Databaskonstruktion

1 Uppgift 1 Databasimplementation

1.1 delfråga a)

Kodifiering är en denormaliseringsteknik där man byter ut en återkommande textsträng mot en siffra som representerar strängen i stället, för att hämtningen av en potentiellt stor mängd data ska ske snabbare (Bjurén, 2023). Tanken är att det går fortare att hämta en siffra gentemot en lång rad textsträng.

```
CREATE TABLE material (
    id          INT PRIMARY KEY,
    material    VARCHAR(50),
    komponent   VARCHAR(50)
);

CREATE TABLE hus (
    adress      VARCHAR(50),
    plats       VARCHAR(50),
    kategori    VARCHAR(50),
    längd       INT,
    bredd       INT,
    materialid  INT,
    kostnad     INT,
    FOREIGN KEY (materialid) REFERENCES material(id)
);

CREATE TABLE takmaterial (
    idnr        INT PRIMARY KEY,
    namn        VARCHAR(50),
    tyngd       INT,
    tillverkare INT,
    material    INT,
    FOREIGN KEY (material) REFERENCES material (id)
);
```

Figur 1 Tabeller material, hus och takmaterial

I en databas med samma tabeller som i figur 1 finns det flera kolumner med datatypen VARCHAR som kan kodifieras. I detta fall hade de lämpligaste kolumnerna att kodifieras varit **plats** och **kategori** i tabellen *hus*, och kolumnerna **namn** och **tillverkare** i tabellen *takmaterial*. Anledningen för att de andra kolumnerna inte är lämpliga att kodifieras är för att de redan är av datatypen integers eller antagligen redan är egna tabeller, exempelvis är material ett flervärt attribut och möjligtvis har en egen tabell redan.

De nämnda kolumnerna har dock strängar med stor potential att återkomma. För denna databas antas det att det är ett företag som har en databas för arbete på tak, i det fallet är det väldigt liten chans att ett hus behöver göra om sitt tak flera gånger, däremot kan platsen eller kategorin av huset återkomma flera gånger.

Detsamma gäller **namn** och **tillverkare** för takmaterialtabellen. För denna databas hade jag valt att kodifiera **tillverkare** eftersom det känns som en rimlig kategori av data som återkommer.

```
CREATE TABLE tillverkare (
    id          INT PRIMARY KEY,
    företag     VARCHAR(50)
);

CREATE TABLE takmaterial (
    idnr        INT PRIMARY KEY,
    namn        VARCHAR(50),
    tyngd       INT,
    tillverkare INT,
    material    INT,
    FOREIGN KEY (material) REFERENCES material (id),
    FOREIGN KEY (tillverkare) REFERENCES tillverkare (id)
);
```

Figur 2 (VG) Kodifiering av kolumn ’tillverkare’ i tabell ’takmaterial’.

Oftast gör man många köp från ett och samma ställe, därmed kan man anta att flera olika material köps in från samma tillverkare. Om databasen skulle utökas med flera tabeller, exempelvis med en inköpstabell eller annat, kan finns det en stor chans att samma typ av tillverkare uppkommer även i dessa, kanske för att dokumentera betalningen av kostnaderna för tillverkning av materialet eller andra skäl.

I min inlämning gjorde jag en kodifiering av **farlighet** precis för de potentiella scenarion som kan komma att uppstå för kolumnen **tillverkare** men med den skillnaden att jag visste från början att det är en kolumn som skulle finnas med i flera olika andra tabeller och hade strängar som var återkommande.

1.2 delfråga b)

Fördelen med procedurer när det kommer till inmatningar är att de är säkrade på två olika sätt, först och främst kan de minska risken att en obetrodd användare kommer åt känsliga data och manipulerar den på ett sätt som är oönskat (Bjurén, 2023), men en annan effekt är också att en logisk och välkonstruerad procedur kan utföra queries mot databasen på ett ”korrekt och sammanhängande” sätt, det vill säga, att exempelvis refererade tabeller innehåller den data som takmaterial pekar mot.

