Caching for Improved Response Times in a Distributed System

Viktor Enzell
Abstract

Caching for Improved Response Times in a Distributed System

Viktor Enzell

To cope with slow response times that emerge in data-centric web applications, caching can be used to avoid unnecessary database queries and recalculations. Slow response times become prevalent when using Insights — a tool that gathers data from continuously expanding databases and summarizes it into statistical information. Insights has a master-slave system architecture, composed of one central server and a number of distributed servers with accompanying databases. A solution that entails caching server responses in each of the distributed servers is proposed, and a prototype is developed. The cache is filled both by computing responses for common requests in advance and by dynamically updating the cache. Randomized tests that simulate expected access patterns show that the prototype has a better average hit ratio than a purely dynamic cache and a notably improved response time compared to having no cache, rendering it a promising cache design to appropriate in the Insights system.
## Contents

1 Introduction ........................................... 4
   1.1 Prior Work ......................................... 4
   1.2 Contributions ...................................... 5

2 Background ............................................. 5
   2.1 The Insights System ................................. 5
   2.2 The Bottleneck ...................................... 7
   2.3 Expected Use of Insights ............................ 7

3 Requirements .......................................... 8
   3.1 Consistency Model ................................. 8
       3.1.1 Eventual Consistency ......................... 9
   3.2 Cache Size ......................................... 9

4 Proposed Methodology .................................. 10
   4.1 SQL vs. NoSQL .................................... 10
   4.2 Centralized vs. Decentralized ...................... 11
   4.3 Static vs. Dynamic Caching ......................... 13
       4.3.1 Prefetching Data ............................ 13
       4.3.2 Replacement Policy ......................... 14
   4.4 In-Memory vs. Persistent Storage ................... 15
   4.5 Redis vs. Memcached ............................... 15
   4.6 Proposed Solution ................................. 16

5 Implementation .......................................... 18
   5.1 The Prototype ..................................... 18
   5.2 The Test Script .................................... 19
   5.3 Determining Cache Size ............................. 20

6 Evaluation ............................................. 22
   6.1 Prototype Performance Compared to a Purely Dynamic Cache .......................... 22
       6.1.1 Comparing Hit Ratios ....................... 22
       6.1.2 Comparing Lookup Time ...................... 23
   6.2 Improvement of Response Time ....................... 24

7 Related Work .......................................... 25

8 Conclusions and Future Work ............................ 26
1 Introduction

Uppsala-based software company Connectel delivers customer service solutions to companies [1]. These solutions include software for companies to communicate with their customers, via IP telephony for example. Information about different events, such as time and duration of phone calls, are stored in database servers to track the usage of the services. In order to make sense of the data, Connectel has developed Insights. Insights is a tool for gathering, summarizing, and displaying data produced by the different software.

There is a large amount of data continuously being produced, which makes gathering data a slow process. This problem could be solved by caching [2]. Caching is the process of storing some data which is believed to be requested more frequently in a memory that allows for a faster time of retrieval. By storing data which is accessed more frequently in a cache, the response time of Insights could be vastly improved.

This thesis aims to establish an efficient solution for caching in the Insights system. The objectives of the thesis are listed below. The system is explained in more detail in section 2, and the requirements of the cache are outlined in section 3.

1. To examine different system-specific implementation choices such as where in the system to implement the cache (section 4).
2. To examine different means of caching, such as dynamically replacing content and prefetching (section 4).
3. To develop a prototype cache based on the previous findings and test its performance by simulating access patterns (sections 5 and 6).

1.1 Prior Work

Caching can be employed in many different ways, a few of which are applicable in the case of Insights. Three studies that have sought to solve similar problems using different methods are mentioned here. A wider range of work can be found in the related work section.

Larson, Goldstein and Zhou developed MTCache for caching in a multi-tier application with a SQL Server as the back-end database server [2]. MTCache implements a cache by introducing an intermediate server between the application server and the back-end server. The intermediate server contains a shadow database that partially replicates data from the database server. What data to replicate is decided in advance, and the data is then kept up to date by SQL Server replication. All queries are routed through the intermediate server where they are either computed locally, remotely or part locally and part remotely — constraining the workload put on the back-end database.

A different but perhaps more frequently used approach is to implement the cache as a NoSQL database, for example as a key-value store [3]. In a comparative study by Markatos, a NoSQL cache is used to store search engine query...
The study compares two different approaches for caching: static caching and dynamic caching. Static caching is done by looking at previous usage and filling the cache in advance (prefetching). Whereas dynamic caching exploits temporal locality, i.e., recently accessed objects are more likely to be accessed again. In dynamic caching, the cache is filled as queries are made, and when the cache has reached its maximum memory capacity, keys are evicted according to a predefined content eviction policy (replacement policy).

Fagni, Perego, and Silvestri suggested an approach to caching that combines static and dynamic caching, which they refer to as SDC (Static Dynamic Cache) [5]. Fagni et al. did this by statically prefetching search engine query results of queries that were known to be popular and reappearing as well as dynamically updating the cache to cover queries that were hard to predict, by looking at historical data. In this way, they developed a very versatile cache that could cover a wide range of popular queries.

1.2 Contributions

The cache solution suggested in this thesis combines static and dynamic caching. A solution similar to the one suggested by Fagni et al., but it is optimized to fit the Insights system architecture [5]. This solution utilizes the fact that there are some predefined requests which are more likely to occur, by prefetching the results of those requests. Whereas other requests that are harder to predict are only stored when they are made and then replaced based on the least recently used (LRU) replacement policy [6].

A prototype based on the solution is developed and tested. Test results show that the prototype performs well compared to a purely dynamic cache and that implementing the cache could vastly improve the response times for the end-user of Insights.

