Användarna förtjänar data som är korrekt

Låt inte antalet långa verksamhetstransaktioner explodera!

2010-12-20: Sven-Håkan Olsson

LÖSA TRANSAKTIONER Sammanhållna transaktioner är ett traditionellt sätt att se på lagrande av data så att informationen hålls ihop och blir konsistent. Men inom SOA och vid andra situationer kanske vi inte kan använda sammanhållna transaktioner. Vad gör vi då?

I en tidigare spaning utlovade jag en text om hur man kan göra för att hålla information korrekt och konsistent. En konst som på senare tid märkligt nog prioriteras ner rejält. Visst måste man ibland kompromissa mellan olika egenskaper som flexibilitet, skalbarhet, tillgänglighet, färskhet och korrekthet, men jag tycker att det stundtals sker på ett oreflekterat sätt.

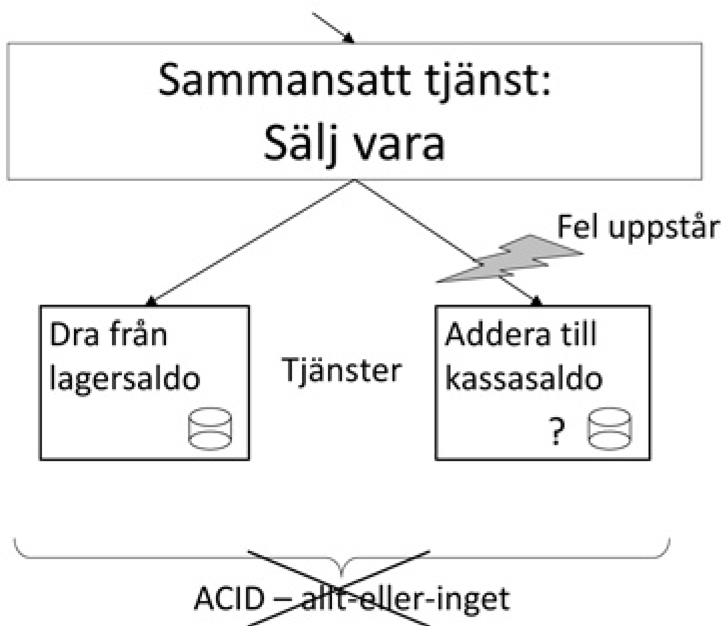
Den traditionella sammanhållna transaktionen beskrivs ofta med förkortningen ACID: Atomicity - Consistency - Independency - Durability. Andra namn är unit-of-work, commit-rollback, obrytbar skrivning, allt-eller-inget och odelbar uppdatering.

Ett exempel är att jag med hjälp av en kassa-applikation säljer en vara. Då ska jag dra bort från lagersaldot och samtidigt addera till kassasaldot. Om endast den ena uppdateringen görs men den andra störs av något skäl, så är inte informationen sammanhållet korrekt längre. En inkonsistens har uppstått och den kan ge allvarliga verksamhetsföljder. ACID är alltså i grunden synnerligen önskvärt.

Allför hård ihopkoppling

Det finns ett antal fördelar med att koppla ihop systemdelar lite lösare (så kallad loose coupling, se en tidigare trendspaning). Flexibiliteten ökar och teknisk tillgänglighet kan bli bättre. Detta har det skrivits mycket om inom SOA-litteraturen. Tyvärr måste ACID anses vara en tekniskt sett hård koppling och går alltså dåligt att använda mellan systemdelar. Däremot går det vanligen fint att använda ACID inom en systemdel.

Här kommer ett exempel: Vi ska utforma en sammansatt tjänst som utför dragning från lagersaldot i en underliggande tjänst och addering till kassasaldot i en annan oberoende tjänst. Då har vi problem vad gäller korrekthet, vi får svårt att använda ACID. Om den ena tjänsten störs blir den resulterande informationen inkonsistent:



Kompromissa med färskheten!

En av de viktigaste kompromisserna för att förbättra situationen är att se ifall det är godtagbart för användarna att försämra färskheten hos information. Observera att det här inte ligger inom teknikdomänen utan inom verksamheten. Det är bedömning av nytta som måste ligga bakom en sådant övervägande.

