Cachematic – Automatic Invalidation in Application-Level Caching Systems

Viktor Holmqvist, Jonathan Nilsfors
Bison
Boston, MA, USA
vholmqvist@bison.co,jnilsfors@bison.co

Philipp Leitner
Chalmers | University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

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. 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 as cacheable. The result of a cacheable function will transparently be cached without the developer having to worry about cache management. We present algorithms that automatically handle cache management, handling the cache dependency tree, and cache invalidation. Our experiments showed that the deployment of Cachematic decreased response time for read requests, compared to a manual cache management strategy for a representative case study conducted in collaboration with Bison, an US-based business intelligence company. We also found that, compared to the manual strategy, the cache hit rate was increased with a factor of around 1.64x. However, we observe a significant increase in response time for write requests. We conclude that automatic cache management as implemented in Cachematic is attractive for read-domminant use cases, but the substantial write overhead in our current proof-of-concept implementation represents a challenge.

ACM Reference Format:
Viktor Holmqvist, Jonathan Nilsfors and Philipp Leitner. 2019. Cachematic – Automatic Invalidation in Application-Level Caching Systems. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3309666

1 INTRODUCTION

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 [16]. As web applications have become more and more complex over time, the need for processing data in an efficient way has become of great importance. A general approach for improving performance in computer systems is caching. Caching can be employed on multiple levels, for example in a network [22, 25], in a computer or on a single CPU [8]. 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 [26].

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 [24]. A challenge 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[1].

A common method for cache management is (explicit) cache invalidation. Cache invalidation works by directly replacing or removing stale data in a caching system [19, 21] (as opposed to, for instance, time based invalidation [1], which simply purges cache entries if they have not been used for a defined time). However, explicit cache invalidation requires that the system knows which database results were used to derive the cache entry. Whenever those resources are updated, the entry should be purged or updated. Today, cache invalidation is implemented in the application layer. Developers needs to explicitly purge invalid results, which is cumbersome and error-prone.

In this paper we present the design, underlying algorithms, and a proof-of-concept implementation of Cachematic, a general-purpose middle-tier caching library with an automatic invalidation strategy. Cachematic has been developed in collaboration with Bison[2], with the goal of improving cache management in their web-based business intelligence solution. Cachematic automatically adds SQL database query results to the cache, but, more importantly, also tracks when these results become stale and updates stale results automatically. Due to technical requirements from Bison, the Cachematic proof-of-concept has been implemented in Python for the SQLAlchemy framework. However, the same concept can also be applied to other systems. The system is enabled on method level through code annotations, and does not require any other developer input.

We evaluate the system through experiments with representative workloads reflecting both, end user and administrator usage. We find that Cachematic improves cache hit rates by 69% as compared

[1]https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919
[2]https://bison.co/

to the existing manual solution, leading to a median read request response time that improved by a factor of 1.3. However, our current implementation imposes a severe write request overhead. We discuss reasons for this and potential remediation strategies, which we plan to work on as part of our future work. However, given that read request performance is much more important to Bison, the company still intends to go forward with Cachematic, despite present limitations.

The work underlying this paper has been conducted by the first two authors as part of their thesis project at Chalmers University of Technology. More information and technical details can be found in the master project report [12].

2 BACKGROUND AND MOTIVATION

Application level caching is the concept of caching data internal to an application, oftentimes database results. The cache is typically implemented as a key-value store or in-memory database, using technologies such as Redis or Memcached [10, 18]. The basic principle is that a key-value store provides $O(1)$ performance on retrieving cached results, which is substantially more efficient than a typical database query.

Cache management is the process of keeping the cache up to date when the underlying data changes, and avoiding stale or inconsistent data being served from the cache. A common method for cache management is cache invalidation. Cache invalidation works by directly replacing or removing stale data in a caching system [19, 21]. 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. A central aspect of cache invalidation, especially in the context of invalidation of cached database queries, is the granularity of the invalidation process, which we illustrate by the following example. A cache entry consists of a set of tuples R from database relation T. With coarse-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 strongly 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 [2, 3].

2.1 Manual Caching at 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 current 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 paper. 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.

```python
@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 1: Example of a cacheable function utilizing the manual solution at Bison.

Figure 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.

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.

```python
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: Example function explicitly invalidating the cache.

Figure 2 illustrates the manual invalidation process, explicitly invalidating hard coded strings as determined by the developers on implementation. This solution has historically caused numerous bugs in the Bison web application. Further, developers often over-invalidate results to be on the safe side. For instance, they will often invalidate an entire table when simply invalidating a specific result would have been sufficient. This leads to a loss of performance. The goal of our work is to automate the cache management, avoiding cache invalidation bugs and improving the performance for read requests (queries).

3 CACHEMATIC

We now present our automatic application-level caching solution, Cachematic, in detail. The design of Cachematic is based on a combination of ideas from existing research and lessons learned from the manual caching system that is currently in use at Bison. The caching library is intended to eventually replace manual invalidation.

