Versionstransparens i evolutionära relationsdatabaser

(HS-IDA-EA-03-101)

Björn Grehag (a99bjogr@student.his.se)

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

Examensarbete på programmet för systemprogrammering under vårterminen 2003.

Handledare: Louise Eriksson

Versionstransparens i evolutionära relationsdatabaser

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

2003-06-05

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: _____

Versionstransparens i evolutionära relationsdatabaser

Björn Grehag (a99bjogr@student.his.se)

Sammanfattning

Många databassystem utsätts ideligen för förändring. Dessa förändringar påverkar databasens schema. Detta har lett till att stöd för dessa förändringar har utvecklas. Schema versioning är det stödet för förändringar av databasscheman som ger det mest omfattande stödet. Med schema versioning menas att DBMS:et kan hantera flera versioner av databasschemat. Ett problem med schema versioning är att användaren måste veta vilken version en viss fråga ska ställas emot för att svaret ska bli korrekt. I detta arbete har det undersökts för vilka typer av förändringar mot en relationsdatabas som detta problem kan undvikas genom att uppnå versionstransparens, vilket innebär att versionshanteringen är osynlig för användaren. Undersökningen gjordes mot en modell som bara använder information ifrån de frågor som ställs mot databasen för att hitta rätt version att ställa frågan emot. Resultatet av undersökningen visar att versionstransparens kan uppnås för alla typer av förändringar utom för ändringar av datatypen för kolumner.

Nyckelord: Databaser, Schemaevolution, Schema versioning, Versionshantering

Innehållsförteckning

1 Introduktion

Vi lever i en värld som ständigt förändrar sig. Företag omorganiseras och effektiviseras, tekniken förbättras och ger nya möjligheter. Dessa förändringar påverkar även de datorsystem som vi använder oss av. De Castro, Grandi och Scalas (1997, s. 249) skriver att: ”Raw data, database structures and applications are evolving entities which require adequate support of past, present and even future versions.”. Det De Castro, et al. vill påpeka är att man måste ta hänsyn till mer än bara de nuvarande kraven på en applikation. Man bör även ta hänsyn till hur kraven kommer att förändras i framtiden eftersom det troligt att kraven på en applikation kommer att ändras under dess livstid (Monk, 1993). För att underlätta dessa förändringar så måste hänsyn tas till detta när applikationen utvecklas.

I detta arbete kommer fokus att ligga på databassystem. I dessa ändras ofta kraven för vad för information som ska lagras i databasen när kraven på applikationerna som använder databasen ändras (Elamsri & Navathe, 2000). Att uppdatera och ominstallera ett databassystem när kraven ändras kan bli kostsamt, eftersom detta arbete kan ta lång tid att genom föra och kan innebära att systemet behöver tas ur bruk under en period. För att undvika dessa kostnader så kan ändringarna utföras under tiden som systemet är igång. Detta skulle innebära att systemet inte behöver tas ur bruk och problemet med att överföra all information från den gamla versionen av systemet till den nya skulle kunna undvikas. Databassystem som kan hantera dessa förändringar under tiden som databassystemet är igång kommer att kallas evolutionära databaser i denna rapport. Peters och Özsu (1997) ger följande exempel på databassystem som behöver vara evolutionära:

- CASE-system
- CAD/CAM-system
- Multimediasystem
- Geografiska informationssystem

Enligt Peters och Özsu (1997) har dessa databassystem gemensamt att de förändras ofta och oregelbundet. I till exempel geografiska informationssystem måste databasen förändras när någon ny typ av geografisk data skall lagras, så som olika typer av kartor. Samma gäller för multimediasystem eftersom dessa lagrar många olika typer av data och nya typer av data tillkommer oftare i dessa system än för mer traditionella databassystem.

Objektorienterade databaser har kommit längre inom evolution än vad relationsdatabaser har gjort eftersom dessa ofta används till applikationer som utsätts för förändringar ofta, så som exemplen som ges ovan av Peters och Özsu. Relationsdatabaser används däremot i större utsträckning idag än objektorienterade databaser. Många stora affärssystem använder också relationsdatabaser eftersom dessa är utvecklade för att uppfylla affärsverksamheters krav på datalagring. Därför kommer detta arbete att inrikta sig på relationsdatabaser.

Evolutionära databaser kan grupperas i ett antal olika kategorier beroende på hur väl förändringar i databasen hanteras. Den kategori av evolutionära databaser som det för tillfället läggs mest forskning på och som erbjuder mest stöd för förändringar i databasen är ”schema versioning” (De Castro, et al., 1997). Schema versioning

innebär att både nuvarande och gamla strukturen på databasen sparas som olika versioner av databasen, det vill säga när en tabell eller klass ändras så skapas en ny version av tabellen eller klassen. Problemet med detta är att man måste veta vilken version en viss fråga ska ställas emot. Det är detta problem som kommer behandlas vidare i detta arbete. Syftet med arbetet är att utreda om det är möjligt att dölja denna hantering av versioner för databasapplikationer, det vill säga en applikation som använder databasen, genom att låta databashanteringssystemet ta hand om all versionshantering.

2 Bakgrund

Många datorsystem i vårt samhälle har till uppgift att lagra, tillhandahålla och bearbeta information. Dessa system använder ofta databaser eftersom dessa ger bra stöd för lagring och bearbetning av information. All lagring och manipulering av data i en databas sker via ett databashanteringssystem (DBMS). Ett DBMS är en samling program som tar hand om uppgiften att skapa, manipulera och underhålla en databas åt de applikationer som använder databasen (Elmasri & Navathe, 2000). Det finns två typer av databaser som används i större utsträckning idag: relationsdatabaser och objektorienterade databaser.

2.1 Relationsdatabaser

Relationsdatabaser är framtagna för att uppfylla kraven på lagring och bearbetning av information för affärsverksamheter. Anledningen var att det fanns ett behov av ett standardiserat informationslagringssystem som kunde minska redundans och inkonsistens i den lagrade informationen och erbjuda stöd för parallell åtkomst och transaktionshantering (Elmasri & Navathe, 2000). Tack vare sitt breda användningsområde och sin långa tid på marknaden är relationsdatabaser den mest använda databastypen.

Ett relationsdatabashanteringssystem (RDBMS) lagrar informationen i tabeller, vilka är relaterade till varandra. Varje rad i en tabell identifieras unikt med en eller flera kolumner som kallas primärnyckel. Relationerna mellan tabellerna realiseras genom att tabellerna delar på en eller flera kolumner som kallas främmande nyckel.

2.2 Objektorienterade databaser

Objektorienterade databaser utvecklades för att möta kraven för mer komplexa databasapplikationer och databassystem som hanterar mer applikationsinriktad data, så som *computer-aided design* (CAD) och *computer-aided software engineering* (CASE) (Elmasri & Navathe, 2000). Dessa system behövde bland annat kunna hantera komplex data, så som grafik, och skapandet av egna datatyper. Det finns också ett behov av att kunna definiera egna, applikationsspecifika operationer. Dessa operationer kopplas till en objektgrupp, en så kallad klass.

Ett objektorienterat databashanteringssystem (OODBMS) lagrar informationen i en samlig relaterade objekt. Varje objekt är en instans av en klass, som är en definition på strukturen hos en viss objekttyp i databasen. I klassen definieras vilken data som objektet lagrar och vilka operationer som kan utföras på objektet. Relationerna mellan objekt realiseras genom till exempel länkar mellan objekten om relationen skulle vara en association.

2.3 Evolutionära databaser

Databasens struktur, det vill säga vad för information som lagras i databasen och hur denna lagras, har länge ansetts vara statisk, men detta har visat sig vara felaktigt. Databasapplikationers krav ändras i takt med dess omgivning och detta medför att kraven på vad databasen ska lagra ändras. Detta har lett fram till att evolutionära DBMS har utvecklas. En evolutionär databas kan hantera ändringar i databasens struktur, i det så kallade schemat, undertiden som systemet är i gång. Därmed kommer man ifrån kostsamma uppdateringar av databasen. I kapitel 2.4 kommer begreppet databasschema att diskuteras vidare eftersom detta är ett centralt begrepp inom evolutionära databaser och i kapitel 2.5 fortsätter diskussionen om evolution av databasscheman.

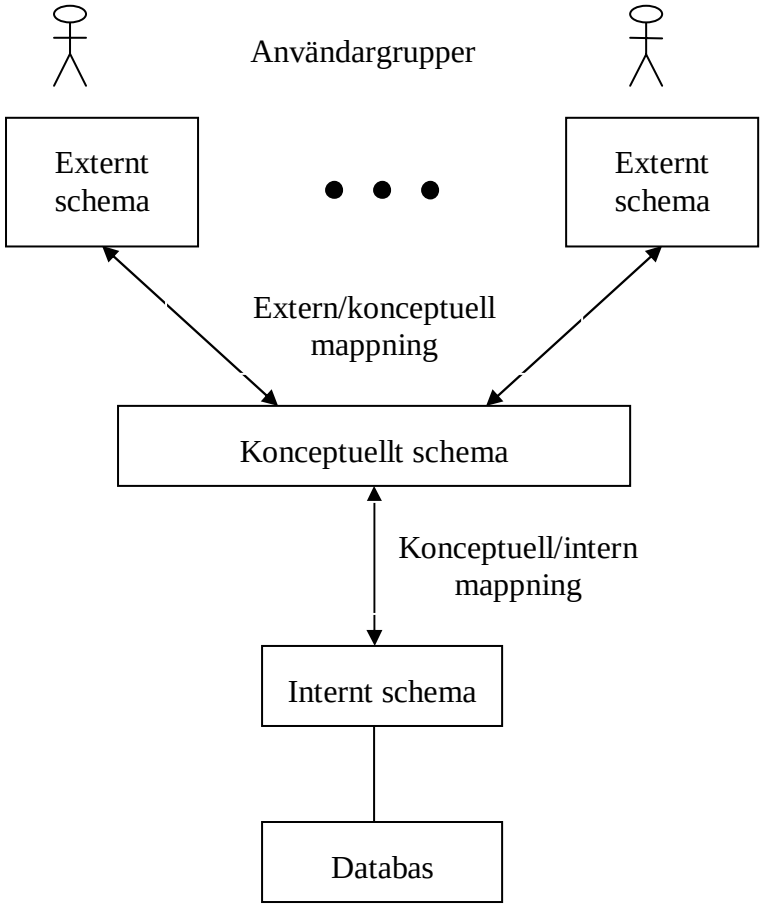
2.4 Databasscheman

Ett databasschema är en beskrivning av en databas, det vill säga en beskrivning på vad för information som lagras i databasen och hur denna är strukturerad (Elmasri & Navathe, 2000). Databasschemats struktur skiljer sig en del beroende på om det tillhör ett RDBMS eller ett OODBMS. I en relationsdatabas beskrivs operationer och kontrollfunktioner separat från tabeller och relationer, medan allt detta kapslas in i klasser i en objektorienterad databas.