2 Background

2.1 The Insights System

Insights is a distributed system composed of servers and databases for the companies using the tool. For each company using Connectel’s customer service software, events are recorded, and data is being generated by a service called Motion. Data from each company is stored in a separate MySQL database, where all events are stored as separate rows [7]. Insights is used to retrieve and summarize data from the Motion databases. Both Insights and the databases are hosted with the same cloud service\(^1\).

An overview of the Insights system architecture can be seen in figure 1. The system consists of a front-end client (referred to as Insight client), the main server (referred to as Insights server) and one server for each of the companies.

\(^1\)Cloud service refers to an on-demand service providing remote computer system resources.
using the system (referred to as Insights node). Each Insights node can access the respective database containing the event data (referred to as Motion database). The server and the nodes are implemented in the Go programming language [8].

Figure 1: An overview of the Insights system architecture. A request contains an id of the company as well as the time period to consider.

Insights can be used by Connectel to display statistics about a specific company’s customer service traffic during a specified time period. The same functionality is present for the other companies using Insights but with the restriction to only view statistics about the own company’s traffic.

A request from the client is routed through the main server to a specific node. When using Insights a user specifies the time period to consider in the user interface, in order to display statistics for the specified period. A request in the form of a JSON object containing an id of the company, the start time and the end time of the period is created. This object is sent to the Insights server and then to the specific Insights node. The node makes several queries to the Motion database collecting all events during the specified period. The node then summarizes the events to statistical parameters and sends it back to the Insights server, which sends it to the Insights client where it is displayed to the user. Whenever a change is made to the start time or end time, a new request is sent to the server. Even if the page is reloaded, the request is sent again.

Some additional options are available for the user when choosing what to display. These options are used by the Insights client to display only specific information and do not affect the request sent to the Insights server.

There is already a naive implementation of caching, which stores the ten latest responses in the Insights server; this method is not sufficient enough and will be discarded. This thesis focuses on the system as it is without regard for the previous cache.

---

2JavaScript Object Notation (JSON) is a standardized data format for web communication.
2.2 The Bottleneck

The delays in the system are not the same for all companies using it due to a couple of reasons. An approximation of different delays in the system can be seen in figure 2. One reason that the delays differ for the different companies is that a few of the companies do not have their servers hosted in the cloud but have their own servers. This introduces a substantially higher end-to-end delay between the server and those nodes, reaching up to three seconds. The majority of companies have their servers hosted with a cloud service, making the end-to-end delay negligible. Another reason is that the amount of data varies a lot depending on how actively the services are used. For some companies, the delay of making all the queries for one request can reach two minutes and for others five seconds.

It is clear that the main bottleneck of the system is in making multiple queries to a database each time a request is sent. In order to decrease the response time for the end-user, query responses from the database can be cached so that the calculation of statistics can be made faster. Alternatively, the calculated responses can be cached to avoid recalculation.

![Diagram of Insights components and delays](image)

Figure 2: An overview of where different components in Insights are hosted and approximated round-trip delays between different components.

2.3 Expected Use of Insights

In order to implement an efficient cache, it is essential to know what type of requests that are common for users to make. Therefore the expected user behavior is examined.
The purpose of Insights is for companies to get an overview of their customer service traffic over time. Therefore it is expected that users most often display statistics of a general time period, e.g., last week or a specific calendar month this year, rather than a more specific period, e.g., from last Tuesday to last Thursday. There are six different predefined time periods that the user can choose from: "today", "yesterday", "this week", "this month", "one week", and "one month". The user also has the choice to change the time of day for the time period, but it is not anticipated to be common use of the system.

There does not exist any statistical data to support if companies use Insights in this way. This could be solved by sending out a survey to the companies using Insights to get a better understanding of their usage of the service. Alternatively, this could be investigated by logging the requests sent to the Insights server during a specified time period. The log files could then be parsed to see the most common time periods in the requests. In this thesis, an access pattern based on the expected user behavior is used instead.

3 Requirements

There are a few requirements and prerequisites that need to be satisfied when exploring different methods. Those involve the consistency model to satisfy and the cache size.

3.1 Consistency Model

Since data from the databases will be replicated in the cache, there needs to be some guarantee that the replicated data reflects the current state of the databases. This can be handled by using a consistency model, which is a set of rules that are used in systems where memory is distributed, like distributed shared memory systems or distributed data stores [10]. Following a consistency model, guarantees that memory will be consistent and that read, write, and update operations on memory will be predictable. It is often the case in distributed systems that there are multiple distributed caches holding the same data; in those cases, consistency must be kept between the caches. In the case of Insights, there is only one cache per database, so consistency must only be maintained between the cache and the database.

There are many different consistency models, ranging from strong to weak. The trade-off for having a strong consistency model is that it is generally slower and more demanding to implement because it might, for example, require concurrent transactions to be ordered in a sequential manner (sequential consistency) [9]. Having a strong consistency model is sometimes needed, for example, when handling bank transactions. Since the validity of bank transactions are not allowed to be compromised, they are allowed to be quite slow. In cases when the validity of data is of utmost importance, the well-known ACID properties (atomicity, consistency, isolation, and durability) often come into play [10]. In applications where validity is not as important, in a chat, for example, speed is
often preferred over validity [9]. In such cases, an eventual consistency model is often used. Two popular eventual consistency models used are BASE (basically available, soft state, eventually consistent) and SALT (sequential, agreed, ledgered, tamper-resistant) [11]. Insights falls into the category of applications where speed is preferred over consistency.

3.1.1 Eventual Consistency

How Insights is used and how the system is structured makes eventual consistency a sufficient consistency model.

