queries are used to update a set of rows in a table. An update query contains the name of the table to update, the columns that are to be updated and what values these should have, together with an optional condition. The update query looks a little like a combination of a select and an insert but it is never creating new rows. It only updates already existing rows. The number of columns being updated can be arbitrary. The update does not have to update all columns.

```
UPDATE table_name
SET column_1 = value_1, column_2 = value_2, ...
WHERE table_name.column_2 < 10
```

2.3.5 Join clause

When two or more tables contain data related to each other it might be desirable to correlate them. This can be achieved by utilizing a join clause. A join clause contains the table to join and an optional condition describing how to join. The result of a join is a new temporary table containing data from the both tables that were joined. Which rows are included and how they are concatenated depends on the join condition. If the join condition is excluded, all rows in both tables will be concatenated; for instance if the first table has 5 rows and the second table has 2 rows the resulting join table will have 10 rows. Usually, a condition is provided to correlate rows. Depending on what type of join being executed, different resulting tables will be achieved. Joins can be applied multiple times sequentially to correlate more than two tables.

2.3.6 Where clause

The where clause is used to filter what rows are being acted on by the query. There exists a wide variety of filter conditions. Some of the most common ones are logical

operators *AND*, *OR* and *NOT*, equality check =, less than < and greater than > and *LIKE*. Different dialects also provide their own filter conditions.

2.3.7 Subqueries

Since the result of a query is a temporary new table, the result of a query can also be used in another query. A query that is used within another query is usually called a subquery or an inner query. Subqueries can be used in all query types described above.

3

Related Work

In this chapter we present related research that exist on the subject of application level caching and in particular aspects of cache management and invalidation strategies.

A Qualitative Study of Application-Level Caching [21], presents a study where ten open source web application software projects were studied in depth. The goal of the study was to extract information on how developers for the different projects handle caching. From the extracted information, some guidelines and patterns were derived which purpose is to help developers in their work on designing, implementing and maintaining application level caches.

One of the guidelines stated is that developers need to evaluate different abstraction levels in the cache. There are multiple layers of data processing in an application and in general cached data closer to the model or database layer offers a higher hit rate than cached data from a business or controller layer. The authors also identified another interesting guideline regarding per user caching. In general, too specific data does not improve hit rates. This is for instance the case if the cached data is specific to a certain user. However, if users tend to utilize the application for an extended period of time, requesting specific data multiple times, it might be desirable to cache user specific data. Especially, if the user requests the same or very similar data multiple times during a time limited session.

Furthermore, the authors found evidence for a guideline they call *Keep the Cache API Simple*. The purpose of this guideline is to highlight the complexity in caching logic when it is spread over an application. The consequence of not having a simple caching API might be messy code and high costs of maintenance.

3.1 Automatic cache management strategies

We have identified a collection of papers describing automatic cache management strategies, most of which are implemented as middle-tier caches that augment the database with a key-value store.

Dependency based cache management is introduced in [29], as an optimization for the caching system used in the 1998 Winter Olympics website. The main algorithm is called DUP (Data Update Propagation). The algorithm describes the construction of a graph for tracking dependencies between cached objects and underlying data, called ODG, or Object Dependence Graph. In the graph, cached objects, such as intermediate html fragments or html pages, and underlying data is represented by nodes. Dependencies are represented by edges. This means that cached objects can be dependent on other cached objects, such as html fragments, as well as underlying data. When underlying data is invalidated, the algorithm simply finds the corresponding node and traverses the graph to find all nodes directly or indirectly dependent on the invalidated node. The visited nodes represent cached objects that should be invalidated, and thus evicted from the cache. The approach described in this paper is very general and applicable in almost any currently existing application. It requires a considerable amount of responsibility from the application utilizing it and has no specification of how dependencies are supposed to be extracted nor in what granularity they should be recorded.

An extended version of DUP, was implemented in the Accessible Business Rules framework (ABR) for IBM's Websphere [30]. The paper extends DUP with a concrete process for automatically constructing the ODG by analyzing SQL queries. It also extends the graph by annotating edges (indicating dependencies) with values used in the query where clause, enabling *value-aware* invalidation. In this paper, the dependency graph is constructed at compile-time, with the exception of parametrized statements, for which the graph is prepared but finalized with actual parameters at run-time. Selective invalidation of the cache is triggered by invalidation code in the attribute setter, creation and deletion methods. The code is automatically generated at compile time.

TxCache is a transactional caching system, where dependencies are represented by invalidation tags [31]. A tag is a description of which column has been referenced in the database to produce a cached result. A tag consists of two parts separated by a colon. The first part represents the database table and the second part a potential referenced column. The second part is set whenever an index equality lookup is performed. If the query for instance is a range query, the second part is explicitly set to a wildcard. Each query executed can have multiple invalidation tags. When a write to the database is executed the database sends a stream of invalidation tags to the cache. The cache can then identify which cached invalidation tags are affected by the write and consequently identify affected cached entries. The system proposed in this paper extends the transactional consistency guarantees of the database to the cache. In order to achieve this, most of the cache management logic is implemented in the database layer, through modifications to PostgresSQL.

Another strategy is implemented in *AutoWebCache* [32], where dependencies between read and write queries are established by finding shared database relations and fields. If the queries share any fields, a basic dependency is established and stored in a data structure resembling the Object Dependence Graph used in DUP. The algorithm also stores information about the database set related to the queries.

When write queries are executed, actual intersection is evaluated in a more precise manner for each of the dependent read queries using the stored information. This evaluation supports three levels of progressively more refined analysis:

1. Table/column intersection, similar to the *fine-grained* analysis in [19]

2. In addition to table/column intersection, match the where clause of the read query to the values of the write query to see if the same rows are being updated.

3. The most refined case involves extending the previous policy by executing extra queries to get information missing in the write query, to be able to test if the same rows are being updated.

The basic dependency ensures the transactional consistency of the underlying database remains intact, as long as the more precise intersection test is sound. The dependency information is itself cached by query template (query without parameters). This cache stabilizes quickly since the number of read/write query combinations normally is relatively small.

A system with a trigger based strategy for cache management, *CacheGenie*, is described in [33]. The system generates database triggers to handle cache invalidation. Database triggers are procedural code that is executed automatically in response to events on a particular database relation. In this paper, the database triggers are simple, and their only job is to notify the cache manager that rows have changed. The actual cache invalidation is then implemented in the cache manager itself. Triggers have to be defined for each query type (insert, update and delete) for each database relation. The authors have chosen an explicit programming model handle this, explained in detail in 3.2. Another interesting concept implemented in CacheGenie is *Semantic Caching*. Semantic caching involves exploiting the semantics of the database in the cache management system, in order to automatically update cached objects instead of invalidating. This can have performance benefits over invalidation in systems with frequent writes, but limits what objects can be cached to database results.

Semantic caching is also employed in the approach described in [19], by checking for containment of a query's expected result within already cached ones. This paper handles dependencies between read and write queries by looking at shared fields, similar to the basic dependency mechanism in AutoWebcache. An interesting detail from this paper is an optimization for queries returning at most a single row, which under certain circumstances can never be invalidated by an insert.

A more formal approach to cache management is described in [34]. The system, *Sqlcache*, is based on compile-time SQL analysis and first-order-logic to create a sound mapping from each update operation (insert, update or delete) to read queries they affect. This mapping is then used to transparently add caching with automatic invalidation. Essentially, the implementation uses the filter variables in the where clause of an SQL query and the filter variables together with the update vector for database updates to determine if invalidation is necessary. In the paper, three

situations where invalidation is required are identified:

- When a row that was formerly not included in a set of rows according to some predicate becomes part of the set according to the predicate
- When a row that formerly was included in a set of rows according to some predicate becomes excluded from the set according to the predicate
- When a row that was formerly included in a set of rows according to some predicate and still is part of the set but with different values

The authors provide a proof of soundness for cache invalidation using quantifier-free first order logic based on the these three situations.

CachePortal, is a cache management system described in [35]. The system largely depends on a sniffer module which task is to log http requests, database queries and mappings between requests and database queries. When a database update query is captured, the query is analyzed to conclude which cached entries need to be invalidated. Whether to invalidate a cached entry or not is determined by comparing the update query to each select query executed to compute a cached entry. The comparison algorithm checks if the where clause for the select query is satisfied by the update query. If the select query is a join of multiple tables and there is not enough information to conclude if the update satisfies the select where condition, an extra polling query can be performed. The system identifies what data is missing to be able to evaluate the query comparison and performs a polling query to retrieve the missing data. When all data necessary for evaluation has been collected, the algorithm can soundly determine if a cache invalidation needs to take place or not.

Query change notifications is a fairly recent feature of Oracle and Microsoft SQL Server. It enables subscribing to changes in the result of individual SQL queries, thereby trivializing the process of implementing cache invalidation. This feature is utilized in [36], to implement *CQN* (Continuous Query change Notifications), an application transparent approach to cache management.

3.2 Programming model

One of the core features for an automatic caching system is to be as transparent as possible to the application implementing it. Any additional code the developers have to write in order to implement the caching system has the potential to introduce bugs or change the behaviour of the application. It also makes it more complex to reason about the semantics of the original application. It is therefore important to provide a simple programming model [21].

In TxCache [31], the results of a function can be automatically cached by using

a decorator function make-cacheable, illustrated in Figure 3.1, that generates a cacheable function. In for example the python programming language, this can easily be achieved by utilizing the decorator pattern. The wrapping function automatically makes sure the cache is being used whenever a call to the decorated function is executed. The only restriction on the developer is that the decorated functions must be deterministic, cannot have side effects and depend only on their arguments and the database state.

```
MAKE-CACHEABLE(fn) -> cacheable-fn
```

Figure 3.1: Programming model for the cacheable function in TxCache

SQLCache [34] is implemented in the domain-specific functional language *Ur/Web*, where the compiler does parsing and type checking of SQL to guarantee that queries adhere to the database schema. The compiler performs static analysis of the source code, including the queries, to find possible caching opportunities. This analysis is made possible by the fact that SQL queries are not strings, but first order constructs in the language. Enabling caching requires no more effort from the developer than passing a flag to the compiler. In the approach described in the paper, only database results are being cached.