Procedurer är ett bra sätt för att standardisera addering eller annan förändring av data eftersom det kan säkerställa att all dataförändring följer de regler och constraints som finns i databasen, eftersom takmaterial har andra tabeller som den refererar till. Dessutom kan det vara bra för att logga ändringar om något går snett, då finns den en historik som man kan se tillbaka på om situationen kräver det. Detta kan jämföras med proceduren i min inlämning, där skapade jag en procedur som heter ”hemligstämpla_på_ras_namn” där jag ändrar informationen i vissa rader i en tabell. Ändringen loggas även så det går att ”ångra” eller återskapa den informationen som har ändrats.

I proceduren loggas all information om rasen samt alla aliens som har rasen i sin profil. Dessutom ändras deras information i raskolumnen från rasens namn till ”Hemligstämplat”. Allt detta sker automatiskt med proceduren och en användare behöver inte själv göra flera ändringar manuellt och riskera att missa något eller göra en feländring.

1.3 delfråga c)

Att använda denormaliseringstekniken horisontell split på takmaterial kan vara lämplig eftersom man då kan reducera mängden data som ska hämtas vilket vidare kan leda till en snabbare hämtning samt en mer responsiv upplevelse för användaren (Bjurén, 2023). Det betyder att de kolumner som efterfrågas mer sällan kan flyttas till en separat tabell som hämtas endast när det behövs. Det är mer tolererbart att den större datan som man efterfrågar mer sällan tar längre tid än att den som efterfrågas ofta tar lång tid.

För takmaterial betyder det att den data som efterfrågas mest också gynnas mest av en horisontell split, av dessa skulle det kunna exempelvis tyngd för att veta hur stor belastning som takmaterialet kommer att vara vid transport, vilket kommer att ske vid varje arbetsuppdrag, eller komponenterna som materialet består av. Materialet kanske behöver vara kompatibla med huset som arbetet ska ske på. Av dessa anledningar hade jag haft kvar just dessa kolumner och plockat ut kolumnen **tillverkare** och satt denne i en annan tabell. Namnet för takmaterialet är bra att ha kvar i tabellen för att det är lättare för en person att prata om och söka på, än om en vag siffra. I stället kan namnet gynnas av att indexeras.

1.4 delfråga d)

Inom databaskontexten är en vy en virtuell tabell som egentligen inte är tabeller, utan är en reflektion av data i en tabellformat för en användare av databasen (Bjurén, 2023), det vill säga det är tabeller som är skapade av färdigskrivna queries. Exempelvis kan det användas för att ”spara” queries med specialiserade data som färdiga tabeller så att man slipper göra egna komplexa queries. Det kan även användas för att kontrollera privilegier och därmed öka säkerheten för databasen.

För att illustrera min poäng tar jag upp ett exempel: I min databasinlämning fanns det vissa aliens som var hemligstämplade i en tabell som inte alla skulle få ha tillgång till, då skapade jag en vy med enbart ”offentliga” aliens med specialiserade data som inte innehöll hemligstämplade information. På så sätt kan obehöriga inte komma åt säkerhetsklassade eller hemlig information.

1.5 delfråga e)

Härledda attribut är attribut som kan beräknas eller härledas från andra attribut i en databas. Att implementera härledda attribut innebär att säkerställa att dessa attribut beräknas, lagras och uppdateras korrekt när de relaterade basattributen ändras. Två tekniker som kan användas för att implementera ett härlett attribut i en databas är genom views.

Views är som sagt virtuella tabeller som kan innehålla komplexa queries som beräknar de härledda attributen baserat på basattributen i en eller flera underliggande tabeller. Vyer ger ett sätt att redovisa de härledda attributen som om de var lagrade i databasen, utan att faktiskt lagra de beräknade värdena.

Ett annat sätt att implementera härledda attribut är genom att använda sig av triggers. Triggers kan användas för att automatiskt utföra vissa åtgärder, såsom beräkningar, när specifika händelser inträffar i databasen. Genom att definiera triggers på basattributen kan man säkerställa att de härledda attributen uppdateras när basattributen ändras. Utlösare kan användas för att upprätthålla konsistensen av de härledda attributen i realtid. Detta är något

som jag även använder mig av i min inlämning, där jag uppdaterar en horisontellt splittad tabell på Vapen när Vapentabellen ändras, exempelvis vid radering av vapen eller uppdatering av vapenägare.