Elmasri och Navathe (2000) talar om tre olika typer av scheman: det konceptuella, det interna och det externa schemat.

- Det konceptuella schemat beskriver strukturen på databasen utan att blanda in detaljer om den fysiska lagringsstrukturen. I denna beskrivs vilka tabeller som finns i databasen, hur dessa är relaterade till varandra, vilka datatyper en viss kolumn använder sig av och vilka operationer som databasen erbjuder.
- Det interna schemat beskriver den fysiska lagringsstrukturen hos databasen, det vill säga beskriver hur och var information lagras i databasen. Detta schema används av DBMS:et när data läggs till, tas bort eller ändras i databasen.
- Det externa schemat beskriver den delen av databasen som en viss användargrupp är intresserad av och döljer resten av databasen. En användargrupp kan till exempel vara ett antal databasapplikationer.

Figur 1 visar hur dessa tre schematyper är kopplade till varandra. Kopplingen mellan de tre schemana är stark. Ändringar i ett av dessa scheman påverkar de andra eftersom dessa mappas, se Figur 1, så att en förfrågan mot databasen om att lagra, hämta eller ändra data kan passera från det externa schemat till det interna.



Figur 1: De tre schematyperna. (Bild hämtad från Elmasri & Navathe, 2000)

Vidare i rapporten kommer det konceptuella schemat att benämnas som schemat eller databasschemat eftersom det är mot detta schema som ändringarna i databasens struktur utförs.

2.5 Evolution av databasscheman

Alla databassystem som kan hantera förändringar av databasschemat under tiden som databassystemet är igång kan enligt De Castro, et al. (1997) grupperas i tre kategorier beroende på hur utvecklat stödet för ändringar i databasschemat är. Dessa kategorier är schemaförändring (kapitel 2.5.1), schemaevolution (kapitel 2.5.2) och schema versioning (kapitel 2.5.3).

2.5.1 Schemaförändring

Schemaförändring är den första av de tre kategorierna som utvecklades. I ett databassystem som har stöd för schemaförändringar är det möjligt att ändra schemat

på databasen medan systemet är igång. All data i de ändrade tabellerna alternativt alla objekt tillhörande de ändrade klasserna kommer dock att förloras, eftersom inget stöd för databevarande finns i kategorin schemaförändringar (De Castro, et al., 1997). Antag till exempel att Tabell 1 ska ändras. Den ändring som ska göras är att kolumnen *Stad* ska bytas ut mot en kolumn *Telefonnummer*. Om ändringen sker i ett databassystem som har stöd för schemaförändring, så kommer all data i tabellen att försvinna. Tabell 2 visar resultatet av förändringen.

Tabell 1: En tabell med persondata

Namn	Personnummer	Stad
Jan Jansson	710111-7117	Gävle
Stina Student	801020-9010	Skövde

Tabell 2: Tabell 1 efter ändringen (schemaförändring)

Namn	Personnummer	Telefonnummer
(tomt)		

I en objektorienterad databas med stöd för schemaändring skulle effekten av en motsvarande ändring vara att alla objekt tillhörande klassen skulle tas bort.

Eftersom denna nivå inte bevarar informationen i tabellen så kan inkonsistens uppstå om andra tabeller har främmande nycklar kopplade till den ändrade tabellen. Detta kan bara lösas genom att ta bort data ur de övriga tabellerna som är beroende av den ändrade tabellen på grund av främmande nycklar. Det är inte många databasapplikationer som kan tolerera en sådan dataförlust och därmed används inte denna nivå särskilt ofta.

2.5.2 Schemaevolution

Schemaevolution är en utökning av föregående nivå. Ett databassystem som har stöd för schemaevolution ger samma möjligheter som ett databassystem som har stöd för schemaförändringar, fast utan att någon data tas bort från den ändrade tabellen (De Castro, et al., 1997). Schemaevolution bevarar dock bara den senaste versionen av databasschemat. Enligt Wei och Elmasri (2000) kan detta göra att applikationer som använder databasen inte fungerar korrekt efter en ändring eftersom applikationen kan begära information från databasen som inte längre lagras. Antag till exempel att Tabell 1 ska ändras. Den ändring som ska göras är att kolumnen *Stad* ska bytas ut mot en kolumn *Telefonnummer*. Om ändringen sker i ett databassystem som har stöd för schemaevolution, så kommer inte någon data i tabellen att försvinna, vilket skulle ha inträffat om databassystemet bara hade stöd för schemaförändring. All data i kolumnen *Stad* försvinner däremot eftersom kolumnen tagits bort från systemet. Applikationer som använder kolumnen *Stad* kommer inte längre att fungera. Tabell 3 visar resultatet av förändringen.

Tabell 3: Tabell 1 efter ändringen (schemaevolution)

Namn	Personnummer	Telefonnummer
Jan Jansson	710111-7117	NULL
Stina Student	801020-9010	NULL

I en objektorienterad databas med stöd för schemaevolution skulle effekten av en motsvarande ändring vara att alla objekt tillhörande klassen skulle finnas kvar efter förändringen. Däremot konverteras objekten om till den struktur som den nya versionen av klassen har.

2.5.3 Schema versioning

Schema versioning är den senast utvecklade av de tre nivåerna och den som det bedrivs mest forskning kring just nu. Därför kommer detta arbete att fokusera på schema versioning eftersom det fortfarande finns en den problematik inom detta som ännu inte granskats. Schema versioning ger samma stöd som schemaevolution, men gamla versioner av schemat bevaras så att applikationer som använder sig av en gammal version av schemat fortfarande kan användas utan att behöva uppdateras (De Castro, et al., 1997). Antag till exempel att Tabell 1 ska ändras. Den ändring som ska göras är att kolumnen *Stad* ska bytas ut mot en kolumn *Telefonnummer*. Om ändringen sker i ett databassystem som har stöd för schema versioning, så kommer en ny tabell att skapas som är en ny version av den gamla tabellen och den gamla tabellen kommer att sparas. Detta gör att applikationer som använder kolumnen *Stad* kommer att kunna använda den gamla versionen av tabellen. Tabell 4.1 och 4.2 visar resultatet av förändringen.

Tabell 4.1: Tabell 1 efter ändringen (gamla versionen)

Namn	Personnummer	Stad
Jan Jansson	710111-7117	Gävle
Stina Student	801020-9010	Skövde

Tabell 4.2: Tabell 1 efter ändringen (nya versionen)

Namn	Personnummer	Telefonnummer
Jan Jansson	710111-7117	NULL
Stina Student	801020-9010	NULL

I en objektorienterad databas med stöd för schema versioning skulle effekten av en motsvarande ändring vara att den gamla versionen av klassen och dess objekt skulle sparas, det vill säga effekten är samma som för en relationsdatabas.

2.6 Mer om schema versioning

Schema versioning inför ett nytt begrepp som inte finns i samband med vanliga databaser och i de två andra nivåerna av evolution av databasscheman. Detta begrepp är versioner och medför att två ställningstaganden måste göras innan stöd för schema versioning implementeras i ett DBMS. Det första ställningstagandet är hur relationen mellan versioner och data ser ut. Det andra ställningstagandet är hur versionerna ska hanteras.

2.6.1 Relationen mellan version och data

I ett DBMS som ger stöd för schema versioning kan relationen mellan version och data ses på två sätt: antigen kan ett dataobjekt bara nås från en version eller så kan dataobjektet nås från alla versioner. Om varje dataobjekt tillhör en version går det bara att nå dessa via den version de tillhör. Varje version behandlas som en enskild tabell med sin egen mängd data. Om däremot varje dataobjekt kan nås från alla versioner, så tolkas eller avgränsas ett dataobjekt av den version som används för att nå den. I den objektorienterade modellen för schema versioning som Monk (1993) har tagit fram måste ett objekt konverteras till den version som används innan den kan användas utanför databasen. Fördelen med att ett dataobjekt kan nås via alla versioner är att applikationer som använder olika versioner kan utbyta information med varandra via databasen.

2.6.2 Hantering av versioner

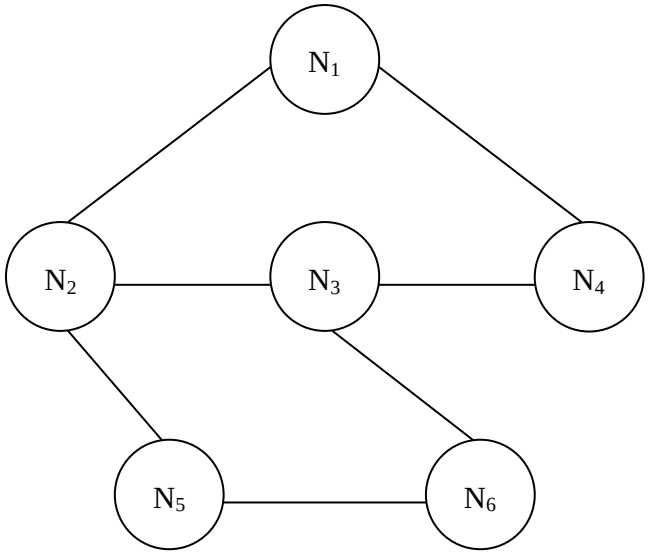
Med hur hantering av versioner ska gå till innebär i detta avseende hur versionerna ska identifieras. En nackdel med schema versioning är att databasapplikationerna måste veta vilken version de ska ställa sina förfrågningar mot för att vara säkra på att få ett korrekt svar. Därför är det viktigt att identifieringen av versioner sker på ett hanterbart sätt så att både användare och DBMS lätt kan skilja på olika versioner. Enligt Jensen och Böhlen (2001) används ofta temporala DBMS som grund när ett DBMS med stöd för schema versioning implementeras eftersom förändringar av databasscheman sker i förhållande till tid, det vill säga det finns oftast bara en aktuell version av schemat åt gången. En temporal databas är en databas som inte bara hanterar nuvarande data i databasen utan lagrar också data från det förflutna och data som ska användas i framtiden (Wei & Elmasri, 2000). Data i en temporal databas datummärks så att det går att se när den är aktuell. Enligt De Castro, et al. (1997) sker hanteringen av versioner på samma sätt. Versionerna märks antigen med datumet för då ändringen utfördes eller med det period då versionen är aktuell. Jensen och Böhlen (2001) påstår dock att i ett generellt perspektiv så är inte denna hantering av versioner intressant för användare av en databas. En användare, så som en databasapplikation, arbetar mot ett externt schema och i externa scheman finns det bara en version av till exempel en tabell. Hantering av versioner är en central del av detta arbete och kommer vidare att diskuteras i kapitel 3.