CacheGenie [33] also caches database query results. The system builds on top of an ORM (Object-relational Mapping) and uses *cache-classes* to define what queries to apply caching to. Different query patterns are represented by different *cache-classes*. If a query result should be cached, the developer creates an instance of the *cache-class* representing the query's pattern. The main model representing the database column is being passed in as an argument when creating the *cache-class* instance. Once the *cache-class* instance is created, the developer can use the original code to query the database through the ORM. No modifications of the previous source code is required. By default, CacheGenie provides four types of query patterns; **Feature Query**, **Link Query**, **Count Query** and **Top-K Query**. If query patterns that are not included in the set of pre-defined cache classes need to be cached, developers can define their own *cache-classes*.

AutoWebCache uses an aspect oriented programming model to help transparently inject caching mechanisms in web applications. Aspect oriented programming (AOP) is a methodology used to modularize applications and separate concerns into *aspects*. The aspects are combined, or *wowen* together, as specified by *weawing rules*. This way, an AOP-based application implementing AutoWebCache can specify the points in the application where mechanisms for cache management need to be injected.

3.3 Comparison

In this chapter, we have presented a number of existing automatic cache invalidation systems. Many of them are pure database caches while others are more high level. Some solutions use compile-time analysis to statically determine caching and some require modifications to external sources such as database management systems. We end this chapter by comparing how the most relevant caching systems differ from each other and especially how they differ from *Cachematic*. A summary of the comparison is visualized in Table 3.1.

	Cached values	Analysis	Note
Cachematic	**Function results**	**Runtime**	
TxCache	Function results	Runtime	DB modifications
CacheGenie	Query results	Runtime	
AutoWebCache	HTML documents	Compile time	
Sqlcache	Query results	Compile time	Language specific

Table 3.1: Comparison table between automatic caching systems

TxCache

TxCache allows for arbitrary cacheable functions as long as the functions fulfill some requirements. The system requires no compilation but modifications to the PostgreSQL DBMS was required to support some of the features provided in TxCache. For the evaluation of TxCache, the RUBiS benchmark was used [37]. The evaluation showed that TxCache increased the throughput with up to 5.2x for the RUBiS benchmark. The RUBiS bidding mix workload, which provides 85 percent read interactions and 15 percent write interaction, was used [31].

CacheGenie

The system CacheGenie provides the ability to cache database results by declaring certain ORM classes to be cache classes. It provides some default caches representing a set of database queries and a developer can implement new classes if necessary. The system requires no compilation and no modifications to the DBMS. CacheGenie used Pinax[1] applications for benchmarking. The benchmark showed that using CacheGenie with cache invalidation improved throughput up to 2.5 times compared to using no cache. Furthermore, if updating cache entries rather than invalidating them, a 25 percent improvement could be observed compared to the invalidation strategy [33].

[1] http://pinaxproject.com/

AutoWebCache

AutoWebCache caches full web page documents created by dynamic data. It performs static analysis during compile time. To our knowledge, no database modifications are required. AutoWebCache was evaluated using the RUBiS benchmark and was able to provide up to a 64 percent improved response time when using the bidding mix workload.

Sqlcache

In Sqlcache, static analysis of queries in the source code is performed during compilation. The analysis identifies potential caching opportunities. Sqlcache caches results of database queries with no modifications of source code or database implementations. The throughput in relation to number of available threads was measured in the evaluation of Sqlcache. The evaluation showed that the throughput doubled for a specific scenario in comparison to the baseline ur/web compiler [34, 38].

Cachematic

Cachematic is a general-purpose automatic cache management system. It provides a programming model that transparently handles caching of function results by simply declaring a function as cacheable. Cachematic requires no compilation and no modifications to database implementations. The query analysis is performed at runtime.

4

Design & Implementation

The invalidation algorithm presented in this chapter is a combination of ideas from the papers described in Chapter 3 including optimizations and lessons learned from the manual invalidation system in use at Bison. The caching library is intended to eventually replace the manual system.

4.1 Algorithm

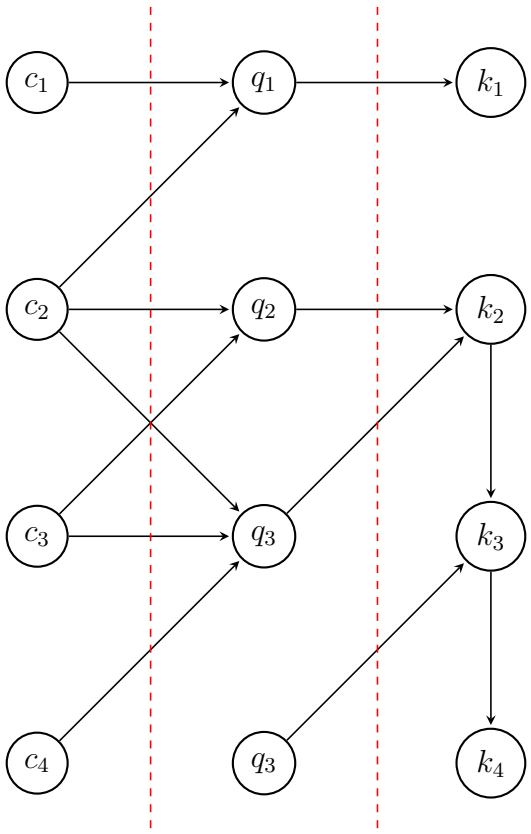
The algorithm is divided into three sub algorithms: the cache algorithm, the dependency algorithm and the invalidation algorithm.

Input : Function to be cached fn and arguments $(a_1, a_2, ..., a_n)$
Output: Return value of $fn(a_1, a_2, ..., a_n)$

1 $k \leftarrow genCacheKey(fn, a_1, a_2, ..., a_n)$;
2 **if** $scopeStarted(k)$ **then**
3 | wait until done;
4 **end**
5 $cached \leftarrow getCached(k)$;
6 **if** *cached* **then**
7 | return *cached*;
8 **end**
9 $startScope(k)$;
10 $result \leftarrow fn(a_1, a_2, ..., a_n)$;
11 $storeCached(k, result)$;
12 $(queries, nestedKeys) \leftarrow endScope(k)$;
13 $dependencyAlgorithm(key, queries, nestedKeys)$;
14 return result;

Figure 4.1: The cache algorithm

The cache algorithm handles caching of the return value of individual function calls. This process includes generating the cache key from the function name and the arguments of the function call and managing scope to capture read queries and nested function calls to pass to the dependency algorithm. The scope also prevents multiple equivalent calls to the same function to be executed simultaneously. The cache algorithm is illustrated in Figure 4.1.



The three layers of the dependency graph, where c_1, c_2, c_3 and c_4 are columns, q_1, q_2, q_3 and q_4 are read queries and k_1, k_2, k_3 and k_4 are function keys.

Figure 4.2: The dependency graph

The dependency and invalidation algorithms handle read and write queries respectively. The dependency algorithm utilizes meta data extracted from read queries executed within the scope of a cacheable function to construct a dependency data structure. The invalidation algorithm combines information from the dependency data structure with meta data extracted from write queries to identify read query candidates for invalidation and eventually carry out the invalidation.

The dependency data structure is modeled as a directed graph consisting of nodes in three layers, hereafter called the *dependency graph*, illustrated in Figure 4.2. The nodes in the first layer represent table columns, which are the entry points of the graph. The nodes in the second layer represent read queries. An edge is established from the first to the second layer if a column occurs in a read query. The nodes in the third layer are cache keys generated by the cache algorithm. A node in the third layer will have an incoming edge from the second layer if a read query was executed

during the scope of the cacheable function. If a cachable function contains a call to another cacheable function, an edge within the third layer will also be established from the key of the nested function call to the key of the parent call.

Input : Cache key k, captured read queries Q and cache keys of nested function calls N
Output: $Nodes$ and $edges$ to be added to dependency graph

$k_{node} \leftarrow KeyNode(k)$;
$nodes \leftarrow \{k_{node}\}$;
$edges \leftarrow \emptyset$;
for $q \in Q$ **do**
    $q_{meta} \leftarrow parseQuery(q)$;
    $q_{hash} \leftarrow computeResultHash(q)$;
    $q_{node} \leftarrow QueryNode(q, q_{meta}, q_{hash})$;
    $db \leftarrow dbId(q_{meta})$;
    $table \leftarrow tableName(q_{meta})$;
    add q_{node} to $nodes$;
    for $c \in columns(q_{meta})$ **do**
        $c_{id} \leftarrow db + table + c$;
        $c_{node} \leftarrow ColNode(c_{id})$;
        add c_{node} to $nodes$;
        add (c_{node}, q_{node}) to $edges$;
    end
    add (q_{node}, k_{node}) to $edges$;
end
for $n \in N$ **do**
    $n_{node} \leftarrow KeyNode(n)$;
    add n_{node} to $nodes$;
    add (n_{node}, k_{node}) to $edges$;
end
return $nodes, edges$;

Figure 4.3: The dependency algorithm

When read queries are passed to the dependency algorithm, the query strings are parsed to extract relevant meta data such as queried tables and columns. The tables and columns are used in conjunction with the database schema to include constraints such as primary and foreign keys in the meta data. In addition, boolean expressions from where and join clauses are converted to abstract syntax trees (ASTs) for easy evaluation. The ASTs generated from where and join clauses are also inspected to construct a set of tables filtered by primary key and a column equality map. These data structures are explained in detail in Section 4.1.2. Canonical column identifiers are produced by concatenating database identifier, table name and column name. The extracted information is then used to build the dependency graph.

Nodes are added to the first layer for each canonical column identifier, with edges to corresponding queries in the second layer. Query nodes consist of the query string, parameters and query meta data extracted earlier including a hash of the query results. The cache key generated for the call is inserted into the third layer with edges from each query node and the cache keys of nested function calls. The dependency algorithm is illustrated in Figure 4.3.

