Linköpings universitet | Institutionen för datavetenskap
Examensarbete på grundnivå, 16hp | Datateknik
2018 | LIU-IDA/LITH-EX-G--18/028--SE

Utvärdering av metoder för temporär lagring av data i en webbapplikation

An Evaluation of Techniques for Caching Data in a Web application

Tom Almqvist

Handledare : Jonas Wallgren
Examinator : Ola Leifler

Extern handledare : Andreas Larsson

LINKÖPINGS UNIVERSITET

Linköpings universitet
SE–581 83 Linköping
013-28 10 00 , www.liu.se

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Tom Almqvist

Sammanfattning

I databasapplikationer är det viktigt att kunna minska belastningen på en databas i syfte att minska responstiden. Detta kan exempelvis åstadkommas med hjälp av olika metoder för temporär lagring av data, något som studerats i detta arbete. De metoder som utvärderats och jämförts i detta arbete är Redis och memcached. Utvärderingen jämförde Redis och memcached med avseende på minnesanvändning, CPU-användning och tidsåtgång för hämtning av data i respektive cache. Dessa egenskaper beräknades med hjälp av verktygen SYSSTAT och valgrind. Det visade sig i slutändan att den interna fragmenteringen i memcached är dess största nackdel, medan Redis är något långsammare än memcached när det gäller att hämta stora mängder data. Utifrån de resultat som anskaffats var det tänkt att använda den metod som är mest lämpad för SysPartners ändamål, vilket ansågs vara Redis.

Författarens tack

Jag vill tacka Ola Leifler och Jonas Wallgren som hjälpt mig i detta arbete genom att ge konstruktiv kritik på denna rapport som skrivits till följd av detta arbete. Jag fick aldrig någon kritik som jag inte höll med om. Dessutom vill jag tacka Andreas Larsson, som agerade som min externa handledare hos företaget som jag utförde arbetet hos. Han hjälpte mig att komma in i deras system och möjliggöra utvecklingen av modulen som utvecklats till följd av detta arbete.

Innehåll

Sammanfattning iii

Författarens tack iv

Innehåll v

Figurer viii

Tabeller ix

1 Introduktion 1
- 1.1 Motivering . . . 1
- 1.2 Syfte . . . 1
- 1.3 Frågeställningar . . . 2

2 Bakgrund 3
- 2.1 SysPartner Consulting AB . . . 3
- 2.2 Problemformulering . . . 3

3 Teori 4
- 3.1 Nyckel-värdelagring . . . 4
- 3.2 Memcached . . . 4
  - 3.2.1 Server-sida . . . 5
  - 3.2.2 Klientsida . . . 5
  - 3.2.3 Minnesallokering . . . 6
    - 3.2.3.1 Extern fragmentering . . . 6
    - 3.2.3.2 Intern fragmentering . . . 7
    - 3.2.3.3 Minnesallokering i memcached . . . 7
  - 3.2.4 Prestanda . . . 8
- 3.3 Redis . . . 9
  - 3.3.1 Minnesanvändning . . . 9
  - 3.3.2 Datastrukturer . . . 9
    - 3.3.2.1 Strängar . . . 9
    - 3.3.2.2 Listor . . . 10
    - 3.3.2.3 Hashtabeller . . . 10
    - 3.3.2.4 Mängder . . . 10
  - 3.3.3 Partitionering . . . 10
  - 3.3.4 Prestanda . . . 11
  - 3.3.5 Pipelining . . . 11
- 3.4 Andra RAM-minnesdatabaser . . . 11
  - 3.4.1 MICA . . . 11
    - 3.4.1.1 Minnesallokering i Cache Mode . . . 12
    - 3.4.1.2 Minnesallokering i Store Mode . . . 13

3.4.2 MemC3 13
3.4.2.1 Gök-algoritmen 14
3.4.2.2 CLOCK-algoritmen 15
3.4.3 CPHash 16
3.4.3.1 LockHash 16
3.5 Valda mätvärden för beräkning av prestanda 16
3.5.1 Minnesanvändning 16
3.5.2 CPU-användning 17
3.5.3 Tidsåtgång för behandling av förfrågningar 17
3.5.4 `memtier_benchmark` 17
3.6 Tidigare arbeten 17
3.6.1 Utvärdering av prestanda med benchmarking-program 18
3.6.2 Utvärdering av prestanda med PageRank-algoritmen 18
3.6.3 Tillämpningar av memcached 19
3.6.4 Tillämpningar av Redis 19

4 Metod **20**
4.1 Förstudie 20
4.1.1 Tekniska frågeställningar 20
4.1.2 Kravinsamling till webbapplikationen 21
4.2 Design 21
4.2.1 Modulen som kommunicerar med databasen 21
4.2.2 Webbapplikationen 22
4.3 Utvärdering 22
4.3.1 Beräkning av CPU-användning 23
4.3.2 Beräkning av minnesanvändning 23
4.3.3 Genomförande 24

5 Resultat **26**
5.1 Förstudie 26
5.1.1 Tekniska frågeställningar 26
5.1.2 Kravinsamling till webbapplikation 27
5.2 Design 28
5.2.1 Visma-modulen 28
5.3 Utvärdering 29
5.3.1 CPU-användning 29
5.3.2 Minnesanvändning 30
5.3.3 Visma-modulen 31

6 Diskussion **32**
6.1 Resultat 32
6.1.1 CPU-användning 32
6.1.2 Minnesanvändning 32
6.1.3 Tidsåtgång för behandling av objekt 33
6.1.4 Visma-modulen 33
6.1.5 Alternativa metoder 34
6.2 Metod 34
6.2.1 Förstudie 34
6.2.2 Design 35
6.2.3 Utvärdering 35
6.2.3.1 Datamängder som skickats 36
6.2.3.2 Beräkning av CPU-användning 36
6.2.3.3 Beräkning av minnesanvändning 36

6.2.3.4 Beräkning av tidsåtgång för behandling av data 36
6.2.4 Val av mätvärden 37
6.3 Felkällor 37
6.4 Källkritik 37
6.5 Arbetet i ett vidare sammanhang 38

7 Slutsats **39**
7.0.1 Framtida arbete 40

Litteratur **41**

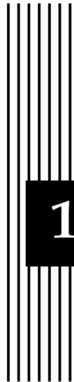
Figurer

3.1 Indelning av slabs och tilldelning av klasser i memcached. 8
3.2 Gök-algoritmen vid insättning av ett värde med nyckel y 15

5.1 Visma-modulens generella struktur. 28
5.2 Memcacheds CPU-användning vid behandling av 100 000 objekt. 29

Tabeller

5.1 Memcacheds minnesanvändning. 30
5.2 Redis minnesanvändning. 30
5.3 Tidsåtgång för hämtning och lagring av data. 31



1 Introduktion

I detta kapitel introduceras arbetet ytterligare: vad är problemet, varför problemet är ett verkligt problem och vad syftet med arbetet var.

1.1 Motivering

Många olika applikationer, framför allt webbapplikationer, använder sig av databaser för att lagra och hämta data. När mängden data i dessa databaser blir stora kan det innebära att databasen belastas till den punkt att det tar lång tid för en applikation att läsa data från databasen. Detta problem kan lösas med hjälp av metoder för temporär lagring av data, vilket bland annat omfattar s.k. *nyckel-värde-databaser* som lagrar data i RAM-minnet istället för i ett sekundärt minne. Dessa databaser kan exempelvis innehålla en delmängd av en annan databas som innehåller stora mängder data, och eftersom dessa data finns i RAM-minnet [24, 14] går det betydligt snabbare att hantera dessa.

I detta arbete studeras två tekniska alternativ för temporär lagring av data: Redis [24] och memcached [14], som båda är databaser vars data lagras i RAM-minnet. SysPartner (företaget där detta arbete utförts) vill ha en webbapplikation som ger en översiktlig bild över anställdas ledigheter och semesterplaner. Dessa data finns i en databas som levereras av Visma [29] som innehåller enorma mängder data och det kan därför ta en väldigt lång stund att läsa data från denna databas. Därför vill SysPartner se en lösning på en modul som kommunicerar med denna databas och även temporärt lagrar dessa med antingen Redis eller memcached.

1.2 Syfte

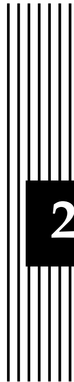
Syftet med detta arbete är att utvärdera Redis och Memcached, främst på grund av det faktum att SysPartner haft ett intresse av dessa och att dessa tekniker har testats och används i praktiken av stora företag. SysPartner vill därför ha en central modul som kommunicerar med deras databas och som också temporärt lagrar data från denna databas. SysPartners krav på lösningen är att den är så effektiv som möjligt med avseende på CPU-användning och minnesanvändning. Det är också viktigt att ta hänsyn till användningsfall för att kunna avgöra om memcached eller Redis är lämpligast i detta fall.

I slutändan kommer den modul som kommunicerar med SysPartners databas att använda sig av Redis eller memcached, beroende på vilken av dem som anses vara bäst med avseende på prestanda och funktionalitet.

1.3 Frågeställningar

För att tydliggöra problemet som ska lösas, vägleds detta arbete med hjälp av följande frågeställningar:

1. Vilken av Redis och memcached är mest lämpad för SysPartner med avseende på funktionalitet och användningsfall?
2. Vilken av Redis och memcached presterar bäst med avseende på CPU-användning, minnesanvändning och tidsåtgång för behandling av läsförfrågningar?



2 Bakgrund

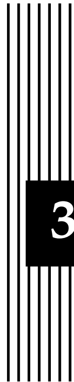
Detta arbete har utförts hos ett konsultföretag vid namn SysPartner Consulting AB i Linköping. Detta kapitel beskriver därför sammanhanget i vilket detta arbete utförts, och varför arbetet var nödvändigt.

2.1 SysPartner Consulting AB

SysPartner Consulting AB är ett konsultföretag som levererar IT-tjänster i form av konsulter och förvaltning av IT-system. Företaget grundades år 2005 och har i skrivande stund ungefär 25 anställda. SysPartner har ett intranät där anställda kan komma åt olika data som rör företaget. I detta fall önskar SysPartner en applikation i intranätet som på ett enkelt och effektivt sätt åskådliggör data som rör anställda. Denna applikation är endast tänkt att användas av anställda med administrativa uppgifter, som exempelvis gruppchefer och ledare.

2.2 Problemformulering

SysPartner önskar en applikation i deras intranät som ger en grafisk överblick över information om anställda. Detta kan bland annat vara vilket konsultuppdrag anställda just nu utför och deras arbetstid. Problemet är att dessa data lagras i en databas som är väldigt långsam. Det kan ta ett flertal sekunder att läsa data från databasen och detta kan innebära att applikationen upplevs som väldigt långsam när en användare vill läsa data. SysPartner vill därför använda sig av metoder för temporär lagring av data för att påskynda läsningen. Eftersom data om anställda sällan uppdateras är det önskvärt att kunna lagra en kopia av data från databasen temporärt i RAM-minnet på en annan server. Detta innebär att alla läsförfrågningar hämtar data från denna cache, istället för att direkt kontakta databasen. Till följd av detta bör läsförfrågningar kunna hanteras betydligt snabbare, eftersom data då finns i RAM-minnet på en server i SysPartners lokaler.



3 Teori

I detta kapitel presenteras teori som var väsentlig för att kunna utföra arbetet. Här beskrivs memcached, Redis, andra typer av cache-tekniker som liknar memcached och Redis och de mätvärden som användes vid utvärderingen av Redis och memcached. Även tidigare arbeten som jämfört Redis och memcached beskrivs här.

3.1 Nyckel-värdelagring

Det finns en mängd olika sätt att implementera databaser på och den typ av databas som studerats i detta arbete är s.k. "*nyckel-värde-databaser*". Denna typ av databas bygger på att data lagras i form av nyckel-värde-par. Dessa data lagras i sin tur i s.k. associativa arrayer [13], som exempelvis i form av en *hash-tabell* (alt. dictionary). I dessa datastrukturer lagras värden med en tillhörande nyckel som identifierar en (eller flera) värden. Dessa nycklar används i sin tur för att hämta och modifiera värden.

Redis [24] och memcached [14] som studerats i detta arbete är av denna typ av databas. De lagrar dessutom sin data i RAM-minnet[1,2], vilket innebär att de är lämpade för att användas för temporär lagring av data. Detta på grund av att det generellt går betydligt fortare att läsa och skriva till RAM-minnet än att skriva/läsa till/från disken, som används i de flesta databaser. Dessutom är det inte nödvändigt att temporärt lagrad data finns kvar när servern i fråga stängs av (därav *temporär lagring* av data).

3.2 Memcached

Memcached [14] är ett distribuerat caching-system som lagrar data i associativa arrayer i RAM-minnet. Dessa typer av databaser brukar kallas för "*In-memory Key-value store*", där *"In-memory"* betyder att data lagras i RAM-minnet och där *"Key-value store"* betyder att data lagras i form av nyckel-värde-par (i associativa arrayer) [2]. memcached används av stora webbsidor som exempelvis Facebook [17] och Reddit [3]. memcached utvecklades till en början för webbsidan *LiveJournal* av grundaren själv, Brad Fitzpatrick [9]. Detta till följd av att

[1] https://redis.io/topics/introduction
[2] https://github.com/memcach ed/memcached/wiki/Overview

Brad Fitzpatrick ville använda oanvänt minne på sina servrar till att temporärt lagra data för att minska belastningen på sin databas.

Memcached bygger på en klient-server-modell, där ena hälften av logiken finns på servern och den andra på klienten[3].

3.2.1 Server-sida

Alla server-processer för memcached [14] är oberoende av varandra. Detta innebär att ingen form av kommunikation sker mellan dem. Redundant lagring av data sker alltså inte; det vill säga, det finns inga kopior av värden som ligger utspridda på andra servrar. Det är sedan upp till klienten att vara informerad om var efterfrågad data ligger någonstans. Klienten vet detta genom att välja vilken server den ska kontakta genom att hasha nyckeln för ett visst givet värde. Denna metod beskrivs i nästa avsnitt (3.2.2).

En memcached-server lyssnar till en början på port 11211, både över TCP (Transport Control Protocol) och UDP (User Datagram Protocol). Dessa inställningar kan givetvis ändras om användaren så vill [14].

Memcached [14] lagrar alltid minst 56 bytes (på 64-bitarsversioner av servern) [8] för ett visst nyckel-värde-par, oavsett hur stort värdet är. Denna data består av metadata för nyckel-värde-paret, som exempelvis nyckelns namn och information som används vid ersättning av värden när cachen blir full. Denna lista fungerar som en stack, där det senast använda värdet alltid finns på toppen. Om cachen skulle bli full skulle detta exempelvis innebära att ett visst värde tas bort beroende på hur långt ner i stacken värdet befinner sig. Detta beror också på bland annat vilken *slab-klass* värdet tillhör (som beskrivs i avsnitt 3.2.3.3). Det värde som ersätts måste tillhöra samma slab-klass som det värde som ska läggas till.

Memcached använder sig av en LRU-algoritm (*Least Recently Used*) för att avgöra vilket data som ska ersättas med ny data som ska läggas till. Detta sker endast när ett visst värde som ska läggas till inte får plats i någon av sidorna för den slab-klass som den tillhör. Om det finns ett värde vars utgångstid har gått ut, tas detta värde bort och ersätts av det nya värde som ska läggas till. Annars letar memcached efter ett värde som använts minst den senaste tiden [27].

Till en början är max-gränsen för antalet samtida uppkopplingar 1024 stycken. Detta kan konfigureras av användaren i en konfigurationsfil för memcached som innehåller de argument som skickas till memcached när memcached startas. memcached är flertrådad för att på ett effektivt sätt kunna behandla flera klienter på samma gång. Det finns alltid en tråd som lyssnar på porten och som därefter skapar "arbetartrådar", där varje tråd har ansvaret för en eller flera uppkopplingar. Gränsen som definierar antalet trådar som blir allokerade är till en början 4 stycken, men den parameter som representerar denna gräns kan ändras i filen *memcached.conf*. memcacheds dokumentation rekommenderar dock att låta denna gräns vara, såvida servern inte är, enligt dokumentationen, "väldigt belastad".