2.7 Träd och grafer

Träd och grafer är två begrepp som kommer att användas längre fram i denna rapport. Därför kommer både träd och grafer att diskuteras översiktligt i detta kapitel och skillnaderna mellan dessa presenteras.

2.7.1 Grafer

En graf är samling sammankopplade noder och används ofta för att beskriva relationerna mellan en grupp element (Rosen, 1999). En nod är en representation av ett sådant element. Kopplingen mellan två noder representerar relationen mellan noderna. Grafer används bland annat till att beskriva hur datornätverk ser ut eller för att beskriva strukturen i en databas. Figur 2 visar ett exempel på en graf som innehåller noderna N_1,..,N_6.

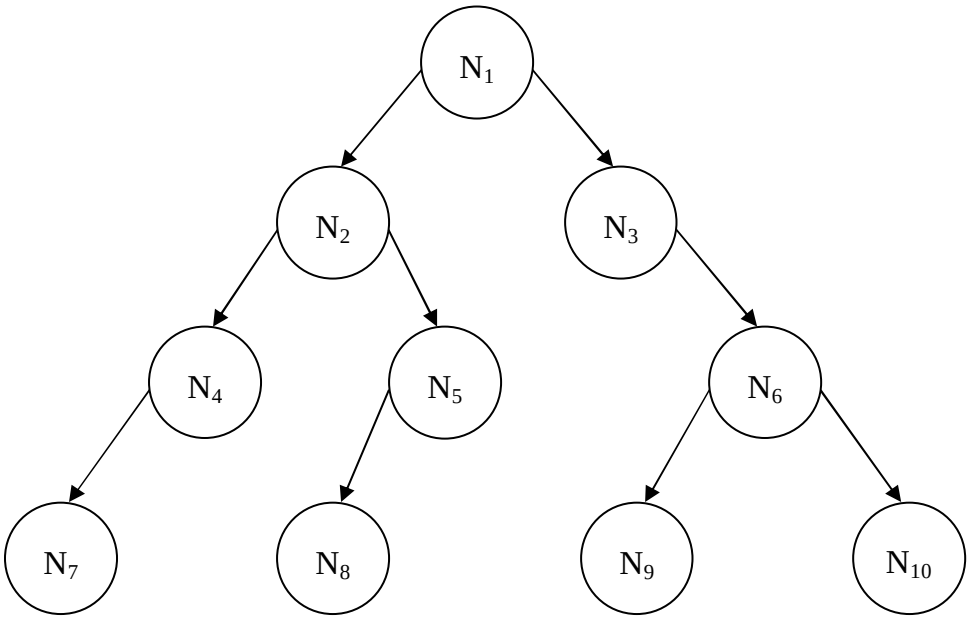


Figur 2: Exempel på en graf

2.7.2 Träd

Ett träd är en graf där det bara finns en väg mellan två noder (Rosen, 1999). I detta arbete kommer bara rotade träd att användas. Ett rotat träd är ett träd där alla kopplingar är enkelriktade. I ett rotat träd finns det alltid exakt en nod som bara har utåtgående kopplingar. Denna nod kallas rotnoden eller bara roten i trädet. En nod i ett rotat träd som inte har någon utgående koppling kallas löv eftersom dessa befinner sig längst ut i trädet. Ett annat begrepp som används i samband med träd är gren. En gren är en del av trädet efter en förgrening i trädet. Första noden i en gren är den nod där förgreningen sker. Barnnoder, eller subnoder, och föräldranoder, eller supernoder, är två andra begrepp som används i samband med träd. Barnnoderna till en viss nod är de noder som den är kopplad till med utåtgående kopplingar. Föräldranod är den nod

som en viss nod är kopplad till med inåtgående koppling. Varje nod har högst en föräldranod. Figur 3 visar ett exempel på hur ett träd som innehåller noderna N_1,..,N_{10} och där N_1 är rotnod.



Figur 3: Exempel på ett träd

3 Problembeskrivning

Även om schema versioning ger ett bra stöd för evolution av databasscheman och ger bakåtkompatibilitet så har den fortfarande en nackdel: databasapplikationen måste veta vilken version av schemat man ska ställa en viss fråga emot för att få ett korrekt svar. Detta gäller dock inte för alla databaser, så som temporala databaser (Jensen och Böhlen, 2001). Detta arbete kommer därför att fokusera på databaser och databasapplikationer där det inte finns något intresse att veta eller ange vilken version en viss fråga ska ställas emot. Hos dessa finns det ett behov av att kunna ställa frågor mot databasen utan att behöva ange version, vilket innebär att DBMS:et måste besluta om vilken version som ska användas. Detta innebär att versionshanteringen är osynlig utanför DBMS:et. Denna egenskap kommer vidare att kallas versionstransparens i denna rapport.

Det finns flera sätt att uppnå versiontransparens på. Ett sätt är att ställa frågan mot en förbestämd version så som den senast skapade versionen eller mot en defaultversion (Roddick, 1995). Andra alternativ skulle vara att ställa frågan mot den senast eller mest använda versionen. Gemensamt för dessa är att de inte kan garantera att ett korrekt svar ges eftersom det inte är säkert att versionen som frågan ställdes emot var den rätta. För att vara säker på att frågan ställs mot rätt version behöver DBMS:et kunna räkna ut vilken version som frågan ska ställas mot. Ett sätt är att använda informationen i själva frågan, det vill säga vilka kolumner som frågan använder sig av. Denna information kan sedan användas för att ta fram rätt version genom att matcha kolumnerna som används i frågan med kolumnerna i versionerna. Denna lösningsansats kommer att behandlas vidare i detta arbete. I ansatsen ingår det att versionerna för en tabell söks igenom för att hitta rätt version. Därmed är det fördelaktigt om versionerna bildar en struktur, så som en graf- eller trädstruktur, eftersom detta kan underlätta sökningen.

3.1 Problemprecisering

Syftet med detta arbete är att undersöka i vilka fall det är möjligt att uppnå versionstransparens i en relationsdatabas med stöd för schema versioning genom att använda ansatsen ovan. Resultatet av undersökningen i detta arbete ska kunna svara på följande fråga:

> I vilka fall går det att uppnå versiontransparens i ett RDBMS med stöd för schema versioning genom att RDBMS:et tar fram rätt version för varje fråga med hjälp av information i själva frågan?

Med att uppnå versiontransparens avses här att ingen del av versionshanteringen får vara synlig för användare av systemet, så som databasapplikationer. DBMS:et ska utåt sett fungera som om den bara har stöd för schemaevolution. Detta innebär även att ingen versionshantering är synlig vid förändringar av databasschemat. De fall som ska undersökas är de typer av förändringar på databasschemat som tas upp i nästa kapitel.

3.2 Avgränsningar

I relationsdatabaser är tabeller det centrala begreppet och därmed kommer bara förändringar av tabeller att tas hänsyn till i denna undersökning. Detta innefattar följande förändringar av databasschemat:

- Skapa tabell
- Ta bort tabell
- Lägga till kolumn till tabell
- Ta bort kolumn från tabell
- Ändra datatyp för en kolumn

Övrig funktionalitet som kan finnas i ett RDBMS kommer inte att behandlas i detta arbete. Funktionalitet som använder tabeller, så som procedurer, triggers och vyer, kommer inte att behandlas eftersom dessa kommer att använda en version av tabeller i ett RDBMS med stöd för schema versioning och därmed inte påverkas av förändringar av dessa. Funktionalitet som används av tabeller, så som domäner, check constraints och härledda attribut, kommer inte att behandlas eftersom dessa inför ny evolutionsproblematik eftersom förändringar av dessa innebär att en ny version måste skapas. I detta arbete kommer heller inte ändringar av namn, till exempel på tabeller och kolumner, att behandlas eftersom problematiken med detta är ett namnkonventionsproblem och därmed inte ingår i det problemområde som behandlas i detta arbete.

4 Metod

I ett arbete där en ny teknik, algoritm eller motsvarande ska prövas så behöver den implementeras för att den ska kunna testas på ett korrekt sätt (Berndtsson, Hansson, Olsson & Lundell, 2002). I detta arbete ska det undersökas om det går att uppnå versiontransparens genom att använda den lösningsansats som beskrivs i kapitel 3, det vill säga att använda informationen i frågan som ställs mot databasen för att ta reda på vilken version den ska ställas emot. Detta innebär att en fullständig lösning som utgår från denna ansats måste implementeras för att undersökningen ska kunna genomföras på ett korrekt sätt. Vilka typer av implementationer som går att välja mellan kommer att diskuteras i kapitel 4.1. För att kunna verifiera att den fullständiga lösningen uppnår versionstransparens måste den testas. Denna testning kommer att diskuteras i kapitel 4.3.

4.1 Val av implementation

Det finns två olika implementationer att välja mellan: en fysisk implementation eller en logisk implementation. En fysisk implementation innebär i detta fall att ett DBMS som använder metoden ovan implementeras. En logisk implementation innebär att skapa en modell av ett sådant DBMS. En sådan modell ska vara en fullständig beskrivning av hur ett tänkt DBMS fungerar. Till exempel måste hanteringen av schema versioning beskrivas och hur ansatsen ovan realiseras.

