HÖGSKOLAN SKÖVDE
Institutionen för kommunikation och information
Examensarbete i datalogi, 20 poäng
C-nivå
Vårterminen 2005

Utvärdering av cachningsalgoritm för dynamiskt genererade webbsidor

Benny Handfast

HS-IKI-EA-05-105

Utvärdering av cachningsalgoritm för dynamiskt genererade webbsidor

Examensrapport inlämnad av Benny Handfast till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information.

05-06-06

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Handledare för examensarbetet: Henrik Engström

Utfört på Lockpick Entertainment AB

Utvärdering av cachningsalgoritm för dynamiskt genererade webbsidor

Benny Handfast

Sammanfattning

Webbservrar på Internet använder idag dynamiska webbsidor genererade med hjälp av databassystem för sina användare. Detta har lett till en stor belastning på webbservrar och en metod för att minska belastningen är att använda cachning. Detta arbete implementerar och utför tester på en specifik cachningsalgoritm kallad Online View Selection i ett webbspelsscenario. Ett potentiellt problem identifieras hos algoritmen som kan leda till att inaktuell information levereras till klienten och algoritmen modifieras för att hantera problemet. Testresultaten visar att både den modifierade algoritmen och originalet ger likvärdig prestanda. Den modifierade algoritmen visar sig fungera men problemet med den ursprungliga algoritmen uppkommer sällan i webbspelsscenariot.

Nyckelord: caching, algoritmer, webbserver, dynamisk sidgenerering, world wide web

Tack till

Jag vill ge ett stort tack till min handledare Henrik Engström för att ha visat stort intresse och givit värdefulla kommentarer och förslag under arbetets gång. Jag vill även tacka David Rosén på Lockpick Entertainment för värdefulla kommentarer och stöd under arbetet.

Innehållsförteckning

1 Introduktion

Webbservrar genererar idag dynamiska webbsidor som kan leda till en stor belastning på de servrar sidorna genereras på. Det är viktigt att webbservrar klarar av den belastning som de ställs inför. Li et al. (2002) redovisar en rapport gjord av Zona Research som visar att 70% av besökare till en e-handels webbplats, lämnar platsen om de får vänta mer än 12 sekunder. Däremot endast 30% vid 8 sekunder och så lite som 2% om de får vänta mindre än 7 sekunder. Dessutom påpekar Labrinidis och Roussopoulos (2004) att om en webbserver belastas för högt, kommer förfrågningar köas på servern och om belastningen fortsätter att vara hög, kommer servern till slut krascha.

De flesta större företag är idag representerade på Internet i form av en egen domän. Många företag bedriver dessutom försäljning och support via Internet. Innehållet på webben är ofta dynamiskt i den mån, att flera sidor uppdateras ofta. Sidorna innehåller ofta specialiserad information åt besökarna, till exempel reklam. Dynamiska sidor genereras antingen på webbservrar eller hos klienter, i form av till exempel Java-applets eller flash-moduler. I denna undersökning avses servergenererade sidor när dynamiska sidor omnämns. För att skapa dessa dynamiska sidor används databassystem som hanterar information och webbservrar med applikationer som konstruerar sidor till besökaren. Det finns problem med att generera dynamiska sidor. Iyengar et al. (1997) visar att dynamiska webbsidor kan försämra prestandan betydligt hos webbservrar. För att öka prestandan hos systemet har nya gränssnitt och språk utvecklats för att snabba upp genereringen av sidor. En vanlig arkitektur på webbserversystem är 3-tier arkitekturen, som består av klienter, webbservrar och databasservrar. Bamford et al. (1999) beskriver hur 3-tier arkitekturen används i system för att lösa en del skalbarhetsproblem. Men i takt med att antalet användare stiger, så ökar kraven på skalbarhet hos databassystem och webbserversystem.

Iyengar och Challenger (1997) föreslår att caching ska användas för att spara redan genererade sidor på webbservern och på så vis lösa problemet med dålig prestanda. Datta et al. (2001) påpekar dock att en sådan lösning inte är tillräckligt bra, eftersom två förfrågningar med samma parametrar till samma sida inte nödvändigtvis genererar samma sida. Istället föreslår författarna att cacha endast delar av sidor eller fragment som de kallar delarna. Det största problemet med traditionell cachning är, enligt Labrinidis och Roussopoulos (2004), att hanteringen av cachade objekt sammanfaller med både förfrågningar och uppdateringar. Det vill säga, att när många uppdateringar sker händer det ofta att ett cacheobjekt måste genereras om under en förfrågan. För att koppla bort den belastningen av en förfrågan används materialiserade vyer. Vyer är html-fragment som bygger upp en webbsida. Materialiserade vyer uppdateras oberoende på förfrågningar för att undvika den belastningen. Labrinidis och Roussopoulos (2004) använder materialiserade vyer i sin algoritm Online View Selection (OVIS).

Detta arbete utvärderar prestanda i OVIS algoritmen i ett interaktivt webbspel. Dessutom identifieras ett problem med algoritmen som gör att inaktuell information kan skickas till en användare. I arbetet tas en modifierade variant av OVIS algoritmen fram, som kan förbättra algoritmen genom att ge garantier för hur gammal information i vyer är. Denna utvärderas genom tester för att undersöka en eventuell prestandaskillnad mellan originalet och förändringen av algoritmen.

2 Bakgrund

I detta kapitel belyses bakgrunden för dynamiska sidor och cachning. Kapitlet börjar med att gå igenom dynamisk sidgenerering och därefter vilka verktyg och språk som används för att generera dem. Efter det presenteras olika arkitekturer som används för att bygga webbplatser, följt av en presentation av den typ av webbplats som detta arbete riktar sig mot. Därefter diskuteras cachning i allmänhet och några existerande lösningar för Internet beskrivs. Sedan presenteras OVIS algoritmen och olika tester som används, för att utvärdera webbsystem samt verktyg för att skapa last på ett system.

2.1 Dynamisk sidgenerering

Statiska webbsidor är grunden för de dynamiska. Statiska sidor innebär helt enkelt att en sida ser likadan ut, oberoende av vilken användare som besöker den. En klient efterfrågar en resurs och servern svarar med att skicka den. En uppdatering sker när administratören eller den som har ansvar för innehållet på webbservern ändrar i filerna på webbservern.

Dynamisk sidgenerering sker på webbservern, med data från en eller flera databaser. När en klient skickar förfrågningar på sidor till en webbserver, kan även data skickas till webbservern via cookies eller fält i adressen. Den data som skickas används som parametrar till det program eller modul som sköter sidgenereringen. Det är programmet eller modulen som hämtar data från databasen och genererar HTML-kod som sedan skickas som svar till klienten via webbservern.

Common Gateway Interface (CGI) är en av de första teknikerna för att generera sidor vid förfrågningar. CGI har problem med prestanda vid höga belastningar, eftersom en ny process startas vid varje förfrågan (Iyengar & Challenger, 1997). Istället användes senare ISAPI och NSAPI som är ett gränssnitt för webbservermjukvara. ISAPI är för Microsofts Internet Information Server och NSAPI är till Netscapes Web Server. Efterföljande tekniker är bland andra PHP och Active Server Pages (ASP) som båda är skriptspråk som tolkas vid körning. ASP.NET och Java Servlets är tekniker som låter programmeraren jobba i .NET respektive Java ramverket. Detta ger fördelar till programmeraren, eftersom mycket finns klart i de ramverken. Dessutom använder både Java och .NET tekniker för *run-time* kompilering, vilket innebär att kod exekveras snabbare än kod som tolkas. Trots detta är det många som har påpekat att dynamisk sidgenerering försämrar prestandan och skalbarheten än statiska sidor hos webbservrar. Några av dessa är Yagoub et al. (2000), Labrinidis och Roussopoulos (2004) och Datta et al. (2001).

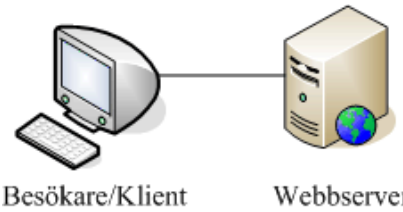
2.2 Arkitekturer

Two-tier och Three-tier arkitekturer beskrivs olika i litteraturen, vilket leder till att beteckningar på arkitekturer kan skilja sig åt i olika källor. Vanliga skillnader i beteckningarna är, huruvida presentationslagret är i webbläsaren hos klienterna eller i webbservern. I denna rapport används följande definitioner.

Two-tier arkitekturen

Beteckningen på two-tier arkitekturen hämtas från Manuel och AlGhamdi (2003). Two-tier arkitekturen kallas även klient-server. Arkitekturen består av två lager, en klient och en server. Klienten och servern kommunicerar med varandra enligt ett specificerat protokoll som de båda måste följa. Klienten brukar kallas för presentationslagret, eftersom den presenterar data från servern. Servern bidrar med

data och kallas därför för datalager.



Figur 1: Exempel på en klient-server arkitektur

Three-tier arkitekturen

3-tier arkitekturen representeras som datalager, applikationslager och presentationslager (Manuel och AlGhamdi, 2003). Datalagret innehåller en eller flera databasnoder. Applikationslagret innehåller specifika regler för systemet. Presentationslagret hämtar information från applikationslagret och presenterar information till användaren.

Figur 2: Exempel på en three-tier arkitektur

N-tier arkitekturer

N-tier arkitekturen är en utökning/generalisering av 2- och 3-tier arkitekturen (Manuel och AlGhamdi, 2003). N-tier har N stycken lager i arkitekturen där kommunikationen mellan varje lager följer ett protokoll.

Arkitektur för webben

Li et al. (2003) beskriver, enligt dem, den grundläggande infrastrukturen för databasdrivna webbsidor. Infrastrukturen består av webbserver, applikationsserver och databassystem.

2.3 Typ av webbplatser

Webbplatser som detta arbete inriktar sig mot är databasbaserade, vilket innebär att det finns sidor som genereras från databaser. Webbplatserna har ett stort antal användare som besöker sidorna. Sidorna består av webbvyer som är dynamiskt genererade. Webbvyer är HTML-fragment som bygger upp en webbsida och kan till exempel, vara indelade efter positionen på sidan där vyn är placerad. Informationen på webbvyer som genereras dynamiskt är antingen

- personlig för en enskild användare,
- unik för en grupp användare,
- eller gemensam för samtliga användare.

Att en vy är personlig innebär att informationen på webbvyn är unik för just den användaren. Om vyn är unik för en grupp användare innebär det att, informationen i vyn innehåller gemensam information till gruppen. Om en vy är gemensam för samtliga användare, riktar sig informationen till alla användare. Ett exempel på det senare är en räknare över hur många som är online. Dessa databasbaserade webbplatser genererar en stor del uppdateringar och förfrågningar mot databaser. Det är viktigt att databassystem klarar den belastningen, eftersom sidgenereringen ofta

beror på dem.

Webbaffärer, communities, nyhetssidor och interaktiva webbspel har dessa egenskaper och belastningen på dessa webbplatser kan vara väldigt höga. Lunarstorm, en populär community sida, hade under januari 2005 drygt 1 245 miljoner sidvisningar enligt KIA Index (2005).

2.4 Cachning

Cachning av data kan ske på flera olika nivåer. I operativsystem cachas data som läses och skrivs till disk (Silberschatz et al., 2003). Databassystem använder cachning för att spara tabeller eller tupler, som oftast används av klienterna. På webben används proxyservrar mellan klienten och webbservrar. Proxyservrarnas uppgift är spara bilder och statiska dokument som efterfrågas. Om en klient efterfrågar en sparad resurs, skickas den till klienten. Denna metod minskar mängden kommunikation på nätverket och avlastar webbservrar. Detta är en av de lösningar som fungerar bäst för statiska webbsidor, men som enligt Altinel et al. (2003) inte är användbar för dynamiska sidor, eftersom prestandavinsten inte är tillräckligt hög. Även klienter använder cachning för att lokalt spara bilder och statiska html-dokument och kan på så sätt minska nätverkstrafiken samt den tid det tar för att hämta en sida.

2.4.1 Problem med cachning

Ett av problemen med cachning är att hålla den cachade datamängden konsistent med källan. Cao och Liu (2003) definierar två nivåer på konsistens: *Weak* och *strong consistency*.

