CHALMERS

GÖTEBORGS UNIVERSITET

Indexering av audit-loggfiler i Microsoft Azure Kubernetes Service

Examensarbete inom högskoleprogrammet Datateknik

Erik Wahlberger

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA, GÖTEBORGS UNIVERSITET

Göteborg 2023

www.chalmers.se

Examensarbete 2023

Indexering av audit-loggfiler i Microsoft Azure Kubernetes Service

Erik Wahlberger

GÖTEBORGS UNIVERSITET

CHALMERS

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola, Göteborgs Universitet
Göteborgs Universitet
Göteborg 2023

Indexering av audit-loggfiler i Microsoft Azure Kubernetes Service
Erik Wahlberger

Handledare: Philip Laine, Xenit AB. Edvin Norling, Xenit AB. Lars Svensson, Chalmers Tekniska Högskola, Embedded Electronics Systems and Computer Graphics.
Examinator: Jonas Almström Duregård

Examensarbete 2023
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola, Göteborgs Universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Skriven i LaTeX
Göteborg 2023

Indexering av audit-loggfiler i Microsoft Azure Kubernetes Service
Erik Wahlberger
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola, Göteborgs Universitet
Göteborgs Universitet

Sammanfattning

Denna rapport syftar till att finna metoder för att öka sökbarheten i så kallade audit-loggfiler från Kubernetes-kluster. Dessa filer fås i JSON-format och lagras i molnplattformen Microsoft Azure.

Två varianter av ett mjukvarusystem har utvecklats. Variant 1 använder databashanteringssystemet MongoDb för att lagra information från audit-loggfiler på ett sätt sådant att sökbarheten ökas. Variant 2 använder ett filbaserat index baserat på ett B+-träd, där endast metadata lagras, som sedan används vid sökning av information i audit-loggfiler.

Båda systemen har jämförts med avseende på indexeringstid, sökningstid, använd lagringskapacitet samt kostnadseffektivitet. Variant 1 lagrar information i databashanteringssystemet, vilket ej är fallet för variant 2 som hämtar information direkt ur filer lagrade i Microsoft Azure. För att lättare jämföra de två varianterna utan påverkan av yttre faktorer såsom nätverksuppkopplingshastighet, utvecklades en modifierad version av variant 2 där information hämtades från lokalt lagrade audit-loggfiler.

Resultaten visar på att variant 2, med lokalt lagrad data, presterar bäst vid samtliga mätningar.

Nyckelord: Microsoft, Azure, Kubernetes, audit, JSON, B+-träd, indexering, MongoDB.

Förord

Denna rapport är en del av mitt examensarbete inom högskoleingenjörsprogrammet med inriktning Datateknik vid Chalmers Tekniska Högskola.

Sedan barnsben har jag fascinerats av datorer och mjukvara. Jag blev tidigt intresserad av mjukvaruutveckling, server-administration och dator-nätverk. Dessa intressen var anledningarna till att jag sökte en utbildning vid Chalmers Tekniska Högskola. Efter ett antal programbyten landade jag i ämnet datateknik, där jag nu avslutar min utbildning.

Jag skulle vilja tacka Philip Laine och Edvin Norling på Xenit AB för deras kontinuerliga engagemang och hjälpsamhet under hela projektets gång. Det hade inte varit möjligt att utföra detta projekt utan er. Jag skulle även vilja tacka Lars Svensson från Chalmers Tekniska Högskola för den värdefulla hjälp och respons du har givit mig för denna rapport.

Erik Wahlberger, Göteborg, april 2023

Akronymer

I listan nedan återfinns de akronymer som har använts i denna rapport, i alfabetisk ordning:

AKS	Azure Kubernetes Service
API	Application Programming Interface
CNI	Container Network Interface
CRI	Container Runtime Interface
CSI	Container Storage Interface
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transport Protocol Secure
MA	Microsoft Azure
NoSQL	Not only SQL
SQL	Structured Query Language

Innehåll

1

Inledning

IT-säkerhet är ett ämne som har blivit alltmer aktuellt i takt med att produkter och tjänster i samhället digitaliseras. Gemensamt för samtliga digitala lösningar är en ökad risk att bli utsatt för attacker över internet med motiv såsom att olovligt extrahera känslig data, överbelasta mjukvarutjänster samt exekvera olovlig programkod. I stora, komplexa mjukvarusystem blir det en utmaning att kartlägga exakt när en attack utförts, vilka system som är påverkade, samt hur dessa system har påverkats av attacken.

I takt med att storleken av internet-uppkopplade mjukvarusystemen, samt kundbasen för dessa, ökar blir även kostnaden för att driva dessa system högre då det krävs fler och kraftfullare datorer för att hantera den ökande belastningen. Detta har lett till ett paradigmskifte inom teknologi-sektorn, där alltfler mjukvarusystem drivs av så kallade molntjänster, där hårdvaru-infrastruktur hyrs ut av företag såsom Microsoft, Amazon och Google. När dessa molntjänster används kan kunderna ta del av billigare hårdvaru-infrastruktur med en hög säkerhetsstandard, som dessutom kan skalas i takt med mjukvarusystemens storlek och komplexitet.

I denna rapport undersöks så kallade audit-loggfiler, vars syfte är att införa spårbarhet i ett Kubernetes-kluster. Kubernetes är en plattform vars syfte är att hantera isolerade applikationsmiljöer som sträcker sig över flera server-datorer, ett så kallat kluster. Med hjälp av audit-loggfiler är det möjligt att spåra varenda aktivitet i ett Kubernetes-kluster. Mer specifikt undersöks möjligheten att läsa in, bearbeta samt lagra data från audit-loggfiler för att på sådant vis finna effektiva metoder för att öka sökbarheten i dessa.

1.1 Syfte

Projektet syftar till att undersöka, samt finna optimala metoder för att inhämta, indexera samt lagra audit-loggfiler från Microsoft Azure Kubernetes Service.

1.2 Mål

Under projektets gång skall en mjukvara utvecklas. Mjukvaran skall indexera samt söka efter data i audit-loggfiler lagrade i Microsoft Azure såväl som lokalt på en dator. Två varianter av denna mjukvara skall utvecklas, sedan utvärderas respektive mjukvaruvariants prestanda med avseende på inläsningstid, hämtningstid, lagringsstorlek och kostnadseffektivitet.

Den ena mjukvaran skall använda ett databashanteringssystem för indexering samt lagring av audit-loggdata. Den andra varianten skall använda en egenutvecklad indexeringslösning. Målet med projektet är således att jämföra prestanda för ett generellt databashanteringssystem och en egenutvecklad indexeringslösning.

1.3 Avgränsningar

Arbetet avgränsas genom att endast framställa två varianter av en mjukvara för inläsning, bearbetning samt lagring av audit-loggfiler. Då arbetet utförs med vägledning samt resurser från Xenit AB avgränsas arbetet ytterligare genom att anpassa den utvecklade mjukvaran för att användas inom molntjänsten Microsoft Azure, då detta är en av de molntjänster Xenit AB använder sig utav. Användning av den utvecklade mjukvaran inom andra molntjänster faller därmed utanför ramen för detta arbete.

2

Teknisk bakgrund

I detta avsnitt berörs bakomliggande koncept och teorier som utgör grunden för denna rapport.

2.1 Containerinkapsling

Containerinkapsling [1] är en metod vars syfte är att skapa isolerade, portabla miljöer för programvaror. En containeravbildning är ett standardprogramvarupaket som paketerar programvaror tillsammans med deras konfigurationer och beroenden, vilket gör det möjligt att distribuera programvaror och samtidigt garantera att miljön där programvaran exekveras hålls konstant. Containeravbildningen används sedan när en container instansieras, det vill säga när programvaran skall exekveras i dess konstanta miljö. Containerinkapslingen delar värdoperativsystemets kärna, vilket leder till en kort uppstartstid av nya containrar samt ett lågt resursanvändande. Därmed är det möjligt att instansiera ett stort antal containrar på en värddator, i jämförelse med liknande metoder för applikations-isolering, såsom virtuella maskiner. Vidare leder den snabba uppstartstiden till att containrar passar väl för programvaror vars belastning varierar, eftersom det är möjligt att variera antalet aktiva containrar baserat på den nuvarande belastningen.

Enligt Atlassian [2] virtualiserar en container operativsystemet, medan en virtuell maskin virtualiserar den underliggande hårdvaran. Virtuella maskiner är därför bättre lämpade när det krävs fullständig isolering av miljöer. I det fall då en virtuell maskin skulle bli infekterad av en mjukvaruattack, är det ej möjligt att samtidigt infektera andra virtuella maskiner på samma värddator. Däremot skulle just en sådan attack vara möjlig då containrar används. Virtuella maskiner är även fördelaktiga då en virtuell miljö behöver direkt åtkomst till hårdvara. Detta eftersom virtuella maskiner gör det möjligt att erhålla exklusiv tillgång till PCI-enheter på värddatorn [3], såsom nätverkskort eller lagringsenheter.