Båda alternativen, att skapa ett DBMS och att skapa en modell, är lika användbara för att bevisa att det går att uppnå versionstransparens med hjälp av metoden som beskrivs i kapitel 3. En fysisk implementation har fördelen att den kan testas genom att köra testfallen (Berndtsson, et al., 2002). En fysisk implementation innehåller däremot inte mer information än en logisk implementation. För att skapa ett DBMS, det vill säga en fysisk implementation, behöver man däremot veta hur denna ska fungera, det vill säga det behövs en modell att utgå ifrån innan implementationen av DBMS:et kan påbörjas. Därför kommer en modell att tas fram i detta arbete istället för att implementera ett DBMS.

4.2 Designval

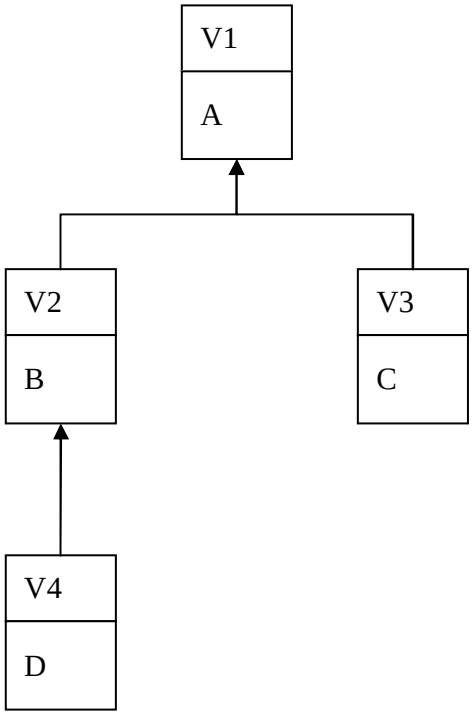
Innan implementationen påbörjas så måste en del grundläggande designval göras. Dessa designval bildar själva lösningsansatsen till en modell och avgör hur modellen i stora drag kommer att fungera. De beslut som redan har tagits tidigare i rapporten är att modellen ska bygga på relationsdatamodellen och ge stöd för schema versioning. Den ska tillåta de ändringar av databasschemat som nämns i kapitel 3.2 och uppnå versionstransparens genom att utgå från den ansats som nämns i kapitel 3. Nedan följer de övriga designval som måste göras innan modellen kan implementeras.

I kapitel 2.6 angavs det att det finns två typer av relationer mellan versioner och data i ett DBMS med stöd för schema versioning: all data tillhör en specifik version, det vill

säga data kan bara nå från dess version, eller all data kan nås från alla versioner. I denna modell kommer all data att kunna nås från alla versioner eftersom detta möjliggör att data delas av databasapplikationer som använder olika versioner.

För att hitta rätt version att ställa en viss fråga emot i en databas där versionerna inte är strukturerade krävs att alla versioner söks igenom för att hitta rätt version. Modellen ska utgå ifrån att det bara finns en korrekt version att ställa en viss fråga emot, även om det är möjligt att ställa frågan mot flera versioner och fortfarande få ett korrekt svar. Det vore fördelaktigt att inte behöva söka igenom alla versioner. Detta kan uppnås om versionerna struktureras i ett träd där varje nod är en version. Det går då att bortse från versioner som tillhör grenar vars första nod inte är förälder till den rätta noden, det vill säga inte innehåller några av de kolumner som ingår i frågan.

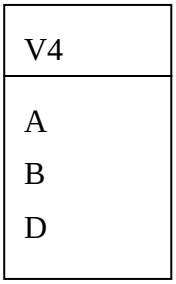
Genom att använda arv kan ovanstående krav uppfyllas. Genom att låta versioner ärva från tidigare versioner bildas en grafstruktur, som vidare kommer att kallas versionsgraf. I versionsgrafen är alla kopplingar mellan versionerna enkelriktade och alla versioner ärver från en och samma superversion, alla versioner har en gemensam nämnare. Detta gör att versionsgrafen kan sökas igenom på samma sätt som om den vore ett träd. Alla versioner ärver från en gemensam superversion, vilket gör att all data i tabellen nås från alla versioner. Figur 4 visar ett exempel på en versionsgraf som innehåller fyra olika versioner (V1, V2, V3 och V4) och fyra kolumner (A, B, C och D).



Figur 4: En versionsgraf.

Arvsmekanismen gör också att en version bara behöver lagra de kolumner som är unika för den, alla andra kolumner ärvs in i versionen. Däremot gör arvsmekanismen att versionen ser fullständig ut gentemot en användare, så som en databasapplikation. Till exempel ser version V4 från Figur 4 ut att innehålla kolumnerna A, B och D

gentemot användaren fast den egentligen bara innehåller kolumn D. Figur 5 visar hur version V4 från Figur 4 ser ut gentemot en användare.



Figur 5: V4:s utseende mot användaren.

4.3 Testning av modellen

För att kunna bevisa att den lösning som ska tas fram verkligen uppnår versionstransparens så måste den testas. Det är nödvändigt att ta fram tillräckligt många testfall för att kunna bevisa att modellen verkligen uppnår versionstransparens. Med tillräckligt många testfall menas att testfallen ska täcka alla situationer som kan påverkas av hanteringen av versioner, så som olika typer av förändringar av databasschemat och speciella fall hos dessa.

I detta arbete ska det undersökas om det går att uppnå versionstransparens genom att använda information i frågan som ställs mot databasen för att ta reda på vilken version den ska ställas emot. Detta ger tre typer av frågor, beroende på mängden information som frågan innehåller, som måste hanteras av modellen. Dessa tre typer av frågor kommer att testas mot var typ av förändring av databasschemat som modellen kan hantera när modellen valideras. De tre typer av frågor som ska ingå i testningen är:

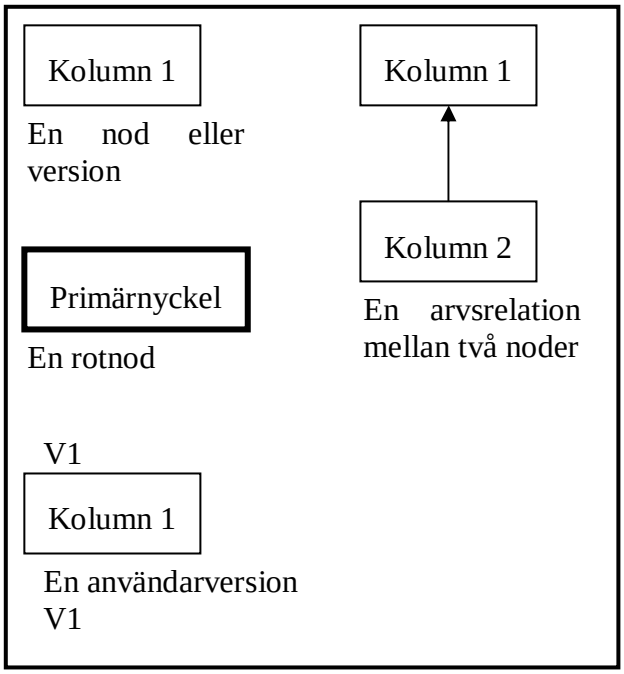
1) Frågor som innehåller tillräcklig med information för att kunna ta fram rätt version att ställa frågan emot.
2) Frågor som inte innehåller någon information som kan användas för att ta fram rätt version att ställa frågan emot.
3) Frågor som inte innehåller tillräckligt med information för att ta fram rätt version, men som kan användas för att eliminera en del av versionerna som inte är den korrekta versionen.

För varje förändring av databasschemat som modellen kan hantera ska de tre typerna av frågor vara möjliga att ställa utan att någon annan information används än den från själva frågan för att hitta rätt version att ställa frågan emot. Det räcker att en av frågetyperna inte kan ställas efter en förändring av databasschemat för att versionstransparens inte ska uppnås.

5 Genomförande

I detta kapitel kommer den modell som har skapats i detta arbete att beskrivas. Beskrivningen av modellen har delats upp i två delar. I första delen kommer det att beskrivas hur modellen hanterar förändringar av schemat och hur versionerna hanteras. I andra delen kommer hanteringen av frågor att beskrivas.

En del exempel kommer att ges för att förtydliga beskrivningen av modellen. Därför kommer den syntax som kommer att användas att förklaras. Ett flertal exempel kommer att behandla versioner och hantering av versioner och därför är det viktigt att versionerna i exemplen tydligt definieras. En version definieras med de kolumner som versionen innehåller satta inom klamrar på följande sätt: {kolumn1, kolumn2, …, kolumnN}. Primärnyckeln är alltid den eller de första kolumnerna i ordningen. En del diagram över versionsgrafer kommer att ingå i exemplen. Syntaxen för dessa diagram finns i Figur 6. En nod representeras av en rektangel med kolumnnamnet inuti. Rotnoden representeras av en rektangel med tjock kant innehållande primärnyckeln. En arvsrelation mellan två noder representeras av en pil där pilen pekar mot föräldranoden.



Figur 6: Syntax för versionsgrafsdiagram

5.1 Implementation av modellen

Vid implementationen av modellen kommer designvalen från kapitel 4.2 att vara utgångspunkt. Detta innebär att modellen kommer att vara en utbyggnad av relationsdatamodellen. Modellen ska ge stöd för schema versioning och vara versionstransparent. Data i en tabell ska delas mellan alla versioner av tabellen och detta ska uppnås genom att använda arv. Att modellen är en utbyggnad av

relationsdatamodellen betyder att modellen ska vara som ett lager mellan en RDBMS och databasapplikationer. Därmed kan modellen använda den funktionalitet som RDBMS:et erbjuder. Samma frågespråk som RDBMS:et använder ska användas av modellen utan att detta byggs ut eller förändras eftersom versionshanteringen ska vara osynlig. Däremot erbjuder inte modellen versionstransparens vid förändringar av databasschemat, vilket innebär att versionen på tabellen som ska förändras måste anges.

