Databasteori och övningar

Fysiska Databasen och Transaktioner

Eva L. Ragnemalm

september 2024, senast uppdaterad april 2025

Boken som refereras till i texten är kursboken, Databasteknik, av T. Padron-McCarthy och T. Risch, Studentlitteratur, 2018.

Innehåll: Sida

Fysiska databasen

Detta område omfattar den fysiska designen av databasen. Detta inkluderar val av datatyper för lagring av relationerna i tabeller, skapande av integritetsvillkor i det valda databashanteringssystemet samt val av organisation av data.

1. Datatyper

Valet av datatyper för det databehov man modellerat i sin relationsmodell handlar om typ och behov av lagringsutrymme. Exempel på datatyper är text, heltal, decimaltal, sanningsvärden och datum. Kortare texter lagras i Char eller Varchar, där man anger en maxlängd, och för Char fylls texten ut med blanktecken om den är kortare än max, medan Varchar i vissa databashanterare faktiskt lagrar bara precis de tecken som faktiskt behövs. Längre texter kan lagras i Text. Heltal (Integer) finns av olika storlek som kan lagra olika stora tal. Decimaltal (Float, decimal) har olika noggrannhet. Boolean lagrar sanningsvärden (true/false). Date är en sammansatt datatyp som lagrar år, månad, och dag.

Det viktiga vid valet av datatyp är att inte välja onödigt stor datatyp, eftersom utrymme reserveras för datatypen även om allt inte behövs, samtidigt måste all data som ska lagras få plats i den valda datatypen. Om man t.ex. försöker lagra ett för stort heltal i en integer förstörs data genom att data som inte får plats trunkeras.

2. Integritetsvillkor

I en relationsdatabas skapas de integritetsvillkor som behövs med hjälp av olika mekanismer. I denna kurs ingår bara de som skapas med SQL. De är primärnycklar (Primary Key), främmande nycklar (Foreign key) samt Not Null (att en cell inte får vara tom).

3. Datalagring

Data som lagras i en tabell i en databashanterare kommer i slutänden att lagras i en fil på en hårddisk, oavsett om det är en roterande hårddisk eller ssd. Detta beror på att vi behöver ett (relativt) permanent sätt att lagra databasen på. Alternativet är att lagra databasen i datorns primärminne, men det är inte permanent (det nollställs då strömmen till datorn stängs av). Data som lagras på en roterande hårddisk eller ssd finns kvar även när strömmen stängs av. Distribuerade databaser ingår inte i denna kurs. Men för att kunna bearbeta data som hämtas från databasen måste detta data flyttas från hårddisken till datorns primärminne (även kallat arbetsminne), som är det ställe där data kan bearbetas. Denna förflyttning tar mätbar tid.

Alla hårddiskar har en minsta mängd data som förs över från den till primärminnet. Denna kallas ett **block**. Ett block är normalt 0,5-4kB (kilobyte) stort och varierar mellan olika hårddiskar. Att föra över ett block från hårddisken tar en viss tid, som kallas **accesstid**. För ssd-er är denna (år 2025) 2-25 μs (μs = mikrosekund, 10^{-6} sekunder) och för roterande hårddiskar 5-10 ms (ms = millisekund, 10^{-3} sekunder).

En tabell som lagras på en hårddisk lagras i en fil, raderna i en tabell lagras i poster i filen. Attributvärdena på en rad i tabellen lagras i fält i posten i filen och tar då upp den plats som den valda datatypen anger. I denna kurs räknar vi inte med poster med variabel storlek utan utgår från att en post i en fil alltid är lika stor oavsett vad som lagras i den.

Hur lång tid det tar att hämta data från hårddisken beror på hårddisken, hur stor filen är och hur den är organiserad.

Beräkning av plats/utrymme

För att beräkna hur stor plats en fil tar på en hårddisk behöver vi veta hur många poster filen har och hur stor varje post är. Vi behöver också veta hur stora hårddiskens block är. Vi lagrar aldrig delar av poster i block eftersom det skulle kräva att vi *ibland* behövde läsa in ytterligare ett block från hårddisken, och det är för opraktiskt.

Blockstorleken anges med **B** i uträkningarna nedan..

Filens **antal poste**r är alltid samma som antalet rader i tabellen och benämns **r** i uträkningarna.

Filens **poststorlek** är samma sak som det utrymme som krävs för att lagra en rad i tabellen som lagras i filen. Den beräknas som summan av det utrymme som behövs för de olika fälten i posten (dvs attributvärdena på en rad i tabellen), och detta bestäms av attributens datatyper. Olika datatyper kräver olika mycket utrymme och beräknas i byte. Poststorlek benämns **R** i uträkningarna.

För att beräkna hur många block en fil tar upp måste vi först beräkna hur många poster som får plats i varje block på hårddisken. Detta kallas **blockningsfaktorn, bfr**, och beror av poststorlek och hårddiskens blockstorlek:

bfr = $\lfloor$**B/R**$\rfloor$ - utläses: blockfaktorn är blockstorleken dividerad med poststorleken, avrundat nedåt till närmsta heltal eftersom vi inte lagrar en bit av en post i ett block.

Antal block som krävs för att lagra filen (**b**) beräknas sedan genom:

b = $\lceil$**r/bfr**$\rceil$- utläses: antal block är antalet poster i filen dividerat med blockfaktorn, avrundat uppåt till närmsta heltal. Vi räknar i hela block och även om det bara är en enda post i sista blocket ingår blocket i de som behövs för filen.

Räkneexempel: En hårddisk har en blockstorlek (B) på 2048 byte. Vi lagrar en fil med 30 000 poster (r) på den, posternas storlek (R) är 100 byte styck.

bfr = $\lfloor 2048/100 \rfloor = \lfloor 20{,}48 \rfloor = 20$ poster/block

b = $\lceil 30\,000/20 \rceil = \lceil 3000/2 \rceil = 1500$ block.

Filen tar alltså upp 1500 block på hårddisken.

Vilka block som ingår i en fil lagras i något som kallas **filhuvud**. Informationen har formen av adresser till block som ingår i filen. Dessa adresser kallas också pekare (till block). Alla filhuvuden lagras på en speciell plats på en hårddisk. Exakt hur filhuvudet ser ut beror på hur filen är organiserad.