3.2.2 Klientsida

Varje klient har en lista med IP-adresser till maskiner som exekverar en memcached-server [14]. Denna lista måste anges manuellt i konfigurationsfilen för memcached, och dokumentationen[4] rekommenderar att alla IP-adresser anges i samma ordning för varje memcached-klient. Detta på grund av det faktum att vissa memcached-klienter sorterar listan medan andra inte gör det. Det finns flera olika memcached-klienter som är implementerade på olika sätt beroende på vilket språk klienten är skriven i. Varje typ av klient erbjuder samma mängd funktioner, dock.

Varje gång en klient avser att hämta eller skriva data med en viss nyckel, applicerar klienten en hash-funktion på denna nyckel [14]. Sedan används detta hash-värde som ett index

[3] https://github.com/memcached/memcached/wiki/Overview#logic-half-in-client-half-in-server
[4] https://github.com/memcached/memcached/wiki/ConfiguringClient

i listan över memcached-servrar för att avgöra var dessa data befinner sig. Hash-funktionen kan vara något så simpelt som:

$$index = mod(key, length(serverList))$$

Där `serverList` är listan med IP-adresser för servrar som finns. `index` används sedan som ett index i listan med servrar för att avgöra vilken server klienten ska kontakta. `length` beräknar antalet element i server-listan. På detta sätt vet klienten var den kan hitta efterfrågad data. Problem uppstår dock om en viss server inte svarar till följd av ett allvarligt fel eller liknande. Om servern inte svarar klienten på en viss förfrågan, tolkar klienten detta helt enkelt som en ”cache miss” och fortsätter sedan exekvera logik utefter detta utfall. Det sker inga förändringar i server-listan och det sker heller inga omhashningar [14].

3.2.3 Minnesallokering

Detta avsnitt beskriver metoden som memcached använder sig av för att allokera minne på ett sådant sätt att den *externa minnesfragmenteringen* [22] minskar. Tidigare använde sig memcached endast av `malloc` [11] för allokering av data, men som sedan visade sig vara ett ineffektivt sätt att utnyttja minnet på [9]. Detta berodde främst på att extern fragmentering uppstod, vilket beskrivs nedan. Den nuvarande metoden för allokering av data i memcached utvecklades med syftet att motverka denna externa fragmentering [14]. Denna metod ger dock upphov till *intern fragmentering* [22], som också beskrivs nedan.

3.2.3.1 Extern fragmentering

Det finns två olika typer av fragmenteringar som kan uppstå i minnet [22]: *extern fragmentering* och *intern fragmentering*. Extern fragmentering definieras som en mängd lediga block i minnet vars storlekar inte är tillräckligt stora för att uppfylla en viss begäran av minne, även om de lediga blockens sammanlagda storlek är tillräckligt stor. Denna typ av fragmentering uppstår när block av olika storlekar allokeras och avallokeras under en viss tidsperiod. På detta sätt bildas lediga block av olika storlekar, där de lediga blocken inte ligger angränsande till varandra.

B. Randell [22] kom fram till att metoden för allokering av minne har betydelse för extern fragmentering. B. Randell undersökte minnesfragmentering (både intern och extern fragmentering) med tre olika metoder för allokering av minne:

1. `MIN`: Minne allokeras från det minsta lediga block som är tillräckligt stort för att kunna hålla en viss begäran av minne.
2. `RANDOM`: Minne allokeras i ett slumpmässigt valt ledigt block som är tillräckligt stort för att hålla en viss begäran av minne.
3. `RELOC`: Minnesblock som ligger angränsande till ett block som avallokeras förflyttas på ett sådant sätt att de täcker det lediga block som uppstått.

B. Randell använde en lista där varje element innehöll ett heltal som representerade storleken på ett visst (imaginärt) minnesblock och om minnesblocket var ledigt eller ej. Vid varje förfrågan om en viss mängd minne användes denna lista för att ”allokera” minnet enligt någon av de ovannämnda algoritmer. Intilliggande element i listan ansågs motsvara intilligande minnesblock. Han nämner dock aldrig hur stor denna lista är eller hur mycket minne listan representerar totalt.

B. Randell kom fram till att algoritmen `RELOC` bidrog till minst extern fragmentering. Han menar att detta talar för att lediga block bör ligga angränsande till varandra, eftersom detta upprätthåller ett enda ledigt block hela tiden.

3.2.3.2 Intern fragmentering

Intern fragmentering definieras som en situation där ledigt minne allokeras i multiplar av något heltal Q [22]. Detta innebär att vid en förfrågan om minne allokeras ett minnesblock vars storlek motsvarar den minsta multipeln av Q som är tillräckligt stor för att kunna hålla den mängd minne som önskas. Detta kan innebära att en viss förfrågan blir allokerad mer minne än vad den önskat sig, vilket i sin tur innebär att överskottet av minnet som allokerats blir bortkastad (såvida inte överskottet av någon anledning utnyttjas senare av den applikation som blivit allokerad minnet).

Genom att allokera minne på detta sätt undviks extern fragmentering, eftersom de block som allokeras är alltid lika stora. B. Randell [22] kom dock fram till att intern fragmentering bidrog till mer bortkastat minne ju större Q är, medan extern fragmentering bidrog till mindre bortkastat minne för $Q \geqslant\sim 256$B.

3.2.3.3 Minnesallokering i memcached

Memcached delar in cache-minnet i ett antal lika stora delar som kallas för *slabs* [14]. Om inget annat anges vid uppstart av memcached är storleken av varje slab *alltid* 1 MB. Dessa slabs kan sedan delas in i mindre segment vars storlekar definieras av den *slab-klass* som respektive slab tillhör. Dessa segment används sedan för att lagra värden. En slab-klass definierar alltså en segmentstorlek för en slab. Vid uppstart av memcached bestäms hur många klasser som finns och vilken segmentstorlek varje klass representerar. Storleken på segment beräknas med hjälp av geometrisk funktion [5]:

$$chunksize(class) = \left\lceil basesize * factor^{class-1} \right\rceil \tag{3.1}$$

Där `class` är ett positivt heltal, `factor` är ett reellt tal sådant att `factor` $\geqslant 1$, `basesize` är ett positivt heltal och `chunksize` är segmentstorleken för en viss klass. `class` beskriver klassnumret som identifierar en klass (en segmentstorlek), `factor` beskriver med hur mycket mer segmentstorleken ska öka jämfört med föregående klass och `basesize` beskriver den minsta segmentstorleken. `basesize` har ett värde på 48 bytes om inget annat anges och det samma gäller för `factor` som annars har ett värde på 1.25. Dessa variabler kan ändras vid uppstart av memcached genom att skicka värden till motsvarande parametrar.

Memcached skapar en mängd klasser som identifieras av ett klassnummer c enligt följande formel:

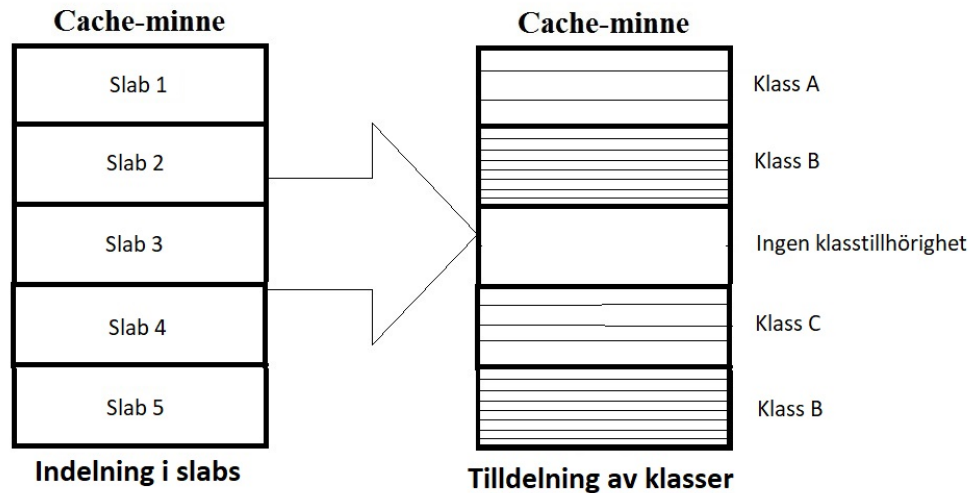
$$classes = \{c : c \in \mathbb{Z}, c \geqslant 1, basesize \leqslant chunksize(c) \leqslant slabsize\} \tag{3.2}$$

där `slabsize` beskriver storleken för varje slab (som ju är 1 MB om inget annat anges) och `chunksize` beräknas enligt ekvation 3.1.

Själva tilldelningen av klasser till slabbar sker endast vid behov när ett värde ska läggas till i minnet. Detta sker på följande vis [14]:

1. Klasstillhörigheten för värdet som ska läggas till avgörs genom att välja den minsta segmentstorleken (klassen) som kan hålla värdet.
2. Om det inte finns en slab som tillhör den klass som valts, tilldelas en ledig slab denna klass. Om detta lyckas, allokeras en av segmenten i slabben för att hålla värdet.
3. Om det inte finns några lediga slabs kvar, försöker memcached ta bort ett värde från någon slab som tillhör samma klass som det värde som ska läggas till. Om ingen sådan slab finns, misslyckas operationen och värdet läggs inte till.

Vid borttagning av data i memcached avallokeras inte minne. Istället markeras segment i slabs som antingen lediga eller ej. Segment som är markerade som borttagna kan inte läsas; memcached hanterar dessa data som om de inte existerade. Däremot är det möjligt att lägga till ett värde i dessa segment, eftersom det minne som funnits där inte längre används.



Figur 3.1: Indelning av slabs och tilldelning av klasser i memcached.

Om segmenten är exempelvis 80 bytes stora i en slab-klass och 120 bytes stora i en annan, lagras värden som är mellan 1 och 80 bytes stora i en slab av den förstnämnda klassen; värden som är mellan 81 och 120 bytes stora lagras i en slab som är av den sistnämnda klassen. Detta innebär alltså, exempelvis, att ett värde som är 32 bytes stort lagras i en slab som tillhör den förstnämnda slab-klassen. Därför allokeras 80 bytes för detta värde. Detta innebär därför en intern fragmentering på 80-32=48 bytes. Värt att notera är att storleken av det data-värde som ska lagras utgörs av summan av storleken för dess nyckel, meta-data för värdet (exempelvis *expiration time*) och själva värdets data i sig. [27] Notera även i figur 3.1 att slab 2 och 5 tillhör samma slab-klass (klass B). Alltså delas de i lika stora segment. Dessutom tillhör inte slab 3 någon klass, vilket innebär att den är ”ledig” och skulle kunna tilldelas en klass i framtiden.

3.2.4 Prestanda

Memcached utvecklades i syfte att åstadkomma en hög exekveringshastighet. [9]. Därför har nästan alla funktioner i memcached en tidskomplexitet av $\mathcal{O}(1)$ [21]. Detta innebär att tidskomplexiteten är konstant, vilket i sin tur betyder att memcached kan utföra operationer med en konstant hastighet som är oberoende av datamängdernas storlek. Enligt Brad Fitzpatrick [9] innebär detta att memcacheds användning av processorn inte beror på faktorer som exempelvis mängden data som överförs vid behandling av förfrågningar eller antalet klienter som är uppkopplade till memcached-servern.

Memcached är också "*lock-less*"[9], vilket innebär att ingen klient som önskar att använda sig av memcached kan blockera någon annan klient som använder sig av cachen.

Memcached använder sig av TCP som transportprotokoll över nätverket. För varje TCP-uppkoppling används minne på servern. Antalet klienter som kan koppla upp sig mot en memcached-server beror alltså på hur mycket ledigt minne som finns på servern i fråga. Dessutom är det värt att notera att om *persistent connections* för uppkopplingarna är avaktiverade innebär detta att ett s.k. TCP-handskak (*three-way handshake*) sker för varje förfrågan som skickas till memcached-servern [21].

3.3 Redis

Redis är, likt memcached, en databas som lagrar data i RAM-minnet i form av nyckel-värde-par [24]. Den största skillnaden mellan memcached och Redis är det faktum att Redis erbjuder stöd för olika datastrukturer där data kan lagras, medan memcached endast använder sig av associativa arrayer [14]. Redis erbjuder bland annat stöd för följande datastrukturer: *strängar, hashtabeller, listor, mängder* och *sorterade mängder* [10].

3.3.1 Minnesanvändning

Redis erbjuder funktioner för att optimera storleken för olika typer av data. [15] Olika datastrukturer kan bli kodade på ett sådant sätt att de använder mindre minne. Detta sker för datastrukturer som har ett visst antal element som är mindre än ett givet tal, samt vars elements storlekar är mindre än en viss storlek. Dessa gränser kan ändras av användaren i en konfigurationsfil för Redis som läses av varje gång Redis startas. Värden som är större än den givna gränsen för storleken på element kodas på ett standardsätt, med andra ord optimeras inte minnesanvändningen för det värdet.

Redis använder sig av `jemalloc` [7] för allokering av minne [24]. Dokumentationen för Redis förklarar inte varför just denna allokeringsmetod används, men `jemalloc` bidrar, enligt utvecklaren, till låg fragmentering i minnet.

Redis allokerar så mycket minne som den är konfigurerad till att göra. Denna inställning kan ändras i konfigurationsfilen för Redis genom att ändra motsvarande parameter. Denna fil innehåller data som Redis läser vid uppstart. Om ingen gräns har definierats, allokerar Redis så mycket minne som är nödvändigt. Om ett värde ska läggas till och det av någon anledning inte finns plats för värdet, försöker Redis göra plats åt detta värde. Redis ersätter då ett värde enligt en algoritm som väljer vilket värde som ska ersättas med det nya. Till skillnad från memcached har Redis inte bara stöd för en LRU-algoritm för detta ändamål, utan också en LFU-algoritm (*Least Frequently Used*) [23]. Detta gäller dock endast version 4.0 eller senare av Redis. Värt att notera är att LRU-algoritmen enligt dokumentationen endast är en approximation av en ”äkta” LRU-algoritm. Detta är på grund av det faktum att det kostar mer minne att implementera en komplett LRU-algoritm. För att spara på minne har utvecklarna därför valt att implementera enbart en approximation av en LRU-algoritm [28].

3.3.2 Datastrukturer

Till skillnad från memcached stödjer Redis fler datastrukturer än bara hash-tabeller. All data i Redis lagras som nyckel-värde-par, där nyckeln är en godtycklig sträng och där värdet består av en datastruktur som erbjuds av Redis. Dessa datastrukturer beskrivs nedan [24].

3.3.2.1 Strängar

Redis stödjer binärsäkra strängar, vilket innebär att strängarna kan innehålla vilka tecken som helst [24]. Exempelvis kan tecken som vanligtvis indikerar på slutet av en sträng (som brukar vara ett värde av 0x00 i de flesta språken) användas på en godtycklig plats i strängen utan att strängen i sig påverkas. Till följd av detta är alla strängar i Redis av en fix längd som är känd. På detta sätt är det möjligt för Redis att veta var en sträng slutar.

Värden av denna typ hanteras med de enkla kommandona *set* och *get* [24]. Det faktum att strängar i Redis är binärsäkra innebär att de kan lagra text och även binär data som exempelvis jpeg-filer. Värt att notera är även att alla nycklar i Redis är binärsäkra strängar. Den maximala storleken för ett värde av typen sträng är i skrivande stund 512 MB.

3.3.2.2 Listor

Listor i Redis är implementerade som länkade listor och är sorterade efter vilken ordning element lagts till i listan [24]. Element kan endast tas bort från och läggas till i huvudet (*head*) och svansen (*tail*). Detta innebär att Redis kan lägga till element i en lista under konstant tid ($\mathcal{O}(1)$). Däremot har indexering av element i en lista en tidskomplexitet av $\mathcal{O}(n)$.

Enligt dokumentationen för Redis [24] är det populärt att använda dessa listor vid kommunikation mellan processer. Ett typiskt exempel är *"worker-producer"*-problemet, där första processen fyller listan medan den andra bearbetar och tar bort de element som den ena processen lagt till. Om listan är tom kan den andra processen vänta tills den första lagt till element i listan. Detta kan implementeras med hjälp av s.k. *polling* där processen upprepade gånger kollar om det finns något i listan. Detta bidrar till onödiga klockcykler för processorn och kommandon som skickas till Redis. Därför har Redis implementerat *blockerande kommandon* för att kunna undvika polling. Detta innebär att processen som önskar läsa från en lista som är tom blir blockerad tills dess att ett nytt element hamnat i listan. När ett element lagts till i listan returneras detta till processen i fråga och blockering upphävs.