Som nämnts tidigare i rapporten så kommer alla versioner till en tabell att lagras i en versionsgraf. Varje nod i versionsgrafen innehåller bara en kolumn. Detta gäller dock inte för rotnoden. Detta gör det lättare att lägga till en ny version av en tabell eftersom inga noder i versionsgrafen behöver förändras. De enda förändringarna som görs på versionsgrafen är att flytta och lägga till noder. Detta gör att inte alla noder i versionsgrafen är en version som skapats av någon användare. På grund av arvsmekanismen kan alla noder användas som en version av tabellen. Detta ger två typer av versioner i versionsgrafen: användarversioner och systemversioner. En användarversion är en version skapad av en användare av databasen. En systemversion är en version som skapats av DBMS:et och denna ska inte kunna användas av eller vara synlig för någon användare. DBMS:et måste kunna skilja på system- och användarversion eftersom dessa ska hanteras olika. Detta görs lättas genom att markera till exempel användarversionerna så att DBMS:et kan veta vilken typ av version en viss nod är.

Primärnyckel i en tabell finns alltid i rotnoden på versionsgrafen eftersom den finns med i alla versioner av en tabell. Detta gör också att primärnyckeln är oförändringsbar eftersom om primärnyckeln ändras för en ny version är det inte säkert att primärnyckeln för de gamla versionerna unikt kan identifiera alla rader i tabellen. En primärnyckel kan innehålla flera kolumner och eftersom en nod bara får innehålla en kolumn måste detta hanteras på ett särskilt sätt. Genom att låta rotnoden vara tom på kolumner och låta den ärva från noder som innehåller de kolumner som ingår i primärnyckeln, så är problemet löst utan att det påverkar hanteringen av versionsgrafen.

I ett sådant RDBMS som modellen arbetar emot finns det två typer av relationer mellan tabeller: association, det vill säga främmande nycklar, och arv. Dessa relationer är inte relationer mellan versioner i versionsgrafen utan relationer mellan tabeller. Relationerna måste hanteras av modellen på ett korrekt sätt så att inga integritetsregler bryts. En främmande nyckel hanteras som en vanlig kolumn eftersom den inte påverkas av versionshanteringen. En arvsrelation i en databas med stöd för schema versioning hanteras inte som en relation mellan två tabeller utan som en relation mellan två versioner av tabellerna och hanteras av modellen därefter.

5.1.1 Ändringar av tabeller

När ett databasschema förändras så har en eller flera av tre förändringar hänt: en eller flera tabeller har skapats, en eller flera tabeller har tagits bort eller en eller flera tabeller har ändrats.

När en tabell skapas så placeras primärnyckeln i rotnoden och resten av kolumnerna placeras i godtycklig ordning i rak följd efter rotnoden. Detta ger en versionsgraf med bara en gren eftersom det bara finns en användarversion av tabellen. Eftersom

ordningen är godtycklig så kan den behöva ändras när en ny användarversion ska läggas till. Detta kan även gälla grenar i versionsgrafer som innehåller flera användarversioner. Denna omflyttning är nödvändig på grund av att ordning på noderna måste vara korrekt om arvsrelationerna ska resultera i korrekta användarversioner. Noderna får därmed byta plats om ingen användarversion förändras av detta, det vill säga ingen annan gren får flyttas och ingen förgrening får passeras när grenen flyttas.

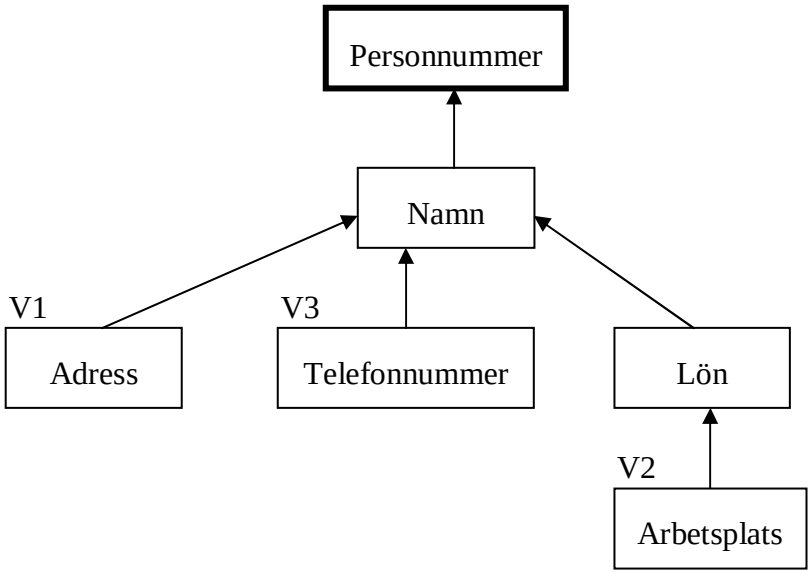
När en tabell tas bort så tas den inte bort fysiskt eftersom alla versioner sparas i en databas med stöd för schema versioning. Tabellen används inte längre av den användargrupp som ändring utfördes för och tabellen ska vara osynlig för dessa. Tabellen tas bort från det externa schemat för den användargrupp som tabellen togs bort för, men är kvar i det gemensamma databasschemat.

När en tabell ändras innebär detta ofta att någon förändring sker på den samling av kolumner som tabellen innehåller. Ändringar på en tabell sker i två nivåer i denna modell: konceptuell förändring av tabellen och ändringar i versionsgrafen. En konceptuell förändring av en tabell skapar en ny användarversion som läggs till i versionsgrafen. Modellen tar fram utseendet på den nya versionen med hjälp av de konceptuella förändringarna och lägger till den nya versionen till versionsgrafen. Denna arbetsmetod används eftersom det inte går att direkt överföra de konceptuella förändringarna till förändringar på versionsgrafen. I nästa kapitel kommer denna arbetsmetod att exemplifieras och det kommer tas upp hur förändringar av versionsgrafen hanteras.

5.1.2 Ändringar av versionsgrafen

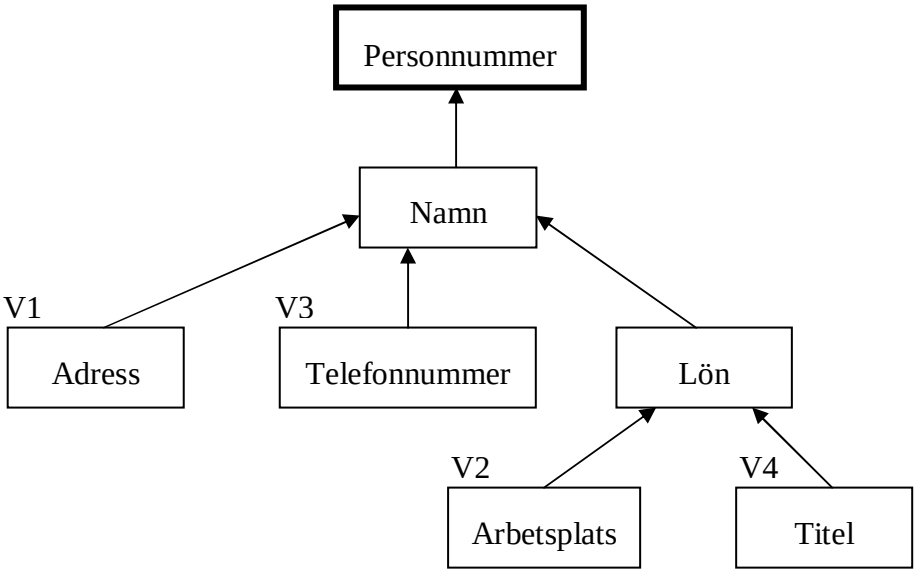
De ändringar av tabeller som tagits med i denna undersökning är lägga till kolumn, ta bort kolumn och ändra datatyp för kolumn. Att lägga till eller ta bort kolumner orsakar inte några svårhanterliga ändringar av versionsgrafen. De ändringar som sker på versionsgrafen i dessa fall är att nya noder och nya förgreningar skapas.

Antag till exempel att en tabell Personregister, som det redan finns tre användarversioner av, ska ändras. De befintliga användarversionerna är V1={Personnummer, Namn, Adress}, V2={Personnummer, Namn, Lön, Arbetsplats} och V3={Personnummer, Namn, Telefonnummer}. Versionsgrafen för Personregister med dessa tre versioner visas i Figur 7.



Figur 7: Versionsgraf av tabellen Personregister (version 1)

Ändringen som ska göras är att i version V2 ska kolumnen Arbetsplats tas bort och ersättas med kolumnen Titel, det vill säga en ny version V4={Personnummer, Namn, Lön, Titel} ska läggas till i versionsgrafen. Denna förändring ger en ny förgrening vid noden som innehåller kolumnen Lön. Figur 8 visar hur versionsgrafen ser ut efter ändringen.

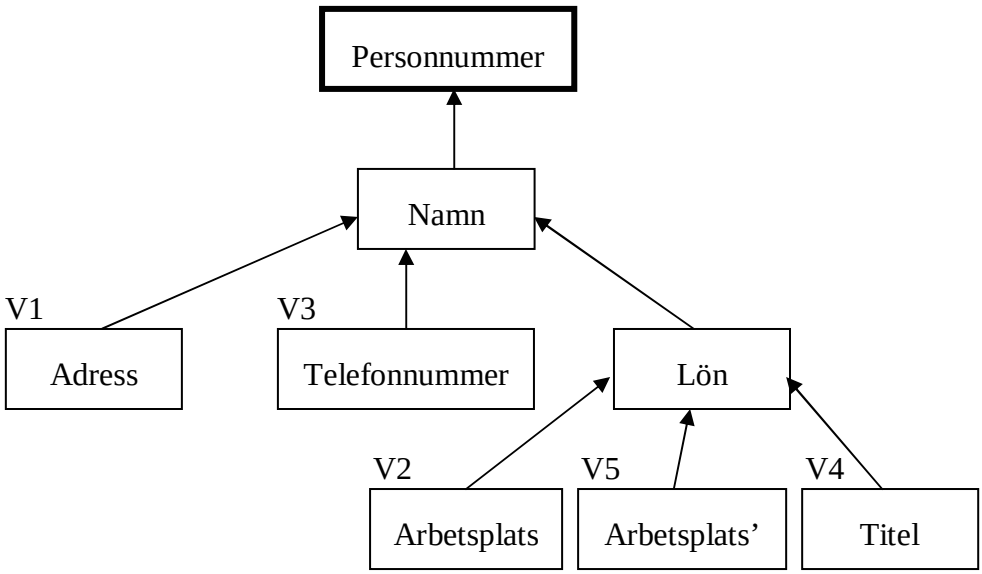


Figur 8: Versionsgraf av tabellen Personregister (version 2)

När datatypen för en kolumn ändras finns det dock en del problematik. För att både den nya och den gamla versionen av kolumnen ska kunna dela data måste all data kunna konverteras mellan de två datatyperna. Om detta inte är möjligt att göra så kan

inte de två versionerna av kolumnen dela data. Att hantera versioner av kolumner ingår dock inte i avgränsningen för detta arbete. Därför kommer modellen att skapa en kolumn med samma namn när datatypen för en kolumn ändras. All data kommer att kopieras och konverteras från den gamla kolumnen till den nya när förändringen utförs för att ingen data ska förloras vid ändringen. Kolumnerna kommer inte att dela data efter förändringen eftersom det inte är säkert att konverteringen längre kan utföras. Konverteringen från den gamla datatypen till den nya måste vara möjlig om förändringen ska tillåtas. Om konverteringen inte är möjlig så måste användaren ta bort den kolumnen som skulle ändras och skapa en ny med samma namn som använder den nya datatypen. Detta tvingar användaren att vara medveten om att all data från den gamla kolumnen går förlorad om ändringen sker.