4. Beräkning av tid för dataöverföring

Egentligen är både insättning, borttagning, ändring och sökning av data relevanta när vi beräknar hur lång tid saker tar, men då insättning, ändring och borttagning nästan alltid innebär sökning är det tiden för sökning som är central att räkna på. Hur lång tid det tar att hitta en specifik post beror på hur många block vi måste ladda in till primärminnet för att titta i innan vi hittar den vi sökte. Detta beror i sin tur på hur lång accesstid hårddisken har och hur filen vi lagrar tabellen i är organiserad. I kursen ingår fyra olika organisationer: Hög (osorterad), Sorterad, Hashtabell och Indexerad. Filorganisationen påverkar också vad som lagras i filhuvudet och hur stort det blir. Detta räknar vi normalt inte in i filens storlek, då det är en mycket mindre datamängd.

Notera att eftersom miniräknare inte är ett tillåtet hjälpmedel på tentan väljs siffor i beräkningarna så att bara enklare beräkningar ska behöva göras (se nedan och övningsexemplen för nivån), och för mer avancerade beräkningar tillhandahålls s.k. "Mattetips".

Hög

När man söker en viss post i en osorterad fil får man *i genomsnitt* leta igenom halva filen. I värsta fall får man leta igenom hela (specialfall: om vi letar något som inte finns eller vill vara säkra på att vi hittat alla poster av en viss sort, så måste vi *alltid* söka igenom hela filen). Eftersom filen lagras i ett antal block kommer man (i genomsnitt) att göra
⌈**b/2**⌉ blockaccesser som tar en viss tid per styck (accesstiden). Vi avrundar uppåt för det går inte att göra mindre än en blockaccess (antingen läser vi in ett block eller så gör vi inte det, vi kan inte läsa mindre än ett helt block pga definitionen av block).

Räkneexempel: Antag att hårddisken har en accesstid på 10ms (dvs $10*10^{-3}$s = 0,01s) och vi söker i den tidigare filen som har b = 1500 block:

Söktiden blir = ⌈b/2⌉st blockaccesser, dvs ⌈1500/2⌉= 750 accesser eller uttryckt i sekunder: 750 *0,01 = 7,5 sekunder.

I värsta fall, när vi måste söka igenom hela filen, blir söktiden b accesser, dvs b* accesstiden sekunder: 1500 * 0,01 = 15 sekunder.

Datalagring för hög

En hög kräver inget extra utrymme utan storleken beräknas enligt "Beräkning av plats/utrymme" ovan. Filhuvudet för en hög har normalt pekaren (adressen) till första blocket och sista blocket. I varje block lagras också en pekare till nästa block i filen. Utrymmet den pekaren tar är redan borträknat när vi anger hårddiskens blockstorlek. Filhuvudets storlek räknas inte med när vi beräknar filens storlek.

Sorterad

Om filen sorteras på något fält (dvs attribut) kommer det att gå fortare att söka på detta attribut. Detta gäller även om vi har ett sammansatt sökattribut, dvs t.ex. efternamn och förnamn så att filen sorteras på efternamn och för icke-unika efternamn sorteras posterna sedan på förnamn. Sökning på något annat än det attribut som filen är sorterad på blir då som att söka i en hög.

För att söka i en sorterad fil använder vi binärsökning, som fungerar så här: läs in och titta på mittersta blocket i filen, är det värde vi söker högre eller lägre än värdet (värdena) som finns där? Har vi tur finns värdet vi sökte i det inlästa blocket och då är vi klara. Om det vi söker har lägre värde än de som finns i blocket, läs in det block som ligger mitt i intervallet mellan början och det mittersta block vi just tittade på. Om högre, titta på det block som ligger mitt i intervallet mellan det block vi just tittade på och slutet av filen. Detta upprepas (hela tiden halveras det intervall man tittar på) tills man hittar rätt block (jämför leken "Hi-Lo"). Matematiskt blir detta att vi får läsa in lika många block som tvålogaritmen (logaritmen med bas 2) av antal block i filen, avrundat uppåt (vi läser alltid in hela block).

Söktiden blir:$\lceil \log_2(b) \rceil$

Räkneexempel: Samma fil och hårddisk som tidigare ger b = 1500 block.

Söktiden blir$\lceil \log_2(1500) \rceil$= ⌈10,55075⌉= 11 accesser, eller i sekunder 11* 0,01 = 0,11 sekunder.

Datalagring för sorterad

En sorterad fil kräver mycket lite extra utrymme jämfört med osorterad fil, bara filhuvudet blir lite större. Filhuvudet för en sorterad fil innehåller normalt en pekare till varje block i filen, i den ordning de ligger i filen. Detta utrymmer räknas normalt inte med när man beräknar filens storlek.

Hash

En hashstruktur består av ett antal block på en hårddisk, där varje block lagrar poster med ett visst värde på hashfunktionen. Hashfunktionens värde beräknas från primärnyckeln till tabellen och

och det värdet används som index till hashstrukturen (hashvärdet är alltså ett positions-nummer från 0 för första blocket, till N där N är antal block i hashstrukturen). En perfekt hashfunktion sprider posterna över hashstrukturen så att varje block fylls med poster och alla får plats, men det är mycket svårt att hitta sådana hashfunktioner eftersom de måste fungera på tabellens primärnyckel. Normalt får man tomrum i vissa block och ett antal poster som inte får plats i det block där de borde lagras, Dessa lagras då i s.k. *spillblock*, som länkas in så att de nås från det rätta blocket, antingen så att man utnyttjar block i strukturen som är tomma eller tillför nya. En hashstruktur har alltså en viss andel tomrum, "luft", helt eller delvis tomma block. Vid utformandet av hashstruktur och hashfunktion brukar man räkna med en viss procent "luft".

Vid sökning i hashstruktur beräknas hashvärdet för den sökta posten från dess nyckel, man får direkt adressen till det rätta blocket och kan läsa in det. Om posten inte kunnat lagras i det rätta blocket behöver man läsa in ett spillblock, eller om hashfunktionen är dålig, ytterligare spillblock. Sökning i hashstruktur på nyckeln tar alltså alltid ett fåtal accesser, normalt 1 men ibland 2-3 st. Vid sökning på annat attribut än nyckeln blir det som att söka i en hög.

Räkneexempel: Samma fil och hårddisk som ovan. Sökning på nyckeln tar 1-3 accesser, dvs 0,01-0,03 sekunder.