Jag anser dessutom att man borde ta med frågan om informationens inneboende färskhetsbehov varje gång man informationsmodellerar. Då finns det strukturerad kunskap om färskhetskraven.

Fullt klart är att inte allt data behöver vara sekundfärskt. En faktura måste inte komma fram till mottagaren på en sekund. En kvart är lyx, en dag är snabbt, en vecka kanske för långsamt. En sekund är löjligt. Framförallt om ett sekundkrav medför att de andra aspekterna man avväger mot blir avsevärt försämrade, till exempel korrekthet.

Men det är fullt klart att följande aspekter inte går att göra fullständigt perfekta samtidigt, de står mot varandra. Man måste kompromissa medvetet:

- **Flexibilitet:** Att kunna dela upp i små och avgränsade munsbitar inom vilka konsekvenser av förändringar blir mycket lättare att överblicka, att kunna bygga ihop lösningar av sådana byggklossar (tjänster).

- **Skalbarhet:** Om jag ska kunna betjäna, säg, över tjugotusen samtidiga användare så klarar hårt kopplade, stora system med ACID vanligen inte att ge rimligt korta svarstider . Ska jag betjäna användarskarorna hos Facebook eller Google är det omöjligt. Å andra sidan, har applikationen endast hundra användare är det inga problem. De som förordar extrem skalbarhet i det här sammanhanget glömmer ofta bort att en mängd lösningar är små.

- **Tillgänglighet:** Hur får jag lösningen stabil? I mycket stora system med hårda kopplingar och många systemdelar gör komplexiteten att detta blir synnerligen svårt. Man må prata om kluster och avancerade tekniska lösningar för att höja tillgängligheten men problemet är att dessa i sig är

så komplicerade att syftet kan motverkas.

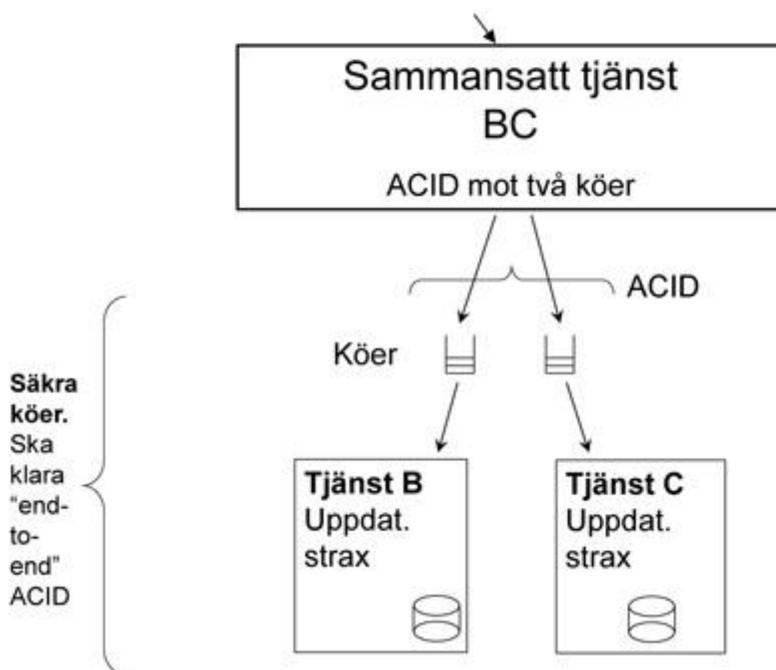
- **Korrekthet:** Får data komma bort eller är det verksamhetskritiskt att informationen är korrekt? Om man verkligen frågar användarna är det ofta otänkbart att data ska få komma bort när det gäller verksamhetskritiska applikationer. Medan däremot, om man räknar statistik och en av tiotusen adderingar till statistiksaldot försvinner så spelar det ingen roll.

- **Färskhet:** Behöver jag verkligen datat på sekunden, duger det om en liten stund eller kanske imorgon?

En av de mest gångbara kompromisslösningarna är att lägga upp en deluppdatering på en säker kö. Den utförs alltså om en stund. *Alla* aspekterna ovan går då att uppfylla, *utom sekundfärskhet*.

