Utvärdering av strategier för prestandaoptimering i relationsdatabaser

(HS-IDA-EA-00-105)

Pia Gunnarsson (a96piagu@ida.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på systemprogrammeringsprogrammet under vårterminen 2000.

Handledare: Henrik Gustavsson

Utvärdering av strategier för prestandaoptimering i relationsdatabaser

Examensrapport inlämnad av Pia Gunnarsson till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för Datavetenskap.

000607

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: _____

Utvärdering av strategier för prestandaoptimering i relationsdatabaser

Pia Gunnarsson (a96piagu@ida.his.se)

Sammanfattning

När ett nytt databassystem ska tas fram och införas i en organisation ska funktioner och krav på systemet identifieras och analyseras i en designprocess. Ett krav på ett databassystem kan vara att systemet ska uppvisa en viss prestanda. Designprocessen leder så småningom fram till fysisk design av databasen och dess applikationer. Det kan finnas flera olika lösningar för fysisk design av databasen och dess applikationer som tillgodoser kraven och funktionerna som ska finnas i systemet. Dessa olika lösningsalternativ ger olika prestanda. Detta arbete ger en inblick i att fysisk design av en databas och dess applikationer påverkar prestanda och att det finns strategier för när olika lösningar kan vara lämpliga att använda för prestandaoptimering.

Nyckelord: Fysisk design, SQL, index, trigger, lagrad procedur, denormalisering.

Innehållsförteckning

1 Introduktion

Information är framtidens produkt. Nya tjänster och nischer för företag dyker hela tiden upp. Samhället blir alltmer beroende av information. För att lagra, bearbeta information och göra den tillgänglig kan databassystem användas. Men det är inte bara viktigt att information finns tillgänglig, det är även viktigt att det är lätt att få fram informationen och att det går snabbt.

Prestanda i ett datasystem är i allra högsta grad förändringsbar, t ex om ytterligare några användare läggs till i ett system så kan detta påverka prestanda märkbart. Val av hårdvara såsom till antal datorer och kapacitet har stor påverkan på prestanda, men om ett system är långsamt behöver inte det innebära att ny hårdvara direkt ska inköpas. Som första steg kan istället vara att mjukvaran undersöks i hopp om att finna flaskhalsar där.

Det finns ett antal faktorer som markant påverkar prestanda i ett databassystem. Dessa faktorer innefattar: hur och var databashanteraren är installerad, hur databasservern är konfigurerad, hur och var logg- och låsfunktioner utförs, och - viktigast enligt Rennhackkamp (1996c) - designen av databasen och dess applikationer.

De flesta databassystem är olika och design av databasen och dess applikationer som ger bra prestanda i ett databassystem ger inte nödvändigtvis bra prestanda i ett annat databassystem. En redan från början väl utförd design av ett databassystem kan vara viktig för att bevara bra prestanda över tiden. Det finns generella metoder och strategier angående hur ett databassystem kan designas för att främja bra prestanda.

Detta arbete syftar till att hitta dessa strategier och undersöka om de ger någon prestandavinst och i så fall i vilka situationer. Undersökningen av strategier har gjorts genom att olika lösningar för den fysiska designen på databasen och dess applikationer har simulerats. För att kunna avgöra vilken av lösningarna som ger bäst prestanda har mätningar av svarstider genomförts i de olika lösningarna. Svarstiderna har sedan jämförts för att ge en uppfattning om vilken lösning som ger minst svarstid och därmed ger bäst prestanda.

2 Bakgrund

2.1 Traditionellt datasystem

I datorns barndom och ungdom (mot slutet av 1960-talet) utformade man informationssystem där varje systems data lagrades i filer (Andersen, 1991). Denna typ av fil är ett datorbaserat register och består av ett antal poster. Dessa system benämns ofta i litteratur som traditionella datasystem. Ett traditionellt lönesystem hade t ex en personfil med en post för varje medarbetare och varje post bestod av en rad termer som innehöll data om personen i fråga. Varje informationssystem - varje applikation - hade sina egna filer. Huvudpoängen var att filerna i en applikation var förbehållna just denna applikation och dessutom uppdaterades filerna av denna.

När man hade utvecklat många informationssystem var det mindre lyckat att ha olika filer i de olika systemen som delvis innehöll redundant data. Det var olämpligt både med tanke på lagringsutrymme och underhållet av data. Samma upplysningar om en person, t ex adress och avdelning, som behövdes i flera olika system lagrades i flera filer. Uppdateringar av datan blev ett omfattande arbete. En ändring kunde behöva göras på flera ställen och risken för att glömma något av dem var stor. Dessutom tog dessa system upp onödigt mycket lagringsutrymme då samma data lagrades flera gånger.

Dessa problem med traditionella datasystem, löstes genom att göra en databas där all data som var relevant för verksamhetens informationssystem var samlade. Alla applikationer hade då tillgång till, och kunde använda samma data.

2.2 Databassystem

Ett databassystem är, enligt Date (1995), i grund och botten ett datoriserat system vars generella syfte är att bevara information och på begäran göra denna information tillgänglig. Information som ska bevaras kan röra sig om information som anses vara betydelsefull för individen eller organisationen som systemet är avsett att tjäna.

En databas har följande underförstådda egenskaper:

- Den representerar någon aspekt den verkliga världen, ibland kallad minivärlden eller Universe of Discourse. Förändringar i minivärlden reflekteras i databasen.
- Den är en logiskt sammanhängande samling av data som har någon inbördes mening. En slumpmässig uppsättning av data sägs inte vara en databas.
- Den är designad, byggd och försedd med data för ett specifikt syfte. Den har en tilltänkt grupp av användare och några applikationer som dessa användare är intresserade av.

Elmasri och Navathe (1994) menar att databaser och databasteknologi har och kommer att ha en stor inverkan på det allt större användandet av datorer. Databaser spelar en kritisk roll inom nästan alla områden där datorer används som t ex affärsverksamhet, verkstadsindustri, sjukvård, utbildning mm.

2.2.1 Databassystemets huvudkomponenter

Ett databassystem kan variera i storlek, alltifrån små system med en användare till stora system med många användare och med mycket data. Date (1995) menar att ett

databassystem kan sägas bestå av fyra större komponenter: data, mjukvara, hårdvara och användare (fig 1).

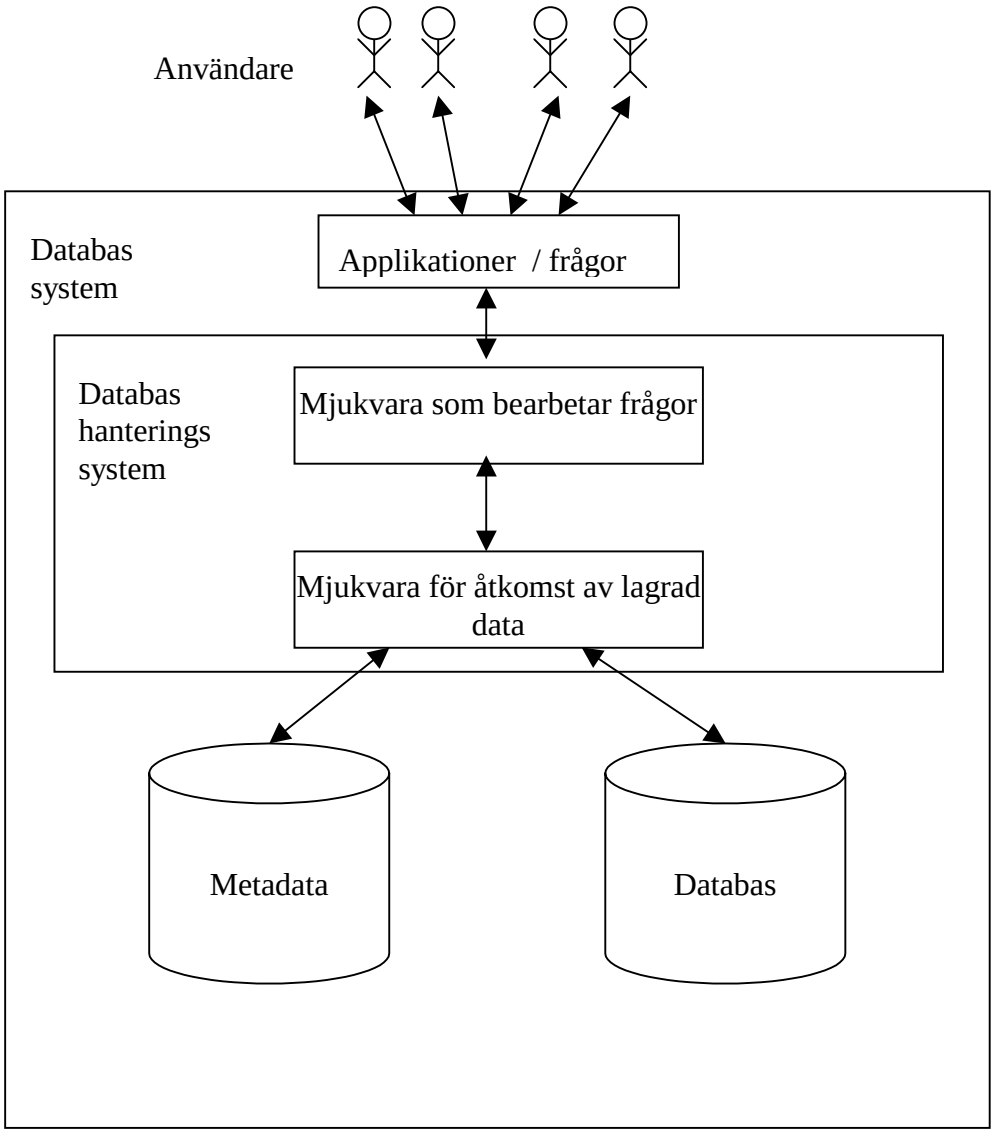


Fig 1. Komponenter i ett databassystem (Bearbetad från Elmasri och Navathe, 1994, sid 3)

Data: Enligt Date (1995) är det vanligt att säga att data i en databas är varaktig (eng, persistent), även om det är möjligt att den inte alltid består så länge. Med varaktig menas i detta sammanhang att data i databasen skiljer sig från annan mer kortlivad data tex indata, utdata, villkor, delresultat mm. En databas består av en samling varaktig data som används av applikationer inom en given organisation. Data i databasen är delad, integrerad och helt eller delvis fri från redundans.

I ett databassystem förekommer två typer av data (Elmasri och Navathe, 1994):

- Data som systemet är avsett för att lagra
- Metadata, dvs data om data.

Mjukvara: Mellan den fysiska databasen, där data i själva verket lagras, och systemets användare finns ett lager av mjukvara, dvs databashanteringssystemet (även kallad databashanterare). En begäran från användaren om åtkomst till databasen hanteras av databashanteraren. Hjälpmedel för att t ex lägga till och ta bort tabeller, hämta och uppdatera data mm är tjänster som databashanteraren erbjuder. En generell funktion som databashanteraren står till tjänst med är att skärma av databasanvändare från hårdvarudetaljer.

Hårdvara: Hårdvaran i databassystemet består bland annat av:

- Sekundärminne, t ex hårddisk som innehåller den lagrade datan, samt tillhörande I/O-enheter.
- En eller flera processorer och tillhörande primärminne, vilka används för att utföra exekveringen av databassystemets mjukvara.

Användare: Databassystemets användare kan delas in i tre större grupper (Elmasri och Navathe, 1994):

- Databasadministratörer (DBA), vilka är ansvariga för att godkänna åtkomst till databasen, för att samordna och kontrollera dess användning och för att förvärva de mjuk- och hårdvaruresurser som behövs.
- Databasdesigner, vilka är ansvariga för att identifiera data som ska lagras i databasen och för att välja lämpliga strukturer för att representera och lagra denna data. Det är också designerns ansvar att tillfredsställa databasanvändarnas krav på systemet.
- Slutanvändare, vars arbete kräver åtkomst till databasen i form av t ex förfrågning, uppdatering, rapportgenerering mm.

2.3 Databasdesign

När en organisation eller en verksamhet står inför att införskaffa eller utöka ett datasystem kan de anlita systemutvecklare för att få hjälp med att analysera verksamheten. Dessa kommer eventuellt fram till att ett databassystem skulle kunna vara en bra lösning.

Databasdesign är en process där ett antal moment kan utföras. Design av själva databasen och dess applikation kan vara två olika aktiviteter som fortgår parallellt i designprocessen (Elmasri och Navathe, 1994).

Kravspecifikation och analys: I det första steget analyseras verksamheten. Andra delar av informationssystemet som ska samverka med databassystemet måste identifieras. I dessa delar ingår t ex användare och applikationer. Kraven från användare och applikationer samlas ihop och analyseras. Kravspecifikationen delas in i två huvuddelar, de krav som ställs direkt på datan och databasen samt de funktionella krav som ställs på applikationen.

Konceptuell design: Den konceptuella designen skapas utifrån de databaskrav som kommit fram under kravspecifikationsfasen. Det finns två olika sätt att konstruera det konceptuella schemat. Ett sätt är att slå ihop kraven från de olika applikationerna och användargrupperna till en gemensam mängd av krav innan det konceptuella schemat konstrueras. Ett annat sätt är att ta fram ett schema för varje applikation och användargrupp och sedan slå samman dessa scheman till ett globalt konceptuellt schema. Vanligtvis används någon lättförstådd diagramteknik (t ex ER-schema) för att skapa det konceptuella schemat.

Transaktionsspecifikation: I transaktionsspecifikationsfasen framställs en övergripande beskrivning av applikationens transaktioner, oberoende av databashanteringssystemet, samt de tjänster som databassystemet ska erbjuda. Denna fas bör genomföras parallellt med design av det konceptuella schemat för att försäkra att all information som krävs av transaktionerna kommer att vara representerad i databasen.

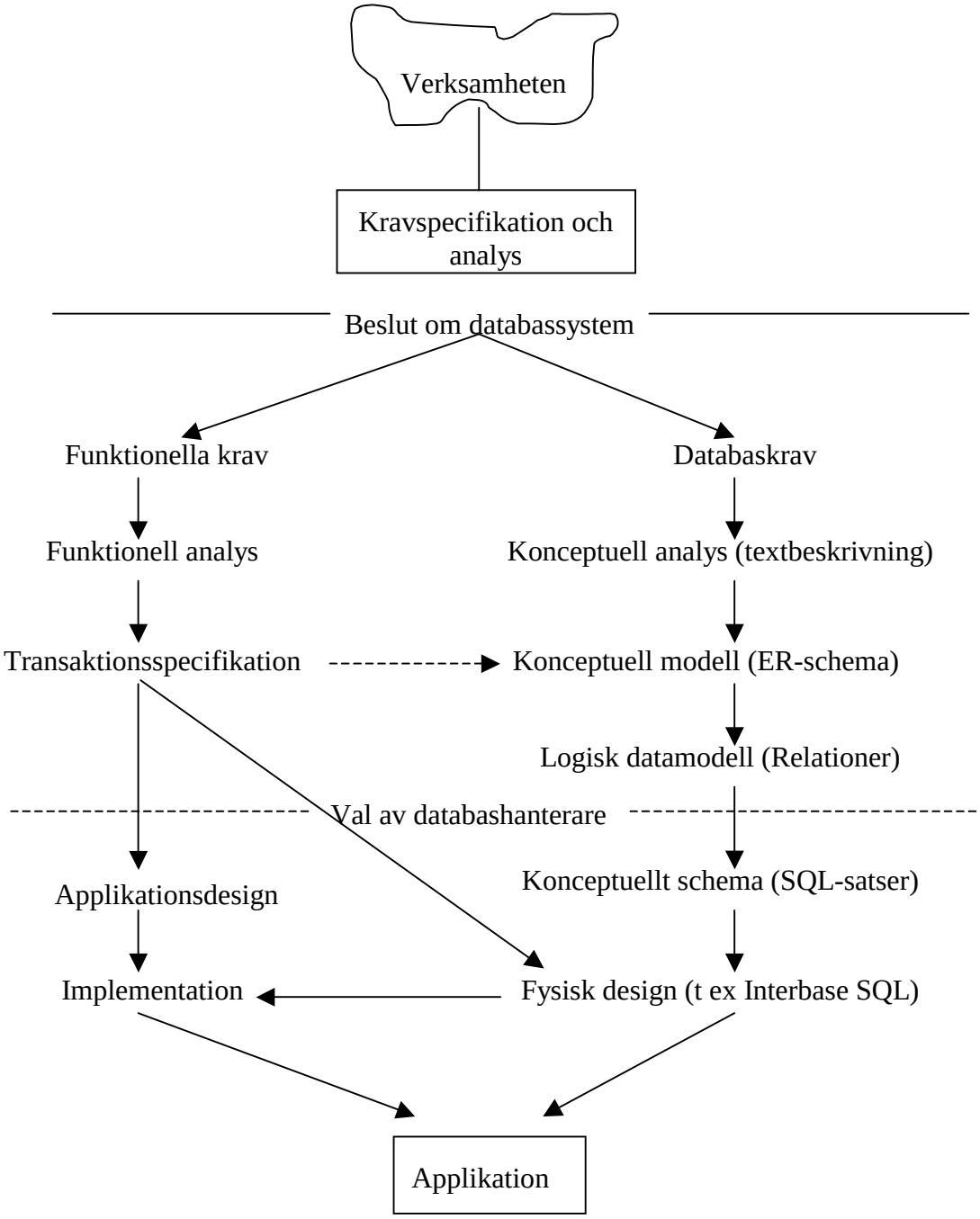


Fig 2. Designprocessen (Bearbetad från Elmasri och Navathe, 1994, sid 41)

Logisk databasdesign: Den logiska databasdesignen framställs genom en direkt mappning från det konceptuella schemat, exempelvis kan Elmasris mappning från ER-schema till relationer användas (Elmasri och Navathe, 1994, sid. 172-177). Detta dokument som är oberoende av databashanterare överförs sedan till ett konceptuellt schema, tex SQL-satser, som är beroende av databashanterare.

Applikationsdesign: Den övergripande funktionalitetsbeskrivningen som finns i transaktionsspecifikationen omsätts i en applikationsdesign som är specifik för databashanterare och programmeringsspråk. Denna design är fortfarande ett högnivådokument och saknar helt implemantationsdetaljer.

Fysisk databasdesign: I den fysiska databasdesignen så förändras det konceptuella schemat till en implementation i databasen. Det är i den fysiska designen som tabeller, index, domäner mm implementeras. Information från transaktionsspecifikationen används för att implementera delar av applikationen i form av programkod i databashanteraren samt förändringar av DDL-satser från det logiska schemat. Att implementera delar av applikationen i databashanteraren kan innebära att tex triggers implementeras. Olika databashanterare tillhandahåller olika hjälpmedel som kan implementeras.

Applikationsimplementation: I denna fas implementeras de högnivåtransaktioner som tidigare specificerats. Denna fas måste till största del genomföras efter att den fysiska databasdesignen är färdigställd eftersom arbetet är direkt beroende av de färdiga DDL-satser som producerats.

Processen för databasdesign kan delas in i en över- och underdel, där den undre delen är beroende av vald databashanterare. Olika databashanterare tillhandahåller olika sätt att implementera den fysiska databasen på.

2.4 Frågeoptimering

I databashanteraren finns en frågeoptimerare. Frågeoptimeraren tar en fråga och hittar den billigaste exekveringsplanen bland flera olika exekveringsplaner som ger samma svar (Silberschatz, 1997). Det finns, enligt Celko (1995), två olika sorters frågeoptimerare, dvs kostnadsbaserade och regelbaserade. En regelbaseradoptimerare tittar på syntaxen i en fråga och planerar för exekvering utan att ta hänsyn till storlek på tabellen eller till statistik om datan. En regelbaseradoptimerare analyserar en fråga och exekverar den i den ordning frågan är skriven, eventuellt omorganiserar den frågan till en ekvivalent form genom att använda några syntaxregler. En kostnadsbaseradoptimerare tittar på både frågan och på den statistiska datan om själva databasen för att bestämma det bästa sättet att exekvera frågan på. Dessa beslut innefattar tex om index ska användas eller ej, vilka tabeller som ska läggas i primärminnet, vilken sorteringsteknik som ska användas, mm. För det mesta, men inte alltid, tar optimeraren bättre beslut än en människa eftersom optimeraren har mer information (Celko, 1995). Frågeoptimeraren utför Select- och Projekt-operationer innan Join-operationer utförs. Detta görs i syfte för att minska storleken på join-operationerna. SQL-uttryck bör skrivas så att frågeoptimeraren utnyttjas.

Som sammanfattning kan sägas att databashanterarens generella funktion är att erbjuda användaren ett gränssnitt mot databassystemet. Användargränssnittet definieras som en gräns mot systemet under, där allting är osynligt för användaren. Därav säger man att användargränssnittet är på en extern nivå i systemet. Användargränssnittet kan definieras i språket SQL.

2.5 SQL

Ett databassystem erbjuder, enligt Silberschatz m fl (1997), två olika typer av språk: ett för att specificera databasschema och ett annat för att uttrycka databasfrågor och uppdateringar.

Ett databasschema är specificerat av en mängd definitioner uttryckta i ett speciellt språk som kallas datadefinitionsspråk (DDL efter engelskans Data Definition Language). Resultatet efter kompilering av DDL-uttryck är en mängd tabeller som lagras i en speciell fil som kallas "data dictionary". Ett "data-dictionary" är en fil som innehåller metadata, dvs data om data. Denna fil konsulteras innan verklig data läses eller ändras i databassystemet.

Lagringsstrukturer och åtkomstmetoder som används av databassystemet specificeras av en mängd definitioner i en speciell typ av DDL och kallas datalagring-och-definitions-språk. Kompilering av dessa definitioner ger av en mängd instruktioner som specificerar implementationsdetaljer av databasscheman, dessa detaljer döljs vanligtvis för användaren.

Ett datamanipuleringsspråk (DML efter engelskans Data Manipulation Language) är ett språk som gör det möjligt för användare att komma åt data och hantera den. Med datahantering menas, enligt Silberschatz m.fl. (1997), att:

- Hämta information som lagras i databasen
- Lägga till ny information i databasen
- Ta bort information från databasen
- Ändra information som lagras i databasen.

Det är, enligt Elmasri och Navathe (1994), vanligt att dagens databashanterare använder ett omfattande språk som innefattar ovanstående språk, dvs DDL och DML. Ett exempel på sådant språk är SQL.