Datalagring för hashstruktur

Organiserad som hög tar filen upp 1500 block och en hashstruktur utan extra "luft" skulle ta lika stor plats. Eftersom hashfunktionen normalt inte är perfekt lägger vi på ett antal procent luft, dvs man reserverar lagringsutrymme för lite mer än de blocken. Exempelvis kan man reservera 10% extra utrymme, filen skulle då ta upp 1500 + 150 = 1650 block.

Index

Ett index för en fil (där en viss tabell lagras) är som ett index eller en innehållsförteckning i en bok: en lista av ord/kapitel som finns i boken med en pekare till var i boken (vilken sida) som det ordet/kapitlet finns. Listan är sorterad i bokstavsordning för att underlätta sökning. Ett index i databasen består av ett (eller flera) attribut som finns i tabellen samt adressen till ett block (pekaren). Varje post (kan ses som rad) i indexet innehåller sedan ett attributvärde och adressen till det block där posten med det attributvärdet lagras.

Det finns olika typer av index: primärindex, sekundärindex, klustrat index (kallas också grupperat index), som beror på hurdant attributet är i tabellen. Ett index kan också vara glest eller tätt, vilket har att göra med ifall indexet innehåller en post per post i datafilen eller färre, t.ex. en post per block. Om indexet innehåller en post per post i datafilen är det ett tätt index. Om indexet innehåller en post per block (givet att varje block innehåller mer än en post) är det ett glest index.

Ett **primärindex** är ett index där attributet man skapar indexet på (som därmed också lagras i indexfilen) är primärnyckel i tabellen som lagrars i datafilen och bygger på att datafilen är sorterad på primärnyckeln. Då behöver man bara en post i indexfilen per block i datafilen (det blir ett glest index). Om nyckeln till tabellen är sammansatt av flera attribut lagras alla dessa i indexet.

Ett **sekundärindex** är ett index där filen är *inte* är sorterad på det attribut som indexet skapas på, men attributet är unikt. Då behöver man en indexpost per post i datafilen, och flera indexposter kan referera till samma block (då de två posterna råkar ligga i samma block). Sekundärindex är ett tätt index.

Ett **klusterindex** är ett index på ett attribut som inte är unikt i datafilen, men som datafilen är sorterad på. Här behövs en indexpost per unikt värde på attributet och den pekare som lagras i posten är pekaren till det första blocket som innehåller det värdet (men det kan finnas fler, då ligger de som nästa block i filen). Klusterindex är en typ av glest index.

Det finns också **multipelindex**, som helt enkelt är ett index på en indexfil, dvs man kan ha flera nivåer av index.

Datalagring för index

Ett index är i sig en sorterad fil som tar upp plats i databasen, för indexfilen lagras också i databasen. Den har tekniskt sett också ett filhuvud som tar plats, men det brukar man bortse från då det utrymmet är försumbart.

Det utrymme en indexfil tar beräknas på samma sätt som vilken fil som helst utifrån poststorlek och antal poster. Storleken på en indexpost fås som summan av det utrymme attributet (eller om det är flera, alla attributen) tar plus storleken av en pekare (adress) för den aktuella hårddisken. Hur många poster ett index har beror på typen av index (se ovan).

Hur mycket utrymme indexfilen tar beräknas sedan på samma sätt som för en sorterad fil (se ovan). När man sedan beräknar hur mycket datautrymme helheten tar får man inte glömma att lägga till själva datafilen.

Räkneexempel: Om det är ett **primärindex**: Samma fil och hårddisk som ovan. Antag att primärnyckeln tar 40 byte och adressen till ett block (pekaren) 10 byte. En indexpost (iR) tar då 50 byte. Eftersom datafilen tar upp 1500 block får vi 1500 poster i indexfilen (ir). Blockningsfaktorn för indexfilen (ibfr) blir:

$ibfr = \lfloor 2048/50 \rfloor = \lfloor 40{,}96 \rfloor = 40$ poster/block

$ib = \lceil 1500/40 \rceil = \lceil 150/4 \rceil = 38$ block.

När man ska söka efter en viss post i en indexerad fil söker man först efter rätt post i indexet på samma sätt som vid sökning i sorterad fil (se ovan). Sedan har man i indexposten, pekaren till rätt block i huvudfilen, varför sökningen beräknas som sökning i sorterad fil på indexfilen plus en blockaccess.

Sökning i ovannämnda index blir då: $\lceil \log_2(ib) \rceil = \lceil \log_2(38) \rceil = \lceil 5{,}2479 \rceil = 6$ accesser, plus en access för att hitta posten i datafilen, dvs 7 accesser. I tid $7 * 0{,}01 = 0{,}07$s.

Ett **sekundärindex** för ovanstående fil och hårddisk skulle få lika många poster som datafilen, dvs 30 000st. Om vi antar att poststorleken för denna indexfil är densamma som för primärindexet får vi samma blockningsfaktor, 40 poster per block. Antalet block för indexfilen blir då:

$ib = \lceil 30000/40 \rceil = 750$ block.

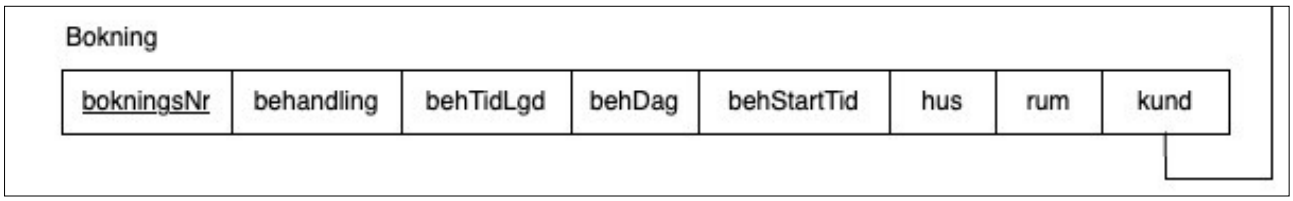
Sökning i dessa 750 block tar då $\lceil \log_2(ib) \rceil = \lceil \log_2(750) \rceil = \lceil 9{,}5507 \rceil = 10$ accesser, plus en access för att hitta posten i datafilen, dvs 11 accesser. I tid $11 * 0{,}01 = 0{,}11$s.

Ett **klusterindex**, för ovanstående fil får så många poster som det finns unika värden på index-attributet. Antag att det finns 10 000 unika värden och att indexfilens poststorlek fortfarande är 50 byte (vilket ger samma ibfr som tidigare) så tar denna indexfil:

$ib = \lceil 10000/40 \rceil = 2500$ block. Sökning i detta beräknas som ovan.