Input : Captured write query w and dependency graph D
Output: Set K of keys to be invalidated

$K_{direct} \leftarrow \emptyset$;
$K_{indirect} \leftarrow \emptyset$;
$Q \leftarrow \emptyset$;
$w_{meta} \leftarrow parseQuery(w)$;
$db \leftarrow dbId(w_{meta})$;
$table \leftarrow tableName(w_{meta})$;
for $c \in columns(w_{meta})$ **do**
    $c_{id} \leftarrow db + table + c$;
    $nodes \leftarrow$ lookup c_{id} in D;
    add $nodes$ to Q;
end
for $node \in Q$ **do**
    $(q, q_{meta}, q_{hash}) \leftarrow node$;
    $invalidate \leftarrow evaluate(w, w_{meta}, q, q_{meta})$;
    if $invalidate = yes$ **then**
        $keys \leftarrow$ lookup $node$ in D;
        add $keys$ to K_{direct};
    else if $invalidate = maybe$ **then**
        if $r_{hash} \neq computeResultHash(q)$ **then**
            $keys \leftarrow$ lookup $node$ in D;
            add $keys$ to K_{direct};
        end
    end
end
for $key \in K_{direct}$ **do**
    $parents \leftarrow$ lookup key in D;
    add $parents$ to $K_{indirect}$
end
$K \leftarrow K_{direct} \cup K_{indirect}$;
return K

Figure 4.4: The invalidation algorithm

In contrast to read queries, which are only processed in the scope of cacheable functions, write queries executed in any scope must be processed to ensure consistency. When a write query is received, the query string is parsed and processed in a similar fashion to read queries with some variation depending on the type of query. In the case of insert and delete, all columns of the affected table are captured since the query will result in an entire row being added or removed. For updates, only the affected columns are captured. Additionally, inserted rows and new values are extracted for inserts and updates respectively. Updates and deletes can also contain where clauses, which are converted to ASTs.

After processing the write query, the columns extracted are looked up in the first layer of the dependency graph. For matching nodes, edges are traversed to the next layer to identify corresponding read queries that will be considered for invalidation. Subsequently, the read queries are tested for invalidation using a series of tests, each a more granular attempt to exclude the query from further testing. The tests determine whether the query should be invalidated, excluded from invalidation or passed through for further testing. If invalidation can not be determined with certainty by any test, the query is passed to a final hash based test. In the final test, the hash of the read query results stored in the dependency graph, is compared to a hash of the results after the write query has been executed. If the hashes differ, the read query results changed due to the write query and any dependent function calls should be invalidated. Queries with unchanged hashes are excluded from invalidation. This ensures invalidation correctness. The exclusion tests are described in detail in Section 4.1.2 and the hash test implementation is described in Section 4.2.1.

Cache keys to be invalidated are retrieved by following the edges from the queries marked for invalidation to the third layer of the dependency graph and traversing the third layer recursively to capture functional dependencies. Invalidation is carried out by deleting the keys in the cache and removing associated nodes from the dependency graph. The invalidation algorithm is illustrated in Figure 4.4.

4.1.1 Invalidation scope

To avoid read queries being evaluated multiple times against multiple write queries within the same execution context, the concept of invalidation scope was introduced. Invalidation scope works similarly to the scope within a cacheable function in that it groups write queries executed within the same context to be processed in bulk. By recording write queries as they happen and deferring processing until the scope ends, evaluation of read queries can be remembered across write queries. For example, if a read query is marked for invalidation, it is not necessary to evaluate it again, and it can be skipped for the remaining write queries within the scope. Invalidation scope also enables many implementation specific optimizations, described in detail in Section 4.2.3.

4.1.2 Hierarchical invalidation

To reduce overhead of the invalidation algorithm, it is critical to reduce the number of read queries that are tested against incoming write queries. For this reason a hierarchical approach has been applied. This process starts with the first layer of the dependency graph, by excluding read queries without matching column nodes. It continues with exclusion tests optimized by query type.

Input : Write query w, read query r and meta data w_{meta} and r_{meta}
Output: Invalidation status yes, no or $maybe$

```
1  r_ast ← whereAst(r_meta);
2  if r_ast = Nothing then
3  |  if limited(r_meta) then
4  |  |  return maybe;
5  |  end
6  |  return yes;
7  end
8  if tableName(w_meta) ∈ pkFiltered(r_meta) then
9  |  return no;
10 end
11 for row ∈ rows(w) do
12 |  fields ← expand(row, r_meta);
13 |  (reduced, unknowns) ← reduceAst(r_ast, fields);
14 |  if unknowns = All then
15 |  |  return maybe;
16 |  end
17 |  if evaluateAst(reduced) then
18 |  |  if unknowns = None ∧ ¬limited(r_meta) then
19 |  |  |  return yes;
20 |  |  end
21 |  |  return maybe;
22 |  end
23 end
24 return no;
```

Figure 4.5: Query evaluation for inserts

A read query where the primary key of a queried table is tested for equality with a constant can never be affected by an insert into that table [19]. A set of tables filtered by primary key, where no other condition can satisfy the where clause, is included in the query meta data stored in the dependency graph. In the first exclusion test for inserts, read queries are excluded from further testing by checking if the table affected by the insert is contained within this set.

The second test for inserts involves evaluating the ASTs generated from the where clauses of the remaining read queries. For each inserted row, field references in the AST are replaced with matching values from the row. The ASTs are then evaluated. If every AST evaluates to false, the read query can be excluded from further testing.

Evaluation of the read query AST is the primary test for updates and deletes, but no rows are available to substitute for fields in the AST. However, equality conditions necessary to satisfy the where clause of the write query can be extracted and substituted for the fields in the read query AST. If there is no where clause in the write query, it is certain the read query is affected unless it is limited, and it can be passed directly to invalidation.

Input : Write query w, read query r and meta data w_{meta} and r_{meta}
Output: Invalidation status yes, no or $maybe$

```
1  r_ast ← whereAst(r_meta);
2  w_ast ← whereAst(w_meta);
3  if r_ast = Nothing ∨ w_ast = Nothing then
4  |  if limited(r_meta) then
5  |  |  return maybe;
6  |  end
7  |  return yes;
8  end
9  fields ← expand(equalityConditions(w_ast), r_meta);
10 (reduced, unknowns) ← reduceAst(r_ast, fields);
11 if unknowns = All then
12 |  return maybe;
13 end
14 if evaluateAst(reduced) then
15 |  if unknowns = None ∧ ¬limited(r_meta) then
16 |  |  return yes;
17 |  end
18 |  return maybe;
19 end
20 return no;
```

Figure 4.6: Query evaluation for updates and deletes

Since read queries often involves more than one table through various joins, many fields in the AST will not be present in the inserted row or the equality conditions of the write query where clause. An AST reduction algorithm was developed in order to be able to evaluate ASTs where only a subset of the variables is available. The goal of the reduction algorithm is to make the tree always evaluate true if any of the unknown values decides the outcome. The reduction is performed by traversing the AST and reducing expressions with unknown values to either $True$ or $False$,

depending on the context. If all variables in the AST are unknown, the tree will always evaluate to true, thus evaluation can be skipped. If no variables are unknown and the tree evaluates to true, it is certain that the read query will be affected unless it has a limit clause, which could exclude arbitrary rows depending on the order.

(a) Read query

```
SELECT user_actions.name, app_user.name FROM user_actions
INNER JOIN app_user ON user_actions.app_user_id = app_user.id
WHERE app_user.id = 9 AND user_actions.name = 'Changed password';
```

(b) Write query

```
INSERT INTO user_actions (id, name, app_user_id)
VALUES (1, 'Changed password', 10);
```

(c) Resulting AST Evaluation

```
>>> column_equality_map
{'user_actions.app_user_id': 'app_user.id'}

>>> row
{
    'user_actions.id': 1,
    'user_actions.name': 'Changed password',
    'user_actions.app_user_id': 10,
}

>>> evaluate(where_ast, row)
=> app_user.id = 9 AND user_actions.name = 'Changed password'
=> unknown AND 'Changed password' = 'Changed password'
=> True AND True
True

>>> extended_row = extend_row(row, column_equality_map)
{
    'user_actions.id': 1,
    'user_actions.name': 'Changed password',
    'user_actions.app_user_id': 10,
    'app_user.id': 10,
}

>>> evaluate(where_ast, extended_row)
=> app_user.id = 9 AND user_actions.name = 'Changed password'
=> 10 = 9 AND 'Changed password' = 'Changed password'
=> False AND True
False
```

The read query (a) is being evaluated for invalidation against the write query (b). The evaluation is illustrated in (c).

Figure 4.7: Column equality map in use

The join clauses of a read query might contain equality conditions between columns that can be considered preconditions of the where clause. The equality conditions are

used to set up a column equality map, included in the query nodes of the dependency graph, that is used to extend the fields from the write query. This process is shown in Figure 4.7.

4.2 Caching Library

The caching library, called *Cachematic*, was designed with three major goals in mind: Portability, usability and performance. Portability is achieved by using the adapter pattern to facilitate communication with external storage, such as the database and the cache backend. Usability is realized through a single-function programming model, and automatic invalidation based on the algorithm described in Section 4.1. By caching internal data structures to reduce overhead, good performance is achieved for reads in the evaluated workload.

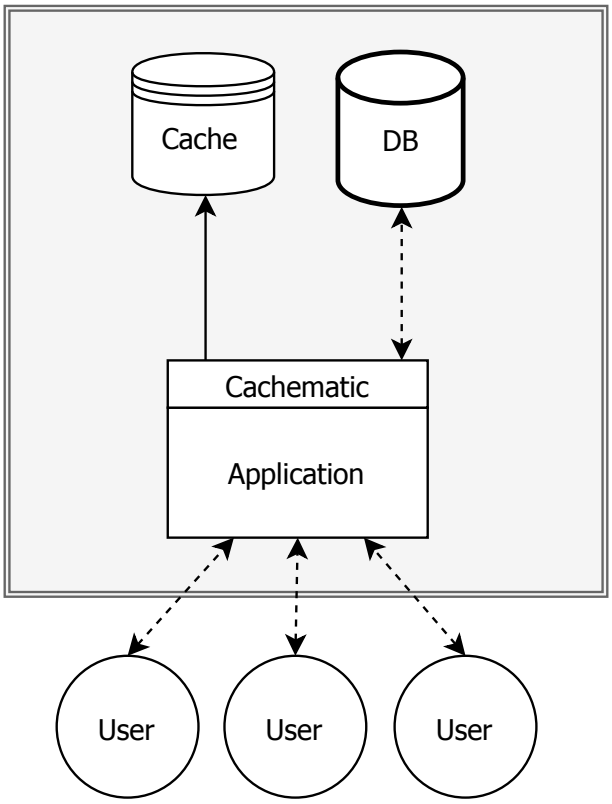


Figure 4.8: Key components in a Cachematic deployment

By deploying *Cachematic* in an application, developers can easily add caching to arbitrary functions containing SQL queries, and be confident that the cached result will always be consistent with the database. The library adds caching to marked functions, and captures SQL queries executed within the application to automatically invalidate the cache according to the algorithm described in 4.1. The library can be described as middleware between the application, the database and the cache, illustrated in Figure 4.8