- *Weak consistency* innebär att en cachad datamängd kan vara gammal, det vill säga att källan har nyare data än vad cachen har. *Weak consistency* kan upprätthållas enligt en Time-To-Live (TTL) approach eller via *polling,* som upprätthåller konsistensen, genom att periodiskt uppdatera cachen från källan (Cao och Liu, 2003). TTL metoden innebär att varje cacheobjekt skapas med en tidstämpel som beskriver hur länge objektet är aktuellt, ungefär som ett bäst före datum.
- *Strong consistency* innebär att en cachad datamängd uppdateras direkt när källan uppdateras. För att upprätthålla *strong consistency* beskriver Cao och Liu (2003) två metoder, invalidering och *polling-every-time*. När invalidering används skickas meddelanden till de som cachar data när källan uppdateras. *Polling-every-time* innebär att en cache alltid kontaktar källan, för att jämföra versionen på den cachade datamängden, innan den skickas vidare.

Ett annat problem är att bestämma vad som ska cachas. I praktiken finns begränsningar på hur stor mängd data som får plats i cachen. Ur prestandasynpunkt är det viktigt att cacha den datamängd som ökar prestanda. På grund av begränsningen med utrymme i cachen, måste det finnas regler, för vilken data som ska tas bort när cachen är full. Det är då vanligt att försöka ta bort data som används minst, används först eller ger minst prestandaökning.

2.4.2 Existerande cachelösningar för dynamiskt genererade webbsidor

Det finns en del existerande cachningslösningar för webben. Grovt indelat finns två olika grupper. Den ena försöker lösa problemet genom att cacha eller optimera databasen. Den andra cachar genererade sidor. Ett exempel på databaslösningar är DBCache som är ett system för att replikera databastabeller till webbservrar (Luo et al., 2002). Ett exempel som cachar genererade sidor är Xcache från XCache Technologies (2005).

Datta et al. (2001) har jämfört en primärminnesdatabas TimesTen (2005) mot två andra lösningar som cachar html-fragment. Den ena från BEA Systems (2005) och

den andra är en lösning som de hade utvecklat själva. Resultatet av jämförelsen visade att primärminnesdatabasen skalade sämst. Dock kan båda grupperna av lösningar kombineras.

I Microsofts ramverk för ASP.NET finns det en inbyggd cachefunktionalitet som möjliggör att WebUserControls (ungefär som webbvyer) cachas. WebUserControls och även inbyggda kontroller kan cachas enligt en time-to-live modell eller programmeringsmässigt via en Cacheklass.

2.5 OVIS algoritmen

Labrinidis och Roussopoulos (2004) föreslår en algoritm, Online View Selection (OVIS), för cachning på webbservrar där gammal data används, även under uppdateringar och på så vis kan genomströmningen öka. Författarna har två mål med algoritmen. Dels ska den ge högre genomströmning och dels ska den förhindra att webbservern ger väldigt höga responstider vid en överbelastning. Detta kapitel beskriver OVIS(θ) algoritmen som Labrinidis och Roussopoulos (2004) presenterar i sitt arbete.

2.5.1 Webbvyer, webbsidor och relationer

Tre huvudtermer som används i OVIS algoritmen är webbvyer, webbsidor och relationer. En relation är en datakälla, som används för att generera en webbvy. Relationer är oftast tabeller i en relationsdatabas, men kan lika gärna vara andra modeller. Relationer uppdateras direkt och i den ordning som de inträffar. En webbvy, även kallad vy i denna rapport, är html-fragment av sidor som genereras. Dessa webbvyer kan vara virtuella, ickematerialiserade eller materialiserade. En virtuell webbvy genereras alltid från datakällan och cachas därför aldrig. En ickematerialiserad webbvy cachas och invalideras när en uppdatering har skett i datakällan som användes för att skapa webbvyn. När webbvyn är invalid uppdateras cachen vid nästa förfrågning på den. En materialiserad webbvy hämtas alltid via cachen och uppdateras i bakgrunden, när uppdateringar har skett på källdatan till webbvyn. Materialiserade webbvyer invalideras aldrig och det innebär att de kan innehålla gammal, ej uppdaterad, data. Genom att materialisera webbvyer kan uppdateringar och förfrågningar på webbvyn ske parallellt. Webbsidor skapas av vyer genom att konkatenera de html-fragment från vyerna som webbsidan är uppbyggd av. Förfrågningar på webbsidor sker från klienter och indirekt sker då förfrågningar på webbvyer.

2.5.2 Arkitektur

OVIS algoritmen använder en *Asynchronous Cache module* (ASC), utöver databassystemet och webbservern, där vyer lagras när de cachas eller är materialiserade. ASC modulen är ett lager mellan databassystemet och webbserver-applikationen. Alla förfrågningar på data, det vill säga vyer, sker via ASC modulen. Arkitekturen baseras på en N-tier arkitektur och ASC modulen kan vara frikopplad både databassystemet och webbserverapplikationen, eller byggas i samma applikation som databassystemet eller webbserverapplikationen.

2.5.3 Mätvärden

Labrinidis och Roussopoulos (2004) använder *f* funktionen för att beskriva kvalitet på både vyer och förfrågningar på sidor.

För att veta om en webbvy är uppdaterad eller ej, definieras en funktion för kvaliteten på webbvyn:

$$f(w_i,t)=\begin{cases}1, & \textit{när webbvyn , } w_i\textit{ , är uppdaterad vid tidpunkt t.}\\ 0, & \textit{när webbvyn , } w_i\textit{ , inte är uppdaterad vid tidpunkt t.}\end{cases} \quad (1)$$

En webbvy är uppdaterad när den är konsistent med de relationer i databasen som används vid generering av webbvyn.

Kvaliteten på en webbsida, p_j, vid tidpunkt t beskrivs i formel nummer 2.

$$f(p_j,t)=\sum_{i=1}^{n}(a_{i,j}\cdot f(w_i,t));\ \textit{där}\ \sum_{i=1}^{n}a_{i,j}=1 \quad (2)$$

Faktorn $a_{i,j}$ är ett sätt att beskriva hur viktig en webbvy, i, är på en sida, j, och eftersom kvalitén på en sida max får vara 1, gäller det också att summa av alla $a_{i,j}$ faktorer för en webbsida p_j blir 1. N är antalet vyer på sidan p_j. En förfrågan, A_k, på en sida p_j vid tidpunkten t_k skrivs som $A_k=(p_j, t_k)$. Kvalitén för A_k blir då:

$$f(A_k)=f(p_j,t_k)$$

Medelkvalitén på en ström av n stycken förfrågningar beskrivs av formel 3, QoD står för *quality of data.*

$$QoD=\frac{1}{n}\cdot\sum_{k=1}^{n}f(A_k) \quad (3)$$

För att kunna utvärdera vilka vyer som ska materialiseras eller inte, används statistiska värden på vyer. Alla statistiska värden estimeras från tidigare perioder, en period definieras nedan. För att estimera värden används formel 4 som ursprungligen kommer från Jacobsen (1988). Variabeln a' är det nya estimaten från tidigare perioder, a är den gamla estimaten och m är värden för den nuvarande perioden. Konstanten g är 0.25 vilket Jacobsen (1988) föreslår.

$$a'=(1-g)a+gm \quad (4)$$

När vyer utvärderas om de ska materialiseras eller inte, används statistik för att avgöra hur mycket prestandan ökar om vyn materialiseras och hur mycket av kvaliten på levererade webbsidor som försämras. Kostnaden i prestanda som beräknas för att behålla en vy ickematerialiserad beräknas enligt formel 5.

$$Cost_{non-mat}(w)=H_r\cdot N_{acc}\cdot A_{hit}+(1-H_r)\cdot N_{acc}\cdot A_{miss} \quad (5)$$

H_r är hur stor sannolikheten är att en vy kan hämtas ur cachen när den förfrågas, på engelska *cache hit ratio*. N_{acc} är antalet förfrågningar som sker på webbvyn under perioden. A_{hit} respektive A_{miss} är hur mycket det kostar att hämta vyn från ASC modulen respektive att generera om den. Kostnader definieras som hur lång tid det tar att utföra en operation, till exempel hämta en vy från ASC modulen. Samtliga av de värden i formel 5 estimeras från den nuvarande perioden med formel 4 för att beräkna kostnaden för nästa period.

Kostnaden att ha en vy materialiserad beräknas enligt formel 6.

$$Cost_{mat}(w)=N_{acc}\cdot A_{hit}+R_r\cdot N_{upd}\cdot U_{mat} \quad (6)$$

N_{upd} är antalet uppdateringar som sker på relationer som webbvyn genereras utifrån under perioden. U_{mat} är hur mycket en generering eller uppdatering av vyn kostar. Eftersom uppdateringar på relationer kan köas används R_r som är andelen utav uppdateringarna som leder till att vyn uppdateras. Även dessa variabler är estimerade med avseende från tidigare perioder.

Formel 7 beräknar hur mycket kvalitet en vy ger till hela webbplatsen.

$$Quality_{Cont}(w) = F_r \cdot \frac{N_{acc-a}}{n} \quad (7)$$

F_r är andelen förfrågningar på vyn w som har resulterat till att uppdaterad data har skickats som svar. F_r variabeln är olika beroende på om vyn är materialiserad eller inte eftersom cachade vyer alltid skickar uppdaterad data. N_{acc-a} är antalet förfrågningar på vyn multiplicerat med faktorn $a_{i,j}$ från formel 2. Har det skett 1000 förfrågningar på vyn och faktorn är 0,3 blir N_{acc-a} 300. Notera att en vy kan existera på flera olika sidor och kan därför ha olika a faktorer.

2.5.4 Algoritmen

Under körning arbetar algoritmen i två lägen, passiv eller aktiv. I det passiva läget samlar algoritmen endast statistik, för att sedan i det aktiva läget kunna utvärdera statusen och förändra om vyerna ska vara materialiserade eller inte. Algoritmen aktiveras periodvis av ett tidsintervall eller ett antal förfrågningar. Då förändras policyn på vyer beroende på hur medelkvaliten på de levererade svaren ser ut. OVIS har en parameter, θ, som är ett värde mellan noll och ett. Värdet representerar målen för medelkvaliteten på webbsidor som levereras till klienter. När algoritmen aktiveras identifieras statusen på systemet. Det finns två lägen då algoritmen behöver förändra policyn på vyer. Dessa lägen kallas för *surplus* och *deficit*.

Är medelkvaliten under värdet, θ, är systemet i *deficit*-läge. Då måste algoritmen minska på materialiseringen av vyer för att då höja medelkvaliten. Om medelkvaliten är över θ är systemet i *surplus*-läge. Då kan fler vyer materialiseras för att öka genomströmningen.

Eftersom materialiseringspolicyn bestäms efter estimater, tillåter inte algoritmen att alla vyer byter materialiseringspolicy, efter en periods slut. Estimater är just estimater och kan vara estimerade fel. Därför tillåts endast en del av alla vyer att byta materialiseringspolicy per period.

I *surplus*-läget materialiseras de vyer som ger mest prestandavinst och som samtidigt inte gör att systemet förlorar så mycket i medelkvalitet att det går under θ värdet. I *deficit*-läget slutar de vyer som ger mest medelkvalitet att materialiseras.

Under en belastning som genererar mer arbete än vad webbservern klarar av, kommer förfrågningar att köas på servern. Detta gör att responstiderna kommer att öka markant. För att undvika att responstiderna ökar materialiseras alla webbvyer. Detta sker när medelkvaliten sjunker kraftigt eller medelresponstiden ökar kraftigt.

2.5.5 Utförda simuleringar

De simuleringar som genomförts av Labrinidis och Roussopoulos (2004) utförs i ett simuleringsprogram som de skrivit. De gör följande antaganden under simuleringen:

- Kostnaden (i beräkning) för att uppdatera relationer i databasen var konstant för alla relationer.
- Kostnaden för att generera/uppdatera en materialiserad webbvy eller uppdatera en cachad webbvy var konstant för alla webbvyer.
- Kostnaden för att hämta en vy från ASC modulen var konstant för samtliga webbvyer.
- ASC modulen har en oändlig mängd utrymme vilket leder till att vyer aldrig behöver tas bort från ASC modulen.

Vid simuleringarna, har OVIS algoritmen gett högre genomströmning av förfrågningar, än när endast icke-materialiserad cachning med invalidering har används. Samtidigt har OVIS algoritmen givit högre kvalitet på svaren än endast när

materialiserade webbvyer används. Kvalitén mättes enligt formel 3.