2 Uppgift 2 Applikation med PHP

2.1 delfråga a)

För att hantera olika scenarier av inserts kan man använda sig av villkor i php. Den enklaste hanteringen är en if-else sats som kontrollerar huruvida kategorin för huset har matats in eller inte genom att kombinera en if() med en isset().

```
<?php
    $adress = $_POST['adress'];

    if (isset($_POST['kategori'])) {
        $kategori = $_POST['kategori'];
    }
    else {
        $kategori = null;
    }

    // Kör olika inserts beroende på användarens inmatning
    if ($kategori != null) {
        $query = "INSERT INTO hus (adress, kategori) VALUES ('$adress',
'$kategori')";
        // code
    }
    else {
        $query = "INSERT INTO hus (adress) VALUES ('$adress')";
        // code
    }
?>
```

Figur 3 Exempelkod på två olika inserts baserat på användarinmatning.

I kodexemplet från figur 3 fångas innehållet från kategori i variabeln $kategori för att sedan kontrolleras om den har blivit satt. Om den är satt så sätts variabeln till det som har matats in, annars sätts variabeln till null. En ny if-else sats definieras och där körs de olika queries beroende på variabeln.

2.2 delfråga b)

Skillnaden mellan hanteringen av länkar och formulär är att länkar används oftast för att skicka GET-requests för att *hämta data från servern*, medan för ett formulär *skickas data till en server* via en POST-request.

```
<a href="page.php?param1=value1&param2=value2">Länk</a>

<?php
$param1 = $_GET['param1'];
$param2 = $_GET['param2'];
echo "Param1: $param1, Param2: $param2";
?>
```

Figur 4 Länk och hantering av GET-requesten

När en användare klickar på en länk, skickas informationen som en del av URL:en, vilket gör att data syns i adressfältet. I exemplet från figur 4 visas hur data hanteras och hur parametrarna skickas via URL:en i länken.

```
<form action="page.php" method="post">
    <input type="text" name="input1">
    <input type="text" name="input2">
    <input type="submit" value="Submit">
</form>

<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $input1 = $_POST['input1'];
    $input2 = $_POST['input2'];
    echo "Input1: $input1, Input2: $input2";
}
?>
```

Figur 5 Formulär och hantering av POST-requesten

Formulär används för att skicka inputs från användaren i olika fält, exempelvis textfält, radioknappar, eller rutor som de bockar i. Data som skickas via POST är inte synlig för användaren och visas inte heller i URL:en som den gör för länkar. Kodexemplet från figur 5 visar hur data hanteras via en POST-förfrågan.

Det är alltså skillnaden i hur överföringen av data sker.

2.3 delfråga c)

Fördelarna med att använda PDO och att binda input från webbsidan till databasen är flera. Bland annat kan man erhålla högre säkerhet genom att förhindra SQL-injections då förberedda queries och bindning av variabler förhindrar att skadlig SQL-kod passeras in genom användarskrivna queries. Webbsidan kan även få en uppfattad upplevelse om bättre prestanda genom att återanvända kompilerade SQL-frågor då förberedda queries kan kompileras och återanvändas, vilket kan leda till bättre prestanda, särskilt vid upprepade databasaccesser.

```
$adress = $_POST['adress'];
$kategori = $_POST['kategori'];

$stmt = $pdo->prepare("INSERT INTO hus (adress, kategori) VALUES (:adress,
:kategori)");
    $stmt->bindParam(':adress', $adress);
    $stmt->bindParam(':kategori', $kategori);
    $stmt->execute();
```

Figur 6 Kodexempel på bindning av variabler för att lagra ett hus i databasen.

2.4 delfråga d)

Fördelen med att bygga en applikation med en enda php fil är att en fil kan vara mer hanterlig för mindre applikationer med begränsad funktionalitet. Det blir även mindre att ha koll på, mindre underhåll och felsökningen kan förenklas då det enbart är en fil som man behöver debugga.