3.1 Overview

A high-level architectural overview of Cachematic is provided in Figure 3. Cachematic is enabled and configured through simple annotations in the application code. The actual library has two main elements, a read request processor (handling SQL `SELECT` statements) and a write request processor (handling data-modifying SQL statements). The library interacts with two data storage systems, a SQL database which is the persistent data storage of the system, and a key-value store, which is used as a cache. The goal of any caching system is to serve as many query results as possible from the fast key-value store, without leading to data inconsistencies with the SQL database. Both, read and write query processors, make use of four additional services, a SQL statement parser, a hash function, which is used to map statements or SQL templates to keys in the key-value store, a service representing the underlying data dependency tree, and a function for serializing data that is to be cached to the key-value store.

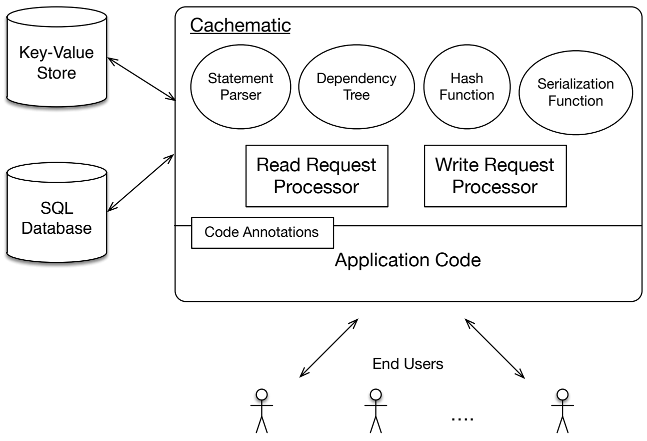

Figure 3: Architectural overview of Cachematic.

We now discuss the Cachematic programming model, the basic algorithms behind the read and write request processors, as well as concrete implementation issues that we faced when building the Cachematic proof-of-concept.

3.2 Programming Model

The current proof-of-concept for Cachematic is implemented in Python. It is enabled by application developers on Python function level, by annotating a function with `@cache_manager.cacheable`. The library scans annotated functions for SQL statements, and handles all statements that it finds. It should be noted that even in a system using Cachematic, developers are free to manually manage caching for a subset of queries, by simply not using this annotation for specific functions. A simple usage example from Bison is given in Figure 4.

```python
@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()
    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
    return user
```

Figure 4: Example of cacheable function fetching a user and optionally it's profile by executing two database queries using the SQLAlchemy Python library.

Note that the example contains multiple SQL queries, which is quite common. Cachematic handles all SQL statements in a function, independently as to whether they are executed all the time (as in the case of the first query) or only based on a specific code condition (second query). Further note that real-life queries are often defined as templates, to be instantiated at runtime (i.e., `:user_id` in the example).

Further note that the example contains only queries that do not mutate the database state. Evidently, this is not always the case. Cachematic needs to react differently to queries (for which it primarily needs to determine whether the result can be served from cache or not) than to, e.g., `UPDATE` statements, for which it may need to invalidate existing cache entries. This is discussed in the following.

3.3 Caching and Invalidation Algorithms

Algorithmically, Cachematic is based on three procedures: a caching algorithm, a dependency management algorithm, and an invalidation algorithm. To optimize performance, invalidation is done hierarchically in multiple steps.

3.3.1 Caching Algorithm. The basic caching algorithm handles determining whether a cached value exists for a given query, as well as entering the return value from the SQL database in case it does not. This process includes generating the cache key from the function name and the arguments of the function call using the hash service, 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 Algorithm 1.

Algorithm 1: The caching 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 $dependencyAlgorithm(k, queries, nestedKeys)$;
14 return result;
```

In essence, the caching algorithm generates a key (to be used in the key-value store) from the query signature. If the cache (i.e., the key-value store) contains a result for this key it is returned (lines 6 – 8). Otherwise, the result of the query is stored back to the cache (lines 9 – 12), and the underlying dependency tree is updated (line 13). For this, we rely on the dependency management algorithm, which is described below. Another core principle of our approach is the usage of invalidation scopes, which avoid queries being evaluated multiple times against multiple write statements within the same execution context. A deeper discussion of invalidation scopes is provided in Section 3.3.5.

3.3.2 Dependency Management Algorithm. The dependency management algorithm is primarily used to handle queries, i.e., statements that do not modify the state of the SQL database. Statements that update results are handled by the invalidation algorithm, which is discussed in Section 3.3.3. The dependency algorithm utilizes meta data extracted from queries executed within the scope of a cacheable function to construct a dependency graph.

The dependency data structure is modeled as a directed graph consisting of nodes in three layers, hereafter called the *dependency graph*, illustrated in Figure 5. The nodes in the first layer represent table columns, which are the entry points of the graph. The nodes in the second layer represent queries. An edge is established from the first to the second layer if a column occurs in a query. The nodes in the third layer are cache keys generated by the cache algorithm, using the hash service. A node in the third layer will have an incoming edge from the second layer if a query was executed during the scope of the cacheable function. If a cacheable 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.

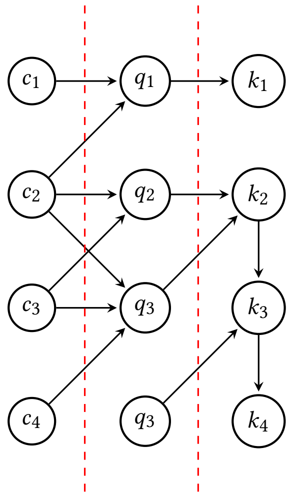

Figure 5: Schematic overview of a dependency graph. c_1, c_2, c_3 and c_4 are columns, q_1, q_2, q_3 and q_4 are queries and k_1, k_2, k_3 and k_4 are function keys.

When queries are passed to the dependency management algorithm, the query strings are parsed using the statement parser 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 easier 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. 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 management algorithm is illustrated in Algorithm 2. The algorithm begins by creating a key node, which is added to the set of nodes to be added to the dependency graph. Captured queries are iterated, and for every query the algorithm extracts meta data, computes a query result hash, and generates

Algorithm 2: Dependency management algorithm.