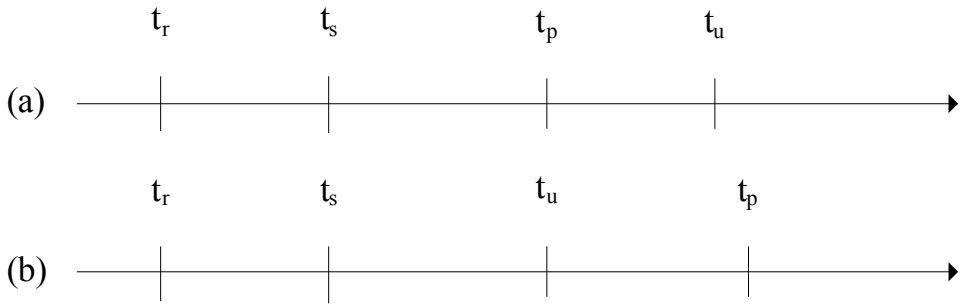
2.6 Kvalité på data

För att kunna mäta hur gammal data är som klienten får, behövs en definition på hur gammal data är vid en tidpunkt t. Att veta att data är gammal, vilket Labrinidis och Roussopoulos (2004) gör, räcker inte i det tänkta scenariot som detta arbete ska undersöka. Följande definition för att mäta ålder på en vy är hämtad från Cho och Garcia-Molina (2003):

$$Age(w_i, t) = \begin{cases} 0, \ om\ webbvyn\ w_i\ är\ uppdaterad\ vid\ tidpunkt\ t \\ t - t_m(w_i), annars \end{cases} \quad (8)$$

Variabeln $t_m(w_i)$ är tidpunkten då en modifikation som påverkar w_i har inträffat. I detta arbete måste vi dock begränsa $t_m(w_i)$ till tidpunkten då modifikationer blir kända inom systemet. I praktiken innebär detta att $t_m(w_i)$ är tidpunkten då en uppdatering kommer in till systemet (istället för när detta skedde i verkligheten).

Data som presenteras hos en klient är inte nödvändigtvis alltid uppdaterad och konsistent med datakällan. Till exempel, om en förfrågan på en webbsida kommer in vid tidpunkt t_r, skickas vid t_s och tas emot av klienten vid tidpunkt t_p. En uppdatering kommer in vid tidpunkt t_u. Figur 3a visar hur ett svar är konsistent med datakällan vid leverans och vid mottagande, men figur 3b visar att svaret inte är konsistent vid mottagandet.



Figur 3: Ålder och konsistens på information vid mottagande av svar.

I figuren 3a kommer data enligt definitionen vara 0 tidsenheter gammal vid tidpunkten t_p, men i 3b är data t_p - t_u tidsenheter gammal vid tidpunkten t_p. Detta exempel visar att data kan vara gammal när den kommer fram till användaren. Det viktiga är att se till, att den inte blir för gammal, att den inte uppfyller användarens krav.

Svarstid definieras som tidpunkten då svaret börjar skickas, t_s, minus tidpunkten när en förfrågan når webbservern, t_r. Svarstiden för en förfrågan $R=(p_j, t_r)$ definieras enligt formel 9.

$$Response(R, t_s) = Response(p_j, t_r, t_s) = t_s - t_r \quad (9)$$

Anledningen till att svarstiden mäts på servern, är för att kunna jämföra svarstiden på olika förfrågningar, oberoende på nätverkets inverkan. Labrinidis och Roussopoulos (2004) jämför prestanda på materialiserade respektive icke-materialiserade webbvyer genom att jämföra svarstiderna.

2.7 Webbscenario

De webbsidor som används i undersökningen, är baserade på webbdelen av ett spel från Lockpick Entertainment. Webbdelen av spelet är ett lagspel där ett lag består av

ett antal spelare. Lagen och spelarna utmanar varandra för att bli starkare och komma vidare i spelet. Spelet utspelas i en persistent värld där spelet hela tiden är igång, oberoende av spelarnas aktivitet. Varje spelare har ett eget rike och är allierad med laget. För att sköta om sitt rike och hjälpa andra i laget kan spelaren logga in via webbdelen av spelet. Förutom de möjligheter som finns i webbgränssnittet och som beskrivs nedan, kan uppdateringar till databasen ske via en klientapplikation. I undersökningen av algoritmerna beaktas endast de möjligheter som webbgränssnittet ger. Dessa möjligheter är:

- In- och utloggning,
- visa information om armér, magi etcetera på sitt rike och de allierade rikena,
- träna arméer,
- utveckla teknologier,
- utöva magi,
- handla med andra riken i laget.

Dessa aktiviteter leder till att databasuppdateringar hela tiden sker och belastar då databasen.

2.7.1 Problem med OVIS algoritmen i scenariot

Webbspel kan vara mycket högt belastade och det finns krav på att besökare inte får gammal information, eftersom spelare då kan ta felaktiga beslut. Vilket kan resultera i att spelet inte blir spelbart. OVIS algoritmen beskriven ovan, ger inga garantier på hur gammal en vy får vara, när den levereras. När en uppdatering sker på en relation, som en materialiserad vy beror på, läggs den vy i kö för uppdatering. Om många vyer finns i kön kan det dröja innan vyn blir uppdaterad.

För att kunna försäkra sig om att användare inte får för gammal information, finns ett krav att kunna garantera att information som lämnar webbservern, inte får bli för gammal. I detta arbetet definieras inaktuell information, när informationen är för gammal för att vara användbar för användare.

2.8 Test och testverktyg

För att undersöka hur webbservrar beter sig vid hög belastning, krävs tillgång till verktyg, som kan generera last på systemet. Det verktyg som krävs, är en programvara som kan skicka stort antal förfrågningar under en tidsperiod. Programvaran ska även kunna skicka förfrågningar med sessioner, vilket är nödvändigt i webbspelsscenariot.

Cassone et al. (2001) beskriver fem typer av test på webbservrar:

- **Smoke test**

  Smoke test är ett snabbt test, som ska visa om applikationen på servern är redo att testas. Detta görs för att hitta uppenbara fel i systemet.

- **Load test**

  Load test används för att testa hur applikationen beter sig under normala förhållanden, när systemet körs i den tänkta miljön. Lasten ökas successivt från en liten last till en hög last för att undersöka hur systemet beter sig.

- **Spike test**

  Spike test undersöker hur väl ett system klarar spikar, en temporär höjning, av förfrågningar under körning.

- **Stress test**

  Stress tester användas för att kontrollera hur systemet beter sig vid oväntat höga laster.

- **Stability test**

  Stability testning sker under långa perioder för att kunna hitta fel som uppträder under en långvarig exekvering. Minnesläckor är ett fel som kan upptäckas vid stability tester.

Load testning och stress testning kan användas för att mäta en webbservers skalbarhet. Skalbarhet definieras enligt Oxford Reference Online (2005), som hur väl ett system som är designat för att fungera på en viss belastning, klarar av ytterligare belastning. I detta fall mäts hur väl en webbserver klarar olika mängder av förfrågningar per tidsenhet.

3 Problembeskrivning

Inriktningen i det här arbetet är att implementera och utvärdera OVIS algoritmen i ett webbspelsscenario samt att ta fram en förändring av OVIS för att undvika problemet med icke aktuell information i vyer. Som beskrivs i föregående kapitel är det vanligt att webbsidor genereras dynamiskt på webbservrar. Enligt Iyengar et al. (1997) finns det dock en del prestandaproblem hos dessa typer av dynamiska webbsidor. Prestandaproblemet beror på att sidor genereras vid förfrågningar. Prestandaproblem kan leda till högre responstider för besökarna. Undersökningen av Zona Research som Li et al. (2002) presenterar visar att besökarna överger sidorna när responstiderna blir för höga.

Proxyservrar är en cachningsmetod som används för statiska webbsidor för att öka prestanda. Altinel et al. (2003) argumenterar för att den lösningen inte är användbar på dynamiska sidor. Iyengar och Challenger (1997) föreslår att cachning av objekt till generering av dynamiska sidor ska användas. Medan Datta et al. (2001) visar att prestanda kan ökas ytterligare om fragment av sidor cachas för sig. Labrinidis och Roussopoulos (2004) föreslår en cachealgoritm som de kallar Online View Selection (OVIS). OVIS är en algoritm för att cacha HTML-fragment, så kallade webbvyer, på en webbserver. Vyer kan vara virtuella, icke-materialiserade eller materialiserade. Traditionellt sett kan invalidering användas för att upprätthålla *weak consistency* med källdata och cachen (Cao och Liu, 2003). I vanlig cachning invalideras objekt när källdata uppdateras och cachen uppdateras när data förfrågas. När webbvyer materialiseras uppdateras inte cachen vid en förfrågan utan i bakgrunden. Vid en hög belastning finns dock inga garantier på hur gammal informationen som levereras kan vara.

I föregående kapitel beskrivs en typ av databasdriven webbplats. Den typen av webbplatser kan vara mycket högt belastade, och i webbspelscenariot som beskrivs, finns det krav på att besökare inte får inaktuell information.

OVIS algoritmen är endast simulerad och är inte testad på en webbserver som ett interaktivt webbspel använder.

3.1 Problemprecisering

Arbetet består av att implementera OVIS algoritmen för att kunna testa och utvärdera den i webbspelscenariot. För att kunna garantera att materialiserade webbvyer inte får leverera äldre information, än vad som besökarna eller systemet tillåter, så förändras OVIS algoritmen för att ta hänsyn till åldern på information. När en materialiserad webbvy innehåller inaktuell information får den inte skickas iväg, förrän webbvyn är uppdaterad. Reglerna för webbvyers ålder definieras enligt hur lång tid som passerat sedan en uppdatering har skett.

Utvärderingen av algoritmerna ska visa, hur väl de olika algoritmerna skalar med antal förfrågningar per tidsenhet, och hur gammal informationen i webbvyer blir vid olika mängd förfrågningar.

3.2 Mål

Målet med arbetet är att ge en inblick i hur OVIS algoritmen presterar i scenariot med ett webbspel. Dessutom är ett mål med arbetet att förändra OVIS algoritmen, för att kunna garantera att åldern, på den levererade informationen, i materialiserade webbvyer inte överstiger ett givet värde.

3.3 Delmål

Målet kan vidare delas upp i följande delmål.

1. Införa garantier på information i materialiserade vyer i OVIS, för att hantera problemet med vyer, som inte innehåller aktuell information.
2. Implementera OVIS algoritmen i webbspelet för att få en grund för testningen i det specifika scenariot.
3. Testa prestanda och åldern på den information som levereras till användare när både OVIS algoritmen och den förändrade algoritmen används.

Undersökningen ska visa hur prestandan för de olika algoritmerna är och hur väl den modifierade algoritmen fungerar.

4 Metod

För att undersöka algoritmerna kan tester utföras som benchmarking, simulering eller matematisk analys. Simulering används för att simulera händelser i system och hela nätverkstopologier kan simuleras i en enda dator. Matematisk analys används för att beräkna hur prestandan kommer att påverkas med olika algoritmer och parametrar. Algoritmer måste analyseras noggrant för att kunna utföra en matematisk analys. Eftersom många olika algoritmer och datastrukturera är inblandade i en webbserver görs inte detta. En annan svårighet med matematisk analys är att analysen på algoritmerna måste vara korrekta annars kan inte resultaten användas som grund för vidare arbete. I en simulering måste alla antaganden som systemet bygger på vara korrekta om ett pålitligt resultat ska uppnås. Till skillnad mot matematisk analys behövs algoritmer inte analyseras lika noggrant här då algoritmer kan implementeras i en simulator. I en benchmarktestning behövs inga analyser på algoritmer eller datastrukturer då beteendet hos dessa uppkommer i testmiljön. All indata i testerna måste dock tas fram, i detta fall databasen och hur förfrågningar sker på systemet.

I detta arbete genomförs en benchmarking för att undvika risker med felaktiga antaganden i matematisk analys och simulering, samt för att testa algoritmerna i en verklig webbserver. För att utvärdera algoritmerna finns ett antal metoder för att genomföra benchmark-testning av dem. Webbplatser som till exempel har över 10000 besökare är svåra att praktiskt genomföra tester på, eftersom det är svårt att få samma indata till olika tester, eftersom både sidor och databaser kan ha förändrats. Vid tester på riktiga webbplatser kan flera tester utföras vid samma tidpunkter flera dagar i följd och sedan via statistik avgöra vilken algoritm som passar bäst för platsen. Dessa tester kan ej utföras i detta arbete på grund av att det tar för lång tid och det inte finns tillgång till en sådan webbplats med den belastningen.