2.2 Kubernetes

Kubernetes [4] är en öppen plattform med syfte att hantera flera containrar som sträcker sig över ett kluster av server-datorer. Plattformen introducerades av Google men är idag ett fristående projekt som tillhandahålls av Cloud Native Computing Foundation [5].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: nginx-webserver
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          name: nginx-container
```

Källkodsexempel 2.1: Exempel på en enkel konfigurationsfil i ett Kubernetes-kluster. I detta exempel konfigureras klustret till att exekvera tre instanser av containeravbildningen `nginx`.

Kubernetes är ett modulärt system med syfte att tillhandahålla exekvering av container-avbildningar, nätverksanslutning mellan resurser i klustret, samt lagringssystem för container-avbildningar. Dessa erhålls genom användning av de tre specifikationerna Container Runtime Interface (CRI) [6], Container Network Interface (CNI) [7] respektive Container Storage Interface (CSI) [8].

Klustret konfigureras genom användning av deklarativa konfigurationsfiler skrivna i serialiseringsspråket YAML [9]. Deklarativ konfiguration [10] definieras enligt Microsoft som filer vars funktion är att beskriva komponenter samt konfigurationer som en viss miljö kräver, utan att nödvändigtvis beskriva hur konfigurationen skall uppnås. Ett konfigurationsexempel återfinns i Källkodsexempel 2.1 där tre instanser av HTTP-servern `nginx` [11] konfigureras. Den deklarativa konfigurationsmodellen

möjliggör därmed användandet av samma konfigurationsfiler oavsett vilken implementation av CRI, CNI respektive CSI som används inom klustret.

2.2.1 Kubernetes audit-logg

En audit-logg är en kronologisk uppsättning data med information om de aktiviteter som har utförts i ett Kubernetes-kluster. Audit-loggfiler innehåller information av de händelser som utförts av användare samt applikationer i klustret [12]. Denna data kan sedan användas för att granska händelser i klustret. Detta är särskilt användbart när en viss applikation avviker från det förväntade beteendet, samt i det fall då en attack sker mot ett Kubernetes-kluster eftersom det då är möjligt att granska attackerarens aktivitet i klustret. Audit-loggfiler genereras i JSON lines-format [13], där varje rad definierar en specifik händelse i klustret.

2.3 Molntjänster

En molntjänst är ett generellt begrepp som innebär att mjukvaru- samt hårdvaruinfrastruktur säljs till kunder som prenumerationstjänster. Exempel på molntjänster riktade mot företag är hårdvaru-infrastruktur i form av virtuella datorer, databastjänster samt webbserver-tjänster. För privatpersoner finns exempelvis lagringstjänster såsom Microsoft OneDrive samt webb-applikationstjänster såsom Microsoft Office 365 som drivs av molntjänster.

Tillgängligheten av molntjänster har lett till ett paradigm-skifte inom teknologisektorn. Tidigare var det vanligt att varje enskilt företag, vars produkter nyttjade server-datorer, också tillhandahöll server-datorerna. Detta innebar att varje företag skapade arbetsmöjligheter för administration, inköp, samt underhåll av server-datorer. Företag kunde tillhandahålla hela byggnader tillägnade dessa server-datorer, så kallade server-hallar. Ur ett ekologiskt perspektiv innebar detta en enorm överanvändning av resurser, dels för att tillhandahålla server-hallar, dels för att tillhandahålla server-datorer samt den kompetens som krävs för att administrera dessa. Då det är vanligt att produkter och tjänster belastas i varierande utsträckning under ett dygn innebar det att företagen behövde köpa in server-datorer för att klara av den högsta möjliga belastningen av ett mjukvarusystem, när den genomsnittliga belastningen ofta var långt under den maximala belastningen.

Dessa problem löses av molntjänster då kompetens, server-hallar, server-datorer, samt underhåll av dessa tillhandahålls av en central entitet, nämligen det företag som erbjuder molntjänsten. Detta innebär att flera företag delar på de server-datorer som molntjänsten tillhandahåller. Molntjänster löser även problemet med en varie-

rande belastningsgrad för ett visst mjukvarusystem. Ett visst företag har möjlighet att dynamiskt prenumerera på fler eller färre instanser av en viss tjänst, vilket gör det möjligt att utnyttja precis den mängd resurser som krävs för en viss belastningsgrad. Detta innebär att molntjänster erbjuder företag ekonomiskt billigare tjänster som anpassas dynamiskt efter de behov företaget har. Vidare innebär detta i sin tur att varje enskilt företag nyttjar färre resurser, vilket i sin tur gynnar miljön.

Ur ett IT-säkerhetsperspektiv finns det både för- och nackdelar med molntjänster. Eftersom företagen som erbjuder molntjänster ofta är stora och internationella, finns det därmed en större kompetens inom företagen just för molntjänster. Detta inkluderar även kompetensen kring IT-säkerhet, vilket innebär att molntjänster kan anses ha en högre säkerhetsstandard än de server-datorer som tillhandahålls av enskilda företag. Trots detta innebär eventuella attacker mot centrala molntjänster en större säkerhetsrisk än attacker mot enskilda företags server-hallar. Detta eftersom attacker mot centrala molntjänster kan påverka tjänster och produkter som används av flera företag samtidigt, vilket innebär en större attack-vektor jämfört med mindre server-hallar.

2.3.1 Microsoft Azure

Den molntjänst Microsoft erbjuder går under namnet Microsoft Azure, hädanefter benämd MA. MA erbjuder över 200 tjänster och produkter. Bland de vanligt förekommande tjänsterna återfinns databastjänster, infrastrukturstjänster i form av virtuella maskiner, samt Azure Kubernetes Service (AKS) för att skapa Kuberneteskluster i MA [14]. I denna rapport nyttjas audit-loggfiler som genererats från MA AKS, samt en av de databastjänster som erbjuds, Cosmos DB. MA Cosmos DB [15] erbjuder mjukvaruutvecklingsgränssnitt som är kompatibla med öppna databashanteringssystem såsom PostgreSQL [16] och MongoDB [17].

2.4 MongoDB

MongoDB [18] grundades år 2007 av Dwight Merriman, Eliot Horowiz samt Kevin Ryan. Företaget erbjuder en databas-plattform, också den namngiven MongoDB, för semi-strukturerad data vars lagringsmekanism härstammar från dataformatet JavaScript Object Notation (JSON). Lagringsmekanismen är en binärkodad serialisering av JSON vars namn är Binary JavaScript Object Notation (BSON). BSON ärver flera egenskaper från JSON, bland andra möjligheten att lagra semi-strukturerad data. Precis som för JSON erbjuds användaren även möjligheten att validera BSON-datans struktur före lagring i databasen, via ett fördefinierat data-schema. Detta

schema kan förändras över tid, vilket innebär att strukturen på den data som lagras i MongoDB kan förändras över tid [19]. Detta gör MongoDB till ett flexibelt databassystem som passar bra till utveckling av prototyper av program eftersom databasens struktur kan förändras utan rekonstruktion av databasens tabeller, vilket är fallet för klassiska relations-databassystem. Förutom prototyper passar även MongoDB bra som databas till program vars data ej har en strikt struktur och program som nyttjar JSON-data, exempelvis ett REST-API [20] på en webbserver.

MongoDB är ett exempel på ett databashanteringssystem inom kategorin Not only SQL (NoSQL) [21]. Denna typ av databashanteringssystem introducerades för att adressera de svårigheter som uppstår med relationsdatabaser såsom SQL-databaser. Förbättringarna som implementerades i NoSQL-databaser var främst ökad skalbarhet, förenklade utvecklingsgränssnitt för mjukvaruutvecklare, flexibiliteten hos dataschemat, samt förbättrad prestanda hos databasuttryck. MongoDB adresserar även dessa svårigheter, vilket resulterar i ett databashanteringssystem som passar väl till mjukvaror som arbetar med en stor mängd data som sträcker sig över flera databasservrar. Det passar även bra för prototyper där dataschemat ändras frekvent, samt för nyare programmeringsspråk vilka har stöd för MongoDB genom klient-bibliotek som används för att interagera med databasen.

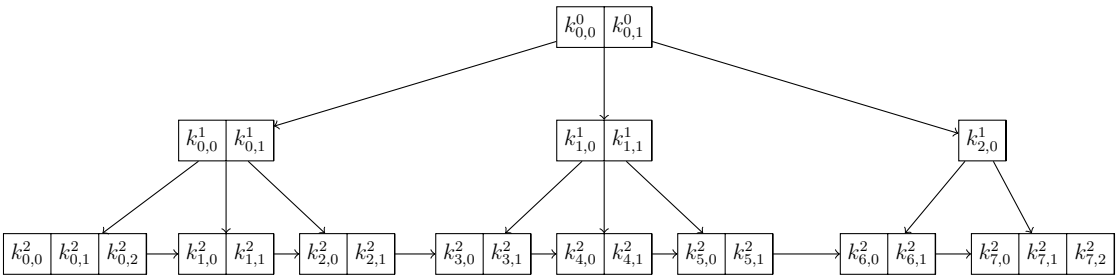
Likväl finns det tillämpningar då relationsdatabaser är bättre lämpad än NoSQL-databaser. Enlig Microsoft [22] är relationsdatabaser fördelaktiga exempelvis då komplexa uttryck krävs för att hämta rätt innehåll från databasen, då relationer mellan data uttrycks som `JOIN`-operationer mellan databastabeller samt då den data som lagras i databasen är väldigt strukturell.

I MongoDB finns motsvarigheter till tabeller och rader. De termer som används inom MongoDB är kollektion som motsvarar en tabell, och dokument som motsvarar en rad.

2.5 B+-träd

Ett B+-träd [23] är en balanserad datastruktur som vanligtvis används för indexering av data inom databashanteringsssystem [24]. B+-träd innehåller två typer av noder: Interna noder, vars innehåll är pekare till andra noder i trädet, och löv-noder, vars innehåll är den data, alternativt pekare till den data, som lagras i trädet. Båda nodtyperna innehåller även nycklar vars syfte är att användas som jämförelsevärden då operationer utförs på trädet. Nycklarna används även för att sortera trädets innehåll. Därmed är det ett krav för B+-träd att nycklarna består av datatyper som är sorterbara. B+-träd definierar termen ordningstal, q, som det maximala antalet nyckel-värdespar i lövnoder, samt det maximala antalet pekare till noder i en intern nod. Genom att tillåta ett ordningstal $q > 2$ har B+-träd generellt en lägre höjd än traditionella binära sökträd, vilket är fördelaktigt vid sökning av data. Figur

2.1 illustrerar ett generellt B+-träd med ordningstal $q = 3$. Då trädets nycklar är sorterade gäller att $k^p_{i,m} < k^p_{i,n}$ för $m < n$, samt $k^p_{i,M} < k^p_{i+1,0}$ där M representerar den sista nyckeln för nod i på nivå p.



Figur 2.1: Ett B+-träd med ordningstal $q = 3$. $k^p_{i,j}$ representerar den j:te nyckeln i nod i på nivå p i trädet.

Figur 2.1 visar även ytterligare en egenskap hos B+-träd: Lövnoder innehåller pekare till nästa lövnod, vilket förenklar sökningar i ett intervall av nycklar. Sökningen sker då endast på den minsta nyckeln i intervallet, sedan utnyttjas nycklarnas sorterade natur tillsammans med pekarna till nästa lövnod för att returnera data i det önskade intervallet. På så sätt kan ett B+-träd ses om en kombination av en ren trädstruktur och en länkad lista.

3
Metod

3.1 Arbetsprocess

Arbetet utförs i samarbete med, samt handledning från, Xenit AB. Xenit har möjlighet att bistå med produkter och tjänster från MA, samt den expertis som behövs för att använda molnplattformen på ett optimalt sätt. Samtal med Xenit kommer bedrivas kontinuerligt under hela arbetets process för att erhålla handledning och säkerställa att samtliga mål uppfylls. Mål, delmål, programkod samt övrigt material som delas med Xenit publiceras på plattformen GitHub [25] där Xenit kan följa projektets utveckling över tid. Vidare utförs arbetet med en agil metod, där varje sprint har en längd på två veckor. Detta innebär att nya delmål sätts upp var 14:e dag, med syfte att uppnå samt utvärdera dessa delmål 14 dagar senare. Den agila utvecklingsmetoden innebär att arbetet kan bedrivas på ett flexibelt sätt där nya delmål kan tillföras under arbetets gång. Likaså kan existerande delmål kontinuerligt omprioriteras alternativt uteslutas från arbetet.

3.2 Val av programspråk

Det programspråk som används för mjukvaruutveckling i detta arbete är Go. Go ses som det naturliga valet av programspråk för projekt som interagerar med Kubernetes. Detta eftersom Kubernetes är utvecklat i Go och exponerar ett officiellt mjukvaruutvecklingsgränssnitt (API) som kan användas utav tredje-partsprogram. Go använder ett så kallat modul-system vilket möjliggör utökning av programspråkets inbyggda funktionalitet genom inkludering av programkod från tredje-part. Detta har resulterat i ett växande ekosystem av Go-moduler såsom Kubernetes API [26] samt Microsoft Azure API [27].

3.3 Utvärdering av prestanda

I detta arbete kommer ett mjukvarusystem konstrueras för att läsa in, indexera, samt lagra data från audit-loggfiler i ett Kubernetes-kluster. Detta kommer utföras

genom utveckling av två varianter, som sedan jämförs med avseende på indexeringstid, tid för att hämta sökt data från audit-loggfiler, samt lagringsstorleken som krävs av lösningarna. Variant 1 kommer använda MongoDB som dess indexerings- och lagringslösning, medan variant 2 kommer använda ett B+-träd för indexering, medan audit-loggfilerna används som källa för data som hämtas.

De audit-loggfiler som används under prestandamätningarna är filer som genererats från Kubernetes-kluster som hanteras av Xenit AB. Filerna innehåller data från produktionskluster och har således inte genererats enbart för detta projekt. Filernas innehåll och storleksmängd motsvarar därmed de som kan förväntas av Kubernetes-kluster som används av diverse företag. De prestandamätningar som utförts använder tolv stycken audit-loggfiler med den totala storleksmängden 3.1 GB.

4

Systemkonstruktion

Detta avsnitt behandlar implementationen för det mjukvarusystem som konstruerats som en del av denna rapport. Nedanstående algoritmer är skrivna i pseudo-kod. Den faktiska implementationen i Go skiljer sig därmed något i utseende gentemot algoritmerna i detta kapitel.

4.1 Microsoft Azure

Mjukvarusystemet som har konstruerats använder data från MA. Närmare bestämt används JSON-data i audit-loggfiler som genererats från AKS.

Den tjänst som använts för lagring i MA kallas bloblagring. Bloblagring [28] är Microsofts benämning för objektlagring. IBM [29] definierar objektlagring som en arkitektur för datalagring med syfte att hantera stora mängder ostrukturerade data. Den data som hanteras lämpas ej för lagring i en relationsdatabas, med anledning av dess ostrukturerade natur. I denna arkitektur sägs varje datafil vara ett objekt. Vidare beskriver IBM att objektlagring innehar en platt struktur. Med detta menas att det ej finns en mappstruktur såsom i traditionella lagringssystem. Varje objekt definieras av dess unika identifieringsnummer, tillsammans med dess metadata. Denna information är nog för att lokalisera, respektive hämta ett objekt i ett objektlagringssystem.

Algoritm 1 Nedladdning av blobfiler från lagringskonto