Övningar Fysiska databasen

1. Spa-företaget, lagringsutrymme



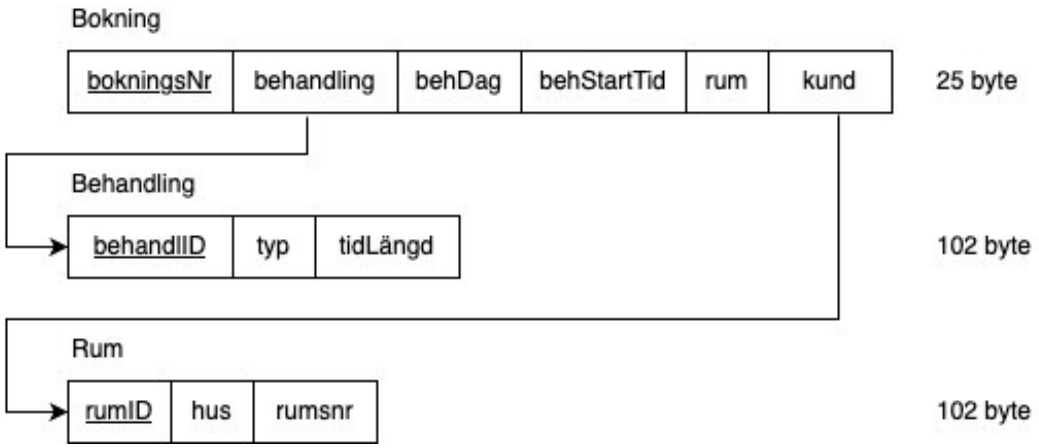
Figur 1: Tabell för ett Spa-företag.

Tabellen Bokning så som den ser ut i figur 1 skulle kunna implementeras så att en post tar 200 byte. Vi kan lagra den på en hårddisk med blockstorleken 1024 byte.

a. Beräkna hur många block filen Bokning tar upp om det finns 1000 bokningar i den.

b. Om vi vet att attributet behTidLgd (behandlingens längd i tid) är fullt funktionellt beroende av attributet behandling (behandlingens namn) kan vi normalisera Bokning och då bryta ut behandling och behTidLgd till en egen tabell. Om vi dessutom inför ett numeriskt ID för varje behandling i den nya tabellen kan vi istället för att lagra behandlingens namn i Bokning lagra detta ID. Om vi dessutom har fallet att ett unikt rum behöver både husets namn och rummets nummer eller namn (vad som nu lagras i attributet rum i figur 1) för att identifieras kan vi på liknande sätt bryta ut hus och rum, göra en egen tabell där varje unikt rum får ett nytt numeriskt ID. Nu har vi delat upp relationen Bokning i tre tabeller, som i figur 2 nedan, där vi alltså ökar databehovet med ett ID för vardera behandling och rum. Vi skulle då kunna få de poststorlekar som visas i figuren. Givet att vi fortfarande har 1000 rader i Bokning, men bara 50 rader i Behandling och 20 rader i Rum, hur många block tar de olika filerna upp då, och hur många blir det tillsammans?

Figur 2. Bokning efter normalisering och ändring av rum.



2. Primärindex

Du har en fil med 10 000 poster som vardera tar 200 Byte, varav primärnyckeln tar 15, som är sorterad enligt primärnyckeln. Filen lagras på en hårddisk med blockstorleken 1024 byte och adressen till varje block tar 10 byte. Du skapar ett primärindex. Besvara nedanstående frågor. Tips:

redovisa alla steg i dina beräkningar så får du kanske poäng för delar även om inte slutsvaret är rätt.

a) Hur stora blir indexfilens poster och hur många poster har den?

b) Hur många blockaccesser tar det att söka en post med hjälp av primärnyckeln och indexet?

Mattetips: $Log_2(2000)=10.97$; $Log_2(1024)=10$; $Log_2(1000)=9.97$; $Log_2(200)$; $Log_2(50)=5{,}64$; $Log_2(32)=5$;

3. Hashstruktur

När man organiserar data i en tabell med hjälp av Hashning behöver man en hashfunktion som genererar hashvärden. Förklara varför Hashfunktionen "det tal som bildas av de första sex siffrorna i personnumret" (alltså personnumret 991210-1234 ger hashvärdet 991210), inte är en bra hashfunktion för Ladokregistret på LiU (som alltså är stort och innehåller alla möjliga personnummer, och har personnumret som primärnyckel).

4. Accesstid

Du har en tabell över alla studenter på LiU. Den innehåller fälten: Personnummer, Efternamn, Förnamn, student-id, Adress, Postadress, telefonnr1, telefonnr2. Posterna tar 500 byte, filen har 20000 poster och är sorterad på personnummer. Hårddisken som används har blockstorlek 2048 bytes och accesstid 10 ms. Besvara nedanstående frågor:

a) Hur lång tid tar det att söka ut en viss person givet personnumret?

b) Hur lång tid tar det att söka ut alla personer som har ett visst namn?

c) Om vi skapar ett primärindex, där vi antar att indexposten blir 50 byte hur lång tid tar det då att söka ut en viss person givet personnumret? Och givet namnet?

d) Om vi istället sorterar filen på namn (efternamn, förnamn) så kan vi inte använda ett primärindex för att söka på personnummer, det blir ett sekundärindex (men indexposten blir lika stor som i föregående deluppgift, 50 byte). Hur lång tid tar det att söka en viss person givet personnumret med det indexet?

e) Om filen är sorterad på namn (som i ovanstående deluppgift), hur lång tid tar det att söka ut alla personer som har ett visst namn?

f) Om vi dessutom skapar ett klusterindex för namn (efternamn, förnamn) till filen som är sorterad på namn, och det finns 15000 unika namn, hur lång tid tar det då att söka ut alla personer som har ett visst namn? Anta att indexposten för detta index blir 100 byte.

Mattetips: $Log_2(125) = 3.32$, $log_2(500)=8.97$, $log_2(750)=9.55$, $log_2(2000)=10.97$, $log_2(2048)=11$, $log_2(5000)=12.29$, $log_2(20000)=14.29$

Transaktioner

En transaktion är en logiskt sammanhängande sekvens av interaktioner mot en databas, där en interaktion definieras som att data läses eller skrivs till databasen. Exempelvis är överföringen av pengar mellan två olika konton på banken en transaktion. De interaktioner som ska hänga samman är då läsningen av saldo från det ena kontot, skrivningen av det nya saldot på det kontot, läsningen av saldot på det andra kontot och skrivningen av det nya saldot på det kontot. Emellan dessa operationer sker beräkningar och annan bearbetning, men det är just läsningen och skrivningen av objekt i databasen som är viktiga.