För att göra tester beskriver Banga och Druschel (1997) två sätt att simulera förfrågningar den verkliga miljön. Den ena metoden är att använda loggar från webbplatser som bas för att simulera last på systemet. Genom att använda loggar blir resultaten specifika mot en webbplats och kan vara lättare att jämföra gentemot simulering. I en simulering av förfrågningar används endast ett fåtal klienter som skickar förfrågningar till en webbserver, men med en högre frekvens för att motsvara de, i exemplet, 10000 besökare som besöker sidor mer sällan. Eftersom den andra metoden inte baseras på loggar, måste proportionen på hur sidor besöks, uppskattas och kan sedan testas enligt en statistisk modell. I denna undersökning används den senare metoden utan loggar av den anledningen att det ej finns tillgång till några loggar.

I kapitel 2 beskrivs ett antal olika benchmarktester för webbsystem. På grund av tidsbegränsning utförs inte alla typer av tester. De testertyper som används är smoke, load och stresstest. Anledningen till att stabilitytest inte utförs är att det tar lång tid att utföra ett sådant test och det är till för att hitta fel som uppkommer under en lång tids körning. Spiketest utförs inte på grund av att algoritmerna ska jämföras mot varandra och då är det enklare att ha samma förfrågningshastighet under testen för att enklare jämföra resultatet.

Webbspelet som tester utförs på i arbetet väljs på grund av kombinationen med vyer för enskilda användare, grupper av användare och samtliga användare. Dessutom är webbspelet i en domän där det är viktigt för användare att informationen på sidor är aktuell. På grund av tidsbegränsning i arbetet begränsas testerna till denna webbplats. Vid vidare tester bör de ske på fler webbprojekt inom databasbaserade webbplatser. Exempel på dessa webbplatser är e-handelsplatser, nyhetssidor och banksidor.

Anledningen till att just en databasbaserad webbplats används i testningen är att OVIS algoritmen är inriktad på sådana webbplatser (Labrinidis och Roussopoulos, 2004).

OVIS algoritmen finns tillgänglig i Labrinidis och Roussopoulos (2004) som teori och en del pseudokod och implementeras därefter. I kapitel 5.1 beskrivs valet av servermjukvara och testverktyg.

5 Genomförande

I detta kapitel presenteras testmiljön och de olika tester som används för att utvärdera OVIS algoritmen och den modifierade varianten som här kallas OVIS-TG.

5.1 Testmiljö

Den miljö testerna utförs i har en stor bredd av möjligheter och varianter. Detta delkapitel täcker den fysiska testmiljön, såsom nätverkslayout och antal klienter, samt de mjukvaror som används och hur de används.

5.1.1 Fysisk miljö

Som beskrivs ovan är det praktiskt svårt att genomföra tester på ett system som är verksamt eller i den skala som systemet ska köras. Anledningen är att verksamma system inte nödvändigtvis har samma distribution av besökare från test till test, vilket medför att testresultaten inte kan jämföras från test till test, utan att ta hänsyn till indatan, det vill säga den olika distributionen av inkommande förfrågningar. Om det finns tillgång till en miljö med samma antal besökande klienter och där det är möjligt att styra dem, kan det bli svårt att koordinera och administrera tester. För att undvika dessa problem med olika indata eller koordinering av många klienter, används endast en eller ett fåtal klientdatorer som skickar ett stort antal förfrågningar istället för att använda många klienter som sällan skickar förfrågningar. Det är även möjligt att köra testen på en dator och då köra både lastgenereringsprogram, webbserver och databas-systemet. Men detta gör ofta att webbservern tar för mycket kraft från lastgenereringsprogrammet eller tvärtom, så att testen blir missvisande. I denna undersökning används en klientdator som skickar förfrågningar till en serverdator, eftersom lastgenereringsprogrammet då inte kan tävla om resurser med webbserverprogramvaran.

Eftersom testningen ej sker på samma dator måste datorerna kommunicera över ett nätverksmedium. Testen kan utföras över Internet eller i en lokal nätverksmiljö, *Local Area Network* (LAN). Om tester utförs över Internet kan störningar och oregelbunden trafik förekomma, vilket gör att många tester behövs göras för att få pålitliga resultat. Det finns olika nätverksmedium som fungerar på båda typerna som till exempel, trådlös, optisk och vanliga Ethernet kablar. Valet av medium är godtyckligt så länge bandbredden inte gör att nätverket blir en flaskhals under testerna. I ett system över Internet förekommer även routrar och brandväggar för att sortera paket. En webbserverdator har ofta någon form av brandvägg, antingen mjukvarubrandvägg eller hårdvarubrandvägg, kopplad mellan en Internet router och datorn. Detta krävs för att förhindra attacker. Routrar och brandväggar påverkar genomströmningen av trafiken, vilket administratörer måste ta hänsyn till, när de bygger infrastrukturen för stora webbplatser. För att minska risken att genomströmningen påverkas används en LAN miljö där ingen extra kommunikation sker förutom mellan klientdatorn och serverdatorn.

Det finns två olika sätt att placera databassystemet. Antingen får databasen köras på en separat dator eller på samma dator som webbserverprogramvaran körs på. Att köra databassystemet på en separat dator ger mer resurser i form av processortid än att köra det på samma dator som webbserverprogramvaran. En nackdel är att det tar en icke försumbar tid att skicka information mellan datorerna. För att undvika att skicka information över nätverket mellan databasdatorn och serverdatorn körs databassystemet på samma dator under testen.

5.1.2 Mjukvarumiljö

Mjukvarumiljön som systemet körs på innefattar operativsystem, webbserver, program för frågegenerering och ett databassystem. Det finns en uppsjö av operativsystem att använda. Valet av operativsystem för både klientdatorn och serverdatorn spelar mindre roll, då det enda kravet är att programmen passar till operativsystemet. I testen används Linux men UNIX och Microsoft Windows fungerar också.

För att implementera OVIS krävs ett högnivåspråk och ett ramverk som går att modifiera så att vyer går att använda. De kandidater som finns är ASP.NET och Java Servlets, eftersom de har stora standardbibliotek än andra tekniker, såsom ASP och PHP. Valet mellan ASP.NET och Java Servlets är godtyckligt men Java Servlets har fler webbservrar som är *open-source,* vilket innebär att källkoden finns tillgänglig. Java stöds under flera plattformar och det gör även .NET plattformen, i till exempel Mono projektet, men den är vanligast i Windows miljön. I undersökningen används Java Servlets. Servlets är en öppen standard med flera implementerade servrar. Exempel på existerande servrar är WebSphere från IBM, WebLogic från BEA Systems, Jboss och Tomcat. Tomcat är open-source och är mer en renodlad webbserver i jämförelse med de andra servrarna och den är relativt lättkonfigurerad, vilket gör att Tomcat väljs till webbserverprogramvara i undersökningen.

För att skicka förfrågningar från en dator till webbservern behövs ett program eller skript för att automatisera testen. Det behövs alldeles för mycket förfrågningar per tidsenhet, vilket gör det svårt att praktiskt genomföra dem manuellt. Det program som genererar förfrågningar kallas här för lastgenererare. Lastgenereraren måste i undersökningen ha stöd för *cookies* eftersom sessioner används i testen. Lastgenereraren bör även ha stöd för att blanda olika förfrågningar enligt vissa statistiska modeller, för att simulera en verklig last på webbservern, där olika besökare besöker sidorna i olika antal och ordningar. Det finns en mängd typer av lastgenererare, Buret och Droze (2005) beskriver några utav dessa. Valet föll på Jmeter, främst på grund av det enkla gränssnittet och att det uppfyller de krav som finns.

Databassystemet som används i undersökningen ska vara ett relationsdatabassystem med stöd för SQL eftersom det används i spelscenariot. Databassystemet i testen är en MySQL databas. MySQL används främst för att den är lättillgänglig och har stöd för de funktioner som är nödvändiga. Olika databaser hanterar cachning och operationer olika vilket kan leda till att resultaten från testen blir annorlunda om olika databaser används. Testningen är till för att jämföra algoritmer i webbservern så resultaten får jämföras relativt.

5.1.3 Störningar vid tester

Nätverket i testen kan ge störningar i resultatet om det överbelastas. Om kollisioner inträffar ökar tiden det tar att skicka ett meddelande, exponentiellt på grund av alla omskickningar av meddelanden som krävs. Blir nätverkstrafiken så hög att kollisioner i nätverkstrafiken blir en flaskhals, minskar antal förfrågningar på grund av detta.

Webbservern och även OVIS algoritmen körs i flera trådar. För att skydda delad information används monitorer för att låsa informationen. När informationen är låst kan inga andra trådar använda den. Detta leder ofta till att det sker en blockering för väntande trådar vilket påverkar genomströmningen.

I testfallen är databassystemet på samma nod som beskrivs ovan. Detta leder till att databassystemet tar kraft och resurser från webbservern, vilket kan påverka prestanda negativt, jämfört med att ha databasen på en separat nod. Att ha databassystemet på en

annan nod kan dock ge problem med störningar i nätverkstrafiken.

Java använder sig av *garbage collection* vilket är en teknik för att avallokera oanvänt minne dynamiskt under körning. *Garbage collection* kräver att objekt i minnet traverseras och det leder till att en del av processortiden går åt till detta. Suns Java VM använder också en teknik som de kallar för *hotspot*. *Hotspot* är en metod för att kompilera Javas byte kod till maskinkod. Tanken med *Hotspot* är att endast kompilera den kod som används mest, därför kompileras inte koden förrän dessa kodbitar är identifierade av *runtime* miljön. Därför måste testfallen köras en ”uppvärmnings” tid innan systemet har kompilerat koden som används mest.

5.2 Tester

De tester som utförs i undersökningen sker på två olika algoritmer. Algoritmerna jämförs relativt mot varandra. De två algoritmerna är OVIS i original utförande och OVIS med tidsgränser på webbvyer (OVIS-TG). Först utförs ett jämförelsetest för att undersöka hur systemet beter sig. Därefter utförs lasttest och stresstest.

Lastgenereraren exekveras på en AMD Athlon XP 2000+ med 512 Mb i RAM. Webbservern i testet har en Intel Pentium 4 3Ghz med 1024 Mb i RAM. Versionerna på mjukvarorna som används i testen redovisas i tabell 1.

Mjukvara	***Version***
Apache Tomcat	5.5.9
Apache Jmeter	2.0.3
Ubuntu Linux	5.04
MySQL	4.1.10
Sun Microsystems Inc. Java	1.5.0_02-b09

Tabell 1: Mjukvara i testen.

5.2.1 Mätvärden

För att kunna jämföra algoritmerna behövs ett antal mätvärden. Labrinidis och Roussopoulos (2004) använder genomsnittlig svarstid på förfrågningar som mätvärde i sina jämförelser. Genomströmning, genomsnittlig last och processortid är andra mätvärden som kan användas. Svarstid definieras, som hur lång tid det tar, från att en förfrågan bearbetas tills ett svar är klart. Svarstid kan mätas både hos en klientdator eller på serverdatorn. Om svarstid mäts på klienten definieras svarstid, som hur lång tid det tar, från att en förfrågan lämnar klienten tills svaret från servern tas emot. Mäts svarstiden på servern definieras det, som hur lång tid det tar, från att en förfrågan kommer till systemet tills svaret skickas. Genomströmning definieras som hur många förfrågningar systemet svarar på under en tidsperiod. Genomsnittlig last på systemet, är det genomsnittliga antalet trådar i systemet, som väntar på att köra på processorn. Processortid är hur lång tid en process har exekverats på processorn. Alla dessa mätvärden är ett indirekt mått på hur effektiv en algoritm är. Till exempel, ju högre genomströmning desto bättre algoritm om förfrågningar hela tiden sker. Om genomsnittlig svarstid mäts på klientdatorn måste nätverkets inverkan beaktas.

Valet av mätningsmetod är godtycklig mellan processortid, genomsnittlig last, genomsnittlig svarstid och genomsnittlig genomströmning. Genomströmning och medelresponstid som mäts på klientdatorn väljs som mätvärden för att jämföra prestanda på algoritmerna. Dessa mätvärden väljs framför allt på grund av att Labrinidis och Roussopoulos (2004) har som mål att öka genomströmningen och minska responstiden. OVIS-TG algoritmen bör dessutom höja medelresponstiden när

materialiserade webbvyer väntar på att uppdateringar ska bli klara, för att garantera att inaktuell information inte skickas. Medelresponstid som mäts på klientdatorns väljs för att få en oberoende syn på svarstiden, jämfört med svarstiden på servern. Köas inkommande förfrågningar på webbservern blir mätvärdet missvisande.