Due to the requirements of the evaluated application, the caching library was implemented in Python 2.7. Cross compatibility with Python 3.6 was considered but was not prioritized due to lack of time. The library utilizes both functional and object-oriented features of Python and consists of two primary components: the cache manager and the dependency handler. In addition, the library contains a set of modules that handle parsing, abstract syntax trees, serialization, communication with external systems and storage for the dependency graph.

```
@cache_manager.cacheable
def get_user(user_id, include_profile=False):
    user_query = sqlalchemy.text("""
        SELECT * FROM app_user WHERE id = :user_id
    """)

    result = db.execute(user_query, user_id=user_id).first()

    if result is None:
        # No user by that id
        return None

    user = dict(result)

    if include_profile:
        profile_query = sqlalchemy.text("""
            SELECT * FROM app_user_profile WHERE user_id = :user_id
        """)

        result = db.execute(profile_query, user_id=user_id).first()

        user['profile'] = dict(result) if result else None

    # Result to be cached
    return user
```

Figure 4.9: Example of cacheable function fetching a user and optionally it's profile by executing two database queries using SQLAlchemy

The cache manager is the primary API of the library. It is implemented as a class intended to be instantiated in the host application. The primary caching interface consists of a single method, used to decorate other functions and denote them *cacheable*. Combined with the decorator syntax available in Python, the result is a very simple but powerful programming model, illustrated in Figure 4.9. To allow for more exotic use cases, the cache manager also exposes the same API used to implement the cachable decorator. This allows developers to define their own decorators or to access the cache and associated functionality in other ways. The API is shown in Figure 4.10.

The cache manager implements the cache algorithm, shown previously in Figure 4.1, including cache key generation, scope, interacting with the cache and capturing queries to the database. Interaction with the cache backend and database is facilitated through the adapter interfaces `CacheAdapter` and `DBAdapter` respectively.

```
class CacheManager(object):
    ...
    def cacheable(self, fn):
        @functools.wraps(fn)
        def wrapper(*args, **kwargs):
            key = self.generate_key(fn, args, kwargs)

            with self.scope(key):
                hit, result = self.check_cache(key)

                if hit:
                    return result

                return self.run_fn(key, fn, args, kwargs)

        return wrapper

    @contextlib.contextmanager
    def scope(self, key):
        self.start_scope(key)
        yield
        self.end_scope()

    def start_scope(self, key):
        ...

    def end_scope(self):
        ...

    def generate_key(self, fn, args, kwargs):
        ...

    def run_fn(self, key, fn, args, kwargs):
        ...

    def check_cache(self, key):
        ...
```

Figure 4.10: Cache Manager API, including cacheable decorator implementation and context manager for simple scope management

The dependency handler is instantiated inside the cache manager, and implements the dependency and invalidation algorithms. The dependency handler exposes a simple interface to the cache manager, shown in 4.11, consisting of the methods `add_dependency`, `prepare_invalidation` and `finalize_invalidation`. The first method exposes the dependency algorithm and the latter two the invalidation algorithm. The dependency handler is also called from the cache manager to clean up the dependency graph when deleting the keys from the cache. The cleanup process is described in 4.2.2.

```
class DependencyHandler(object):
    ...
    def add_dependency(self, key, read_queries, child_keys):
        # Populate dependency graph
        ...

    def prepare_invalidation(self, write_query, scope=None):
        # Test read queries and prepare internal data structures
        # for the hash test performed in finalize_invalidation
        ...

    def finalize_invalidation(self, scope):
        # Perform hash test and recursively identify cache keys to
        # be invalidated
        ...
        return keys # Keys to be invalidated

    def cleanup(self, keys):
        # Remove associated nodes from the dependency graph
        ...

    def clear(self):
        # Remove everything from the dependency graph
        ...
```

Figure 4.11: Dependency Handler API

4.2.1 Modules

Both the dependency and invalidation algorithms rely on queries executed within the application being communicated to the algorithm. There are many ways to achieve this, including database triggers, parsing SQL query log files and applying listeners within a library. The optimal choice largely depends on the language and libraries used and in what environment the application is deployed. A `DBAdapter` implementation for SQLAlchemy has been developed for the reference implementation used in

the evaluation. This implementation utilizes the SQLAlchemy event system[1] to set up listeners for the core events `before_cursor_execute` and `after_cursor_execute`. The SQLAlchemy adapter also implements the methods `get_query_result_hash` and `get_schema`. The schema is necessary for post processing parsed queries, since the schema holds primary and foreign key information. The schema is also necessary to retrieve column information not available in the queries. The query result hashes are computed in the database by executing an extra query, containing the read query as a subquery. The extra query, shown in Figure 4.12, generates the hash by casting each row to text and combining the strings to a single entry using the aggregate function `array_agg`. The resulting string is passed to the hash function. Any hash algorithm with sufficient distribution can be used to generate the hash. The hash algorithm *md5* was used in the library since it is supported by default in PostgreSQL. A comment is appended to the extra query to be able to ignore it in the listeners. Generating the hash in the database removes network and mapping overhead associated with fetching the entire result set to the application.

```
SELECT md5(array_agg(hash_query::text)::text)
FROM (:read_query) hash_query --extra_query
```

Figure 4.12: Extra query computing the hash of read query results

Another module that is utilized by both parts of the algorithm is the SQL parser, including grammar for a large part of PostgresSQL and a post processing procedure to extract relevant information from the queries and converting where clauses to abstract syntax trees. The parser was implemented using pyparsing[2], a monadic parsing combinator library for creating recursive-descent parsers. The grammar is based on an existing example for parsing SQLLite select statements, included with pyparsing. It has been extended to support a large subset of PostgreSQL, including insert, update and delete statements, to cover the needs of the evaluated application. Supporting the entire SQL language including various dialects would be a significant task, and was out of scope for this thesis.

By default, the result of parsing a string in pyparsing is a tree represented by lists of lists and strings. For each expression in the grammar, a parse action can be set to convert the resulting part of the tree to a more complex type. This was taken advantage of to create a tree where the relevant information can easily be extracted. The post processing engine converts this tree to instances of the class `ParsedQuery`, with sub classes for each query type. Where-clauses are converted to abstract syntax trees using the Python abstract syntax grammar from the `ast` module[3]. This is the grammar used by Python itself, and can be compiled and evaluated using the builtins `compile` and `eval`. Inspecting and transforming trees in this form can be done by extending `ast.NodeVisitor` and `ast.NodeTransformer`. This is how information is extracted from the where clauses, and how the reduction

[1] http://docs.sqlalchemy.org/en/latest/core/events.html
[2] http://pyparsing.wikispaces.com/
[3] https://docs.python.org/2/library/ast.html

algorithm is implemented. Boolean, numeric and comparison expressions in a where clause can be directly mapped to the Python AST. Function calls are translated to predefined equivalent calls in Python, and the `LIKE` expression is translated to a call to a predefined function executing a regular expression. Similarly, the SQL case expression is translated to a function call, evaluating each case sequentially.

The cache manager keeps track of the current scope to enable tracking of queries and nested function calls necessary to build the dependency graph. A context interface with a default implementation was developed to handle this. It keeps a stack of cache keys, representing the scope of function calls. A cache key is pushed on top of the stack when a scope starts and popped when a scope ends, such that the cache key for the innermost function call is always on top of the stack. The context also keeps track of read queries and nested keys for each key on the stack. Queries get recorded to the key on top of the stack whenever they occur. Nested calls are captured at the end of every scope. If the stack is nonempty after popping a key, the popped key is captured as a child of the key on top of the stack. The evaluated application is implemented in Flask[4], a Python web application framework, and deployed using Gunicorn[5], a web server that can be configured to run with multiple processes and threading. If threading is enabled, each request of the application will be executed in a new thread. With multiprocessing, each process will run a distinct instance of the application, while threads share memory within the same instance. To avoid race conditions with multiple threads executing cacheable functions simultaneously, a thread-local implementation of the context for use with Flask was developed. The thread-local context takes advantage of the Flask application context, which is unique for each thread. Implementations supporting other execution models would be straightforward to develop and the cache manager can easily be configured to use implementations other that the default.

A serialization module was developed to help generate cache keys and serialize cached results. Part of the requirements for the evaluated application is caching of values of user defined types such as class instances, in addition to built-in types[6]. Serializing arbitrary objects using standard serialization formats such as json in Python requires custom code per user defined type, which was not reasonable to do within the scope of this thesis. A python-specific serialization method which can handle arbitrary objects is the pickle module[7]. There is also a very performant C-extension equivalent called cPickle. Unfortunately, pickle has a bad reputation for being insecure since a malicious object can perform arbitrary code injection on de-serialization [39]. In order to avoid this, a wrapper for pickle was developed, combining the serialized object with a hmac [40] signature. The signature is generated from the serialized object and a secret key, and is prepended to the output. On de-serialization, the signature is compared to a newly generated one before unpickling the serialized object and raises an exception on mismatch.

[4]http://flask.pocoo.org/
[5]http://gunicorn.org/
[6]https://docs.python.org/2/library/stdtypes.html
[7]https://docs.python.org/2.7/library/pickle.html

Cache keys are generated by concatenating a fully qualified name of the wrapped function with a serialized version of all the arguments. The fully qualified name includes the full module path of the function and its name, including any local scopes, since you can define functions almost anywhere in Python. For the purpose of serializing the arguments, it was discovered that a custom one-way serialization function supporting built-in types was the fastest after evaluating standard serialization formats.

Redis was used both as cache backend and for storage of the dependency graph. An implementation of the `CacheAdapter` interface for Redis was developed for the reference implementation. A wrapper for Redis was also implemented to handle the storage of the dependency graph.

4.2.2 Graph representation

The dependency graph is represented by six hash tables, listed in Table 4.1. In the reference implementation these are stored in Redis with a key prefix for each table. Tables storing sets are implemented using Redis Sets[8] to take advantage of $O(1)$ performance for add and remove operations, thus reducing the overhead of maintaining the dependency graph.