Insights only performs read operations on the databases. Data is produced by Motions when an event has occurred, that data is then written to the database with the time and date of the event included. A row in one of the database tables will never be altered since it is just a recording of a past event, so there will only be new rows added to the database. As long as the end time of a query has already passed when the database is accessed, it is safe to cache the result without further measures to guarantee consistency. This, however, is not the case if the end time has not passed yet, like in the case of choosing the predefined choice to gather information about today. In the case of gathering information about today, the queries gather results from all 24 hours of the current day, even though the whole day has not yet unfolded. So if such a query is cached earlier in the day, trying to access it later in the day will yield the same result even though more events might have been written to the database. So in order to always keep the cache consistent with the database, queries with an end time later than the time of when the database is accessed must be continuously updated or not cached at all.

For the cache needed in Insights, a general eventual consistency model can be implemented since Insights is not meant as a real-time tool and not having the absolute latest data is not that important. Eventual consistency means that if no updates are made to a data item, then all replicas of the item will eventually have the latest updated value [9]. This is a weak consistency model and for it to mean anything in the context of Insights, a maximum time limit for how long a value in the cache is allowed to be inconsistent with the database needs to be set.

3.2 Cache Size

The size of the cache has to be a compromise between available RAM in the server where it is placed, and the size needed to make the cache efficient enough. The size needed to make the cache efficient depends on if the cache will be placed in the server or in the nodes since if it is placed in the server, it will have to cache data for all of the companies. It also depends on the size of the objects being stored in the cache.

Both the Insights server and the nodes are running on cloud servers with 4 GB of RAM. The cloud servers can utilize more RAM if the workload is exceeded but for an additional cost. A sample of the workload in the Insights
server was measured to 3.2 GB. A sample workload for one of the nodes was 2.2 GB. There are about 30 companies using Insights, which has to be taken into consideration if the cache is placed in the server. These are things that need to be considered when a solution is proposed.

4 Proposed Methodology

Caching does not entail a specific method but is instead a concept which can be used in different areas of computer science. Caching needs to be tailored to the specific case at hand, which is the purpose of this section. System-specific design choices to consider are in what system component the cache should be implemented and what type of storage to use. More general design choices such as to prefetch or to dynamically replace content are also considered.

4.1 SQL vs. NoSQL

One important aspect to consider when implementing the cache is whether to implement it as a MySQL database that shadows the Motion database like Larson et al. or to implement it as a NoSQL database like Markatos [2, 4]. The benefits and drawbacks of these methods will be investigated in the following paragraphs in order to settle on the seemingly most efficient and appropriate method for Insights.

Shadowing a MySQL database might not be the standard way of caching, but it has a few advantages. The way it would work is by having a MySQL database in each of the nodes that shadow the corresponding Motion database. The shadow database would initially be empty but would be filled by defining in advance, which materialized views to cache. The materialized views would then be kept up to date by master-slave replication [7]. A query from the node would first be sent to the cache which either computes the query or forwards it to the Motion database. One advantage of this approach is that the cache can be kept consistent with the Motion database with barely any delay through replication. Another advantage is that the same queries that are being made from the node can be sent to the new database without having to restructure much in the existing code.

The MySQL approach also has a few disadvantages. One disadvantage is that the summation of the gathered statistics needs to be repeated for each request that is sent to the node. This does not only lead to a higher workload being put on the node, but it also leads to more memory being used since event data is cached instead of only caching summations of the event data. Another disadvantage is that it would likely be quite challenging to dynamically update the cache based on recent requests since that would require a reconfiguration of the shadow database, making it more suited for prefetching.

The more straightforward approach of using a NoSQL database as a cache also has its advantages and disadvantages. This would be done by caching the already calculated response from the node server and storing it with the time
period as the key. Using this approach enables the possibility to choose whether to have one cache for all the companies in the server or to have a separate cache for each node. An advantage of this approach is that the summation of event data does not need to be repeated for each request, leading to a decreased workload in the server. This would also take up less memory since only the summation of event data is stored. Additionally, this approach is quite versatile since it can be implemented as a static cache, dynamic cache, or as a combination of both. The main drawback of the NoSQL approach is that it is harder to keep consistent with the Motion database since there is no master-slave relationship between the cache and the database as there is in the MySQL approach.

In conclusion, the NoSQL approach is very versatile, and storing already calculated responses has the potential to be quite efficient compared to storing event data. The drawback of the NoSQL approach is that it might be harder to keep consistent with the database. However, considering that eventual consistency is the consistency model that will be used, this is not a major drawback. Based on these conclusions a NoSQL cache will be used. The cache will either be centralized (in the Insights server) and contain data from all the companies, or it will be decentralized (in the Insights nodes) and only contain data from the specific companies.

4.2 Centralized vs. Decentralized

Based on the system architecture of Insights, there are three places where caching could occur: in the web browser/client, in the central server, or in each of the nodes. Only caching in the web browser will not be considered an alternative since that would limit caching to only be available for the current user during the current session. Therefore it is the benefits and drawbacks of centralized caching and decentralized caching that will be investigated, see figure 3. Based on which method has the greatest potential based on overall speedup and resource efficiency, a method will be chosen.
Figure 3: The possible locations to implement the cache. Either only one cache is implemented in the Insights server which stores data from all companies in one cache, or one cache per Insights node is implemented to only store the data relevant to the specific company.

Having a centralized cache can be summarized by the following pros and cons. Probably the best reason for having the cache centralized is that it would remove the round-trip time (RTT) between the server and the nodes on a cache hit. This would be very beneficial for the nodes that are not hosted on cloud servers since the RTTs to those nodes are quite substantial. One drawback of a centralized cache is that there is a single source of failure, i.e., if the server goes down, then the cache is emptied for all companies using Insights. Another possible drawback is if all companies share the same cache and if the cache is dynamically replacing keys, then the companies compete for the same memory space. Leading to a disadvantage for companies that use Insights more seldom since many replacement policies favor keys that are used more frequently [6]. Another drawback is that the memory size of the cache has to be larger if it is centralized in order to fit the same amount of data as if it is decentralized. There are about 30 companies using Insights, so if the cache is centralized, it would have to be about 30 times larger.