När det gäller ålder på den information som levereras till klienten behövs mätvärde för att undersöka hur åldern på information i vyerna är. I undersökningen finns möjlighet att mäta om informationen är för gammal och hur gammal informationen är. För att kunna jämföra algoritmer och få förståelse på hur bra algoritmerna är, behövs ett mätvärde som beskriver hur gammal informationen är. Att säga att informationen är för gammal eller till exempel 20 sekunder för gammal säger inte tillräckligt mycket i sammanhanget. Istället ”normaliseras” åldern på informationen i vyerna. Eftersom det är känt hur gammal informationen är när den levereras och hur gammal den får vara, kan åldern vid leverans divideras med hur gammal den får vara. Resultatet blir att om värdet hamnar mellan noll och ett är information inte för gammal. Är informationen för gammal är vyn inaktuell och då får den ett värde över ett. Ett värde på till exempel två innebär att informationen som levereras är dubbelt så gammal som den får vara. Vid undersökningen av algoritmen jämförs medelvärdet av den levererade information mellan algoritmerna samt andelen av förfrågningar som får för gammal data. Åldern på den levererade informationen mäts endast på materialiserade vyer eftersom endast de som kan leverera inaktuell information.

5.2.2 Jämförelsetest

För att försäkra att systemet fungerar görs ett smoketest på webbspelet med OVIS algoritmen och utan någon cachning. Testfallet i tabell 2 körs sekvensiellt i 10 trådar med olika antal förfrågningar per sekund. Anledningen till att trådar används är för att simulera flera samtidiga användare som besöker sidor oberoende på varandra. Målet med testet är att visa att ju mer förfrågningar som skickas desto mer får webbservern arbeta. Detta mäts genom att mäta medelresponstiden vid de olika testfallen. Det förväntade resultatet är att medelresponstiden ökar ju mer förfrågningar som skickas per sekund. Ytterligare ett mål är att undersöka hur lång tid som behövs för testning innan mätvärdena blir pålitliga. Detta görs genom att köra samma tester på olika långa tidsperioder och sedan jämföra värdena.

Nr	***Förfrågan***	***Databasuppdatering***
1	Inloggningssida	Nej
2	Startsida	Nej
3	Arménsida	Nej
4	Handla armé	Ja
5	Allianssida / Handelssida	Nej
6	Handel med ett rike	Ja
7	Magi sida	Nej
8	Utöva magi	Ja
9	Logga ut	Nej

Tabell 2: Förfrågningar för testet.

De sidförfrågningar som uppdaterar databasen visas i tredje kolumnen i tabell 2.

Antalet förfrågningar i testet varieras från 100 förfrågningar per sekund och ökas med 50 förfrågningar per sekund tills servern inte klarar mer. En överbelastning upptäcks genom att kontrollera responstiden kontra genomströmningen. Om genomströmningen sjunker under nivån som servern lastas med och responstiderna bara ökar är servern överbelastad. Testerna körs med både OVIS och OVIS med samtliga vyer virtuella

(vilket inte ger någon cachning). OVIS-TG jämförs mot OVIS i load- och stresstesten.

5.2.3 Scenarion för load- och stresstestet

I detta delkapitel beskrivs de scenarion av förfrågningar som används i testningen. Scenarierna grundas från webbspelscenariot som beskrivs i kapitel 2. På grund av det stora antalet möjliga kombinationer av olika förfrågningar används bara scenarier där en viss typ av förfrågan endast kan ske en gång. Varje scenario består av tre delscenarion samt in- och utloggning. De tre delscenariona är följande:

1) Förfrågan på armésida och en förfrågan som genererar en beställning av en armé enhet.
2) Förfrågan på handelssida och en förfrågan som genererar handel med ett rike.
3) Förfrågan på magisida och en förfrågan som leder till att magi utförs.

Samtliga delscenarion har gemensamt att först sker en förfrågan mot en sida och sedan sker en förfrågan som genererar en databasuppdatering. Alla delscenarion i scenariot sker i en slumpmässig likformig ordning. Detta innebär att förfrågningar kan ske i ordningen 1, 2, 3 eller exempelvis 2,1,3.

5.2.4 Load- och stresstest

För att göra en så realistisk undersökning som möjligt används testdata som representerar en normal last på webbspelet som beskrevs i kapitel 2. En databas används med 10000 spelare och 500 lag. Varje spelare och lag representeras av en post i databasen där lagen representeras som en tabell och spelare i en annan. Spelare i samma lag lagras inte ihop utan är utsprida i tabellen. Varje lag tar 405 bytes i genomsnitt att lagra och varje spelare tar i genomsnitt 220 bytes. Samtliga spelare och lag har samma förutsättningar och databasen återställs innan varje test. För att minska risken för felmätningar körs varje test tre gånger och medianen av resultaten används.

Som beskrivs i början av kapitel 4 väntar verkliga klienter mellan sidförfrågningar. För att simulera detta används en så kallad *think time* mellan varje sidförfrågan (Menascé, 2002; Banga och Druschel, 1997). Lastgenereraren använder 200 trådar för att generera last, vilket är nödvändigt för att generera tillräckligt hög last, och samtidigt undvika en sekvensiell ordning av scenariot.

De variabler som förändras under testfallen är antal förfrågningar som lastgenereraren skickar och tidsgränsen som vyer har. Alla testfall har en uppvärmningstid på fyra perioder, där en period är 1000 sidförfrågningar. Uppvärmningstiden är 4 perioder eftersom då har alla estimater fått ett värde utifrån körningen. De får det eftersom g faktorn i formel 4 är en fjärdedel. När 4 perioder har kört har estimaterna värden från exekvering. Den tid som varje test körs fastställs av jämförelsetesten vilket visar hur långt ett test bör köras för att få tillräckligt pålitliga resultat. Vid varje period i testfallen sparas mätvärden till en fil som sedan sammanställs efter testen. θ värdet under testen är 0.70 för att få en andel materialiserade vyer. Kostnaden för att uppdatera/generera en vy är 150ms och kostnaden för att hämta en cachad vy är 10 ms. Dessa värden är de som Labrinidis och Roussopoulos (2004) använder i deras simulering.

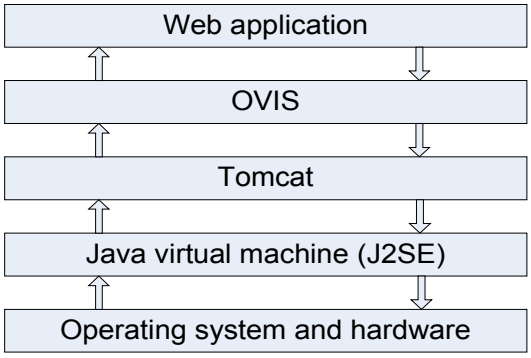
Antalet förfrågningar i testen varieras från 100 förfrågningar per sekund och ökas med 50 förfrågningar per sekund upp till 300 förfrågningar per sekund. Tidsgränserna varieras mellan 250, 500 och 1000 millisekunder. Även under lasttesten körs alla testfall tre gånger.

6 Implementation

I detta kapitel beskrivs hur arkitekturen är uppbyggd i implementationen. Samt de implementationsspecifika förändringarna som utfördes på grund av, att Labrinidis och Roussopoulos (2004) inte exakt beskriver hur vissa saker ska fungera.

6.1 Arkitekturen på implementation av OVIS

För att kunna samla information om vyer och kunna förändra materialiseringspolicyn på vyer, måste OVIS algoritmen implementeras ihop med webbservermjukvaran och webbapplikationen. Ett sätt att uppnå detta är att låta webbapplikationen hantera uppdateringen av statistik och hanteringen av hur materialiseringspolicyn implementeras. Ett alternativt sätt är att bygga OVIS algoritmen som ett lager på webbservern och låta webbapplikationen köras via detta lager. Det första alternativet tvingar programmeraren av vyer att hantera specifika aspekter ur OVIS algoritmen, såsom registrering av vyer, vilket lageralternativet inte gör. Lageralternativet måste dock se till att alla nödvändiga möjligheter i programmeringsgränssnittet finns kvar för programmeraren. I vissa fall kan portbarheten försämras om extra lager läggs till, eftersom applikationen då byggs på lagret och blir beroende av det. OVIS algoritmen implementeras som ett lager ovan Tomcat för att undvika att hantera specifika aspekter ur OVIS algoritmen vid konstruktionen av vyer. Figur 4 visar lagerstrukturen. Webbsidorna konstrueras av vyer och med *Java Server Pages* (JSP) eller *Servlets*.



Figur 4: Arkitekturen ur ett lagerperspektiv.

6.2 Implementationsspecifika förändringar

I arbetet av Labrinidis och Roussopoulos (2004) framgår inte hur vyer avallokeras när minnestillgången på serverdatorn börjar ta slut. Anledningen är att i simuleringarna antar författarna, att det finns en oändlig mängd minne för cachning. I praktiken när det tillgängliga RAM-minnet på serverdatorn tar slut måste minne frigöras för att undvika att använda virtuellt minne. Generellt sett, för att undvika att använda virtuellt minne, kan minnesmängden i serverdatorn dimensioneras, så att alla vyer garanterat kan få plats i RAM minnet eller så kan en metod för avallokering användas, här kallad ersättningspolicy. Det senare alternativet är att föredra, eftersom det kan vara svårt att räkna fram hur mycket minne som behövs, innan systemet tas i bruk. Det finns en mängd algoritmer för att välja vad som ska avallokeras när minnet minskar. Katsaros och Manolopoulos (2003) redovisar ett antal ersättningspolicys och kommer fram till att det inte finns någon policy som passar alla typer av applikationer. De grundläggande icke-modifierade policys är, enligt Katsaros och Manolopoulos (2003), *least recently used* (LRU), *least frequently used* (LFU) och *first in first out* (FIFO). Dessa algoritmer finns i olika varianter, som har en del förbättringar för att öka

sannolikheten, att de vyer som minst används tas bort först. Katsaros och Manolopoulos (2003) påpekar att de enklaste ersättningpolicys används i kommersiella system på grund av att de inte kräver komplexa datastrukturer och har relativt låg last på systemet. I implementationen av OVIS används FIFO policyn när minnestillgången minskar. FIFO policyn används för att den är enkel att implementera. FIFO policyn innebär att den först skapade vyn av vyerna i minnet tas bort när minnestillgången minskar. Om den borttagna vyn behövs, skapas den igen och nästa vy i kön tas bort, om minnestillgången minskar igen.

OVIS algoritmen aktiveras i intervall och avgör då vilka vyer som ska materialiseras och vilka som inte ska vara materialiserade (Labrinidis och Roussopoulos, 2004). Intervallet kan antingen baseras på tidsintervall eller på antal förfrågningar. Om algoritmen är baserad på ett tidsintervall, kan förändringen av materialisering till exempel ske var tionde minut. Om intervallet är baserat på förfrågningar sker förändringen till exempel, vid var tusende sidförfrågan. I undersökningen sker förändringen av materialiseringspolicy efter ett antal förfrågningar, efter den senaste förändringen. Denna metod används för att OVIS algoritmen ska få tillfälle att förändra materialiseringspolicyn på webbvyer, oberoende på hur lång tid testen körs.

Det framgår ej i Labrinidis och Roussopoulos (2004) hur specifikt relationer definieras. I implementationen definieras relationer med dess nyckel, vilket gör att webbvyer blir beroende på en databasrelation (tabell) och en frivillig nyckel. Nyckeln används för att snabbt hitta de webbvyer som är beroende av en relation när den uppdateras. Om webbvyer beror på en relation, men inte en specifik nyckel, förloras stor effektivitet när antalet webbvyer som är beroende av relationen blir stort. Detta beror på att alla vyer måste köas för uppdatering och uppdateras även, om de inte datamässigt sett, behöver detta. I praktiken kan det vara svårt att veta exakt vilka nycklar en vy beror på men det finns lösningar på problemet. Candan et al. (2002) beskriver en sådan lösning.

7 OVIS-TG

Det här kapitlet beskriver de förändringar i OVIS algoritmen som utvecklades under arbetet, för att kunna garantera att inaktuell information, inte lämnar webbservern.