Det finns dock många nackdelar med att bygga en applikation på enbart en fil. Bland annat kommer skalbarheten att lida mycket. Med större applikationer tillkommer även mer komplexitet, då kan det vara en fördel att ha flera filer och ett mer objektorienterat arbetssätt under utvecklingen av applikationen.

I den situationen är det en fördel att kunna dela upp applikationen i flera filer så att man kan skapa en bättre struktur med hjälp av uppdelning av olika funktioner och ansvarsområden, vilket vidare kan underlätta förståelsen och underhållet av koden.

2.5 delfråga e)

```
<form action="radera_takmaterial.php" method="post">
    <input type="hidden" name="takmaterial_id" value="1">
    <input type="submit" value="Radera Takmaterial">
</form>
```

Figur 7 Kodexempel för hur man skapar en dold input.

I kodexemplet i figur 7 finns ett input element av typen ”hidden”. Ett ”hidden” eller gömt fält låter utvecklaren att inkludera osynliga data i formulär som användaren inte kan ändra på. Användningen av detta är oftast för att lagra information om vilken databaspost som behöver uppdateras. Det finns dock en hake med detta, och det är att användaren fortfarande har möjlighet att se den gömda informationen genom att inspektera element eller källkoden med webbläsarens utvecklarverktyg (vanligtvis F12).

3 Uppgift 3 Applikation ASP.NET MVC

3.1 delfråga a)

Min uppfattning om att programmera procedurellt i php är att det först och främst är ett paradigm där koden körs ”uppifrån och ner”, det vill säga att det körs lite som en instruktionsbok där man börjar från sida 1 och går stegvis genom boken till slutet, medan MVC är ett designmönster som delar upp applikationerna i 3 delar, modell, vy och kontroll, precis som MVC är en förkortning av. Jag skulle vilja säga att MVC är en mer modulärt utvecklingsstruktur i jämförelse, där procedurellt kan ses som en linjär.

Styrkan i procedurell programmering är att för nybörjare är det enkelt att ta till sig. Att tänka sig att man skriver instruktioner som ska köras stegvist. Det gör det snabbt och enkelt att bygga mindre applikationer och det är ganska straight forward.

I och med enkelheten med procedurell programmering leder detta även till en begränsad skalbarhet och vidareutveckling. Återigen blir större applikationer också snabbt mer komplexa, vilket kan göra det svårt att utveckla och hantera procedurellt. Svagheten är väl att i en instruktionsbok bör allt vara ”glasklart”, vilket även gör det svårare att skapa abstrakta modeller.

Styrkan i MVC är många, varav några är redan nämnda. Att kunna separera tydliga ansvarsområden för datahantering, vyer och hantering av interaktioner underlättar både underhållet och organiseringen av utvecklandet av applikationen. MVC är modulärt, vilket gör att man kan bygga flera små delar och återanvända samma delar på olika ställen i applikationen.

Svagheter som finns med MVC är en mycket högre inlärningskurva på grund av komplexiteten med en struktur som är mer ingående. Paradigmet kan anses vara överkomplicerad med så många delar och abstrakta modeller.

Jag skulle säga att jag har en lättare tid att förstå och skriva procedurell kod eftersom det paradigmet som sagt är väldigt enkel och step-by-step liknande i sitt utförande, men jag strävar efter att bli bättre på MVC strukturen för att bli en bättre utvecklare som kan förstå mer komplexa applikationer med mer djup i sig. Genom att bli bättre och kunna bygga modulärt kan man förenkla utvecklandet, eftersom koden kan återanvändas mycket mer. Det blir även enklare att debugga koden eftersom man kan leta efter fel i specifika områden av koden, då den är uppdelad. När ett fel uppstår kan man även åtgärda dessa i de mindre relaterade modulerna eller komponenterna i stället för att gå igenom hela projektet.