```
1: S                                                  ▷ Microsoft lagringskonto
2: blobs ← [ ]                                        ▷ Tomt fält
3: for all c ∈ CONTAINERS(S) do
4:     blobs ← APPEND(blobs, GETBLOBS(c))
5: end for
6:
7: for all b ∈ blobs do
8:     DOWNLOAD(b)
9: end for
```

Innan de konstruerade mjukvarusystemen beskrivs definieras först relaterade termer. Samtliga termers definitioner återfinns i Microsofts dokumentation för bloblagring [28] respektive lagringskonto [30]. Objekten som lagras i MA, lagras i en lagringscontainer. En lagringscontainer innehåller en uppsättning av objekt, och kan jämföras med kataloger i traditionella filsystem. En lagringscontainer möjliggör därmed isolering av uppsättningar av objekt. Lagringscontainrar befinner sig i ett lagringskonto. Ett lagringskonto definierar ett unikt namnområde, som används för att tillgängliggöra lagringscontainrar via HTTP eller HTTPS. Lagringskontots namn blir därmed en del av den slutpunkt som används för att interagera med de objekt som existerar i lagringskontot.

Microsoft tillhandahåller flera API:er för att interagera med olika komponenter i MA från programmeringsspråket Go [27]. Ett av dessa är ett API för att interagera med bloblagring [31]. Genom detta API är det möjligt att läsa audit-loggfiler från AKS som lagras genom bloblagring i ett lagringskonto. Detta utförs enligt Algoritm 1.

4.2 Förenklat system

Initialt byggdes ett förenklat mjukvarusystem där MongoDB användes som lagrings- samt indexeringslösning. Detta eftersom MongoDB är byggd från grunden för att lagra semi-strukturerad data, där formatet på datan som lagras är mycket lik JSON. Det förenklade systemet liknar till stor del det primära verktyget som används för administration av Kubernetes-kluster, `kubectl` [32]. Eftersom `kubectl` är väldigt centralt för Kubernetes-administration var det även en god utgångspunkt för den utvecklade mjukvaran. Syftet med det förenklade systemet var att implementera en mjukvaruprototyp för att behandla data från ett lagringskonto och därmed göra data sökbar, samt presentera utvald data i ett kommandotolksgränssnitt. Prototypen kunde delas upp i två distinkta delar: Inläsning samt behandling av blobfiler, och hämtning av relevant data från MongoDB.

4.2.1 Inläsning samt behandling av data

Inläsning samt behandling av blobfiler implementerades genom ett distinkt kommando i programmet, `process`. När detta kommando används sker följande: Mjukvaran skapar ett antal trådar motsvarande antalet processorkärnor som återfinns på datorn där mjukvaran exekveras. Dessa trådar används för att parallellt ladda ner innehållet i flera filer. Trådarna skapas genom så kallade `goroutines`, en inbyggd mekanism i Go för just detta syfte.

Algoritm 2 Behandling av blobfiler

```
1: c_i                                              ▷ Channel (blobfiler)
2: for all b ∈ c_i do
3:     c_o                                          ▷ Channel (behandlad data)
4:     for all l ∈ LINES(b) do
5:         l_p ← PROCESS(l)
6:         c_o ← l_p
7:     end for
8: end for
```

Vidare använder varje tråd en och samma `channel`, vilken är en synkroniseringsmekanism för trådar i Go. En `channel` gör det möjligt att skicka till, samt ta emot data från trådar, utan explicit användning av synkroniseringsmekanismer såsom lås eller semaforer. Mjukvaran börjar med att hämta en lista av filer som existerar i lagringskontot. Sedan skickas var och en av filnamnen, genom en `channel`, till de tidigare nämnda trådarna, som läser in filnamnet på en fil från samma `channel`. Trådarna fortsätter genom att hämta innehållet i filerna, rad för rad. Varje rad behandlas genom att filtrera ut onödig data, och skickas sedan genom en ny `channel` till en tråd som ansvarar för att sätta in data i MongoDB, se Algoritm 2.

Algoritm 3 Temporär lagring av behandlad data

```
1: c_i                                              ▷ Channel (behandlade filer)
2: files ← [ ]                                      ▷ Tomt fält
3: for all b ∈ c_i do
4:     files ← APPEND(b)
5:     if LENGTH(files) == 20000 || ISCLOSED(c_i) then
6:         INSERTINTODB(files)
7:         files ← [ ]                              ▷ Töm fält
8:     end if
9: end for
```

En funktion som erbjuds via klienten som interagerar med MongoDB via Go är möjligheten att lägga till flera rader samtidigt i MongoDB och är namngiven INSERTMANY. För att utnyttja funktionen INSERTMANY väntar den tråd som tar emot behandlad data tills dess att 20000 behandlade rader har tagits emot. Först då läggs data till i MongoDB. Detta förfarande återfinns i Algoritm 3. Metoden som används för att hämta innehållet i blob-filer är alltså en variant av Algoritm 1, där implementationen av DOWNLOAD består av att skicka parametern b genom en `channel` till de trådar som används för nedladdning respektive behandling av data.

När data väl läggs till i MongoDB görs detta via hjälpfunktionen INSERTINTODB i Algoritm 3. Denna funktion används för att ytterligare behandla data, samt slutligen lägga till denna i MongoDB. Anledningen till att datan behöver behandlas ytterligare är för att mjukvarusystemet implementerar differentiell lagring av data, med målet att minimera mängden dataduplicering. Detta åstadkoms genom att dela

upp lagringen i två olika MongoDB-kollektioner. Den ena kollektionen innehåller den första instansen av en händelse för en viss resurs i Kubernetes-klustret. Hädanefter benämns denna kollektion `events`. Den andra kollektionen innehåller resterande händelser för objekt som existerar i `events`-kollektionen, i form av differensen mellan händelsen som skall lagras, och den som återfinns i `events`-kollektionen. Hädanefter benämns denna kollektion `diffs`.

Denna metod introducerar en viss svårighet vid insättning av nya data. Det finns tre

Algoritm 4 Insättning av data i MongoDB

```
1:  events                                   ▷ Fält med data som skall läggas till i MongoDB
2:  d_e ← [ ]                                ▷ Tomt fält, data som skall läggas till i events
3:  d_d ← [ ]                                ▷ Tomt fält, data som skall läggas till i diffs
4:
5:  for all e ∈ events do
6:      baseEvent                            ▷ Bas-händelse för händelse e
7:      metadata ← PARSEMETADATA(e)
8:
9:      if CONTAINS(d_e, metadata) then
10:         baseEvent ← GETEVENTFROMLIST(d_e, metadata)
11:         INSERTINTODB(files)
12:         files ← [ ]                      ▷ Töm fält
13:     else if MONGODBCONTAINS(metadata) then
14:         baseEvent ← GETEVENTFROMDB(metadata)   ▷ Hämta händelse från events-kollektionen
15:     end if
16:
17:     if NOTEMPTY(baseEvent) then
18:         diff ← GETDIFF(baseEvent, e)
19:         APPEND(d_d, diff)                ▷ Lägg till differens i d_d
20:     else
21:         APPEND(d_e, e)                   ▷ Lägg till händelse i d_e
22:     end if
23: end for
```

distinkta fall som kan inträffa vid insättning. Fall 1 består av att den händelse som skall läggas till i MongoDB är den första för den specifika resursen i Kubernetes-klustret, samt att denna är den enda händelse avseende just denna resurs i den uppsättning av händelser som skall läggas till i MongoDB. Detta fall är det lättaste att behandla, eftersom händelsen då helt enkelt skall läggas till i `events`-kollektionen i MongoDB.

Fall 2 ges av att den Kubernetes-resurs som relateras till den händelse som skall läggas till i MongoDB redan har ett dokument i `events`-kollektionen. I detta fall behöver detta dokument hämtas från `events`-kollektionen, följt av en beräkning av differensen mellan de två händelserna. Slutligen läggs differensen till i `diffs`-kollektionen.

Fall 3 fås då den Kubernetes-resurs som relateras till den händelse som skall läggas till i MongoDB ej är den enda händelse för den specifika Kubernetes-resursen i uppsättningen av händelser som skall läggas till i MongoDB. Det kan tyckas att detta ej behöver hanteras som ett separat fall, ty det bör kunna reduceras till fall 2. Detta gäller dock endast då händelser läggs till en och en i MongoDB. Som tidigare nämnts används funktionen `insertMany` från MongoDB-klienten för att lägga till data i databasen. För att kunna använda denna funktion i kombination med en differenslagring blir det därmed nödvändigt att använda två listor: En för händelser som skall läggas till i `events`-kollektionen och en för händelser som skall läggas till i `diffs`-kollektionen. Därmed kan fall 3 ej längre reduceras till fall 2.

Lösningen på fall 3 blir därmed att söka efter en händelse som motsvarar den aktuella Kubernetes-resursen i listan innehållandes händelser som skall läggas till i `events`-kollektionen. I det fall då en sådan händelse återfinns, används denna för att finna differensen för händelsen som skall läggas till i MongoDB, annars utförs en sökning efter händelsen i `events`-kollektionen i MongoDB. Algoritm 4 visar hela metoden för fall 1-3 i detalj.

4.2.2 Hämtning av data

Datahämtningen implementerades genom två distinkta kommandon, `get` respektive `follow`, med syftet att försöka efterlikna de kommandon som återfinns i `kubectl`.

Kommandot `get` används för att visa de senaste händelserna i Kubernetes-klustret för en viss resurstyp i klustret. Detta innebär att två typer av hämtningar från MongoDB behöver genomföras: En hämtning från `events`-kollektionen för att finna samtliga Kubernetes-resurser av en viss kategori. Sedan behöver `diffs`-kollektionen genomsökas för att finna den senaste händelsen för en viss Kubernetes-resurs. Denna sökning sker därmed ej en gång per resurs-typ, utan en gång per resurs som returneras i sökningen av `events`-kollektionen.

Logiken för denna hämtningsmetod återfinns i algoritm 5. Slutligen slås varje differens samman med dess motsvarande bas-händelse från `events`-kollektionen, för att på så vis återskapa den senaste händelsen för en viss resurs i Kubernetes-klustret. Även detta genomförs för samtliga resultat i `diffs`-kollektionen.

Kommandot `follow` hämtar samtliga händelser för en specifik resurs i Kubernetes-klustret. Metoden för att hämtning av data är mycket lik den för `get`-kommandot.

Algoritm 5 Hämtning av senaste händelse för multipla resurser