Antag till exempel att tabellen Personregister ska ändras ytterligare en gång. Figur 8 visar versionsgrafen innan ändringen. Ändringen som ska göras är att datatyp för kolumnen Arbetsplats i version V2 ska ändras. Detta bildar en ny version V5={Personnummer, Namn, Lön, Arbetsplats'} i versionsgrafen för Personregister. Figur 9 visar hur versionsgrafen ser ut efter ändringen.

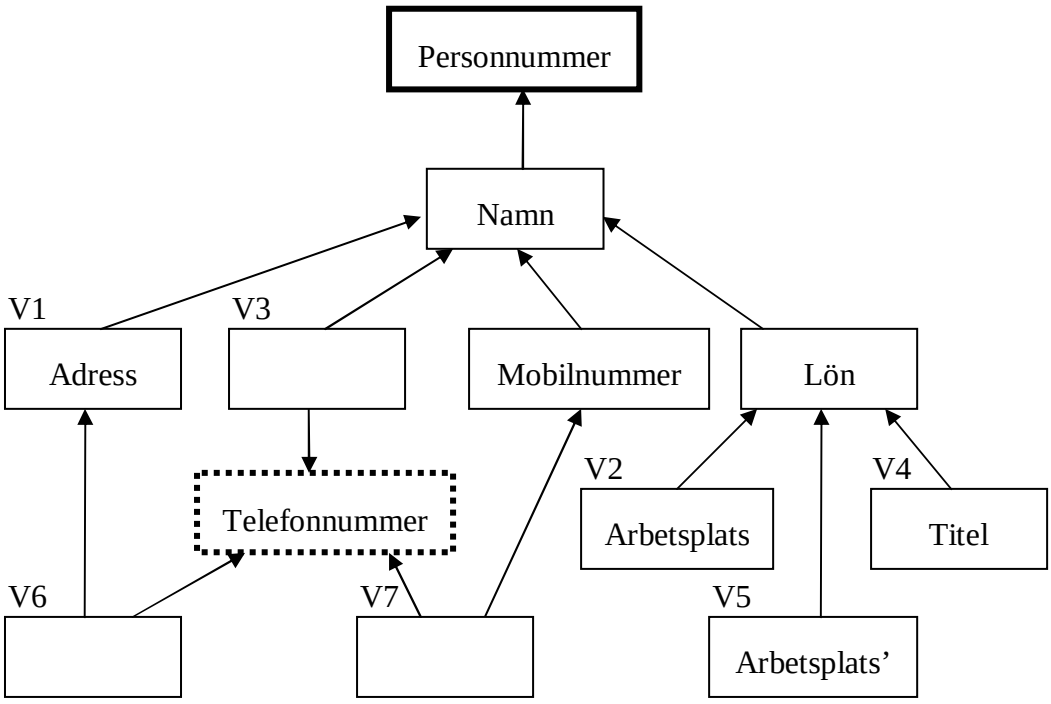


Figur 9: Versionsgraf av tabellen Personregister (version 3)

Ett problem kan uppstå när versionsgrafen förändras. En kolumn kan behöva användas av två olika grenar. Lösningen på detta problem är att placera kolumnen i en subrot. En subrot är en nod som inte ärver från någon annan nod, som en rotnod, men kan ärvas in direkt av vilken nod som helst i versionsgrafen, förutom av rotnoden och andra subrötter. Noder som ärver från en subrot får inte innehålla någon egen kolumn eftersom det ska se ut som om den innehåller kolumnen från subroten. På detta sätt kan en kolumn finnas i flera grenar. Figur 10 visar syntaxen för subrötter i diagram som beskriver versionsgrafen.

Figur 10: Syntax för subrötter i versionsgrafsdiagram

Antag att två nya versioner ska läggas till i versionsgrafen: version V6={Personnummer, Namn, Adress, Telefonnummer} och version V7={Personnummer, Namn, Telefonnummer, Mobilnummer}. Detta gör att en subrot som innehåller kolumnen Telefonnummer måste läggas in i versionsgrafen på grund av att tre grenar i versionsgrafen kommer att använda kolumnen Telefonnummer efter ändringen. Figur 11 visar hur versionsgrafen för tabellen Personregister ser ut efter denna ändring.



Figur 11: Versionsgraf av tabellen Personregister (version 4)

5.2 Hantering av frågor

I detta kapitel kommer frågehanteringen att beskrivas. Det finns tre olika typer av frågor som modellen måste kunna hantera (se kapitel 4.3). Dessa frågetyper är:

1) Frågor som innehåller tillräcklig med information för att kunna ta fram rätt version att ställa frågan emot.
2) Frågor som inte innehåller någon information som kan användas för att ta fram rätt version att ställa frågan emot.

3) Frågor som inte innehåller tillräckligt med information för att ta fram rätt version, men som kan användas för att eliminera en del av versionerna som inte är den korrekta versionen.

För att bättre kunna förstå vilken information som används från frågorna som ställs mot databasen kommer strukturen på frågor mot databaser och strukturen på frågor från de tre olika frågetyperna att beskrivas. Vidare kommer sökningsmetoden för att hitta rätt version att beskrivas med hjälp av informationen från frågan och hur frågetyperna påverkar denna att beskrivas.

5.2.1 Databasfrågor

Många RDBMS använder frågespråk som strukturerar frågor i Hämta-Från-Där-block. Ett sådant block kan delas upp i tre delar (Elmasri & Navathe, 2000). I första delen anges vilka kolumner som ska ingå i svaret, i andra delen anges mot vilken eller vilka tabeller frågan ska ställas emot och i tredje delen anges kraven som avgör vilka rader som kommer att ingå i svaret. Strukturen på frågorna kan sammanfattas på följande sätt:

Hämta kolumner k_1, ..,k_N från tabell T där kraven D gäller.

I de flesta fall är det inte nödvändigt att skriva ut alla kolumner om svaret ska innehålla samtliga kolumner från tabellen. Det är bara att skriva i frågan att alla kolumner ska hämtas från tabellen. Det är inte heller inte nödvändigt att ange krav om alla rader ska hämtas från tabellen. Den information som är intressant från en fråga för att hitta rätt version att ställa frågan emot är de kolumner som ingår i första och tredje delen av frågan. Omfattningen på denna information avgör till vilken av de tre frågetyperna en viss fråga tillhör (se kapitel 4.3). Nedanför följer en mer frågenära beskrivning av de tre frågetyperna:

- Frågetyp 1 är frågor där de kolumner som ska ingå i svaret är angivna. Detta innefattar följande två frågestrukturer:
    - Hämta kolumner k_1, ..,k_N från tabell T där kraven D gäller.
    - Hämta kolumner k_1, ..,k_N från tabell T.
- Frågetyp 2 är frågor där alla kolumner från tabellen eller tabellerna ska ingå i svaret och inga krav har angetts. Detta ger följande frågestruktur:
    - Hämta alla kolumner från tabell T.
- Frågetyp 3 är frågor där alla kolumner från tabellen eller tabellerna ska ingå och där krav angetts. Detta ger följande frågestruktur:
    - Hämta alla kolumner från tabell T där kraven D gäller.

5.2.2 Sökmetod

Grundprincipen för den sökmetoden som används av modellen är att bara söka igenom grenar vars första nod innehåller en kolumn som används i frågan. Denna test görs för varje barnnod till den senast korrekta noden. Sökningen börjar från rotnoden

och fortsätter en nivå i taget tills hela versionsgrafen är genomsökt eller den rätta noden, det vill säga den korrekta versionen, är funnen. Sökningen sker en gren i taget och om ingen av barnnoderna till den senast korrekta noden skulle innehålla en kolumn som används i frågan så testas alla barnnoder till de testade noderna och så vidare.

Beroende på vilken frågetyp en viss fråga tillhör kommer frågan att behandlas olika. Om frågan tillhör frågetyp 1 så kommer rätt version att sökas fram med sökmetoden ovan. Frågan behöver inte i detta fall ställas mot en användarversion eftersom de kolumner som ingår i svaret sållas av frågan, det vill säga utåt sett syns det inte att frågan ställs mot en systemversion. Tillhör frågan frågetyp 2 så görs ingen sökning eftersom frågan inte innehåller någon användbar information. Frågan ställs emot en union (Rosen, 1999) av alla användarversioner vilket gör att en stor mängd data i svaret inte kommer att användas, men det går att garantera att efterfrågad data ingår i svaret. Om frågan tillhör frågetyp 3 så utförs sökningen så långt som möjligt för att sålla bort så många användarversioner som möjligt. Subrötter kan, som nämnts tidigare, ärvas av noder från flera olika grenar i versionsgrafen. Därför måste hela versionsgrafen sökas igenom för att kunna garantera att inte den rätta versionen att ställa frågan emot sållas bort. Frågan ställs mot en union av återstående användarversioner. Detta minskar mängden svarsdata i jämförelse med svarsdata från en fråga av frågetyp 2.