3.3.2.3 Hashtabeller

Vid användning av hashtabeller kan värdena i hashtabellen utgöras av en enkel sträng (som beskrevs tidigare), eller också ett objekt som består av flera fält, vars värden består av strängar [24]. Hashtabellen är avsedd att användas för att kunna lagra objekt som innehåller flera fält. Detta går att åstadkomma utan användning av hashtabeller i Redis, genom att istället lagra enkla JSON-kodade strängar (i detta arbete lagrades data på detta sätt).

3.3.2.4 Mängder

Mängder i Redis finns både som osorterade och sorterade [24]. Denna datastruktur innehåller en samling av strängar. När en osorterad mängd hämtas från Redis är elementen i denna mängd oordnad, vilken innebär att Redis inte tar hänsyn till någon ordning av elementen när de returneras. När ett enstaka element ska hämtas från mängden hämtar Redis ett slumpmässigt sådant. Alla element i en mängd i Redis måste vara unika för den mängden, det finns alltså inga duplicerade element i en mängd.

Mängdoperationer såsom snittet, unionen och differensen mellan mängder kan appliceras på mängder i Redis [24]. På grund av detta menar Redis dokumentation[5] att mängder kan beskriva relationen mellan arbiträra objekt i ett system.

Det finns som tidigare nämnt även sorterade mängder [24]. Till skillnad från osorterade mängder upprätthåller Redis en viss ordning mellan elementen i en sorterad mängd (därav namnet *sorterad* mängd). Varje element i en sorterad mängd är associerad med ett slumpmässigt valt flyttal som används för att ordna elementen i ökande ordning. Detta flyttal kallas för *"the score"*.

3.3.3 Partitionering

Likt memcached kan Redis lagringsutrymme spridas ut över flera servrar. [20]. Detta innebär alltså en cache vars storlek består av alla servrars sammanlagda lediga minne. En Redis-klient kan därefter ta reda på var ett visst värde befinner sig genom att använda sig av en tillhörande nyckel till värdet i fråga. Det kan exempelvis ske genom hashning, som används på just detta sätt i memcached, eller också med hjälp av s.k. *"range partitioning"* [20]. Detta innebär att delmängder av den totala mängden nycklar beskriver var ett visst tillhörande data befinner sig. Exempelvis kan värden vars nycklar är mellan 1 - 1000 lagras på en server och värden vars nycklar är mellan 1001 - 2000 lagras på en annan. Enligt Redis dokumentation [20] är det

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

däremot bättre att använda sig av hashning istället för range partitioning, eftersom en tabell måste lagras i minnet vid det senare alternativet. Denna tabell definierar de olika intervall för nycklar och vilken server värden vars nycklar faller inom ett visst intervall lagras.

3.3.4 Prestanda

Precis som memcached [21], strävar Redis efter algoritmer som har en tidskomplexitet av $\mathcal{O}(1)$, för åtminstone get- och set-funktioner. Men eftersom Redis även erbjuder andra typer av datastrukturer än endast hashtabeller, är det inte möjligt att åstadkomma denna tidskomplexitet för alla datastrukturer. Exempelvis har algoritmer för att hämta ett visst element i en länkad lista en tidskomplexitet av $\mathcal{O}(n)$. Därför rekommenderar Redis dokumentation [15] att utvecklare använder sig av hashtabeller så mycket som möjligt. Enligt Redis dokumentation [15] är hashtabellerna i Redis också mer effektiva än i memcached.

3.3.5 Pipelining

Redis [24] erbjuder ett sätt för att minimera antalet paket som den skickar vid förfrågningar om data och liknande. Redis kan konfigureras på ett sådant sätt att den skickar fler förfrågningar i ett och samma paket. Denna metod kallas för pipelining[6]. Varje gång Redis skickar förfrågningar behandlar TCP dessa data, vilket leder till att TCP utgör en flaskhals i hur snabbt Redis kan skicka och ta emot data. Därför är det möjligt att konfigurera Redis på ett sådant sätt att Redis först lagrar förfrågningar i en buffert, och sedan skickar dessa i ett och samma paket. På detta sätt undviks overhead för varje förfrågan, och istället skickas dessa förfrågningar i ett enda TCP-paket.

3.4 Andra RAM-minnesdatabaser

Här beskrivs tre andra nyckel-värde-databaser som lagrar sina data i RAM-minnet i form av nyckel-värde-par. En av dessa, MemC3 [8], bygger på memcached, men utvecklades i syfte att optimera den befintliga versionen av memcached. Det som är gemensamt med dessa databaser, memcached och Redis är det faktum att de lagrar sina data i RAM-minnet i form av nyckel-värde-par. De lagrar även sina data i RAM-minnet. De databaser som beskrivs här har också jämförts med memcached och utges för att bland annat kunna utnyttja flerkärniga processorer bättre än bland annat memcached. Detta möjliggörs i MICA [12] genom att minnet delas upp i ett antal partitioner som är direkt proportionerlig mot antalet kärnor i processorn på vilken processen exekverar (som beskrivs nedan). MemC3 utvecklades med målet att öka samtidigheten i memcached och att minimera antalet mutex-lås i applikationen.

3.4.1 MICA

MICA [12] är en RAM-minnesdatabas som utnyttjar flerkärniga processorer för att åstadkomma en hög prestanda med avseende på *throughput* (operationer per sekund) och minnesanvändning. I artikeln konstaterade författarna att MICA fungerar 4 - 13.5 gånger snabbare än dagens moderna RAM-minnesdatabaser. De databaser som jämfördes med MICA var bland annat memcached, MemC3, Masstree och RAMcloud. MICA kan behandla ca 76 miljoner operationer per sekund, vilket enligt författarna är minst 4 gånger fler än RAMcloud.

MICA partionerar minnet i ett antal delar i direkt proportion till antalet kärnor i processorn. Detta innebär att varje kärna har ansvar för sin egna partition av minnet när det gäller läs- och skriv-förfrågningar. Denna partitionering implementeras via hashning, på ett sätt som liknar memcacheds metod för att avgöra vilken server ett visst objekt befinner sig i. MICA beräknar ett 64-bitars värde från nyckeln som är associerad med ett visst data. Hashvärdet används sedan för att avgöra vilken kärna en förfrågan om data ska omdirigeras till.

[6] https://redis.io/topics/pipelining/

Denna idé - att dela in minnet i partitioner för varje kärna - ligger till grund för de olika typer av moder som MICA kan exekvera i:

1. `CREW` (Concurrent Read Exclusive Write): tillåter alla kärnor att läsa från vilken partition som helst, men kärnorna kan endast skriva till den partition som de själva ansvarar för.
2. `EREW` (Exclusive Read Exclusive Write): kärnor kan endast läsa från och skriva till sina egna respektive partitioner.
3. `CRCW` (Concurrent Read Concurrent Write): kärnor kan läsa från och skriva till vilken partition som helst.

Enligt författarna eliminerar EREW-moden all form av synkroniserings- och inter-kärna-kommunikation, vilket betyder att overhead för dessa kommunikationer försvinner. Däremot måste dessa typer av kommunikationer införas i CREW-moden, eftersom det nu är möjligt för kärnor att läsa från vilken partition som helst. Det innebär att en kärna skulle kunna skriva till en plats i minnet som en annan kärna samtidigt försöker läsa. Det innebär också att kärnorna måste kommunicera med varandra för att upprätthålla "cachekoherens" mellan cache-minnena i kärnorna, eftersom samma data kan finnas i flera olika cacheminnen i processorn.

Den tredje moden, CRCW, erbjuds främst för att kunna modellera system där minnet inte partitioneras. Författarna menar att denna mode erbjuds med syftet att kunna visa hur mycket bättre prestanda CREW och EREW har än CRCW.

Enligt författarna ska EREW alltid användas eftersom denna mode är mest effektiv, på grund av det faktum att ingen kommunikation mellan kärnorna krävs. Om det dock skulle råda stor obalans i utspridningen av data mellan partitionerna är det enligt författarna en god idé att använda sig av CREW. Då är det möjligt för kärnorna att läsa data från andra partitioner än dess egna.

Vad gäller datastruktur för allokering av data kan MICA köras i två moder: *cache mode* och *store mode*, där varje mod stödjer annorlunda semantik och allokeringsmetoder. Dessa beskrivs kortfattat nedan.

3.4.1.1 Minnesallokering i Cache Mode

Cache-moden är avsedd att användas när MICA används som en cache [12]. Denna mode implementerar en cirkulär buffert tillsammans med ett hash-index för att snabbt kunna hämta och skriva data till bufferten.

Varje partition i minnet håller en cirkulär buffert där värden lagras [12]. När värden ska lagras, läggs dessa till i slutet av bufferten (tail). Det innebär alltså att det äldsta värdet alltid finns i början av bufferten (head). När bufferten är slut, tas värdet i början av bufferten bort. Detta innebär att MICA tar bort värden enligt en FIFO-princip (First In First Out). Borttagning och inläggning av data sker med en tidskomplexitet av $O(1)$, alltså lika snabbt som att hämta data från bufferten via hash-indexet. Dessutom undviks extern fragmentering genom att borttagning av data endast sker vid huvudet av bufferten. På detta sätt ligger alla lediga block angränsande till varandra, istället för att vara utspridda.

Hash-indexet består av ett antal hinkar och enligt författarna fungerar detta hash-index som en set-associativ cache som vanligen implementeras i processorer [12]. Varje gång ett värde ska läsas eller skrivas används ett 64-bitars hashvärde som beräknas utifrån nyckeln som är associerad med ett visst värde (samma hashvärde som används för att avgöra vilken partition värdet ska lagras i). MICA använder därefter en del av detta hashvärde för att avgöra i vilken hink värdet ska hamna. När en sådan hink hittats, lagras information om var i den cirkulära bufferten som värdet befinner sig (såvida hinken inte är full). Om hinken är full tas det äldsta värdet i hinken bort. Enligt författarna är detta effektivt sätt för att hantera hash-kollisioner jämfört med andra metoder som exempelvis kedjning av element.

Författarna [12] menar att denna metod för lagring av data utnyttjar semantik för cache-funktioner för att bidra med effektiva algoritmer för läsning och skrivning av data. Det finns dock inga timers för hur länge objekt lever. Den enda gång som värden blir borttagna är när bufferten blir full, då tas det värde som ligger i början av bufferten bort, utan att klienten vet om det. Detta är dock en lämplig lösning enligt författarna eftersom en cache används för temporär lagring av data. Detta skulle exempelvis inte vara en lämplig lösning för en databas där värden ska lagras i minnet tills vidare (store mode används för detta ändamål).

3.4.1.2 Minnesallokering i Store Mode

Store-moden används vid permanentlagring av data i RAM-minnet. Det innebär alltså att data inte tas bort utan klientens vetskap. Data kan endast ta bort om klienten skickar en förfrågan om att göra detta [12].

Minnesallokeringen i denna mode liknar memcacheds slab-allokerare. MICA definierar en mängd storleksklasser med en start på 8 bytes och som sedan ökar med 8 bytes för varje klass [12]. Hur många storleksklasser som definieras beror på antalet storlekar som stöds, vilket aldrig nämns explicit i artikeln. För varje storleksklass lagras en lista med pekare till block i minnet som är minst lika stora som storleksklassen och som är lediga. När ett värde ska lagras, väljer MICA att allokera så mycket minne som den minsta storleksklassen som är tillräckligt stor för att lagra värdet.

Skillnaden mellan memcached och MICA i detta avseende är det faktum att MICA använder en partition för *alla* storleksklasser. Enligt författarna [12] är det effektivare att använda en partition för alla storleksklasser, istället för att endast allokera en partition för en viss storleksklass. Detta eftersom en partition kan innehålla väldigt få värden, vilket innebär att den interna fragmenteringen blir stor för denna partition. Författarna menar att detta är dåligt utnyttjande av minne, eftersom andra värden av en annan storleksklass hade kunna göra nytta av det överskott som finns i partitionen.

För att komma åt data används även här ett hash-index med hinkar [12]. I varje hink finns information om värden vars nycklar hashar till just den hinken. Denna information används för att kunna hitta värdet i minnet. När dessa hinkar blir överfulla, används en *spare bucket* som är kopplad till den överfulla hinken. När ett värde ska lagras, används det 64-bitar stora hashvärde som beräknats utifrån värdets nyckel. Om hinken som värdet hashats till är full, letar MICA efter utrymme i en reservhink för den hinken. Om även denna är full, avböjs klientens förfrågan om att lägga till ett värde.

Skillnaden mellan denna mode och cache-mode är dels att minnesallokeringen är annorlunda, men också att borttagning av data endast kan ske på uppdrag av en klient [12]. I cache-mode kan värden tas bort utan klientens vetskap, vilket, enligt författarna [12], är rimligt i de fall där MICA används för *temporär* lagring av data. Däremot om data ska finnas kvar i minnet tills vidare, kan data inte försvinna utan klientens vetskap. Därför är det inte lämpligt att använda sig av en cirkulär buffert på samma sätt som i cache-mode för detta ändamål.

3.4.2 MemC3

MemC3 (memcached with CLOCK and Concurrent Cuckoo hashing) [8] är en nyckel-värde-databas som är baserad på memcached [14]. Denna applikation utvecklades med syftet att optimera memcached med avseende på samtidighet och minnesanvändning. Författarna motiverade denna förbättring med att memcached inte är särskilt skalbar på flerkärniga processorer, eftersom den använder väldigt många lås i synkroniseringssyfte. Bland annat använder memcached ett globalt lås för den datastruktur som används för LRU-ersättning av data, ett lås för varje nyckel och ett globalt lås för hash-tabellen. Dessa lås måste hämtas av varje tråd som avser att läsa eller skriva till hash-tabellen. Det innebär alltså att läs- och

skriv-kommandon är serialiserade i memcached, och idén med MemC3 är att minska denna serialisering med syftet att öka samtidigheten i memcached.

Resultatet av denna applikation visade att MemC3 kunde behandla tre gånger så många operationer per sekund som memcached under ett antal test där olika mängder data skickades till MemC3 [8]. MemC3 använder dessutom 20 färre bytes per värde i cachen än memcached. Detta till följd av att MemC3 använder sig av CLOCK-algoritmen (se 3.4.2.2) vid ersättning av data, där endast en bit per nyckel-värde-par behövs. Jämför detta med memcached där bland annat två pekare behövs för varje nyckel-värde-par i LRU-listan, vars storlekar beror på arkitekturen på vilken memcached exekverar. Nedan följer de metoder som användes för att åstadkomma de mål som utvecklarna av MemC3 ställde.

3.4.2.1 Gök-algoritmen

Målet med MemC3 var bland annat att tillåta trådar att läsa från hash-tabellen samtidigt, medan operationer för att skriva till den förblev serialiserade (atomiska). För att uppnå detta mål utvecklades och implementerades en optimerad version av den s.k. gök-algoritmen [19]. Denna algoritm och tillhörande datastruktur ersatte sedan memcacheds befintliga algoritm för hash-kollisioner och memcacheds hash-tabell.

Grundidén med gök-algoritmen är att det finns 2 hashvärden för varje nyckel [19]. I MemC3 är dessa hashvärden kopplade till hinkar som kan innehålla 4 värden vardera. Varje gång ett värde ska läggas till, beräknas 2 hashvärden utifrån värdets nyckel. Dessa hashvärden pekar på 2 olika hinkar. Sedan läggs det nya värdet in i en av de 2 hinkar som har ledigt utrymme. Finns inget ledigt utrymme måste ett värde ur någon av de 2 hinkarna ersättas med värdet som ska läggas till. Det värde som ersätts måste i sin tur förflyttas till en annan godtycklig hink. Algoritmen väljer slumpmässigt det värde som ska ersättas. Sedan förflyttas detta värde till en godtycklig hink, som kan vara full. Om hinken är full, måste ännu ett värde förflyttas. Detta pågår tills dess att ett maximalt antal förflyttningar skett (i MemC3 är denna gräns 500). Varje värde i hinkarna är en pekare till det faktiska nyckel-värde-paret. Figur 3.2 illustrerar denna algoritm tillsammans med ett exempel.