```
1: filter                          ▷ Sträng, används för att hämta rätt data från MongoDB
2: events ← getBaseEvents(filter)  ▷ Hämta händelser från events-kollektionen i MongoDB
3: d_e ← [ ]                       ▷ Tomt fält, fylls med returvärden
4:
5: for all e ∈ events do
6:     latest                      ▷ Senaste händelsen, initialt tom
7:     if mongoDbContainsDiff(metadata) then
8:         diff ← getLatestDiff(metadata)   ▷ Hämta senaste händelse från diffs-kollektionen
9:         latest ← mergeEvents(e, diff)
10:    else
11:        latest ← e
12:    end if
13:    append(d_e, latest)         ▷ Lägg till händelse till returvärden
14: end for
```

Sökningen i `events`-kollektionen är annorlunda, ty endast en händelse skall hämtas ur denna, nämligen händelsen för den sökta resursen. När denna återfunnits hämtas sedan samtliga dokument i `diffs`-kollektionen för den sökta resursen. Metoden för att hämta denna data från MongoDB återfinns i Algoritm 6.

Algoritm 6 Hämtning av samtliga händelser för enskild resurs

```
1: filter                          ▷ Sträng, används för att hämta rätt data från MongoDB
2: baseEvent ← getBaseEvent(filter)  ▷ Hämta en händelse från events-kollektionen i MongoDB
3: d_e ← [ ]                       ▷ Tomt fält, fylls med returvärden
4: diffs ← getDiffs(filter)        ▷ Hämta differenser från diffs-kollektionen i MongoDB
5:
6: for all d ∈ diffs do
7:     event ← mergeEvents(baseEvent, d)
8:     append(d_e, event)          ▷ Lägg till händelse till returvärden
9: end for
```

Slutligen slås dessa differenser samman med bas-händelsen från `events`-kollektionen, på samma sätt som för `get`-kommandot.

4.3 Fullständigt system

Det förenklade systemet synliggjorde ett tydligt problem: Den data som hämtas från ett lagringskonto i MA lagras ytterligare en gång i MongoDB. Därmed duplicerades data, vilket resulterade i oönskad överanvändning av resurser i MA. Därmed utvecklades ett resurssnålare system, där MongoDB ersattes utav ett filbaserat index med målsättning att indexet skulle lagras i ett lagringskonto tillsammans med audit-loggfiler från AKS. Detta mjukvarusystem, hädanefter benämt det fullständiga systemet, baseras på det förenklade systemet och delar därmed även källkod med detta. Alltså ärver det fullständiga mjukvarusystemet det förenklade mjukvarusystemets egenskaper, exempelvis dess kommandotolksgränssnitt. Mer exakt implementerar det fullständiga systemet nya variationer av Algoritm 3 till och med Algoritm 6.

4.3.1 Indexering av data

Som tidigare nämnts resulterade det förenklade systemet i duplicering av data. En möjlig lösning hade varit att endast implementera ett index i MongoDB, där varje rad innehåller en pekare till en JSON-fil, samt positionen i denna fil vars innehåll är den sökta datan. Istället valdes en lösning baserad på ett index där lagringsmekanismen är ett lagringskonto. Detta eftersom ett sådant index skulle resultera i en högre kostnadseffektivitet i jämförelse med en lösning baserad på MongoDB.

Det fullständiga systemet implementerar alltså ett filbaserat index. Indexet använder ett B+-träd som bakomliggande datastruktur, som lagras permanent på disk genom ett egenutvecklat textbaserat filformat. Indexet är endast en metod för att öka sökbarheten i de filer som lagras i MA. Därmed lagras endast extra metadata berörande blobfilerna från MA, vilket resulterar i ett system där lagrings- såväl som kostnadseffektiviteten är högre än i det förenklade systemet där MongoDB används.

4.3.2 Filformat

Filformatet som används för data-indexet är textbaserat. Detta eftersom den data som indexeras har en semi-strukturerad natur. Speciellt har nycklarna en varierande längd, där en maximal längd ej kan definieras utan att införa begränsningar för den data som indexeras. Varje fil lagrar en enskild nod i ett B+-träd. Filformatet kan uttryckas i termer av fyra sektioner. En illustration över filformatets utseende återfinns i Källkodsexempel 4.1.

```
true
leases,kube-system,kube-scheduler,2023-03-05T08:55:55.645299Z
↪  leases,kube-system,kube-scheduler,2023-03-05T08:55:57.672160Z
data/audit.json,283633435,1497 data/audit.json,284195079,1497
39818241-2084-42fd-a685-ca27ba02a93d
c334192c-3cec-4c8a-a2b8-fd135dfe5508
```

Källkodsexempel 4.1: Filformatet för B+-trädet. Rad 1 visar att denna nod är en lövnod. Rad 2 innehåller nodens nycklar separerade med blanksteg. Rad 3 innehåller nodens värden, också dessa separerade med blanksteg. Rad 4 innehåller UUID-strängen för den föregående lövnoden i trädet. Rad 5 innehåller UUID-strängen för nästa nod i trädet.

Sektion 1 består av nodens metadata. Denna metadata innehåller endast en boolesk flagga vars syfte är att indikera om noden är en lövnod eller en intern nod. Sektion 2 innehåller nodens nycklar, där nycklarna separeras med ett blanksteg. Detta betyder att nycklarna antas ej innehålla blanksteg. Innehållet för sektion 3 varierar beroende av värdet av den booleska flagga som definierats i sektion 1. I det fall då den aktuella noden är en lövnod innehåller den tredje sektionen nodens värden, också dessa separerade med ett blanksteg. Värdena i en lövnod består av en sökväg till filen där den önskade informationen kan hämtas, följt av startpositionen(given i antal bytes) där datan återfinns, följt av storleken(given i antal bytes) för datan. Är den aktuella noden en intern nod innehåller istället sektion 3 pekare till ytterligare noder, i form av nodernas identifierare. Även i detta fall är värdena separerade med blanksteg. Sektion 4 sträcker sig över rad 4 respektive 5 i filen. Sektion 4 existerar endast då noden är en lövnod. Rad 4 innehåller då UUID-strängen för föregående nod i trädet, medan rad 5 innehåller UUID-strängen för nästkommande nod i trädet. Nodernas identifierare definieras som en Universally Unique Identifier-sträng(UUID) [33]. Slutligen namnges varje fil enligt dess identifierare.

```
100 0.0.1
03e4f2b7-114c-488e-8a1c-9efa7ae98641
```

Källkodsexempel 4.2: Filformatet för den speciella filen `metadata.db`. Rad 1 innehåller B+-trädets ordningstal följt av en versionssträng. Rad 2 innehåller UUID-strängen för trädets rot.

För att kunna nyttja dessa filer finns även en fil med ett konstant namn, `metadata.db`, där metadata för B+-trädet definieras, såsom vilken nod som motsvarar roten av B+-trädet. Se Källkodsexempel 4.2, där formatet för denna fil illustreras.

4.3.3 Insättning av data

Insättning av data sker helt enligt algoritmen för ett B+-träd. Insättningen kan delas upp i två distinkta fall.

Algoritm 7 Insättning i B+-träd utan delning

```
1:  function addDataToNode(d)
2:      d                                  ▷ Nyckel-värdespar som skall läggas till i trädet
3:      key ← key(d)
4:      value ← value(d)
5:      node ← findLeafNode(d)             ▷ Finn rätt lövnod i trädet
6:      isInserted ← false
7:      newKeys ← [ ]   ▷ Initialt tomt fält, lagrar nycklar efter insättning av data
8:      newValues ← [ ] ▷ Initialt tomt fält, lagrar värden efter insättning av data
9:
10:     for all k ∈ keys(node) do
11:         v ← getValue(k)
12:         append(newKeys, k)
13:         append(newValues, v)
14:
15:         if key < k and !isInserted then
16:             append(newKeys, key)
17:             append(newValues, value)
18:             isInserted ← true
19:         end if
20:     end for
21:
22:     setKeys(node, newKeys)
23:     setValues(node, newValues)
24:     storeNode(node)          ▷ Skriv den uppdaterade noden till lagringsmedia
25: end function
```

Fall 1 motsvarar situationen då lövnoden där data ska läggas till innehåller färre nyckel-värdespar än B+-trädets ordningstal, se Algoritm 7. Detta fall påminner om algoritmen för ett binärt sökträd, där noderna i trädet söks igenom från trädets rot, sedan nedåt i trädet, nivå för nivå.

Som nämndes i avsnitt 2.5 innehåller ett B+-träd interna noder och lövnoder. Lövnoderna är de noder som befinner sig längst ned i trädet, och det är dessa som innehåller den data som lagras i trädet. För att sätta in data i trädet utförs därmed en sökning i trädet för att finna den lövnod där datan ska lagras. Denna sökning beskrivs mer ingående i avsnitt 4.3.4. Eftersom datan är sorterad i lövnoden utförs en linjär sökning i noden för att finna positionen där datan ska läggas till, så att

Algoritm 8 Delning av nod i B+-träd