7.1 Införande av tidsgräns

För att uppfylla ett av målen med undersökningen, måste en modifierad variant av OVIS algoritmen utvecklas. Den varianten måste kunna garantera att inaktuell information inte lämnar webbservern. För att kunna garantera detta har varje webbvy en variabel, som beskriver hur länge informationen i en webbvy är aktuell, efter att en uppdatering har skett på en beroende relation. För att garantera att tidsgränsen som är inställd på en webbvy inte bryts, måste en förfrågan på en webbvy med ej uppdaterad information vänta tills vyn är uppdaterad. En webbvy kan då antingen uppdateras i samma tråd som hanterar förfrågan eller i en separat tråd, som har till uppgift att uppdatera de materialiserade vyerna. Att hantera uppdateringen i samma tråd har fördelen, att det är relativt enkelt att implementera, men uppdateringar måste då synkroniseras för att förhindra, att en uppdatering på en vy inte utförs fler gånger än nödvändigt. Om uppdateringen köas och sedan körs i en separat uppdateringstråd, uppdateras webbvyn inte fler gånger än nödvändigt. Systemet lägger inte till en vy i kön som redan finns där. Det kan ta längre tid, ifall flera andra uppdateringar står före i uppdateringskön till uppdateringstrådarna, om uppdateringarna körs i separata trådar. I undersökningen används separata uppdateringstrådar för att uppdatera materialiserade vyer, vilket gör att systemet undviker extra uppdateringar.

Som beskrevs i kapitel 2 använder OVIS statistik och matematiska funktioner för att avgöra om en vy ska vara materialiserad eller inte. Funktionen som beräknar hur mycket en vy kostar att ha materialiserad, förändras så att vyer som är materialiserade och ofta uppdateras på grund av inaktuell information, inte materialiseras om det kostar mer att materialisera än att inte göra det. Kostnaden för att materialisera en vy i originalutförandet av OVIS beräknas enligt formel 6 som även beskrivs i kapitel 2:

$$Cost_{mat}(w) = N_{acc} \cdot A_{hit} + R_r \cdot N_{upd} \cdot U_{mat} \qquad (6)$$

Definitionen för att beräkna kostnaden för en materialiserad webbvy måste förändras för att ta hänsyn till hur mycket det kostar att vänta på att uppdateringar ska ske, när en vy med inaktuell information efterfrågas. För att hålla reda på hur mycket extra det kostar att vänta, definieras en variabel, U_{wait} , som är den genomsnittliga tiden som systemet har fått vänta på en uppdatering, efter att en förfrågan på vyn har skett. Definitionen ovan kan då skrivas om till:

$$Cost_{mat}(w) = N_{acc} \cdot A_{hit} + R_r \cdot N_{upd} \cdot U_{mat} + U_{wait} \cdot N_{TooOld}$$

N_{TooOld} är hur många förfrågningar som sker när åldern på vyn är inaktuell, vilket leder till att förfrågan blockeras tills den är uppdaterad. $U_{wait} \cdot N_{TooOld}$ är den extra kostnaden för att alltid skicka uppdaterade vyer (inom tidsgränsen). N_{TooOld} och U_{wait} estimeras med avseende på tidigare perioder likt den ursprungliga definitionen.

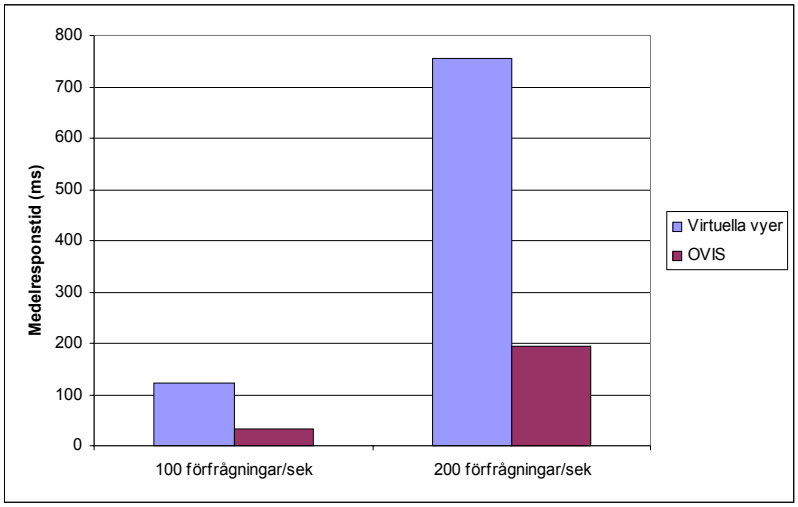
Svar som ska skickas över Internet kan inte garanteras att komma inom en viss tid utan systemet eller användarna måste försäkra sig om, att svaren kommer fram i tid (Li et al., 2003). Administratören för en webbplats måste alltså ta hänsyn till, att det tar tid att skicka en sida, när en tidsgräns sätts för en webbvy.

8 Resultat

I det här kapitlet presenteras resultat från benchmarktesterna. Kapitlet börjar med att presentera jämförelsetesten och därefter load- och stresstesten. Det är 12 tester som presenteras i jämförelsetesten och ytterliggare 6 tester för att ta reda på standardavvikelsen mellan 3 och 10 minuters tester. 90 tester ligger bakom de resultat som presenteras i load- och stresstesten. Ytterligare ett fåtal tester har utförts för att utvärdera testmetoden.

8.1 Jämförelsetesten

Testen visar att det räcker att köra testerna tre minuter för att få tillförlitliga värden. Tester som körs 10 minuter ger väldigt lika medelresponstider och standardavvikelser mot de tester som körs i 3 minuter.

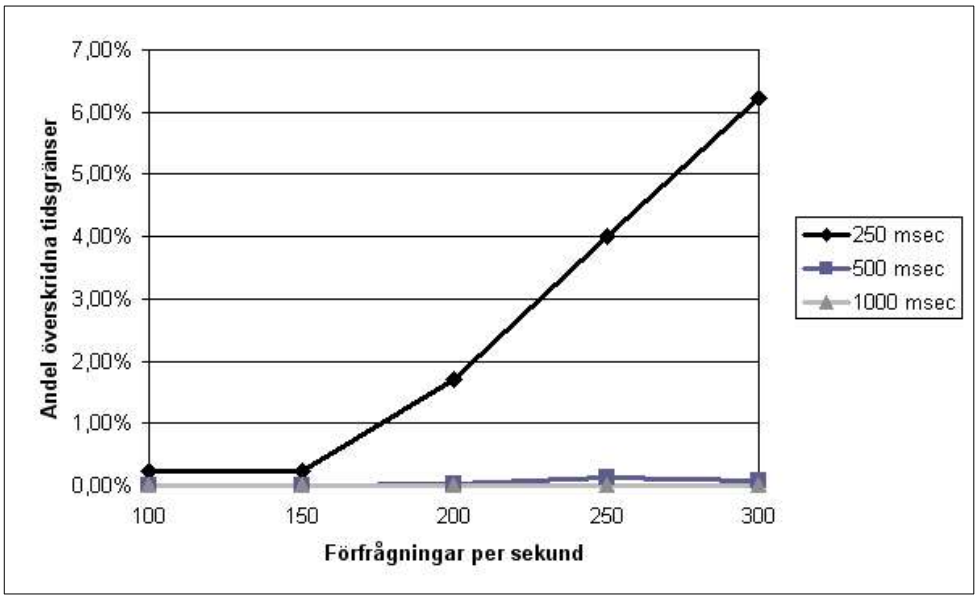


Figur 5: Responstider med och utan OVIS.

Testen visade att OVIS algoritmen förbättrar effektiviteten jämfört med att inte använda den, se figur 5. Medelresponstiden är betydligt mindre för OVIS algoritmen vid både 100 och 200 förfrågningar per sekund. Eftersom OVIS algoritmen ger bättre responstid än utan cachning studeras den vidare i load- och stresstesten.

8.2 Load- och stresstesten

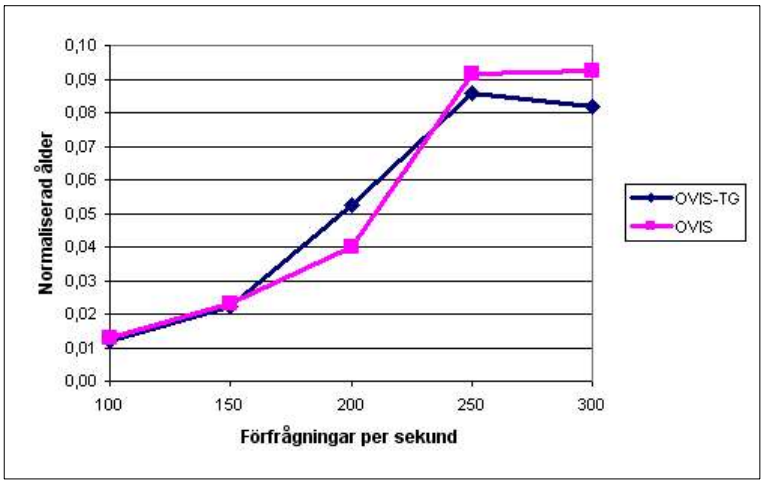
Load- och stresstesten visar att de två algoritmerna presterar ungefär lika när det gäller prestanda. OVIS algoritmen missar ibland de tidsgränser som anges i testen i webbspelet. Figur 6 visar hur andelen vyer som levererar inaktuell information av de materialiserade vyerna i OVIS algoritmen. Vid 500 och 1000 millisekunder är det knappt några vyer som är inaktuella när de levereras men andelen ökar snabbt vid 250 millisekunder.



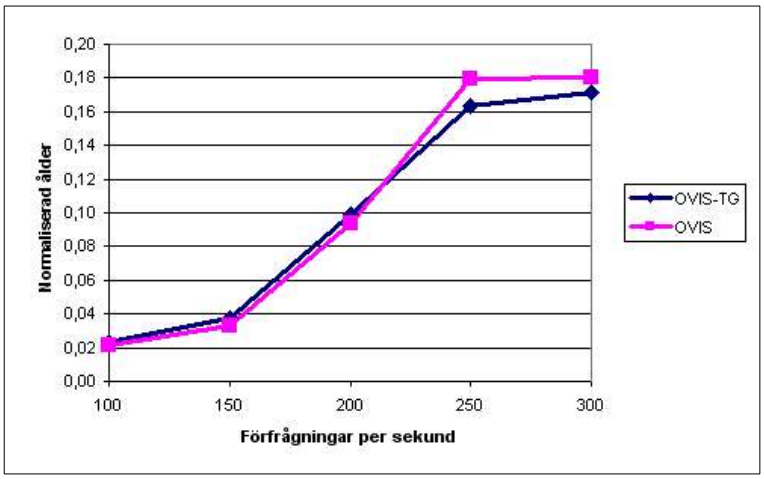
Figur 6: Andel överskridna tidsgränser i testet av OVIS algoritmen.

OVIS-TG algoritmen visade sig inte leverera några vyer med inaktuell information, vilket är tanken med algoritmen.

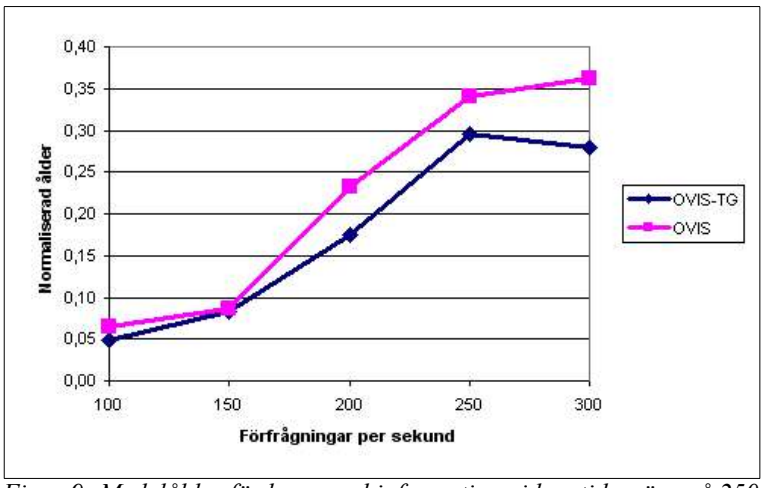
Medelåldern på den levererade informationen redovisas i figur 7, 8 och 9. En medelålder under ett betyder att flertalet av de levererade vyerna innehåller uppdaterad information.



Figur 7: Medelålder för levererad information vid en tidsgräns på 1000 millisekunder.