```
Input  : Cache key $k$, captured queries $Q$ and cache keys
         of nested function calls $N$
Output : $Nodes$ and $edges$ to be added to dependency
         graph
1  $k_{node} \leftarrow KeyNode(k)$;
2  $nodes \leftarrow \{k_{node}\}$;
3  $edges \leftarrow \emptyset$;
4  for $q \in Q$ do
5      $q_{meta} \leftarrow parseQuery(q)$;
6      $q_{hash} \leftarrow computeResultHash(q)$;
7      $q_{node} \leftarrow QueryNode(q, q_{meta}, q_{hash})$;
8      $db \leftarrow dbId(q_{meta})$;
9      $table \leftarrow tableName(q_{meta})$;
10     add $q_{node}$ to $nodes$;
11     for $c \in columns(q_{meta})$ do
12         $c_{id} \leftarrow db + table + c$;
13         $c_{node} \leftarrow ColNode(c_{id})$;
14         add $c_{node}$ to $nodes$;
15         add $(c_{node}, q_{node})$ to $edges$;
16     end
17     add $(q_{node}, k_{node})$ to $edges$;
18 end
19 for $n \in N$ do
20     $n_{node} \leftarrow KeyNode(n)$;
21     add $n_{node}$ to $nodes$;
22     add $(n_{node}, k_{node})$ to $edges$;
23 end
24 return $nodes, edges$;
```

a query node, which is added to the set (lines 5 – 10). For each column in the query, a column node is generated and is added to the set of nodes (lines 11 – 16). Furthermore, an edge is added from the column node to the query node (line 15). An edge is also added from the query node to the key node (line 17). Finally, a node is added for every nested function call and consequently, edges from these nodes to the key node (lines 19 – 23).

3.3.3 *Invalidation Algorithm.*

The invalidation algorithm combines information from the dependency graph with meta data extracted from write statements to identify query candidates for invalidation and eventually carry out the invalidation itself.

In contrast to queries, which are only processed in the scope of cacheable functions, write statements executed in any scope must be processed to ensure consistency. When a write statements is received, the SQL statement is parsed and processed in a similar fashion to queries, with some technical variations depending on the type of statement. In the case of `INSERT INTO` and `DELETE FROM`, all columns of the affected table are captured as the query will result in an entire row being added or removed. For `UPDATE`, 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 again parsed into ASTs.

After processing the write query, the extracted columns are looked up in the first layer of the dependency graph. For matching nodes, edges are traversed to the next layer to identify corresponding queries that will be considered for invalidation. Subsequently, the 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 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 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. These exclusion tests are described in detail in Section 3.3.4.

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 Algorithm 3. To summarize, the invalidation algorithm parses the write query (line 4) and looks up the canonical column identifiers in the dependency graph to find the corresponding nodes (lines 7 – 10). Each node, representing a read query, is evaluated against the write query to find if invalidation of the corresponding keys is necessary (lines 12 – 24). Keys marked for invalidation are extended with dependent keys by following the edges in the dependency graph (lines 25 – 29). The keys are returned to the caller (line 30).

3.3.4 *Hierarchical Invalidation.*

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

The exclusion test for inserts is depicted in Algorithm 4. The algorithm returns "yes", "no" or "maybe" to indicate whether invalidation is required and begins by extracting the AST of the "where" clause of the read query (line 1). If there is no "where" clause, "yes" is returned unless the read query is limited, in which case it needs further evaluation and "maybe" is returned (lines 2 – 7). If the table in the write query is filtered by primary key in the read query, "no" is returned (lines 8 – 10). Each inserted row is now evaluated against the "where" AST of the read query, by reducing the AST and substituting variables for the values in the insert (lines 11 – 13). If none of the variables in the AST are known, "maybe" is returned (lines 14 – 16). If a row satisfies the AST and all variables are known, "yes" is returned (lines 17 – 20). If a row satisfies the AST and some variables are unknown, "maybe" is returned (line 21). If none of these tests passed, "no" is returned (line 24).

Algorithm 3: The invalidation algorithm.

```
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 $K$
```

A 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 [2]. 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, 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 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 query can be excluded from further testing.

Evaluation of the 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

Algorithm 4: Query evaluation for inserts.