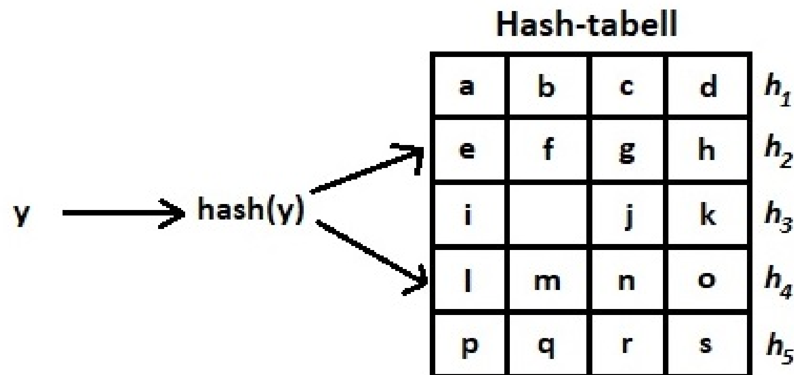
Figur 3.2 används för att illustrera ett exempel på gök-algoritmen [19]. I figuren ovan finns 5 olika hinkar med värden. Ett värde med nyckeln y ska läggas till, dock blir detta problematiskt eftersom hash-värdet för nyckeln pekar på 2 hinkar som är fulla. Detta innebär att något befintligt värde i någon av dessa 2 hinkar måste förflyttas. Hinkarna beskrivs som 5 mängder: $h_1 = \{a, b, c, d\}$, $h_2 = \{e, f, g, h\}$, $h_3 = \{i, null, j, k\}$, $h_4 = \{l, m, n, o\}$ och $h_5 = \{p, q, r, s\}$. Indexering i hinkarna kommer även att beskrivas som $h_{i,j}$ där i är hinkens nummer och j är kolumnens nummer i hinken, räknat från vänster i hinken och med start på 1.

Algoritmen [19] väljer att förflytta f till en slumpmässigt vald plats, men själva förflyttningen sker inte förrän algoritmen är säker på att det finns en sekvens omflyttningar sådan att y får plats i hash-tabellen. I detta fall lyckas algoritmen finna en sekvens förflyttningar som gör att y får plats:

$$f \Longrightarrow j \Longrightarrow m \Longrightarrow h_{3,2}$$

Sekvensen beskriver hur de olika nycklarna ersätter andra nycklars platser. I detta fall hamnar f på den ursprungliga platsen för j; j hamnar på den ursprungliga platsen för m och m hamnar i en tom plats som beskrivs av $h_{3,2}$. Eftersom denna sekvens hittades väljer alltså algoritmen att placera y på den ursprungliga platsen för f, som beskrivs av $h_{2,2}$, medan övriga nycklar i sekvensen förflyttas enligt beskrivningen för ovanstående sekvens. Därefter ser berörda hinkar ut på följande vis: $h_2 = \{e, y, g, h\}$, $h_3 = \{i, m, f, k\}$ och $h_4 = \{l, j, n, o\}$.

Utvecklarna av MemC3 [8] menar att både den vanliga gök-algoritmen [19] som beskrevs ovan och memcacheds algoritm för uppslagning av nycklar i hash-tabellen är ineffektiva. Detta på grund av antalet pekar-avreferenser som sker. I gök-algoritmen måste exempelvis varje element i 2 hinkar undersökas för att hitta ett visst objekt med en viss nyckel, vilket innebär att en pekare måste avrefereras för varje element. Det samma gäller för memcached,



Figur 3.2: Gök-algoritmen vid insättning av ett värde med nyckel y

där en länkad lista måste traverseras för att hitta ett visst värde [8]. Till följd av detta lagrar MemC3 en *tagg*, som är en 1 byte stor, för varje element tillsammans med pekaren som pekar på ett visst nyckel-värde-par. Denna tag beräknas genom att hasha nyckeln, sedan används denna tagg för att hitta rätt nyckel-värde-par. På detta sätt minimeras antalet pekare som avrefereras. Detta är en mer cache-medveten lösning (från processorns perspektiv) enligt författarna [8] eftersom färre läsningar från minnet sker och hinkarna får plats i en cache-linje i processorn. Detta innebär att funktionen för att hitta ett värde använder läser av i genomsnitt 2 cache-linjer.

3.4.2.2 CLOCK-algoritmen

Ett annat mål med MemC3 var att minska den mängd minne som allokeras för metadata i LRU-algoritmen. Memcached använder sig nämligen av 18 bytes [8] för varje nyckel i cachen, medan MemC3 använder sig endast av 1 bit per nyckel. Detta på grund av det faktum att MemC3 ersätter memcacheds LRU-algoritm med en annan som är baserad på CLOCK-algoritmen [6, 8]. Information om nycklar som använts lagras i en cirkulär buffert, där varje enskild bit representerar ett värde som är lagrat i cachen. Om biten är nollställd (0), innebär det att det tillhörande värdet inte använts på sistone. En bit som inte är nollställd (1) indikerar att det tillhörande värdet har använts nyligen.

När ett värde läses in eller skrivs till, sätts motsvarande bit i den cirkulära bufferten till 1. Detta betyder därefter att värdet har nyligen använts. [6]

Varje gång ett värde ska läggas till och det inte finns plats, förflyttas en pekare över den cirkulära bufferten, som letar efter en nollställd bit [6]. När den itererar över bufferten nollställer den bitar som inte är nollställda tills dess att den hittar en bit som är nollställd. Den väljer därefter att ersätta det värde som denna bit representerar med det nya värdet. Sedan avslutas sökningen.

3.4.3 CPHash

CPHash [16] är en nyckel-värde-databas som lagrar nyckel-värde-par i cacheminnena i processorns kärnor. Detta innebär att minnet delas upp i partitioner, där en partition utgör L1/L2-cachen hos en kärna. Därefter har varje kärna ansvar för sin egna hashtabell som finns i dess egna cacheminne. Denna lösning liknar MICAs [12] lösning där minnet delas upp i partitioner och som därefter tilldelas en kärna var. MICA använder sig dock av RAM-minnet i detta fall, istället för endast cache-minnen, som ju CPHash gör.

CPHash är designad för flerkärniga processorer, där varje kärna har minst två hårdvarutrådar [16]. För varje kärna finns en server-tråd, som hämtar och lagrar data i hashtabellen, och en klient-tråd, som skickar förfrågningar till server-tråden för att kunna läsa och lagra data i hashtabellen. För att avgöra vilken server-tråd klienten ska kontakta, används en hash-funktion på nyckeln som är associerad med ett visst värde. Därefter kontaktas motsvarande server-tråd genom att använda hash-värdet. Klient- och server-tråden kommunicerar med varandra genom två buffertar. Klienten skapar förfrågningar och lagrar dessa i den ena bufferten, medan server-tråden skapar motsvarande resultat för varje förfrågning och lagrar dessa i den andra. Enligt författarna ökar detta parallelismen i applikationen, eftersom klienten kan skapa kommandon samtidigt som servern svarar på dessa och lagrar resultat i en annan buffert. Exempelvis skulle det inte vara möjligt för klienten att skapa förfrågningar samtidigt som servern besvarar dessa om de enbart kommunicerade via en delad buffert.

Utvecklarna [16] motiverar denna lösning - partitionering av minnet genom att sprida data över cacheminnena i kärnor - med att detta minskar inter-kommunikation mellan kärnorna och att lås undviks. Denna motivering liknar den motivering som gavs av utvecklarna för MICA. Detta motiverades också med att lösningen bidrar till en god skalbarhet för CPHash i flerkärniga processorer. Detta på grund av det faktum att varje kärna ansvarar för sin partition av hash-tabellen, vilket eliminerar inter-kommunikation mellan kärnorna och även synkroniseringskommunikation.

3.4.3.1 LockHash

I samband med utvärderingen av CPHash utvecklade författarna en annan version av CPHash, nämligen ”LockHash” [16]. Denna version var avsedd att användas för att undersöka skalbarheten hos CPHash. I LockHash delas ett delat minne upp i ett antal partitioner som är direkt proportionerlig mot antalet kärnor. Varje partition är skyddat av ett lås. För att en klienttråd ska kunna läsa och skriva till en partition måste ett lås för motsvarande partition hämtas. Därefter kan klienten läsa och skriva till partitionen.

3.5 Valda mätvärden för beräkning av prestanda

Nedan beskrivs de mätvärden som valts för att utvärdera prestandan hos de två olika tekniska alternativ [24, 14] för temporär lagring som valts. Valet av dessa mätvärden och hur de beräknats grundar sig på en utvärdering gjord av Wenqui Cao, et al [4] (se 3.6.1).

3.5.1 Minnesanvändning

Minnesanvändningen för Redis och memcached kommer att undersökas. Detta kommer att undersökas i form av deras totala fysiska minne som allokerats, vilket även kallas för resident set size (RSS), deras externa fragmentering i heap-minnet och hur mycket minne som avallokeras. Dessa faktorer - hur mycket minne som avallokeras och allokeras - är viktiga för att även kunna avgöra deras interna fragmentering (minne som allokerats men som inte används). För att undersöka den interna fragmenteringen är det därför nödvändigt att ta bort alla värden som lagts till i Redis och memcached och därefter notera deras RSS.

`valgrind` kommer att användas vid undersökning av memcacheds heap-minne. `valgrind` är en mängd verktyg som kan användas för att undersöka en process minne på

olika sätt. Av dessa kommer verktygen `massif` och `exp-dhat` användas vid undersökning av minnesanvändning i Redis och memcached. `massif` kommer att användas i syfte att undersöka fragmenteringen i heapen för memcached och Redis, medan `exp-dhat` kommer att användas för att undersöka storleken på de minnesblock som allokeras av memcached och Redis.

3.5.2 CPU-användning

CPU-användningen av Redis och memcached kommer att beräknas i form av det genomsnittliga antalet kärnor som varje program använder sig av. Detta beräknas med hjälp av SYSSTAT [26] som kan beräkna andelen av all CPU-tid en process utnyttjar. Ett värde som är större än 100% indikerar att processen använder sig av fler kärnor än en. Exempelvis innebär ett värde av 200% att processen använder sig av två kärnor. Ju fler kärnor processen använder sig av, desto bättre.

3.5.3 Tidsåtgång för behandling av förfrågningar

Den tid det tar för en klient att skicka en förfrågan till servern och få svar kommer att undersökas. I detta fall kommer detta att ske internt i själva applikationen som utvecklats i samband med detta arbete. Tidsåtgången kommer att beräknas som differensen mellan tidpunkten då en klient fått svar på en förfrågan och tidpunkten då klienten skickade samma förfrågan. Detta kan uttryckas med följande formel:

$$\Delta t = t_2 - t_1$$

Där t_2 är tidpunkten då klienten upplevt att den fått ett svar och kan bearbeta det svar som den fått från servern, och t_1 är tidpunkten då klienten skickade sin förfrågan. Δt beskriver alltså åtgången tid för behandling av en förfrågan. Det är även underförstått att $t_2 \geqslant t_1$, eftersom det är omöjligt för klienten att få ett svar på en förfrågan som den ännu inte skickat.

3.5.4 `memtier_benchmark`

Eftersom ovanstående mätvärden (med undantag för tidsåtgången för lagring och hämtning av data) beräknades under tester där verktyget `memtier_benchmark`[7] användes, beskrivs detta verktyg kort här.

`memtier_benchmark` är ett verktyg som utvecklats av samma utvecklare som utvecklat Redis [24]. Detta verktyg används främst med syftet att kunna skicka datamängder med olika storlek och form till memcached och Redis. `memtier_benchmark` erbjuder sätt att konfigurera hur stora datamängderna som skickas är, hur varierad storleken för objekten i datamängderna är och hur många `set`-operationer i förhållande till `get`-operationer som skickas via verktyget.

3.6 Tidigare arbeten

Nedan följer liknande arbeten där författarna utvärderat memcached och Redis med avseende på bland annat CPU-användning och minnesanvändning. Hur dessa arbeten utfört utvärderingarna samt deras resultat beskrivs nedan. Dessa artiklar undersöker samma mätvärden som undersöks i detta arbete, men båda artiklarna använder sig av olika metoder för undersökningen. Bland annat använder den andra artikeln inte memtier_benchmark för att skicka data till Redis och memcached, något som den första artikeln (och detta arbete) gör.

Även tillämpningar av memcached och Redis presenteras.

[7] https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/

3.6.1 Utvärdering av prestanda med benchmarking-program

I en utvärdering utförd av Wenqi Cao et al. [4] användes bland annat verktyg som Perf, SYSSTAT och valgrind (som nämndes tidigare i kapitlet) för beräkning av mätvärden som CPU-användning och minnesanvändning. Författarna använde sig också av *memtier_benchmark* och *GITKVWR* för att kunna belasta applikationerna (Redis och memcached) med godtycklig data. Sedan undersöktes, utöver andra mätvärden, CPU-användning och fragmentering i olika scenarion där olika mängder data skickades till cache-servern. Författarna använde bland annat 5 GB stora mängder data, vilket de definierade som "små", och 10 GB stora mängder data, vilket de definierade som "stora". Sedan undersöktes Redis och memcached separat för att bland annat undersöka deras utnyttjande av minne och CPU-användning. CPU-användningen undersöktes också beroende på vilka aktiviteter applikationen utförde. Dessa aktiviteter kunde omfatta kod som exekverades i kernel-nivå, användarnivå, och hur mycket applikationen i fråga väntade på I/O-operationer.

Wenqi Cao et al. [4] kom bland annat fram till att memcached är mycket mer stabil (konsekvent) med avseende på *throughput* (operationer per sekund) än Redis under belastningarna. Detta berodde på, enligt författarna, att Redis är enkeltrådat medan memcached är flertrådat. Däremot visade det sig att memcached var mindre stabil med avseende på CPU-användning (CPU-användningen varierade kraftigt under körningstid).

När det gäller minnesanvändning använde både memcached och Redis ungefär lika mycket minne, där Redis använde något mindre minne än memcached [4]. Under två test där två olika stora mängder data skickades till Redis och memcached visade det sig att memcached använde sig av ca. 1.27% mer minne med en liten mängd data (5 GB) och ca 1.32% mer minne med en stor mängd data (10 GB). Det visade sig även att memcached endast avallokerade 0.58% minne under båda testernas körningar, medan Redis avallokerade 7.19% under en liten belastningsmängd (5 GB) och 7.49% under en stor belastningsmängd (10 GB). Med andra ord: Redis använde något mindre minne än vad memcached gjorde. Författarna kom fram till slutsatsen att både Redis allokeringsmetod (jemalloc) och memcacheds allokeringsmetod (*slab allocator* som beskrevs tidigare i kapitlet) utnyttjade minnet väl och bidrog till en låg extern fragmentering i minnet.

3.6.2 Utvärdering av prestanda med PageRank-algoritmen

I en annan utvärdering utförd av Hao Zhang et al. [30], användes en annorlunda metod för att utvärdera Redis och memcached med avseende på prestanda. Författarna använde sig av ett *datorkluster* där algoritmen *PageRank* [18] användes för att belasta Redis- och memcached-noderna med data. För varje memcached-server utvecklades en drivrutin som implementerade algoritmen PageRank som i sin tur använde sig av memcached för lagring av data. För varje Redis-instans användes Redis inbyggda funktioner för att skriva Lua-skript för att implementera PageRank-algoritmen. Författarna belastade sedan noderna genom att exekvera algoritmen för 10 iterationer och undersökte sedan bland annat CPU-användning på både användarnivå och kernelnivå. Detta var dock CPU-användning i form av mängden tid som den exekverat på de olika nivåerna. Denna tid beräknades med hjälp av programmet *time* i Linux. Även verktyget Perf användes för att kunna undersöka prestanda på arkitekturnivå.

Författarna kom fram till att memcached och Redis inte utnyttjar flerkärniga processorer på ett effektivt sätt. Dessutom kom de fram till att memcached och Redis har en "instruction cache miss rate" mellan 10 och 15% för små objekt, vilket var enligt dem oväntat. Detta har en koppling till, enligt författarna, Redis och memcacheds utnyttjande av moderna processorer, eftersom de flesta moderna processorer använder sig av "out-of-order"-exekvering. Detta innebär att instruktioner inte nödvändigtvis exekveras i den ordning som programmeraren tänkt sig. Moderna processorer som exekverar instruktioner på detta sätt tillåter flera cache-missar att bli hämtade från L1- och L2-cache och RAM-minne på en gång, medan endast en miss i instruktions-cachen åt gången kan hämtas. Författarna menar på att det-

ta innebär en dold väntetid som får memcached och Redis att verka vara aktiva vad gäller CPU-användning, men i själva verkat spenderar memcached och Redis 70% av tiden på att vänta på instruktioner som ska hämtas från minnet.