Decentralizing can be considered the opposite of centralizing, making the pros and cons of decentralizing the opposite to those of centralizing. A benefit of letting each node have its own cache is that the memory size can be kept substantially lower and yet gain the same benefit. Another benefit is that there is not a single source of failure if one of the nodes or the server goes down. However, the RTT between the server and the nodes will not be escaped on a cache hit.

It is not obvious that one of the methods has a clear advantage over the other in terms of overall speedup and resource efficiency. Decentralizing might gain a higher overall hit ratio since the companies do not need to compete for the same memory, and since all caches will not be emptied if a server or node goes down. Centralizing might be faster in the long run since the round-trip to the nodes can be avoided. In terms of resource efficiency, if the cache is decentralized,
the caches can be smaller in size, but that would also make the overall usage of network and server resources greater. It is hard to conclude anything about which method has a better overall speedup and resource efficiency, but it can be concluded that decentralizing is more scalable. If Insights grew with many more companies connecting, it would be hard to maintain an efficient cache if it was stored in the central server. Therefore the decentralized approach will be chosen. Whether the cache should be static or dynamic is left to consider.

4.3 Static vs. Dynamic Caching

Another essential design choice when caching is whether to have a static cache which is filled in advance or a dynamic cache which is dynamically updated. How these two approaches would be implemented, and the benefits of each approach will be further investigated. A choice between static and dynamic caching or a combination of both is then made based on what is best suited for the expected usage of Insights.

Static caching demands some knowledge of expected usage of a service in order to know what to prefetch, and there are a few things that can be said about the expected usage of Insights. Firstly, since there are six predefined options a user can choose from when displaying information, those are good candidates for prefetching. Secondly, it is expected that users request information about general time periods such as a specific calendar month or week. This makes those time periods candidates for prefetching as well.

Static caching has its advantages and disadvantages. One advantage is that objects are cached before they are ever used, so if the access pattern is predictable, the cache can perform very well. A disadvantage is that a static cache does not adapt to the current access pattern. This is something that a dynamic cache does very well.

A dynamic cache can assist with tasks that a static cache is not capable of. One task is that a dynamic cache will store the latest request so that if a user reloads the page or makes a few other requests, then the request will not have to be computed again. For prolonged usage, the cache can adapt quite well to the situation based on its replacement policy.

To conclude, Insights could benefit from both static and dynamic caching since both are good at handling different things. An approach that combines prefetching regular time periods with dynamically filling the cache with previous requests without removing the prefetched ones is what seems to be the most suitable. This is much like the SDC approach suggested by Fagni et al. [5]. Exactly what data to prefetch and how often needs to be considered.

4.3.1 Prefetching Data

What time periods to prefetch and how often to prefetch needs to be considered in order to make an efficient and consistent cache. The time periods to prefetch are chosen based on the expected use of the system and how often prefetching needs to occur is decided based on the need for consistency.
There are a few time periods that seem reasonable to prefetch. The six predefined time periods are reasonable to prefetch since they can be expected to be used quite often. Also, it seems reasonable to prefetch each calendar month one year back and each week one month back. With these time periods prefetched, it would be easy to get an overview of past usage. This sums up to 22 different prefetched objects which will not take up a considerable amount of memory.

All time periods will not have to be updated at the same time. All predefined time periods except "yesterday" include today, i.e., "one week" is the time period starting six days back and including the current day. This makes requesting "one week" return data produced during six days if the request is made in the morning and seven days if the request is made in the evening. So if "one week" would be prefetched and cached, it would be inconsistent with the motion database as soon as more data was produced. It is questionable if this is actually wanted behaviour of the system since the terminology is inconsistent with what is actually retrieved. This could be solved so that "one week" always yields data from yesterday and the six days before that. However, this thesis focuses on how the current system can be improved through caching, so improvements to other parts of the system will not be considered. Instead of changing the terminology, the rate at which the prefetching occurs can be increased. Since Insights is not intended as a real-time tool, it is okay if the very latest produced data is not included. Therefore, prefetching once every hour should keep the cache consistent enough. In this way, the response of for example, "one week" never discards more than the latest hour of produced data. The calendar months need to be updated each new month, and the weeks need to be updated each new week.

4.3.2 Replacement Policy

Since a combination of a static and a dynamic cache will be used, a replacement policy will only be relevant for the dynamic part of the cache. A replacement policy for the dynamic part of the cache is chosen based on its ability to satisfy the access patterns of Insights.

There are generally two types of locality that explain access patterns to a cache: temporal locality and spatial locality [6]. Temporal locality refers to access patterns where an object that was recently accessed is more likely to be accessed again. Spatial locality refers to access patterns where access to some objects can be a predictor of future accesses to other objects. What can be said about Insights is that temporal locality is applicable since a request will be sent again if the web page is updated, making a request that was recently made more likely to occur again. There are, however, no justified conclusions that can be made about the spatial locality of the access pattern since there are no request logs to study.

Since there is not a lot more that can be predicted about the access pattern of Insights, a replacement policy that exploits temporal locality should be used. LRU is a commonly used replacement policy that does exactly this. The
memory overhead of LRU is low and performing delete and insert operations take constant time [6]. With this in consideration, LRU will be the choice of replacement policy for the dynamic part of the cache. Most decisions about the design of the cache have now been made, but whether to save the cache to persistent storage is left to consider.