```
Input  : Write query $w$, read query $r$ and meta data $w_{meta}$
         and $r_{meta}$
Output : Invalidation status $yes$, $no$ or $maybe$
1  $r_{ast} \leftarrow whereAst(r_{meta})$;
2  if $r_{ast} = Nothing$ then
3      if $limited(r_{meta})$ then
4          return $maybe$;
5      end
6      return $yes$;
7  end
8  if $tableName(w_{meta}) \in pkFiltered(r_{meta})$ then
9      return $no$;
10 end
11 for $row \in rows(w)$ do
12     $fields \leftarrow expand(row, r_{meta})$;
13     $(reduced, unknowns) \leftarrow reduceAst(r_{ast}, fields)$;
14     if $unknowns = All$ then
15         return $maybe$;
16     end
17     if $evaluateAst(reduced)$ then
18         if $unknowns = None \wedge \neg limited(r_{meta})$ then
19             return $yes$;
20         end
21         return $maybe$;
22     end
23 end
24 return $no$;
```

fields in the query AST. If there is no “where” clause in the write statement, it is certain the query is affected unless it is limited, and it can be passed directly to invalidation.

This algorithm is specified more detailedly in Algorithm 5. The evaluation algorithm for updates and deletes returns "yes", "no" or "maybe" similarly to the evaluation algorithm for inserts. The algorithm first extracts the AST of the "where" clauses for both the query and the write query (line 1 – 2). If either of the ASTs are empty, no comparison between the two can be made. In this case the algorithm will check if the query includes a limit clause (line 4) and in that case return "maybe". If no limit clause was included in the query, the algorithm will return "yes". If both ASTs are not empty, the two are compared to find common columns and the query AST is then reduced based on the common columns (lines 9 – 10). If neither of the columns used for filtering in the query is present in the write query the algorithm returns "maybe". Otherwise, the reduced AST is evaluated. If the evaluation returns true, there are no unknown variables in the read AST and the query is not limited, the algorithm returns "yes" (lines 14 – 16). If the evaluation of the AST returned true, but there are unknown variables or the query is limited, the algorithm returns "maybe". Lastly, if the algorithm did not determine invalidation to be necessary or potentially necessary, "no" is returned.

Algorithm 5: Query evaluation for updates and deletes.