Även om SQL sägs vara, enligt Silberschatz m fl (1997), ett frågespråk så innehåller det många andra möjligheter än att ställa frågor mot en databas. SQL innehåller också funktioner för att definiera strukturen på data, för att ändra data i databasen och för att specificera säkerhetsbegränsningar. SQL innehåller flera delar (Silberschatz m fl, 1997):

Datadefinitionsspråk (DDL). SQL DDL erbjuder kommandon för att definiera relationsschema, ta bort relationer, skapa index och ändra relationsschema.

Interaktivt datamanipuleringsspråk (DML). SQL DML innefattar ett frågespråk baserat på både relationsalgebra och relationskalkyl. Det innehåller kommandon för att lägga till, ta bort och ändra rader i databasen.

Inbäddat SQL. Den inbäddade formen av SQL är designad för att användas inom programmeringsspråk som tex C, Pascal mm.

Definition av vyer. SQL DDL innehåller kommandon för att definiera vyer.

Rättigheter. SQL DDL innehåller kommandon för att specificera åtkomsträttigheter till relationer och vyer.

Integritet. SQL DDL innehåller kommandon för att specificera integritets-begränsningar som data lagrad i databasen måste tillfredsställa. Uppdateringar som bryter mot integritetsbegränsningar är ej tillåtna.

Transaktionskontroll. SQL innehåller kommandon för att specificera början och slut på transaktioner. Flera implementeringar tillåter också explicit låsning av data för samtidighetskontroll.

SQL är ett datasubspråk för åtkomst till relationsdatabaser som hanteras av ett databashanteringssystem (Melton m fl, 1993). Det är standardiserat enligt ANSI (American National Standards Institute) och går under namnet SQL-92. I SQL-standarden ingår en del kommandon tex SELECT, CREATE, INSERT mm.

2.5.1 Select

Den grundläggande formen på en SQL-fråga innehåller satserna SELECT, FROM och WHERE och kallas SELECT-kommando. SELECT-kommandot har följande form (Elmasri och Navathe, 1994):

SELECT <attribut>
FROM <tabeller>
WHERE <villkor>

Där

<attribut> är en lista av attribut vars värde ska hämtas av frågan,

<tabeller> är en lista på de tabellnamn som krävs för att behandla frågan,

<villkor> är ett booleskt villkor som identifierar raderna som frågan ska hämta.

Med hjälp av villkor och andra operander tex **ORDER BY**, **GROUP BY**, **LIKE** mm i SELECT-kommandot kan olika frågor uttryckas som ger olika resultat. Dessa frågor kan sedan ta olika lång tid att utföra beroende på hur de är skrivna.

2.5.2 Create

För att skapa schema, relationer, index mm används i SQL kommandot CREATE. För att lägga upp en relation i databasen används CREATE TABLE. I detta kommando ges relationen ett namn och relationens attribut definieras. Varje attribut namnges och de tilldelas en datatyp. Denna datatyp anger inom vilken domän attributens värde får anta. I kommandot CREATE TABLE anges också vilket attribut som är primärnyckel och vilka attribut som är främmande nycklar i relationen. Kommandot CREATE TABLE kan se ut som följande:

CREATE TABLE kund(
kundnr integer,
kundnamn char (20) not null,
adress char (20),
Primary key (kundnr));

CREATE används på ett liknande sätt för att skapa tex schema, index, vyer och domäner. Om en skapad relation, ett index, en vy, eller dylikt, sedan ska tas bort används kommandot **DROP**.

2.5.3 Insert

Den enklaste formen av INSERT används för att lägga till en enda rad i en relation. Relationens namn och en lista över de värden som ska läggas in i relationen måste specificeras. Värden som ska läggas in i relationen måste skrivas i samma ordning som motsvarande attribut är specificerade i kommandot CREATE TABLE. För att lägga in en rad i kundrelationen skulle följande kunna skrivas:

```
INSERT INTO  kund
VALUES       (123,'Sven','Sveg')
```

Ett INSERT-kommando kan även utökas med tex ett SELECT-kommando för att på så vis lägga in flera rader i en relation.

2.5.4 Delete

För att ta bort en eller flera rader i en relation används kommandot DELETE. För att ta bort en rad från relationen kund skrivs följande:

```
DELETE FROM kund
WHERE kundnamn='Sven'
```

Om WHERE-satsen utesluts så menas det att alla rader i relationen ska tas bort. Det går också att ta bort flera specifika rader med hjälp av DELETE-kommandot och tex SELECT-kommandot.

2.5.5 Update

Om värdet på attribut som redan är inlagda i en relation ändras då kan dessa attribut ändras med hjälp av kommandot UPDATE. Om en kund, som är inlagd i relationen kund, flyttar kan värdet i attributet adress ändras enligt följande:

```
UPDATE kund
SET    adress='Skövde'
WHERE KUNDNR=123
```

Flera rader i en relation kan ändras genom att tex ett SELECT-kommando läggs in i UPDATE-kommandot.

2.5.6 Vyer

En vy i SQL är en tabell som härstammar från andra tabeller. En vy behöver inte nödvändigtvis existera i fysisk form. En vy betraktas som en virtuell tabell i kontrast till en grundtabell vars rader verkligen är lagrade i databasen. Att en vy är virtuell och därmed inte direkt lagrad i databasen medför att möjligheterna att använda UPDATE-kommandot begränsas. Möjligheter att använda SELECT-kommandot på en vy begränsas inte. En vy skapas med kommandot CREATE:

```
CREATE VIEW    arbetar_på1
AS SELECT      fnamn, enamn, pnamn, timmar
FROM           anställd, projekt, arbetar_på
WHERE          pnr=pnum AND anr=avdnr
```

Om det är så att attribut från flera relationer ofta hämtas från databasen går det att skapa en vy över dessa attribut och på så vis minska antalet joinoperationer. Om en vy ska tas bort så görs det med hjälp av kommandot DROP.

2.6 Serverspecifika konstruktioner

Förutom funktionalitet som finns beskriven i SQL-standarden så har många leverantörer av databashanterare lagt till ytterligare funktionalitet. Denna funktionalitet är specifik för varje leverantör och ger ofta stora fördelar. Om denna funktionalitet önskas användas, så kommer databassystemet att stödja endast denna databashanterare.

2.6.1 Index

Ett index för en relation är organiserad data som gör det möjligt för vissa frågor att erhålla snabb åtkomst till en eller flera rader i den relationen (Elmasri och Navathe, 1994). Indexet består av två värden: ett datavärde, som är det värde som finns i relationen, och ett pekarvärde. Detta pekarvärde innehåller adressen till det block där raden finns eller adressen till raden. Om det är som flest en pekare från lagringsstrukturen till varje block sägs indexet vara glest. Om det finns en pekare till varje rad i relationen så sägs indexet vara tätt. Det finns olika sorters index t ex primärindex, sekundärindex och "cluster"-index m fl. Primärindex och "cluster"-index är ordnade efter primärnyckelfältet i en relation medan sekundärindex är ordnat efter ett icke-ordnat fält i relationen. Det går att ha flera olika sekundärindex per relation. Index kan implementeras mha lagringsstrukturer som t ex B*-träd eller hashteknik.

I SQL skapas ett index genom att använda kommandot CREATE:

```
CREATE INDEX    enamn_index
ON              student(enamn)
```

Detta index sorteras i stigande ordning efter värdet i attributet dvs från A-Ö. Om indexet ska sorteras i fallande ordning skrivs nyckelordet DESC efter attributet. Det går att skapa ett index till en kombination av attribut. Genom att skriva nyckelordet UNIQUE innan kommandot CREATE går det att skapa en unik nyckel till relationen. Frågor som exekveras tar mindre tid om det i frågevillkoret finns attribut som har index kopplade till dem.

2.6.2 Lagrad procedur

Date (1995) hävdar att antalet meddelanden mellan klient och server kan minskas om systemet tillhandahåller någon mekanism för lagrade procedurer. En lagrad procedur är ett förkompilerat program som lagras i servern. Den lagrade proceduren anropas från klienten genom ett fjärranrop (eng remote procedure call, RPC). Date (1995) menar att användandet av lagrade procedurer **förbättrar prestanda** men att det inte är den enda fördelen. Andra fördelar är följande:

Dölja detaljer. En lagrad procedur kan användas för att dölja systemspecifika och/eller databasspecifika detaljer för användaren och på så vis tillhandahålla en större grad av dataoberoende.

Delning av kod. En lagrad procedur kan delas av många klienter.

Optimering. Optimering kan göras vid tidpunkten då den lagrade proceduren skapas istället för tidpunkten för exekvering.

Säkerhet. Lagrade procedurer kan ge bättre säkerhet. En användare kan ha rättigheter att utföra en lagrad procedur men har kanske inte rättigheter att behandla datan, som proceduren har åtkomst till, direkt.

Språket som används för att skriva lagrade procedurer innehåller alla vanliga SQL-kommandon samt även satser som används i andra programmeringsspråk. Detta gör att det går att skriva modulära program som exekverar på servern och därmed minskar nätverkstrafiken och ökar databassystemets prestanda. En lagrad procedur skapas med nyckelorden CREATE PROCEDURE.

2.6.3 Trigger

En trigger är en lagrad procedur som exekverar som ett resultat av en händelse. I ett relationsdatabassystem är en händelse vanligtvis en förändring av databasen tex att man lägger till, tar bort eller uppdaterar data (Shasha, 1992). En trigger exekveras automatiskt av databashanteraren under speciella villkor. Med automatiskt menas här att applikationer inte aktiverar triggers utan de avfyras automatiskt när en applikation utför specifika operationer på databasen. En trigger lagras och exekveras i databasen vilket medför följande fördelar (Rennhackkamp, 1996a):

- Triggern avfyras alltid när tillhörande händelse inträffar. Applikationsutvecklare behöver inte komma ihåg att inkludera funktionaliteten i varje applikation.
- Trigger administreras centralt. De kodas och testas en gång och upprätthålls sedan för alla applikationer som har åtkomst till databasen
- Den centrala aktiveringen och behandlingen av triggers passar klient-server arkitekturen bra. Ett enda anrop från en klient kan resultera i en sekvens av kontroller och efterföljande operationer som utförs av databasen. Data och operationer behöver inte skickas genom nätverket mellan klienten och servern.

Eftersom triggers är så kraftfulla bör de hanteras noggrant och användas på ett korrekt sätt. Ineffektiva triggers kan få databasservern "att gå på knäna" beroende på storleken på de jobb som avfyras i databasen och därmed försämra prestanda. Triggers kan användas för olika syften (Rennhackkamp, 1996a):

Affärsregler: Triggers kan användas för att centralt upprätthålla affärsregler. Affärsregler är restriktioner som gäller för relationer mellan tabeller eller mellan vissa rader i samma tabell.

Applikationslogik: För att centralt upprätthålla affärslogik kan trigger användas, för att tex lägga till rader i Order- och Orderpost- tabellerna när Kvantitetsvärdet i Lager-tabellen går under ett visst värde.

Säkerhet: Triggers kan användas för att kontrollera värdebaserade säkerhetskontroller tex när en operation ska utföras på en känslig tabell så kan en trigger avfyras för att kontrollera om användaren har tillåtelse att utföra operationen.

Granskning: Triggers kan lägga till poster i en granskningstabell för att logga alla operationer som utförts på känsliga tabeller.

Replikering: Vid replikering av databasen kan triggers användas som bandningsmekanism (eng recording). När en replikerad tabell ändras så avfyras en trigger som bandar ändringarna i en buffertabell. En replikerad server sprider sedan operationerna från buffertabellen vidare till måldatabaser.

Användandet av triggers begränsas av funktionaliteten som tillhandahålls av databashanteraren. Programdelen i triggers definieras med samma programmeringsspråk som används för att definiera lagrade procedurer. Triggers skapas med nyckelorden CREATE TRIGGER.

2.6.4 Domän

I en del databashanterare går det att definiera domäner som sedan kan användas när relationer skapas i databasen. Fördelen med domäner är att de inbygga datatyperna tex integer, char mm kan tilldelas ytterligare funktionalitet. Det går att definiera en domän så att den tilldelar ett defaultvärde till rader i relationen om inget värde tilldelas genom INSERT-kommandot. En domän kan också tilldelas funktionalitet så att den kontrollerar att ett värde som anges i ett INSERT-kommando befinner sig inom ett giltigt intervall. Ett domän-kommando kan se ut på följande sätt:

CREATE DOMAIN nummer INTEGER

DEFAULT 3

CHECK (VALUE BETWEEN 1 AND 4)

Denna domän nummer kontrollerar att ett värde, från tex ett INSERT-kommando, är ett heltalsvärde mellan 1 och 4. Om så inte är fallet tilldelas attributet i fråga värdet 3.

2.7 Transaktioner

Ett databassystem delar upp sitt arbete i transaktioner. En transaktion är, enligt Elmasri och Navathe (1994), en atomär enhet (av arbete) som antingen fullföljs helt och hållet eller inte alls. Transaktioner behövs, menar Date (1995), bland annat för att ett databassystem ska kunna återhämtas (eng recovery) och för att systemet ska kunna användas av flera användare samtidigt.

Med återhämtning i ett databassystem menas att databasen i sig själv återställs till ett känt korrekt tillstånd (eller åtminstone antas vara korrekt) efter att ett fel har orsakat att databasens tillstånd blivit inkorrekt (eller tvivelaktigt). För att detta ska kunna hanteras måste det säkerställas att informationen som databasen innehåller kan rekonstrueras från annan information som finns lagrad (redundant) någon annanstans i systemet (Date, 1995).

Samtidighet (eng concurrency) innebär att databashanteraren tillåter att flera transaktioner har åtkomst till samma data vid samma tidpunkt. I system som tillåter samtidighet behövs någon sorts av samtidighetskontroll för att försäkra att samtidiga transaktioner inte stör varandra (Date, 1995).

En transaktion bör upprätthåll fyra egenskaper (Elmasri och Navathe, 1994):

Atomisk (eng atomicity): Transaktioner är atomära.

Konsistens (eng consistency): En korrekt exekvering av en transaktion måste ta databasen från ett konsistent tillstånd till ett annat.

Isolering (eng isolation): En transaktion ska inte göra sina uppdateringar synliga för andra transaktioner innan den är fastställd.

Varaktighet (eng durability): När en transaktion förändrat databasen och ändringarna är fastställda, får inte dessa ändringar förloras på grund av något efterföljande fel.

Längden på en transaktion är viktig, enligt Shasha (1992). Det är viktigt eftersom längden på en transaktion har två effekter på prestanda:

- När en transaktion exekverar sätter den lås på data i databasen för att förhindra att andra transaktioner blandar sig i och stör. Ju fler lås en transaktion begär desto större är sannolikheten att transaktionen får vänta på att andra transaktioner ska släppa ett lås.
- Ju längre en transaktion *T* exekverar, desto längre tid får andra transaktioner vänta om de är blockerade av *T*.

Att dela upp större transaktioner i mindre kan äventyra korrektheten i databasen medan det förbättrar prestanda (Shasha, 1992). Låsen som en transaktion begär påverkar också prestanda i form av hur många lås den håller, vilka sorters lås den håller och hur länge. Färre lås ger bättre prestanda, läslås är bättre för prestanda än skrivlås och ju kortare en transaktion håller låsen desto bättre prestanda.

2.8 Prestanda i databassystem

Tid är pengar. Det är ord som dyker upp i olika sammanhang med jämna mellanrum. För de flesta företag är tid i allra högsta grad pengar och ett sätt för att vinna tid kan vara att företagens datasystem har hög prestanda. För personal i företagen som använder datasystemen i sitt arbete kan snabba svarstider vara en avgörande faktor om de kan utföra sin arbetsuppgifter effektivt eller ej. Schumacher (1998, sid 30) skriver: "Fråga databasadministratörer om vad deras viktigaste uppgift är beträffande hantering av databassystemen och de kommer sannolikt att svara: att försäkra utmärkt prestanda". Schumacher menar också att prestanda i ett relationsdatabassystem är direkt relaterat till hur databasen och SQL-satserna är designade.

Leverantörer av databashanteringssystem använder som försäljningsargument att deras produkt tillhandahåller en viss prestanda. För att mäta prestanda och för att kunna göra en jämförelse av prestanda i olika miljöer kan ett benchmarkingverktyg användas (Halloran m fl, 1993).

2.8.1 Prestanda

Prestanda kan sägas vara ett mått på hur effektivt ett databassystem är. Det finns ett antal faktorer som kan användas för att mäta effektivitet (Connolly, 1996).

Transaktionskapacitet. Antalet transaktioner som kan behandlas under ett givet tidsintervall.

Svarstid. Tiden det tar för en transaktion att utföras helt och hållet. Användarens synvinkel på svarstid är att den ska vara så kort som möjligt. Det finns några faktorer som påverkar svarstider som en designer inte har något inflytande över tex systemladdning och kommunikationstider.

Minneslagring. Storleken på det minnesutrymme som används av databasfilerna. En designer önskar kanske minska storleken på det minnesutrymme som används.

För en användare är prestanda i stort sett detsamma som svarstider. Ingen vill vänta för länge på ett svar eller resultat efter det att man har givit ett kommando. Det finns många faktorer som påverkar prestanda. Vad för sorts dator som används påverkar svarstider. En snabbare dator ger högre prestanda. Hur det nätverk som används är uppbyggt och vilken kapacitet det har är andra faktorer som påverkar prestanda.

Den fysiska designen av databasen kan ge upphov till flaskhalsar som försämrar prestanda. Flaskhalsar kan finnas i olika former som tex onödig uppdatering. Ett exempel på onödig uppdatering kan vara att det finns ett index till en relation men indexet används aldrig vid behandling av relationen.

2.8.2 Fysisk databasdesign

Fysisk databasdesign är, enligt Elmasri och Navathe (1994), processen där beslut tas om lagringsstrukturer och åtkomstvägar till databasen. Målet är inte bara att komma fram till lämplig struktur på hur data ska lagras i databasen, utan även att göra det på ett sätt som garanterar bra prestanda. För ett givet konceptuellt schema finns det flera alternativ på fysisk design till en given databashanterare (se fig. 3).

I den fysiska designen implementeras relationer i databasen i form av tabeller. Normalt förekommande är att dessa tabeller är normaliserade. Normalisering är en procedur där attribut grupperas efter deras funktionella beroende (Connolly m fl 1996). Resultatet av normalisering är en logisk databasdesign som är strukturellt konsistent och innehåller minimal redundans. En normaliserad databasdesign tillhandahåller inte alltid en optimal prestanda. Det kan finnas omständigheter då det kan vara nödvändigt att acceptera att vissa fördelar som en normaliserad design ger försvinner till fördel för prestanda. För att optimera prestanda kan relationer denormaliseras. Detta innebär att de delas upp i flera relationer, att relationer slås ihop eller att attribut lagras redundant i någon relation. Innan denormalisering sker bör, enligt Connolly m fl (1996), följande faktorer betraktas:

- Denormalisering gör implementation mer komplex.
- Denormalisering kan äventyra flexibiliteten.
- Denormalisering ökar prestanda vid hämtning av data men minskar prestanda vid uppdateringar.

När denormalisering görs genom att dela upp en relation i flera relationer kan detta göras genom både horisontell och vertikal uppdelning av relationer. Denormalisering ska användas för att snabba upp applikationer och hur uppdelning görs beror på organisationens krav på dessa applikationer.

För att optimera prestanda i ett databassystem kan index skapas i databasen. Index används för frågeoptimering. Det tar, enligt Elmasri och Navathe (1994), mindre tid att exekvera en fråga om något av attributen i frågevillkoret har ett index relaterat till sig. Denna prestandavinst kan vara stor om frågan rör stora relationer. Om en fråga innehåller selektion- eller join-villkor och attributen i dessa villkor har index kopplade till sig kommer tiden för att exekvera frågan att minska märkbart (Elmasri och Navathe, 1994).

I databasen kan vyer skapas. En vy är en tabell som härstammar från andra tabeller (Elmasri och Navathe, 1994). En virtuell vy ser ut som en vanlig tabell i databasen men lagras inte fysiskt i databasen. Varje gång den virtuella vyn anropas produceras den med hjälp av vydefinitionen som lagras i databasen. Att använda virtuella vyer försämrar prestanda i synnerhet om vyn härstammar från flera tabeller (Connolly mfl, 1996). En vy kan även lagras i databasen och benämns då materialiserad vy. En materialiserad vy ger, enligt Gupta m fl (1995), snabbare åtkomst av data än en virtuell vy och användas med fördel i applikationer som datalager, replikering av server mm.

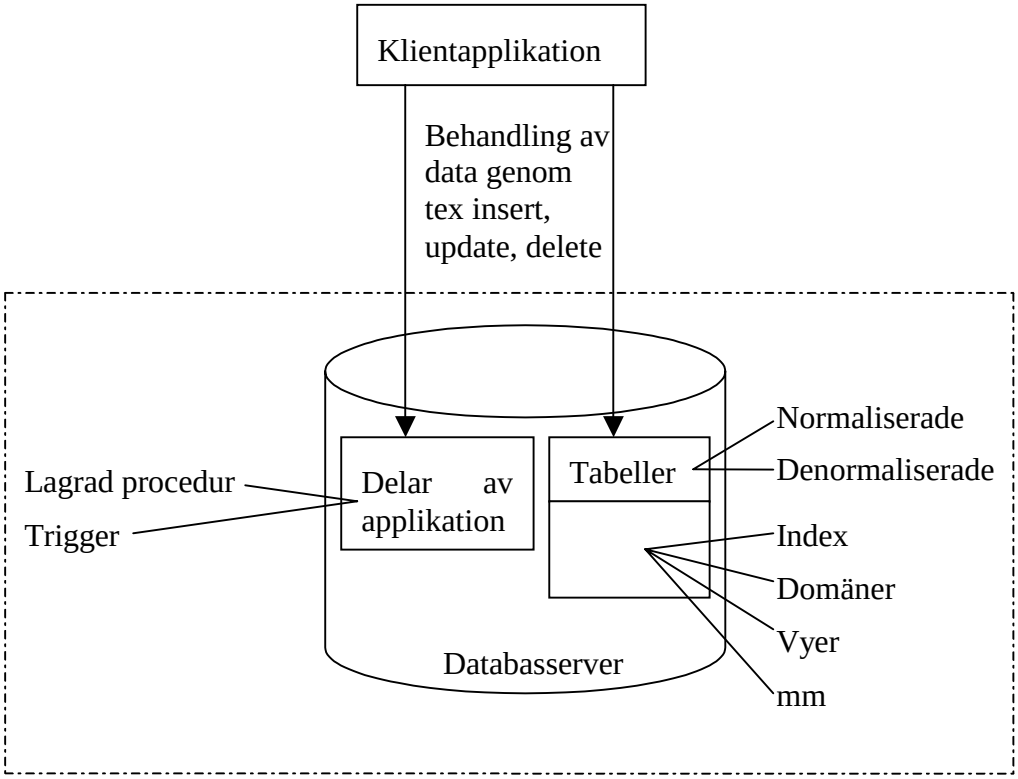


Fig 3. Fysisk databasdesign

Delar av applikationen kan implementeras i databasen i form av lagrade procedurer och triggers. Dessa definieras med ett språk som innehåller vanliga SQL-kommandon samt även satser som återfinns i andra programmeringsspråk. Detta medför att det går att skriva program som exekverar på servern och därmed minskar nätverkstrafiken. Detta i sig ökar prestandan i det totala databassystemet.