En dator fungerar så att den bara kör ett program i taget, även om vi uppfattar det som att den gör flera saker samtidigt eftersom den växlar mellan dem så fort. Program i detta sammanhang kan vara både olika delar av operativsystemet, t.ex. tangentbordshanteraren och bildskärmshanteraren, men också själva ordbehandlingsprogrammet som anropar tangentbordshanteraren för att läsa in de tecken som skrivs, samtidigt som ljuduppspelnings programmet spelar musik i hörlurarna med hjälp av bluetooth-programmet. Varje program som körs på datorn får en viss tid att utföras, och sedan får nästa program köra en stund och efter en stund kommer man runt till det första programmet igen och det får köra vidare. Dessa avbrott kommer på slumpmässiga ställen i programmen vilket gör att vi aldrig kan garantera att ett visst bit av programmet (antal kommandon) kommer att utföras utan avbrott. Dessa avbrott är dock inte märkbara för oss användare eftersom växlingen sker så snabbt. Det betyder till exempel att vi uppfattar att två användare av en databas gör något samtidigt, när de i praktiken delas upp så att det de gör händer i sekvens, men vi kan inte förutsäga hur sekvensen kommer att se ut.

Detta kan göra att vi får problem ifall datorn avbryts på ett okontrollerat sätt, t.ex. vid en hängning eller om den plötsligt stängs av. Det är ju lite trist om en överföring av pengar från mitt konto till en mottagare stoppar efter att pengarna dragits från mitt konto men innan de lagts in på mottagarens.

ACID

En transaktion ska vara ACID, Atomic (odelbar), Consistent (konsistent, korrekthetsbevarande, inte införa motsägelser i data), Isolated (inte påverkas av andra transaktioner även om de körs mer eller mindre samtidigt), Durable (inte kunna försvinna när de väl genomförts). Se kursboken kapitel 24.

Databashanteringssystemet har ett antal mekanismer för att implementera dessa egenskaper.

Atomic (Atomär, odelbar): För att hålla transaktioner odelbara trots att ett program kan avbrytas när som helst används markeringarna **start**/**commit** som skrivs i en **loggfil** samt funktionen **rollback**. Detta fungerar så att en transaktion skriver "start" i loggfilen när den startar, loggar (skriver i loggfilen) de interaktioner mot databasen som görs, och när den är klar markeras det med "commit" i loggfilen. Om transaktionen avbryts av någon anledning innan den är klar måste de ändringar som gjorts i databasen (skrivningar) ändras tillbaka till de gamla värdena (rollback).

Consistent (korrekt): Under antagandet att databasen var korrekt (utan inre motsägelser) vid transaktionens start ska transaktionen också lämna databasen i korrekt skick. Detta implementeras genom att databasens integritetsvillkor kontrolleras vid varje ändring av databasen.

Isolated: (isolerad) Transaktioner hindras från att påverka varandra med hjälp av **lås** (binära eller läs/skrivlås). Lås innebär att ett dataobjekt (tabell, rad eller cell) reserveras så att bara den transaktion som har låset får använda objektet. Vid användning av lås måste grundprotokoll alltid följas men det finns fler, mer restriktiva protokoll som förhindrar mer avancerade typer av problem. Ett **protokoll** är en uppsättning regler för hur och i vilka situationer kommandona som hanterar låsen ska användas (se nedan).

Durable (bestående): Databasens innehåll bevaras genom att man tar **backup** regelbundet och också loggar genomförda transaktioner i en loggfil. Vid den typ av krascher där lagringsmediet som databasen finns på förstörs läser man först tillbaka data från backupen, och sedan behöver man gå igenom loggfilen för att repetera alla ändringar som genomförts efter att backupen togs. I denna situation repeteras alla databasändringar som utförts av transaktioner som slutförts (skrivit commit i loggfilen). Transaktioner som ej är slutförda ska ändras tillbaka (rollback), dvs ändringar i databasen återställs till ursprungsvärde. Vid en krasch som bara bryter pågående aktiviteter men ej förstör datalagringen (hängningar etc) behöver ingen backup läsas tillbaka och i loggfilen går man bara igenom transaktionerna efter senaste checkpoint, eftersom transaktioner innan checkpoint är avslutade och utskrivna till databasen. På samma sätt som tidigare ska avslutade transaktioner (efter checkpoint) repeteras och oavslutade ändras tillbaka. Det är viktigt att loggfilen inte lagras på samma hårddisk som själva databasen.

Lås

Problem med isoleringen av transaktioner uppstår eftersom vi vill att flera användare ska kunna komma åt databasen mer eller mindre samtidigt. När två användarprogram läser och skriver i samma databasobjekt kan vi få de problem som beskrivs i kapitel 5.11.

För att implementera isoleringen av transaktioner används lås. I det följande antar vi att ett användarprogram startar en transaktion och det är den transaktionen som diskuteras.

Ett lås innebär att en transaktion reserverar åtkomsten till ett databasobjekt för egen del. När ett databasobjekt är låst kan bara den transaktion som har låst objektet komma åt det (läsa från eller skriva till det). En transaktion låser ett databasobjekt med ett kommando, oftast "Lock(objekt)" (eller på svenska "Lås(objekt)"). För att släppa objektet, dvs göra det åtkomligt för andra transaktioner igen, används kommandot "Unlock(objekt)", på svenska "LåsUpp(objekt)". En transaktion som ber om att få låsa ett databasobjekt som redan är låst av en annan transaktion kommer att pausas i väntan på att objektet frigörs (låset släpps). Ett databasobjekt i detta sammanhang kan vara en hel tabell, en rad i en tabell eller en cell. Eftersom vi ändå eftersträvar att flera användare ska kunna arbeta med databasen samtidigt vill vi inte låsa onödigt stora delar av databasen och inte onödigt länge.