6 Resultat

I detta kapitel kommer resultatet av undersökningen av den modell som tagits fram i detta arbete att presenteras. I undersökningen så undersöktes de typer av förändringar av databasschemat som modellen, som tagits fram i detta arbete, skulle kunna hantera (se kapitel 3.2). Resultatet av varje enskild typ av förändring kommer att presenteras var för sig. Dessa förändringar kan delas in i två grupper: förändringar som påverkar versionsgrafen och förändringar som inte påverkar versionsgrafen. De förändringar som påverkar versionsgrafen är förändringar av tabellens struktur, så som att ta bort och lägga till kolumner. Förändringar som inte orsakar förändringar i versionsgrafen tillhör gruppen av förändringar som inte påverkar versionsgrafen. Därför kommer dessa att presenteras i var sitt kapitel, det vill säga kapitel 6.1 och 6.2.

6.1 Förändringar som inte påverkar versionsgrafen

Det finns två typer av förändringar som inte påverkar versionsgrafens struktur, se ovan. Dessa förändringar är att lägga till och att ta bort tabeller från databasschemat.

När en ny tabell precis lagts till i databasschemat så innehåller dess versionsgraf bara en enda användarversion, vilket medför att det bara finns en version av tabellen att ställa frågor emot. Detta gör att informationen i frågan inte behövs för att veta vilken version den ska ställas emot, vilket innebär att versionstransparens uppnås efter det att en tabell lagts till.

När en tabell tas bort tas den bort från det externa schemat för den användargrupp som ändringen utfördes för. Detta innebär att användare från användargruppen inte längre kan ställa frågor mot den borttagna tabellen. Därmed uppnås versionstransparens efter det att en tabell tagits bort.

6.2 Förändringar som påverkar versionsgrafen

De förändringar som påverkar versionsgrafen är förändringar som orsakar ändringar i versionsgrafen. Dessa ändringar av versionsgrafen är att lägga till kolumn, ta bort kolumn och ändra datatyp för kolumn. I undersökningen så kontrollerades det att dessa inte påverkade versionstransparensen genom att visa att frågor från de tre frågetyperna kunde ställas efter det att dessa ändringar utförts. Det kontrollerades också att inte subrötter förhindrar att rätt version hittas av sökmetoden (se kapitel 5.2.2). I detta kapitel kommer tabellen Personregister från kapitel 5.1.2 att användas i exempel som visar att de olika ändringarna av versionsgrafen inte påverkar versionstransparensen.

6.2.1 Lägga till eller ta bort kolumn

De ändringar av versionsgrafen som orsakas av att kolumner läggs till eller tas bort ur en tabell är att nya noder läggs till och att systemversioner görs om till användarversioner. Dessa ändringar påverkar bara sökmetoden genom att nya förgreningar tillkommer i versionsgrafen. Detta innebär att om sökmetoden kan hitta rätt version av en tabell att ställa en viss fråga emot i en versionsgraf som innehåller förgreningar, så upprätthålls versionstransparensen efter det att kolumner lagts till eller tagits bort. Grunden i sökmetoden går ut på att välja rätt gren vid en förgrening i versionsgrafen, det vill säga modellen som tagits fram i detta arbete kan hantera förgreningar i versionsgrafen. Därmed bevaras versionstransparensen även efter kolumner lagts till och tagits bort från en tabell.

Antag att följande fråga från frågetyp 1 ställs mot version V4 av tabellen Personregister som finns i versionerna V1,..,V4 (Figur 8 visar hur versionsgrafen för tabellen ser ut):

Hämta kolumnerna Namn och Lön från tabell Personregister.

Sökningen efter rätt version att ställa frågan emot börjar med att kontrollera barnnoderna till rotnoden. Rotnodens enda barnnod innehåller kolumnen Namn. Kolumnen Namn ingår i frågan och därför kommer denna nods barnnoder att kontrolleras. Barnnoderna till noden med kolumnen Namn innehåller kolumnerna Adress, Telefonnummer och Lön. Kolumnen Lön ingår i frågan och eftersom alla kolumner i frågan finns i versionen som representeras av noden med kolumnen Lön så avslutas sökningen här. Denna version av tabellen är en systemversion, det vill säga versionen ingår inte i något schema över databasen. Frågan är dock en fråga av frågetyp 1 vilka kan ställas mot systemversioner (se kapitel 5.2.2).

Om frågan skulle vara av frågetyp 2 så hanteras den på ett annat sätt. Frågan skulle då se ut på följande sätt:

Hämta alla kolumner från tabell Personregister.

Frågan innehåller i detta fall inga kolumnnamn. Detta löses genom att slå samman alla användarversioner i en union (se kapitel 5.2.2). Frågan ställs därmed mot en tabell med följande utseende:

{Personnummer, Namn, Adress, Telefonnummer, Lön, Arbetsplats, Titel}

Version V4, det vill säga den version som frågan var tänk att ställas emot, är en delmängd av den unionstabell som frågan ställs emot, vilket innebär att svaret blir korrekt.

Frågor av frågetyp 3 är frågor som innehåller kolumnnamn men inte tillräckligt många för att hitta rätt version (se kapitel 5.2.2). Versionsgrafen söks igenom för att versioner som inte innehåller någon kolumn som finns med i frågan ska kunna sållas bort och resterande versioner slå ihop i en union. Antag att följande fråga av frågetyp 3 ställs mot tabellen Personregister:

Hämta alla kolumner från tabell Personregister där Lön är mindre än 25000.

Sökningen kommer i detta fall att hitta två versioner som kan vara den rätta att ställa frågan emot. Dessa versioner är version V2 och version V4. Versionerna V2 och V4 slås samman i en union vilken fråga ställs emot. Denna union ser ut på följande sätt:

{Personnummer, Namn, Lön, Arbetsplats, Titel}

Version V4, den version som frågan var tänk att ställas emot, är en delmängd av den union av versionerna V2 och V4 som frågan ställs emot, vilket innebär att svaret blir korrekt.

6.2.2 Ändra datatyp för kolumn

När datatypen för en kolumn ändras så skapas en ny nod som innehåller en kolumn som har samma namn som föregående kolumn men som använder den nya datatypen (se kapitel 5.1.2). Detta gör att det finns två noder som innehåller olika kolumner fast med samma namn. Den ansats som sökmetoden bygger på använder kolumnnamnen i frågan och matchar dessa med kolumnerna i de olika versionerna av tabellen. Detta medför att sökmetoden inte kan skilja kolumnerna åt om de har samma namn och därmed kan sökmetoden inte skilja versionerna av tabellen åt. För att hitta rätt version att ställa en fråga emot måste det anges i frågan vilken av kolumnerna som ingår i frågan och därmed uppnås inte versionstransparens.

Antag att följande fråga ställs mot version V2 av tabellen Personregister som finns i versionerna V1,..,V5 (Figur 9 visar hur versionsgrafen för tabellen ser ut):

> Hämta kolumnerna Namn och Lön från tabell Personregister där Arbetsplats är lika med volvo.

Sökningen efter rätt version att ställa frågan emot går till på samma sätt som i första exemplet i kapitel 6.2.1. Däremot så uppstår ett problem när barnnoderna till noden som innehåller kolumnen Lön ska kontrolleras. Det finns två barnnoder som innehåller en kolumn Arbetsplats, vilket innebär att modellen, som tagits fram i detta arbete, inte kan skilja på kolumnerna. Därmed kan inte versionstransparens uppnås om det finns kolumner med samma namn i versionsgrafen för en tabell.

6.2.3 Subrötter

En subrot fungerar som en rotnod (se kapitel 5.1.2). Enda skillnaden är att alla noder, oavsett var i versionsgrafen som noden är placerad, kan den ärva direkt från subroten. De enda kraven på en nod som ärver direkt från en subrot är att den inte får innehålla en kolumn och att den bara får ärva direkt från en subrot. En subrot passeras aldrig när versionsgrafen söks igenom på grund av att den inte ärver från någon annan nod. Noder som ärver direkt från en subrot fungerar som en vanlig nod och tillför bara en kolumn till den version som noden representerar, det vill säga subrotens kolumn. Detta gör att subrötter inte påverkar versionstransparensen hos modellen.

Antag att följande fråga från frågetyp 1 ställs mot tabellen Personregister som finns i versionerna V1,..,V7 (Figur 11 visar hur versionsgrafen för tabellen ser ut):

> Hämta kolumnerna Namn och Telefonnummer från tabell Personregister där Adress = ”Skolgatan 7”.

Tanken är att frågan ska ställas mot version V6. Sökningen går tillväga på samma sätt som i första exemplet i kapitel 6.2.1. Det enda som skiljer exemplen åt är att i detta exempel så ingår en subrot och att två grenar visar sig vara möjliga vägar till den rätta versionen. När barnnoderna till noden som innehåller kolumnen Namn kontrolleras så

hittas två noder som innehåller kolumner från frågan, en som innehåller kolumnen Adress och en som innehåller kolumnen Telefonnummer. Detta innebär att båda grenarna måste sökas igenom. Noden med kolumnen Telefonnummer har inga barnnoder och innehåller inte själv alla kolumnerna i frågan, vilket upptäcks när kontrollen av barnnoderna sker. Den rätta versionen kan alltså inte finnas i den grenen. När barnnoderna till noden som innehåller kolumnen Adress kontrolleras så hittas en barnnod som innehåller kolumnen Telefonnummer. Versionen som representeras av denna nod är version V6, vilket innebär att rätt version har hittats.

Frågor av frågetyp 2 hanteras på samma sätt som beskrivs i kapitel 6.2.1 eftersom hanteringen av dessa frågor bara involverar användarversionerna. Även frågor av frågetyp 3 hanteras på samma sätt som beskrivs i kapitel 6.2.1. När en fråga av frågetyp 3 ställs så söks hela versionsgrafen igenom. Detta görs eftersom om subrötter finns i versionsgrafen så innebär detta att en och samma kolumn kan finnas i flera grenar. Följande exempel illustrerar denna effekt som subrötter ger och hur detta hanteras.