Författarna kom även fram till att TCP utgör en flaskhals i prestandan för Redis och memcached, eftersom de noterade att TCP inte hanterar små paket på ett effektivt sätt. Det faktum att det oftast är små nycklar (mindre än 32 bytes) och små data (några hundratals bytes) [1] som läses och skrivs till applikationer som memcached och Redis, talar detta ytterligare för att TCP är en bidragande faktor för ineffektivitet i detta avseende.

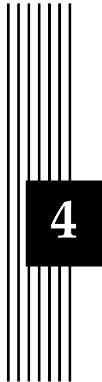
3.6.3 Tillämpningar av memcached

Facebook tillämpar memcached för temporär lagring av data i sina datorkluster [17]. De har dock lagt till ytterligare funktioner och optimeringar i den befintliga källkoden för memcached, för att det på ett bättre sätt ska kunna tillämpas för Facebooks ändamål. Bland annat har Facebook använt sig av sin egenutvecklade router (mjukvara), `mcrouter`, för att kunna sammanfoga en mängd TCP-uppkopplingar på en server. `mcrouter` fungerar alltså som en proxy för klienter som önskar att kommunicera med memcached-servrar. Detta är till följd av att data som krävs för att etablera en TCP-uppkoppling kräver mycket minne, vilket i sin tur kan skapa stora fördröjningar när en server måste upprätthålla flera uppkopplingar från en stor mängd klienter. Detta innebär också mer CPU-användning för memcached-servrarna, men genom att samla alla uppkopplingar på en annan enhet krävs mindre CPU-tid för TCP på memcached-servrarna. Facebook har även valt att skicka get-förfrågningar via UDP istället för TCP.

3.6.4 Tillämpningar av Redis

Likt Facebook, använder Craiglist sig av Redis för temporär lagring av data i datorkluster [25]. Syftet med Craiglists användning av Redis är att utnyttja flerkärniga processorer väl. För varje kärna körs en Redis-process, som antingen är en master-server eller en slave-server. Slave-servrarna används för replikering av data som finns på master-servern, medan den huvudsakliga traffiken sker via master-servern.

Craigslist mappar alla Redis-noder till olika namn. När en klient ska hämta ett visst värde med en viss nyckel används nyckelns värde för att beräkna ett hash-värde som mappas till ett nodnamn. Zawodny [25] menar att det faktum att noder är kopplade till nodnamn gör det enkelt att ändra vilken nod ett visst nodnamn pekar på. Enligt blogginlägget [25] finns 10 maskiner i Craigslists datorkluster, där varje maskin har en 4-kärnig processor. Detta innebär att $4 * 10 = 40$ Redis-processer finns i datorklustret, där 20 Redis-processer fungerar som master-servrar och resterande som slave-servrar.



4 Metod

I detta kapitel beskrivs det huvudsakliga tänkta genomförandet av arbetet. Här beskrivs planen för hur förstudien, designen och utvärderingen var tänkt att genomföras. Denna metod utvärderas sedan i resultatkapitlet (5).

4.1 Förstudie

Innan någon utvärdering av de två olika tekniska alternativ för temporär lagring av data som valts kunde utföras, var det nödvändigt att bilda en djupare uppfattning om problemet som skulle lösas. Metoden för att anskaffa denna information beskrivs nedan.

4.1.1 Tekniska frågeställningar

Det grundläggande problemet för SysPartner i detta fall var att de önskade en webbapplikation som kunde ge en överskådlig blick över anställdas semesterplaner och ledigheter. Dessa typer av data finns på SysPartners databas som innehåller stora mängder data och det kan därför ta en lång stund innan en läs-förfrågan till databasen får ett svar. På grund av detta ville alltså SysPartner se en lösning, med temporär lagring av data, som minskar belastningen på databasen. För att kunna komma fram till en lämplig lösning var det därför nödvändigt att ställa följande frågor till utvecklare på företaget:

1. Hur ofta uppdateras databasen vars data ska temporärt lagras?
2. Används någon form av temporär lagring av data i databasen just nu? Hur ofta lagras data från databasen i en cache isåfall?
3. Ska cachen innehålla en spegling av *hela* databasen?
4. Hur viktigt är det att cachen alltid är uppdaterad?
5. Hur lagras tidsrapporteringar i databasen just nu?

Dessa frågeställningar är tänkta att användas i syfte att få en bättre bild över vad som ska lösas, men även också svar på frågan om Redis eller memcached är bäst i detta fall med

avseende på funktionalitet som erbjuds. Den sistnämnda frågeställningen, *"Hur lagras tidsrapporteringar i databasen just nu?"*, ställs i syfte att få svar på inte bara hur de lagras, utan också hur och när tidsrapporteringar hamnar i databasen. Hamnar de direkt i databasen när en anställd tidsrapporterar, eller finns det något mellanliggande medium som tar emot dessa data?

4.1.2 Kravinsamling till webbapplikationen

Förutom de ovannämnda frågor som ställts i syfte att få en tydligare bild av problemet, var det också nödvändigt att intervjua berörda personer angående webbapplikationen. Berörda personer i detta fall omfattade gruppchefer och ledare inom företaget. Dessa intervjuer utfördes med syftet att samla information om användningsfall och liknande som skulle kunna påverka den slutgiltiga lösningen på problemet. De frågor som låg till grund för intervjuerna var följande:

1. Hur utför du ditt administrativa arbete just nu?
2. Vad är det för information om anställda som du vill se?
3. Hur hanterar du denna information just nu?
4. Kan denna applikation komma att bli en del av något annat system i framtiden?

Med hjälp av dessa frågor anskaffades information om användningsfall i databasen vilket bidrog med information som kunde vara av stor nytta när Redis och memcached undersöktes med avseende på funktionalitet och prestanda.

De som hade synpunkter på applikationen var verkställande direktör för SysPartner och 2 gruppchefer. Av dessa intervjuer utfördes en intervju via telefon (med en gruppchef), medan övriga intervjuer utfördes på plats hos SysPartners kontor.

4.2 Design

När information anskaffats via de ovannämnda frågorna var det lämpligt att designa en preliminär lösning på problemet. Hur denna planering gått till beskrivs nedan.

4.2.1 Modulen som kommunicerar med databasen

Denna modul är den centrala delen av arbetet och är tänkt att fungera som ett gränssnitt mellan ett godtyckligt program i SysPartners intranät och databasen. Hur denna modul kommer att implementeras beror på den information som anskaffats i förstudien. Däremot finns det krav utöver de krav som tagits fram till följd av förstudien:

1. **Expanderbarhet**. Det ska vara möjligt att expandera modulen med minimala förändringar av modulen.
2. **Temporär lagring**. Modulen ska temporärt lagra en delmängd av den data som finns i databasen. Det ska vara möjligt att ändra vilken delmängd av databasen som ska lagras.
3. **Kommunikation med SysPartners databas**. Modulen ska kunna kommunicera med SysPartners databas. Detta för att kunna hämta data från databasen.
4. **Använda en godtycklig cache**. Det ska vara möjligt för modulen att kunna använda en godtycklig cache, inte bara memcached och Redis.
5. **Generell lösning**. Det ska vara möjligt för en godtycklig applikation i intranätet att använda sig av modulen.

Dessa krav togs fram bland annat i samarbete med den externa handledare som fungerat som stöd för detta arbete på plats. Kraven togs fram med information från förstudien som grund till viss del; kravet på kommunikation med en databas och temporär lösning är just dessa krav. Övriga krav bestämdes av den externa handledaren, som ansåg att underhållsbarhet bör beaktas, vilket i slutändan resulterade i krav 1, 4 och 5.

Krav 1 och 4 talar för en lösning på problemet som ger upphov till ett expanderbart system. Därför användes enkla UML-diagram för att underlätta designen av modulen. Exempelvis talar en "generell lösning" för att modulen bör bygga på abstraktioner på det sätt att i princip vilken typ av data som helst ska kunna hämtas via modulen. Detta måste dock kunna ske i samband med att kommunikation med cachen eller databasen abstraheras, eftersom detta gör det enkelt för användare av modulen att hämta data via modulen.

För att modulen skulle kunna kommunicera med Redis och memcached krävdes bibliotek för respektive cache. Eftersom att modulen skrevs i Python användes en Python-modul för Redis respektive memcached. De moduler som användes har utvecklats av utvecklarna för respektive cache. Till följd av krav 4 ovan, bör det vara enkelt att ändra modulen på det viset att den kan använda en godtycklig cache. Detta innebär i sin tur att det ska vara enkelt att lägga till ytterligare cacheningsmetoder utan att behöva ändra den befintliga modulen. Detta implementerades med konkreta klasser som fungerade som adaptrar som modulen kunde använda för att kommunicera med en viss cache.

På grund av krav 3 ska det vara möjligt för modulen att kommunicera med SysPartners databas. Detta innebär att modulen ska kunna skicka SQL-förfrågningar till databasen för att kunna läsa data. Värt att notera är att modulen endast har läs-rättigheter i databasen, vilket innebär att det inte finns något krav på att modulen ska kunna skriva data till databasen. För kommunikation med databasen användes en modul vid namn "PyODBC". Den mängd data som modulen skulle hämta var ungefär 50 MB stor.

Med kraven i hand och vilka problem som uppstår till följd av dessa, användes alltså UML-diagram för att komma fram till en lösning som uppfyller kraven. Därefter implementerades lösningen i intranätet och sedan testades modulen i form av enhetstester.

Denna modul kommer att refereras till som *"Visma-modulen"* i fortsättningen.

4.2.2 Webbapplikationen

Utvecklingen av webbapplikationen bestod främst av front-end i detta fall. Denna applikation är inte den centrala delen i arbetet, men den ligger till grund för exempel på användningsfall och vad modulen som beskrevs ovan ska kunna göra. Vad för typ av information ska Visma-modulen kunna tillhandahålla?

Designen av webbapplikationen kommer främst att utgå från de krav som gruppledare och chefer ställt. Även här användes metoder som enkla UML-diagram och skelettkod vid designen av applikationen. Front-end kommer sedan att utvecklas i JavaScript tillsammans med biblioteken ReactJS och Bootstrap. Sedan binds front-end ihop med intranätets back-end som använder sig av Django som ramverk för webbutveckling.

4.3 Utvärdering

Den absolut nödvändigaste delen av applikationen i detta arbete var, som tidigare nämnt, Visma-modulen. Det är denna som kommer att användas som en klient för Redis och memcached. När Visma-modulen ansågs vara färdig undersöktes Redis och memcached till en början med avseende på deras CPU- och minnesanvändning genom användning av verktyg för att skicka stora mängder data till dessa. Sedan undersöktes även tidsåtgången för behandlingen av en förfrågan som skickats från Visma-modulen.

När det gäller memtier_benchmark skickas två typer av datamängder:

1. En datamängd där 100 000 förfrågningar om objekt skickas, där varje objekt är 1 KB stor.
2. En datamängd där 30 000 förfrågningar om objekt skickas, där varje objekt är 10 KB stor.

Den första mängden data resulterar i en hantering av 100 MB data och den andra mängden resulterar i en hantering av 300 MB data. Dessa storlekar valdes för att dels studera hur memcached och Redis presterar under 2 olika stora datamängder. Den största anledningen till att dessa mängder valdes var dock för att undersöka hur memcached och Redis använder minnet när blocken som allokeras är små (1 KB) och stora (10 KB). Detta för att undersöka hur Redis och memcached utnyttjar minnet beroende på storleken av de minnesblock som allokeras.

Andelen `set`- och `get`-förfrågningar var inställt på 1:10 (`SET`:`GET`) för båda datamängderna. Det skedde ingen variation mellan storleken på objekten vid varje datamängd; objekten var lika stora. En fix storlek på objekten innebar att memcached och Redis utsattes för exakt samma mängd data, vilket annars inte hade kunnat garanterats. Det var sedan förhållandet mellan memcached och Redis som undersöktes. Memcached och Redis exekverades vid separata tillfällen på en fyrakärnig processor. Memcached och Redis var konfigurerade till att använda maximalt 200 MB minne. Memcached var också konfigurerad till att använda 4 trådar (standardinställning).

4.3.1 Beräkning av CPU-användning

Det verktyg som valts för beräkning av CPU-användning är `SYSSTAT`, som bland annat användes av Wenqui Cao et al. [4] för samma ändamål. Memcached och Redis kommer att testas var för sig, de kommer alltså inte att testas samtidigt. Detta för att de inte ska kunna påverka varandra. För både Redis och memcached skickades de två datamängderna ovan till respektive cache. Samtidigt som detta pågick användes `SYSSTAT` för att samla CPU-användningen varje sekund medan datamängderna skickas.

När det gäller metoden för att beräkna CPU-användning användes alltså SYSSTAT, tillsammans med ett enkelt skript för att extrahera relevanta siffror kopplade till CPU-användning för memcached och Redis. De värden som extraherades var `cpu%` och `RSS` (som användes för undersökning av minnesanvändningen för Redis och memcached). `cpu%` betecknar CPU-användning i form av antalet kärnor en process nyttjar och `RSS` (Resident Set Size) betecknar den totala mängd RAM-minne som en process nyttjar. När SYSSTAT rapporterar 100% för CPU-användningen innebär detta att processen använder resurser som motsvarar 100% av en kärna i processorn (exempelvis 80% av en kärna och 20% av en annan). Detta värde visar därför hur väl processen i fråga utnyttjar de kärnor som finns i processorn. Även om fler processer möjligtvis påverkade denna beräkning är det ändå hur memcached och Redis förhåller sig till varandra med avseende på CPU-användning som är viktig i detta fall.

4.3.2 Beräkning av minnesanvändning

Minnesanvändningen undersöktes på samma sätt som vid undersökningen av CPU-användningen. Däremot undersöktes inte minnesanvändningen samtidigt som när CPU-användningen undersöktes, eftersom verktyget (`valgrind`) som användes för att samla data om minnesanvändning kunde påverka CPU-användningen. Verktyget `exp-dhat` användes för att samla information om den genomsnittliga storleken på blocken som allokerades i respektive cache. `SYSSTAT` användes för att beräkna RSS för memcached och Redis när de två olika datamängderna skickades till respektive cache. Därefter togs all data som lagts till i respektive cache bort, varpå RSS beräknades igen via `SYSSTAT`, detta för att undersöka hur mycket minne som avallokerats efteråt.

Vid undersökning av minnesanvändningen för memcached och Redis var det nödvändigt att utföra flera separata tester. Dessa tester utfördes för varje datamängd som skickades: den lilla datamängden (100 000 objekt med en storlek av 1 KB) och den stora datamängden (30 000 objekt med en storlek av 10 KB). För varje test skickades dessa mängder vid två olika tillfällen. Vid det första testet exekverades Redis eller memcached utan `valgrind`. När datamängden i fråga skickats, noterades RSS-värdet för Redis eller memcached. Därefter togs alla värden som lagts till i cachen bort och RSS-värdet noterades återigen. Vid det andra testet användes `exp-dhat` för att kolla hur stora block Redis eller memcached allokerade vid de två olika datamängderna. Vid det tredje testet användes `massif` på memcached för att undersöka fragmenteringen i heap-minnet, medan fragmenteringsgraden som rapporterades för av Redis-klienten användes (via kommandot `info memory`) för Redis.

Fragmenteringsgraden beräknades med hjälp av `valgrind` för memcached, medan Redis egenrapporterade fragmentering användes. Detta för att Redis inte var kompatibel med `valgrind`s `massif`-verktyg. `massif` beräknar det totala allokerade minnet i heapen och hur mycket överflödigt minne som allokerats (vilket indikerar på intern fragmentering i heap-minnet). Redis beräknar sin fragmenteringsgrad på liknande sätt där det totala oanvända minnet i heapen divideras med det använda (det som allokerats).

4.3.3 Genomförande

CPU-användning och minnesanvändning beräknades på det sätt som beskrevs tidigare, där två olika datamängder skickades till respektive cache. Anledningen till att dessa två storlekar valdes var för att studera hur minnesanvändningen förändrades beroende på storleken på objekten som hanterades.