4.4 In-Memory vs. Persistent Storage

A cache is in many cases only implemented as an in-memory storage since it is supposed to be fast and is not a way of backing up data, but when the cost of the cache going down is high, the cache might be backed up in persistent storage. The benefits and drawbacks of these two approaches will be compared, and the approach that contributes to the best overall performance will be chosen.

Backing the cache up to persistent storage has one main benefit but comes with a few drawbacks. The benefit of using persistent storage is that if an Insights node crashes then the cache is not emptied when it starts up again, which would improve the overall hit ratio. However, since the caches are distributed to the nodes, the consequence of the cache being emptied would only affect the specific company for a while before the cache has filled up again. Another drawback of backing up to persistent storage is that it demands more memory resources as well as computational resources for always keeping the persistent storage consistent with the cache.

In conclusion, the one benefit of backing up to persistent storage would likely not have a large enough positive impact on the hit ratio to compensate for the extra use of resources. Therefore the caches will only operate in-memory and not be backed up to persistent storage. A service that supports these design choices needs to be found.

4.5 Redis vs. Memcached

A cache that is compatible with the development environment and that supports the previous implementation choices must be chosen. Hence, the cache must be compatible with Go since the servers are written in Go, it must be possible to implement it both as a static and as a dynamic cache and the LRU replacement policy must be possible to utilize.

Two in-memory storages will be considered, namely Redis and Memcached [3]. Both alternatives are compatible with Go and can be implemented as LRU caches as well as static caches. These caches are considered since they are widely used, fast, free and provide different features that might make them well suited for the purpose. Redis is an in-memory key-value store that is used as a database, cache or a message broker. Redis supports storing a wide range of data structures and allows for a lot of custom configuration. Memcached is an in-memory key-value store with support for storing strings, it is mainly used as a cache. Memcached has fewer features than Redis but utilizes concurrency and can be faster in some cases.
Memcached is a bit less versatile than Redis, but it is possible to implement it as a combined static and dynamic cache. In order to do that, a time to live (TTL) can be set for the static keys in the cache. With this approach, the keys without a specified TTL would be replaced dynamically, and the static keys would be evicted when the TTL has expired. The prefetching of keys would happen based on timers that are set for when the different keys should be prefetched.

Redis could be implemented as a static and dynamic cache in a couple of ways. One way is to have the same approach as for Memcached. However, mixing usage in the same instance is advised against in the Redis documentation [12]. The documentation instead encourages the use of multiple Redis instances in such cases, where two Redis server instances could be run, one instance acting as a dynamic cache with an LRU replacement policy and another instance with no replacement policy but where all keys have a TTL. Having two Redis instances is not optimal since both instances might have to be searched. This should however not be a problem since the lookup time in Redis is constant [12]. Therefore, adding another instance is not something that should have a significant effect on performance.

Redis and Memcached have chosen different approaches for solving certain problems [3]. Redis uses naive memory allocation which can make the memory more fragmented than the slab-based memory allocation strategy that Memcached uses. However, Redis makes up for this by only allocating the amount of data needed at the moment. Memcached provides a maximum object size limit of 1 MB and Redis provides a limit of 512 MB.

In most aspects, both Redis and Memcached are good alternatives for the Insights cache. One aspect that disqualifies Memcached, however, is the maximum object size limit of 1 MB. This is because an object stored in the cache can be expected to have a size of around 750 kB (sample measurement of the size of a server response), but there is no guarantee that it will not exceed 1 MB. Therefore, it is not guaranteed that Memcached can hold all objects so Redis is the cache that will be used. All design choices have now been made, and a summary of the design of the cache can be seen in the next subsection.

### 4.6 Proposed Solution

The design choices and methods in the previous subsections led to the following solution, of which an overview can be seen in figure 4. The solution which will be referred to as the Insights Cache (IC) entails having a separate cache for each of the companies, so one cache per node. The IC is a key-value store that stores the responses that would otherwise be calculated in the node each time by making multiple database queries and summarizing the results into statistics. The objects are stored in JSON object format so that a response can be sent immediately if a request is received and the response is already cached. The cache only operates in-memory.

The IC is a combination of a dynamic and a static cache. The dynamic part of the cache is filled while being used, and keys are replaced when the memory limit is reached based on the LRU replacement policy. The static part of the
cache is filled by prefetching based on what the user is believed to request. Prefetching is scheduled based on when the different keys are outdated. This can be done with the help of timers that invoke objects to be computed. A TTL with the same length as the duration of the corresponding timer will be set for all the prefetched objects. In this way the old keys get evicted at the same time as the new ones are retrieved. Redis is the choice of service and to implement the combined static and dynamic cache, two Redis instances will be used: one with the LRU replacement policy, and one with no replacement policy.

Time periods that will be prefetched and continuously updated can be seen in table 1. The predefined choices "today", "this week", "this month", "one week" and "one month" will be updated every new hour to keep the cache sufficiently consistent. The predefined choice "yesterday" will be updated every new day. A few additional objects will be prefetched: all previous months one year back and all previous weeks one month back. The months need to be updated every new month, and the weeks need to be updated every new week. Before adding a key to the dynamic cache, there must be a check to see if the end date of the time period is in the future, if it is then it should not be added. This is because a key with an end date in the future should only be cached if it is prefetched, since it could otherwise corrupt the result of a future request when more data has been added to the database. Ideally, the static cache would implement a first-in, first-out replacement policy but it is not supported by Redis. Instead, a TTL is set for all keys in the process of prefetching, which will result in similar behavior.