```
1:  function splitNode(node)
2:      node                                          ▷ Nod som skall delas
3:      i_s                                           ▷ Index där noden skall delas
4:      left          ▷ Vänstra noden efter delning. Ärver UUID från node
5:      right                ▷ Högra noden efter delning. Tilldelas nytt UUID
6:
7:      if isLeaf(node) then
8:          i_s ← ⌈length(keys(node)) / 2⌉
9:      else
10:         i_s ← ⌈length(keys(node)) / 2 − 1⌉
11:     end if
12:
13:     setKeys(left, 0, i_s)
14:     setValues(left, 0, i_s)
15:     setKeys(right, i_s, length(keys(node)))
16:     setValues(left, i_s, length(keys(node)))
17:
18:     if isLeaf(node) then
19:         setPrev(right, left)
20:         setNext(left, right)
21:         prev ← getPrev(node)
22:         next ← getNext(node)
23:
24:         if notEmpty(prev) then
25:             setPrev(left, prev)
26:         end if
27:
28:         if notEmpty(next) then
29:             setNext(right, next)
30:         end if
31:     end if
32:
33:     return (left, right, key(node, i_s))
34: end function
```

lövnoden är sorterad även efter insättning av nya data.

Fall 2 motsvarar situationen då lövnoden där den data som skall läggas till innehåller q nyckel-värdespar efter insättning, där q motsvarar trädets ordningstal. Detta fall delas upp i två separata algoritmer. Algoritm 8 beskriver hur delning av en nod utförs. Noden delas i två ungefär lika stora delar, följt av att pekare till föregående respektive nästa lövnod i trädet uppdateras. Denna delningsalgoritm används sedan när en nod skall delas efter insättning av data, se Algoritm 9. Tillvägagångssättet

Algoritm 9 Insättning i B+-träd med delning

```
1: function splitAndInsertNode(node)    ▷ Dela en node och lägg till i trädet
2:     if length(keys(node)) < q then
3:         return
4:     end if
5:
6:     left, right, key ← splitNode(node)          ▷ Dela nod enligt Algoritm 8
7:     storeNode(left)                              ▷ Lagra node efter delning
8:     parent ← findParentNode(left)                ▷ Finn överordnad nod i trädet
9:
10:    if isEmpty(parent) then      ▷ Ingen överordnad nod, node är trädets rot
11:        insertNewRoot(key, left, right)      ▷ Lägg till ny rot med nyckeln k
   samt de två noderna left resp. right
12:        return
13:    end if
14:
15:    isInserted ← false
16:    newKeys ← [ ]    ▷ Initialt tomt fält, lagrar nycklar efter insättning av data
17:    newChildren ← [ ]      ▷ Initialt tomt fält, lagrar underordnade noder efter
   insättning
18:
19:    for all k ∈ keys(node) do
20:        c ← getLeftChild(k)
21:        append(newKeys, k)
22:        append(newChildren, c)
23:
24:        if key < k and ! isInserted then
25:            append(newKeys, key)
26:            append(newValues, value)
27:            isInserted ← true
28:        end if
29:    end for
30:
31:    c ← getLastChild(node)
32:    append(newChildren, c)
33:    setKeys(node, newKeys)
34:    setValues(node, newValues)
35:    storeNode(node)          ▷ Skriv den uppdaterade noden till lagringsmedia
36:    splitAndInsertNode(parent)
37: end function
```

för insättning är detsamma som det i fall 1. Skillnaden är att noden efter insättning av data även delas. Detta kan leda till att fler noder i trädet behöver delas, varför algoritmen exekveras rekursivt på den överordnade noden.

Algoritm 10 Insättning av data i B+-träd

```
1: function insertIntoIndex(data)
2:     metadata ← getMetadata(data)
3:     addDataToNode(metadata)
4:     node ← findLeafNode(metadata)
5:     splitAndInsertNode(node)
6: end function
```

Algoritmen avslutas då en nod återfunnits som ej behöver delas, alternativt då en ny rot satts in i trädet efter att den existerande roten delats i två noder. Algoritmen för insättning i ett B+-träd blir därmed kombinationen av Algoritm 7 samt Algoritm 9. Denna kombination återfinns i Algoritm 10, och motsvarar Algoritm 4 för det förenklade systemet.

En skillnad mot det förenklade systemet är att det filbaserade indexet ej har en motsvarighet till funktionen insertMany som återfinns i MongoDB-klienten för Go. Istället läggs data till i indexet för en händelse åt gången. Motsvarigheten till Algoritm 3 för det filbaserade indexet återfinns därmed i Algoritm 11.

Algoritm 11 Mottagning av behandlade filer

```
1: c_i                                   ▷ Channel (behandlade filer)
2: for all b ∈ c_i do
3:     insertIntoIndex(b)
4: end for
```

4.3.4 Sökning av data

Sökning i ett B+-träd efterliknar sökningsalgoritmen i ett binärt sökträd. I ett binärt sökträd har varje nod endast en nyckel samt två underordnade noder, vilket resulterar i en enkel sökningsalgoritm. I ett B+-träd generaliseras algoritmen till att tillåta n nycklar samt $n + 1$ underordnade noder i varje nod, där $n < q$, där B+-trädets ordningstal benämns q, se Algoritm 12.

Eftersom nycklarna, och därmed de underordnade noderna, är sorterade, genomförs sökning i varje nod genom att finna den första nyckel som är större än nyckeln som eftersöks. När denna nyckel funnits på index i, väljs den underordnade noden med index i för nästa iteration av sökningsalgoritmen. I det fall då ingen nyckel funnits

Algoritm 12 Sökning i B+-träd

```
1:  function findLeafNode(d)
2:      d                                          ▷ Eftersökt nyckel-värdespar
3:      key ← key(d)
4:      node ← getRoot()          ▷ Returvärde. Påbörja sökningen i trädets rot
5:
6:      for all k ∈ keys(node) do
7:          if isLeaf(node) then          ▷ Lövnoder finns på sista nivån i trädet
8:              break
9:          end if
10:
11:         if key < k then                              ▷ Skifta till nästa nod-nivå
12:             node ← getLeftChild(node, k)
13:         else if isLastKey(k) then ▷ Interna noder har alltid k + 1 värden, där
    k är antalet nycklar
14:             node ← getLastChild(node)
15:         end if
16:     end for
17:
18:     return node
19: end function
```

som är större än den eftersökta nyckeln, väljs den sista underordnade noden med index $n + 1$, där n är antalet nycklar i noden. Algoritmen avslutas då en lövnod funnits, ty det är i lövnoderna data existerar i ett B+-träd.

Som sågs i det förenklade systemet är det av intresse att hämta data för flera händelser. Algoritm 12 visar endast hur en specifik händelse kan hämtas från trädet. För att hämta flera händelser från trädet behövs därmed en metod för att söka och hämta data motsvarande ett intervall av nycklar från trädet. I ett B+-träd blir detta trivialt eftersom nycklarna i trädet är sorterade. Varje lövnod innehåller även en pekare till nästkommande respektive föregående lövnod i trädet. Genom att låta nycklarna i trädet innehålla information från audit-händelsen, närmare bestämt resursens typ, namnområde, namn samt tidsstämpeln för händelsen, och sedan ordna dessa i ordning från minst till mest specifik, blir det därmed möjligt att hämta all relevant information som krävs av den utvecklade mjukvaran från B+-trädet.

För att göra det möjligt att söka i ett intervall i trädet behövs en hjälpfunktion för att finna den sista nyckeln med ett visst prefix. Funktionen findLeafNode kan användas för att finna den första lövnoden för ett visst prefix. Därmed införs hjälpfunktionen GetLastKeyWithPrefix. Funktionen inkrementerar sista karaktären i prefixet, finner den första lövnoden för det inkrementerade prefixet, och söker slutligen bakåt i trädet för att finna den sista nyckeln med det sökta prefixet.

Algoritm 13 Sökning av sista nyckel med specifikt prefix i B+-träd

```
1:  function GetLastKeyWithPrefix(prefix)
2:      key                                   ▷ Tom sträng, funktionens returvärde
3:      searchKey ← incrementLastChar(prefix)                    ▷ Söksträng
4:      node ← findLeafNode(searchKey)
5:      keys ← keys(node)
6:      i ← length(keys) − 1
7:
8:      for i ≥ 0 do
9:          k ← keys[i]
10:         if hasPrefix(k, prefix) then
11:             keys ← k
12:             break
13:         else
14:             i ← i − 1
15:         end if
16:
17:         if i == 0 then
18:             if isEmpty(getPrevNode(node)) then
19:                 break
20:             end if
21:
22:             node ← getPrevNode(node)
23:             keys ← keys(node)
24:             i ← length(keys) − 1
25:         end if
26:     end for
27:
28:     return key
29: end function
```

Algoritm 13 visar detta förfarande. Slutligen implementeras intervallsökningen, se Algoritm 14, genom att finna lövnoden för start-nyckeln i intervallet. Sedan söks lövnoderna igenom tills dess att en nyckel funnits som är större än slut-nyckeln i intervallet, alternativ tills slutet av den sista lövnoden i trädet nås.

Notera att Algoritm 13 samt Algoritm 14 ej är exakta motsvarigheter av Algoritm 5 respektive Algoritm 6. Mjukvarusystemet använder Algoritm 13 samt Algoritm 14 för att söka i trädet. Sedan filtreras den återfunna datan ut så att endast de relevanta händelserna hämtas från audit-loggfilerna, och slutligen presenteras för användaren. Denna filtrering beskrivs ej då den inte är relaterad till funktionaliteten för ett B+-träd, samt eftersom filtreringen i sig är trivial att implementera.

Algoritm 14 Intervallsökning i B+-träd