Ett skript konstruerades där CPU-användningen extraherades från den output som SYSSTAT gav. Detta skedde, som tidigare nämnt, i samband med att två olika datamängder skickades till memcached eller Redis. Datamängderna skickades dock inte samtidigt, utan de skickades vid två olika tillfällen. Mellan dessa tillfällen tömdes också memcached och Redis på den data som lagts till. Detta för att den data som lagts till inte skulle kunna påverka nästkommande test.

När data om CPU-användningen anskaffades, undersöktes minnesanvändningen. Detta skedde genom att skicka de två datamängder som tidigare presenterats, och detta skedde vid två separata tillfällen för memcached och Redis. Det första testet anskaffade information om den genomsnittliga blockstorleken som allokerades i memcached och Redis. Detta skedde genom att extrahera information om det totala minne som allokerades, vilket gavs av verktyget `exp-dhat` i `valgrind`. Den genomsnittliga blockstorleken beräknades genom att dividera den totala mängden minne som allokerades och antalet block som allokerades.

Sedan beräknades fragmenteringsgraden i heap-minnet för Redis och memcached. Detta skedde på samma sätt som tidigare, där två olika datamängder skickades vid separata tillfällen. Fragmenteringsgraden för memcached beräknades via `massif`-verktyget som tidigare beskrevs.

RSS för memcached och Redis beräknades också. Detta skedde, som tidigare nämnt, genom att använda verktyget `SYSSTAT`. Detta skedde i samband med beräkningen av CPU-användningen. När datamängderna skickades, noterades RSS-värdet för cachen. Sedan togs alla värden som lagts till i cachen bort, varpå RSS-värdet noterades ännu en gång.

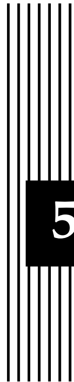
Redis och memcached konfigurerades via deras respektive konfigurationsfiler för att sätta en max-gräns för mängden minne som kan allokeras av respektive cache. Denna max-gräns var inställd på 200 MB under testerna.

Dessutom undersöktes tidsåtgången för att hämta data via Visma-modulen. Det första testet undersökte den tid det tog för att temporärt lagra en delmängd data från databasen i cachen (ungefär 60 MB data). Detta skedde vid separata tillfällen för både memcached och Redis. Sedan undersöktes även tidsåtgången för att hämta data från cachen via Visma-modulen genom att skapa 1000 Visma-objekt. Dessa Visma-objekt hämtade data från bland annat den

största tabellen som innehöll ca 44 000 tuplar. Detta beräknades genom att undersöka den tid det tog att skapa dessa 1000 objekt och sedan dividera denna tid med antalet objekt som skapats (som är 1000 st).

Tidsåtgången för att lagra en delmängd data (som motsvarade ungefär 60 MB), undersöktes också. Detta test skedde inte i samband med övriga mätningar. Redis och memcached testades var för sig, där en delmängd lagrades med hjälp av Redis respektive memcached. Detta skedde 10 gånger för både memcached och Redis; tiden det tog för respektive verktyg att lagra denna delmängd beräknades med hjälp av `time`[1] som finns i de flesta Linux-distributionerna.

[1] https://linux.die.net/man/1/time



5 Resultat

I detta kapitel presenteras resultaten som uppkommit till följd av utförandet av arbetet enligt den metod som beskrevs tidigare (4). En analys av dessa resultat återfinnes sedan i kapitel 6 (6). De resultat som presenteras här är den information som anskaffats i förstudien, den slutliga designen av Visma-modulen, samt CPU- och minnesanvändningen som noterades för Redis och memcached.

5.1 Förstudie

Nedan presenteras den information som anskaffats i förstudien. Denna information var viktig för att kunna få en bättre bild över problemet som skulle lösas.

5.1.1 Tekniska frågeställningar

Som nämndes i metodkapitlet var det nödvändigt att anskaffa information som kunde användas för att komma fram till en rimlig lösning vad gäller Visma-modulen. Varje frågeställning med respektive svar presenteras nedan.

Hur ofta uppdateras databasen vars data ska temporärt lagras? Enligt handledaren från företaget, uppdateras databasen två gånger om dagen: en gång på morgonen och en annan gång på kvällen. Detta sköts av vd för SysPartner, som trycker på en knapp för att överföra nya data till databasen.

Används någon form av temporär lagring av data i databasen just nu? Hur ofta lagras data från databasen i en cache isåfall? Vissa data i intranätet lagras temporärt i memcached, detta omfattar vissa data om anställda och liknande. Dessutom lagras vissa andra typer av data i en cache som tillhör det databassystem från Visma som SysPartner använder. Alla tidsrapporteringar hamnar till en början i denna cache, och det är sedan vd:n som trycker på en knapp för att dessa nya tidsrapporteringar ska hamna i deras databas.

Ska cachen innehålla en spegling av hela databasen? Cachen ska spegla en delmängd av databasen, alltså data som faktiskt används av applikationer i intranätet och liknande.

Hur viktigt är det att cachen alltid är uppdaterad? Enligt handledaren från företaget uppdateras deras nuvarande cache en gång varje kvart. Detta trots att databasen endast uppdateras en gång på morgonen och en gång på kvällen.

Hur lagras tidsrapporteringar i databasen just nu? Tidsrapporteringar lagras i en enskild tabell i databasen, men till en början lagras dessa i en cache som tillhör Vismas databassystem, som nämndes tidigare. Denna cache är dock oåtkomlig.

Utöver denna tekniska information bestämdes också andra tekniska krav på applikationen, med syfte att göra det enkelt att utvärdera Redis och memcached. Bland annat ska det vara möjligt att använda sig av en godtycklig cache för Visma-modulen och det ska även vara möjligt att enkelt ändra vilken cache Visma-modulen ska använda. Dessutom var det önskvärt att enkelt kunna beräkna den tid det tar för Visma-modulen att hämta och skriva data.

5.1.2 Kravinsamling till webbapplikation

Nedan presenteras den information som anskaffats angående den webbapplikation som skulle utvecklas. Denna information anskaffades genom att intervjua gruppchefer och gruppledare. Totalt blev fyra (4) personer intervjuade. Det var främst Thomas som hade synpunkter på webbapplikationen dock, medan övriga hade mindre önskemål utöver det som Thomas önskat. Därför kommer svaren inte att presenteras per intervju och istället presenteras främst Thomas svar på frågorna. Dessa svar presenteras heller inte per fråga, eftersom frågorna visade sig vara något överflödiga och flera frågor besvarades därför samtidigt under intervjuerna.

Just nu har Thomas ingen god koll alls på ledigheter och liknande kring anställda. Därför skulle han vilja ha en översiktlig bild där ledigheter för varje anställd presenteras i form av ett schema. Detta schema ska på något sätt beskriva en viss anställds ledighet för vissa veckor. När en anställd avser att ta ledigt under vissa veckor ska Thomas även få ett mail angående detta. Sedan ska det vara möjligt för Thomas att bevilja denna önskan om ledighet eller också kunna avböja denna. Detta ska alltså ske via webbapplikationen.

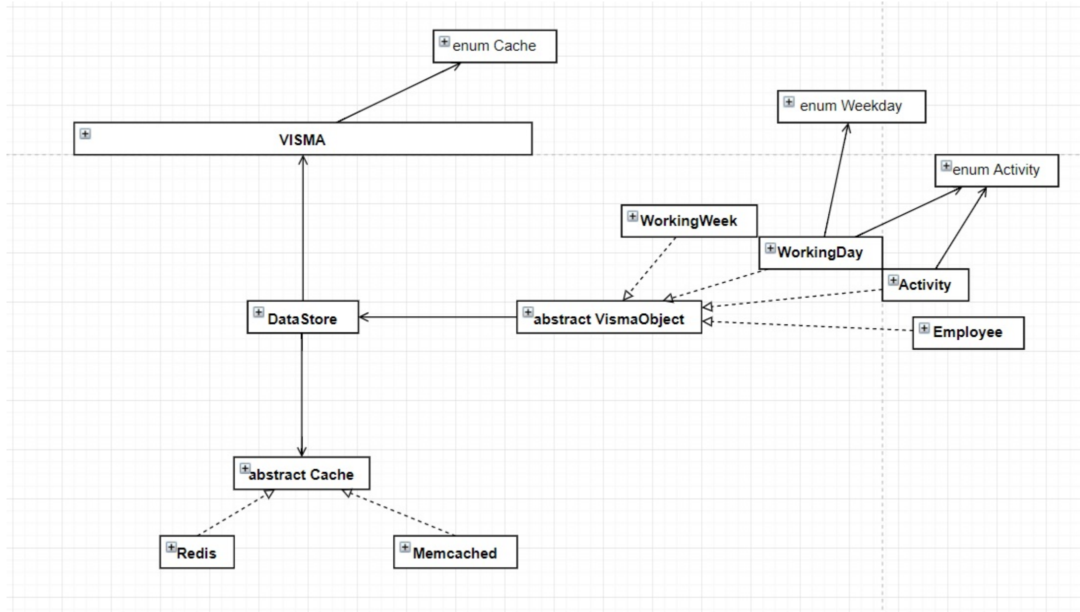
Dessutom ville Thomas ha en översiktlig bild över konsultuppdrag, där han enkelt kan se deras slut- och startdatum m.m. Just nu finns denna information i en excel-fil, men det skulle vara enklare att hantera dessa data i webbapplikationen. En annan synpunkt som en annan anställd yttrade var att man skulle kunna lägga till ”arbiträr” data till konsultuppdragen. Denna data skulle kunna beskriva någonting som har med konsultuppdraget att göra, exempelvis arbetsmiljö, vilken typ av konsultuppdrag det är osv.

5.2 Design

Nedan beskrivs den slutliga designen för främst Visma-modulen men också webbapplikationen.

5.2.1 Visma-modulen

Nedan presenteras designen för Visma-modulen (som består av flera klasser). Data som metoder och liknande har på grund av säkerhetsskäl dolts.



Figur 5.1: Visma-modulens generella struktur.

Visma-modulen består av flera moduler (Visma-modulen är, med andra ord, ett python-paket). Koden är skriven i Python.

`abstract Cache` definierar ett gränssnitt som alla klasser som används som ett gränssnitt till en viss cache måste implementera. Klasser som implementerar detta gränssnitt ska också vara ”statiska klasser”, det vill säga, klasserna ska endast inneha statiska metoder. Dessa metoder ska motsvara bland annat enkla `get`- och `set`-funktioner.

`DataStore` fungerar som ett gränssnitt mellan databasen och cachen. Denna klass har i uppgift att temporärt lagra data från databasen. Detta sker genom att kommunicera med en ytterligare klass, `VISMA`, som sköter all direkt kommunikation med själva databasen. Därefter använder sig `DataStore` av `abstract Cache` för att lagra dessa data i den cache som den för tillfället är konfigurerad till att använda. `DataStore` har alltså ingen direkt kontakt med databasen, utan använder `VISMA` som ett gränssnitt mot databasen istället.

`abstract VismaObject` definierar ett gränssnitt för att hämta data från databasen i form av python-objekt. Alla klasser som implementerar detta gränssnitt har själva ansvaret för hur de ska hämta data från databasen. De kommunicerar med `DataStore` för att hämta data. All data som hämtas, hämtas alltid från cachen, såvida klassen (`DataStore`) inte blivit konfigurerad till att inte använda sig av en cache. Användare av Visma-modulen använder sig därför av `VismaObject`-objekt för att enkelt hämta data från databasen.

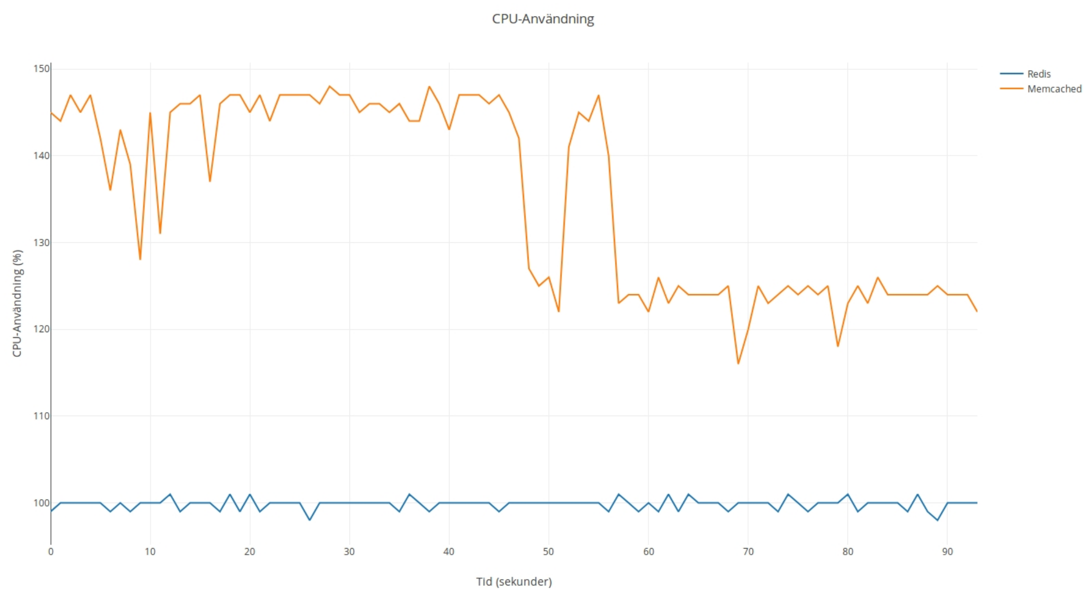
Som nämndes tidigare, `VISMA` fungerar som ett gränssnitt till databasen. Detta innebär att klassen sköter all kommunikation som innefattar SQL-förfrågningar. Sedan returnerar klassen dessa data i form av python-objekt som `dictionary` och `list`.

5.3 Utvärdering

Nedan presenteras resultatet av den utvärdering av memcached och Redis som utfördes.

5.3.1 CPU-användning

Här presenteras CPU-användningen för memcached och Redis när förfrågningar om 100 000 objekt skickades (där varje objekt var 1 KB stor). CPU-användningen visas i form av CPU-tid som processerna använt sig av. Ett värde högre än 100% indikerar att processen utnyttjar resurser som motsvarar fler kärnor än en. CPU-användningen visas endast för den första datamängden, eftersom att CPU-användningen visade sig vara i stort sett den samma vid båda datamängderna. Datamängderna skickades i en genomsnittlig konstant hastighet med 30MB/s, vilket var den hastighet som rapporterades av `memtier_benchmark`. Första halvan av datamängden bestod av endast `set`-operationer, följt av 10 gånger så många `get`-operationer.



Figur 5.2: Memcacheds CPU-användning vid behandling av 100 000 objekt.

5.3.2 Minnesanvändning

Nedan följer en tabell som visar memcacheds och Redis minnesanvändning för de två datamängder som skickats. De variabler som visas är (i) mängden data som hanterades (ii) mängden fysiskt minne som processen använde innan objekt som lagts till togs bort (iii) mängden fysiskt minne som processen använde efter borttagning av objekt som lagts till (iv) den genomsnittliga storleken på block som processen allokerat (v) fragmenteringsgraden i heap-minnet.

Datamängd	**RSS (KB)**	**RSS efter flush (KB)**	**Blockstorlek** (B)	**Fragmenteringsgrad** (%)
100 000 * 1KB	15960	15960	9 876	0.10
30 000 * 10KB	36296	36296	4 419 062	0.02

Tabell 5.1: Memcacheds minnesanvändning.

Datamängd	**RSS (KB)**	**RSS efter flush (KB)**	**Blockstorlek** (B)	**Fragmenteringsgrad** (%)
100 000 * 1KB	19900	10600	126	1.96
30 000 * 10KB	41860	9252	117	1.45

Tabell 5.2: Redis minnesanvändning.

5.3.3 Visma-modulen

Nedan följer den genomsnittliga tiden som gick åt för att lagra all data i memcached och Redis. Även den genomsnittliga tidsåtgången för behandling av data via Visma-modulen presenteras.

Cache	Tidsåtgång temporär lagring	Tidsåtgång för hämtning av objekt
Memcached	6m 34s	8 ms
Redis	6m 35s	110 ms

Tabell 5.3: Tidsåtgång för hämtning och lagring av data.

Med tidsåtgång för temporär lagring av data menas den tid det tog för memcached respektive Redis att lagra en delmängd data från databasen (via användning av Visma-modulen). Tidsåtgång för hämtning av objekt är den genomsnittliga tid det tog för memcached respektive Redis att hämta data som behövs för ett objekt i Visma-modulen. Detta objekt bygger upp sig själv genom att hämta data från en tabell som innehåller enorma mängder data och en annan tabell som innehåller något färre tupplar.