<table>
<thead>
<tr>
<th>Time periods</th>
<th>When to update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Today, this week, this month, one</td>
<td>Every new hour</td>
</tr>
<tr>
<td>week, and one month</td>
<td></td>
</tr>
<tr>
<td>Yesterday</td>
<td>Every new day</td>
</tr>
<tr>
<td>All previous weeks one month back</td>
<td>Every new week</td>
</tr>
<tr>
<td>All previous months one year back</td>
<td>Every new month</td>
</tr>
</tbody>
</table>

Table 1: The time periods of the requests to prefetch in the static part of the cache. As well as the time when each request needs to be evicted and fetched again.
5  Implementation

A prototype cache based on the proposed solution in the previous section is implemented. A script is developed in order to test the performance of the prototype with requests of a partly random access pattern. An appropriate size of the cache is then estimated based on the hit ratio for different cache sizes of the prototype.

5.1  The Prototype

The proposed solution is a cache that uses two Redis instances, one with the LRU replacement policy and one with no eviction policy which is updated through prefetching. A prototype referred to as the IC prototype is developed based on this solution in order to demonstrate how the cache can be implemented and to test its performance. However, it is developed independently from the Insights node. Therefore a replication of one of the Motion databases is created in order to develop the IC prototype locally. The main difference between the IC prototype and the IC is that the IC prototype makes database queries to the replicated database and stores the query results instead of storing the calculated node responses, see figure 5. Hence, the IC prototype cannot be directly integrated with the node, but the same logic can be used.

Figure 5: An overview of the implemented prototype.

Python is the programming language used to develop the IC prototype since it is a well-suited language for prototyping, and it has support for Redis [13]. When setting up a Redis cache, the actual cache is called a server instance [12]. One server instance is one in-memory storage that has its own configuration, and its memory space is separate from other instances. In order to connect to a server instance, a Redis client is used. To implement the IC prototype, two server instances with different configurations are needed. One instance for the dynamic part of the cache and one for the static part of the cache. The IC prototype is a Python class that when instantiated opens a connection to the MySQL database and starts two Redis clients, connecting to the two Redis instances.
A request that is sent to the IC prototype has the form of a time period as a string containing a start date, end date, start time and end time, e.g., "2019-02-04 2019-02-10 00:00 23:59". To query the database, one of the queries that are used in the Insights node to gather statistics from the database is used. The query gathers data within the specified time period. Time period strings are used as the keys in the cache, and the corresponding database query results as the values. When a request is sent to the IC prototype, the static cache is first searched to see if any key matches the request, if it does then that value is returned. If no matching key is found, the dynamic cache is searched and a value is returned if it is found. Otherwise, the request is sent to the database and the result is stored in the dynamic cache and then returned. The method for making a request can be seen below.

```python
def make_request(self, time_period):
    cache_hit = True
    result = self.static_cache.get(str(time_period))
    if result is None:
        result = self.dynamic_cache.get(str(time_period))
        if result is None:
            cache_hit = False
            result = self.database.get(time_period)
            self.dynamic_cache.set(str(time_period), result)
    return result, cache_hit
```

Since the IC prototype is not running on a server, no timers are set for prefetching. Instead, the static cache instance is filled with the results of a pre-defined set of keys. This is done through manually calling a method instead of having a set of timers invoking the method. The IC prototype can be instantiated and called independently but to test the performance of the cache a test script that simulates an expected access pattern is developed.

5.2 The Test Script

A script was developed to test the IC prototype. Since there are no user logs to take common requests from, randomly simulated user behavior is used. One iteration in the test corresponds to one user making requests during one session.

Before the first test iteration starts, a pool of 200 random time periods are created, and the 22 predefined time periods are prefetched into the cache. The random time periods are generated by randomizing a start date and an end date within a time period of one year, while the time of the day always stays the same. This yields an amount of $\sum_{n=1}^{365} n = 66795$ possible requests since there are 365 days in a year and any day can be the start date followed by an end date that is the same or further in the future.

Every iteration, a random amount of requests are made. Some of which are picked from the set of random time periods and some of which are picked
from the set of predefined time periods. The number of random requests each iteration is between 0 and 30 and the number of predefined requests is between 0 and 10, which sums up to 0 to 40 requests being made each iteration. The reason for having more random requests than predefined requests is that the predefined requests will result in a cache hit each time since they are prefetched. The reason for having a pool of 200 random periods instead of having totally random periods each iteration is that having totally random periods would render the dynamic cache useless since requests would not be reappearing. The test runs for a number of iterations, and the hit ratio in the cache is measured for each iteration.

This test has a major flaw in that the performance of the cache is completely dependent on the amount of randomness of the requests that are made. Less randomness will yield a better cache performance since the same requests will be reappearing more often, whereas more randomness would yield a worse result since the same requests will appear more seldom. However, predefined requests are expected to be frequently used, and requests are expected to reoccur quite often, and the test has been written thereafter.

5.3 Determining Cache Size

How much memory the cache is allowed to allocate affects the performance, so the hit ratio of the IC prototype is tested in order to find an appropriate size for the IC. The size of an object in the IC prototype does not correspond to the size of an object in the IC. However, the average size of an object that would be stored in the IC is known and the average size of an object in the IC prototype is known. To determine what is an effective cache size for the dynamic instance of the IC prototype, the test is run with different configurations of the memory limit and the hit ratio is measured. The size is then multiplied so that the same amount of keys would fit in the IC as in the IC prototype.