Typer av lås: Det finns **binära** lås, där ett objekt antingen kan vara låst eller inte låst, men många databashanteringssystem har också möjligheten att skilja på när en transaktion bara vill läsa ett objekt från när transaktionen också vill skriva i ett objekt. Att flera transaktioner läser ett objekt "samtidigt" ställer inte till några problem, utan det är skrivning (att ändra värdet på ett objekt) som ställer till problem. Med "samtidigt" menas här att flera transaktioner är startade men inte avslutade. Binära lås hanteras som tidigare nämnts med "lås/låsUpp", medan **läs- och skrivlås** har operationerna "ReadLock" (läsLås), "WriteLock" (skrivLås) och "Unlock" (låsUpp). Om en transaktion har läsLåst ett databasobjekt kan andra transaktioner få läsLås på samma databasobjekt, men inte skrivLås. Om en transaktion har skrivLåst ett objekt kan ingen annan transaktion får något annat lås på objektet.

Protokoll: Användningen av lås måste följa vissa regler, som sammanfattas under begreppet protokoll. Binära lås har ett grundprotokoll som innehåller följande regler: Innan man läser eller skriver till ett databasobjekt måste det objektet låsas (Lock(X) utföras). När man är färdig med ett databasobjekt som man har låst ska låset släppas (Unlock(X) utföras). Man kan inte låsaUpp (Unlock) ett databasobjekt man inte har låst, och man får inte försöka låsa ett databasobjekt som man redan har låst.

Läs- och Skrivlås har ett grundprotokoll som innehåller följande regler: Innan man skriver till ett databasobjekt måste det låsas för skrivning (WriteLock(X) utföras). Innan man läser från ett databasobjekt måste det låsas antingen för läsning (ReadLock(X) utföras) eller för skrivning (WriteLock(X) utföras). När man är klar med objektet ska låset släppas (UnLock (X) utföras). Man

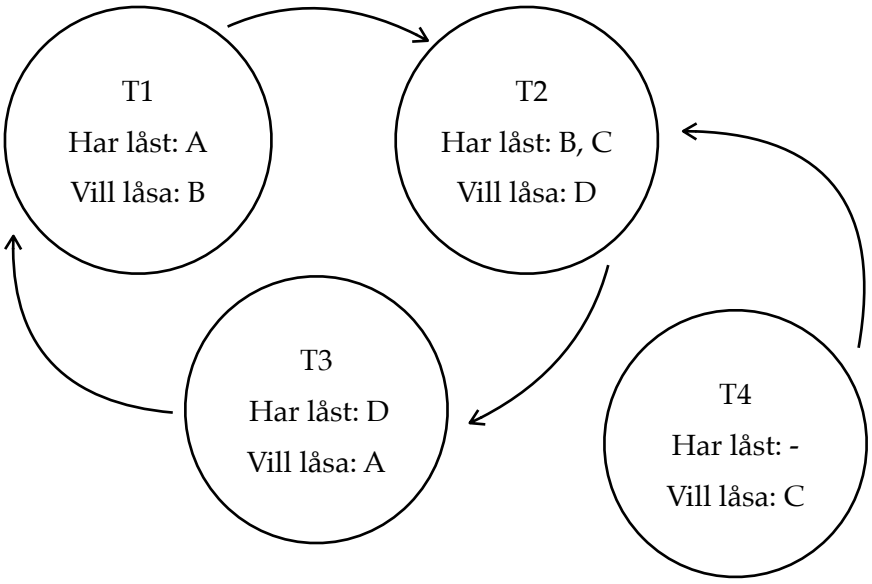
kan inte låsa upp ett databasobjekt man inte har låst och man får inte försöka göra läsLås på ett objekt man redan har läsLås på, och inte skrivLås på ett objekt man redan har skrivLås på. MEN man får göra skrivLås på ett objekt man har läsLås på (det kallas att gradera upp låset).

Grundprotokollen för att hantera lås löser dock inte alla problem (se kap 25.19.3), utan man kan vara tvungen att använda protokollet **Tvåfaslåsning** (låsen hanteras i två faser, en låsningsfas och en upplåsningsfas). Detta innebär att man inte får låsa upp något databasobjekt innan alla objekt som behöver låsas har blivit låsta. I kap 25.19.4 illustreras ett problem med tvåfaslåsning som löses genom **Rigorös tvåfaslåsning**, som innebär att man inte släpper några lås förrän transaktionen är helt klar, har gjort Commit (eller har avbrutits). När man använder läs-och-skrivlås kan man använda protokollet **Strikt tvåfaslåsning** som tillåter att läslås släpps i upplåsningsfasen, så fort de inte behövs, medan skrivlås behålls tills transaktionen är klar (eller avbruten).

Notera att vi ändå strävar efter att låsa objekten så kort tid som möjligt, eftersom vi inte vill blockera andra transaktioner i onödan.

Problem med lås: Deadlock

När databashanteraren använder lås kan man hamna i situationen som beskrivs i 25.19.5, att två transaktioner (eller fler) väntar på varandra och kan inte köra färdigt innan de fått låsa ett databasobjekt som den andra transaktion har låst. Detta kallas Deadlock. Det finns flera sätt att förhindra deadlock t.ex. att alla transaktioner alltid låser objekten i en viss ordning (oavsett vilken ordning de ska användas i), eller att en transaktion alltid låser alla databasobjekt den behöver på en och samma gång (och om den inte får alla låsen släpper den de lås den fått och försöker på nytt senare), detta kallas **Konservativ tvåfaslåsning,**. Ofta är det svårt att förutsäga vilka databasobjekt en transaktion kommer att behöva och det är därför vanligare med upptäckande strategier: **Timeout** innebär att om en transaktion väntat en viss tid på ett lås utan att få det antas att deadlock uppstått och transaktionen avbryts (då släpps eventuella lås transaktionen hade och transaktionen rullas tillbaka). Här kan man råka ut för att en transaktion avbryts i onödan, och för att undvika det kan man istället **undersöka "Wait-for-grafen**". Man skapar en graf av alla transaktioner som väntar på databasobjekt, och vilka transaktioner de väntar på, och undersöker om det finns cirklar i denna graf, se figuren nedan:



Figur 3. Wait-for-graf. Pil betyder att transaktionen som pilen startar i väntar på ett objekt som är låst av transaktionen som pilen pekar på.

I figur 3 kan ses att transaktionerna T1, T2 och T3 har hamnat i deadlock (det finns pilar som beskriver en cirkel mellan dem), medan T4 visserligen inte kan köra klart innan T2 är klar, men inte egentligen är involverad i deadlocken (ingen annan beror av T4).