Innan några beslut angående den fysiska databasdesignen kan tas, måste det finnas en väl utformad idé om på vilket sätt databassystemet är tänkt att användas (Elmasri och Navathe, 1994). Detta görs genom att frågor och transaktioner analyseras och definieras i högnivåform.

För varje fråga bör följande specificeras (Elmasri och Navathe, 1994):

- Vilka tabeller frågan kommer att ha åtkomst till.
- Vilka fält som omfattas av selektions-villkor i frågan.
- Fälten som join-villkor i frågan specificerar.
- Fälten vilkas värden ska hämtas av frågan.

För varje uppdateringstransaktion eller operation bör följande specificeras (Elmasri och Navathe, 1994):

- Vilka fält som ska uppdateras.
- Typen av uppdatering av tabell, om något ska läggas till, ändras eller tas bort.
- Fälten som selektions-villkor i delete- eller modify-operationen specificerar.

- Fälten vars värden kommer att ändras av en modify-operation.

Förutom att analysera frågor och transaktioner måste även frekvensen av anrop till dessa uppskattats (Elmasri och Navathe, 1994). Om frågorna och transaktionerna har tidsbegränsningar måste även dessa dokumenteras. När nämnda faktorer har analyserats kan beslut om den fysiska databasdesignen tas.

Nu är det så att alla frågor och transaktioner sällan är kända vid tidpunkten för fysisk design. Databasen kan i efterhand behöva undersökas på nytt och ändras för att erhålla tillräcklig prestanda. Detta kallas för tuning. Enligt Rennhackkamp (1996b) är det generellt tre områden som kan "tunas" för att förbättra prestanda i ett databassystem. Dessa områden är: operativsystemet, databasservern och databasen. Tuning av databasserver innebär justering av databasmjukvarans installation och konfigurering av databasservern så att den samspelar bättre med operativsystemet. Databastuning innebär justering av implementationen av objekt i databasen för att förbättra applikationernas åtkomst av dessa objekt.

2.8.3 Benchmarkingverktyg

Databashanterare blir allt mer standardiserade, men en faktor som fortfarande skiljer dessa från varandra är prestandan (Silberschatz, 1997). Benchmarkingverktyg är en metod för att mäta prestanda i mjukvarusystem. Eftersom mjukvara i tex databassystem är komplex finns det ett antal olika implementationer bland olika leverantörer. Som resultat av detta skiljer sig prestanda dem emellan i utförandet av olika uppgifter. Ett system kan vara mest effektivt på att utföra en viss uppgift medan ett annat system är effektivare på en annan uppgift. En enda uppgift räcker vanligtvis inte som underlag för att ange prestandan.

Med hjälp av ett benchmarkingverktyg går det att mäta prestanda i komplexa system och ta hänsyn till detaljer som tex transaktionsmix, svarstid för transaktioner, användarbelastning och den generella transaktionskapaciteten (Fried 1998). Det finns standardiserade benchmarkingverktyg som kan användas för att undersöka prestanda i speciella system. Styrkan i dessa benchmarkingverktyg ligger i deras möjlighet att visa skalbarhet till olika plattformar med avseende på storlek på databasen, användarbelastning och svarstid för transaktioner. En nackdel med dessa benchmarkingverktyg är att de inte kan förutse faktorer som tex särskilda prestandakrav när speciella jobb körs.

Relationsdatabassystem har allsidigheten för att kunna användas i många olika situationer. TPC (Transactional Processing perfomance council's standard benchmark) erbjuder ett bra test för ett OLTP-system (Shasha, 1992). Den testar frågeoptimering och join-metoder, fast i liten utsträckning. Relationsbenchmarkingverktyg med större täckning är Wisconsin Benchmark som utvecklats av David DeWitt m fl. Utifrån Wisconsin Benchmark har sedan ytterligare benchmarkingverktyg utvecklats (Shasha, 1992), AS^{3}AP (ANSI SQL Standard Scalable and Portable Benchmark for Relation Systems) och Set Query Benchmark. Boken "The benchmark handbook" av Jim Gray (1997), behandlar dessa standardiserade benchmarkingverktyg på en mer detaljrik nivå.

Det går även att konstruera ett eget benchmarkingverktyg. Att göra ett eget benchmarkingverktyg är en kompromiss mellan realism och tillgänglig tid (Shasha, 1992). För att ett egenutvecklat benchmarkingverktyg ska vara till någon nytta överhuvudtaget måste databasen och transaktionerna specificeras. Det är viktigt att modellera data ordentligt. Om en testdatabas inte reflekterar den verkliga

applikationens tabellstorlek, kolumnvärde eller datadistributionen kan optimeraren använda en väg i testdatabasen och en annan i den verkliga databasen, vilket resulterar i ett missvisande resultat.

3 Problemprecisering

3.1 Problemdefinition

Verksamheter befinner sig i ständig förändring och så gör även deras databassystem. Funktionalitet läggs till och tas bort, användare läggs till och systemet utökas. Detta gör att prestanda förändras och kanske drastiskt försämras. Många gånger löses problemen med otillräcklig prestanda genom att snabbare hårdvara köps in och tillförs till systemet. Det kan dock finnas andra lösningar som bör undersökas först. Databassystemet kan kanske vara i behov av finjustering, sk tuning. Tuning kan utföras på flera nivåer i ett databassystem, bl a på designnivå. Tuning på denna nivå kan innebära tex denormalisering, lägga till eller ta bort index, optimera frågor mm. Databasen bör designas väl från början då det kan vara ett problem att ändra i efterhand.

Kirkwood (1992) menar att det finns en generation av designers som inte blivit lärda att designa databaser med avsikt på prestanda. Den logiska designen betraktas som den enda datadesignen som behövs och implementeras direkt utan justeringar för prestanda. När systemet sedan inte uppträder effektivt så tillförs mer hårdvara. En sådan lösning kommer endast att lyckas till en viss gräns då grundläggande flaskhalsar som tex låsning inte kommer att försvinna oavsett hur snabb hårdvaran är. Vad som behövs är en metod för fysisk databasdesign där prestandaoptimering ingår som en del av metoden. Valet av databashanterare har stor betydelse för den fysiska databasdesignen.

Fysisk databasdesign behandlas av Silberschatz (1997) och Date (1995) i form av att relationer ska normaliseras. Att nöja sig med att normalisera relationer och lägga upp databasen efter detta ger inte, enligt Connolly m fl (1996), den optimala prestandan. I syfte att optimera prestanda kan vissa delar av databasen denormaliseras, därmed inte sagt att normalisering ska utelämnas. Beroende på databassystemets syfte kan denormalisering ge förbättrad eller försämrad prestanda. Connolly ger en tumregel för när denormalisering kan användas. Denormalisering kan vara en valmöjlighet då prestanda ej är tillräcklig och relationen sällan uppdateras medan data från den hämtas ofta. Hur stor vinsten blir vid denormalisering behandlas däremot inte(Connolly m fl, 1996).

För ett givet konceptuellt schema finns det flera alternativ på fysisk design till en given databashanterare. Behandling av data kan designas att utföras på olika ställen i databassystemet, tex via SELECT-satser i applikationen eller med hjälp av procedurer i databashanteraren. I litteraturen (Elmasri och Navathe, 1994; Date, 1995; Connolly m fl, 1996; Silberschatz, 1997) står det att läsa om olika operatorer och konstruktioner som kan konstrueras i SQL, tex index, vyer, domäner, triggers mm. Finesserna med dessa operatorer och konstruktioner behandlas till viss del i databaslitteratur, men ytterst lite handlar om prestanda. Index och triggers kan förbättra prestanda, men inte alltid. Används de på ett felaktigt sätt kan de till och med försämra prestanda. I vilka situationer och under vilka förutsättningar dessa operatorer och konstruktioner ska användas för att optimera prestanda behandlas väldigt lite i litteraturen. I vilka situationer ska dessa operatorer och konstruktioner användas och i vilka situationer ska de absolut inte ska användas? Detta projekt är avsett att leda fram till några riktlinjer beträffande detta.

Det ej är väl beskrivet i databaslitteratur om när olika operatorer och konstruktioner i SQL ska användas för att optimera prestanda och det står inte heller något nämnvärt

om hur stora prestandavinsterna blir om dessa operatorer används. Ett försök till att undersöka detta skulle kunna göras genom att fysiskt designa en databas och dess applikationer på några olika sätt.

Projektet ska undersöka hur stora prestandavinsterna blir vid användandet av dessa operatorer och konstruktioner i jämförelse mot att inte använda dem. Syftet med detta projekt är att undersöka om det finns några strategier för när olika operatorer och konstruktioner i SQL är lämpligast att använda för att optimera prestanda.

3.2 Avgränsning

Projektet kommer att inrikta sig på faktorer, som finns i den fysiska databasen och i applikationerna, som påverkar prestanda. Faktorer som rör hårdvara och nätverk kommer inte att beaktas. Projektet kommer enbart att rikta sig mot relationsdatabaser och ej några andra typer av databaser. Detta för att relationsdatabaser är vanligt förekommande och att det antas finnas litteratur som behandlar prestandaoptimering i dessa databaser.

Det finns två olika sorters vyer, nämligen virtuella och materialiserade. De fördelar som vyer för med sig, tex säkerhet, får betalas i form av prestanda. Materialiserade vyer kan användas för att optimera prestanda inom vissa områden tex datalager. Vyer kommer inte att beaktas i detta arbete.

Frågeoptimerare kommer att ha betydelse för detta arbete men det innefattar inte att göra en ny eller att modifiera en frågeoptimerare. Projektet kommer istället att inriktas mot att hitta sätt att använda SQL-uttryck så att frågeoptimeraren verkligen används.

När situationer för att optimera prestanda med hjälp av olika operatorer och konstruktioner i SQL är identifierade ska testdatabassystem som reflekterar dessa tas fram. Ett testsystem som enbart är normaliserat och utan konstruktioner för optimerad prestanda ska designas samt ett system som är designat för att optimera prestanda. Testdatabassystemen ska skrivas i ett och samma frågespråk, SQL. Databashanteraren Interbase 5.0 kommer att användas då den är tillgänglig samt att det i Interbase finns funktioner för att konstruera lagrade procedurer och triggers mm. Begränsade tester med begränsade antal klienter och mot en given server ska utföras på detta testsystem.

För att mäta prestanda i databassystem kan ett benchmarkingverktyg användas. Standardiserade benchmarkingverktyg är konstruerade att mäta prestanda i ett databassystem som är uppbyggda med en enda designlösning. Denna designlösning testas sedan på olika plattformar och en prestandamätning erhålls. Benchmarkingverktyget som ska tas fram i detta arbete är tänkt att fungera lite annorlunda. Det ska mäta prestanda i olika designlösningar men på samma plattform, dvs mot en och samma server. Tanken är att ta fram flera olika designlösningar på en databas och utföra mätningar på dessa lösningar för att sedan kunna jämföra resultatet och eventuellt fastställa att vissa lösningar ger bättre prestanda än en annan lösning. Mätningar av prestanda ska göras utifrån svarstider.

3.3 Förväntat resultat

Det förväntade resultatet är att detta arbete fått fram mätningar av svarstider som visar om det finns situationer där en konstruktion ger en prestandavinst. Förväntat resultat är även en sammanfattning över de situationer och förutsättningar då det är lämpligast att använda olika operatorer och konstruktioner från SQL i syfte att optimera prestanda.

4 Metoder

För att uppnå de målsättningar som satts upp i en undersökning behövs metoder. Utan grundläggande kunskaper i och förståelse för metodfrågor kan det bli svårt att nå dessa mål. Metod är en nödvändig - men inte en tillräcklig - förutsättning för att kunna utföra en seriös undersökning. En metod är i sig enbart ett redskap och ger inte några svar, men är en förutsättning för att de resultat som erhålls i en undersökning ska ge en bättre och sannare uppfattning om de förhållanden som undersöks (Holme m fl, 1996).

Det finns en mängd skilda typer av undersökningar. Några av de vanligaste har fått beteckningar så att det går att skilja dem åt utan att behöva gå in på omfattande förklaringar. De flesta undersökningar kan klassificeras utifrån hur mycket kunskap som finns om problemområdet innan undersökningen startar (Patel m fl, 1994).

Explorativa: När det finns stora luckor i kunskapen kan undersökningen vara utforskande. Det främsta syftet med explorativa undersökningar är att inhämta så mycket kunskap som möjligt om ett bestämt problemområde. En explorativ studie undersöker ett nytt, ganska okänt område och resultatet kan utgöra en startpunkt för vidare undersökningar.

Deskriptiva: Inom problemområden där det redan finns en viss mängd kunskap, som börjat systematiserats i form av modeller, kan undersökningen vara beskrivande och kallas då deskriptiv. Denna studie beskriver förhållanden inom ett område.

Hypotesprövande: När kunskapsmängden inom ett problemområde blivit mer omfattande och teorier utvecklats kan undersökningen vara hypotesprövande. Hypotesprövande undersökningar förutsätter att det finns tillräcklig kunskap inom ett område så att det går att härleda antaganden från teorin om förhållanden i verkligheten.

Beroende på undersökningens natur, på problemdefinitionen, vilka resurser som finns, vilka mål som är uppsatta mm kan vissa metoder vara mer lämpliga än andra. Det kan vara lämpligt att använda sig av flera metoder. Exempel på metoder kan vara litteraturstudie, intervju, enkät, fallstudie, experiment mm.

När bakgrund till problemområde och problemdefinitionen är klar är det hög tid att fundera på tillvägagångssätt för att få tag på information som behövs för undersökningen. Det är inte bara att kasta sig över första bästa metod och tillämpa den, utan flera möjliga metoder bör beaktas innan definitivt val av metod görs. Inför ett val av metod bör följande frågor besvaras:

- Vilken information behövs för att lösa problemet?
- Varför behövs just denna information?

Först därefter går det att gå över till frågor som rör vilket tillvägagångssätt som är det bästa när det gäller att samla in information och hur dessa data ska bearbetas. Vilken metod som väljs för insamling av information måste alltid kritiskt granskas för att avgöra hur tillförlitlig och giltig den informationen är som erhålls. Reliabilitet eller tillförlitlighet är ett mått på i vilken utsträckning ett instrument eller tillvägagångssätt ger samma resultat vid olika tillfällen under i övrigt samma omständigheter. Validitet eller giltighet är ett betydligt mer komplicerat begrepp. Det är ett mått på om en metod undersöker det som avses att undersöka.

Detta projekt kan delas upp i flera faser, dvs förstudie, testfall, implementation och simulering (se fig 4).

Förstudie	Testfall	Implementation	Simulering
	Design av testfall då konstruktioner är lämpliga för prestandaoptimering	Design och implementation av benchmarking-verktyg	Simulering av testfall

Fig 4. Översikt över projektfaser.

Detta projekt kan ses som en blandning av en explorativ studie och en hypotesprövning. Hypotesen kan sägas vara: det finns konstruktioner som kan användas i fysisk databasdesign som optimerar prestanda i databassystemet. Förstudien till projektet ska resultera i att identifiera dessa konstruktioner och sammanställa dem. Det är också av intresse att få veta när konstruktioner är lämpliga att använda och hur stora prestandavinsterna i fråga är. Om det finns specifika situationer för när konstruktioner är lämpliga att använda så ska dessa resultera i ett antal testfall. Testfallen ska sedan användas som underlag för simuleringar i ett implementerat databassystem. Mätningar på prestanda ska göras i dessa simuleringar. För att inhämta information rörande detta projekt skulle metoder som tex litteraturstudie, enkät och experiment kunna vara lämpliga. Litteraturstudie eller enkät för att identifiera konstruktionerna och situationer. Experiment kan vara lämpliga för att undersöka prestandavinster.

4.1 Litteraturstudie

De vanligaste källorna till information är böcker, artiklar publicerade i vetenskapliga tidskrifter samt rapporter (Patel m fl, 1994). I böcker hittas oftast försök till sammanställningar och systematiseringar av den kunskap som finns inom ett problemområde. I artiklar, rapporter och konferensskrifter hittas däremot de senaste rönen eftersom böcker tar relativt lång tid att förlägga.

Den litteratur som behövs i en undersökning kan fås genom sökning på biblioteken. Ett annat ställe att hitta information i är manualer. Tillförlitligheten i denna information kan däremot ifrågasättas då manualerna är framtagna av leverantörer till systemen. Leverantörerna är i första hand intresserade av att påvisa fördelar med sina system och redovisar inte eventuella nackdelar.

För detta projekt skulle litteraturstudien i första hand inrikta sig på litteratursökning i böcker som behandlar databassystem och artiklar utgivna i vetenskapliga databastidskrifter, tex DBMS. Sökord för att avgränsa sökningen: prestanda, benchmark, relationsdatabaser, trigger, lagrad procedur, index o.dyl. Påträffad litteratur ska värderas för att avgöra om informationen är aktuell och trovärdig. Beträffande källkritik så antas att författare till böcker och artiklar i vetenskapliga tidskrifter kan betraktas som tillförlitliga. Införskaffad litteratur ska sedan studeras för att hitta relevant information angående problemställningen. Relevant fakta ska sedan analyseras och sammanställas på ett överskådligt sätt i rapporten.

Fördelar och nackdelar med att använda litteraturstudie som metod i detta projekt.

Fördelar:

- Detta arbete kan i litteratur hitta information om konstruktioner som kan användas i fysisk databasdesign för att optimera prestanda.
- Detta arbete kan i litteratur finna information om hur ett benchmarkingverktyg är uppbyggt och hur dessa konstrueras.
- Tillförlitlighet hos författare till böcker och artklar kan betraktas som god.
- Tiden det tar att utföra en litteraturstudie går lättare att kontrollera än tex med att skicka ut en enkät.

Nackdelar:

- Det är svårt att göra en bra analys på informationen
- Litteraturen kan vara föråldrad.

4.2 Enkät

Intervjuer och enkäter är tekniker för att samla in information genom frågor. Det som skiljer en intervju från en enkät är att en intervju är personlig, dvs den som intervjuar träffar svarspersonen (Patel m fl, 1994). Enkät är en bra metod för att samla in en viss typ av information på ett snabbt och förhållandevis billigt sätt. En enkät måste konstrueras så att den kommer att ge den information som behövs, så att svarspersonerna kan acceptera den och som inte medför några problem vid analys och tolkning av svaren.

Det är svårare att konstruera en bra enkät än vad som normalt föreställs (Bell, 1993). Arbete och stor möda måste läggas ner på att välja frågeområden, göra frågorna entydiga, utforma lämplig layout, utprova enkäten, distribuera den och se till att enkätsvaren kommer tillbaka. Redan på planeringsstadiet måste viss uppmärksamhet ägnas åt hur svaren ska analyseras, det räcker inte med att göra det när svaren börjar komma in. Tid måste också ägnas åt att få kontakt med svarspersoner och motivera dem till att avsätta tid för att besvara enkäten. Visst källkritiskt arbete behövs läggas ner. Information från böcker och artiklar bör kunna betraktas som mer tillförlitliga än information från enkätfrågor besvarade av personer som inte har något intresse av att fakta i undersökningen blir korrekta.

Den förberedande inläsningen och projektplaneringen innebär att viktiga områden som ska undersökas identifieras. Målsättningen och problemdefinitionen får avgöra vilka frågor som är viktiga att ställa för att målen ska uppfyllas. Därefter skrivs tänkbara frågor ner på kort eller separata papper. Det krävs normalt flera stadier i formuleringen av frågorna för att minska mångtydigheten och för att få precisa frågor så att svarspersonerna förstår vad som menas med frågorna. Ju mer strukturerad en fråga är , desto lättare är det att analysera den.

För detta projekt skulle en enkätstudie kunna ge svar på vad det finns för konstruktioner att tillgå för fysisk databasdesign i syfte att optimera prestanda. Studien skulle rikta sig till databasdesigners eftersom dessa borde vara insatta i problemområdet. Enkätstudien skulle också kunna resultera i att specifika situationer för när olika konstruktioner är lämpliga att använda identifierades. Svarspersonerna har kanske även tips och idéer som inte påträffas i litteraturen. Framför allt kan de säkert tala om när konstruktioner är olämpliga att använda för att optimera prestanda

och vad för konsekvenser olika alternativ kan ge. Även en viss undersökning av prestandavinster skulle kunna innefattas i en enkätstudie. Responsen på en fråga angående prestandavinster skulle med stor sannolikhet röra sig om rena uppskattningar från svarspersonernas sida och därmed sakna vetenskaplig grund.

Fördelar och nackdelar med att använda enkät som metod i detta projekt.

Fördelar:

- Detta arbete kan få information från en stor grupp tex databasdesigners, som är familjära med databassystem och prestanda.
- Billigare och snabbare än en intervju

Nackdelar:

- Det är svårt att hitta en grupp databasdesigners som har liknande förutsättningar och som är intresserade av att deltaga i undersökningen genom att besvara enkäten. Om svarspersonerna arbetar med tex helt olika databassystem kan det vara svårt att jämföra och analysera deras svar.
- Det är svårt att avgöra hur tillförlitlig informationen är som erhålls genom en enkätstudie.
- En enkätstudie kan ta lång tid och tidsåtgången kan vara svår att kontrollera om svarspersonerna inte svarar och extra påminnelser måste skickas ut.
- Stor möda måste läggas ner på enkätfrågorna så att dessa resulterar i relevant information.

4.3 Experiment

Experiment omfattar en undersökning av orsaker och samband genom att använda tester som kontrolleras av projektet (Dawson, 2000). Experiment innefattar i detta projekt implementation av ett databassystem och simuleringar som ska köras mot detta. Dawson (2000) refererar till Saunders m fl (1997) som beskriver vad ett experiment vanligen innefattar. Saunders beskrivning har tolkats och översatts för att passa till detta projekt:

- Definition av en teoretisk hypotes, tex att triggers ger prestandavinster.
- Välja ut exempel från problemområde, tex situationer då konstruktioner sägs ge prestandavinster.
- Tilldela dessa exempel till olika experimentella villkor, dvs testfall och simuleringar.
- Utföra planerade förändringar av databasen, tex införa triggers, index mm i den fysiska databasen.
- Mätningar innan och efter förändringar av databasen, dvs mäta svarstider.