Antag att följande fråga ställs mot version V6 av tabellen Personregister som finns i versionerna V1,..,V7 (Figur 11 visar hur versionsgrafen för tabellen ser ut):

> Hämta alla kolumner från tabell Personnummer där Telefonnummer = ”08-123456”.

Om sökningen skulle ske som för frågor av frågetyp 1 så skulle det i detta exempel innebära att sökningen stannar på noden som representerar användarversionen V3. Versionen V6, som är den versionen som frågan var tänkt att ställas emot, är inte en delmängd av versionen V3, vilket innebär att svaret inte blir korrekt på grund av att svaret inte innehåller rätt kolumner. Om sökningen däremot sker på rätt sätt, det vill säga hanteras som frågor av frågetyp 3 ska hanteras enligt sökmetoden, så hittas tre versioner som kan vara den korrekta. Dessa versioner är versionerna V3, V6 och V7. Frågan ställs mot en union av dessa versioner. V6 är en delmängd av denna union, vilket innebär att svaret blir korrekt.

6.3 Slutsats

Den undersökning som genomförts i detta arbete skulle undersöka för vilka fall som versionstransparens uppnås (se kapitel 3.1). De fall som undersöktes i detta arbete var:

- Skapa tabell
- Ta bort tabell
- Lägga till kolumn till tabell
- Ta bort kolumn från tabell
- Ändra datatyp för en kolumn

Resultatet av denna undersökning visar att versionstransparens uppnås för alla fallen utom ett. Det fall i vilket versionstransparens inte uppnås är ändringar av datatyp för en kolumn. Anledningen till att versionstransparens inte uppnås i detta fall är att när

datatypen för en kolumn ändras skapas en ny version av kolumnen eftersom ingen data får förloras när en ändring av databasschemat sker i en databas med stöd för schema versioning. Den modell som tagits fram i detta arbete stödjer inte versioner av kolumner på grund av att kolumnnamn används för att hitta rätt version att ställa en viss fråga emot.

7 Diskussion

I detta kapitel kommer en diskussion att tas upp om arbetet som bedrivits i detta projekt. I samband med detta kommer situationen och utvecklingen inom problemområdet och hur detta påverkat detta arbete att diskuteras. I slutet av kapitlet kommer de erfarenhet som inhämtats från detta arbete att diskuteras.

7.1 Diskussion kring arbetet

Utvecklingen av schema versioning har ofta fokuserats på hur versioner och förändringar av databasschemat ska hanteras. En typ av frågor kring schema versioning som ofta ignoreras är hur denna hantering av versioner fungerar gentemot användaren av DBMS:et. Exempel på dessa frågor är vilket sätt att identifiera versioner som är lätta för användaren att hantera eller hur frågor mot en databas med stöd för schema versioning ska se ut. Ett exempel på ett arbete som lägger fokus på den tekniska delen av versionshanteringen är Rashid och Sawyer (2000). Rashid och Sawyer (2000) har tagit fram en modell för schema versioning för objektorienterade databaser som kan hantera versioner av både objekt och klasser. Trots att modellen har implementerats så tar Rashid och Sawyer (2000) inte upp något om hur versionshanteringen fungerar gentemot användaren. Till skillnad från Rashid och Sawyer (2000) så har detta arbete haft fokus på versionshanteringen ur ett användarperspektiv.

Det motto som har varit utgångspunkt i detta arbete är att versionshanteringen inte ska påverka användaren, det vill säga målet har varit att uppnå versionstransparens. Tanken var att beteendet på den modell som togs fram i detta arbete skulle vara samma som för ett DBMS med stöd för schemaevolution. Jensen och Böhlen (2001) har tagit fram en lösning för schema versioning i relationsdatabaser som uppnår versionstransparens. Deras lösning bygger på att alla schemaförändringar har krav kopplade till dem. Kraven avgör vilken version som ska användas för att lagra en viss mängd data. Detta sätt att hantera schemaförändringar kallas *conditional schema change*. Frågor ställs mot alla versioner som innehåller alla de kolumner som ingår i frågan. En potentiell nackdel med denna lösning är att den kanske inte kan appliceras i alla situationer på grund av att förändringarna hanteras på ett särskilt sätt. Detta är en anledning till att modellen som tagits fram i detta arbete ska bete sig som det DBMS som har stöd för schemaevolution.

Det finns dock situationer då det finns ett intresse för att användaren ska kunna ange vilken version en viss fråga ska ställas emot. Temporala databaser används ofta i dessa fall eftersom evolution ofta ses som förändring över tid. En temporal databas hanterar inte bara nuvarande data i databasen utan lagrar också data från det förflutna och data som ska användas i framtiden (Wei & Elmasri, 2000). De Castro, et al. (1997) har utvecklat en modell för schema versioning i temporala relationsdatabaser. All data i databasen och alla versioner av schemat datummärks så att det går att avgöra när den är aktuell. När en fråga ska ställas mot en specifik version i den modell som De Castro, et al. (1997) har tagit fram anges även ett datum. Genom att jämföra detta datum med datumen för när versionerna var aktuella så kan man få fram rätt version. Fördelen med att realisera schema versioning med hjälp av en temporal

databas är att hanteringen av versioner blir lättare för användaren på grund av att identifiera en version med datumet för till exempel då versionen skapades är mindre abstrakt än att identifiera versionen med ett index eller motsvarande. Nackdelen är dock att evolutionen av databasschemat bara kan ske linjärt, det vill säga det går inte att förändra två olika versioner oberoende av varandra. Detta beror på att två versioner inte ska kunna identifieras med samma datum. Det är delvis på grund av detta som temporala databaser inte användes i detta arbete. En annan anledning för att temporala databaser inte användes är att de inte ger några fördelar när versionstransparens ska uppnås.

7.2 Erfarenheter från arbetet

I detta arbete har två större misstag begåtts. Det första är att ingen förundersökning bedrevs i början av arbetet, det vill säga att den tänkta lösningen verifierades så att det var säkert att den var lösbar och att den verkligen löste problemet. Detta ledde till att en del kompromisser fick göras, så som att frågeställningen fick ändras drastiskt. Lösningen granskades i samband med implementationen, vilket innebar att det inte fanns tid att göra ändringar i den. Anledningen till att detta påbörjades så sent berodde delvis på nästa misstag. Det andra misstaget var att förstudien, det vill säga att läsa in sig i problemområdet inte påbörjades förrän arbetet med projektet var igång. Detta gjorde så att tiden till förstudien fick tas från andra moment i projektet. Dessa misstag hade lätt undvikts om förarbetet inför detta projekt hade påbörjats i god tid.

7.3 Framtida arbeten

Detta arbete har inte inriktat sig på att ta fram en fullständig lösning, vilket innebär att det finns många aspekter och problem som inte tagits upp i undersökningen som gjordes i detta arbete.

Till exempel så hanterar den modell, som togs fram i detta arbete, bara de vanligaste och enklaste förändringarna av databasschemat. Ett arbetsuppslag är att utöka den befintliga modellen så att den kan hantera fler typer av ändringar av databasschemat.

Undersökningen som gjordes i detta arbete visade att versionstransparens inte uppnås efter det att datatypen för en kolumn har ändrats. De slutsatser som gjorts på resultatet av undersökningen visar att detta beror på att modellen inte kan hantera versioner av kolumner på ett sådant sätt så att versionstransparens uppnås. Därmed skulle ett arbete omfatta utvecklingen av ett kolumnversionshanteringsstöd för modellen som uppnår versionstransparens.

Ytterligare en avgränsning som gjorts är att modellen erbjuder versionstransparens när förändringar av databasschemat görs. För närvarande måste version anges när databasschemat ändras. Därmed är det inte lämpligt att låta databasapplikationer göra ändringar i databasschemat eftersom databasapplikationen då måste veta vilken version den arbetar emot. Detta gör att den versionstransparens som erbjuds när frågor ställs mot databasen blir överflödig. Ett framtida arbete skulle därmed kunna vara att utveckla modellen så att versionstransparens erbjuds vid förändringar av databasschemat.

Referenser

Berndtsson, M., Hansson, J., Olsson, B. & Lundell, B. (2002). *Planning and implementing your final year project with success*. Gateshead: Springer-Verlag London Limited.

De Castro, C., Grandi, F. & Scalas, M. R. (1997). Schema versioning for multitemporal relational databases. *Information Systems, 22*, 249-290. [Elektronisk version] Tillgänglig på Internet: http://www.sciencedirect.com [Hämtad 03.02.06].

Elmasri, R. & Navathe, S. (2000). *Fundamentals of database systems* (Third edition). USA: Addison-Wesley.

Jensen, O. G. & Böhlen, M. H. (2001). Evolving relations. *Lecture Notes in Computer Science, 2065, 115-132*. [Elektronisk version]. Tillgänglig på Internet: http://link.springer.de [Hämtad 03.03.26].

Monk, S. (1993). A *Model for Schema Evolution in Object-Oriented Database Systems*. Doktorsavhandling. Computer Department, Lancaster University. [Elektronisk version]. Tillgänglig på Internet: http://www.comp.lancs.ac.uk [Hämtad 03.02.06].

Peters, R. J. & Özsu, T. M. (1997). An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Transactions on Database Systems, 22*, 75-114. [Elektronisk version]. Tillgänglig på Internet: http://citeseer.nj.nec.com [Hämtad 03.01.30].

Rashid, A. & Sawyer, P. (2000). *Towards "database evolution" –a taxonamy for object oriented databases*. Lancaster University. Tillgänglig på Internet: http://www.comp.lancs.ac.uk/computing/reseach/cseg/00_rep.html [Hämtad 03.01.30].

Roddick, J. F. (1995).. A survey of schema versioning issues for database systems. *Information and Software Technology, 37, 383-393*. [Elektronisk version]. Tillgänglig på Internet: http://www.sciencedirect.com [Hämtad 03.03.26].

Rosen, K. H. (1999). *Discrete mathematics and its applications* (Fourth edition). Singapore: McGraw-Hill.

Wei, H. & Elmasri, R. (2000). Schema evolution and database conversion techniques for bi-temporal databases. *Annals of Mathematics and Artificial Intelligence, 30*, 23-52. [Elektronisk version]. Tillgänglig på Internet: http://www.kluweronline.con [Hämtad 03.02.18].