Ett annat test där ingen cache användes utfördes också. Däremot skapades inte 1000 objekt i detta fall, utan endast 10 stycken (av samma typ). I detta fall skedde ingen kommunikation med en cache, utan data som behövdes för att konstruera ett objekt hämtades direkt från databasen. Det visade sig att det tog ungefär 40 sekunder att hämta den data som behövdes för att konstruera objektet.

6 Diskussion

I detta kapitel utvärderas resultaten som åstadkommits i arbetet. Dessa resultat jämförs sedan med själva metoden som användes. Memcached och Redis jämförs även i detta kapitel, där slutsatsen till följd av denna analys presenteras i kapitel 7 (7).

6.1 Resultat

Nedan diskuteras resultatet av den utvärdering som utfört och även den modul som implementerats.

6.1.1 CPU-användning

Enligt figur 5.2 är inte memcached särskilt konsekvent i sitt användande av processorn (inte särskilt stabil). Detta är något som även författarna i [4] konstaterade. Däremot använder memcached sig av fler kärnor, vilket visar att flerkärniga processorer är till memcacheds fördel, vilket även konstaterades av Hao Zhang et al. [30]. Enligt figuren (5.2) använder sig memcached av resurser i processorn som motsvarar mellan ca 120 och 140% av processorns kapacitet. Eftersom att grafen visar att memcached använder sig av mer än 100% av processorn, innebär detta att memcached använder sig av fler kärnor än en. Däremot använder Redis sig sällan av fler än en kära. Detta är väntat, eftersom Redis till största del är enkeltrådat. Men enligt dokumentationen för Redis [24] använder Redis numera en extra tråd för backup-lagring av data i sekundärminne. Detta kan vara en förklaring till varför Redis ibland överstiger 100% i grafen (se 5.2).

6.1.2 Minnesanvändning

När det gäller minnesanvändningen verkar den interna fragmenteringen vara en stor nackdel med memcached. Enligt tabell 5.1 är mängden minne allokerat för memcached den samma både före och efter borttagning av alla objekt. Detta är på grund av det faktum att memcached inte avallokerar segment i slabs; istället markerar memcached bara dessa segment som ”borttagna”. Detta talar för ett dåligt utnyttjande av minne, eftersom att detta minne som inte används hade kunnat användas av en annan process. Däremot är det möjligt att detta inte är ett problem om det är så att memcached är den enda processen som körs på en server. Värt

att notera är också att slabbar vars segment inte innehåller något värde anses vara ”lediga”. Detta för att det ska vara möjligt att lägga till värden av en annorlunda slabklass som inte tilldelats någon av slabbarna. Annars hade detta utgjort ännu en stor nackdel med memcached: det faktum att cachen till slut endast kan lagra värden av de slabklassar som tilldelats de olika slabbarna.

När det gäller Redis (tabell 5.2) verkar den interna fragmenteringen inte vara ett lika stort problem som det är för memcached. När alla värden tas bort från Redis avallokeras det minne som allokerats för att hålla respektive värde. Däremot verkar fragmenteringen i heapen för Redis vara något högre än memcached, vilket kan bero på den genomsnittliga blockstorleken för allokering av data. Redis allokerar i genomsnitt block med en storlek på 126 och 117 byte, medan memcached allokerar betydligt större block med en storlek på 9 876 och 4 419 062 byte. Detta visar tydligt att memcached och Redis använder sig av väldigt olika metoder för allokering av data, något som även författarna i [4] konstaterade. Båda allokeringsmetoderna (för memcached och Redis) verkar dock bidra till en liten fragmenteringsgrad i heapen. Detta är något som samma författare även kuna konstatera.

Memcached verkar använda något mindre data totalt (när objekt inte tagits bort) än Redis. Detta kan, som även Wenqi Cao et al. [4] konstaterade, förklaras med att Redis har betydligt mer meta-data som måste lagras för olika datastrukturer. Memcached erbjuder ju bara en hash-tabell som dess huvudsakliga datastruktur.

6.1.3 Tidsåtgång för behandling av objekt

Tiden det tog att temporärt lagra en delmängd data från databasen var ungefär den samma för både memcached och Redis (se tabell 5.3). Däremot tog det betydligt längre tid för Redis att hämta den data som krävdes för att skapa ett Visma-objekt (som finns i figur 5.1). Detta kan bero på hur Redis hanterar vissa kommandon. För att bygga upp Visma-objektet som användes för att se hur lång tid det tog, krävdes det att hämta vissa tupplar som uppfyllde ett visst villkor. Dessa tupplar finns i den största tabellen i databasen (som har ungefär 44 000 tupplar). För att hämta alla dessa användes kommandot `mget` (multiple get) i Redis, medan vanliga upprepade `get`-kommandon användes i memcached (memcached stödjer endast ett enkelt `get`-kommando). `mget` ber Redis att hämta en mängd nycklar på en gång i en enda RTT (round trip time), istället för att Redis skickar ett enskilt paket för varje enskilt objekt som ska hämtas. Det är mycket möjligt att detta är ett långsamt kommando, vilket diskuteras i nästa avsnitt om metoden.

Däremot går det betydligt snabbare att hämta data via cache för att konstruera ett objekt, vilket ju från hela början var anledningen till att implementera temporär lagring. Det tog trots allt 8 till 118 ms att hämta data via cachen, jämför detta med den tid det tog att hämta samma data från databasen (ca 40 sekunder).

6.1.4 Visma-modulen

Visma-modulen är, som nämndes tidigare, den modul som kommunicerar med databasen och cachen för att hämta och lagra data. Eftersom att denna modul kan komma att ändras mycket i framtiden och att denna modul är tänkt att användas av en godtycklig applikation i SysPartners intranät, är det viktigt att en generell lösning på problemet används. Detta är varför modulen bland annat bygger på designprincipen ”*programmera mot ett gränssnitt, inte en implementation*”. Bland annat appliceras denna princip genom att lägga till den abstrakta klassen `abstract Cache`. Genom att använda detta gränssnitt är det möjligt att byta cache i modulen utan att något annat måste förändras i programmet. Även denna typ av lösning applicerades via den abstrakta klassen `abstract VismaObject`. Detta gör det möjligt för användare av modulen att hämta data från databasen eller cachen utan att behöva veta hur databasen eller cachen lagrar sina data. Istället kan användarna av modulen använda sig av enkla klass-objekt för att lätt kunna hämta data.

Designen kan ha en viss påverkan på effektiviteten hos modulen. Detta beror främst på de konkreta klasser som implementerar den abstrakta klassen `abstract Cache`; dels för att det beror på hur mycket data som krävs ett visst objekt och dels för att mellanliggande funktioner i modulen kallas på i samband med hämtning av data. De delar som kan ha påverkan på den utvärdering som utförts kan kopplas till de konkreta klasser som implementerar `abstract Cache`. Det finns risk att algoritmerna som används för kommunikation med Redis är mindre effektiva än de som används för memcached. Det är mycket möjligt att denna modul kan förbättras med avseende på effektivitet, men just nu anses modulen vara tillräckligt effektiv för att kunna användas.

Den temporära lagringen av data från databasen sker automatiskt var 15:e minut tills vidare. Funktionen för att temporärt lagra data finns som en metod i Visma-modulen, den temporära lagringen kan därför enkelt startas via ett externt skript. I framtiden skulle det därför vara bättre om en trigger implementerades i databasen som kan anropa på funktionen för temporär lagring av data. Detta skulle undvika att temporärt lagra data "i onödan" varje kvart.

6.1.5 Alternativa metoder

I teorikapitlet presenterades ett antal andra metoder som löser samma problem som Redis och memcached. I detta arbete valdes däremot inte dessa att undersökas, eftersom SysPartner hade ett större intresse av att använda memcached eller Redis. Den främsta anledningen till detta är på grund av det faktum att memcached och Redis har testats och används i praktiken av en mängd olika företag. Dessa två metoder har dessutom en mycket mer genomgående dokumentation än de övriga metoderna. Med andra ord anses memcached och Redis vara ett säkrare alternativ till en början.

Det är möjligt att övriga metoder som presenterats kan användas i framtiden, dock är det värt att notera att endast Redis erbjuder extra funktionalitet som övriga metoder inte erbjuder. Exempelvis erbjuder ju Redis fler datastrukturer och möjligheten att kunna lagra kopior av data i sekundärminnet. Dessa funktioner anses göra Redis till ett mer attraktivt val för SysPartners ändamål i detta fall.

6.2 Metod

Nedan diskuteras och kritiseras den valda metoden som använts i detta arbete.

6.2.1 Förstudie

Till en början var det oklart vad som var det faktiska problemet. För att förstå de tekniska motgångar och problem som kunde finnas, användes en mängd tekniska frågeställningar för att kunna uppenbara dessa. Eftersom att dessa frågor ställdes i ett specifikt scenario (ett examensarbete som utförts hos SysPartner), är det inte säkert att dessa frågor skulle kunna ge en bättre bild över det problem som ska lösas i ett annat scenario. Det viktiga i detta avseende är att ta reda på varför temporär lagring behövs, hur ofta måste det ske och vad cachen ska innehålla. Någonting som saknades i den valda metoden för att få en bättre bild över problemet var hur en anställd faktiskt anses vara ledig enligt den data som finns i databasen. Det krävdes nämligen i slutändan att studera databasen i sig för att få en bild över vad som behövs för att lösa problemet. Medan de valda tekniska frågeställningarna hjälpte till att belysa problemet ännu mer, belös de inte vad som *behövdes* tillräckligt bra för att lösa problemet. Ett annat exempel på detta var att den nuvarande modulen (Visma-modulen) var skriven i Python 3, medan intranätet var skrivet i Python 2. Annars anser jag att frågorna gav en bra bild över det faktiska problemet.

Kravinsamlingen till den webbapplikation som skulle utvecklas användes mest som grund för användningsfall: hur används databasen, hur ofta används databasen och vad är

det som ska hämtas från databasen? Detta ger en god bild över hur mycket databasen blir belastad och vad för typ av data som ska finnas i cachen. När det gäller de faktiska frågor som ställdes visade det sig att de var redundanta: svar på en fråga kunde användas som svar på en annan. Det var heller inte särskilt många som hade många synpunkter på webbapplikationen, utan endast företagets vd hade synpunkter på den. Den viktigaste typen av information som ska anskaffas i denna process är information kring den faktiska arbetsuppgiften som personen i fråga utför. Utifrån denna information är det sedan enklare att ställa frågor om vad som behövs, till exempel: vilken data behövs? Var finns dessa data? Dessa frågor saknades under intervjuerna, men som sedan blev besvarade under arbetets gång.

6.2.2 Design

Som nämnts i tidigare avsnitt fanns det krav på expanderbarhet och effektivitet hos Visma-modulen. Därför användes en generell lösning på problemet som bland annat gör det möjligt att använda flera olika typer av cachningsmetoder för att lösa problemet. Den slutliga designen visade sig vara bra vad gäller expanderbarhet. Det är möjligt att lägga till fler Visma-objekt som representerar olika typer av data som kan sammanställas från databasen. Det är t.o.m. möjligt att antalet Visma-objekt blir väldigt många i framtiden. Det kanske skulle vara bättre om det istället fanns ett objekt för varje tabell i databasen, så att objekten istället representerar tabeller, istället för specifika typer av objekt som sammanställts med hjälp av flera olika tabeller.

För att kunna upprepa en liknande lösning på ett liknande problem - att konstruera en modul som hämtar data från en godtycklig databas och lagrar dessa i en cache, samt hanterar dessa i cachen - är det en god idé att följa samma designprincip som nämndes i resultatkapitlet. I detta fall var det tänkt att memcached och Redis skulle kunna användas i modulen och att det skulle vara enkelt att ändra vilken av dem modulen skulle använda. Detta kan generaliseras på sådant vis att modulen ska kunna kommunicera med en *godtycklig cache*, vilket exempelvis implementeras via en abstrakt klass (eller gränssnitt, beroende på vilket språk man använder sig av). På detta sätt är det enkelt att lägga till en ny cache, eller ändra kommunikationen med en befintlig cache, utan att behöva ändra något annat i programmet. I den slutliga lösningen i detta fall kan man säga att de konkreta klasser som implementerar `abstract Cache` fungerar som adaptrar vilket gör det möjligt för modulen att kommunicera med en godtycklig cache.

Det är mycket möjligt att den design som användes i slutändan inte är särskilt lämpad för det språk som modulen skrevs i (Python 3) eftersom begrepp som gränssnitt (i Java-jargong) inte finns i Python. Istället implementerades dessa som abstrakta klasser, endast för att kunna tilldela ett ultimatum för varje subklass. De klasser som härleder denna klass måste i sin tur vara statiska, eftersom det ska bara kunna finnas en klass som talar med motsvarande cache. Statiska klasser kan anses vara underliga att använda i Python, eftersom moduler i Python redan uppfyller denna funktion. Men i detta fall används statiska klasser som härleder en abstrakt klass endast för att kunna typ-kontrollera data i Visma-modulen. Därav är det möjligt att den nuvarande lösningen är mer komplex än vad den hade kunnat vara, vilket skulle kunna förbättras i framtiden.

6.2.3 Utvärdering

Nedan beskrivs metoden som använts för att beräkna varje egenskap hos memcached och Redis (och Visma-module): CPU-användning, minnesanvändning och tidsåtgång för temporär lagring av data från databasen, samt den tid som gick åt för att hämta data från respektive cache.

6.2.3.1 Datamängder som skickats

Som nämndes i metodkapitlet skickades en mängd data av en fix storlek till memcached och Redis. Det är möjligt att resultatet hade sett betydligt annorlundare ut om storleken på objekten var mer varierad. I värsta fallet för memcached, där varje objekt är av olika slabklasser, skulle detta innebära att den interna fragmenteringen för memcached är betydligt större än vad som beräknats i detta arbete.

6.2.3.2 Beräkning av CPU-användning

Vid ungefär 50 sekunder i grafen för memcacheds CPU-användning (se 5.2), uppstår plötsliga förändringar i CPU-användningen. Det är inte helt klart vad detta beror på, men det kan bero på en plötslig mängd data som orsakade skrivningar till minnet. Dessutom verkar det som att `get`-operationerna kräver betydligt mindre resurser, eftersom ju första halvan innefattar endast `set`-operationer och den andra halvan innefattar endast `get`-operationer.

Det beräknade värdet för CPU-användningen beror alltså på följande faktorer: hur många processer körs på maskinen på vilken memcached och Redis exekveras och processorns arkitektur. I detta fall exekverades memcached och Redis på en fyrakärnig processor.

6.2.3.3 Beräkning av minnesanvändning

Med tanke på att testerna utfördes isolerade från varandra är det möjligt att minnesanvändningen inte såg exakt likadan ut för varje test. Detta kan innebära exempelvis att fragmenteringsgraden var annorlunda under test 2 och test 3. Det viktiga är dock hur memcached och Redis förhåller sig till varandra med avseende på de mätvärden som tagits fram. Det är helheten som är det viktiga. Exempelvis är det inte sannolikt att den genomsnittliga blockstorleken av minne som allokeras är exakt den samma vid varje körning. Däremot är det tydligt att exempelvis memcached använder betydligt större block än vad Redis gör vid varje körning.

Vid upprepning av denna metod bör liknande slutsatser kunna dras. Däremot är det som sagt inte säkert att mätvärdena ser exakt likadana ut.

6.2.3.4 Beräkning av tidsåtgång för behandling av data

Tidsåtgången för behandling av data beräknades via Visma-modulen. Den tid som beräknades för Redis och memcached kan ha påverkats av själva algoritmerna i Visma-modulen för att skapa ett objekt av data som finns i cachen. Även här är det dock förhållandet mellan Redis och memcached som spelar roll. Det tog i genomsnitt 118 ms (tabell 5.3) för Redis att hämta en stor mängd data för ett objekt, vilket var betydligt mer än memcached som gjorde samma sak på 8 ms. Det är dock mycket möjligt att Redis kan optimeras till att hämta dessa data snabbare. Till en början hämtades all data via Redis genom att skicka flera `get`-kommandon. Då tog det hela 4 sekunder för Redis att skapa samma objekt. Detta visade sig bero på att ett paket skickades för varje `get`-kommando, vilken innebar att en RTT spenderades för varje kommando. För att åtgärda detta testades sedan `pipelining` [24] i Redis, vilket innebär att kommandon lagras i en buffert och sedan kan dessa kommandon skickas i ett enda paket. Detta innebar att tidsåtgången för att hämta samma data sjönk till 2 sekunder, vilket var en förbättring men fortfarande långsamt. Därför användes slutligen kommandot `mget` för att hämta alla nycklar, vilket innebar att tidsåtgången sjönk till 118 ms.