Det blir alltså viktigt att användarna ges bra verktyg att förstå vad som händer ifall applikationsarkitekten har valt fördröjd uppdatering. Om man slår upp datat precis efter man tror sig ha lagrat det så finns det inte där. – *"Skumt, bäst att försöka igen"*. Risken finns då att resultatet blir dubbelt. Eller att man avvisar en kund i tron att det inte gick att registrera en order just nu.

Många användargränssnitt är inte alls bra på att upplysa om sådant, användaren har ingen aning om vad hon eller han ska kunna förvänta sig. Felhantering blir också mycket mer komplex att designa än vid vanlig ACID, och därmed dyr.



Mönstret i figuren visar en variant för sammansatta tjänster (composite services), ett mönster som är omhuldat inom SOA-litteraturen. Däremot brukar skribenterna glömma bort frågan om hur en sammanhållen uppdatering måste gå till.

Andra varianter är tänkbara. Den ena tjänsten skulle kunna ta ansvar både för sig själv och för en underliggande tjänst, via kö. Själva anroparen ("ovanför" figuren ovan) kan behöva låta en ACID-uppdatering ske som omfattar en intern databas och en kö mot underliggande tjänst.

Viktigt är också att den kö-lösning som man väljer, verkligen är säker och klarar av successiv vidarebofordring så att inte data i alla fall tappas bort i de på varandra följande stegen. Därför måste man vanligen kräva att kön lagrar på disk och att det finns en lösning för säkerhetskopiering/återställning som kan återskapa ett konsistent läge i tiden.

BASE

Mönstret med uppdatering via kö, asynkront, har funnits i många år. I vissa sammanhang har man kallat det ”deferred update” eller bara asynkron uppdatering.

Ett ganska nytt begrepp är dock BASE: Basically Available - Soft-state - Eventual consistency, som asynkront alternativ till ACID. Detta är förstås samtidigt en ordlek, jämför med syra-bas. Observera också att engelskans ”eventual” betyder så småningom, inte eventuellt. Då skulle hela vitsen försvinna!

BASE myntades av Dan Pritchett (se lästips nedan) och handlar i hög grad om sajter som behöver mycket hög skalbarhet. Han baserar i sin tur artikeln på Eric Brewers CAP-teorem där det beskrivs hur de tre aspekterna Consistency, Availability och Partition tolerance står i motsats och att inte alla tre går att åstadkomma samtidigt. Ett intressant resonemang som går bra ihop med aspekterna jag har listat ovan. Men jag tycker inte att CAP-teoremet riktigt får fram några verksamhetsmässigt sett intressanta konsekvenser: Vilken korrekthet och vilken färskhet åstadkoms?

Som vanligt kan en princip som BASE tillämpas ortodoxt och mer pragmatiskt. Den mer ortodoxa vägen kräver en speciell databasdesign och behövs kanske inte alltid utifrån prestanda- och tillgänglighetsskäl.

Långa verksamhetstransaktioner

Ett sammanfattande begrepp för relaterade saker som följer på varandra i tiden är långa transaktioner (long-running transactions). Jag försöker istället att alltid benämna dem långa verksamhetstransaktioner för att minska risken för missförstånd.

Dels har man ibland programmerat med långa ACID-transaktioner vilket det inte är frågan om här och vilket vanligen inte har fungerat bra alls. Dels är det bra att påpeka att medan ACID, BASE och så vidare handlar om teknik så måste långa verksamhetstransaktioner tas om hand, just det, av verksamheten. Man kan säga att vi har inneboende tekniska ofullkomligheter vars konsekvenser vi tyvärr måste lägga i knät på användarna. Till exempel, vi kan inte uppdatera ena delsystemet förrän om en stund och innan dess kanske vissa manuella rutiner inte kan starta.

En lång verksamhetstransaktion kan pågå i några minuter eller kanske i veckor innan den kan deklareras klar. Under mellantiden bör användarna veta om att detta mellanläge gäller och att resultaten i datasystemen inte är helt slutgiltiga. Som jag skrev ovan så svävar dock ibland användarna i okunnighet om detta. Vissa långa verksamhetstransaktioner är däremot helt självklara och verkligen en del av affärslogiken. Exempelvis avtalsunderskrift - förskott - slutbetalning - leverans (eller för den delen, att köparen ångrar sig).