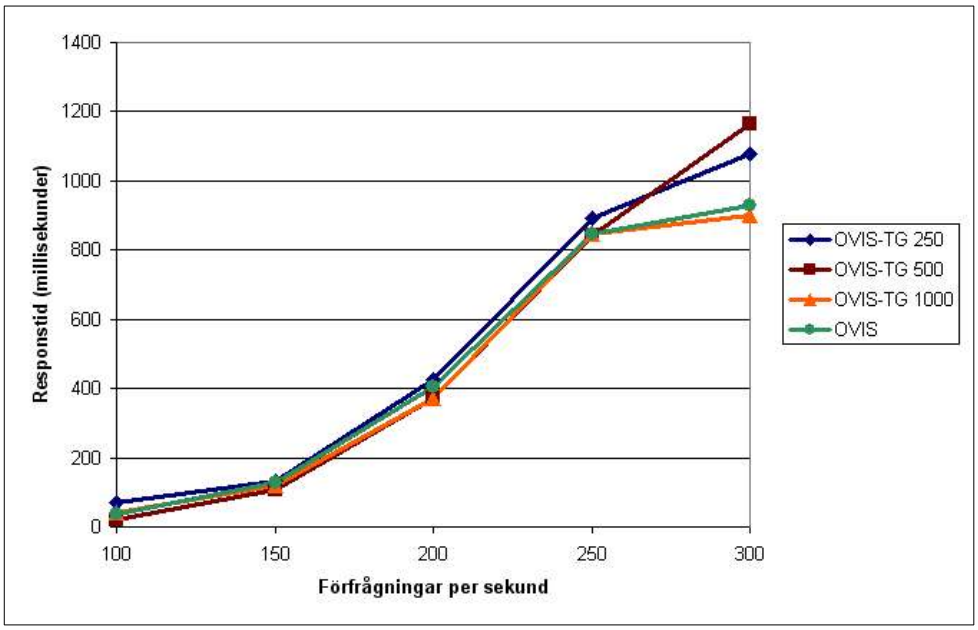
Figur 8: Medelålder för levererad information vid en tidsgräns på 500 millisekunder.



Figur 9: Medelålder för levererad information vid en tidsgräns på 250 millisekunder.

Medelåldern för 500 och 1000 millisekunder, figur 7 respektive 8, är ganska lika för både OVIS och OVIS-TG. Vid en tidsgräns på 250 millisekunder, figur 9, får OVIS algoritmen högre medelålder på de levererade vyerna, vilket kan förklaras med att andelen tidsgränser som överskrids, är betydligt större i det fallet.

Prestandan i de olika algoritmerna är väldigt lika, vilket figur 10 visar. En jämförelse av medelresponstiderna visar att båda algoritmerna följer varandra åt.



Figur 10: Medelresponstid vid olika belastningar

Vid en förfrågningsfrekvens över 250 millisekunder blir servern överbelastad. Anledningen till att medelresponstiden bara ökar lite mellan 250 och 300 förfrågningar per sekund är att det är stor skillnad mellan olika förfrågningars svarstider. Vissa förfrågningar i testet tog mer än 30 sekunder när servern var överbelastad. Eftersom materialiserade vyer fortfarande levereras snabbt vid en överbelastning (men inte i lika hög grad) är medelresponstiden fortfarande ganska låg. Varje test genomfördes tre gånger och vid 300 förfrågningar per sekund skiljer medelresponstiden mellan testerna relativt mycket. Detta syns inte i medianvärdena som presenteras i figur 10. Mätvärdena vid 300 förfrågningar per sekund varierade mellan 850 millisekunder till 1250 millisekunder.

9 Analys

För att undersöka de båda algoritmerna gjordes ett antal tester. Testen baserades på scenarion från ett webbspel. I testerna varierades antalet förfrågningar per sekund och tidsgränserna för när information i vyerna blev för gamla för att levereras. Under testningen skedde hela tiden databasuppdateringar mot systemet genom att vissa förfrågningar genererar uppdateringar. Förfrågningarna blandades i olika trådar för att undvika sekvensiella testfall. I föregående kapitel presenteras resultaten från de tester som genomfördes. Resultaten visar att både OVIS och OVIS-TG algoritmerna ger likvärdiga responstider, vilket pekar på att algoritmerna skalar ungefär lika eller att det finns andra programdelar i webbserverapplikationen som tar längre tid. En orsak som höjde medelresponstiden i testen av OVIS och OVIS-TG är att vid varje inloggning sker en förfrågan mot databasen. I och med att inloggningen inte cachas tar denna längre tid än en vanlig förfrågan, när användaren väl är inloggad. De förfrågningar som genererar uppdateringar tar också längre tid på grund av att databasen kontaktas vid varje uppdatering. För att visa hur bra OVIS algoritmen är, kan tester göras på scenarion där uppdateringar inte kommer från förfrågningar, utan från andra transaktioner i systemet. Sådana tester gjordes inte på grund av, att på dynamiska webbplatser sker oftast uppdateringar via webben. I webbspelscenariot måste användare logga in, innan de kan se sina sidor, vilket gör att systemet använder databasen vid förfrågningar.

De skillnader som resultaten visar mellan OVIS och OVIS-TG är, skillnader på medelåldern på levererade vyer och andelen överskridna tidsgränser. Tidsgränsen då flest vyer bröt mot gränsen var 250 millisekunder. Samtidigt var responstiden kring 1 sekund. Eftersom gränsen var 250 millisekunder är nyttan av OVIS-TG algoritmen inte särskilt stor i spelscenariot. OVIS-TG algoritmen klarade av att hantera gränserna och samtidigt behålla samma responstider som OVIS algoritmen. Om en webbplats vill garantera att informationen är aktuell, när den levereras, kan OVIS-TG algoritmen användas.

Jämförelsetesten visar att OVIS algoritmen ger kortare responstider än att inte använda cachning, vilket också simuleringarna i Labrinidis och Roussopoulos (2004) visar. Skillnaden mellan simuleringarna och benchmarktesten i denna undersökning är att benchmarktesten gjordes i en riktig webbserverapplikation och inte i en simulator. Webbapplikationen som testerna genomfördes i är specialiserad medan de simuleringar som utfördes av Labrinidis och Roussopoulos (2004) är mer generaliserade mot databasbaserade webbapplikationer. I simuleringarna använde författarna fler relationer än vad benchmarktesterna använde i denna undersökning. Det är dock oklart om författarna använder relationsnycklar i beroenden för vyer.

Labrinidis och Roussopoulos (2004) använder en Zipf liknande distribution av förfrågningar. Zipf distributionen beskrivs i Breslau et al. (1999) men kortfattat så distribueras förfrågningar enligt sannolikheten att den i:te populäraste sidan besöks är proportionell mot $\frac{1}{i^a}$ där $0 < a \leq 1$.

Detta arbete använder inte Zipf distributionen i testningen utan distributionen i testen grundar sig på scenario från webbspelet. Hade Zipf distributionen används i testen så bör resultaten förändras något. Om de populäraste förfrågningarna hade lett till uppdateringar av relationer, hade förmodligen medelåldern på den levererade informationen ökat något.

Antalet uppdateringar var i testen beroende på antalet förfrågningar per sekund. Om

andelen uppdateringar hade varit större är det möjligt att resultatet hade blivit annorlunda. Uppdateringstrådarna hade då fått fler vyer att uppdatera, vilket kan leda till att responstiderna ökar för OVIS-TG algoritmen och åldern på informationen hade ökat i OVIS fallet. Uppdateringarna i webbspelet var främst koncentrerade till små grupper och enskilda användare. I de applikationer där uppdateringar får en inverkan på ett större antal vyer får uppdateringstrådarna även då mer att uppdatera.

Labrinidis och Roussopoulos (2004) simulerade även *quality of data* (QoD), se formel 3, vilket inte mättes i testerna. Anledningen till detta var att testerna jämförde OVIS och OVIS-TG algoritmen ur prestandasynpunkt och hur gammal den levererade informationen var.

I testerna användes en uppvärmningtid på fyra perioder. Alla vyer som enligt OVIS algoritmen ska materialiseras, materialiseras inte efter dessa fyra perioder. OVIS algoritmen har en maxgräns på hur många vyer som får ändra materialiseringspolicys per period. Fler materialiserade vyer ökar risken för att inaktuell information levereras. Andelen materialiserade vyer stabiliserades inte efter fyra perioder utan efter ungefär 15 perioder, vilket kan göra att andelen överskrida tidsgränser är något mindre, i jämförelse om mätningarna skulle utförts under senare perioder i testen.

Det finns faktorer som inte beaktades i testningen men som existerar på Internet. Informationen på Internet går via routrar som dirigerar trafiken. Routrar är ofta den begränsande faktorn, för hur mycket trafik som kan nå en webbserver. Routrar är vanligtvis snabbare på att hantera trafik än webbservrar är på att hantera förfrågningar. I praktiken kan det komma tillräckligt med förfrågningar för att en server ska bli överbelastad. Brandväggar är precis som routrar en begränsande faktor på informationsflödet men klarar vanligtvis av att hantera informationen snabbare än en webbserver. Mjukvarubrandväggar som arbetar på samma dator som webbservern påverkar prestandan negativt eftersom då minskas datorresurserna för webbservern.

I benchmarktesten var det inga spikar i belastningen utan den ökades från test till test. I praktiken är inte belastningen jämn utan det förekommer variationer i belastningen. När spikar sker i belastningen och antalet uppdateringar ökar som följd, kommer cachade vyer generera större last på systemet, eftersom uppdateringarna ökar.

10 Slutsats

Belastningen i dagens webbservrar är höga. För att behålla besökare på webbplatserna vill företag, att deras system ska klara av den belastning, som systemet ställs inför. Detta krävs för att besökare inte ska lämna platsen (Li et al., 2002; Menascé, 2002). Ett vanligt sätt att avlasta ett system är att använda replikering av databaser och webbservrar, samt fördela lasten mellan de replikerade delarna. För att öka effektiviteten på servrarna används cachning. Labrinidis och Roussopoulos (2004) redovisar OVIS algoritmen som är en webbserverbaserad algoritm för cachning och använder materialiserade vyer, vilket andra cachetekniker inte använder.

De tre delmål som definieras i kapitel 3 återkopplas här.

1. **Införa garantier på information i materialiserade vyer i OVIS**

   De förändringar i OVIS algoritmen som implementerades beskrivs i kapitel 7. Hanteringen av förfrågningar förändrades så att en förfrågan på en inaktuell vy väntar tills uppdateringen är klar och därigenom uppnås en garanti på informationen. En kostnadsfunktion förändras för att ta hänsyn till att förfrågningar kan få vänta på uppdateringar på vyer.

2. **Implementera OVIS och OVIS-TG algoritmen i ett webbspel**

   OVIS algoritmen implementerades i Java utifrån utifrån pseudokod och teori i Labrinidis och Roussopoulos (2004). OVIS algoritmerna implementerades som ett lager ovan webbservern Tomcat. Som ersättningspolicy när minnet minskar till en viss gräns används FIFO. En optimering av vyers beroende på relationer utfördes för att öka prestanda på uppdatering av vyer.

   Det behövs inga större förändringar för att realisera OVIS-TG algoritmen när väl implementationen av OVIS är klar. Den stora skillnaden är att förfrågningar måste vänta på att uppdateringar av vyer är klara och det kan lösas relativt enkelt med en monitor i Java.

3. **Testa prestanda och ålder på den information som levereras till användare**

   För att undersöka algoritmerna utfördes ett antal benchmarktester. I testerna användes en serverdator och en klientdator. Databassystemet samt webbservern kördes på servern och en lastgenererare kördes på klienten. Under testerna mättes medelresponstiden, medelålder för information i vyer samt andelen överskrida tidsgränser. Labrinidis och Roussopoulos (2004) simuleringar visar att OVIS algoritmen ger en prestandaförbättring mot endast cachning. Testerna i denna undersökningen visar att OVIS algoritmen är bättre än ingen cachning. Hur stor förbättringen blir, är svårt att säga, men jämförelsetesten som utfördes gav mer än en halvering av medelresponstiden när OVIS algoritmen användes.

   Testerna visar även att OVIS-TG algoritmen fungerar, men skillnaden mellan OVIS algoritmen och OVIS-TG algoritmen visade sig inte förrän vid relativt korta tidsgränser. OVIS-TG algoritmen är idag inte så användbar i liknande webbapplikationer. Hade OVIS algoritmen i webbspelet haft problem vid tidsgränser på flera sekunder hade OVIS-TG algoritmen kommit bättre till hands. I specifika fall där genereringen av en vy tar relativt lång tid på grund av att en stor mängd data ska processas är materialisering ett måste för att inte få långa responstider. OVIS-TG algoritmen kan användas till sådana applikationer för att både ge prestanda och garantier för att informationen är aktuell. Ett exempel på en vy som tar lång tid på sig att genereras kan vara kurser som sammanställs från dataserier som sträcker sig över en lång tidsperiod. Dataserier kan till exempel vara aktiekurser eller valutakurser över flera år som presenteras som diagram.