```
1:  function SearchRange(fromKey, toKey)
2:      returnValues ← [ ]                          ▷ Tomt fält, returvärden
3:      node ← findLeafNode(fromKey)
4:      keys ← keys(node)
5:      values ← values(node)
6:      i ← length(keys) − 1
7:
8:      for i < length(keys) do
9:          k ← keys[i]
10:         if k < fromKey then
11:             i ← i + 1
12:             continue
13:         end if
14:
15:         if k > toKey then
16:             break
17:         end if
18:
19:         returnValues ← append(returnValues, values[i])
20:
21:         if i == length(keys) − 1 then
22:             if isEmpty(getNextNode(node)) then
23:                 break
24:             end if
25:
26:             node ← getNextNode(node)
27:             keys ← keys(node)
28:             values ← values(node)
29:             i ← 0
30:         end if
31:     end for
32: end function
```

5
Resultat

I detta avsnitt presenteras de prestandamätningar som utförts. Samtliga mätningar har utförts på en Macbook Pro 16 tum (årsmodell 2021) med 32GB arbetsminne, Apple M1 Max-processor samt en intern SSD med 1TB lagringskapacitet. Dessutom har datorn använt en Cat6-nätverkskabel, som anslutits till ett internt nätverk med hastighet 1 Gbit/s, som i sin tur anslutits vidare till en bredbandsuppkoppling med den maximala nedladdningshastigheten 300Mbit/s. I de fall där MongoDB använts, har även MongoDB varit aktiv på samma dator som använts för prestandamätningarna, i form av en Docker-container.

Samtliga audit-loggfiler som använts under dessa mätningar är data som erhållits genom Xenit AB.

De mätningar som utförts jämför prestandan hos B+-trädet respektive MongoDB, där B+-trädet delas upp i två separata fall. Det ena fallet är hämtning av sökt data i audit-loggfiler lagrade i ett lagringskonto i MA. Detta resulterar i att data hämtas från MA, vilket inte är fallet för lösningen där MongoDB används. MongoDB-lösningen används som en kombinerad indexerings- samt lagringsmetod, vilket innebär att all information hämtas direkt från MongoDB efter att datan har indexerats. För att förhindra att nätverkslatens och begränsad nedladdningshastighet påverkar de erhållna mätvärdena introducerades därför en tredje testmetod. Även denna metod använder B+-trädet som index. Skillnaden är att B+-trädet indexerar lokala filer, vilket resulterar i att datan i slutändan läses från lokala filer.

Mätningarna är uppdelade i fyra delar. I figurerna förekommer förkortningarna L respektive MA. Dessa står för lokala filer, respektive filer hämtade direkt från Microsoft Azure.

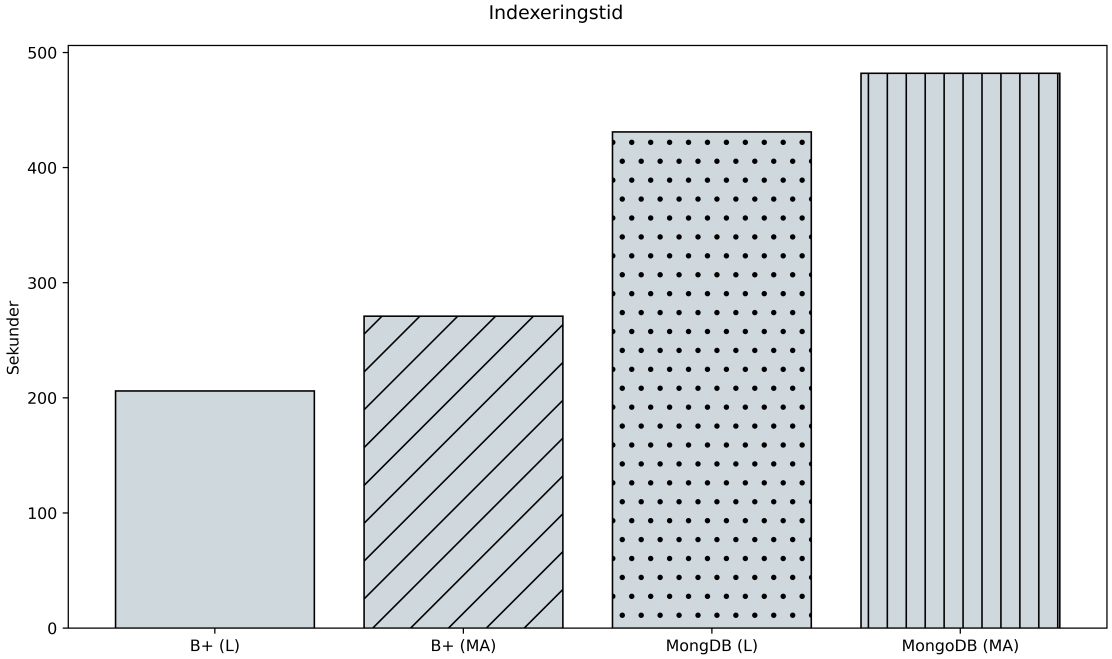
Den första delen visar tiden det tar att indexera tolv stycken audit-loggfiler med den ackumulerade storleken 3.1GB, se Figur 5.1.

Den andra delen av mätningarna visar tiden det tar att hämta sökt data i audit-loggfiler för de olika lösningarna, se Figur 5.2.

Den tredje delen av mätningarna jämför lagringsstorleken som används av respektive implementation, se Figur 5.4.

Den fjärde delen av mätningarna innehåller lösningarnas respektive kostnad, se Fi-

gur 5.5.

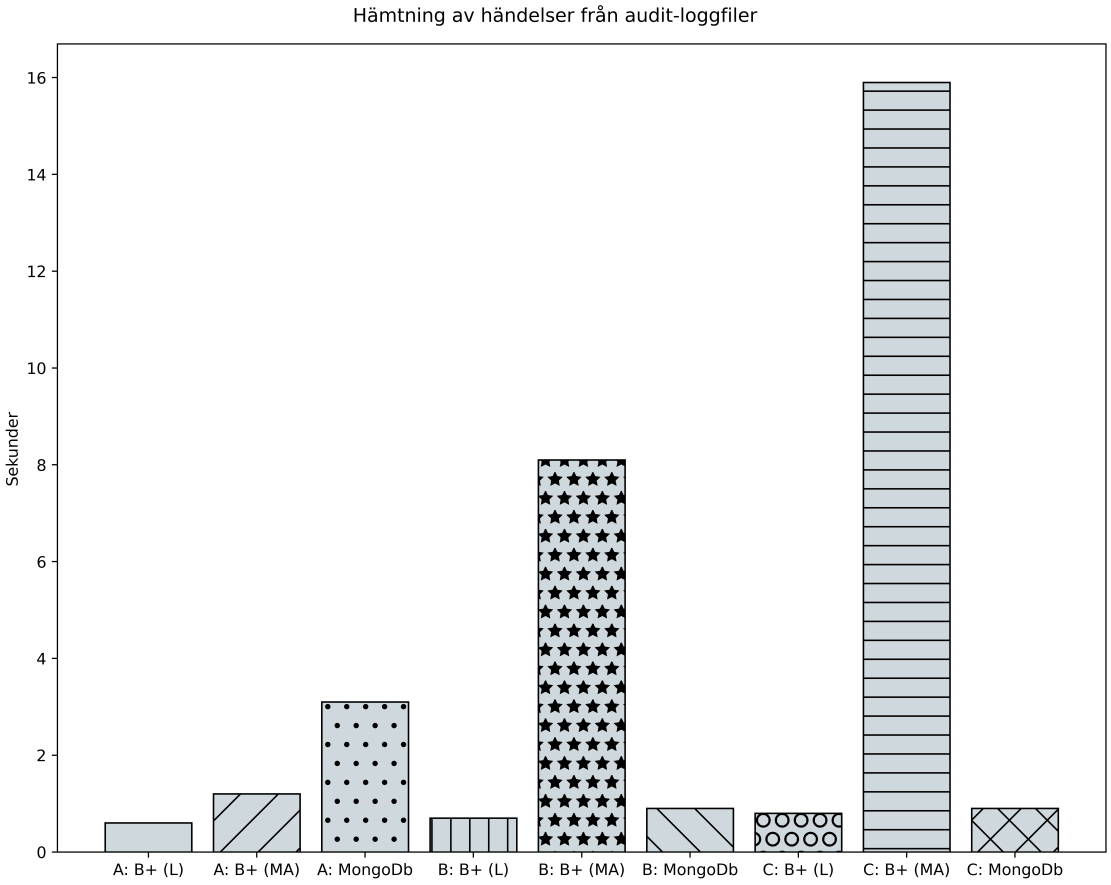


Figur 5.1: Indexeringstid för de olika lösningarna. L motsvarar lokala filer, MA motsvarar filer direkt från Microsoft Azure.

Figur 5.1 visar att den kortaste indexeringstiden fås då B+-trädet används samt indexerar lokala filer på datorn. Det ses också att den längsta indexeringstiden fås då MongoDB används samt då filerna hämtas från MA. Värt att notera är att användandet av MongoDB resulterar i en längre indexeringstid jämfört med B+-trädet i samtliga fall. Även då B+-trädet indexerar filer direkt från ett lagringskonto i MA är indexeringstiden snabbare än då MongoDB indexerar lokala filer.

I Figur 5.2 ses det att differensen mellan den längsta respektive kortaste exekveringstiden är utav storleksordningen 10 s. Då den senaste händelsen hämtas för objekt av samma typ (se Figur 5.2, staplar med prefix A) ger MongoDB den längsta exekveringstiden. Däremot, när flera händelser hämtas för samma typ av objekt (se Figur 5.2, staplar med prefix B respektive C) ger B+-trädet i kombination med filer lagrade i MA, den längsta exekveringstiden. I samtliga fall ger B+-trädet i kombination med lokalt lagrade filer den kortaste exekveringstiden. Detta tydliggörs i Figur 5.3.

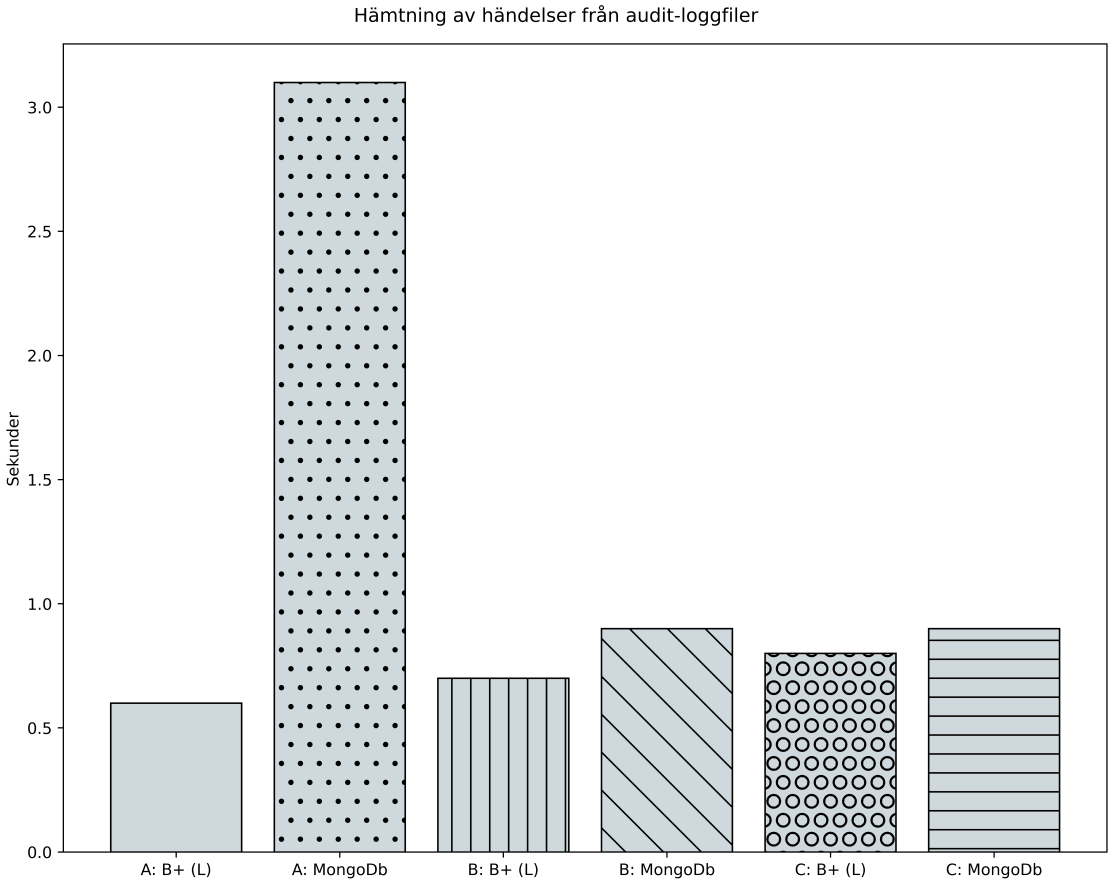
Jämförelser av de olika lösningarnas lagringsstorlek återfinns i Figur 5.4. För B+-trädet innebär lagringsstorleken den ackumulerade storleken av samtliga indexeringsfiler. För MongoDB-implementationen innebär lagringsstorleken summan av lagringen som krävs för att lagra datan, samt de interna index i MongoDB-databasen. Dessa erhålls genom ett MongoDB-skript som återfinns i Appendix A.1.



Figur 5.2: Hämtningstid för diverse data. A visar exekveringstiden för kommandot `get nodes`, vilken hämtar den senaste händelsen för 14 noder. B motsvarar kommandot `follow node`, följt av en specifik nod i Kubernetes-klustret, vilket hämtar 1446 händelser för en specifik nod. C motsvarar kommandot `follow service`, följt av en specifik tjänst i Kubernetes-klustret, vilket hämtar 3272 händelser för en specifik tjänst.

Lagringsstorleken för B+-trädet är cirka 4 % av den ackumulerade storleken för audit-loggfilerna. MongoDB-lösningen resulterar i lagringsstorleken 8.9 %. Värt att notera är att lagringsstorleken för B+-trädet ej är densamma då lokala filer indexeras respektive då filer från MA indexeras. Anledningen till detta är att sökvägen till korrekt fil lagras som en del av värdet i lövnoderna. Då lokala filer används är sökvägen till filerna kortare än sökvägen till motsvarande fil i MA, vilket resulterar i en mindre storleksskillnad.

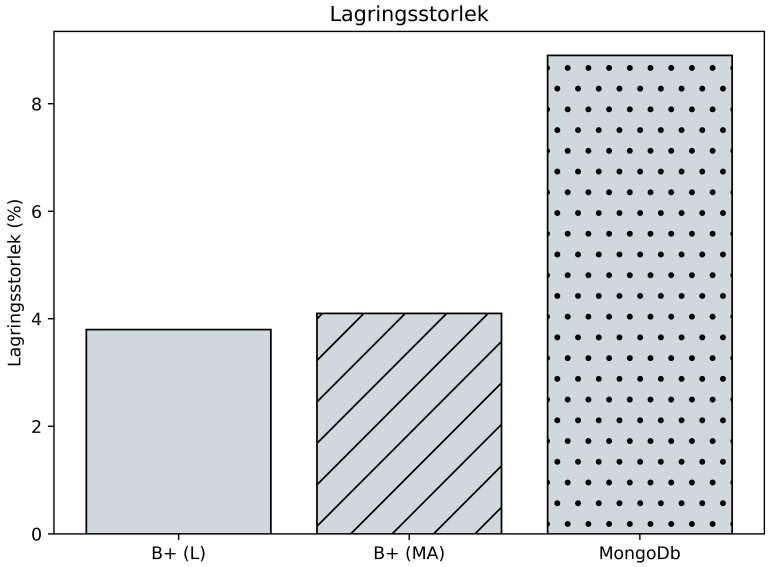
De olika lösningarnas respektive kostnad jämförs i Figur 5.5. Dessa värden fås genom Microsofts kostnadskalkylator [34]. Ett lagringskonto prissätts genom mängden data som lagras, samt de läs- och skrivåtgärder som utförs på lagringskontot. Den data som använts vid indexering motsvarar 12 timmar data. Mängden data som använts vid uppskattning av kostnaden är därmed de värden som erhållits av 12 timmars data, multiplicerats med 60, för att motsvara 30 dagar, 24 timmar per dygn. Antalet skrivåtgärder fås av antalet filer som skall skrivas till lagringskontot. Med 12 timmar



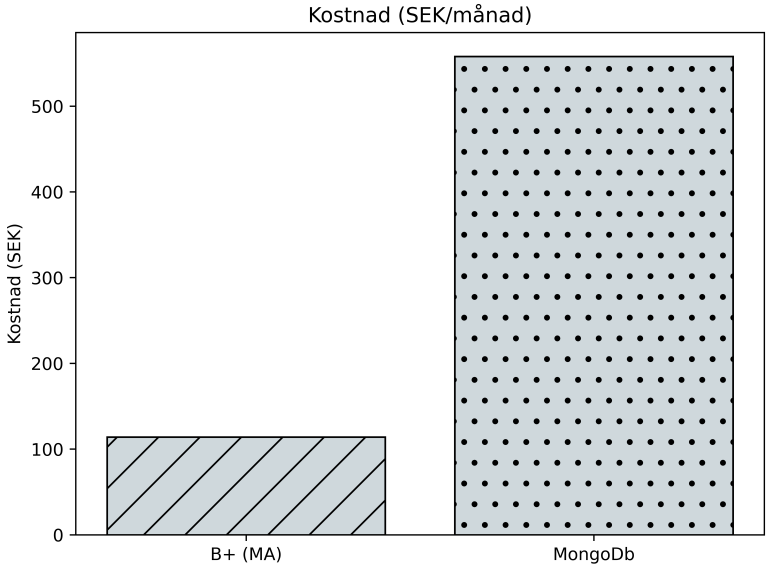
Figur 5.3: Figuren visar tiden det tar att hämta diverse data. Uppsättningen är densamma som i Figur 5.2. I denna figur visas endast den fulländade lösningen, där B+-trädet indexerar lokala filer, samt den förenklade lösningen som utnyttjar MongoDB som lagrings- samt indexeringsmetod.

data skapas $2.9 \cdot 10^4$ filer, detta avrundas till $3.0 \cdot 10^4$. Över en period på 30 dagar motsvarar detta $1.8 \cdot 10^6$ skrivåtgärder. Läsningsåtgärder har uppskattats till $2.0 \cdot 10^6$ över en period på 30 dagar. Slutligen uppskattas 100GB data hämtas från lagringskontot, och 10GB data uppskattas skrivas till lagringskontot, per månad. Storleken för B+-trädets data efter indexering av 12 timmar data är 121 MB. Räknat över 30 dagar, 24 timmar per dygn, motsvarar detta ca. 7.2 GB.