Table	**Key**	**Value**
Column Lookup	Canonical Column	Set of Can. Query Templates
Query Lookup	Canonical Query Template	Set of Query Ids
Instance Lookup	Query Id	Set of Cache Keys
Key Lookup	Cache Key	Set of Parent Cache Keys
Query Nodes	Query Id	Query Params
Query Cache	Canonical Query Template	Parsed Query

Table 4.1: Hash tables representing the dependency graph

The four lookup tables represent most of the dependency graph, including nodes in the first and third layer and all edges in the graph. Query node data is stored in a separate table to keep the sets simple and without serialization. Query nodes contain query parameters for each individual query instance.

The query cache stores the query meta data in the form of a `ParsedQuery`, extracted after parsing and analyzing each query, keyed on the canonical query template string. A query template is a SQL query string before the parameter names have been substituted with actual values. A canonical query template is produced by concatenating the template with the database id. The query cache takes advantage of the fact that the number of unique query templates is relatively low in most applications [32]. Parsing and analyzing the query strings is a time consuming

[8] https://redis.io/topics/data-types

operation, and the query cache is therefore essential to achieve good performance in both the dependency and invalidation algorithm. Both the query cache and the column lookup table stabilized fairly quickly in our evaluation tests.

Using sets to represent most of the dependency graph also simplifies the cleanup process. The process starts by deleting invalidated keys from the key lookup table. Thereafter, the keys are removed from all sets in the instance lookup table, recording query identifiers with empty sets in the process. These queries no longer have any dependencies and can therefore be deleted from the query nodes and removed from the sets in the query lookup table. The column and query lookup tables and the query cache only need cleanup when the schema changes, which happens very rarely. These tables can be cleared manually through the `clear` method in the dependency handler. It is important that the cleanup happens before deleting the keys from the cache, in order to avoid cache misses for the affected keys while modifying the graph. A cache miss would trigger the dependency algorithm, which could modify the same part of the graph simultaneously, cause race conditions, and potentially corrupt the dependency graph. The query part of the tree can still be subject to race conditions, since it is shared between many cache keys. By using distributed locks identified by each query id, the subgraph associated with that query is effectively locked, and modifications can be done safely. The locks were implemented using using Redis [9].

The actual invalidation of cache keys and cleanup of the dependency graph can be done externally through an invalidation callback, optionally configured during instantiation of the cache manager. In the evaluated application, invalidation was configured to be done in a background task. This increases the staleness of the cache slightly, but reduces the overhead imposed on write endpoints.

4.2.3 Invalidation algorithm

As described in Section 4.2.2, read queries are grouped by query template in the dependency graph. This enables bulk testing of read query instances that share query template. For example, the primary key test for inserts, described in Section 4.1.2, is applied on query templates, and can exclude many queries before loading instance specific data. ASTs are constructed only once per pair of write and read query templates, and if all variables are unknown in the AST, all read queries with that template can be directly passed to the hash test.

Utilizing invalidation scope enables further optimizations. For instance, read queries marked for potential invalidation do not have to be evaluated again within a scope. Additionally, instance specific data for read queries are cached in memory within an invalidation scope, to avoid repeated deserialization. At the end of the invalidation scope, each read query needs to be hash tested only once, and cache keys associated with the queries can be loaded in bulk.

[9] https://redis.io/topics/distlock

Due to the availability of events triggered before and after execution of queries in SQLAlchemy, the invalidation algorithm was implemented as a two-phase process. This enables hashing of read query results on demand before and after the execution of a write query instead of hashing when the read query is executed, and reduces the overhead of the dependency algorithm. Within invalidation scope, only one initial hash has to be computed per read query. If an implementation of the `DBAdapter` can only report queries after execution, the cache manager will fall back to single phase invalidation and do the initial hash in the dependency algorithm.

4.3 Testing

A collection of unit tests was developed to verify the correctness of the dependency and invalidation algorithms, including the parser, query tests and dependency graph maintenance. To implement the tests for the dependency and invalidation algorithms, a debug mode was added to the library. In debug mode, the library writes internal data to a log file, including captured queries, dependency data structures and invalidation actions. Unit and regression tests were implemented continuously by inspecting and analyzing the log file, generated by running the system manually and through the evaluation suite.

4.4 Limitations

For *Cachematic* to be able to work properly, cacheable functions have to be dependent only on the arguments passed to the function, a database that is setup to be listened to and any deterministic sub call. If a function that is using any kind of non-determinism outside of the database, like an operating system call, would be annotated cacheable there would be no guarantees that the cache would work. In fact, for instance if a result of a function that used current time would be cached, *Cachematic* would incorrectly return a cached result of that function for any subsequent calls. That is, if the cache was not invalidated for some other reason.

Since the process of invalidating can be time-consuming and happens independently of data access, stale data might remain in the cache for a limited amount of time until the invalidation process is done.

5

Evaluation

In this chapter we present the evaluation of *Cachematic*, performed utilizing Amazon Web Services. The evaluation suite was implemented using a python library, *Locust*[1]. Two baselines are included for comparison. The first baseline is the manual invalidation solution currently in use in the Bison platform. The second is an implementation without caching.

The chapter begins with a description of the library used to develop the test suite and a description of the implementation including a specification of the AWS setup. Following is a section presenting a number of graphs displaying the response times for the different implementations. Next, the cache hit rate for the two caching implementation is presented. The chapter ends with a discussion about the evaluation and an outline of potential threats to the validity of the evaluation.

5.1 Approach

To evaluate the performance of *Cachematic*, the load test library *Locust* was used. *Locust* is a python library that allows developers to load test an http API. For every http request sent from the test locust records the response time, the name of the endpoint, the sequential number of the request and some additional data. A test consists of clients and task sets. The clients serve as users and they perform tasks on a certain interval, specified by the developers. There can be different types of clients which perform different sets of tasks. The ratio of how many of each type of client should exist can be specified.

The task sets are also specified by the developers. The sets contain either other task sets or concrete tasks that specify http requests to the API to be tested. Both task sets and concrete tasks support weighting. When specifying a weight on a task the probability of that task being executed is altered.

When a test is initialized, each client is spawned and assigned a task set. The clients then begin sending requests according to the task set they were assigned. Because of

[1] https://locust.io

the weighting of the tasks and task sets the test scenario becomes non-deterministic but influenced. We used this feature to create a representative workload of the actions performed by users of the Bison platform.

To be able to measure hit rate, which is not measured in *Locust* by default, we made a minor modification to the *Locust* source code. In the code that records relevant data from a response we added extraction of a header(*x-cache-hit*) which indicates if the response data was served from the cache or not.

5.2 Test Setup

The test consists of two sets of users called Client Admin and Regular Client. The Client Admin users are responsible for creating, modifying and sharing data. The characteristic of these users is that they call endpoints that trigger writes to the database. Ultimately, these requests will invalidate cached data, but also enable the analysis in the first place. Regular Client users are responsible for requesting time consuming calculations. The requests sent by Regular Client users trigger a significant number of reads from the database.

The test included 4 Client Admins and 16 Regular Users. All users were assigned a wait time of two seconds between execution of task sets. Weights ranging from one to five was used on different tasks. The test was run for 60 minutes. With a two second wait time and 20 users, an average of 10 tasks are executed per second. Each task executes multiple http requests, resulting in an observed rate of about 30 requests per second.

In the Bison platform, a user belongs to a client, not to be confused with the clients in *Locust*. The users created for the test were equally distributed between two Bison clients, resulting in each Bison client having two Client Admins and eight Regular Users. Data created by the Client Admins is shared with all users belonging to the same Bison client.

The two types of users were assigned one main task set each. Regular Users were assigned the task set UserBehavior and Client Admins were assigned the task set AdminBehavior. The main task sets included multiple sub task sets that are described below.

The test setup was based on insights from Bison employees with specific knowledge about the user behaviors in the platform.

5.2.1 User Behavior

The User Behavior task set includes two different sub task sets to execute: BrowseBenchmark and VehicleAnalysis.

BrowseBenchmark includes two requests. First, the user requests a list of available benchmarks. Second, the user randomly selects one of the available benchmarks and requests the actual benchmark data. The task has weight 1.

VehicleAnalysis includes more steps than the first task set. First, the user requests a list of all available vehicle entities. Next, the user selects a vehicle and requests a list of available reporting-dates. Third, the user selects one of the dates and requests meta information about the vehicle. Lastly, a number of analyses are performed on the vehicle, given the vehicle type, the user's selections, and other input parameters. The task has weight 5.

5.2.2 Admin Behavior

The Admin Behavior task set includes four different sub task sets to execute: UploadCashflow, DeleteEntity, ShareEntity and ChangeAttribute.

UploadCashflow begins with the user selecting a spreadsheet from a pre-defined set of spreadsheets with example data. The spreadsheet is then uploaded through a number of sequential requests. The task has weight 5.

In **DeleteEntity**, the user requests a list of available entities and then selects one that it requests to delete. The task has weight 1.

ShareEntity begins with the user requesting a list of available entities. Next, the user shares all available entities with the other users belonging to the users Bison client. The task has weight 1.

In **ChangeAttribute**, the user requests a list of selects a name and requests to update the entity with the new name. Furthermore, the user requests a list of available attributes for the new entity, selects a random attribute and updates the entity with the new attribute. The task has weight 1.

5.3 Test environment

The evaluated application was deployed to a VPC in Amazon Web Services using AWS CloudFormation to easily replicate the production environment where the application is normally deployed. The environment consists of a RDS database of type `db.m4.large`, a set of four EC2 instances of type `m5.large` and a ElastiCache

cluster with a single Redis node of type `cache.m3.large`. The EC2 instances are provisioned with Ubuntu 16.04, two as web servers and the other two as background workers. PyPy 5.8 was used as python interpreter. The web servers run the application through Gunicorn 19.7.1, and the background workers use Celery 4.1.0. The RDS database is running PostgreSQL 9.6.6.

Before each test, the application was deployed with the target solution and the PostgreSQL database and Redis cluster was reset to a clean state.

To reduce network overhead, the evaluation script was executed on another instance of type `m5.large` in the same VPC.

5.4 Response time

In this section a number of graphs delineating the response times for the different implementations are presented. Every test was divided into samples of 30 seconds and metrics were collected from each of the samples.