4.3.1 Implementation och simuleringar

Innan några simuleringar kan göras i detta projekt måste en databas implementeras. När en mjukvara ska implementeras är det inte bara att ta fasta på första bästa idé och börja koda. För att mjukvaran som ska konstrueras ska motsvara krav och förväntningar behövs metoder. En implementationsmetod ska ge svar på hur mjukvaran ska implementeras. Metoder för att bygga en mjukvara innehåller bland

annat (Pressman, 1997): Analys, design, programkonstruktion och testning. I analysfasen ska syftet med systemet samt krav på detta identifieras. De identifierade kraven förfinas med avseende på data, funktioner och beteende. Oavsett storlek på ett system finns det ett antal frågor som bör besvaras och analysera. I designfasen görs en modell över data, funktioner och beteende som skall verka som en guide vid kodning och testning. Sedan sker själva kodningen och testkörningar för att kontrollera att systemet uppfyller kraven samt att felaktigheter som gjorts under design och konstruktion avslöjas och rättas till.

I detta projekt är det aktuellt att implementera ett databassystem. Databassystemet ska ha funktioner som gör det möjligt att mäta svarstider, dvs systemet ska agera som ett benchmarkingverktyg. Syftet med systemet är att utföra mätningar av prestanda i olika lösningar i fysiskdesign av databasen. Framtagandet av databassystemet kommer i stort sett följa den designprocess som beskrivs i kapitel 2.3 i denna rapport. En noggrann design av data som ska lagras i databasen, transaktioner och den fysiska databasen måste utföras. En databas utan konstruktioner för optimerad prestanda ska först tas fram.

I projektet ska ett antal testfall tas fram. Dessa ska representera situationer när det är lämpligt att använda trigger, lagrade procedurer, index mm för att optimera prestanda. Simuleringar av dessa testfall ska köras mot databasen och mätningar av svarstider ska utföras. Simuleringarna kommer att vara olika transaktionsmixar bestående av förfrågningar och uppdateringar. Databasen ska sedan vidareutvecklas med konstruktioner som ska optimera databassystemet. Nya simuleringar ska sedan köras mot databasen. Ett antal frågor som måste besvaras och dokumenteras under designprocessen är: Hur ska svarstider mätas? Hur ska simuleringarna se ut? Vilka transaktionsmixar är intressanta för undersökningen? För att erhålla bra resultat av mätningar måste simuleringarna mot systemet dokumenteras. Detta för att en analys av mätningarna sedan ska kunna göras.

Fördelar och nackdelar med att utföra implementation i detta projekt:

Fördelar:

- Det är möjligt att skräddarsy systemet så att mätningar av prestanda utförs på ett sätt som är relevant för projektet.
- Inriktar sig på faktorer som är relevanta för projektet.
- Det är billigare än att köpa in ett benchmarkingverktyg

Nackdelar:

- Det är lätt att införa fel i implementationen vilket kan medföra att mätningar inte blir korrekta

4.3.2 Standardiserade benchmarkingverktyg och simuleringar

Det är ofta mest kostnadseffektivt att köpa än att göra själv. Behövs däremot mycket modifiering av den inköpta komponenterna kan det vara billigare att utveckla dem helt och hållet på egen hand. Inköp kräver inte mindre kompetens, bara annorlunda sådan. Det behövs kunskap för att utvärdera produkten, leverantören samt att kunna foga in den köpta produkten i det övriga systemet.

I stället för att implementera ett eget benchmarkingverktyg och köra simuleringar mot kan standardiserade benchmarkingverktyg användas. Simuleringar körs då mot den databas som följer med benchmarkingverktyget. Standardiserade

benchmarkingverktyg undersöker dock inte prestanda på ett sätt som är intressant för detta projekt. Standardiserade benchmarkingverktyg undersöker hur prestanda förändras när antal processorer, minnen, användare och storlek på tabeller mm ändras (Halloran m fl, 1993). De undersöker inte hur prestanda förändras om den fysiska designen och applikationerna ändras och det är dessa förändringar som detta projekt intresserar sig för.

4.4 Val av metoder

För att hitta information om konstruktioner som kan användas i fysisk databasdesign för att optimera prestanda väljs metoden litteraturstudie. Detta baseras på bedömningen att information som kan fås genom en enkät inte skulle tillföra någon information som inte kan fås genom en litteraturstudie. Därför väljs litteraturstudie framför en enkät. För att få ut någon värdefull information från en enkät skulle den behöva konstrueras med öppna frågor. Svaren som erhålls från öppna frågor är svåra att analysera. För detta projekt bedöms det att en litteraturstudie täcker problemområdet bättre än vad resultatet från en enkätstudie skulle göra. Ett alternativ skulle kunna vara att använda sig av både litteraturstudie och enkät. Inom tidsramen för projektet finns det inte tid att använda båda metoderna.

I projektet ska ett benchmarkingverktyg implementeras. Benchmarkingverktyget ska innan implementation designas. För design av verktyget kommer metoden litteraturstudie att användas. Det finns böcker och artiklar som behandlar standardiserade benchmarkingverktyg. En del av denna litteratur innehåller även tips och råd för hur ett benchmarkingverktyg ska konstrueras. Utifrån denna information ska benchmarkingverktyget i detta projekt designas.

Förstudie	Testfall	Implementation	Simulering
Förslag och riktlinjer från litteratur om när konstruktioner ger förbättrad prestanda	Design av testfall då konstruktioner är lämpliga för prestandaoptimering	Design och implementation av benchmarking-verktyg	Simulering av testfall
Litteraturstudie	Litteraturstudie	Experiment	Experiment

Fig 5. Översikt över projektfaser och valda metoder.

För att undersöka hur stora prestandavinsterna blir vid användandet av olika konstruktioner i en databas väljs metoden experiment, som i detta projekt består av implementation och simuleringar. I den inledande litteraturstudien har det visat sig att det finns att läsa om olika konstruktioner som kan förbättra prestanda. Ibland nämns också att felaktigt använda konstruktioner kan försämra prestanda. Däremot har inget påträffats i den inledande litteraturstudien som behandlar hur stora prestandavinster det rör sig om. Syftet med detta projekt är att undersöka när konstruktioner ger prestandavinster och hur stora dessa är. Experimentet i detta projekt ska simulera

olika belastningar på databassystemet och mäta svarstider. Resultatet kan sedan ses som en fingervisning om när olika konstruktioner är lämpliga att använda i fysisk databasdesign för att optimera prestanda. Eftersom standardiserade benchmarkingverktyg inte undersöker prestanda på ett sätt som är intressant för projektet ses inte dessa som ett alternativ till att implementera ett eget benchmarkingverktyg.

En fråga som måste ställas och besvaras i samband med att experiment valts som metod är: Vad händer om experimentet ej blir klar i tid? Har projektet ändå något resultat att redovisa? Experimentet föregås av en litteraturstudie. Tidigare genomgång av litteratur i detta projekt har visat att det finns mycket skrivet om konstruktioner men att dessa inte är sammanställda i samband med prestandaoptimering. Litteraturstudien kommer att resultera i någon sorts sammanställning på detta - vilket kan ses som ett resultat.

5 Genomförande

Under litteraturstudien som genomfördes i detta arbete påträffades generella förslag och tumregler för hur bland annat SQL-frågor kan skrivas för att optimera prestanda. Kapitel 5.1 ger en beskrivning av en del av dessa tumregler. Det finns litteratur som behandlar tuning, tex "Database Tuning: a principled approach" (Shasha, 1992). I denna litteratur finns förslag till prestandaoptimering av databassystem. Dessa förslag har använts som underlag för de testfall som finns beskrivna i kapitel 5.2. Tre av testfallen simulerades i detta arbete. Beskrivning av dessa simuleringar finns i kapitel 5.4.

5.1 Generella förslag för prestandaoptimering

Det finns, enligt Celko (1995), inga regler för hur kod ska skrivas som tar vara på de bästa fördelarna i alla frågeoptimerare, dessa är helt enkelt för olika. En del frågeoptimerare är kostnadsbaserade och en del är regelbaserade. Det går inte ge några allmänna regler, däremot går det att göra några generella förslag till prestandaoptimering. Vad som förbättrar prestanda i en SQL-implementation kanske inte förbättrar prestandan i en annan implementation, den kanske till och med försämrar den. Litteraturstudien har resulterat i några generella förslag för prestandaoptimering beträffande:

- Index
- Denormalisering
- Lagrade procedurer och triggers

5.1.1 Index

Den primära anledningen till att skapa index är, enligt Roti (1996), för att förbättra prestanda, en annan anledning är göra rader i en tabell i databasen unika. Celko (1995) menar att index ska skapas till tabeller i databasen för att optimera söktiden för frågor, men han menar även att inte fler index än vad som är absolut nödvändigt ska skapas. Index måste uppdateras och eventuellt omorganiseras när INSERT, UPDATE, eller DELETE av rader sker i en tabell. För många index kan resultera i att extra tid går åt att vårda index som sällan används. Men ännu värre, närvaron av ett index kan lura optimeraren att använda det fast den inte ska. Prestanda kan både förbättras och försämras beroende på hur SQL-frågor skrivs. Här följer ett antal generella förslag och tumregler om hur SQL-frågor kan skrivas för att optimera prestanda.

Tumregel 1: Sätt de mest restriktiva först.

När en fråga innefattar flera AND-predikat som söker efter konstanta värden, är det en bra idé att sätt de mest restriktiva predikaten först (Celko, 1995).

Alternativ 1	Alternativ 2
SELECT * FROM Student WHERE kön = 'kvinna' AND betyg = 'A',	SELECT * FROM Student WHERE betyg = 'A' AND kön = 'kvinna';

Antag att det finns färre studenter med betyg "A" än det finns kvinnliga studenter, dvs betyg är det mest restriktiva predikatet i detta exempel. Då kommer antagligen frågan i alternativ 1 att ta längre tid att exekvera (Celko, 1995).

Tumregel 2: Små tabeller sist i FROM och först WHERE.

För att förena tex två tabeller och hämta information från dessa kan svarstiden påverkas genom hur frågan skrivs. Beroende på hur frågan skrivs kommer frågeoptimeraren att välja vilket index den ska använda. Genom att placera tabellen med minst antal rader sist i FROM-satsen och uttrycket som använder den tabellen först i WHERE-satsen kan frågan snabbas upp (Celko, 1995). Antag att det finns två tabeller som har attributet kod gemensamt och att tabellen Koder har minst antal rader. För att snabba upp frågan kan den då skrivas enligt följande:

```
SELECT *
FROM Order AS O1, Koder AS C1
WHERE C1.kod = O1.kod;
```

Tumregel 3: Vid skilt från (<>) används inte index.

Jämförelseuttrycket <> (skilt från) har några unika problem (Celko, 1995). De flesta optimerare antar att denna jämförelse kommer att returnera fler rader än den förkastar, så optimerare föredrar en sekventiell sökning i tabellen och kommer inte att använda något index som finns till en kolumn.

Exempel: För att hitta någon på Irland som inte är katolik kan frågan skrivas som i alternativ 1. Det går också att dela upp olikheten enligt alternativ 2. Frågan i alternativ 2 tvingar fram användandet av ett index. Utan ett index på religion kan dockOR-versionen på predikatet (alt 2) ta längre tid att exekvera.

Alternativ 1	Alternativ 2
SELECT * FROM Irland WHERE religion <> 'Katolik';	SELECT * FROM Irland WHERE religion < 'Katolik'; OR religion > 'Katolik';

Tumregel 4: Använd index vid COUNT()

Ett annat trick som ofta fungerar är att använda sig av ett index vid COUNT(), eftersom själva indexet redan kan ha räknat ut antal rader (Celko, 1995). Alternativ 1 i detta exempel kan vara långsammare är alternativ 2. En smart optimerare kontrollerar automatiskt indexerade kolumner när den upptäcker COUNT(*), men det kan vara värt att prova alternativ två.

Alternativ 1	Alternativ 2
SELECT COUNT(*) FROM Sales;	SELECT COUNT(salesid) FROM Sales;

Tumregel 5: Undvik LIKE-predikat vid strängar

Ett speciellt problem med strängar är att optimerare ofta stannar vid '%' eller '_' i ett LIKE-predikat (Celko, 1995). '%' och '_' kallas för wildcard och ersätter en eller flera tecken i en sträng. I alternativ 1 kommer inte frågeoptimeraren att använda sig av ett index som eventuellt finns på namnkolumnen, men det kommer den att göra i alternativ 2.

Alternativ 1	Alternativ 2
SELECT * FROM Student WHERE namn LIKE 'Sm_th';	SELECT * FROM Student WHERE namn BETWEEN 'Smath' and 'Smzth';

Tumregel 6: Ge optimeraren information.

Optimerare har inte alltid samma möjlighet att dra slutsatser som en människa kan göra (Celko, 1995). Ju mer information som finns i en fråga desto större är chansen att optimeraren har en möjlighet att hitta en förbättrad exekveringsplan. Till exempel för att förena (eng join) tre tabeller på en gemensam kolumn går det att skriva på följande sätt:

Alternativ 1	Alternativ 2
SELECT * From Tabell1, Tabell2, Tabell3 WHERE Tabell2.common = Tabell3.common And Tabell3.common =Tabell1.common;	SELECT * From Tabell1, Tabell2, Tabell3 WHERE Tabell1.common = Tabell2.common And Tabell1.common =Tabell3.common;

En del optimerare kommer att förena par av tabeller baserade på equi-join-villkoret i WHERE-satsen i den ordning som de är skrivna. Antag att Tabell1 är en väldigt liten tabell och att Tabell2 och Tabell3 är stora tabeller. I alternativ 1 kommer då resultatet av att förena Tabell2 och Tabell3 bli en stor mängd som sedan kommer att skäras ner av föreningen av Tabell1 och Tabell3. I alternativ 2 däremot kommer föreningen av Tabell1 och Tabell2 resultera i en liten mängd som sedan matchas med en liten mängd från föreningen mellan Tabell1 och Tabell3.

Det bästa sättet är däremot att förse frågan med all information så att optimeraren själv kan ta besluten om tabellernas storlek förändras, se alternativ 3. Detta leder till redundans i WHERE-satsen.

Alternativ 3
SELECT * FROM Tabell1, Tabell2, Tabell3 WHERE Tabell1.common = Tabell2.common And Tabell2.common =Tabell3.common And Tabell3.common =Tabell1.common;

Tumregel 7: Undvik UNION

UNION görs ofta genom att två resultatmängder skapas och som sedan sorteras och slås ihop (Celko, 1995). Optimeraren arbetar endast med ett SELECT-uttryck eller med en subfråga. För att hämta information om personal som arbetar i Stockholm eller i bor i Göteborg kan en SQL-fråga uttryckas som i alternativ 1 och 2, där alternativ 2 är snabbare.

Alternativ 1	Alternativ 2
SELECT * FROM Personal WHERE arbetar = 'Stockholm' UNION SELECT * FROM Personal WHERE bor = 'Göteborg';	SELECT DISTINCT * FROM Personal WHERE arbetar = 'Stockholm' OR bor = 'Göteborg';

Tumregel 8: Föredra JOIN framför nästlade frågor

En nästlad fråga kan vara svåra att optimera (Celko, 1995). Optimerare försöker att platta till nästlade frågor så att de kan uttryckas som JOIN för att den bästa exekveringen ska kunna undersökas. Betrakta följande databas:

Författare(författarnr, författarnamn);

Titel(titelnr, boknr, år);

TitelFörfattare(författarnr, royalty);

Dessa frågor hittar författare som får mindre än 50% i royalty. Alternativen ger samma resultat, men alternativ 2 är den snabbare versionen.

Alternativ 1	Alternativ 2
SELECT författarnr FROM Författare WHERE författarnr IN (SELECT författarnr FROM TitelFörfattare WHERE royalty < 0.50);	SELECT DISTINCT Författare.författarnr FROM Författare, TitelFörfattare WHERE Författare.författarnr = TitelFörfattare.författarnr AND (royalty < 0.50);

Tumregel 9: Undvik uttryck på kolumner som har ett index.

Om en kolumn finns med i ett matematisktutryck eller ett stränguttryck så kan inte indexet användas av optimeraren (Celko, 1995). Till exempel, givet en tabell över uppdrag och deras start- och slutdatum, för att hitta de uppdrag som tog tre dagar att slutföra under 1994 går det att skriva följande:

Alternativ 1	Alternativ 2
SELECT uppdragnr FROM Uppdrag WHERE (slut - start) = 3 AND start >= '1994-01-01';	SELECT uppdragnr FROM Uppdrag WHERE slut = (start + 3) AND start >= '1994-01-01';

Antag att slutdatum ofta används i andra applikationer i databassystemet i detta exempel och att det därför finns ett index till slutdatum. Då innebär det att alternativ 2 exekveras snabbare än alternativ 1.

Samma princip gäller för kolumner i strängfunktioner och ofta för LIKE-predikat (Celko, 1995). Det kan däremot vara en bra lösning vid små tabeller, då lösningen tvingar dessa tabeller att lagras i primärminnet istället för att sökas igenom av ett index.

Tumregel 10: Undvik sortering.

SELECT DISTINCT, UNION, INTERSECT och EXCEPT satserna kan utföra sorteringen för att ta bort dubbletter i resultatet, undantaget är när ett index existerar som kan användas för att ta bort dubbletter utan sortering (Celko, 1995). GROUP BY använder ofta en sortering för att samla ihop grupper för att sedan utföra aggregatfunktioner och sedan reducera varje grupp till en enda rad. Varje sortering kostar (n*log2(n)) operationer, vilket innebär att det finns mycket tid att spara om dessa satser inte används (Celko, 1995).

Om SELECT DISTINCT innefattar en mängd av nyckelkolumner, då är redan alla rader unika och DISTINCT är överflödigt (Celko, 1995). Det går ofta att ersätta SELECT DISTINCT-satsen en EXIST subfråga, som därmed bryter mot tumregel 8, nämligen att onästlade frågor är att föredra före nästlade frågor. Antag att en fråga ska

hitta studenter vars huvudämne är datavetenskap. Frågan skulle kunna skrivas på två olika sätt, där alternativ 2 är snabbare än alternativ 1.

Alternativ 1	Alternativ 2
SELECT DISTINCT S1.name FROM Student AS S1, Institution AS I1 WHERE S1.inst = I1.inst;	SELECT S1.name FROM Student AS S1, Institution AS I1 WHERE EXISTS (SELECT * FROM Institution AS I1 WHERE S1.inst = I1.inst);

5.1.2 Denormalisering

Denormalisering av tabeller kan göras för att optimera prestanda. Denormalisering kan ske på olika sätt, tex uppdelning av tabeller, sammanslagning av tabeller, tillägg av attribut och lagring av härledbara attribut. Det kallas denormalisering eftersom graden på en tabells normalisering kan bli lägre än den var från början (Teorey, 1999).

Teorey (1999) ger ett exempel på när denormalisering kan användas. Han anger även hur lång tid det tar för att hämta data och uppdatera data för de olika lösningarna. Exemplet ser ut som följande:

Fråga: Hämta alla ordernummer som kunder som är dataingenjörer har beställt.

```
Select o.onr, k.knr, k.titel
From order as o, kund as k
Where k.knr = o.knr and k.titel = dataingenjör;
```

Uppdatera: Lägg till en ny kund, målare, med kundnummer 423378 och kundens ordernummer, 763521601, i databasen.

```
Insert into kund(knr, titel) values ('423378','målare');
Insert into order(onr, knr) values ('763521601','423378');
```

I normaliserad form så består databasen av två tabeller, nämligen kund och order. Kundnummer (knr) är primärnyckel i kundtabellen och ordernummer (onr) är primärnyckel i ordertabellen. För att snabba upp frågan kan tabellerna denormaliseras i forma av att de slås ihop till en tabell, dvs order_kund. Ordernummer (onr) blir primärnyckel och kundnummer och titel är inte några nycklar i tabellen.

Teorey (1999) ger en jämförelse av prestanda för exemplet, fig 6. Tiderna visar att det går fortare att hämta information när schemat är denormaliserat men att det tar längre tid att utföra uppdateringar.

	Normaliserat schema (order och kund)	Denormaliserat schema (order_kund)
Fråga (select)	34,2 sek	4,0 sek
Uppdatering (insert)	5,0 sek	7,2 sek

Fig 6. Prestandajämförelse (Bearbetad från Teorey, 1999, sid 184)

5.1.3 Lagrade procedurer och triggers

Lagrade procedurer och triggers kan förbättra prestanda. Vid användandet av en lagrad procedur så utförs behandlingen av en fråga i servern istället för i klienten. Prestandavinsten ligger i att mängden data som skickas i nätverket minskas. Men denna prestandavinst beror på belastningen av klienten respektive servern. Om det är så att servern utgör en flaskhals i form av att den är hårt belastad så kan en lagrad procedur medföra att server blir hårdare belastad och kan därmed försämra prestandan.

5.2 Testfall

Testfallen består av två lösningsförslag; ett lösningsförslag utan konstruktioner och ett med konstruktioner. Testfallen numreras från 1, 2, osv för att underlätta för vidare dokumentation. Varje testfall delas upp i två lösningsförslag; 1a, 1b, 1c och 1d, 2a, 2b, 2c och 2d osv, där a och c är lösningsförslag utan konstruktioner och där b och d är lösningsförslag med konstruktioner. Testfallen 1, 2 och 3 simulerades i detta arbete.

Testfall 1

Index kan användas för att optimera prestanda, men det finns situationer då index ska undvikas (Roti 1996). Gränsen för när index ska användas eller ej varierar, enligt Roti (1996), och gränsen beror på hårdvaran. Roti (1996) anser att en gräns vid 100 rader i en tabell kan vara en riktpunkt.

I databassystem där information lagras om personer kan primärnyckel för dessa relationer vara tex personnummer eller anställningsnummer. Det är kanske av intresse att ta ut listor med information om dessa personer och att denna lista är sorterad i bokstavsordning istället för nummerordning.

Att använda sig av ett select-uttryck och en order-by-sats för att få fram en sorterad lista är en tidskrävande process (Shasha, 1992). Lösning på detta kan vara att skapa ett sekundärt index på, tex namn. Detta testfall kommer att hämta information från en tabell och sortera informationen. Det attribut som informationen ska sorteras efter kommer inte vara lagrat i ordning i databasen. Detta testfall kommer även att mäta hur lång tid det tar att lägga till rader i tabellen.

1a - Lägga till information i en tabell som ej har ett index kopplat till sig.

1b - Lägga till information i en tabell som har ett index kopplat till ett attribut.