The test is run with nine different configurations of the memory limit of the dynamic cache instance, which can be seen in table 2. The actual size of the objects combined is not the same as the memory limit when the memory limit is very low [12]. So the actual size is considered instead of the memory limit since it is the amount of objects that fit in the cache that matters. The results from the tests can be seen in figure 6. For each cache size the test ran for 20 iterations and the figure shows an average hit ratio for each size. The results show that the hit ratio stops increasing after a certain limit has been reached. After the 150 kB mark, the hit ratio does not increase much. Therefore this will set the memory limit for the dynamic part of the IC prototype.
<table>
<thead>
<tr>
<th>Memory limit (kB)</th>
<th>Amount of keys</th>
<th>Actual size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>880</td>
<td>12</td>
<td>22</td>
</tr>
<tr>
<td>920</td>
<td>30</td>
<td>58</td>
</tr>
<tr>
<td>960</td>
<td>46</td>
<td>89</td>
</tr>
<tr>
<td>1000</td>
<td>68</td>
<td>122</td>
</tr>
<tr>
<td>1040</td>
<td>84</td>
<td>156</td>
</tr>
<tr>
<td>1080</td>
<td>100</td>
<td>191</td>
</tr>
<tr>
<td>1120</td>
<td>120</td>
<td>225</td>
</tr>
<tr>
<td>1160</td>
<td>141</td>
<td>260</td>
</tr>
<tr>
<td>1200</td>
<td>152</td>
<td>286</td>
</tr>
</tbody>
</table>

Table 2: The different memory limit configurations that the test is run with, the actual amount of keys that fit in the cache and the combined size of the values at those keys.

Figure 6: The average hit ratio for the dynamic instance of the IC prototype using different memory limits.

An average object in the IC prototype has a size of 2259 B. Whereas an average JSON response from one of the nodes (which would be what is cached in the IC) has a size of 750 kB. The difference in object size is given by $\frac{750000}{2259} \approx 332$, so the IC needs to be 332 times larger than the IC prototype to fit the same amount of keys. When the memory limit is larger it approaches the actual size of the objects. Hence, the memory limit for the dynamic instance of the IC will be $150 \times 332 = 49800$ kB $\approx 50$ MB.

For the static instance of the IC there needs to fit 22 objects with an average
size of 750 kB. That yields a cache size of 16.5 MB, but to have some margin if the objects are larger a memory limit of 20 MB should suffice. This adds up to a combined size of 70 MB for the IC. An Insights node has 4 GB of memory with a sample workload of 2.2 GB which gives room for an additional 1.8 GB, so 70 MB seems like a reasonable amount of memory to spare for the cache.

6 Evaluation

The IC prototype is evaluated by using the test script that was developed. In order to determine the efficiency of the IC prototype it is compared to another implementation. The other implementation is a purely dynamic cache developed by changing the IC prototype to a dynamic cache. The two implementations are compared by running the test script and comparing the hit ratios achieved. In order to approximate the possible decrease in response time of deploying the IC, the RTT on a cache hit is compared to the RTT on a cache miss.

6.1 Prototype Performance Compared to a Purely Dynamic Cache

The IC prototype is compared to a purely dynamic cache of the same size. First, by comparing hit ratios when running the test script, and then by comparing the time it takes to search through the cache when it is full.

6.1.1 Comparing Hit Ratios

As mentioned in the background, there is already a dynamic LRU cache that stores the ten latest responses in the main server. It is evident that the IC would outperform the existing cache since it is given more memory. So for comparison a dynamic cache referred to as the dynamic prototype is developed. The memory limit of the dynamic prototype is of the same size as the combined size of the two Redis instances of the IC prototype. It is implemented by removing the static instance from the IC prototype and changing the memory limit in the configuration file of the dynamic instance.

Both implementations show similar results from the test script, see figure 7. The test runs for 20 iterations, measuring the hit ratio each iteration. It is run three times for each implementation and then the hit ratio of each iteration is averaged. The IC prototype performs better from the beginning since there are already some prefetched keys in the cache. But when both caches are filled it is hard to tell any difference in performance. The IC prototype had an average hit ratio of 57.48 % whereas the dynamic prototype had an average hit ratio of 49.07 %.
6.1.2 Comparing Lookup Time

The IC prototype might have a bit slower lookup time since it is implemented using two Redis instances. Therefore, the difference of having to search through two instances instead of one is measured. Both implementations are configured to the real size of the cache, i.e., 20 MB for the static part of the IC prototype, 50 MB for the dynamic part of the IC prototype and 70 MB for the dynamic prototype. The objects that are stored have a size of about 750 kB to reflect the JSON objects in the IC.

The lookup time of both implementations are measured. This is done by filling the caches and measuring the average time of searching through both implementations 100 times. The average time was measured to 118.47 \( \mu s \) for the IC prototype and 45.15 \( \mu s \) for the dynamic prototype. This gives a difference of \( \frac{118.47}{45.15} \approx 2.62 \). So searching through the IC prototype is about 2.62 times slower than if the cache had been implemented in one instance. However, a latency of 118.47 \( \mu s \) is far from the bottleneck in Insights. So having two instances cannot be considered to have a noticeable negative effect on the overall performance.

Figure 7: The test being run for 20 iterations with the two different implementations. The test was run three times for each implementation and then the hit ratio per iteration was averaged.
In conclusion, the IC prototype could start retrieving from the cache from the beginning, giving it an advantage in the start. When both caches had filled up they showed similar performance. The IC prototype had a slower lookup time but was still very fast. The IC prototype has shown promising results compared to the dynamic prototype and will now be compared to not having a cache at all.

6.2 Improvement of Response Time

The potential decrease in response times is estimated from the previous results. This is done by calculating the difference in time between a cache hit and a cache miss. As well as calculating the average response time based on the average hit ratio and comparing it to having no cache.