```
Input  : Write query $w$, read query $r$ and meta data $w_{meta}$ and $r_{meta}$
Output : Invalidation status $yes$, $no$ or $maybe$
1  $r_{ast} \leftarrow whereAst(r_{meta})$;
2  $w_{ast} \leftarrow whereAst(w_{meta})$;
3  if $r_{ast} = Nothing \vee w_{ast} = Nothing$ then
4      if $limited(r_{meta})$ then
5          return $maybe$;
6      end
7      return $yes$;
8  end
9  $fields \leftarrow expand(equalityConditions(w_{ast}), r_{meta})$;
10 $(reduced, unknowns) \leftarrow reduceAst(r_{ast}, fields)$;
11 if $unknowns = All$ then
12     return $maybe$;
13 end
14 if $evaluateAst(reduced)$ then
15     if $unknowns = None \wedge \neg limited(r_{meta})$ then
16         return $yes$;
17     end
18     return $maybe$;
19 end
20 return $no$;
```

3.3.5 Invalidation Scopes. It is necessary to avoid queries being evaluated multiple times against multiple write statements within the same execution context. This is handled through invalidation scopes in Cachematic. Invalidation scopes work 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 statements as they happen and deferring processing until the scope ends, evaluation of queries can be remembered across write statements. For example, if a query is marked for invalidation, it is not necessary to evaluate it again, and it can be skipped for the remaining writes within the same scope. Invalidation scopes are also the basis of many implementation specific optimizations, which have already been outlined in the invalidation algorithm in Figure 3.

3.4 Cachematic Implementation

We implemented the ideas described so far in a proof-of-concept Python library called Cachematic. 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 key-value store. Usability is realized through a single-function programming model, and automatic invalidation based on the algorithms described earlier. By caching internal data structures to reduce overhead, we attempted to improve performance for queries, which was a primary goal of this work.

We implemented Cachematic in Python 2.7, and for the SQLAlchemy SQL library. We use the SQLAlchemy event system[3] 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, as the schema holds primary and foreign key information. The schema is also necessary to retrieve column information not available in the queries.

We used PostgreSQL[4] as a SQL database and Redis[5] as key-value cache implementation, although our system is not dependend on the specific databases being used. As a hashing function, we have made use of the PostgreSQL implementation of MD5. Generating the hash in the database removes network and mapping overhead associated with fetching the entire result set to the application.

We are now in the process of releasing Cachematic as open source software to the community, pending agreements with Bison.

3.4.1 SQL Parsing. The SQL statement parser, which is required for multiple of the algorithms that Cachematic is based on, was implemented using pyparsing[6], a monadic parsing combinator library for creating recursive-descent parsers. We developed a custom grammar based on an existing example for parsing SQLLite `SELECT` statements. Our custom parser supports a large subset of PostgreSQL's dialect of SQL, including `SELECT`, `INSERT INTO`, `UPDATE` and `DELETE FROM` statements, driven by the concrete needs of the Bison web application. Supporting the entire SQL language including various dialects would be a significant task, and was out of scope for this proof-of-concept.

"Where" clauses are converted to ASTs trees using the Python abstract syntax grammar from the `ast` module[7]. This is the grammar used by Python itself, and can be compiled and evaluated using the builtins `compile` and `eval`.

3.4.2 Scope Management. Cachematic 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 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 not empty after popping a key, the popped key is captured as a child of the key on top of the stack.

3.4.3 Serialization. A serialization module was developed to help generate cache keys and serialize cached results. Part of the requirements for usage in Bison is that caching of values of user defined types such as class instances, in addition to built-in types, is possible. Serializing arbitrary objects using standard serialization formats such as JSON in Python requires custom code per user defined type,

[3]http://docs.sqlalchemy.org/en/latest/core/events.html
[4]https://www.postgresql.org
[5]https://redis.io
[6]http://pyparsing.wikispaces.com/
[7]https://docs.python.org/2/library/ast.html

which was not reasonable to do within the scope of this project. Hence, we used the Python-specific serialization module pickle[8], which can handle arbitrary objects. Unfortunately, pickle has a bad reputation for being insecure, as a malicious entity can perform arbitrary code injection on de-serialization [7]. In order to avoid this, a wrapper for pickle was developed, combining the serialized object with a hmac [13] 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 EXPERIMENTAL EVALUATION

In order to evaluate the usefulness and performance of Cachematic, we have conducted measurement experiments using representative workloads from Bison and a test installation of the Bison production web application in Amazon EC2.

4.1 Experimental Setup

The main goal of our experiments was to evaluate the usefulness of Cachematic in an industrial context. Our expectation going into the experiment was that Cachematic would be able to cache more requests than the manual solution (as developers sometimes would forget to cache results, or implement manual caching suboptimally), and that this would in turn significantly improve the response time of read requests. However, due to the overhead imposed by runtime evaluation of cache results (and the simple fact that caching does not help with write requests *per se*), we expect write requests to become slower. Hence, we have formulated the following three research questions.

RQ1*: Can Cachematic improve the cache hit rate w.r.t. manual caching?*

RQ2*: Does using Cachematic improve the response time in read requests w.r.t. manual caching?*

RQ3*: What overhead does Cachematic impose on write requests w.r.t. manual caching?*

Experiment Workloads. In close collaboration with application experts from Bison, we have defined the workload patterns described in Table 1 to use for our experiments. These were considered by Bison to be representative of production usage of the application. For the purpose of this study it is not essential to understand the semantics behind the vocabulary given in the table – descriptions of tasks are primarily given as an illustration.