1c - Hämta information om personer sorterade efter namn mha order-by-sats.

1d - Hämta information om personer sorterade efter namn mha sekundärt index.

Testfall 2.

Ett härlett attribut är ett attribut som går att beräkna från data som finns lagrad i databasen. Beräkningar kan ta lång tid att utföra och om det härledda attributet ska beräknas ofta kan det vara en bra ide att beräkna det och sedan lagra attributet i en tabell i databasen. Att lagra det härledda attributet medför att tabellen blir denormaliserad. Detta testfall kommer att innefatta två tabeller, förslagsvis försäljare och order. Testfallet ska räkna hur många order in försäljare har sålt. Anta att detta görs ofta och att det därmed kanske är en bra ide att räkna fram denna siffra och sedan lagra det i tabellen för försäljare. För att försäkra att siffran på antal order är rätt så räknas denna fram med hjälp av en trigger som utlöses så fort en ny order läggs till i order-tabellen.

2a - Uppdatering av information i tabell, ingen trigger i databasen.

2a - Uppdatering av information i tabell, trigger i databasen.

2a - Antal order per försäljare, utan trigger.

2b - Antal order per försäljare, med trigger och lagring av attribut.

Testfall 3

Vid denormalisering av databasen kan prestandavinster erhållas när data ska hämtas från databasen, men denna prestandavinst kostar prestanda vid uppdateringar.

Antag att en användare av ett databassystem ofta vill ta fram information om vilka order som kunder på en viss ort beställt. Låt även säga att om databasen är normaliserad så lagras data om order och kund i separata tabeller, samt att kundnummer även är lagrat i order-tabellen som främmande nyckel. Den denormaliserade formen består av en tabell där tabellerna order och kund är hopslagna till en tabell order_kund. Detta testfall ska mäta tider för både uppdatering av rader och hämtning av information från tabellerna. Detta testfall är modifierat från ett exempel på prestandaoptimering från Teorey (1999)**.**

3a - Lagring av information i normaliserade tabeller.

Information om kund och order läggs till i respektive tabell

3b - Lagring av information denormaliserad tabell.

Information om kund och order läggs till i order_kund-tabellen.

3c - Hämtning av information från normaliserade tabeller.

Information erhålls genom ett select-uttryck med ett join-villkor, dvs en join-fråga.

3d - Hämtning av information denormaliserad tabell.

Ordertabellen och kundtabellen slås ihop till en tabell och informationen erhålls genom ett select-uttryck från denna tabell.

Testfall - 4 Många frågeoptimerare använder inte ett befintligt index om en fråga innehåller ett beräkningsuttryck och detta är formulerat på ett felaktigt sätt (Shasha, 1992). Detta testfall ska jämföra två frågor som hämtar information från en tabell. Attributet, lon, har ett sekundärt index kopplat till sig och detta gäller för båda frågorna. Den första frågan är skriven på ett sätt som göra att frågeoptimeraren ej använder indexet medan den andra frågan är skriven så att frågeoptimeraren utnyttjar indexet.

4a - Frågeoptimerare använder inte befintligt index.

*Select **

From forsaljare

Where lon/12 >= 4000;

4b - Frågeoptimerare använder befintligt index.

*Select **

From forsaljare

*Where lon>= 4000*12;*

Testfall 5

Betrakta följande relationer: forsaljare(fnr, namn, adr, avd,..., lon) och kund(knr, namn, adr, ..., telnr) och situationen att en försäljare även kan vara kund. Antag att tabellen forsaljare innehåller mindre rader än kundtabellen. För att hämta information om vilka försäljare som också är kunder används en join-fråga. Ordningen på tabellerna i FROM-satsen kan avgöra hur frågan exekveras (Shasha, 1992).

5a - Join-fråga med största tabellen först i villkoret

Select forsaljare.namn

From forsaljare, kund

Where forsaljare.namn = kund.namn;

5b - Join-fråga med största tabellen sist i villkoret

Select forsaljare.namn

From kund , forsaljare

Where forsaljare.namn = kund.namn;

Testfall 6.

Istället för att en fråga behandlas i en klient kan lagrade procedurer användas. En lagrad procedur anropas av klienten och exekveringen av frågan sker i servern.

6a - Select-fråga som exekveras i klient.

6b - Select-fråga som exekveras i server.

5.3 Design av benchmarkingverktyg

Oavsett storlek på en implementation så finns det ett antal frågor som ska besvaras och beslut som ska tas, dvs benchmarkingverktyget ska designas. Under designen tas beslut som slutligen kommer att påverka framgången vid implementationen (Pressman, 1997). Utan design är risken stor att verktyget som implementeras får ett felaktigt beteende.

5.3.1 Faktorer

Experiment är en beteckning på en undersökningsuppläggning där enstaka variabler studeras. Patel m fl (1994) skiljer mellan två variabler, nämligen oberoende variabler och beroende variabler. De oberoende variablerna är de som blir manipulerade och de beroende variablerna är de som är beroende av manipulationen. Den beroende variabeln i detta projekt är svarstiden. I detta projekt finnas många faktorer som är påverkbara av projektet och därmed skulle kunna vara oberoende variabler. Några av dessa faktorer kommer agera som oberoende variabler medan de andra kommer att vara konstanter, (se fig. 7).

Oberoende variabler	Konstanter
Antal rader i tabeller	Antal tabeller
Olika lösningsförslag	Antal kolumner i tabeller
Antal användare	Kolumnernas datatyp

Beroende variabel = svarstid

Fig 7. Översikt över variabler och konstanter i projektet.

Prestanda beror på posternas storlek och antal poster i databasen (Elmasri och Navathe, 1994). I detta projekt kommer inte tabellernas storlek att förändras med avseende på antal kolumner, ej eller kommer attributens datatyper att förändras, dvs kolumnernas storlek i byte. Antal tabeller i databasen kommer inte förändras. Däremot så kommer antalet rader i tabellerna att öka successivt och kommer därmed vara en oberoende variabel till den beroende variabeln svarstid. En annan oberoende variabel är simuleringarna av testfall, dessa kommer att består av två olika lösningar som löser samma uppgift.

5.3.2 Reflektion av verkligheten

Data i databassystem är till stor del representerade i form av numeriska tal och textsträngar. Syftet med benchmarkingverktyget och databassystemet i detta projekt är att mäta svarstider. Innehållet i databasen ska reflektera verkligheten i den mån att den innehåller attribut bestående av numeriska tal och textsträngar. Att dessa numeriska tal och textsträngar sedan kan tolkas och ges en meningsfull innebörd anses som mindre viktigt i detta fall. En siffra är det samma som ett tecken i en textsträng. Så om

en textsträng "12345678987654432" lagras i databasen istället "Sven Svensson" så påverkar inte detta svarstiden om någon av textsträngarna ska hämtas från databasen.

Databasen kommer att innehålla redundans i form av tabeller och data. Både normaliserade och denormaliserade tabeller kommer att finnas i databasen. Anledningen till denna redundans är att mätningar på uppdateringar i dessa tabeller ska göras.

5.3.3 Scriptbaserat eller hårdkodat (metadata)

För detta projekt är det främsta syftet inte att skapa ett revolutionerande benchmarkingverktyg utan att mäta svarstider för olika lösningar på fysisk databasdesign. Benchmarkingverktyget ska mäta svarstider, men att det sedan ska kunna användas i andra situationer än för just detta projekt betraktas inte med någon större prioritet. Att koda benchmarkingverktyget i en scriptbaserad form skulle eventuellt medföra att verktyget blev mer flexibelt och att det därmed kanske kunde tänkas användas av andra. Då detta inte är syftet kommer systemet att hårdkodas och därmed bli mindre flexibelt. Att hårdkoda systemet istället för att koda systemet med script har fördelen att det går fortare att implementera. Benchmarkingverktyget kommer att byggas in i databassystemet i form av att tider lagras i tabeller.

5.3.4 Storlek på databasen

Antal rader i tabellerna i databasen kommer att variera. Kapaciteten på den dator som simuleringarna körs på får avgöra hur stora några av tabellerna kommer att bli. Benchmarkingverktyget the Wisconsin benchmark innehöll när den konstruerades från början, enligt Gray (1997), en databas bestående av 10 000 rader. Efter att ha utvecklats så innehåller den nu, enligt Gray (1997), en databas som kan innehålla 100 miljoner rader. Benchmarkingverktyget the Set Query Benchmark innehåller en databas på 1 miljon rader (Gray, 1997). I både Wisconsin och Set Query Benchmark består varje rad av ca 200 bytes. I detta projekt kommer storleken på raderna i tabellerna att vara på ca 200 bytes.

5.4 Simuleringar

Benchmarkingverktyget som simuleringarna av testfallen genomförts i består av en relationsdatabas och applikationer som arbetar mot denna. Databasen är upplagd i Interbase och applikationerna är framtagna i Delphi. Databasen består av två eller tre tabeller, beroende på vilket testfall som simulerats. En av tabellerna i databasen lagrar tider som sedan använts för att ta fram svarstider. Tabellerna som simuleringarna körs mot heter Tab1 och Tab2 medan tabellen som lagrar tider benämns som Tider.

Applikationsgränssnitten, som i Delphi kallas för formulär, är alla uppbyggda enligt en och samma princip (se fig 8). På formuläret finns ett antal komponenter, tex textrutor, knappar, query- och menykomponent. Querykomponenten behövs för hantering av databasen och menykomponenten har använts för att dels avsluta programmet och för att hoppa till andra formulär. För att mäta tid i systemet har en funktion i Delphi kallad Time använts. Time-funktionen tilldelar en variabel ett klockslag bestående av timmar, minuter och sekunder. Dessa klockslag har lagrats i en tabell i databasen och för att kunna särskilja klockslagen åt har ett unikt identifikationsnummer angivits i en textruta på formuläret. För att aktivera frågor och uppdateringar av databasen har knappar använts. För att etablera en koppling mellan applikationerna och databasen kräver Interbase att ett lösenord anges. För att undvika att denna uppkoppling påverkar svarstiderna i simuleringarna har en lösenordsknapp

lagts på formuläret (fig 8). Denna knapp aktiverar funktionen för uppkoppling mot databasen och när funktionen är färdig går det att utföra simuleringarna utan att svarstiderna påverkas.

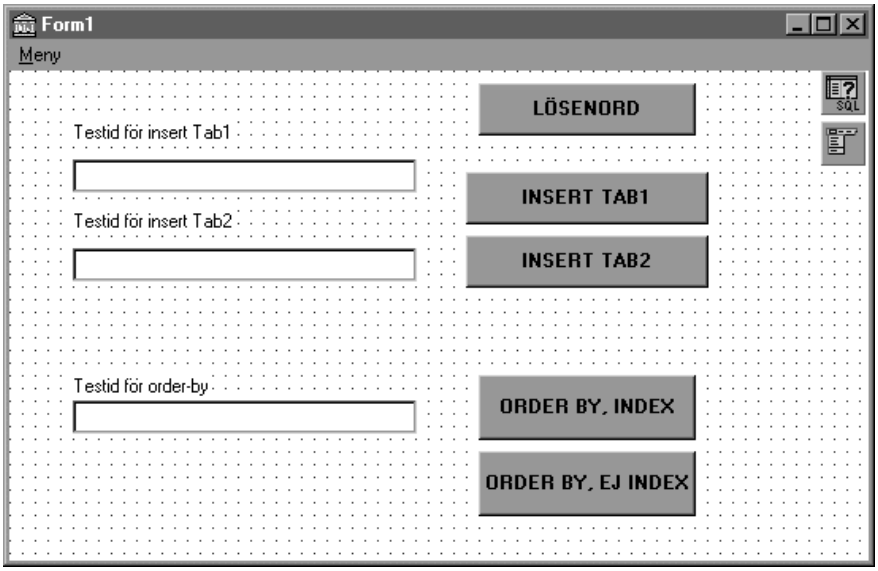


Fig 8. Applikationsgränssnitt.

Innehållet i tabellerna är slumpmässiga värden i textsträngar och heltal. Dessa slumpmässiga värden har tagits fram med hjälp av en procedur i Delphi, dvs randomize och kommandot random. Proceduren randomize har placerats i händelsen form.show för att den ska aktiveras direkt när programmet startar upp. Detta visas kodexempel 1.

Kodexempel 1:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Randomize;
end;
```

Kodexempel 2:

```
function str48t: string;      {slumpar fram 48 st siffror}
    var
      i:integer;
      j:integer;
      s:string;
    begin
        s:=' ';
        for i:=0 to 5 do begin
            j:= Random(99999999);
            s:=s + IntToStr(j);
        end;
        str48t:=s;
end;
```

För att slumpa fram olika heltal och textsträngar av olika längd har funktioner kodats, se kodexempel 2. Exemplet visar en funktion som slumpar fram en textsträng på 48 tecken. Övriga funktioner finns redovisade i Appendix B. Dessa funktioner anropas vid tillfällen då data ska läggas till i en tabell, tex i tabell Tab1.

5.4.1 Testfall 1

För att kunna utföra simulering av testfall 1 skapades en tabell i databasen (kodexempel 3). Denna tabell kan i denna simulering ses som en tabell som innehåller information om personer och den sorterade listan som ska tas fram kan ses som en lista över dessa personer. Låt säga att primärnyckeln i denna tabell är ett personnummer. Listan som tas fram kan sägas vara sorterat efter namn på personerna.

Kodexempel 3:

```
CREATE TABLE Tab2(
        okol1       INTEGER NOT NULL,
        okol2       Char(50)NOT NULL,
        okol3       Char(50)NOT NULL,
        okol4       Char(50)NOT NULL,
        okol5       Char(30)NOT NULL,
        okol6       INTEGER NOT NULL,
        okol7       INTEGER NOT NULL,
        PRIMARY KEY(okol1));
```

För simuleringar som gjordes mot en databas som hade ett sekundärindex kopplat till tabellen lades nedanstående kommandorad till efter kodexempel 3. Kommandoraden skapar ett index vid namn "obinx" och det är kopplat till okol3-kolumnen i Tab2.

```
CREATE ASC INDEX obinx ON Tab2(okol3);
```

Simuleringarna har undersökt hur lång tid det tar att lägga upp olika storlekar på en databas och hur lång tid det tar att hämta en sorterad lista beroende på hur stor databasen är. Först lades en databas upp med tusen rader i en tabell. Inget sekundärt index var kopplat till tabellen i de första simuleringarna. Information till tabellen uppdaterades med hjälp av en loop av insert-satser i Delphi (se kodexempel 4).

Kodexempel 4:

```
procedure TForm1.Button2Click(Sender: TObject);

var
d:integer;
e:string;
f:string;

begin

   e:= TimeToStr(Time);     {starttid för test}

   for d:=1 to 1000 do          {loop för att lägga upp rader i Tab2}

      begin

      query1.sql.clear;
      query1.sql.Add('INSERT INTO
Tab2(okol1,okol2,okol3,okol4,okol5,okol6,okol7) VALUES
(:okol1,:okol2,:okol3,:okol4,:okol5,:okol6,:okol7)');

      query1.ParamByName('okol1').asstring:=strpn;
      query1.ParamByName('okol2').asstring:=str48t;
      query1.ParamByName('okol3').asstring:=str48t;
      query1.ParamByName('okol4').asstring:=str48t;
      query1.ParamByName('okol5').asstring:=str27t;
      query1.ParamByName('okol6').asstring:=str999;
      query1.ParamByName('okol7').asstring:=str999;
      query1.execsql;

   end;

      query1.sql.clear;
      query1.sql.Add('COMMIT');
      query1.execsql;

      f:= TimeToStr(Time);  {sluttid}

      query1.sql.clear;
      query1.sql.Add('INSERT INTO Tider(test,start,slut)
      VALUES   (:test,:start,:slut)'); {tider för simuleringen lagas}

       query1.ParamByName('test').asstring:=Edit2.text;
       query1.ParamByName('start').asstring:=e;
            query1.ParamByName('slut').asstring:=f;
            query1.execsql;

end;
```

Simuleringarna av testfall 1 gick delvis ut på att undersöka om det blev någon prestandavinst vid användandet av ett sekundärindex i jämförelse med att inte använda det. För att hämta en sorterad lista från databasen användes samma kod i båda alternativen (kodexempel 5). Skillnaden mellan alternativen är att ett sekundärindex skapas i databasen och att frågeoptimeraren då använder sig av detta när frågan exekveras.

Kodexempel 5: (komplett kod återfinns i Appendix B)

```
query1.sql.clear;
query1.sql.Add('SELECT okol3 FROM Tab2 order by okol3');
query1.open;
```

Simuleringarna gjordes genom att en databas på 1000 rader först lades upp och en lista från denna hämtades därefter. Efter detta togs databasen bort och en ny databas med 5000 rader lades upp. En ny lista hämtades och därefter togs databasen bort igen. Detta scenario upprepades och den sista databasen som lades upp var på 150 000 rader. Tider för både uppdatering och hämtning av data togs genom att en starttid och en sluttid lagrades i en separat tabell. Dessa tider redovisas här i form av diagram. Exakta tider återfinns i tabeller i Appendix A.

Diagrammet (fig. 9), visar hur många sekunder det tog att lägga upp databasen vid olika storlekar. Linjen "utan index" visar tiden när databasen inte innehåll något sekundärindex och linjen "med index" visar tiden då databasen även uppdaterade ett sekundärindex. Diagrammet visar att det tar längre tid att lägga upp en databas som innehåller sekundärindex än om databasen inte gör det. Att lägga upp en databas på 150 000 rader tog 1689 sekunder när inget sekundärindex fanns, och 5012 sekunder då det fanns ett sekundärindex kopplat till tabellen.

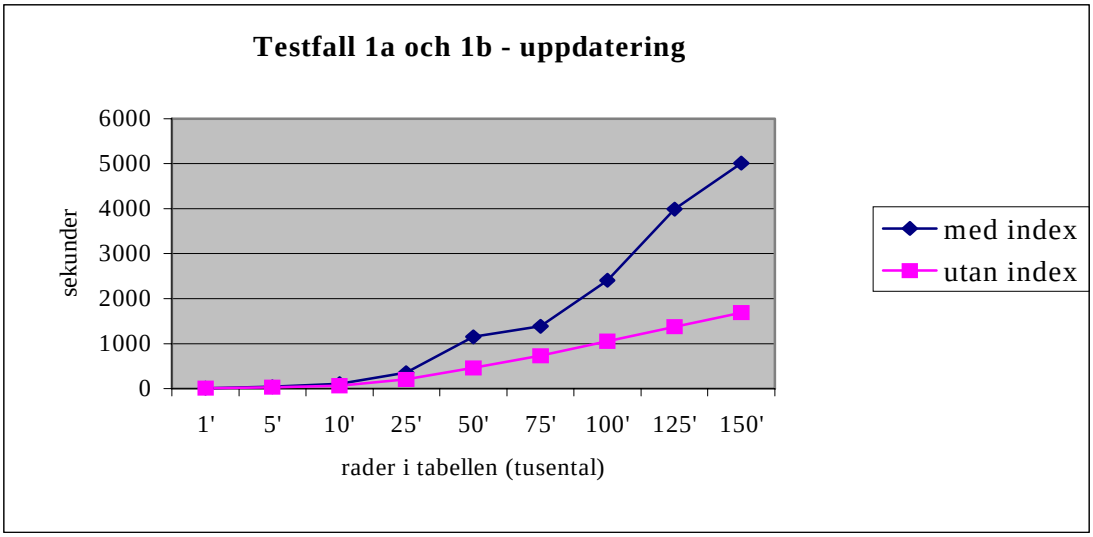


Fig 9. Diagram för testfall 1a och 1b.

Diagrammet (fig. 10), visar tiderna det tog att hämta en sorterad lista vid olika storlekar på databasen. Linjen "utan index" visar tiden det tog att hämta en sorterad lista när frågeoptimeraren inte hade något sekundärindex till sin hjälp. Linjen "med index" visar däremot tiden det tog att hämta sorterad information då frågeoptimeraren hade ett sekundärindex att tillgå. Som diagrammet visar så blev det en väsentlig tidsskillnad då databasen växte i storlek. När databasen bestod av 150 000 rader tog det 35 sekunder att hämta och sortera information då inget index fanns att tillgå. Dessa 35 sekunder kan jämföras med de 4 sekunderna det tog för att hämta samma information med hjälp av ett sekundärindex.

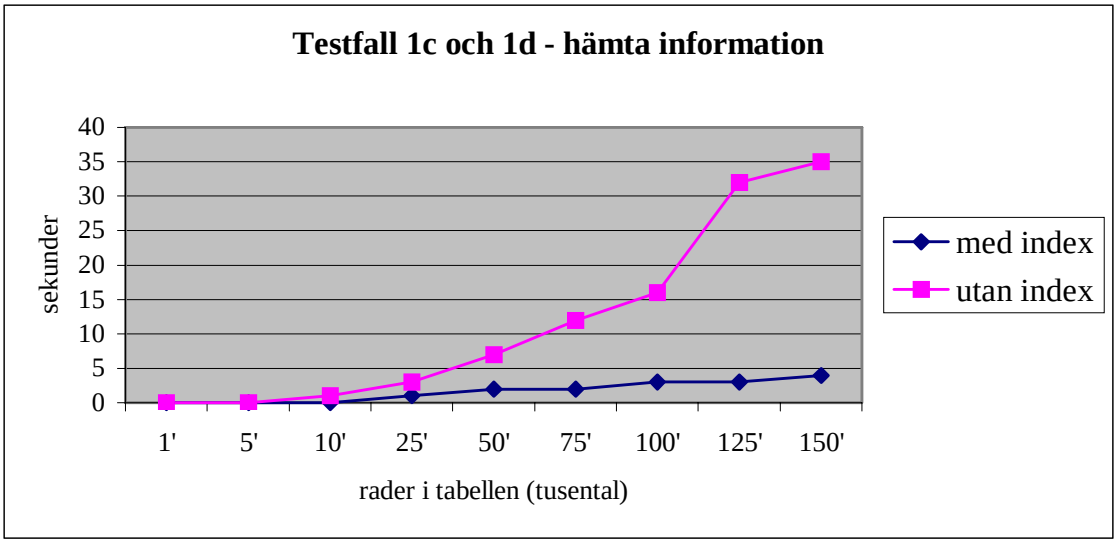


Fig 10. Diagram för testfall 1c och 1d.

5.4.2 Testfall 2

Simuleringarna av testfall 2 gick ut på att undersöka två olika alternativ för att räkna poster i en tabell. Låt säga att simuleringarna ska räkna hur många order som varje försäljare sålt. I detta fall motsvarar då tabellen Tab1 försäljartabellen och Tab2 ordertabellen. Tabellen Tab1 har ett heltal mellan 0-999 som primärnyckel (kodexempel 6). Denna primärnyckel kan sägas vara ett anställningsnummer. Eftersom tabellen, Tab2, har detta anställningsnummer som en främmande nyckel så lades först Tab1 upp med 1000 rader innan några rader tillfördes till Tab2. Tab1 har under simuleringarna alltid varit av storleken 1000 rader medan Tab2 har varierat i storlek upp till 125 000 rader.

Kodexempel 6:

```
CREATE TABLE Tab1(
        kol1        INTEGER NOT NULL,
        kol2        Char(30)NOT NULL,
        kol3        Char(50)NOT NULL,
        kol4        Char(50)NOT NULL,
        kol5        Char(20)NOT NULL,
        kol6        Char(30)NOT NULL,
        kol7        INTEGER NOT NULL,
        kol8        INTEGER NOT NULL,
        PRIMARY KEY(kol1));

CREATE TABLE Tab2(
        okol1       INTEGER NOT NULL,
        okol2       Char(50)NOT NULL,
        okol3       Char(50)NOT NULL,
        okol4       Char(50)NOT NULL,
        okol5       Char(30)NOT NULL,
        okol6       INTEGER NOT NULL,
        okol7        INTEGER NOT NULL,
        PRIMARY KEY(okol1),
        FOREIGN KEY (okol7) REFERENCES Tab1(kol1));
```

Testfallet består av två lösningar där en lösning använder en trigger och en andra lösning som räknar ut "orderantal" varje gång applikationen anropas. Kodexempel 7 är Delphikoden som räknar ut hur många order som varje försäljare har sålt.

Kodexempel 7:

```
query1.sql.clear;
query1.sql.Add('SELECT kol1 , kol2 FROM Tab1');
query1.open;
while (not(query1.eof)) do begin
        Edit2.text:= query1.fieldbyname('kol1').asstring;
        Edit3.text:= query1.fieldbyname('kol2').asstring;
        query2.sql.clear; {Poster räknade sätt värden i editboxarna}
        query2.sql.Add('SELECT count(*)as ant1 FROM Tab2 where
        okol7=:kol1');
        query2.ParamByName('kol1').asstring:=Edit2.text;
        query2.open;
        Edit4.text:= query2.fieldbyname('ant1').asstring;
        query1.next;
end;
```

Denna kod aktiveras genom att klicka på knappen "order/försäljare count" (fig. 11. Vid simuleringar av testfall2 användes förutom gränssnittet i fig. 8, även gränssnittet som visas i fig. 11. För att kunna identifiera tider som lagrats i databasen under simuleringarna skrivs ett unikt identifikationsnummer i textrutan för detta ändamål. Resultatet av frågorna, som aktiveras med hjälp av knapparna, visas upp i de andra textrutorna på formuläret. Koden som finns bakom knappen "Order/försäljare trigger" återfinns i Appendix B och består av en enkel select-sats utan några villkor.

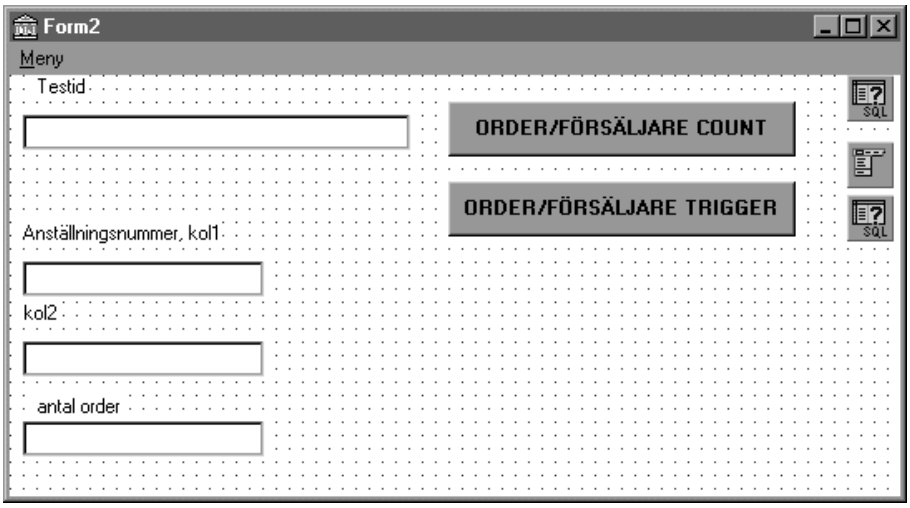


Fig 11. Applikationsgränssnitt för testfall-2

En trigger är en procedur som utlöses när en viss händelse inträffar. I simuleringarna av den lösning som använder en trigger är händelsen den att en orderrad läggs till i ordertabellen, dvs Tab2. I den lösning som använder en trigger har ytterligare ett attribut lagts till i Tab1 (se Appendix C). Detta för att när triggern exekverar och räknar ut antal order för en försäljare så ska resultatet lagras ihop med försäljaren i tabellen Tab1. Kodexempel 8 visar hur triggern har kodats i Interbase. Koden för triggern placeras efter det att tabellerna har skapats (se Appendix C).

Kodexempel 8:

```
SET TERM !!;
CREATE TRIGGER orderant FOR Tab2
AFTER INSERT AS
DECLARE VARIABLE TEMP integer;
BEGIN
    SELECT Count(*)
    FROM tab2
    WHERE NEW.okol7=okol7
    INTO :TEMP;
    UPDATE Tab1
    SET kol9 = :TEMP
    WHERE NEW.okol7=kol1 ;
END!!
SET TERM ;!!
```

Databasen har i simuleringarna för testfall-2 lagts upp successivt från noll rader i Tab2 till 125000 rader. Tider har lagrats för hur lång tid det har tagit att öka databasens storlek från tex 0-1000 rader, 1001-5000 rader osv. Tiderna för att uppdatera databasen är redovisade i figur 12. I simuleringarna som använder en trigger har även tiden för hur lång tid det tog att uppdatera 5 rader i Tab2 då databasen är olika stor. Detta för att detta arbete uppskattar att det är vanligare att uppdatera enstaka rader i en ordertabell än det är att uppdatera tex 25000 rader i taget. Det tog 8 sekunder att uppdatera 5 rader när tabellen innehöll 125000 rader. Detta kan jämföras med 1 sekund som det tog att uppdatera 5 rader i den lösning som inte använde en trigger. Alla tider finns representerade i Appendix A.

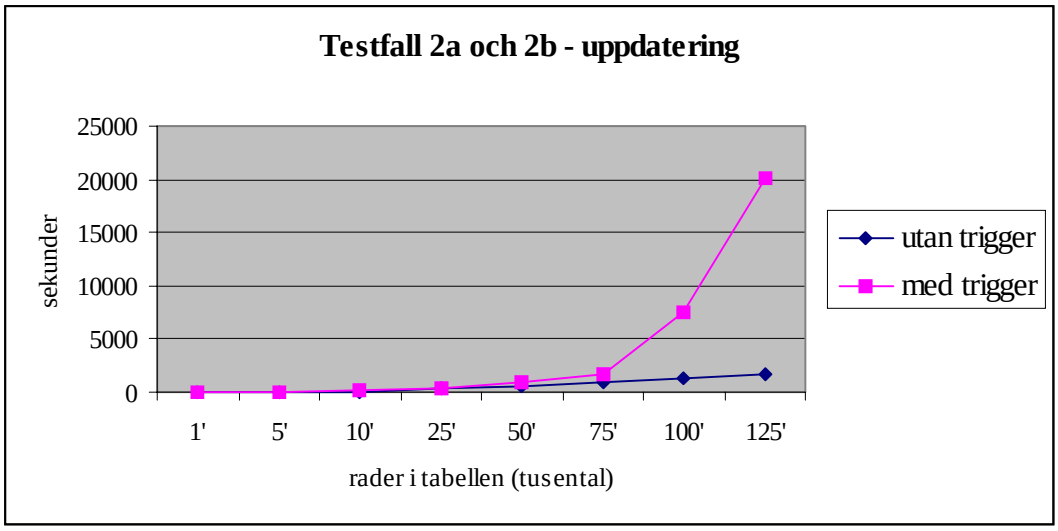


Fig 12. Tider för att uppdatera databasen för testfall-2.

Tider för att exekvera frågorna i testfallet har tagits fram vid olika storlekar på databasen (fig. 13). Simuleringar har gjorts på två olika lösningar, varav en där

exekveringen utförs i klienten med hjälp av count-kommandot. Vid denna lösning måste klienten utföra samma beräkning varje gång applikationen anropas. Den andra lösningen utförs i servern. Denna lösning görs med en trigger som räknar fram "orderantal". Det framräknade resultatet lagras i databasen och ny beräkning utförs endast om "orderantalet" förändras. Vid en databasstorlek av 125 000 rader tog det 588 sekunder att räkna fram antal order när exekveringen utfördes i klienten medan det tog 3 sekunder i den lösning som utnyttjade en trigger.

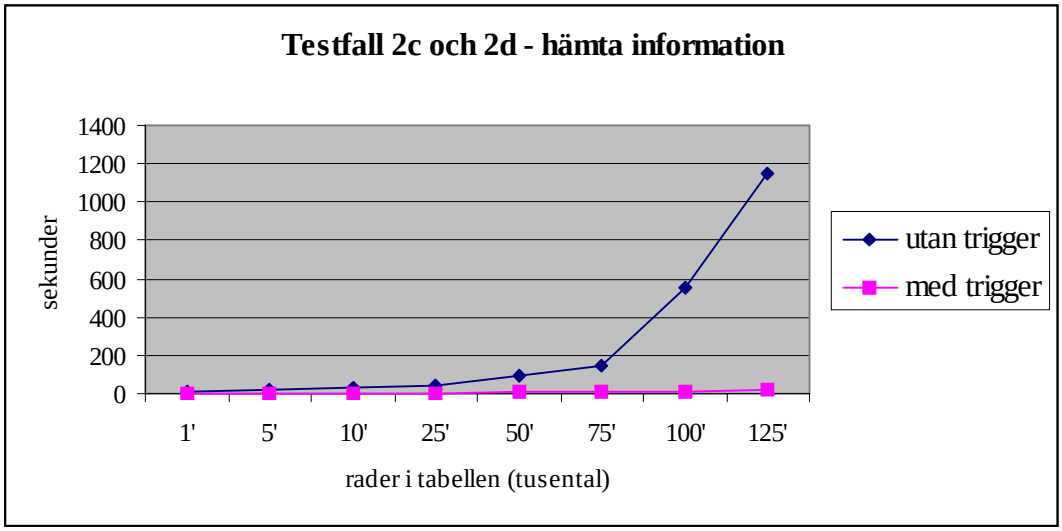


Fig 13. Tider för att exekvera frågor i testfall 2.

5.4.3 Testfall 3

Simuleringarna av testfall3 kan ses som en modifikation av Teoreys (1999) exempel på denormalisering (se punkt 5.1.2). Simuleringarna undersökte om det blev några prestandavinster när två tabeller i en normaliserad databas slogs ihop och därmed gjorde databasen denormaliserad. I detta fall motsvarade tabellen Tab1 kundtabellen och Tab2 motsvarade ordertabellen. I simuleringarna av 3a och 3c var databasen normaliserad och bestod av två tabeller (för kod se Appendix C). Tab1 lades upp med 1 000 rader medan Tab2 har varierat i storlek upp till 125 000 rader. I simuleringarna av 3b och 3d slogs tabellerna ihop till en tabell (kodexempel 9). Denna tabell byggdes upp med rader under simuleringarna till 125 000 rader.

Kodexempel 9:

```
CREATE TABLE Tab2(
    okol1       INTEGER NOT NULL,
    okol2       Char(50)NOT NULL,
    okol3       Char(50)NOT NULL,
    okol4       Char(50)NOT NULL,

    okol5       Char(30)NOT NULL,
    okol6       INTEGER NOT NULL,
    okol7       INTEGER NOT NULL,

    kol2        Char(30)NOT NULL,
    kol3        Char(50)NOT NULL,
    kol4        Char(50)NOT NULL,
    kol5        Char(20)NOT NULL,

    kol6        Char(30)NOT NULL,
    kol7        INTEGER NOT NULL,
    kol8        INTEGER NOT NULL,
    PRIMARY KEY(okol1));
```

Även detta testfall består av två olika lösningar för att hämta aktuell information. Information som ska hämtas är kundnr (okol7) som befinner sig på en viss ort (kol8). Alla ordernr (okol1) som dessa kunder har ska också hämtas. I en normaliserad lösning så lagras denna information i två tabeller och informationen hämtas med hjälp av ett select-uttryck med join-villkor (kodexempel 10). I den denormaliserade lösningen, så lagras information endast i en tabell och informationen hämtas med ett enkelt select-uttryck (kodexempel 11).

Kodexempel 10:

```
query1.sql.clear;
query1.sql.Add('SELECT okol1, kol1 FROM Tab2, Tab1 Where kol8 = 4 and
Tab2.okol7=Tab1.kol1');
query1.open;
```

Kodexempel 11:

```
query1.sql.clear;
query1.sql.Add('SELECT okol1, okol7 FROM Tab2 Where kol8 = 4');
query1.open;
```

Precis som i testfall 2 har databasen i testfall 3 lagts upp successivt från noll rader i Tab2 till 125 000 rader. Tider för hur lång tid det tog att öka databasens storlek har lagrats och är redovisade i figur 14. De exakta tiderna finns redovisade i tabeller i Appendix A. Linjen "normaliserad" visar tiden det tog för att uppdatera information i tabellen Tab2 när databasen var normaliserad. Tiden det tog att uppdatera information i Tab2 när databasen var denormaliserad visas med linjen "denormaliserad".

Diagrammet (fig. 15) redovisar tiderna det tog att hämta aktuell information i simuleringarna av testfall 3c och 3d. Linjen "normaliserad" motsvarar testfall 3c, dvs då databasen är normaliserad, medan linjen "denormaliserad" visar tiden för testfall 3d. Simuleringarna av testfall 3 uppvisade inte några uppenbara prestandavinster.

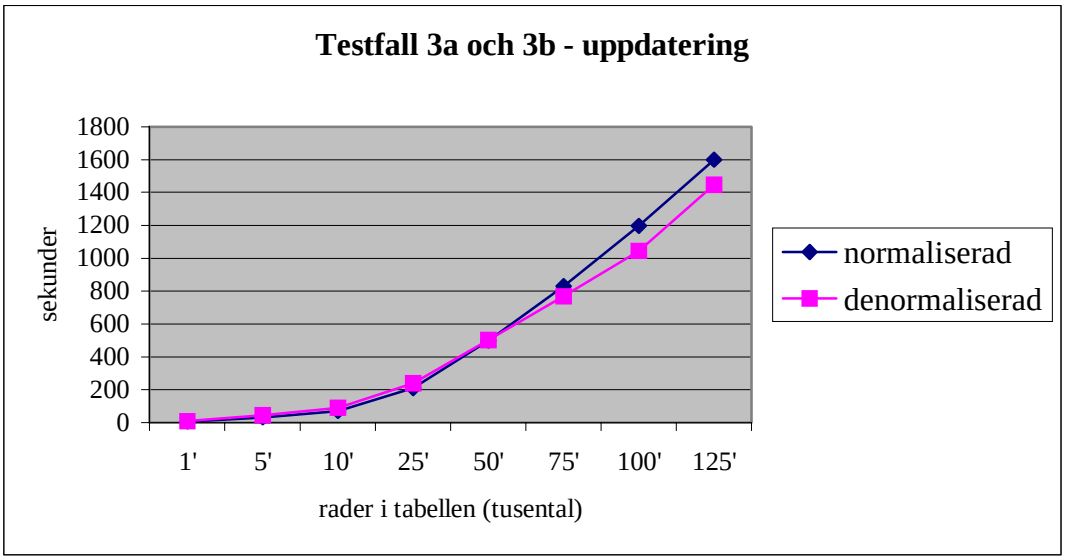


Fig 14. Diagram för testfall 3a och 3b.

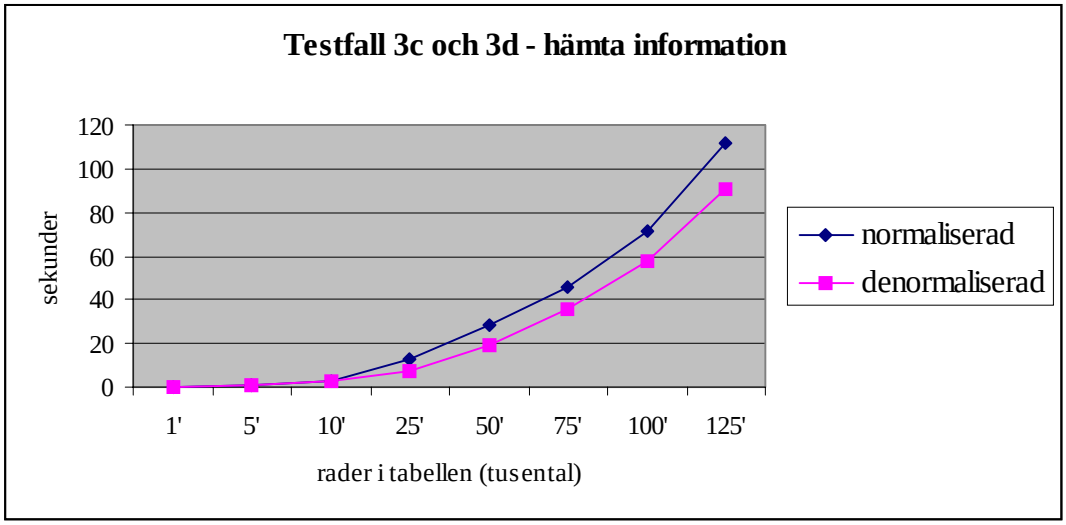


Fig 15. Diagram för testfall 3c och 3d.

6 Slutsats

6.1 Resultat

6.1.1 Strategier

Detta arbete har undersökt om det finns situationer och strategier då det är lämpligt att använda olika operatorer och konstruktioner från SQL i syfte att optimera prestanda. Av den litteratur som har behandlats i detta arbete var det bara en bok, "SQL for smarties" (Celko, 1995), som behandlade problemområdet på ett direkt sätt. Celko (1995) skriver att beroende på hur SQL skrivs så påverkar detta prestanda i ett databassystem. Men han menar också att det inte finns några generella regler för hur SQL ska skrivas för att alla fördelar med alla frågeoptimerare ska kunna utnyttjas. Celko (1995) ger några generella förslag till prestandaoptimering. Han kallar dessa generella förslag för tumregler. Dessa tumregler finns representerade i rapporten. Det har i detta arbete inte påträffats någon annan litteratur som motsäger det Celko (1995) påstår.

Övrig litteratur som hanterats i detta arbete har inte som behandlat problemområdet på samma träffande sätt som Celko (1995) gör. Relevant information som påträffats i annan litteratur behandlas ofta i samband med tuning. Om ett databassystem efter implementation inte uppnår önskvärd prestanda så kan en så kallad tuning utföras. Detta kan innebära att SQL-uttryck skrivs om för att erhålla en snabbare hantering. I litteratur som har behandlat tuning har några förslag till hur SQL-uttryck kan optimeras påträffats. Dessa förslag har använts som underlag till testfallen som simulerats i detta arbete.

6.1.2 Metod

Ett förväntat resultat i detta arbete var att få fram mätningar av svarstider och på så vis utvärdera strategier för prestandaoptimering. Detta resulterade i en metod för att mäta svarstider, ett benchmarkingverktyg. Det hela har gått ut på att två likvärdiga lösningar på en applikation har tagits fram och att dessa lösningar sedan har jämförts ifråga om svarstider. Lösningarna har skilts sig ifrån varandra ifråga om utformningen på själva applikationen och i den fysiska designen på databasen. Applikationerna har utformats i Delphi och databasen har skapats i Interbase. Mätningar av svarstider har gjorts i applikationerna med hjälp av en Time-funktion i Delphi. Denna funktion anger hela sekunder som minsta tidsenhet. Mätningen av svarstider är därför gjorda i hela sekunder. Konsekvensen av detta är att det i flera simuleringar ser ut som att det tagit noll sekunder att exekvera en applikation. En svarstid på noll sekunder innebär att det tagit mer än noll sekunder att exekvera en applikation men det har inte tagit så mycket som en sekund. Metoden har använts för att utvärdera tre olika strategier, dvs användandet av ett sekundärt index, trigger och denormalisering.

6.1.3 Sekundärt index

Den första utvärderingen av en strategi för prestandaoptimering behandlade användandet av ett sekundärindex. Applikationen i denna utvärdering skulle ta fram en sorterad lista. Denna lista skulle vara sorterad på ett attribut som ej var primärnyckel i relationen.

Det förväntade resultatet av denna utvärdering var att det skulle ta längre tid vid uppdatering av rader i en tabell när ett sekundärindex var inblandat jämfört med om

det inte var det. För förfrågningar så förväntades det istället att det skulle bli en prestandavinst om frågeoptimeraren hade ett sekundärindex att tillgå vid exekvering av frågan.

Resultatet av simuleringarna blev som förväntat. Det tog nästan tre gånger så lång tid att uppdatera 150 000 rader i en tabell med ett sekundärindex än vad det tog att uppdatera samma antal rader i en tabell utan index. För att hämta en sorterad lista när databasen innehöll 150 000 rader så gick det ca 8,75 gånger så fort att hämta listan när ett sekundärindex fanns att tillgå.

Resultatet av denna utvärdering understryker det faktum att det är viktigt vid konstruktion av ett databassystem att undersöka frekvensen på uppdateringar och förfrågningar. Om systemet har en hög frekvens på uppdateringar och en låg frekvens på förfrågningar så tillför ett sekundärindex ingen prestandaoptimering.

6.1.4 Trigger

En annan strategi som undersökts i arbetet har varit att använda en trigger för att ta fram efterfrågad information. Lösningarna i detta testfall skiljer sig från varandra till största delen i att i den ena lösningen exekveras frågan i klienten medan i den andra lösningen exekveras frågan i servern. Det är lösningen som använder en trigger som utförs i servern. Själva triggern räknar fram hur många order en viss kund har och sparar sedan det framräknade resultatet i kundtabellen. Triggern aktiveras så fort en ny rad läggs till ordertabellen.

Det förväntade resultatet i dessa lösningar var att det skulle ta längre tid att uppdatera rader i den lösning som använder en trigger jämfört med den lösning som inte gör det. Detta för att ett extra moment utförs för varje rad som läggs till orderraden, nämligen triggern. Resultatet efter simuleringarna visade sig bli som förväntat. Det tog 6 minuter och 38 sekunder att lägga upp raderna 100 001 - 125 000 i den lösning som inte använde en trigger. Detta ska jämföras med lösningen som använde en trigger. För denna lösning tog det 3 timmar, 30 minuter och 23 sekunder, dvs ca 32 gånger så lång tid. Det är en väsentlig skillnad i svarstid. Ett antagande från min sida är att det inte är så vanligt att 125 000 rader läggs till en ordertabell på en gång utan att det är vanligare att registrera en order per gång. För att få en uppfattning om hur prestanda skiljer sig ifråga om enstaka rader mättes svarstider för hur lång tid det tog att lägga upp 5 rader när tabellen innehöll 125 000 rader. Det tog 8 sekunder att uppdatera 5 rader i trigger-lösningen medan det endast tog 1 sekund i lösningen som inte använde sig av triggern.

För att hämta information var det förväntade resultatet att det skulle ta längre tid för den lösning som inte använde en trigger. Detta för att den lösning som inte använder en trigger måste räkna ut antalet order per kund varje gång frågan exekveras. Den lösning som använder triggern har redan räknat ut detta och lagrat resultatet i databasen. Simuleringarna av testfallet visade att resultatet blev som förväntat. När databasen innehöll 125 000 rader tog det 3 sekunder att hämta informationen med trigger-lösningen jämfört med 9 minuter och 48 sekunder som det tog för lösningen utan trigger. Det är 196 gånger så lång tid.

Frekvensen på uppdatering och förfrågningar bör beaktas i val av vilken lösning som passar bäst. Men i detta fall bör även belastningen på klient respektive server undersökas eftersom lösningarna skiljer sig från varandra ifråga var exekveringen sker. Om servern redan är överbelastad och kanske till och med betraktas som en flaskhals då är lösningen med trigger inte en bra idé eftersom den skulle belasta

servern ännu mer. Är däremot klienten hårt belastad så är lösningen med trigger en god idé eftersom den avlastar klienten.

6.1.5 Denormalisering

Denormalisering kan vara en strategi för att optimera prestanda i ett databassystem. Denna strategi har utvärderats i detta arbete. Testfallet gjordes med utgångspunkt från Teoreys (1999, sid 182) exempel med denormalisering i form av hopslagning av tabeller. Vad som går att utläsa av Teoreys exempel är att han har använt två tabeller i sin undersökning. Den ena tabellen, order, verkar innehålla endast ett attribut, dvs ordernr, som är primärnyckel. Den andra tabellen, kund, verkar innehålla två attribut, dvs kundnr och befattning. Dessa två tabeller har denormaliserats till en tabell innehållande de tre attributen.

I detta arbete har utvärderingen gjorts på två tabeller, dvs Tab1 och Tab2. I den normaliserade formen bestod Tab1 av åtta attribut medan Tab2 två bestod av sju attribut. Dessa två tabeller denormaliserades till en tabell som bestod av 14 attribut. Enligt Teoreys (1999) undersökning så skulle denormaliseringen ge en prestandavinst vid förfrågningar och en prestandaförlust vid uppdateringar. Det förväntade resultatet i detta arbete var att svarstiderna skulle bli enligt detta. Resultatet blev emellertid inte riktigt så. Vid uppdatering av databasen till 25 000 rader så gick det fortare att uppdatera tabellen i normaliserad form. Så långt är resultatet som förväntat. När databasen sedan uppdateras i intervallen 25 001 till 100 000 rader så gick det snabbare att uppdatera den denormaliserade formen, vilket inte motsvarade förväntningarna.

6.2 Diskussion

Efter utfört arbete kan det konstateras att det finns information om triggers och lagrade procedurer, hur dessa konstrueras och fördelarna med dessa. Det finns kodexempel på dessa konstruktioner men dessa kodexempel syftar inte till att förbättra prestanda utan till att uppfylla andra egenskaper som tex säkerhet och integritet. Lättare är det däremot att hitta exempel angående index och hur dessa ska användas för att optimera prestanda. En anledning till att information om index förekommer i större utsträckning än triggers och lagrade procedurer kan vara att index är ett mer generellt begrepp och därmed mer standardiserat. En annan anledning kan vara att alla frågeoptimerare inte tillhandahåller hjälpmedel för att konstruera procedurer och triggers.

Det finns standardiserade benchmarkingverktyg att tillgå men dessa är inriktade på att undersöka hur prestanda ändras beroende på tex vilken databashanterare som används. En färdig designad databas medföljer dessa standardiserade benchmarkingverktyg och med hjälp av denna databas undersöks om prestandan blir bättre eller sämre i olika miljöer. Detta arbete hade som syfte att undersöka prestanda på ett annat sätt, dvs hur svarstider förändras när lösningen på den fysiska designen och applikationerna ändras. Olika lösningar för att erhålla samma resultat från en applikation har tagits fram och svarstider för de olika lösningarna har jämförts i detta arbete.

Svarstiderna som har mäts upp i benchmarkingverktyget har varit hela sekunder. Detta arbete hade som syfte att påvisa trenden för skillnad i svarstider och inte några exakta siffror. Om arbetet skulle uppvisa exakta siffror så skulle simuleringarna varit tvungna att utföras betydligt fler gånger än vad som är gjort i detta arbete.

Simuleringarna av testfall 1 och testfall 2 som innefattade sekundärindex respektive använde en trigger gav ett resultat som var förväntat. De resulterade i att det tog längre tid att uppdatera databasen vid de lösningar som använde ett sekundärindex och en trigger. Klyftan mellan svarstiderna blev större när databasen växte. Detsamma gällde för förfrågningar men där tog lösningarna som använde sig av ett sekundärindex och en trigger mindre tid. Databasen i testfall 1 simulerades upp till 150 000 rader medan i testfall 2 så simulerades databasen upp till 125 000 rader. Detta för att datorn som simuleringarna gjordes på inte klarade av att utföra simuleringar på 150 000 rader i testfall 2. Om klyftan mellan svarstider fortsätter att öka när databasen växer ytterligare kan vara intressant att veta. Ökar den eller minskar den för att senare gå ihop igen?

Simuleringen av testfall 3 gav inte alls ett resultat som var förväntat. Till skillnad från de andra testfallen så genomfördes två simuleringar av detta testfall. De andra genomfördes bara en gång då de gav förväntat resultat. Båda simuleringarna av testfall 3 gav ett likartat resultat, vilket detta arbete ställer sig frågande till. Det som detta arbete ställer sig mest frågande till är tiden det tog att uppdatera information. I den denormaliserade lösningen så innehåller tabellen mer information än i den normaliserade lösningen och det borde ta längre tid att uppdatera den denormaliserade lösningen. Så är också fallet, om än marginellt, vid uppdateringar av databasen upp till 25 000 rader. Efter detta och upp till 100 000 rader så tar det mindre tid. Uppdateringarna från 100 001 rader till 125 000 rader tog lika lång tid för båda lösningarna.

Efter att ha simulerat några testfall och undersökt prestandan kan det konstateras att beroende på hur SQL skrivs så påverkar det prestandan i ett databassystem. Det finns inga generella regler för hur SQL ska skrivas eftersom olika frågeoptimerare fungerar på olika sätt. Däremot finns det riktlinjer för hur SQL kan skrivas för att prestandan i databassystemet i alla fall inte ska försämras. Dessa riktlinjer anser detta arbete kan vara bra att ha i bakhuvudet om man som, tex databasdesigner, antingen ska konstruera ett nytt databassystem eller göra en justering i efterhand, en så kallad tuning.

Simuleringarna av testfallen har visat att det som ger vinst vid förfrågningar ger förlust vid uppdateringar. Denna vinst och förlust får vägas mot frekvensen på uppdateringar respektive förfrågningar. Det är därför viktigt att undersöka hur systemet är tänkt att fungera och att försöka uppskatta frekvensen på uppdateringar och förfrågningar innan någon fysik design av databasen och applikationerna sker. Simuleringarna av testfallen har visat att några större prestandavinster inte görs förrän databasen innehåller ett visst antal rader. Simuleringarna av testfall 2, dvs användandet av en trigger, är de simuleringar som tydligt visat en brytpunkt för prestandavinster. Denna brytpunkt sker när tabellen blir större än 75 000 rader. I detta fall kan det sägas att någon optimering av prestanda med hjälp av en trigger inte behöver genomföras så länge som tabellen inte blir större än 75 000 rader. Faktorer som storlek och frekvens påverkar prestanda men det är även viktigt att förstå hur frågeoptimeraren som ska användas fungerar så att man som, tex databasdesigner, kan skriva SQL på ett sätt som förbättrar prestanda.

Resultatet från detta arbete kan användas i verksamheter som använder relationsdatabassystem för tex artikel- och lagerhantering. I sådana verksamheter kan 125 miljoner rader i en databas betraktas som stort och därmed kan storleken på databaser som simulerats i detta arbete betraktas som relevanta. Verksamheter av

större dimension, tex bankverksamheter, använder inte relationsdatabaser som lösning i deras datasystem.

6.3 Fortsatt arbete

Detta arbete kan ses som en startpunkt för ytterligare flera arbeten. I detta arbete har ett antal testfall tagits fram, varav tre stycken har simulerats. Dessutom så gav simuleringarna av testfall 3, dvs denormalisering, inte förväntat resultat. Ett fortsatt arbete skulle kunna göra om simuleringarna av testfall-3 och undersöka om ett liknande resultat erhålls. Om så är fallet så kan arbetet undersöka varför det blir som det blir. De testfall som inte simulerades i detta arbete kan simuleras och eventuella prestandavinster kan undersökas.

Ett antal tumregler för prestandaoptimering har dokumenterats i denna rapport. Dessa tumregler kan ses som riktlinjer för vad som kan göras för att optimera prestanda men inte som några konkreta regler. Detta eftersom de beroende på frågeoptimerare inte alltid ger prestandavinster. Frågeoptimerarna fungerar alla olika och en lösning som ger prestandavinst i en optimerare behöver inte nödvändigtvis göra det i en annan. Så ett annat förslag på fortsatt arbete skulle kunna vara att titta på dessa tumregler och applicera dessa på olika frågeoptimerare och undersöka i vilka optimerare som tumreglerna ger en prestandavinst.

I simuleringarna som gjorts i detta arbete har endast en användare och en server ingått. Ett fortsatt arbete skulle kunna vidareutveckla detta arbete med att i första hand innefatta flera användare som använder systemet samtidigt och utvärdera vad som händer med svarstiderna under dessa förutsättningar. Vidare skulle ett fortsatt arbete kunna undersöka vad som händer med svarstider om databassystemet fördelas på flera servrar.

Referenser

Andersen E.S. (1991) *Systemutveckling - principer, metoder och tekniker*, Studentlitteratur, Lund.

Bell, J. (1993) *Introduktion till forskningsmetodik*, 12:e upplagan, Studentlitteratur, Lund.

Celko, J. (1995) *SQL for smarties: advanced SQL programming*, Morgan Kaufmann, San Francisco

Connolly, T., Begg, C., Strachan, A. (1996) *Database Systems - A Practical Approach to Design, Implementation and Management*, Addison-Wesley, Harlow.

Date, C.J. (1995) *An introduction to Database Systems*, 6:e upplagan, Addison-Wesley, Reading.

Dawson, C.W. (2000) *The essence of computing projects: a student´s guide*, 1:a upplagan, Prentice Hall, Harlow.

Elmasri, R., Navathe S. B. (1994) *Fundamentals of Database Systems*, 2:a upplagan, Benjamin/Cummings, Redwood City.

Fried, K. (1998) Benchmarking OLTP Systems, *DBMS*, Volym 11, Nr 7, sid 59.

Gupta, A., Mumick I.S. (1995) Maintenance of Materialized Views: Problems, Techniques, and Applications, *IEEE Data Engineering Bullentine*, Volym 18, Nr 2.

Gray, J. (1997) *The benchmark handbook*, 2:a upplagan, Morgan Kaufmann, San Francisco.

Holme, I.M., Solvang, B.K. (1996) *Forskningsmetodik: om kvalitativa och kvantitativa metoder*, 3:e upplagan, Studentlitteratur, Lund.

Halloran, T.J., Roth M.A. (1993) Magic mirror on the wall, who´s the fastest Databases of them all?, *Technical Report No AFIT/EN-TR-93-05*, US Air Force Institute of Technology.

Kirkwood, J. (1992) *High Performance Relational Database Design*, Ellis Horwood, New York.

Melton, J., Simon, A.R. (1993) *Understanding the new SQL: a complete guide*, Morgan Kaufman, San Francisco.

North, K. (1994) Understanding Multidatabases APIs and ODBC, *DBMS*, Volym 7, Nr 3, sid 44.

Patel, R., Davidson, B. (1994) *Forskningsmetodikens grunder - Att planera, genomföra och rapportera en undersökning*, 2:a upplagan, Studentlitteratur, Lund.

Pressman, R.S. (1997) *Software engineering: A practitioner´s approach*, 4:e upplagan, McGraw-Hill, New York.

Rennhackkamp, M. (1996a) Trigger Happy - a look at the many implementations of database triggers, *DBMS*, Volym 9, Nr 5, sid 89.

Rennhackkamp, M. (1996b) Performance Tuning, *DBMS*, Volym 9, Nr 11, sid 85.

Rennhackkamp, M. (1996c) Performance Monitoring, *DBMS*, Volym 9, Nr 10, sid 85.

Roti, S. (1996) Indexing and Access mechanisms, *DBMS*, Volym 9, Nr5, sid 65.

Saunders, M., Lewis, P., Thornhill, A. (1997) *Reserch Methods for Business Students,* Pitman, London.

Schumacher, R. (1998) SQL-Optimizer 1.0.4, *DBMS,* Volym 11, Nr 1, sid 30.

Shasha, D.E. (1992) *Database Tuning: a principled approach*, Prentice Hall, New Jersey.

Silberschatz, A., Korth, H. F., Sudarshan, S. (1997) *Database system concepts,* 3:e upplagan, McGraw-Hill, New York.

Teorey, T.J. (1999) *Database modeling & design*, 3:e upplagan, Morgan Kaufmann, San Francisco.

Appendix A

Svarstider.

****

Tider för testfall 1a och 1b. Databasen har uppdaterats med först 1000 rader, tiden för denna uppdatering har tagits och lagrats, därefter har databasen tagits bort för att läggas upp igen med 5000 rader osv.

Antal rader i tabellen	Tid i sekunder, med index	Tid i sekunder, utan index
1 000	7	7
5 000	43	31
10 000	107	67
25 000	360	199
50 000	1148	459
75 000	1385	735
100 000	2412	1055
125 000	3992	1381
150 000	5012	1689

Tider för testfall 1c och 1d. Svarstider för att hämta sorterad lista vid olika storlekar på databasen.

Antal rader i tabellen	Tid i sekunder, med index	Tid i sekunder, utan index
1 000	0	0
5 000	0	0
10 000	0	1
25 000	1	3
50 000	2	7
75 000	2	12
100 000	3	16
125 000	3	32
150 000	4	35

Appendix A

****

Tider för uppdatera Tab2 i testfall 2a och 2b. Först uppdaterades 0 - 1000 rader, tiden för denna uppdatering har tagits och lagrats, därefter uppdaterades raderna 1001-5000, osv. Det tog 7 sekunder för att lägga upp 1000 rader i Tab1.

Antal rader i Tab2	Sekunder, utan trigger	Sekunder, med trigger
0 - 1000	6	10
1001 - 5000	26	46
5001 - 10000	48	168
10001 - 25000	227	208
25001 - 50000	292	568
50001 - 75000	320	752
75001 - 100000	368	5809
100001 - 125000	398	12623

Tider för testfall 2c och 2d. Tid det tog att räkna ut antal order per försäljare.

Antal rader i Tab2	Sekunder, utan trigger	Sekunder, med trigger
1000	8	1
5000	9	1
10000	10	1
25000	13	2
50000	49	2
75000	55	3
100000	413	3
125000	588	3

Tid för att uppdatera 5 rader i Tab2 vid olika storlekar på tabellen och när en trigger aktiveras vid varje uppdatering.

Antal rader i Tab2	
1000	1
5000	1
10000	2
25000	3

50000	3
75000	5
100000	6
125000	8

****

Tider för uppdatera Tab2 i testfall 3a och 3b. Först uppdaterades 0 - 1000 rader, tider lagrades, och sedan uppdaterades ytterligare rader. Det tog 6 sekunder för att lägga upp 1000 rader i Tab1.

Antal rader i tabellen	Tid i sekunder, normaliserad databas	Tid i sekunder, denormaliserad databas
1 000	7	9
5 000	26	36
10 000	38	47
25 000	140	147
50 000	288	265
75 000	331	264
100 000	368	276
125 000	401	401

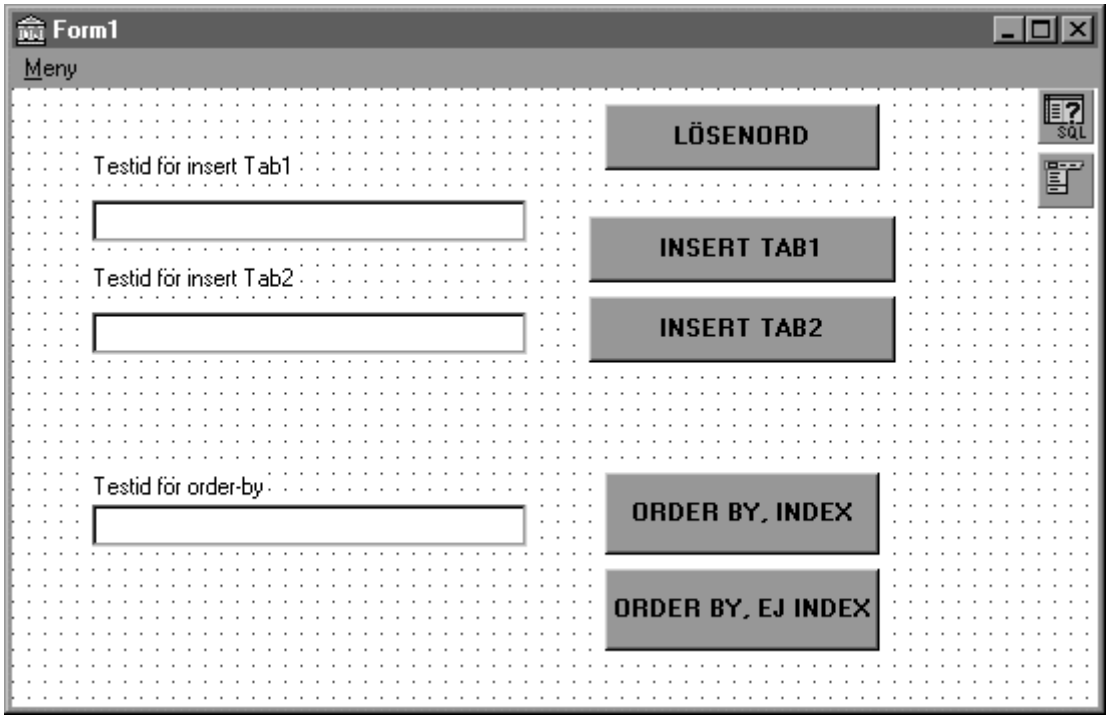
Tider för testfall 3c och 3d. Svarstider för att hämta information.

Antal rader i tabellen	Tid i sekunder, normaliserad databas	Tid i sekunder, denormaliserad databas
1 000	0	0
5 000	1	1
10 000	2	2
25 000	10	4
50 000	15	12
75 000	18	17
100 000	25	22
125 000	41	33

Appendix B

Delhpikod.

Gränssnitt som användes i testfall 1, 2 och 3. Knappen "Insert Tab1" användes inte i testfall 1 och knapparna "order-by" användes inte i testfall-2 och testfall-3.




```
*****************************************************************
*        Testfall  1, 2 och 3                                   *
*****************************************************************
unit exarb;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Db, DBTables, Grids, Menus;
type
  TForm1 = class(TForm)
    Query1: TQuery;
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Button2: TButton;
    Edit2: TEdit;
    Button3: TButton;
    Edit3: TEdit;
    MainMenu1: TMainMenu;
    Meny1: TMenuItem;
    OrderBY1: TMenuItem;
```

```
    Label2: TLabel;
    Label3: TLabel;
    Avsluta1: TMenuItem;
    Button4: TButton;
    Button5: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure OrderBY1Click(Sender: TObject);
    procedure Avsluta1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
  j:integer;
implementation
uses exarb22, exarb3, exarb4;

{$R *.DFM}
function str999: string;    {slumpar fram tal 0-999}
    var
      j:integer;
      s:string;
    begin
        j:= Random(1000);
        s:=IntToStr(j);
        str999:=s;
end;


function strpn: string;    {slumpar fram primärnyckel}
    var
      j:integer;
      s:string;
    begin
        j:= Random(2147483645);
        s:=IntToStr(j);
        strpn:=s;
end;
```

```
function str27t: string;    {slumpar fram 27 st siffror}
    var
      i:integer;
      j:integer;
      s:string;
    begin
        s:='';
        for i:=0 to 2 do
        begin
            j:= Random(999999999);
            s:=s + IntToStr(j);
        end;
        str27t:=s;
end;


function str48t: string;      {slumpar fram 48 st siffror}
    var
      i:integer;
      j:integer;
      s:string;
    begin
        s:='';
        for i:=0 to 5 do
        begin
            j:= Random(99999999);
            s:=s + IntToStr(j);
        end;
        str48t:=s;
end;


procedure TForm1.FormShow(Sender: TObject);
begin
Randomize;
end;


procedure TForm1.OrderBY1Click(Sender: TObject);
begin
Form2.Show;
end;


procedure TForm1.Avsluta1Click(Sender: TObject);
begin
    close;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
{används i testfall 1a, 1b, 2a}

var
d:integer;
e:string;
f:string;

begin

    e:= TimeToStr(Time);     {starttid för test}

    for d:=1 to 5 do

        begin

         query1.sql.clear;
         query1.sql.Add('INSERT INTO
         Tab2(okol1,okol2,okol3,okol4,okol5,okol6,okol7)
         VALUES (:okol1,:okol2,:okol3,:okol4,:okol5,:okol6,:okol7)');
         query1.ParamByName('okol1').asstring:=strpn;
         query1.ParamByName('okol2').asstring:=str48t;
         query1.ParamByName('okol3').asstring:=str48t;
         query1.ParamByName('okol4').asstring:=str48t;

         query1.ParamByName('okol5').asstring:=str27t;
         query1.ParamByName('okol6').asstring:=str999;
         query1.ParamByName('okol7').asstring:=str999;
         query1.execsql;
    end;

        query1.sql.clear;
        query1.sql.Add('COMMIT');
        query1.execsql;
        f:= TimeToStr(Time);  {sluttid}

query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit2.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=e;            {sluttid för test}
        query1.ParamByName('slut').asstring:=f;
        query1.execsql;
end;


**********************************************************************

*        Testfall  1                                                 *

**********************************************************************

procedure TForm1.Button3Click(Sender: TObject); {testfall 1c}
var
g:string;
h:string;
begin
```

```
        g:= TimeToStr(Time);     {starttid för test}

        query1.sql.clear;
        query1.sql.Add('SELECT okol3 FROM Tab2 order by okol3');
        query1.open;
        h:= TimeToStr(Time);  {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
       (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit3.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=g;            {sluttid för test}
        query1.ParamByName('slut').asstring:=h;
        query1.execsql;
end;



procedure TForm1.Button5Click(Sender: TObject); {testfall 1d}
var
g:string;
h:string;
begin
        g:= TimeToStr(Time);     {starttid för test}

        query1.sql.clear;
        query1.sql.Add('SELECT okol4 FROM Tab2 order by okol4');
        query1.open;

        h:= TimeToStr(Time);  {sluttid}
        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit3.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=g;            {tider lagras i databasen}
        query1.ParamByName('slut').asstring:=h;
        query1.execsql;
end;

end.