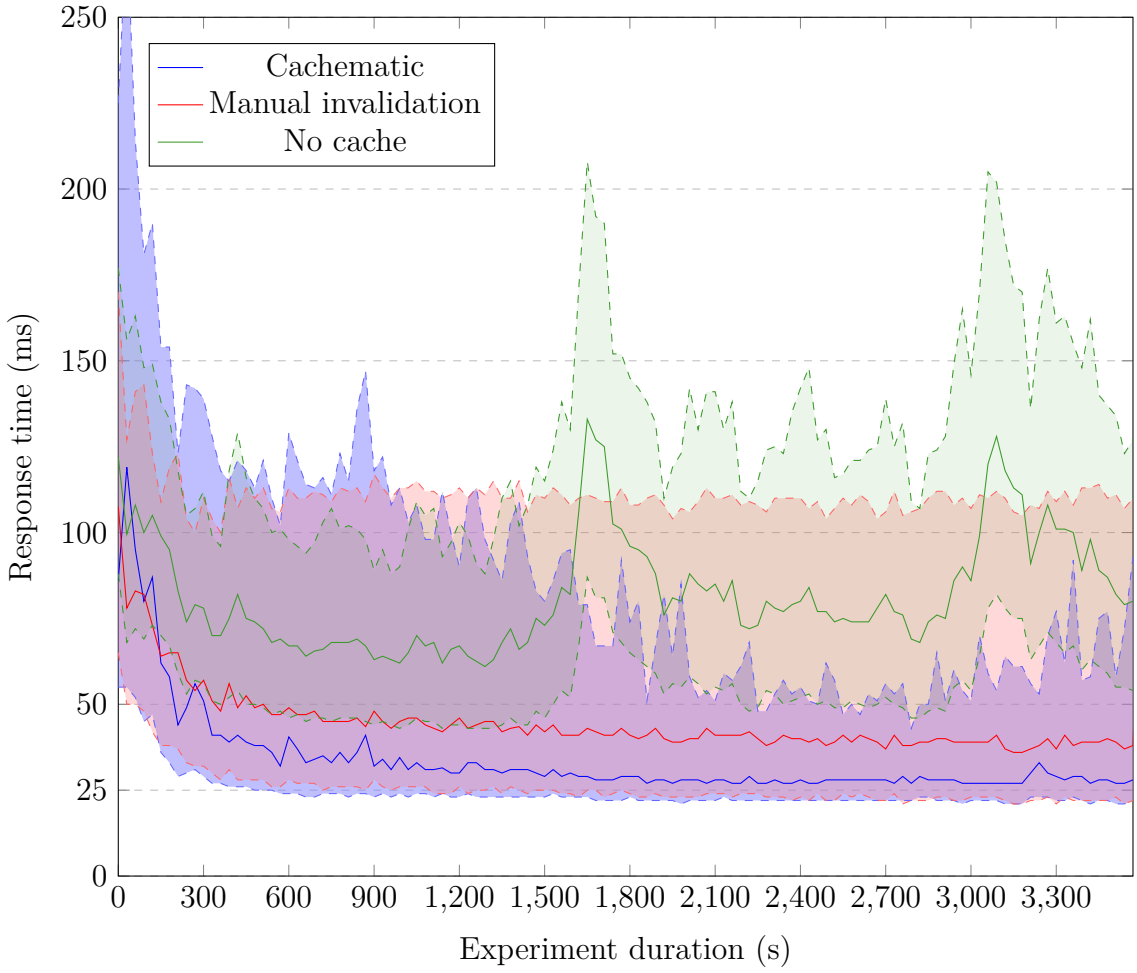


Figure 5.1: Median response time with 25% quartile and 75% quartile

Figure 5.1 plots the median response times of read requests in the evaluation of each implementation. The dashed lines highlights the 25% quartile and the 75% quartile. *Cachematic* performs slightly better than the manual solution. Note that *Cachematic* deviates more than the manual solution in the beginning, highlighting that *Cachematic* requires warmup in addition to just populating the cache. This additional warmup is due to initial parsing of queries and population of the first layer in the dependency graph. The implementation without caching performs significantly worse than the other two implementations. The spikes in response time of the implementation without caching are due to high load on the web server. The solutions with caching are more resilient to load since the web server has to do considerably less work when serving requests from the cache.

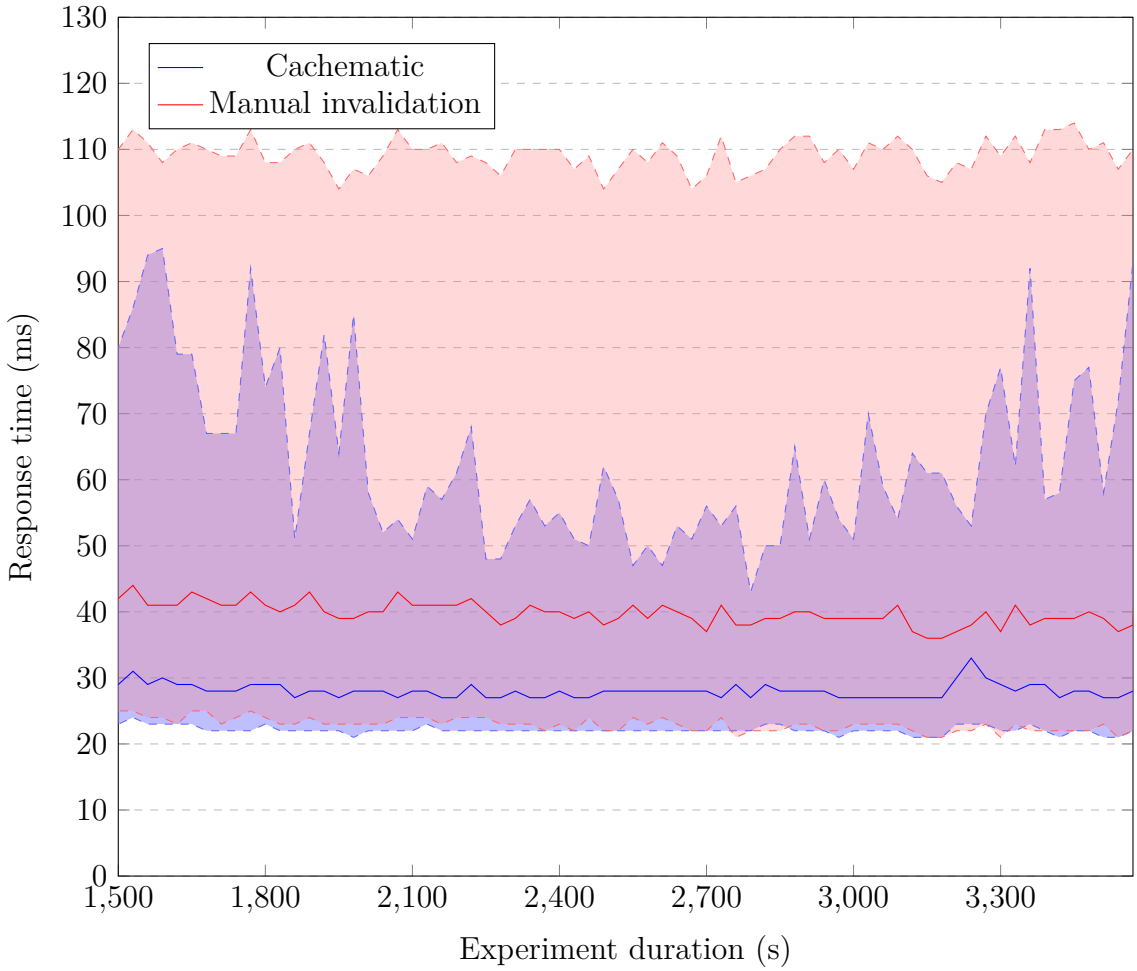


Figure 5.2: Close up median response time with 25% quartile and 75% quartile after warmup

In Figure 5.2 the no cache solution and the warmup time has been stripped out. From this data, a more detailed analysis of the comparison between *Cachematic* and the manual invalidation system can be performed. We observe that, when two caches have stabilized, the median for the manual solution is constantly around 40 ms or just below. The median for *Cachematic* is below 30 ms for almost every sample, establishing that the median response time is decreased to around 75% of the manual solution.

The 75% quartile is significantly lower in *Cachematic* but it is fluctuating more than the 75% quartile for the manual solution. In the manual solution the quartile is around 110 ms while in *Cachematic* it ranges from around 50 ms up to around 90 ms. The cause of the fluctuation is likely because of the cache hit rate presented in Table 5.5. Since the cache hit rate for cachematic is 69%, the 75% quartile probably includes only cache hits for some samples and non-cache hits for some samples. The 75% quartile for the manual solution is supposedly always including non-cache hits.