Med vyer som tar lång tid att generera kan systemet byggas så att uppdateringstrådarna prioriterar dessa, för att så snabbt som möjligt uppdatera de vyerna. Blandas vyer som har tidskrav med vyer som inte har det, får systemet realtidsliknande egenskaper med schemaläggning.

Den kritiska faktorn när det gäller OVIS-TG algoritmen är antalet uppdateringar. Även när det gäller cachning med invalidering är antalet uppdateringar den kritiska faktorn. Den stora skillnaden är att OVIS-TG algoritmen utför uppdateringar i en separat tråd och cachning med invalidering utför uppdateringar när en förfrågan sker. OVIS-TG algoritmen hanterar täta uppdateringar mer effektivt, då de uppdateringar som redan är köade för uppdatering, inte läggs i uppdateringskön igen.

OVIS algoritmen kan användas i de webbsystem som cachning redan används och OVIS-TG algoritmen kan förutom webbspel också användas i webbplatser, där det är viktigt att kunna garantera att informationen är uppdaterad, som till exempel auktionswebbplatser och webbplatser som hanterar väldigt mycket information för att konstruera vyer.

10.1 Diskussion

Detta arbete implementerar och testar en cachningsalgoritm för servergenererade dynamiska hemsidor. I introduktionen nämndes andra dynamiska hemsidor som byggs med exempelvis flash och java-applets. Dessa typer av dynamiska sidor kan innehålla information som hämtas från databaser som klientdatorer kopplar upp sig mot. Webbapplikationen flyttas från en server till klienten. OVIS algoritmen är inriktad på webbservrar och kan således inte användas för att snabba upp en sådant informationsflöde. Databascachning kan då användas istället för att öka prestanda för databasservrarna i detta fallet.

Cachning kan vara användbart i chatrum och annan realtids kommunikation mellan människor via webben när små datamängder används. När små laster borde inte materialisering behövas eftersom systemet då kan hantera förfrågningar. Chatrum har relativt höga krav på aktuell information, något som OVIS-TG algoritmen kan användas till att garantera vid höga belastningar.

Proxyservrar används för att lagra statisk information såsom bilder och vanliga html-dokument. De kan fortfarande användas i framtiden och existera tillsammans med serverbaserad cachning. Det finns nu även *Content Delivery Networks* (CDN) som är samling server på utspridda ställen i världen som har till uppgift att sprida stora mängder information till slutanvändare. Informationen sprids mellan servrarna i nätverken för att uppnå låga responstider. Akamai (2005) tillhandahåller ett *Content Delivery* nätverk.

WebServices är en teknik på frammarsch på Internet. Hur väl cachning går att applicera på den tekniken beror vilka tjänster som används och hur de används men hög belastade tjänster kan ha användning både av cachning och replikering precis som webbplatser.

10.2 Framtida arbete

Vid ytterligare studier av OVIS och OVIS-TG algoritmerna skulle det vara intressant, att undersöka hur användbar OVIS algoritmen är, i jämförelse mot andra cachningsalgoritmer kombinerat med databascachning samt hur användbar OVIS-TG algoritmen blir med vyer som tar lång tid att generera/uppdatera.

OVIS algoritmen har visat sig fungera i webbspelscenariot. För att kunna använda systemet i större skala, med flera tusen olika sidor, med olika sammansättningar av

vyer finns en möjlighet att implementera OVIS algoritmen i ett *Content Management System* (CMS). En del CMS system använder även databassystem för att lagra sidorna på webbplatsen. Ett framtida arbete är, att ta fram en bra modell, för att representera vyer, webbsidor och hur viktiga vyerna på sidorna är. I den nuvarande implementationen används speciella JSP taggar som beskriver vilken vy, som ska placeras på sidan och vilka parametrar vyn har. Vyerna är konstruerade i Java, vilket kan utökas för att passa ett CMS system.

Candan et al. (2002) beskriver en lösning på problemet med att veta vilka uppdateringar som berör vyer. Ett framtida arbete är att kombinera en sådan lösning för att automatiskt veta vilka relationer och nycklar som en vy beror på. I implementation som används i detta arbete definierar varje vy vilka relationer (och nycklar) som de beror på. Komplexa vyer kan ha många beroenden, vilket kan göra det svårt att hitta alla.

Ersättningspolicyn som används i implementation är FIFO. Ett framtida arbete kan undersöka om olika ersättningspolicys förändrar resultatet om många vyer används.

I undersökningen används fasta kostnader för vyuppdatering och hämtning. En ytterligare förbättring, vore att implementera ett system som under körning förändrar kostnaderna, beroende på hur krävande en vy är att generera eller hämta. Olika komplexa vyer med varierande mängd data tar olika lång tid att generera och mängden data kan variera under tiden systemet körs. Att systemet då upptäcker att det tar längre tid/kraft att uppdatera en vy kan göra att algoritmens beslut förbättras och prestanda kan förbättras.

I implementationen används monitorer för att få exklusiv tillgång till alla vyer när algoritmen aktiveras periodvis. Ett framtida arbete är att undersöka hur stor påverkan detta har på genomströmningen och om det är möjligt att låta algoritmen köra asynkront vilket gör att förfrågningar kan behandlas under algoritmens aktiva period. Det största problemet bör vara hur statistiken hanteras, eftersom den ändras när algoritmen använder den.

Referenser

Akamai [Elektronisk]. 2005. Tillgänglig: http://www.akamai.com/en/html/services/content_delivery_services.html. [20050601].

Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, J. & Reinwald, B., 2003. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of 29th International Conference on Very Large Data Bases*, Berlin, Germany, September 9-12, pp. 718-729.

Bamford, R., Ahad, R. & Pruscino, A., 1999. A Scalable and Highly Available Networked Database Architecture, In *proceedings of the 25th VLDB Conference*, Edinburgh, Scotland, pp. 199-201.

Banga, G. & Druschel, P., 1997. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997

BEA Systems - BEA WebLogic Server [Elektronisk]. 2005. Tillgänglig: http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/ [20050228]

Breslau, L., Cao, P., Fan, L., Phillips, G. & Shenker, S. 1999. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of INFOCOM 1999*, New York, USA, pp. 126-134.

Buret, J. & Droze, N., 2005. An overview of load test tools [Elektronisk]. Tillgänglig: http://clif.objectweb.org/load_tools_overview.pdf [20050422]

Candan, K. S., Agrawal, D., Li, W.-S., Po, O. & Hsiung, W.-P., 2002. View Invalidation for Dynamic Content Caching in Multitiered Architectures. In *Proceedings of the 28th Very Large Data Bases Conference*, Hongkong, China

Cao, P. & Liu, C., 1998. Maintaining Strong Cache Consistency in the World-Wide Web. *IEEE Transactions on Computers*, 47(4), 445-457, April

Cassone, G., Elia, G., Gotta, D., Mola, F. & Pinnola, A., 2001. Web Performance Testing and Measurement: a complete approach. In *CMG ITALIA 2001 and CMG USA 2001 conference proceedings*.

Cho, J. & Garcia-Molina, H., 2003. Effective Page Refresh Policies for Web Crawlers, *ACM Transactions on Database Systems*, Vol. 28, No. 4, December.

Datta, A., Dutta, K., Thomas, H., VanderMeer, D., Ramamritham, K. & Fishman, D., 2001. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *proceedings of the 27th international conference on very large data bases (VLDB)*, Rome, Italy, 667-670.

Iyengar, A. & Challenger, J., 1997. Improving Web Server Performance by Caching Dynamic Data. In *proceedings of the USENIX Symposium on Internet Technologies and Systems*. Monterey, California, USA, December 8-11.

Iyengar, A., MacNair, E. & Nguyen, T., 1997. An Analysis of Web Server Performance. In *Proceedings of the IEEE 1997 Global Telecommunications Conference (GLOBECOM '97),* Phoenix, AZ, November

Jacobsen V., 1988. Congestion avoidance and control. In *Proceedings of SIGCOMM '88,* Stanford, CA, Aug, 314-329

Katsaros, D. & Manolopoulos, Y., 2003. Cache Management for Web-Powered Databases. *Web-Powered Databases*. IDEA Group Publishing. pp. 203-244.

KIA Index [Elektronisk]. 2005. Tillgänglig: http://www.mediacom.it-norr.se/23/kia/final_web/con_web_month_mt.asp?Sort=PI&MonthSelect=200501&Sort_ret=desc. [20050216]

Labrinidis, A. & Roussopoulos, N., 2004. Exploring the tradeoff between performance and data freshness in database-driven Web servers. In *proceedings of the VLDB Conference*, Toronto, Canada, 13(3), 240-255.

Li, W.-S., Po, O., Hsiung, W.-P., Candan, K. S., & Agrawal, D., 2003. Freshness-driven adaptive caching for dynamic content Web sites. *Data and Knowledge Engineering,* 47(2), North-Holland, pp. 269-296

Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G. & Naughton, J. F., 2002. Middle-Tier Database Caching for e-Business. In *proceedings of ACM SIGMOD*, Madison, Wisconsin, USA, June 4-6.

Manuel, P.D. & AlGhamdi, J., 2003 . A data-centric design for n-tier architecture. *Information Sciences*, 150(3-4), Elsevier Science, pp. 195-207

Menascé, D. A., 2002. Load Testing of Web Sites. *IEEE Internet Computing*, July/August.

Scalability : Oxford Reference Online [Elektronisk]. 2005. Tillgänglig: http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t11.e4606. [20050301]

Silberschatz, A., Galvin, P. B. & Gagne, G., 2003. *Operating System Concepts*. 6th edition, John Wiley & Sons, Inc. USA.

TimesTen: Cache - Real-Time Dynamic Data Caching System [Elektronisk]. 2005. Tillgänglig: http://www.timesten.com/products/cache.html. [20050228].

XCache Technologies [Elektronisk]. 2005. Tillgänglig: http://www.xcache.com/home/default.asp?c=45&p=352. [20050210].

Yagoub, K., Florescu, D., Issarny, V. & Valduriez, P., 2000. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of 26th International Conference on Very Large Data Bases*, Cairo, Egypt, pp. 188-199.

Bilaga 1 – Mätvärden

Req/sec	Virtuella vyer	OVIS
100	121,3	32,7
200	754,8	195,2

Tabell 3: Medelsesponstid (millisekunder) utan cachning och med OVIS.

Req/sec	3 min	10 min
100	32,1	29,8
200	78,2	75,2

Tabell 4: Standardavvikelse för OVIS algoritmen.

Medelålder OVIS	250 msec	500 msec	1000 msec
100	0,06	0,02	0,01
150	0,09	0,03	0,02
200	0,23	0,09	0,04
250	0,34	0,18	0,09
300	0,36	0,18	0,09

Tabell 5: Medelålder för OVIS.

Medelålder OVIS-TG	250 msec	500 msec	1000 msec
100	0,05	0,02	0,01
150	0,08	0,04	0,02
200	0,17	0,10	0,05
250	0,30	0,16	0,09
300	0,28	0,17	0,08

Tabell 6: Medelålder för OVIS-TG.

Req/sec	250 msec	500 msec	1000 msec
100	0,24%	0,00%	0%
150	0,22%	0,00%	0%
200	1,71%	0,02%	0%
250	4,01%	0,12%	0%
300	6,21%	0,08%	0%

Tabell 7: Andel överskrida tidsgränser för materialiserade vyer i OVIS algoritmen.

Req/sec	250 msec	500 msec	1000 msec
100	0,00%	0,00%	0,00%
150	0,00%	0,00%	0,00%
200	0,00%	0,00%	0,00%
250	0,00%	0,00%	0,00%
300	0,00%	0,00%	0,00%

Tabell 8: Andel överskrida tidsgränser för materialiserade vyer i OVIS-TG algoritmen.

Req/sek	OVIS-TG 250	OVIS-TG 500	OVIS-TG 1000	OVIS
100	69	21	42	36
150	134	108	121	129
200	424	371	370	406
250	892	841	845	847
300	1076	1163	899	931

Tabell 9: Medelresponstid för OVIS-TG vid 250, 500, 1000 milliekunders tidsgräns samt OVIS.

Period	Antal materialiserade vyer
1	0
2	13
3	26
4	39
5	52
6	65
7	78
8	91
9	104
10	117
11	130
12	143
13	156
14	169
15	182
16	181
17	180
18	193
19	192
20	191
21	190
22	189
23	188
24	191
25	190
26	189
27	188
28	189
29	188
30	187
31	186
32	185

Tabell 10: Antal materialiserade vyer i ett typiskt testfall.