Övningar Transaktioner

1. Strikt tvåfaslåsning

Transaktioner: Antag att du har transaktionen T1 nedan. Placera in kommandona ReadLock/LäsLås, WriteLock/SkrivLås och Unlock/LåsUpp i transaktionen enligt protokollet för Strikt tvåfaslåsning. Sträva efter att låsa varje objekt så kort tid som möjligt och ändå följa protokollet. (OBS: skriv ut alla nedanstående operationerna i ditt svar så att jag ser hela sammanhanget). Var noga med att visa vilket objekt som låses och om du använder svenska se till att jag ser skillnad på Lås och Läs).

```
Start (T1)
Read(A)
Read(B)
B = B - A
Write (B)
Read(C)
C = C - B
If C<0 then Rollback (T1)
Write=(C)
Commit (T1)
```

2. Problem med isolering

Transaktioner: Givet transaktionerna T1 och T2 nedan, ge två exempel (i form av sekvenser av operationer ur transaktionerna - tänk på hur du illustrerar ordningen så att det blir tydligt) på problem som kan uppstå om inte någon form av låsning av används. Beskriv också i ord hur det inträffade skiljer sig från det önskvärda utfallet.

T1

```
Read(X)
X=X+1
Write(X)
Read(Y)
Y=Y-1
Write(Y)
```

T2

```
Read(X)
X=X-5
Write(X)
```

3. Solåsen - problem med isolering

(Detta är utdrag ur en längre kombinationsupptift om Sjukhemmet Solåsen, där man lagrar data om patienter, provsvar och läkare).

Den inkompetente programmeraren skrev sedan några program som hanterade två filer, en med patientdata och en med data om läkare. Filerna lagrades på en server centralt så att alla skulle kunna komma åt dem från sina kontorsdatorer när de ville. Programmet som lade in ett nytt prov fungerade så att det först öppnade Patientfilen som låg på servern, läste in den till en intern fil, stängde den externa (den på servern), uppdaterade patientposten, skrev tillbaks hela den ändrade interna filen till servern (skriver över den gamla filen).

Programmet som lade in en ny patient fungerade så att det först öppnade Patientfilen, läste in den till en intern fil, stängde den externa (den på servern), lade in patientposten, öppnade Läkarfilen (på samma sätt som patientfilen), lade in en ny patient på den aktuella läkaren, skrev tillbaks båda de interna filerna till servern (skriver över de båda gamla).

Efter ett tag upptäckte man att slumpvisa (såvitt man kunde se) ändringar försvann fastän läkaren som ändrat svor på att han/hon kört programmet korrekt. Man anställde då en extra sekreterare som utbildades speciellt på programmet och som sedan var den ende som införde ändringar. Alla uppdateringar i systemet meddelades till denna person, som lade in dem, istället för att varje läkare lade in dem själv. Då försvann problemet.

Vad orsakade problemet (förklara i detalj vad som hände när det blev fel) och vad innehåller ett DBMS som den klantige programmeraren glömt lägga in i sitt program?

4. Problem med lås

(Detta är en del av en kombinationsuppgift med flera tabeller men det spelar ingen roll för frågan, det räcker med en tabell).

Antag att det är flera användare som arbetar mot de ovannämnda tabellerna. Man lägger därför in låshantering för att förhindra att uppdateringar skrivs över. Man följer tvåfas-protokollet för alla transaktioner. Dock får man problem ibland, som yttrar sig genom att databasen verkar hänga sig, två eller tre transaktioner bara stannar och kan inte slutföras ens efter lång väntan. Det som konfunderar användarna är att även om några användare inte kan slutföra sina transaktioner så kan andra använda databasen samtidigt. Vilket fenomen som vi behandlat i kursen kan yttra sig på detta sätt och hur kan man undvika det?

Facit

Tips: du lär bättre om du faktiskt arbetar igenom uppgifterna innan du tittar på facit.

Fysiska databasen:

1. Spaföretaget

a) (beräkna hur många block filen tar upp): Poststorlek R = 200byte, Blockstorlek B = 1024byte, antal poster r = 1000st
bfr = ⌊1024/200⌋ = ⌊5.12⌋= 5 poster/block
b = ⌈1000/5⌉ = 200
Svar: filen tar upp 200 block.

b) (beräkna hur många block de tre nya filerna tar upp)
Börja med den nya Bokning: Poststorlek R = 25 byte, Blockstorlek B = 1024byte, antal poster r = 1000st
bfr = ⌊1024/25⌋ = ⌊40.?⌋= 40 poster/block (tänk att varje post i den gamla filen tar samma plats som 8 st, då blir det 5*8 poster i blocket (kontroll: 40*25=1000 och sedan går det inte i en till).
b = ⌈1000/40⌉ = 25 block
Filen för Behandling har: Poststorlek R = 102 byte, Blockstorlek B = 1024 byte, antal poster r = 50 st
bfr = ⌊1024/102⌋ = ⌊10.24⌋= 10 poster/block.
b = ⌈50/10⌉ = 5 block
Tabellen Rum har Poststorlek R = 102 byte, Blockstorlek B = 1024 byte, antal poster r = 20 st
bfr = ⌊1024/102⌋ = ⌊10.24⌋= 10 poster/block.
b = ⌈20/10⌉ = 2 block
Svar: De tre nya filerna tar tillsammans upp 25 + 5 + 2 = 32 block.

2. Primärindex

a) Ett primärindex: varje indexpost ska bestå av primärnyckeln (25 byte) och adressen till ett block på hårddisken (10 byte). Indexposten blir 15+10=25 byte (iR). Hur många poster har indexfilen? Ett primärindex får lika många poster som datafilen (huvudfilen) har block. Hur många block tar vår datafil upp?
Datafilen: Poststorlek R = 200 byte, Blockstorlek B = 1024 byte, antal poster r = 10 000 st
bfr = ⌊1024/200⌋ = ⌊5.12⌋= 5 poster/block
b = ⌈10000/5⌉ = 2000 block.
Indexfilen får alltså 2000 poster (ir = 2000)
Svar: indexfilens post blir 25 byte och indexfilen har 2000 poster.

b) Sökning i indexerad fil beräknas som sökning i indexfilen plus en access för att plocka blocket från datafilen. Sökning i indexfilen görs som sökning i sorterad fil (med binärsökning genom blocken), $\lceil \log_2(b) \rceil$ blockaccesser krävs. Vad är b för indexfilen (ib)?
ibfr = ⌊1024/25⌋ = ⌊40,96⌋= 40 poster/block (tips för divisionen 1024/25 se upg 1 ovan)
ib = ⌈2000/40⌉ = 50
Vi ska söka i 50 block: bland tipsen finns $\text{Log}_2(50)$=5,64, dvs 6 blockaccesser krävs för att hitta rätt post i indexfilen. Ytterligare en krävs för att hitta rätt post i datafilen.
Svar: 7 blockaccesser.