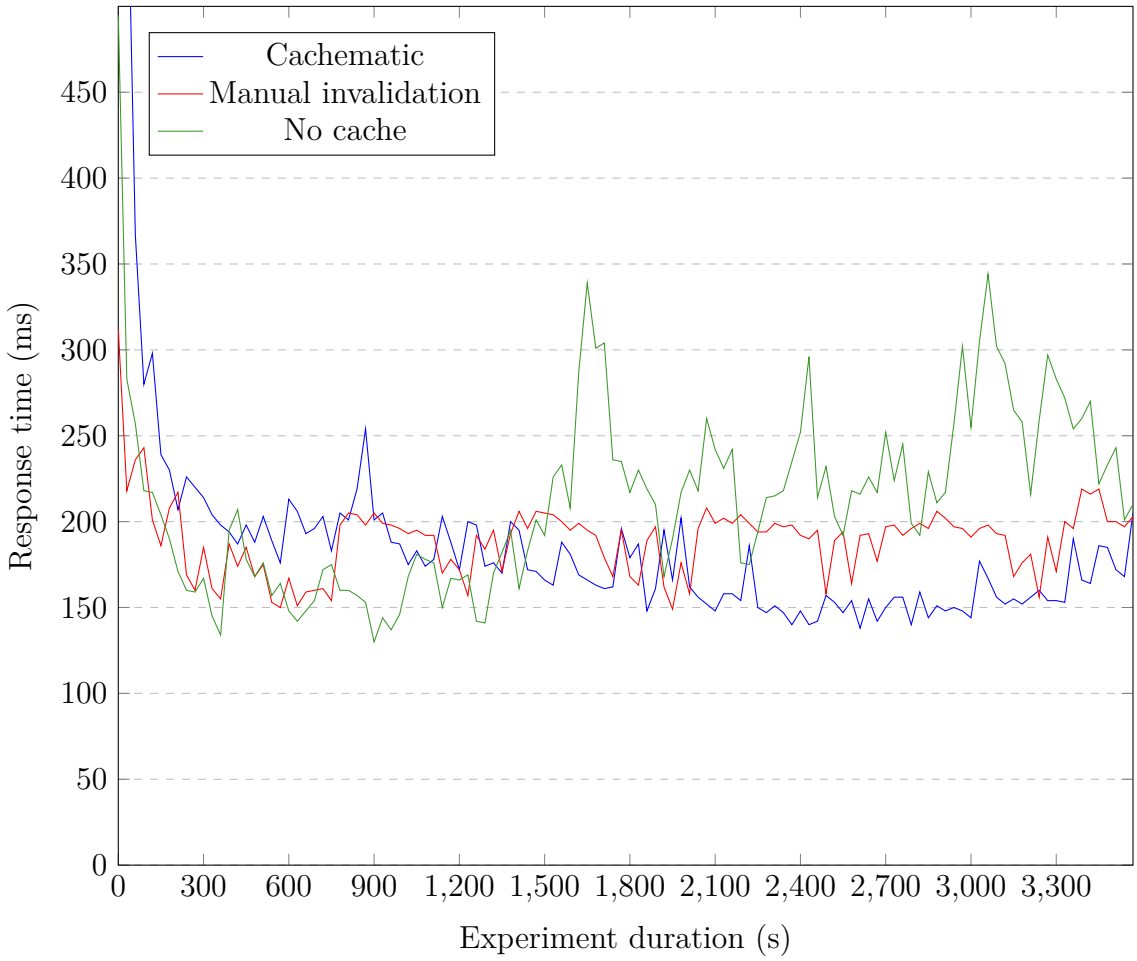


Figure 5.3: 90$^{\text{th}}$ percentile response time for read requests

For the 90$^{\text{th}}$ percentile, plotted in Figure 5.3, *Cachematic* outperforms both the other implementations after the initial warmup period. This can be explained by the higher hit rate achieved with *Cachematic*. The higher hit rate means a significant portion of the 90$^{\text{th}}$ percentile will be cache hits. Requests to different endpoints should take approximately the same amount of time if the result is fetched from the cache. If the request requires actual execution of functions and database calls, the result can vary greatly depending on exactly which endpoint was called.

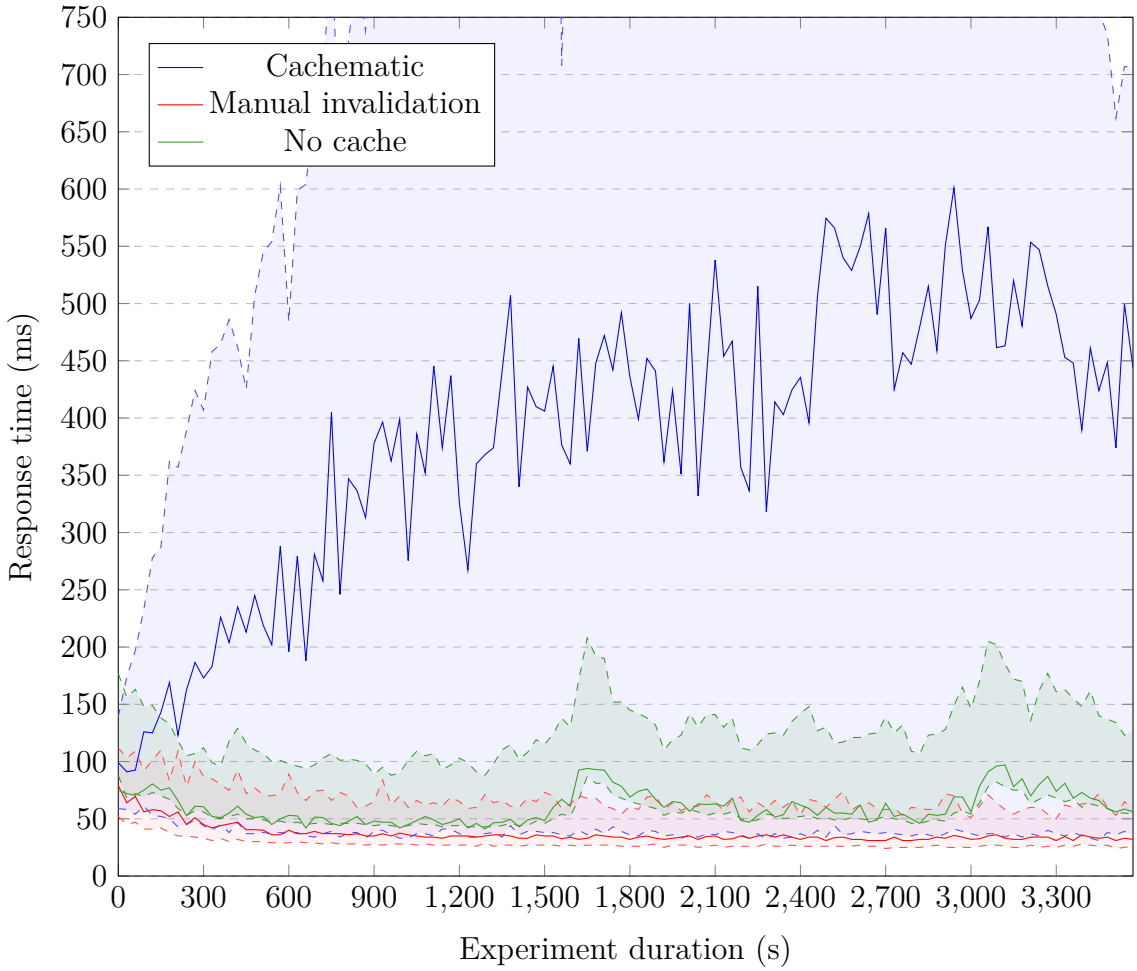


Figure 5.4: Median response time with 25% quartile and 75% quartile

Figure 5.4 plots the median response times of write requests in the evaluation of each implementation. The dashed lines highlights the 25% quartile and the 75% quartile. The plot clearly illustrates the overhead imposed by *Cachematic* on write requests. As the cache gets populated the overhead increases since each write query has to be tested against more read queries. Suggestions for reducing this overhead further are discussed in Section 6.1. Note the spikes in the standard deviation of the implementation without caching. These are due to high load on the web server, since all the calculations have to be done from scratch in each read request. The high load also results in a higher rate of error than in the solutions with caching.

5.5 Cache hit rate

Because of the implementation described in Chapter 4, we know that the cache hit rate for *Cachematic* is close to optimal. What is interesting is how the hit rate compares to the hit rate of the current manual invalidation implementation.

The difference in hit rate gives an indication of how complex the task of manually invalidating cache entries can be for developers. The result can be seen in Figure 5.5.

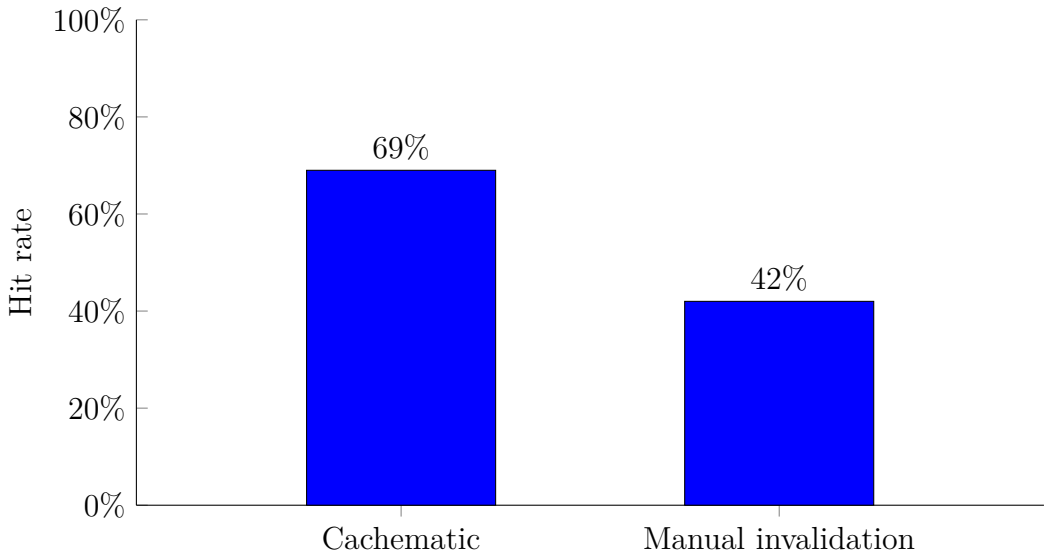


Figure 5.5: Cache hit rate for cachematic and manual solution

From the measured hit rate it can be concluded that *Cachematic* provides a hit rate that is approximately 1.64x higher than the current manual solution. We can also conclude that the manual implementation is invalidating cached entries that do not have to be invalidated.

5.6 Overhead

As can be seen in the graphs in Section 5.4, the overhead for read requests does not seem to have significant impact on the response time. At least, when comparing the manual invalidation implementation to *Cachematic*, the benefits of the higher hit rate provided by *Cachematic* combined with the overhead still performs better.

On the contrary, the overhead for write requests is more considerable. Not only is the response time for write requests very high but it is also fluctuating significantly. It is largely depending on what exact endpoint is being requested and the current status of the cache.

The high overhead in certain write requests can be explained by the execution of hash queries. For some write queries included in the test, none of the exclusion tests will be able to determine invalidation, and a hash query will always be executed. As the cache is populated, and there are more read queries to test, the hash queries will impose significant overhead. The number of read queries in the cache stabilizes over time.

Another factor contributing to the overhead is the high number of complex queries executed in the evaluated application. Most queries join at least one other table, and many have complex where clauses. In the evaluation of other systems, such as *AutoWebCache*, it is highlighted that most of the queries executed by the benchmarks TCP-W and RUBiS are simple, and the where clause often consists of a single equality condition [32]. It is highly likely that the overhead for write requests would be significantly reduced if *Cachematic* was deployed in a system with more simple queries.

5.7 Threats to validity

The evaluation is only representative for a portion of the user behaviors in Bison. The workload used for the evaluation is based on behaviors common for the users of the Bison platform. Therefore, the workload is arguably a reasonable representation of a real world scenario. Furthermore, the workload in Bison might differ from other applications.

Since the test is non deterministic, different test runs might execute different loads on the system. To understand the impact of the non deterministic test, multiple tests were run for the same implementation. We found that for a single implementation, the results did not deviate significantly. To account for the minor differences, the test results from multiple test runs were combined to create the plots in Section 5.4.