We created two types of workload patterns. *User tasks* simulate typical behaviors of a Bison end user. All of these tasks are read-only. Conversely, *admin tasks* represent typical behaviors of Bison administrators. These are a combinations of read and write requests. All workload patterns consist of 2 or more consecutive HTTP requests (denoted by *Req.* in the table). Further, patterns are assigned a weight (column *Wgt.*). Patterns with weight 5 are, on average, executed 5 times more often than patterns with weight 1.

Test Environment. We used AWS CloudFormation to provision a testbed in AWS EC2. The testbed consisted of a RDS database of type `db.m4.large`, four EC2 instances of type `m5.large`, and a ElastiCache cluster with a single Redis node of type `cache.m3.large`. The EC2 instances were provisioned with Ubuntu 16.04 as operating system, two as web servers running the Bison application, and the other two are used as background workers in Bison, to offload long-running tasks from the web servers. PyPy 5.8 was used as python interpreter. The web servers ran the application through Gunicorn 19.7.1, and the background workers used Celery 4.1.0. The RDS database was running PostgreSQL 9.6.6. Another separate instance was used for load generation. Data was collected in spring 2018.

Tooling and Setup. We used the Python-based open source load testing tool Locust[9] to execute our experiments. To answer RQ1, we added a small modification to Locust to record whether a specific request was served from cache. To this end, we extracted and recorded an HTTP header (`x-cache-hit`) which indicates if the response data was served from the cache.

Our experiments were set up with 20 concurrent simulated users. Based on discussions with application experts from Bison, we designated 4 admin users (primarily executing write requests) and 16 end users (exclusively executing read requests, as per Bison's internal application business logic). An experimental run took a total of 60 minutes, with each user periodically executing one of their designated tasks against the system. Each user was configured with a 2-second delay between tasks. Each task comprised multiple HTTP requests. Consequently, our setup comprised an average of 10 concurrent tasks per second, and 30 concurrent HTTP requests per second.

4.2 Results

We now discuss the results of our measurements in these experiments.

Cache Hit Rate. Answering RQ1, we have found that the existing Bison system with manual caching leads to an overall cache hit rate of 42% of requests in our experiment runs (i.e., 42% of requests can be served from the cache rather than by querying the database). Applying Cachematic improved the cache hit rate to 69%, or by a factor of 1.64. Consequently, we concluded that **using Cachematic indeed improves the cache hit rate compared to the current manual caching solution.**

However, more interesting than whether the cache hit rate increases is whether this improved cache hit rate also led to measurably better user performance, i.e., whether the response time of HTTP requests improved as well. For instance, it is possible that the requests not hit by manual caching are not crucial to performance, or that the additional overhead of caching is higher than the performance gains. This will be investigated in the next two sections.

Read Requests. We plot the response times for read requests (i.e., referring to the *user tasks* in Table 1) in Figure 6. Solid lines indicate median response times, and the shading for each line indicates 25%

[8]https://docs.python.org/2.7/library/pickle.html

[9]https://locust.io

	Name	Description	Req.	Wgt.
User Tasks	**BrowseBenchmark**	User requests a list of available benchmarks; user randomly selects one of the available benchmarks and requests the actual benchmark data	2	1
	VehicleAnalysis	User requests a list of all available vehicle entities; user selects a vehicle and requests a list of available reporting dates; user selects one of the dates and requests meta information about the vehicle; a number of analyses are performed on the vehicle, given the vehicle type, the user's selections, and other input parameters	4	5
Admin T.	**UploadCashflow**	User selects a spreadsheet from a pre-defined set of spreadsheets with example data; spreadsheet is uploaded through a number of sequential requests	4	5
	DeleteEntity	User requests a list of available entities; user selects one that it requests to delete it	2	1
	ShareEntity	User requests a list of available entities; user shares all available entities with another user	2	1
	ChangeAttribute	User requests a list of available entities; user selects a name and requests to update this name	2	1

Table 1: Representative workload patterns of for the experiment. Column *Req.* indicates the number of HTTP requests that this pattern includes, and column *Wgt.* indicates the weight that this pattern is given during execution.

and 75% quartiles. We compare Cachematic to the current manual caching solution, as well as, as a baseline, to disabling caching alltogether. Data is aggregated in tumbling time windows of 30 seconds.

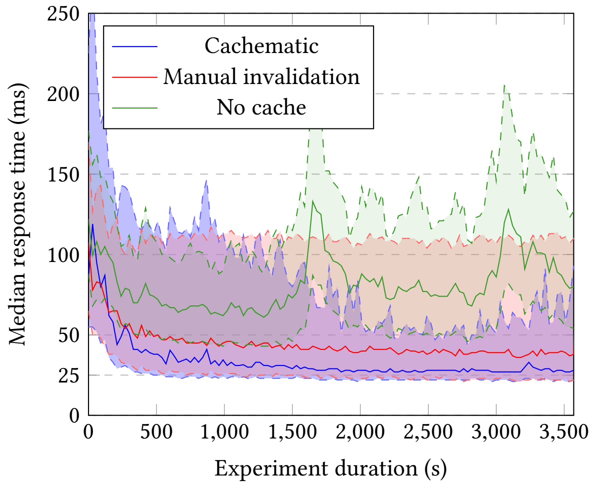

Figure 6: Median response time with 25% quartile and 75% quartile

Evidently, both caching solutions noticably improve read request response time. Without caching, median response time fluctuates around 75ms, with considerable variation. The 25% quartile (i.e., a realistic best case for most users) is around 50ms, the 75% quartile can go beyond 150ms. After a short warmup phase, using the manual caching solution improves the median response time to a fairly stable 37ms. The 25% quartile is around 25ms, and the 75% quartile at around 110 ms. When enabling Cachematic, we are indeed able to further reduce the median response time to around 28ms, which is an improvement by a factor of 1.3. The 25% quartile is similar to the manual caching solution, but the 75% quartile improves considerably, going sometimes as low as 50ms. This reflects that the manual caching solution sometimes "misses" caches which Cachematic is able to handle.

In Figure 7 we depict the 90th percentile response times for read requests, again without cache, with the manual caching, and using Cachematic. This represents a worst-case analysis. We see that initially, while caches are still warming up, the "no cache" setting

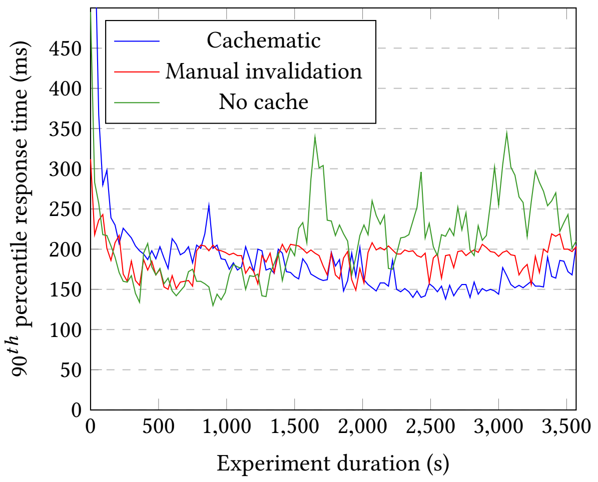

Figure 7: 90th percentile response time for read requests