**********************************************************************

*         Testfall  2                                                  *

**********************************************************************

procedure TForm1.Button1Click(Sender: TObject);
{används i testfall 2a, 2b }

var
a:integer;
b:string;
c:string;
```

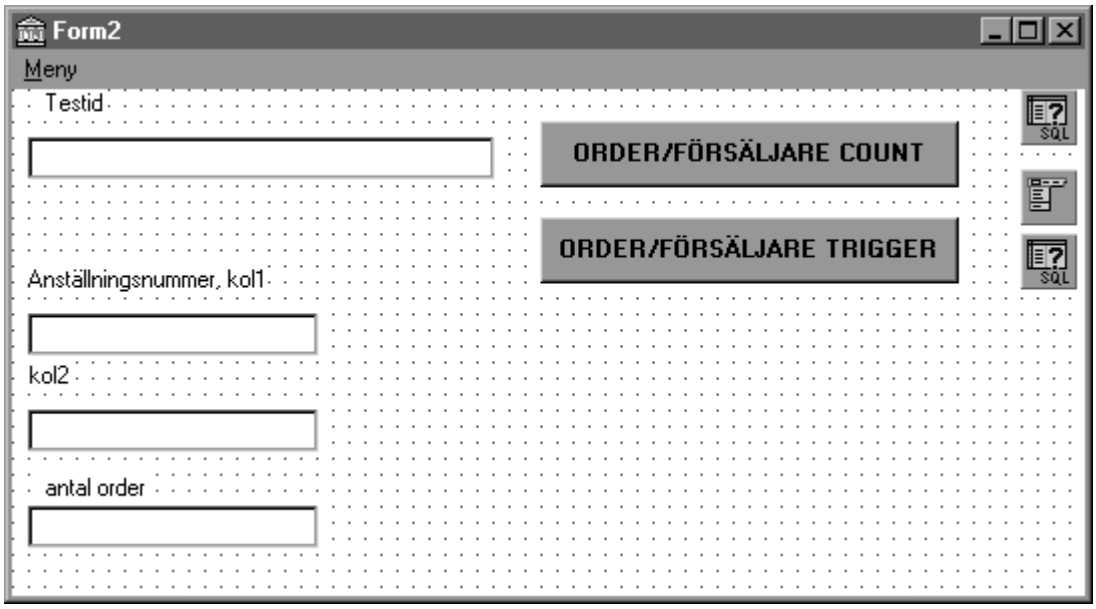
```
begin
    b:= TimeToStr(Time);     {starttid för test}
    for a:=0 to 999 do
        begin
        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tab1(kol1,kol2,kol3,kol4,kol5,kol6,kol7,kol8)
        VALUES (:kol1,:kol2,:kol3,:kol4,:kol5,:kol6,:kol7,:kol8)');
        query1.ParamByName('kol1').asstring:=IntToStr(a);
        query1.ParamByName('kol2').asstring:=str27t;
        query1.ParamByName('kol3').asstring:=str48t;
        query1.ParamByName('kol4').asstring:=str48t;
        query1.ParamByName('kol5').asstring:=TimeToStr(Time);
        query1.ParamByName('kol6').asstring:=str27t;
        query1.ParamByName('kol7').asstring:=str999;
        query1.ParamByName('kol8').asstring:=str999;
        query1.execsql;
    end;
        query1.sql.clear;
        query1.sql.Add('COMMIT');
        query1.execsql;
        c:= TimeToStr(Time);    {sluttid}
        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit1.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=b;            {sluttid för test}
        query1.ParamByName('slut').asstring:=c;
        query1.execsql;
end;
```

****

* Testfall - 2c och 2 d *

****

I testfall 2 användes förutom gränssnittet Form1 nedanstående gränssnitt, Form2.



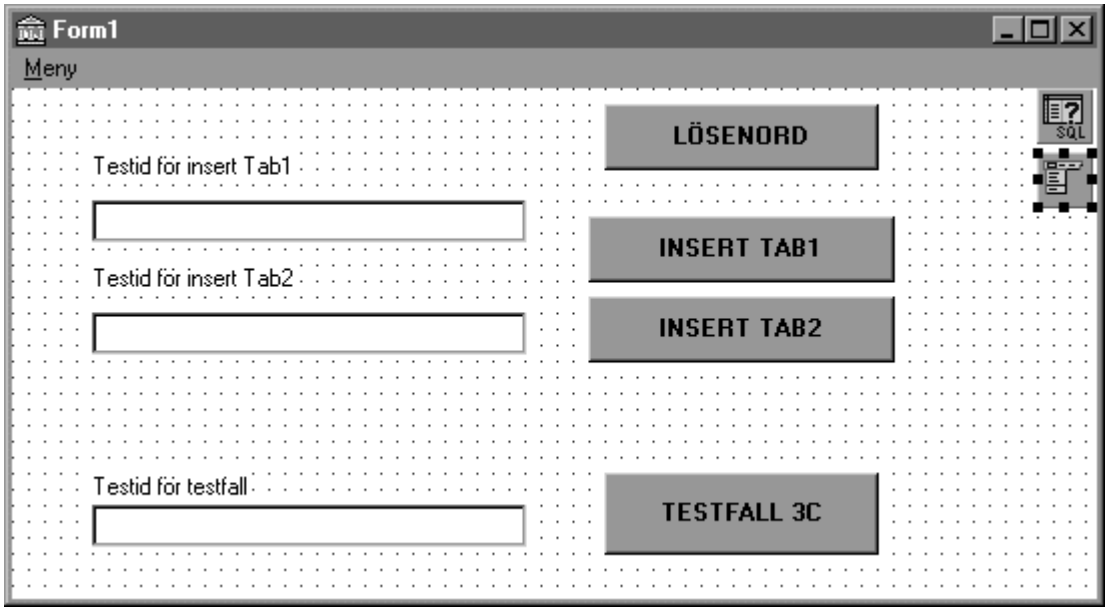

```
procedure TForm2.Button1Click(Sender: TObject);
var
g:string;
h:string;
begin
        g:= TimeToStr(Time);     {starttid för test}
        query1.sql.clear;
        query1.sql.Add('SELECT kol1 , kol2 FROM Tab1');
        query1.open;
        while (not(query1.eof)) do begin
                Edit2.text:= query1.fieldbyname('kol1').asstring;
                Edit3.text:= query1.fieldbyname('kol2').asstring;
                query2.sql.clear; {Poster räknade sätt värden i editboxarna}
                query2.sql.Add('SELECT count(*)as ant1 FROM Tab2
                where okol7=:kol1');
                query2.ParamByName('kol1').asstring:=Edit2.text;
                query2.open;
                Edit4.text:= query2.fieldbyname('ant1').asstring;
                query1.next;
        end;
        h:= TimeToStr(Time);  {sluttid}
```

```
        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit1.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=g;           {sluttid för test}
        query1.ParamByName('slut').asstring:=h;
        query1.execsql;
end;


procedure TForm2.Avsluta1Click(Sender: TObject);
begin
close;
end;


procedure TForm2.Button2Click(Sender: TObject);
var
g:string;
h:string;
begin
      g:= TimeToStr(Time);     {starttid för test}
      query1.sql.clear;
      query1.sql.Add('SELECT kol1,kol2,kol9 FROM Tab1');
      query1.open;
      while (not(query1.eof)) do begin
          Edit2.text:= query1.fieldbyname('kol1').asstring;
          Edit3.text:= query1.fieldbyname('kol2').asstring;
          Edit4.text:= query1.fieldbyname('kol9').asstring;
          query1.next;
       end;
       h:= TimeToStr(Time);  {sluttid}
      query1.sql.clear;
      query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
      (:test,:start,:slut)');
      query1.ParamByName('test').asstring:=Edit1.text;    {testnamn, primärnyckel}
      query1.ParamByName('start').asstring:=g;           {sluttid för test}
      query1.ParamByName('slut').asstring:=h;
      query1.execsql;
end;
end.
```

****

* Testfall - 3b , 3c och 3d *

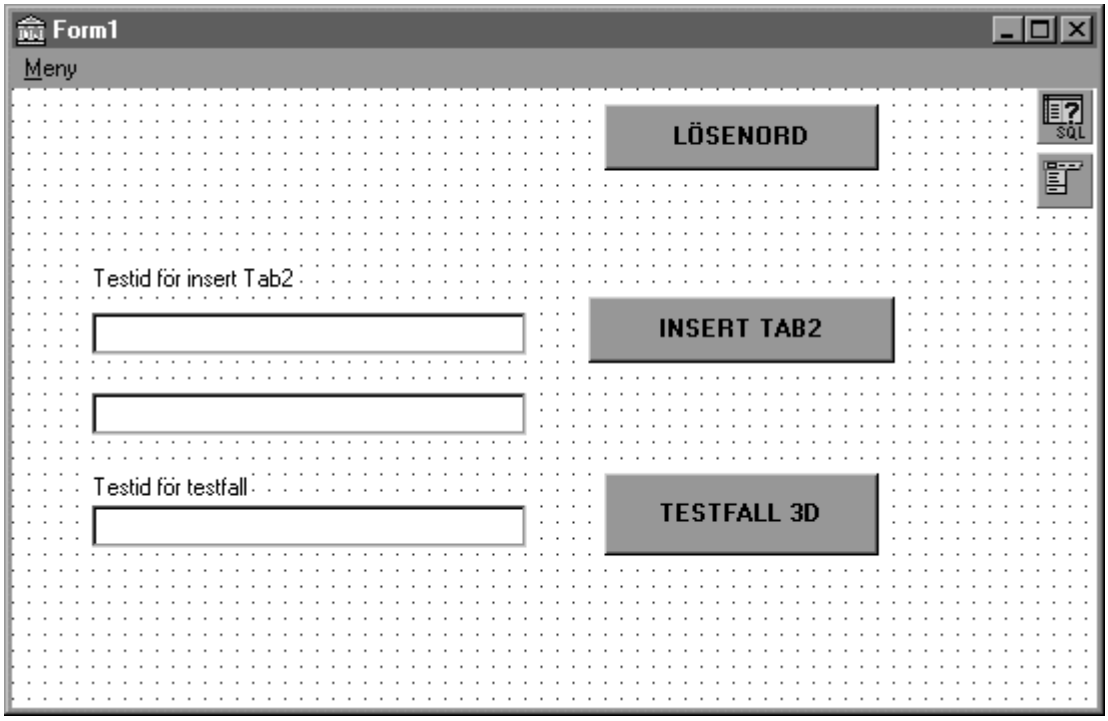
****




```
procedure TForm1.Button1Click(Sender: TObject);

{testfall 3a}
var
a:integer;
b:string;
c:string;

begin
    b:= TimeToStr(Time);     {starttid för test}

    for a:=0 to 999 do

        begin

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tab1(kol1,kol2,kol3,kol4,kol5,kol6,kol7,kol8)
        VALUES (:kol1,:kol2,:kol3,:kol4,:kol5,:kol6,:kol7,:kol8)');

        query1.ParamByName('kol1').asstring:=IntToStr(a);
        query1.ParamByName('kol2').asstring:=str27t;
        query1.ParamByName('kol3').asstring:=str48t;
        query1.ParamByName('kol4').asstring:=str48t;

        query1.ParamByName('kol5').asstring:=TimeToStr(Time);
        query1.ParamByName('kol6').asstring:=str27t;
        query1.ParamByName('kol7').asstring:=str999;
        query1.ParamByName('kol8').asstring:=str5t;
        query1.execsql;

    end;
```

```
        query1.sql.clear;
        query1.sql.Add('COMMIT');
        query1.execsql;

        c:= TimeToStr(Time);    {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');

        query1.ParamByName('test').asstring:=Edit1.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=b;            {sluttid för test}
        query1.ParamByName('slut').asstring:=c;
        query1.execsql;
end;


procedure TForm1.Button2Click(Sender: TObject);
{testfall 3a}
var
d:integer;
e:string;
f:string;
begin
    e:= TimeToStr(Time);     {starttid för test}
    for d:=1 to 25000 do
        begin
        query1.sql.clear;
        query1.sql.Add('INSERT INTO
        Tab2(okol1,okol2,okol3,okol4,okol5,okol6,okol7) VALUES
        (:okol1,:okol2,:okol3,:okol4,:okol5,:okol6,:okol7)');

        query1.ParamByName('okol1').asstring:=strpn;
        query1.ParamByName('okol2').asstring:=str48t;
        query1.ParamByName('okol3').asstring:=str48t;
        query1.ParamByName('okol4').asstring:=str48t;

        query1.ParamByName('okol5').asstring:=str27t;
        query1.ParamByName('okol6').asstring:=str999;
        query1.ParamByName('okol7').asstring:=str999;
        query1.execsql;
    end;
        query1.sql.clear;
        query1.sql.Add('COMMIT');
        query1.execsql;

        f:= TimeToStr(Time);  {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
```

```
        query1.ParamByName('test').asstring:=Edit2.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=e;            {sluttid för test}
        query1.ParamByName('slut').asstring:=f;

        query1.execsql;
end;


procedure TForm1.Button3Click(Sender: TObject);

{testfall 3c}

var
g:string;
h:string;

begin

        g:= TimeToStr(Time);     {starttid för test}

        query1.sql.clear;
        query1.sql.Add('SELECT okol1, kol1 FROM Tab2, Tab1 Where kol8 = 4 and
        Tab2.okol7=Tab1.kol1');
        query1.open;

        while (not(query1.eof)) do begin

            Edit1.text:= query1.fieldbyname('okol1').asstring;
            Edit2.text:= query1.fieldbyname('kol1').asstring;
            query1.next;

         end;

        h:= TimeToStr(Time);  {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
         (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit3.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=g;            {sluttid för test}
        query1.ParamByName('slut').asstring:=h;
        query1.execsql;
end;
```




```
procedure TForm1.Button2Click(Sender: TObject);

{testfall 3b}
var
d:integer;
e:string;
f:string;

begin
    e:= TimeToStr(Time);     {starttid för test}
    for d:=1 to 25000 do
        begin
        query1.sql.clear;
        query1.sql.Add('INSERT INTO
        Tab2(okol1,okol2,okol3,okol4,okol5,okol6,okol7,
        kol2,kol3,kol4,kol5,kol6,kol7,kol8) VALUES
        (:okol1,:okol2,:okol3,:okol4,:okol5,:okol6,:okol7,
        :kol2,:kol3,:kol4,:kol5,:kol6,:kol7,:kol8)');

        query1.ParamByName('okol1').asstring:=strpn;
        query1.ParamByName('okol2').asstring:=str48t;
        query1.ParamByName('okol3').asstring:=str48t;
        query1.ParamByName('okol4').asstring:=str48t;

        query1.ParamByName('okol5').asstring:=str27t;
        query1.ParamByName('okol6').asstring:=str999;
        query1.ParamByName('okol7').asstring:=str999;
        query1.ParamByName('kol2').asstring:=str27t;

        query1.ParamByName('kol3').asstring:=str48t;
        query1.ParamByName('kol4').asstring:=str48t;
```

```
        query1.ParamByName('kol5').asstring:=TimeToStr(Time);
        query1.ParamByName('kol6').asstring:=str27t;

        query1.ParamByName('kol7').asstring:=str999;
        query1.ParamByName('kol8').asstring:=str5t;
        query1.execsql;
    end;

        query1.sql.clear;
        query1.sql.Add('COMMIT');
        query1.execsql;
        f:= TimeToStr(Time);  {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
        (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit2.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=e;           {sluttid för test}
        query1.ParamByName('slut').asstring:=f;
        query1.execsql;

end;


procedure TForm1.Button3Click(Sender: TObject);

{testfall 3d}
var
g:string;
h:string;

begin

        g:= TimeToStr(Time);     {starttid för test}

        query1.sql.clear;
        query1.sql.Add('SELECT okol1, okol7 FROM Tab2 Where kol8 = 4');
        query1.open;

        while (not(query1.eof)) do begin

            Edit1.text:= query1.fieldbyname('okol1').asstring;
            Edit2.text:= query1.fieldbyname('okol7').asstring;
            query1.next;

         end;


        h:= TimeToStr(Time);  {sluttid}

        query1.sql.clear;
        query1.sql.Add('INSERT INTO Tider(test,start,slut) VALUES
         (:test,:start,:slut)');
        query1.ParamByName('test').asstring:=Edit3.text;    {testnamn, primärnyckel}
        query1.ParamByName('start').asstring:=g;           {sluttid för test}
        query1.ParamByName('slut').asstring:=h;
        query1.execsql;
end;
```

Appendix C

Interbasekod

```
**********************************************************************
*       Tabellen som lagrar tiderna i simuleringarna. Ser likadan ut i alla testfall.*
**********************************************************************
CREATE TABLE Tider(
        Test            Char(20) NOT NULL,
        start           Char(20)NOT NULL,
        slut            Char(20) NOT NULL,
        PRIMARY KEY(test));


**********************************************************************
*       Testfall 1                                                    .*
**********************************************************************
CREATE TABLE Tab2(
        okol1           INTEGER NOT NULL,
        okol2           Char(50)NOT NULL,
        okol3           Char(50)NOT NULL,
        okol4           Char(50)NOT NULL,
        okol5           Char(30)NOT NULL,
        okol6           INTEGER NOT NULL,
        okol7           INTEGER NOT NULL,
        PRIMARY KEY(okol1));


**********************************************************************
*       Kommandoraden för att skapa ett sekundärindex. Testfall 1b, 1d.         *
**********************************************************************
CREATE ASC INDEX obinx ON Tab2(okol3);


**********************************************************************
*       Tabellerna för testfall 2a.                                             *
**********************************************************************
CREATE TABLE Tab1(
        kol1        INTEGER NOT NULL,
        kol2        Char(30)NOT NULL,
        kol3            Char(50)NOT NULL,
        kol4            Char(50)NOT NULL,
        kol5            Char(20)NOT NULL,
        kol6            Char(30)NOT NULL,
```

```
        kol7        INTEGER NOT NULL,
        kol8        INTEGER NOT NULL,
        PRIMARY KEY(kol1));


CREATE TABLE Tab2(

        okol1       INTEGER NOT NULL,
        okol2       Char(50)NOT NULL,
        okol3       Char(50)NOT NULL,
        okol4       Char(50)NOT NULL,

        okol5       Char(30)NOT NULL,
        okol6       INTEGER NOT NULL,
        okol7       INTEGER NOT NULL,

        PRIMARY KEY(okol1),
        FOREIGN KEY (okol7) REFERENCES Tab1(kol1));


**********************************************************************

*   Tabellen Tab1 utökandes med ett attribut i testfall 2b. Tab2 oförändrad      *

**********************************************************************

CREATE TABLE Tab1(

        kol1        INTEGER NOT NULL,
        kol2        Char(30)NOT NULL,
        kol3        Char(50)NOT NULL,
        kol4        Char(50)NOT NULL,

        kol5        Char(20)NOT NULL,
        kol6        Char(30)NOT NULL,
        kol7        INTEGER NOT NULL,
        kol8        INTEGER NOT NULL,
        kol9        integer,

        PRIMARY KEY(kol1));


**********************************************************************

*       Trigger i testfall 2b. Räknar ut antal order som är kopplade till en        *

*       försäljare och sparar sedan resultatet i Tab1                               *

**********************************************************************

SET TERM !!;

CREATE TRIGGER orderant FOR Tab2
AFTER INSERT AS
DECLARE VARIABLE TEMP integer;
BEGIN

        SELECT Count(*)
        FROM tab2
```

```
        WHERE NEW.okol7=okol7
        INTO :TEMP;

        UPDATE Tab1
        SET kol9 = :TEMP
        WHERE NEW.okol7=kol1 ;

END!!

SET TERM ;!!


**********************************************************************

*       Tabellerna för testfall 3a och 3c.                            *

**********************************************************************

CREATE TABLE Tab1(

        kol1        INTEGER NOT NULL,
        kol2        Char(30)NOT NULL,
        kol3            Char(50)NOT NULL,
        kol4            Char(50)NOT NULL,

        kol5            Char(20)NOT NULL,
        kol6            Char(30)NOT NULL,
        kol7        INTEGER NOT NULL,
        kol8        INTEGER NOT NULL,
        PRIMARY KEY(kol1));


CREATE TABLE Tab2(

        okol1           INTEGER NOT NULL,
        okol2           Char(50)NOT NULL,
        okol3           Char(50)NOT NULL,
        okol4           Char(50)NOT NULL,

        okol5           Char(30)NOT NULL,
        okol6           INTEGER NOT NULL,
        okol7           INTEGER NOT NULL,

        PRIMARY KEY(okol1),
        FOREIGN KEY (okol7) REFERENCES Tab1(kol1));


**********************************************************************

*   Tabellerna Tab1 och Tab2 slogs ihop till Tab2 i testfall 3b och 3d.  *

**********************************************************************

CREATE TABLE Tab2(

        okol1           INTEGER NOT NULL,
        okol2           Char(50)NOT NULL,
        okol3           Char(50)NOT NULL,
        okol4           Char(50)NOT NULL,
```

```
okol5          Char(30)NOT NULL,
okol6          INTEGER NOT NULL,
okol7          INTEGER NOT NULL,

kol2           Char(30)NOT NULL,
kol3           Char(50)NOT NULL,
kol4           Char(50)NOT NULL,
kol5           Char(20)NOT NULL,

kol6           Char(30)NOT NULL,
kol7           INTEGER NOT NULL,
kol8           INTEGER NOT NULL,
   PRIMARY KEY(okol1));
```