The difference in time of a cache hit and a cache miss can be calculated from the previous measurements and additional timing of a cache miss. On a normal request to the server there are eight queries being made to a Motion database from the node. These eight queries take a varying amount of time depending on the size of the database. However, for this test the replicated database was used to measure the combined time of making these queries, it took 7.48 sec. As mentioned in the previous subsection the time to search through the IC prototype when the cache was full was 118.47 µs. This time is negligibly small in the context. Therefore the delay is heavily rounded up to 10 ms or 0.01 sec to account for any other delays that could occur in connection with the cache. As mentioned in the background section, the RTT to one of the nodes is about 200 ms or 0.20 sec. If the additional latency introduced by summarizing the statistics in the node is disregarded, then the total RTT on a cache miss is 0.20 + 7.48 = 7.68 sec. The total RTT on a cache hit is 0.20 + 0.01 = 0.21 sec.
This results in a difference of \( \frac{7.68}{0.21} \approx 36.57 \), i.e., a decrease in response time of 36.57 times for a cache hit compared to a cache miss.

![Time taken for a request sent from a client until it is answered](image)

Figure 9: The time taken for a request sent from a client until it is answered, for a cache miss compared to a cache hit.

The average decrease in response time for using the cache can be approximated by including the average hit ratio. The average hit ratio for the IC prototype in the test results in figure 7 was 57.48 \%, the total RTT on a cache miss is 7.68 sec and the total RTT on a cache hit is 0.21 sec. This means that the average RTT would be 0.5748 * 0.21 + 7.68 * (1 - 0.5748) = 3.3862. From this the average decrease in response time is given by \( \frac{7.68}{3.3862} \approx 2.27 \). Making the approximate average decrease in response time 2.27 times.

It can be concluded that the decrease in response time of a cache hit compared to a cache miss was about 36 times. The average decrease in response time with the cache implemented compared to having no cache was about two times.

7 Related Work

Caching provides a solution to the problem of retrieving data quickly from sources that initially have a slow time of retrieval. Since this is a reoccurring problem in many areas of computer science, the same concepts of caching can be applied in many different areas. Many reoccurring concepts about caching are thoroughly explained in a study by Alan Jay Smith, in the context of computer architecture [14]. This section however will focus on solutions to database caching in web applications other than the ones mentioned in the introduction.

Large scale distributed systems such as social networks need advanced caching systems to maintain a responsive end-user experience. The social media platform Facebook has solved this by developing a large scale distributed key-value
store with Memcached as a main building block [3, 15]. The name of the system is Memcache (not to be confused with Memcached). Memcache is used for two primary purposes: query caching and generic caching. Query caching refers to caching database query results to avoid repeating the same queries, and generic caching refers to caching precomputed results from machine learning algorithms. Query caching is an approach which could be used in Insights to store queries instead of computations. The benefit that could be gained from query caching in Insights is that some queries are independent of the time period, and thus there would be a slight decrease in database lookup time even if the exact same request has not been made before since not all queries need to be done each time. However, it would probably be slower than the proposed solution since the summation of query results would have to be repeated even on a cache hit.

Another approach for database caching is suggested by Luo et al. in the context of e-commerce applications [16]. The approach is similar to the SQL Server cache proposed by Larson et al. but does not require an additional caching-tier [2]. Instead the solution uses a commercial relational database management system in the application server as a cache for the back-end database. The advantage of a solution like this one compared to the proposed solution is that a lot more information can be stored by not only storing a key-value pair. In this way data from for example the latest month could be cached and thus all requests of a time period within the latest month could be answered without having to query the database. However, this solution is not very flexible compared to the proposed solution since with the proposed solution the content of the cache is dynamically adapted to the usage of the service.

Although most of the techniques mentioned build on the same principles, they differ a lot in how they are implemented. The proposed solution is well adapted for the Insights system but there are many ways in which the cache could be implemented, and they all have their pros and cons.

8 Conclusions and Future Work

The purpose of this thesis was to establish an efficient solution for caching in the Insights system. This was accomplished by examining the system and how it is used to determine where and how to implement a cache. As well as exploring different methods for caching, such as dynamic and static caching. A prototype was developed independently of the system. The prototype was tested with random values simulating user behavior to find an efficient memory limit and comparing the hit ratio to a purely dynamic cache.

A well-adapted caching solution was proposed, and the tests performed on the prototype showed good potential. The proposed solution was established in section 4 by weighing the benefits and drawbacks of different methods. The solution is a decentralized cache that combines static and dynamic caching to store calculated server responses. In section 5, a prototype was developed based on the solution. The prototype was developed independently and does not store calculated server responses but stores database query responses instead.
implementation choice makes the size of the objects stored in the prototype not correspond to the size of the actual objects that should be stored. Due to the difference in object size and cache size, the tests performed are less reliable. The test that was developed for the prototype, simulated user behaviour in a somewhat random way. The test was used in section 6 to compare the prototype to a purely dynamic cache, and the prototype showed a good hit ratio compared to the dynamic cache. However, the user behavior simulated in the test was purely based on expected usage, and if it would have been changed to include less predefined requests or include more randomness, the results would have been different. An improvement in response time was estimated based on timing the RTT to a node and measuring the lookup time of the cache as well as the time of making queries to the database. Though the time varies a lot between different nodes due to the different amounts of data, it is clear that implementing the cache would contribute to a large improvement in response time in the case of a cache hit.

In order to further develop the cache and make it more efficient, requests coming in to the Insights server could be logged. The logs could then be parsed in order to get a better overview of how the system is being used. For example by calculating the percentage of predefined requests that are being made and tune the cache accordingly. Furthermore, when implementing the cache in the Insights node, it can be tested with different setups in the deployed system and the setup which yields the best hit ratio can be used.

It can be concluded that the prototype and the test do not reflect the behaviour of the system well enough to rely on the results achieved fully. However, the proposed solution is well adapted for the system, and the benefit of implementing it would have a substantial positive effect on response times for cached requests. A stable ground has been laid for a cache in Insights, and the only thing left is to implement it.
References