Långa verksamhetstransaktioner brukar nämnas som standardlösningen på problemet med borttappade uppdateringar då man inte kan ha ACID (eller BASE). Man tänker sig att felet ska upptäckas på något sätt och åtgärdas via någon slags ”kompensering”. Automatiskt eller manuellt av användarna.

Kompensering är gott och väl, men om man designar systemsamverkan på ett naivt sätt och bara struntar i avvägningar kring ACID/BASE vilket man ibland har gjort så **exploderar antalet långa verksamhetstransaktioner**. Att tänka ut logiken i sådana är komplicerat och därmed dyrt och ger felrisker. Att åtgärda dem manuellt kostar tid för användarna vilket är dyrt. Att testa alla varianter i en lång verksamhetstransaktion är komplicerat och därmed dyrt och riskfyllt. Med andra ord, håll ner antalet långa verksamhetstransaktioner!

En liten checklista

Hur ska man nu göra? Några tips följer här.

1. Som nämns ovan bör man hålla ner antalet långa verksamhetstransaktioner. Detta gör man främst genom att modellera tjänster så att de är rimligt stora och har stora anropsgränssnitt (de kallas då grovgranulära). Inuti tjänster kan man nämligen för det mesta använda ACID och då får man inga halva uppdateringar. Man trycker så att säga ner uppdateringen under tjänsteytan.

   Mot detta står en önskan att istället modellera små tjänster för att underlätta återanvändning, flexibilitet, komponerbarhet och utbytbarhet. Man måste alltså kompromissa utgående ifrån aktuella förutsättningar.

2. Man kan också hålla ner antalet långa verksamhetstransaktioner genom att använda BASE (ortodoxt eller pragmatiskt som enklare, köad uppdatering).

3. Skapa god felupptäckt ifall långa verksamhetstransaktioner ska användas. Detta görs via loggar, automatiska larm och genom gammaldags avstämningsrutiner: Skicka en dagssumma, lång checksumma eller liknande via en säker väg till mottagaren och jämför där.

4. När man har mycket kunniga användare kan man ibland presentera direkt i användargränssnittet att ett uppdateringsfel skett och ge råd om hur användaren ska kompensera manuellt.

5. Minska antalet uppdateringsfel som leder till inkonsistens genom att optimera i vilken ordning uppdateringarna görs (den med högst felrisk görs först). Kombineras med omförsök vid fel. Som följd av att omförsök används måste dubletteliminering och annan så kallad idempotency i sin tur tillföras.

6. Ibland kan man tillåta inkonsistens i lagrat data ett tag och först vid ett senare tillfälle avstämma och finna ut att kompensering behövs, ungefär som arbetsprocessen kring bokslut traditionellt är.

7. Under mycket speciella förutsättningar kan man använda ACID över tjänstegränsen, men detta måste övervägas mycket noga eftersom nackdelarna också är stora.

Läs mer

Jag utvecklar resonemanget mer i en [bildserie om transaktioner från SOA Symposium 2010](#).

Min tidigare trendspaning Korrekt data i Molnet.

Trendspaningen om Hur den lösa kopplingen ändå blir hård.

Här myntades begreppet BASE.

“Starbucks Does Not Use Two-Phase Commit”.

Sven-Håkan Olsson är en fristående konsult som särskilt arbetar med att kombinera verksamhetsnytta med teknikhöjd. Han har en lång karriär sedan 70-talet som it-konsult (it-arkitektur, systemdesign, programmering, reviewer, utredningar, kursledning). Sven-Håkan är också medgrundare av Know IT och var dess teknikchef 1990-2003. Han utsågs till en av "Sveriges topputvecklare" av Computer Sweden 2008 och 2010.

Sven-Håkan håller regelbundet kurser åt Dataföreningen Kompetens. Läs gärna mer på hans blogg www.definitivus.se.

Sven-Håkan Olsson