Det är möjligt att Redis kan optimeras ytterligare för att hämta samma mängd data. Det är däremot inte helt klart på vilket sätt Redis kan optimeras ytterligare och det är dessutom möjligt att algoritmen i Visma-modulen inte är optimal i nuläget. Men att algoritmen i Visma-modulen skulle bidra till den långa tiden som det tog för Redis att hämta stora mängder data är tveksamt. Detta på grund av det faktum att genomsnittstiden för Redis att hämta samma typ av data beräknades också utan Visma-modulen, i syfte att undersöka om det är

Visma-modulens algoritmer eller själva Redis-kommandot i sig som var långsamt. Liknande fördröjning uppstod vid denna undersökning.

6.2.4 Val av mätvärden

De mätvärden som valts grundade sig mest i ett tidigare arbete som beskrevs i teorin [4]. Detta tidigare arbete jämförde Redis och memcached med avseende på en rad olika egenskaper, varav minnesanvändningen och CPU-användningen i det arbetet valdes även i detta arbete. Främst valdes dessa för att de ansågs vara mest relevanta för arbetets ändamål (minnesanvändning och effektivitet med avseende på CPU-användning), men också för att undersöka om det var möjligt att åstadkomma liknande resultat som Wenqui Cao et al. [4] kom fram till. Dessa mätvärden undersöktes i detta arbete med samma verktyg som dessa författare använde. Det visade sig i slutändan att ungefär samma slutsatser kunde dras i detta arbete som Wenqui Cao et al. kom fram till.

6.3 Felkällor

Eftersom mätvärdena beräknades i en miljö där memcached och Redis inte kördes isolerade kunde olika faktorer påverka mätvärdenas resultat. De valda alternativen för temporär lagring av data exekverades på samma maskin som SysPartners web-server. Detta kunde innebära att den tid det tog för memcached eller Redis att behandla en förfrågan tog olika lång tid. Men även traffiken i det nätverk som testerna utfördes i kunde påverka den tid det tog för förfrågningar att bli behandlade, även om dessa utfördes i ett privat lokalt nätverk. Det samma gäller för beräkningen av CPU-användning eftersom memcached och Redis inte exekverades i en isolerad miljö, utan i en maskin där andra processer som exempelvis en web-server körs.

Dessutom användes Redis egenrapporterade data vid undersökning av fragmentering för Redis, medan `massif` användes för att beräkna detsamma i memcached. Detta kan innebära att värdena som rapporterades av dessa 2 verktyg beräknades på annorlunda sätt, vilket skulle kunna ge ett missvisande resultat i slutändan.

En annan felkälla är att Redis och memcached inte optimerats. Exempelvis var Redis betydligt långsammare vid hämtning av data från cachen än vad memcached var. Om man beaktar resultatet som Wenqui Cao et al. [4] borde Redis ha varit minst lika snabb som memcached. Det är möjligt att den nuvarande konfigurationen av Redis inte är optimal, vilket skulle kunna ha en påverkan på det slutliga resultatet för tidsåtgång vid hämtning av data.

6.4 Källkritik

Många av källorna som använts har publicerats i kända tidskrifter och konferenser. Däremot finns det en mängd webbsidor som besökts för att anskaffa information kring Redis och memcached. Den dokumentation som finns på github för memcached [21, 27] är hänvisade till via huvudsidan för memcached [14]; därför anses dessa sidor vara pålitliga. Vissa källor är dock ganska gamla [19, 18, 22, 6]. Det gemensamma med dessa artiklar är dock det faktum att de presenterar nya begrepp och algoritmer, något som övriga artiklar som refereras till i detta arbete använt sig av [8, 30]. De algoritmer som presenterats i dessa artiklar är något som tidigare arbeten byggt på. Däremot bygger detta arbete på att undersöka fragmentering, något som togs upp i [22], som publicerades 1969. Eftersom de begrepp som definierades i artikeln - extern och intern fragmentering - verkar användas i moderna artiklar [4, 7], bedöms idéerna vara lika aktuella idag.

6.5 Arbetet i ett vidare sammanhang

Under förstudien togs ett antal krav fram som hanterade både tekniska aspekter och användar-aspekter. De tekniska kraven byggde mycket på att förenkla underhållet av modulen i framtiden. Ett exempel är att det skulle kunna vara möjligt att använda sig av en godtycklig cache utan att några större förändringar i modulen krävdes. Skulle detta krav inte ha uppfyllts, skulle det förmodligen ske mycket arbete med underhåll av modulen i framtiden om många applikationer i intranätet berodde på modulen. Vid framtagning av kraven var det därför önskvärt att kunna minimera denna påverkan för andra applikationer. Målet var att utveckla en modul som minimerade det underhåll som krävs för att fler applikationer ska kunna använda modulen, utan att behöva modifieras om modulen i sig ändrades.

Därför definierades ett gemensamt gränssnitt för cache-verktyg, där en utvecklare sedan kan lägga till en ny cache i form av en adapter som översätter gränssnittet för cache-verktyget till det gränssnitt som modulen är byggd på. Däremot om gränssnittet för modulen ändras kan det innebära att övriga applikationer som beror på modulen måste underhållas i form av anpassningar till det modifierade gränssnittet.

I det stora hela innebär detta ytterligare arbete i form av underhåll av en ny komponent i ett system. Detta arbete hamnar i SysPartners händer. Istället skulle man kunna exempelvis förhandla med Visma för att kunna åtgärda detta problem och istället lägga detta ansvar på Visma, eller en annan tredje part som erbjuder ett system som fyller samma funktion som den modul som utvecklats i detta arbete. Huruvida detta blir valet i framtiden eller ej avgör SysPartner själva. Även om det må vara enkelt att ändra modulen utan att övriga applikationer som beror på modulen påverkas nämnvärt, kan denna modul komma att expanderas i framtiden vilket också kan påverka underhållsbarheten för modulen. Dokumentation måste i samband med detta uppdateras och underhållsbarheten måste upprätthållas. I framtiden kanske andra utvecklare arbetar med modulen och då är det viktigt att dessa utvecklare kan sätta sig in i koden snabbt. Det är även viktigt att dessa utvecklare, om de exempelvis expanderar modulen, upprätthåller den standard för underhållsbarhet som SysPartner bestämmer.

Underhållsbarheten i modulen påverkar direkt kostnaden för att underhålla den. Huruvida det är billigare att köpa in ett nytt system som gör detsamma fast som lägger ansvaret på en tredje part eller ej beror därför på hur lätt det är att underhålla modulen. Är modulen svår att underhålla kan det innebära att utvecklare hos SysPartner i framtiden lägger ner mycket tid (och pengar) på att underhålla modulen, istället för att exempelvis expandera och förbättra funktionaliteten i modulen. Detta skulle även hindra utvecklare från att vidareutveckla andra komponenter i intranätet, just på grund av det faktum att modulen måste underhållas, vilket kan kosta mycket tid och resurser.

7 Slutsats

Syftet med arbetet var att undersöka memcached och Redis med avseende på CPU-användning, minnesanvädning och tidsåtgång för att hämta stora mängder data från deras respektive cache. Syftet var också att hitta en lösning för SysPartner där data från deras databas kan lagras i ett centralt medium, som utgörs av Visma-modulen i detta fall. Sedan används denna modul av godtyckliga applikationer i SysPartners intranät för att öka responstiden när data från databasen ska läsas av.

Det viktigaste målet uppnåddes, nämligen att påskynda processen att hämta data från databasen genom att använda temporär lagring av data. Med utgång från svaret på de två frågeställningar som ställts är det alltså tänkt att avgöra om Redis eller memcached ska användas. Båda frågeställningarna presenteras nedan tillsammans med kortfattade sammanfattningar av hur Redis och memcached förhåller sig till frågorna.

1. *"Vilken av Redis och memcached är mest lämpad för SysPartner med avseende på funktionalitet och användningsfall?"* Med tanke på att cachen väldigt sällan skrivs till och att cachen alltid ska hålla en viss delmängd av data, talar detta för att värden som lagras i cachen inte ska tas bort till följd av att minnet blir fullt. Det ska heller inte finnas någon typ av utgångstid för värdena, eftersom det skulle annars innebära att data som ska finnas i cachen saknas. Eftersom att Redis erbjuder funktionalitet som förhindrar att data försvinner talar detta för att Redis är bättre i detta avseende. Det är dock möjligt att åstadkomma samma effekt med memcached genom att sätta utgångstiden för varje värde till 0, vilket indikerar att värdet aldrig ska tas bort. Däremot exekveras LRU-algoritmen för att ersätta värden när minnet tar slut. Därför bör minnet alltid kunna hålla den delmängd data som lagras, om memcached ska användas. Eftersom Redis erbjuder funktioner för detta ändamål är därför Redis bäst i detta avseende.

2. *"Vilken av Redis och memcached presterar bäst med avseende på CPU-användning, minnesanvändning och tidsåtgång för behandling av läsförfrågningar?"* När det gäller CPU-användning verkar memcached prestera bäst, eftersom den ju är flertrådad och är därför mer produktiv än vad Redis är. Däremot är den interna fragmenteringen i memcached ett stort problem, speciellt om slabklasstillhörigheten för alla värden är väldigt utspridd. Redis verkar därför utnyttja minnet bättre än memcached. Däremot kan memcached hämta stora mängder data på en kortare tid än Redis, men detta kan möjligtivs, som tidigare nämnts, optimeras.

Med ovanstående information till hands: ska SysPartner använda sig av Redis eller memcached? Just nu är det en fråga om tid. Eftersom det tar betydligt längre för Redis att hämta stora mängder data i Visma-modulen kan det vara fördelaktigt att använda sig av memcached. Det verkar inte heller som att SysPartner i nuläget är i behov av de extra funktioner som Redis erbjuder, som exempelvis backup-lagring av data. Detta behövs inte i detta fall, eftersom cachen inte används för den huvudsakliga lagringen av datan (detta är SysPartners databas uppgift). Däremot kan det vara så att SysPartner i framtiden kan komma att bli i behov av funktionalitet som erbjuds av Redis. Dessutom, som svaret på frågeställning 1 antydde, är Redis mer lämpad för det ändamål som cachen används till.

Minnesanvändningen i memcached är dock en stor brist. Med svaret på frågeställning nummer 1 ovan - att memcached måste alltid kunna hålla all data som temporärt lagras - och det faktum att den interna fragmenteringen är stor i memcached, talar detta ytterligare för att den interna fragmenteringen i memcached är en stor nackdel. Jag skulle därför rekommendera att använda Redis istället för memcached, men det beror givetvis på hur mycket minnet värderas på SysPartners servrar. Det är värt att ha i åtanke att ju mer spridd storlekarna på objekten är, desto större blir den interna fragmenteringen i memcached, som kort diskuterades i diskussionen. Även om variationen är liten, anser jag att, med ovanstående resonemang, Redis är mest lämpad för SysPartner.

7.0.1 Framtida arbete

Det är en god idé att cachen inte temporärt lagrar data varje kvart och istället endast lagrar data när databasen uppdaterats. Detta innebär att resurser inte spenderas på många onödiga temporära lagringar under dagen, eftersom databasen endast uppdateras en gång på morgonen och en gång på kvällen. Detta skulle kunna implementeras via en trigger i SysPartners databas, som kallar på funktionen i Visma-modulen som temporärt lagrar data. I samband med detta skulle det även vara bra om endast data som tagits bort från databasen tas bort från cachen, och data som lagts till i databasen läggs till i cachen. Detta istället för att hämta all data på nytt från databasen, vilket innebär redundant och onödigt arbete eftersom den data som lagras på nytt redan finns i cachen.

Litteratur

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang och Mike Paleczny. ”Workload analysis of a large-scale key-value store”. I: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 40. 1. ACM. 2012, s. 53–64.

[2] Brian T Berkowitz, Sreenivas Simhadri, Peter A Christofferson och Gunnar Mein. *Inmemory database system*. US Patent 6,457,021. Sept. 2002.

[3] *Caching at Reddit*. `https://redditblog.com/2017/01/17/caching-at-reddit/`. Besökt: 2017-10-05.

[4] Wenqi Cao, Semih Sahin, Ling Liu och Xianqiang Bao. ”Evaluation and Analysis of In-Memory Key-Value Systems”. I: *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE. 2016, s. 26–33.

[5] Damiano Carra och Pietro Michiardi. ”Memory partitioning in memcached: An experimental performance analysis”. I: *Communications (ICC), 2014 IEEE International Conference on*. IEEE. 2014, s. 1154–1159.

[6] Fernando J Corbato. *A paging experiment with the multics system*. Tekn. rapport. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[7] Jason Evans. ”A scalable concurrent malloc (3) implementation for FreeBSD”. I: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[8] Bin Fan, David G Andersen och Michael Kaminsky. ”MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.” I: *NSDI*. Vol. 13. 2013, s. 385–398.

[9] Brad Fitzpatrick. ”Distributed caching with memcached”. I: *Linux journal* 2004.124 (2004), s. 5.

[10] *Introduction to Redis*. `https://redis.io/topics/introduction`. Besökt: 2017-10-06.

[11] Poul-Henning Kamp. ”Malloc (3) revisited.” I: *USENIX Annual Technical Conference*. 1998, s. 45.

[12] Hyeontaek Lim, Donsu Han, David G Andersen och Michael Kaminsky. ”MICA: A holistic approach to fast in-memory key-value storage”. I: USENIX. 2014.

[13] Kurt Mehlhorn och Peter Sanders. ”Hash Tables and Associative Arrays”. I: *Algorithms and Data Structures: The Basic Toolbox* (2008), s. 81–98.

[14] *Memcached*. https://memcached.org/. Besökt: 2017-10-04.

[15] *Memory Optimization*. https://redis.io/topics/memory-optimization. Besökt: 2017-10-09.

[16] Zviad Metreveli, Nickolai Zeldovich och M Frans Kaashoek. ”Cphash: A cache-partitioned hash table”. I: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, s. 319–320.

[17] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab m. fl. ”Scaling Memcache at Facebook.” I: *nsdi*. Vol. 13. 2013, s. 385–398.

[18] Lawrence Page, Sergey Brin, Rajeev Motwani och Terry Winograd. *The PageRank citation ranking: Bringing order to the web.* Tekn. rapport. Stanford InfoLab, 1999.

[19] Rasmus Pagh och Flemming Friche Rodler. ”Cuckoo hashing”. I: *European Symposium on Algorithms*. Springer. 2001, s. 121–133.

[20] *Partitioning: How to split data among multiple Redis instances.* https://redis.io/topics/partitioning.

[21] *Performance*. https://github.com/memcached/memcached/wiki/Performance. Besökt: 2017-10-05.

[22] Brian Randell. ”A note on storage fragmentation and program segmentation”. I: *Communications of the ACM* 12.7 (1969), 365–ff.

[23] *Random notes on improving the Redis LRU algorithm*. http://antirez.com/news/109. Besökt: 2017-10-16.

[24] *Redis*. https://redis.io/. Besökt: 2017-10-05.

[25] *Redis Sharding at Craigslist*. https://blog.zawodny.com/2011/02/26/redis-sharding-at-craigslist/. Besökt: 2017-12-05.

[26] *SYSSTAT*. http://sebastien.godard.pagesperso-orange.fr/. Besökt: 2017-10-17.

[27] *UserInternals*. https://github.com/memcached/memcached/wiki/UserInternals. Besökt: 2017-10-05.

[28] *Using Redis as an LRU cache*. https://redis.io/topics/lru-cache. Besökt: 2017-10-10.

[29] *Visma*. https://vismaspcs.se/. Besökt: 2017-11-20.

[30] Hao Zhang, Bogdan Marius Tudor, Gang Chen och Beng Chin Ooi. ”Efficient in-memory data management: An analysis”. I: *Proceedings of the VLDB Endowment* 7.10 (2014), s. 833–836.