Cosmos DB prissätts baserat på tre parametrar: Antalet programbegäranden per sekund, antalet timmar per månad som programbegäranden utförs, samt lagringsmängden som används. En programbegäran motsvarar en operation mot databasen (insättning, uppdatering, sökning, borttagning). Antalet programbegäranden uppskattas vara $2 \cdot l$, eftersom en sökning i `events`-kollektionen följt av en insättning i någon av de två kollektionerna sker vid varje insättning av ett dokument. Detta motsvarar ca. $1.5 \cdot 10^6$ programbegäranden per dygn. Indexeringstiden för 12 timmar data är 8 minuter och 2 sekunder, vilket innebär att indexeringstiden per dygn är 16 minuter och 4 sekunder. Detta avrundas till 15 minuter, det vill säga 900 s. Då



Figur 5.4: Lagringsstorleken för implementationerna, givna i procent av den ackumulerade storleken av samtliga audit-loggfiler.



Figur 5.5: Kostnad för implementationerna. Värdena är baserade på Microsofts priskalkylator, där kostnaderna under en månads tid har uppskattats.

fås 1667 programbegäranden per sekund. Detta ökas till 2000 programbegäranden per sekund för att även inkludera kostnaden för hämtning av data efter indexering i uppskattningen. Den tid som används för programbegäranden under indexeringstid samt hämtning av data uppskattas till 30 minuter per dygn. Detta motsvarar då 360 timmar per månad. Lagringsstorleken för 12 timmars data är 282 MB. Denna storlek multipliceras med 60 för att uppskatta en månads lagringsanvändning, vilket resulterar i 16904 MB, och avrundas till 17 GB.

Kostnaden för lagringskontot respektive Cosmos DB använder förutom de nämnda värdena regionen ”West Europe”. Resterande inställningar är oförändrade i kalkylatorn. Den uppskattade kostnaden visas i Figur 5.5 då samtliga värden matats in i kostnadskalkylatorn.

6
Diskussion

Resultaten tyder på att ett B+-träd i kombination med lokalt indexerade filer ger högre genomgående prestanda än användning av MongoDB som lagrings- respektive indexeringslösning. Detta är något förvånande eftersom det B+-träd som implementerats är ett B+-träd i dess enklaste form. Det finns flera studier på variationer av B+-träd som optimerar datastrukturen för lagring på FLASH-media såsom SSD-diskar, samt undersöker metoder för att minimera antalet skrivningar som sker till lagringsmediat. Exempel på dessa är CircTree [35] och Lazy-split [36]. Dessa implementationer har visat på en mycket högre prestanda än det B+-träd som implementerats i denna rapport. Det finns alltså ytterligare prestanda att hämta från B+-trädet. Detsamma gäller för MongoDB-lösningen, dels eftersom datan lagras i MongoDB, dels eftersom MongoDB ej har finjusterats för bästa prestanda genom användning av interna index eller justering av databasens konfiguration.

Eftersom målet med den utvecklade programvaran var att implementera en programvara för att söka efter händelser från audit-loggfiler som genererats av ett Kuberneteskluster i MA, är det mindre tillfredsställande att just implementationen som använder ett B+-träd för att indexera samt hämta händelser från filer i ett lagringskonto, får sämst resultat när det kommer till hämtningstid för en större mängd data. Dock är detta förväntat eftersom detta är den enda variant som använder en nätverksuppkoppling, vilken kan vara en faktor som påverar resultaten. Med detta sagt behöver ytterligare mätningar utföras där den utvecklade programvaran exekveras direkt i MA i form av en virtuell maskin eller liknande, för att säkerställa att hastigheten på internetuppkopplingen som använts under mätningarna kan uteslutas som en potentiell begränsning av prestandan. Alltså behöver ytterligare mätningar utföras för att erhålla ett mer rättvist resultat för kombinationen B+-träd samt filer lagrade i MA.

Två resultat som erhållits återfinns i Figur 5.3. Det ena resultatet är att hämtning av distinkta objekt av samma typ resulterar i en betydligt längre hämtningstid för MongoDB-lösningen jämfört med fallet då flera händelser hämtas för ett objekt, trots att antalet händelser som hämtas från databasen är större i det andra fallet. Detta tyder på att det inte är antalet händelser som orsakar sämre prestanda, utan istället att implementationen ej är optimal för att hämta flera distinkta objekt ifrån databasen. Det andra resultatet är att MongoDB-lösningen tycks ge en liknande hämtningstid då 1446 händelser hämtas för ett objekt (se Figur 5.3, stapel med prefix B), respektive då 3272 händelser hämtas (se Figur 5.3, stapel med prefix C). Jämför detta med lösningen som använder ett B+-träd, där hämtningstiden blir allt

längre ju fler objekt som hämtas från audit-loggfilerna. Fler mätningar skulle kunna göras här för att säkerställa om, och i sådant fall när, hämtningstiden för B+-trädet överskrider hämtningstiden för MongoDB.

Ett annat perspektiv utav resultaten är lösningarnas kostnadseffektivitet när de exekveras i MA. Värdena som erhållits är endast uppskattningar, men är ändå värda att jämföra då de ger en uppfattning av olika tjänsters kostnader i MA. B+-trädet skulle använt sig utav ett lagringskonto, medan MongoDB-lösningen skulle använt Microsoft Cosmos DB. Kostnadseffektivitet är ett viktigt perspektiv att ha i åtanke när teori skall överföras till praktik. En lösning skall ej vara alltför dyr i drift, samtidigt som den ej skall vara långsam när stora mängder data lagras. Vad som anses vara en accepterad balans mellan kostnad och prestanda varierar givetvis mellan användare. Att jämföra kostnaden för ett lagringskonto och en Cosmos DB-instans är något svårt, eftersom dessa tjänster prissätts på olika sätt. Det går dock att göra en enklare uppskattning av priset, vilken återfinns i Figur 5.5. Resultaten visar att lösningen som använder ett B+-träd ger en prestanda som är likvärdig MongoDB-lösningen, samtidigt som lagringsmängden halveras och kostnaden är ca fem gånger mindre. Räkneexemplet är baserat på den mängd data som använts i samtliga utförda mätningar i denna rapport. Detta motsvarar tolv timmar data, det vill säga mindre än ett dygns data.

Ännu viktigare ur ett kostnadsperspektiv är därför att resonera kring skalbarheten hos de olika lösningarna. Mängden audit-loggfiler som en användare önskar att lagra är ej heller den entydigt bestämd, utan varierar från användare till användare. Låt oss anta att samtliga användare söker att lagra åtminstone data motsvarande det senaste dygnet. Den förenklade lösningen skulle då kosta 18.5 SEK per dygn, jämfört med 3.8 SEK per dygn för den fullständiga lösningen som baseras på ett B+-träd. Detta innebär att den fullständiga lösningen skulle kunna lagra data motsvarande ca 5 dagar och ändå ha en kostnad motsvarande ett dygns data med den förenklade lösningen.

Trots de lovande resultaten för B+-trädet finns det även fördelar med MongoDB-lösningen. En sådan är tiden det tar från programutvecklingens start tills dess att en färdig prototyp finns att tillgå. Med MongoDB, eller ett annat databashanteringssystem, försvinner mycket av mjukvarans komplexitet. Utvecklingstid är ett annat perspektiv på mjukvarans totala kostnad. Dock blir denna en engångskostnad jämfört med de högre prenumerationsavgifter som fås genom MA.