3. Hashstruktur

En hashfunktion enligt uppgiften, som omvandlar personnumret på formen yymmdd till motsvarande 6-siffriga tal, där mm bara kan vara 1-12 istället för 00-99 och dd bara 1-31 istället för 00-99, att lämna de block som representerar månader 0, 13-99 och dagar 0, 32-99 att lämnas tomma. Detta händer oavsett om hashfunktionen gör något mer än bara dettanoch hur stor hashtabellen är.

4. Accesstid

a) För att beräkna söktid måste vi först beräkna hur många block filen tar upp:
Poststorlek R = 500 byte, Blockstorlek B = 2048 byte, antal poster r = 20 000 st
bfr = ⌊2048/500⌋ = ⌊4.096⌋= 4 poster/block
b = ⌈20000/4⌉ = 5000 block.
Filen anges vara sorterad på personnummer. Sökning på personnummer blir då sökning i sorterad fil, dvs binärsökning, $\lceil \log_2(b) \rceil$. $\text{Log}_2(5000) = 12.29$ finns bland tipsen, dvs 13 blockaccesser krävs. Varje blockaccess tar 10ms, dvs 0.01s*13 = 0.13 s.
Svar: 0.13 s (= 130 ms).

b) Sökning på något filen inte är sorterad på blir som sökning i hög, i medeltal ⌈b/2⌉ accesser krävs för att hitta en viss post, MEN i detta fall ska vi söka ut ALLA som har ett visst namn, det kan alltså finnas flera. Därmed måste vi söka genom hela filen för att vara säkra på att vi hittat alla. Vi kommer att behöva b blockaccesser, dvs 5000 * 0.01s = 50 s.
Svar: 50 sekunder.

c) Primärindex på ovanstående: indexpostens storlek anges till iR=50 byte, antal poster i indexfilen blir datafilens antal block, ir = 5000 st. Denna fil får blockningsfaktor ibfr= ⌊2048/50⌋ = ⌊40,96⌋= 40 poster/block och antal block för den blir ib = ⌈5000/40⌉ = 500/4 = 125 block. Sökning i 125 block tar $\lceil \log_2(ib) \rceil$ accesser. $\text{Log}_2(125) = 3{,}3219$ finns bland tipsen, dvs 4 blockaccesser krävs, plus en för att hitta i datafilen, 5 blockaccesser. Varje blockaccess tar 10ms, dvs 0.01s*5 = 0.05 s.
Sökning på namnet påverkas inte alls, dvs samma svar som i deluppgift b.

d) Sekundärindex på personnummer: iR=50 byte, ir=20 000 poster. ibfr blir samma som i deluppgift c ovan, 40 poster per block. Antal block i indexfilen blir i ib = ⌈20000/40⌉ = 500 block. Sökning i 500 block tar $\lceil \log_2(ib) \rceil$ accesser. $\text{Log}_2(500) = 8{,}97$ finns bland tipsen, dvs 9 blockaccesser krävs, plus en för att hitta i datafilen, 10 blockaccesser. Varje blockaccess tar 10ms, dvs 0.01s*10 = 0.10 s

e) Tiden att söka ut en person givet namnet blir i detta fall precis samma beräkning som för att söka på personnummer med filen sorterad på personnummer (deluppgift a), med den reservationen att eftersom det kan finnas flera personer med samma namn kan man behöva läsa in ett block framför och bakom det block man hittar namnet i (tills man hittar ett block framför och bakom med andra namn i). Svaret blir 13 blockaccesser, dvs 0.13s plus ett fåtal hundradels sekunder till.

f) Klusterindex, iR=100 och ir=15000. ibfr= ⌊2048/100⌋ = ⌊20,48⌋= 20 poster/block och antal block för den blir ib = ⌈15000/20⌉ = 750 block. Sökning i 750 block tar $\lceil \log_2(ib) \rceil$ accesser. $\text{Log}_2(750) = 9{,}55$ finns bland tipsen, dvs 10 blockaccesser krävs, plus en till för att hitta i datafilen, plus, eftersom det kan finnas flera med samma namn, ett fåtal blockaccesser till. Alltså drygt 11 blockaccesser. Varje blockaccess tar 10ms, dvs 0.01s*11 = 0.11 s plus några hundradelar till.

Transaktioner

1. Strikt tvåfaslåsning

De inlagda operationerna visas i fetstil. Givet att dataobjekten ska låsas så kort tid som möjligt är detta den enda lösningen.

```
Start (T1)
LäsLås(A)
Read(A)
SkrivLås(B)
Read(B)
```

```
B = B - A
Write (B)
SkrivLås(C)
UnLock(A)
Read(C)
C = C - B
If C<0 then Rollback (T1)
Write=(C)
Commit (T1)
UnLock(B)
UnLock(C)
```

2. Problem med isolering

Exempel på förlorad uppdatering (det är viktigt att man visar att operationerna från T1 och T2 faktiskt inte sker samtidgt).

tid	T1	T2	
1	`Read(X)`		
2	`X=X+1`		
3		`Read(X)`	
4		`X=X-5`	
5			
6		`Write(X)`	Här skrivs den ändring T1 gjorde av X över av T2.
7	...	...	

Exempel på smutsig läsning:

tid	T1	T2	
1	`Read(X)`		
2	`X=X+1`		
3	`Write(X)`		
4		`Read(X)`	
5		`X=X-5`	
6	`ROLLBACK`		Här återställer T1 det värde på X som var innan steg 3.
7		`Write(X)`	Här skriver T2 ut ett värde baserat på det X som T1 hade ändrat men som nu är återställt.
8	...	...	

3. Solåsen - problem med isolering

Problemet som uppstår är förlorad uppdatering. Problemet kan lösas med hjälp av lås, både binära och läs-skrivlås.

4. Problem med lås

Problemet som syftas på är Deadlock. Det som händer är att vissa transaktioner (användare) låser objekt som andra är beroende av. Att vissa transaktioner samtidig kan köra beror på att bara vissa transaktioner är inblandade i Deadlocken. Lösningar på Deadlock är: förebyggande strategier: databasobjekt låses i global ordning eller Konservativ tvåfaslåsning, samt upptäckande strategier: timeout eller undersökning av Wait-for-grafen.