actually provides better worst-case performance. However, after around 1250s of experiment time, web servers in the no-caching setup start to overload, and we experienced timeout issues, as well as higher and more variable response times. The caching solutions overtake in performance at this point, and settled at a 90th percentile response time of around 150ms (Cachematic) and 200ms (manual caching). Hence, we observe a similar speedup in the range of a factor of 1.3.

We conclude that **using Cachematic indeed improves the user-observed performance (response time) in read requests for Bison by a factor of 1.3 with regard to the existing manual caching solution.**

Write Requests. We now shift our attention to write requests, i.e., the *admin tasks* in Table 1. However, observe that even these workloads have some read components to them (e.g., before editing an entity in Bison, it first needs to be found from the database). Caching can be expected to actually decrease the performance of write requests, but it may improve the performance of the read components of the admin tasks. We again depict median, 25% quartile, and 75% quartile performance in 30s tumbling time windows in Figure 8.

We observe that write requests indeed drastically slow down when using Cachematic compared to both, the no-cache as well as the manual caching solution. Further, write performance of

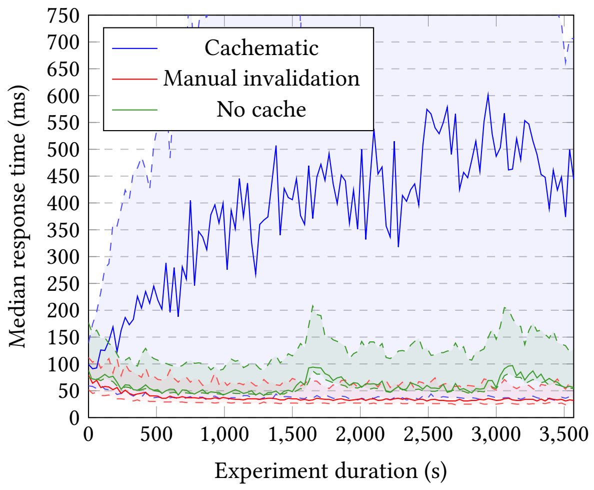

Figure 8: Median response time with 25% quartile and 75% quartile

Cachematic is decreasing over time, which we have traced back to problems with our cache expiration procedure in the experiments. Interestingly, the manual caching solution (which does not need to rely on an automatic cache management) is actually slightly faster than using no caching at all, due to the improved performance of the read requests in writing tasks.

We conclude that the current version of **Cachematic imposes a significant overhead to write requests at the time of study**. After careful analysis, we are confident that some of this overhead is due to limitations in the implementation of the first version of Cachematic, where we focused primarily on a proof-of-concept and improving read performance. We now discuss these limitations in more detail.

4.3 Discussion and Limitations

We have observed that Cachematic performs well on read requests, but imposes a severe overhead on write requests. This is due to three reasons, which we now elaborate.

- Firstly, the high overhead in certain write requests is due to the execution of hash queries. For some write queries, none of the exclusion tests will be able to determine invalidation, and an expensive hash query will be executed. As the cache is populated, and there are more read queries to test, the hash queries will impose significant overhead. In future work, we will experiment with faster hashing functions. Further, we will improve our cache eviction strategy to decrease the number of hashes to evaluate. Currently we used Redis simple least-recently-used eviction, which was not triggered at all in our experiments.
- Secondly, we observe that our experiments have included a higher number of comparatively complex queries. In the evaluation of other systems, such as AutoWebCache [4], the authors highlight that most of the queries executed by the benchmarks such as TCP-W and RUBiS are simple, and the where clause often consists of a single equality condition. 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 (but these would not be representative for the Bison application).
- Thirdly, Cachematic is currently in a proof-of-concept phase, and not much time has been dedicated to optimizing write performance in the implementation of the system. We expect that significant performance improvements can be achieved by re-engineering the implementation of the library with write performance in mind.

After discussion with application experts from Bison, the company has decided to go forward with Cachematic. This is because read request performance is deemed of substantially higher importance to the company's baseline, as virtually all actual end user interactions consist almost exclusively of read requests. However, this substantial overhead in write requests needs to be taken into account, and further engineering improvements along this line will be necessary before production readiness.

4.4 Threats to Validity

It should be noted that our experiments are 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, which we have constructed together with domain experts. Therefore, the workload is arguably a reasonable representation of a real world scenario. However, readers should not assume that our experiments are necessarily representative for a different application.

Further, we have executed our experiments on AWS EC2, a public cloud provider. Previous work has established challenges when using such a system for performance testing [14], particularly as clouds tend to provide a variable amount of systems resources [15]. However, we have repeated our experiments 15 times in the course of our study, and have not experienced notable differences in the outcomes. Hence, we consider this threat to be relatively small.

5 RELATED RESEARCH

We now present related research on the subject of application level caching and, in particular, aspects of cache management and invalidation strategies.

[17] presents a study of ten open source Web application projects. 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 to help developers in designing, implementing and maintaining application level caches. 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, which we address directly using Cachematic.

Further, a multitude of papers describing automatic cache management strategies exist, 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 [6], 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. The approach 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 [9]. 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.

TxCache is a transactional caching system, where dependencies are represented by invalidation tags [20]. 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 indicates whether an index equality lookup is performed. If the query is, for instance, a range query, the second part is explicitly set to a wildcard. Each executed query 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 of PostgreSQL.