3.2 delfråga b)

Kopplingen som sker när ett nytt hus med data från textinputs läggs till börjar när datan tas emot i användargränssnittet som är mappat till en ViewModel.

```
@model Hemtentamen_Databaskonstruktion.ViewModels.HusViewModel

<form method="post" action="AdderaHus">

        <label>Adress:</label><input type="text" id="Adress" name="Adress"
required>
        <label>Plats:</label><input type="text" id="Plats" name="Plats">
        <label>Kategori:</label><input type="text" id="Kategori" name="Kategori">
        <label>Längd:</label><input type="number" id="Längd" name="Längd">
        <label>Bredd:</label><input type="number" id="Bredd" name="Bredd">
        <label>MaterialId:</label><input type="number" id="MaterialId"
name="MaterialId">
        <label>Kostnad:</label><input type="number" id="Kostnad" name="Kostnad">

    <button type="submit">Lägg till Hus</button>
</form>
```

Figur 8 Exempel på en ViewModel.

Denna modell skickas sedan vidare tillsammans med en request till controllern. Controllern gör en slags översättning från ViewModellen till en HusModel.

```
namespace Hemtentamen_Databaskonstruktion.Models
{
    public class Hus
    {
        public string Adress { get; set; }
        public string Plats { get; set; }
        public string Kategori { get; set; }
        public int Längd { get; set; }
        public int Bredd { get; set; }
        public int MaterialId { get; set; }
        public int Kostnad { get; set; }
    }
}
```

Figur 9 Exempel på HusModel klass.

HusModel är en klass som definierar vad ett hus är, så den innehåller alla properties som sedan sätts till det som skickades från ViewModellen, det vill säga, om ett formulär fylldes i

med gatuadress, plats och kategori så kommer dessa värden att sättas in som properties.

```
[HttpPost]
        public IActionResult AdderaHus(HusViewModel husViewModel)
        {
            if (ModelState.IsValid)
            {
                Hus hus = new Hus
                {
                    Adress = husViewModel.Adress,
                    Plats = husViewModel.Plats,
                    Kategori = husViewModel.Kategori,
                    Längd = husViewModel.Längd,
                    Bredd = husViewModel.Bredd,
                    MaterialId = husViewModel.MaterialId,
                    Kostnad = husViewModel.Kostnad
                };

                return RedirectToAction("Success");
            }
            return View("Index", husViewModel)
        }
```

Figur 10 Exempel på en HusController klass.

Med modellen kan controllern sedan anropa en databas och lägga till det nya huset och informationen som matades in av användaren.

3.3 delfråga c)

```
public IActionResult Search(string searchString)
        {
            List<Takmaterial> searchResult = new List<Takmaterial>();

            using (SqlConnection connection = new
SqlConnection(_connectionString))
            {
                connection.Open();
                SqlCommand command = connection.CreateCommand();
                command.CommandText = "SELECT * FROM Takmaterial WHERE Namn LIKE
@SearchString OR Tillverkare LIKE @SearchString";
                command.Parameters.AddWithValue("@SearchString", "%" +
searchString + "%");

                SqlDataReader reader = command.ExecuteReader();

                while (reader.Read())
                {
                    Takmaterial takmaterial = new Takmaterial
                    {
                        Idnr = reader["Idnr"].ToString(),
                        Namn = reader["Namn"].ToString(),
                        Tyngd = reader["Tyngd"].ToString(),
                        Tillverkare = reader["Tillverkare"].ToString(),
                        Material = reader["Material"].ToString()
                    };
                    searchResult.Add(takmaterial);
                }
            }
            return View("SearchResult", searchResult);
        }
```