Vidare kan resultatet från Figur 5.4 tolkas från ett hållbarhetsperspektiv. Oavsett ekonomisk kostnad för de båda lösningarna är det även viktigt att resonera kring hållbarhet. I detta fall är det tydligt att B+-trädet använder ca. halva lagringsmängden gentemot MongoDB. Även detta är fördelaktigt för den fullständiga lösningen då lagringsmängd påverkar ett antal hållbarhetsaspekter. Mängden datorhårdvara som behöver produceras och användas för den utvecklade mjukvaran är mindre. Detta bidrar i sin tur till en mindre användning av elektricitet. Med tanke på den stora mängd serverdatorer som är placerade i Microsofts serverhallar runtom i världen

är det rimligt att anta att just denna mjukvara ej påverkar hållbarheten hos MA i någon större grad. Hållbarheten är något som samtliga aktörer, såväl mjukvaruutvecklare som DevOps-utvecklare och molntjänst-aktörer behöver ha i åtanke för att minimera sin miljöpåverkan.

Slutligen finns det ett antal intressanta aspekter att undersöka vidare. En sådan är att vidareutveckla implementationen av B+-trädet genom att studera de metoder som finns för att öka prestandan samt minimera antalet skrivningar till lagringsmedia, såsom CircTree respektive Lazy-split. Nästa aspekt att undersöka är huruvida kostnaden för hela mjukvaran kan minimeras, exempelvis genom att undersöka metoder för att skala upp respektive ned antalet instanser av mjukvaran, genom fasta schemaläggningsprinciper såväl som genom användning av artificiell intelligens för schemaläggning av applikationen. Slutligen finner författaren det intressant att undersöka användningen av komprimering av de filer som skapas då B+-trädets indexeringsprocess. Detta för att minimera de lagringsresurser som används av mjukvaran, för att öka både hållbarheten samt kostnadseffektiviteten.

7
Slutsats

För att knyta an till rapportens syfte kan följande slutsatser dras: Utifrån de två variationer av mjukvaran som har utvecklats, tyder resultaten på att prestandan, lagrings- samt kostnadseffektiviteten är fördelaktig för den fullständiga lösningen som baseras på ett B+-träd, när lokala filer används vid indexering samt datahämtning. Vidare ses även att B+-trädet är fördelaktigt ur ett hållbarhets-, respektive ekonomiskt perspektiv. Trots detta finns det ytterligare möjligheter att optimera lösningen genom att implementera ett mer avancerat B+-träd med syfte att öka mjukvarans prestanda. Det finns även en möjlighet att undersöka användningen av komprimeringsalgoritmer för att ytterligare öka mjukvarans resurs- respektive kostnadseffektivitet.

Referenser

[1] ”Vad är en container?” URL: https://azure.microsoft.com/sv-se/resources/cloud-computing-dictionary/what-is-a-container/#overview (hämtad 2022-10-29).

[2] ”Containers vs. virtual machines,” URL: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms (hämtad 2023-03-26).

[3] ”Configuring PCI passthrough,” URL: https://docs.openshift.com/container-platform/4.10/virt/virtual_machines/advanced_vm_management/virt-configuring-pci-passthrough.html (hämtad 2023-03-26).

[4] ”Overview.” (3 jan. 2023), URL: https://kubernetes.io/docs/concepts/overview/ (hämtad 2022-10-24).

[5] ”Kubernetes,” URL: https://www.cncf.io/projects/kubernetes/ (hämtad 2023-03-26).

[6] ”Container Runtime Interface (CRI).” (23 nov. 2022), URL: https://kubernetes.io/docs/concepts/architecture/cri/ (hämtad 2023-03-21).

[7] ”CNI,” URL: https://www.cni.dev/ (hämtad 2022-03-21).

[8] ”Introduction - Kubernetes CSI Developer Documentation,” URL: https://kubernetes-csi.github.io/docs/ (hämtad 2023-03-21).

[9] ”The Official YAML Web Site,” URL: https://yaml.org/ (hämtad 2023-03-21).

[10] ”What is infrastructure as code (IaC)?” (28 nov. 2022), URL: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code (hämtad 2023-03-21).

[11] ”nginx,” URL: https://nginx.org/en/ (hämtad 2023-03-21).

[12] ”Auditing.” (26 april 2022), URL: https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/ (hämtad 2022-10-24).

[13] ”JSON Lines,” URL: https://jsonlines.org/ (hämtad 2023-03-21).

[14] ”What is Azure?” URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/ (hämtad 2022-11-21).

[15] ”Welcome to Azure Cosmos DB.” (6 dec. 2022), URL: https://learn.microsoft.com/en-us/azure/cosmos-db/introduction (hämtad 2023-03-26).

[16] ”PostgreSQL: The World’s Most Advanced Open Source Relational Database,” URL: https://www.postgresql.org (hämtad 2023-03-26).

[17] ”MongoDB: The Developer Data Platform,” URL: https://www.mongodb.com (hämtad 2023-03-26).

[18] ”Our Mission,” URL: https://www.mongodb.com/company (hämtad 2023-02-19).

[19] ”MongoDB Architecture Guide,” URL: https://www.mongodb.com/collateral/mongodb-architecture-guide (hämtad 2023-02-19).

[20] ”What is a REST API?” (8 maj 2020), URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api (hämtad 2023-03-26).

[21] ”NoSQL vs. SQL Databases,” URL: https://www.mongodb.com/nosql-explained/nosql-vs-sql (hämtad 2023-03-21).

[22] ”Relational vs. NoSQL data.” (7 april 2022), URL: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data (hämtad 2023-03-21).

[23] V. H. Patil, *13.3.3 B+ Tree*, 2012. URL: https://app.knovel.com/hotlink/khtml/id:kt00U5GER4/data-structures-using/nodes-of-b-tree (hämtad 2023-02-19).

[24] P. Kieseberg, S. Schrittwieser, P. Frühwirt och E. Weippl, ”Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective,” i *2019 International Conference on Software Security and Assurance (ICSSA)*, 2019, s. 46–51. DOI: 10.1109/ICSSA48308.2019.00013.

[25] ”GitHub,” URL: https://github.com/ (hämtad 2023-03-20).

[26] ”GitHub - kubernetes/client-go: Go client for Kubernetes,” URL: https://github.com/kubernetes/client-go (hämtad 2023-03-26).

[27] ”Azure SDK for Go.” (19 jan. 2023), URL: https://pkg.go.dev/github.com/Azure/azure-sdk-for-go (hämtad 2023-02-19).

[28] ”Introduktion till bloblagring (objekt).” (16 febr. 2023), URL: https://learn.microsoft.com/sv-se/azure/storage/blobs/storage-blobs-introduction (hämtad 2023-02-19).

[29] ”What is object storage?” URL: https://www.ibm.com/topics/object-storage (hämtad 2023-03-21).

[30] ”Översikt över lagringskonto.” (25 mars 2023), URL: https://learn.microsoft.com/sv-se/azure/storage/common/storage-account-overview (hämtad 2023-02-19).

[31] ”Azure Blob Storage SDK for Go.” (7 febr. 2023), URL: https://pkg.go.dev/github.com/Azure/azure-sdk-for-go/sdk/storage/azblob (hämtad 2023-02-19).

[32] ”Command line tool (kubectl).” (28 nov. 2022), URL: https://kubernetes.io/docs/reference/kubectl/ (hämtad 2023-03-26).

[33] ”UUID.” (21 febr. 2023), URL: https://developer.mozilla.org/en-US/docs/Glossary/UUID (hämtad 2023-03-26).

[34] ”Priskalkylator,” URL: https://azure.microsoft.com/sv-se/pricing/calculator/ (hämtad 2023-03-26).

[35] C. Wang, G. Brihadiswarn, X. Jiang och S. Chattopadhyay, ”Circ-tree: A B+-Tree variant with circular design for persistent memory,” *IEEE Transactions on Computers*, årg. 71, nr 2, s. 296–308, 2021.

[36] R. Jin, H.-J. Cho, S.-W. Lee och T.-S. Chung, ”Lazy-split B+-tree: a novel B+-tree index scheme for flash-based database systems,” *Design Automation for Embedded Systems*, årg. 17, s. 167–191, 2013.

A

Appendix 1

A.1 Källkod för MongoDB-skript

```
db = connect('mongodb://localhost/auditctl');
print("Size: " +
  (db.events.stats({ scale: 1024 * 1024 }).totalSize +
   db.diffs.stats({ scale: 1024 * 1024 }).totalSize) +
   " MB");
```

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

GÖTEBORGS
UNIVERSITET

CHALMERS