Another strategy is implemented in *AutoWebCache* [4], 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, the actual intersection is evaluated in a more precise manner for each of the dependent read queries using the stored information. A system with a trigger based strategy for cache management is *CacheGenie* [11]. 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. The used database triggers are simple, and their only task 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. 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.

The idea of semantic caching is also employed in the approach described in [2], 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 [23]. 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. 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 [5]. 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.

For our work on Cachematic, we have taken various technical and conceptual aspects from a number of these influential research works on caching, and adopted them to our's and Bison's needs. Particularly, we have adopted the notion of cache dependency graphs, as originally presented in [6], as formal basis for our own cache invalidation algorithms. We also adopted the granularity present in AutoWebCache [4] (i.e., invalidating based on table and row scopes). Finally, the trigger-based approach pioneered by CacheGenie [11] is the basis for how we implemented the Cachematic proof-of-concept. Our main contribution was to adapt and integrate these pre-existing ideas into a common Python library, which is easy to use for industrial developers, thereby transferring this research into practice.

6 CONCLUSIONS

We have presented the design, algorithms, and a proof-of-concept of Cachematic, a general-purpose middle-tier caching library for Python. Cachematic has been built for, and in close collaboration with, Bison, a company building a web-based business intelligence platform. The primary goal of the library is to relieve developers from having to manually manage and invalidate application-level cache entries, which has been high-effort, low performance, and bug-prone in the past. Cachematic uses a simple, annotation-based programming model, which makes it easy for developers to integrate into existing systems. The heart of Cachematic are a dependency management and an invalidation algorithm. In our experimental results, we have shown that using Cachematic indeed improves the cache hit rate and read request performance by a factor of 1.64 and 1.3, respectively. However, due to the necessary automated procedures and limitations in cache expiration procedures, write requests appear to be very slow in the current proof-of-concept implementation of Cachematic.

Consequently, our primary ongoing work is to improve performance in write requests, with the ultimate goal of integrating Cachematic into the Bison production environment. To this end, we are working towards improving the implementation of the system with write requests in mind, as so far we focused primarily on read requests. Further, we are working on releasing the library to the community as an open source tool.

ACKNOWLEDGEMENTS

The research underlying this work has been conducted as a master's thesis project while Viktor Holmqvist and Jonathan Nilsfors were students of Chalmers University of Technology. The first two authors contributed equally to this work. We are very thankful for the support provided by Bison in conducting this work. Philipp Leitner acknowledges the financial support granted by the ICT Area of Advance at Chalmers.

REFERENCES

[1] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy. Adaptive Time-to-Live Strategies for Query Result Caching in Web Search Engines. In R. Baeza-Yates, A. P. de Vries, H. Zaragoza, B. B. Cambazoglu, V. Murdock, R. Lempel, and F. Silvestri, editors, *Advances in Information Retrieval*, pages 401–412, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] C. Amza, G. Soundararajan, and E. 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.

[3] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive Database Caching with DBCache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.

[4] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. 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.

[5] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. 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.

[6] 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.

[7] H. Chen, C. Cutler, T. Kim, Y. Mao, X. Wang, N. Zeldovich, and M. F. 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.

[8] 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.

[9] L. Degenaro, A. Iyengar, I. Lipkind, and I. 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.

[10] B. Fitzpatrick. Distributed Caching With Memcached. *Linux Journal*, 2004(124):5, 2004.

[11] P. Gupta, N. Zeldovich, and S. 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.

[12] V. Holmqvist and J. Nilsfors. Cachematic – Automatic Invalidation in Application-Level Caching Systems. Technical report, Chalmers University of Technology, 2018.

[13] H. Krawczyk, R. Canetti, and M. Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[14] C. Laaber, J. Scheuner, and P. Leitner. Performance Testing in the Cloud. How Bad is it Really? *Empirical Software Engineering*, 2019. To appear.

[15] P. Leitner and J. Cito. Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, Apr. 2016.

[16] C. V. Manikandan, P. Manimozhi, B. Suganyadevi, K. Radhika, and M. Asha. Efficient Load Reduction and Congession Control in Internet Through Multi-level Border Gateway Proxy Caching. In *2010 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–4, Dec 2010.

[17] J. Mertz and I. Nunes. A Qualitative Study of Application-Level Caching. *IEEE Transactions on Software Engineering*, 43(9):798–816, Sept 2017.

[18] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[19] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, Dec. 2003.

[20] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. 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.

[21] 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.

[22] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of Web Caching Architectures: Hierarchical and Distributed Caching. *IEEE/ACM Trans. Netw.*, 9(4):404–418, Aug. 2001.

[23] Z. Scully and A. 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.

[24] S. Sivasubramanian, G. Pierre, M. Van Steen, and G. Alonso. Analysis of Caching and Replication Strategies for Web Applications. *IEEE Internet Computing*, 11(1), 2007.

[25] J. Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.

[26] N. Zaidenberg, L. Gavish, and Y. 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.