Another factor that could affect the test results is the test environment, more specifically variable performance of the cloud instances used in the evaluation. Due to the small deviations in the results between test runs, this factor was not considered further.

The evaluation in this thesis does not consider or optimize for cases when the cache is full. This is largely not an issue in modern application level caching deployments, since arbitrarily large caches are available for cheap. The cache used in the evaluation was sufficiently large to avoid automatic eviction entirely. If necessary, evictions would be determined by Redis' LRU policy .

5.8 Discussion

In this chapter we have showed the evaluation of *Cachematic* in comparison to the current manual invalidation implementation and an implementation without caching. It can be concluded from the evaluation that the two cache implementations greatly improves the response time for read requests sent to the API. Furthermore for read requests, *Cachematic* performs slightly better than the manual invalidation implementation both in terms of median response time and standard

deviation response time. The fact that *Cachematic* provides a higher hit rate explains the improved performance since more requests can be satisfied by the cache. Fetching cached results decreases the number of database queries and application logic that has to be performed, which is often time consuming tasks.

The overhead on write requests imposed by the automatic invalidation algorithm is significant when the cache is populated. Depending on application, the overhead might be considered acceptable, especially if write requests happen rarely and fast read access is highly valued. It also removes a significant workload from the developers, and can help speed up development of new features and reduce the frequency of bugs.

Cachematic has proven to be sufficient enough to replace the current manual invalidation system deployed in the Bison platform. The fact that *Cachematic* significantly reduces the amount of work required by the developers is of great value for Bison. Even though the increased response time in write-intense requests is substantial, the total benefits are considered to weigh up for the increased response times. Some more work will be put into improving implementation details to minimize the disadvantages before the system is finally deployed in the Bison platform.

6

Conclusion

Managing application level caches manually can be a tedious and error-prone task. In particular, when a code base grows in size and complexity, the task of invalidating the cache correctly becomes troublesome. One way of solving the complex task of invalidating cached entries correctly is to use an automatic invalidation system. In this thesis, we introduced *Cachematic*, a general-purpose caching library with an automatic invalidation strategy. *Cachematic* provides a simple programming interface for developers to cache the result of arbitrary functions by simply annotating the function as cacheable, and requires only minor modifications to the application it is deployed in. The algorithms implemented incorporates ideas from related research to efficiently test for invalidation and introduces a hash based test to guarantee the invalidation is performed optimally. A reference implementation, developed in Python, was used for evaluation and comparison with a manual solution deployed in the Bison platform. The evaluation showed that *Cachematic* increased the cache hit rate and reduced the median response time of read requests for a representative workload scenario. The evaluation also showed that the invalidation algorithm is expensive, in particular when the cache is populated with a significant amount of data and complex queries are executed.

6.1 Future Work

There is still work to be done in order to improve the library presented in this thesis. There was not enough time to implement extra queries, described in [32], which fetch additional information to enable more accurate testing of certain read queries. Extra queries can increase the accuracy of read query testing and significantly reduce overhead in the invalidation algorithm. Many of the ancillary modules implemented to support the library are incomplete or unoptimized due to lack of time. The parser needs work to reduce overhead during warmup, and to support the full SQL language including the many dialects. Ideally the grammar should be defined using a common notation such as BNF.

Other potential future work would be to compare *Cachematic* to similar automatic cache management libraries presented in this thesis. For instance, a reference im-

plementation in a programming language supported by RUBiS would enable direct comparison to TxCache and AutoWebCache. Being able to compare *Cachematic* to similar solutions using the exact same evaluation setup would undoubtedly give a better understanding of how *Cachematic* performs in a more general setting, outside of the Bison platform. In [41], three benchmarks for testing performance of a web application with a setting such as described in this thesis are described. It might be of value to further investigate whether it would be reasonable to utilize the benchmarks described in the paper.

The algorithms and the query tests have been informally verified in a collection of simple unit tests. Formal verification remains to be done.

Bibliography

[1] Jayashree Ravi, Zhifeng Yu, and Weisong Shi. A survey on dynamic web content generation and delivery techniques. *Journal of Network and Computer Applications*, 32(5):943 – 960, 2009. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2009.03.005. URL `http://www.sciencedirect.com/science/article/pii/S1084804509000526`. Next Generation Content Networks.

[2] J. Farrell and G. S. Nezlek. Rich internet applications the next stage of application development. In *2007 29th International Conference on Information Technology Interfaces*, pages 413–418, June 2007. doi: 10.1109/ITI.2007.4283806.

[3] Harald Weinreich, Hartmut Obendorf, Eelco Herder, and Matthias Mayer. Not quite the average: An empirical study of web use. *ACM Trans. Web*, 2(1): 5:1–5:31, March 2008. ISSN 1559-1131. doi: 10.1145/1326561.1326566. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1326561.1326566`.

[4] C. V. Manikandan, P. Manimozhi, B. Suganyadevi, K. Radhika, and M. Asha. Efficient load reduction and congession control in internet through multilevel border gateway proxy caching. In *2010 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–4, Dec 2010. doi: 10.1109/ICCIC.2010.5705859.

[5] Pablo Rodriguez, Christian Spanner, and Ernst W. Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Trans. Netw.*, 9(4):404–418, August 2001. ISSN 1063-6692. doi: 10.1109/90.944339. URL `http://dx.doi.org.proxy.lib.chalmers.se/10.1109/90.944339`.

[6] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999. ISSN 0146-4833. doi: 10.1145/505696.505701. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/505696.505701`.

[7] T. Chiueh and P. Pradhan. High-performance ip routing table lookup using cpu caching. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1421–1428 vol.3, Mar 1999. doi: 10.1109/INFCOM.1999.752162.

[8] Nezer Zaidenberg, Limor Gavish, and Yuval Meir. New caching algorithms

performance evaluation. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, Spects '15, pages 1–7, San Diego, CA, USA, 2015. Society for Computer Simulation International. ISBN 978-1-5108-1060-0. URL http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=2874988.2875009.

[9] Probir Ghosh and Andrew Rau-Chaplin. Performance of dynamic web page generation for database-driven web sites. In *Proceedings of the International Conference on Next Generation Web Services Practices*, NWESP '06, pages 56–63, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2664-0. doi: 10.1109/NWESP.2006.24. URL https://doi.org/10.1109/NWESP.2006.24.

[10] L. D. Paulson. Building rich web applications with ajax. *Computer*, 38(10): 14–17, Oct 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.330.

[11] Fabian Schneider, Sachin Agarwal, Tansu Alpcan, and Anja Feldmann. The new web: Characterizing ajax traffic. In Mark Claypool and Steve Uhlig, editors, *Passive and Active Network Measurement*, pages 31–40, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-79232-1.

[12] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten Van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1), 2007.

[13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[14] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004 (124):5, 2004.

[15] Robert Johnson. More details on today's outage, Sept 2010. URL https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[16] X. Qin and X. Zhou. Db facade: A web cache with improved data freshness. In *2009 Second International Symposium on Electronic Commerce and Security*, volume 2, pages 483–487, May 2009. doi: 10.1109/ISECS.2009.236.

[17] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003. ISSN 0360-0300. doi: 10.1145/954339.954341. URL http://doi.acm.org.proxy.lib.chalmers.se/10.1145/954339.954341.

[18] Sadiye Alici, Ismail Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. Adaptive time-to-live strategies for query result caching in web search engines. In Ricardo Baeza-Yates, Arjen P. de Vries, Hugo Zaragoza, B. Barla Cambazoglu, Vanessa Murdock, Ronny Lempel, and Fabrizio Silvestri, editors, *Advances in Information Retrieval*, pages 401–412, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28997-2.

[19] 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. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088185. URL `http://doi.acm.org/10.1145/1088149.1088185`.

[20] Christof Bornhövd, Mehmet Altinel, C Mohan, Hamid Pirahesh, and Berthold Reinwald. Adaptive database caching with dbcache. *IEEE Data Eng. Bull.*, 27 (2):11–18, 2004.

[21] J. Mertz and I. Nunes. A qualitative study of application-level caching. *IEEE Transactions on Software Engineering*, 43(9):798–816, Sept 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2633992.

[22] Wei Wang, Zhaohui Liu, Yong Jiang, Xinchen Yuan, and Jun Wei. Easycache: A transparent in-memory data caching approach for internetware. In *Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware*, INTERNETWARE 2014, pages 35–44, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3303-0. doi: 10.1145/2677832.2677837. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2677832.2677837`.

[23] Tse-Hsun Chen, Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 666–677, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950303. URL `http://doi.acm.org/10.1145/2950290.2950303`.

[24] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[25] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4): 12–27, 2011.

[26] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[28] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, 2017.

USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

[29] 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. doi: 10.1109/INFCOM.1999.749295.

[30] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 24–44. Springer, 2000.

[31] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 279–292, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924963.

[32] Sara Bouchenak, Alan Cox, Steven Dropsho, Sumit Mittal, and Willy Zwaenepoel. Caching dynamic web content: Designing and analysing an aspect-oriented solution. In *Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'06, pages 1–21, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-49023-X, 978-3-540-49023-4. doi: 10.1007/11925071_1. URL http://dx.doi.org/10.1007/11925071_1.

[33] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for orms. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'11, pages 329–349, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25820-6. doi: 10.1007/978-3-642-25821-3_17. URL http://dx.doi.org/10.1007/978-3-642-25821-3_17.

[34] Ziv Scully and Adam Chlipala. A program optimization for automatic database result caching. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 271–284, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009891. URL http://doi.acm.org/10.1145/3009837.3009891.

[35] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 532–543, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4. doi: 10.1145/375663.375736. URL http://doi.acm.org.proxy.lib.chalmers.se/10.1145/375663.375736.

[36] Shahram Ghandeharizadeh, Jason Yap, and Sumita Barahmand. Cosar-cqn: an application transparent approach to cache consistency. In *Twenty First International Conference On Software Engineering and Data Engineering*, 2012.

[37] Cristiana Amza, Anupam Ch, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *In 5th IEEE Workshop on Workload Characterization*, pages 3–13, 2002.

[38] Adam Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784741. URL `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2784731.2784741`.

[39] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 17:1–17:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2316-1. doi: 10.1145/2500727.2500747. URL `http://doi.acm.org/10.1145/2500727.2500747`.

[40] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[41] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*, pages 3–13, Nov 2002. doi: 10.1109/WWC.2002.1226489.