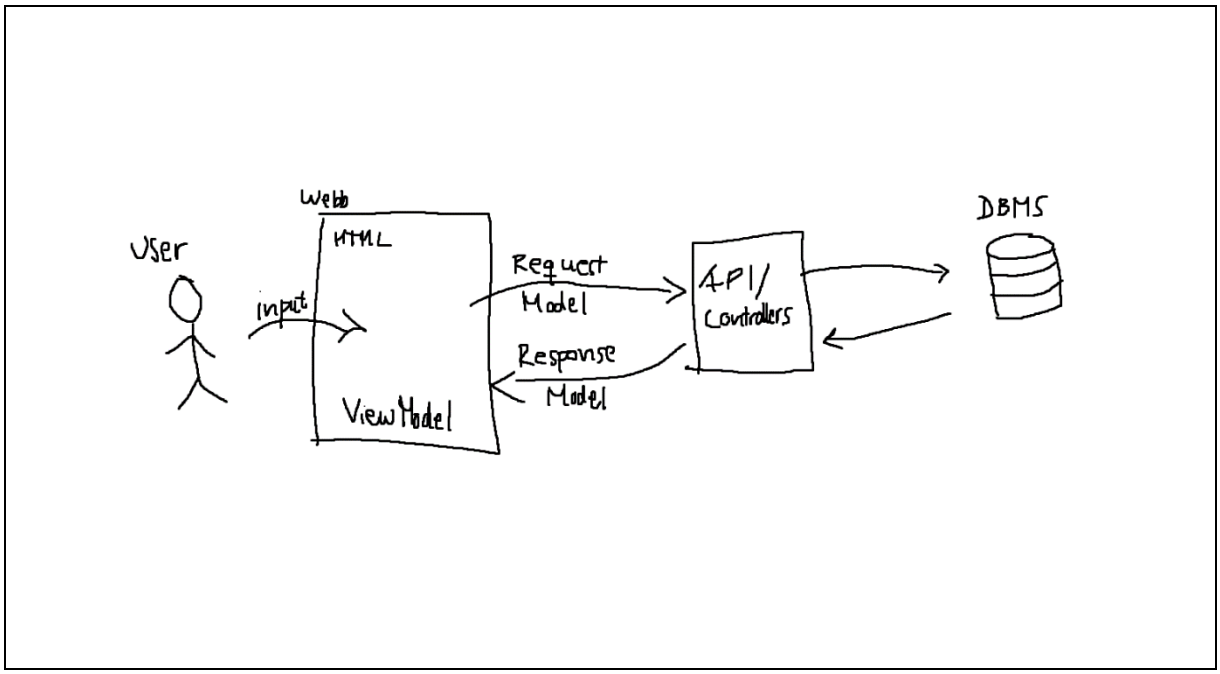
Figur 11 Exempelkod på hur sökning på något fält i material-tabellen.

3.4 delfråga d)

```
<table class="table">
    <thead>
        <tr>
            <th>Idnr</th>
            <th>Namn</th>
            <th>Tyngd</th>
            <th>Tillverkare</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.Idnr</td>
                <td>@item.Namn</td>
                <td>@item.Tyngd</td>
                <td>@item.Tillverkare</td>
            </tr>
        }
    </tbody>
</table>
```

Figur 12 Exempelkod på TakmaterialViewModel utan fältet ’material’.

3.5 delfråga e)



Figur 13 Dataöverföring mellan de olika MVC-komponenterna.

En användare interagerar med en webbsida som innehåller olika element i forms, exempelvis input element eller länkar. Vad användaren ser är en vy som är skapad för att kunna interagera med databasen. När formsen skickas får controllern datan som användaren har matat in. Controllern är ansvarig för att kommunicera med databasen. Det finns olika controllers för olika behov. För att göra en liknelse kan man tänka på en restaurang.

Om du vill få sittplatser så pratar du med en restaurangvärd. Värden är en typ av controller. Vill du i stället beställa mat så pratar du med en servitris, hon är en annan controller. En bartender är en annan controller som ger dig drinkar. Så varje controller har en viss roll som den ska utföra. Model eller modellen är en representation av den data som är lagrad. I vårt exempel är sittplatserna en typ av model, maten en annan och även drinkarna. Controllern mappar användarens request till den specifika databastabellen.

Databasen är där den data som efterfrågas är lagrad, det är även där som ny data kan tillsättas och lagras.

Referenser

Bjurén, Johan. (2023) Codes. [Video]. https://his.instructure.com/courses/6983/pages/databas [2023-08-28]

Bjurén, Johan. (2023) Stored procedures. [Video]. https://his.instructure.com/courses/6983/pages/databas [2023-08-28]

Bjurén, Johan. (2023) Horizontal split. [Video]. https://his.instructure.com/courses/6983/pages/databas [2023-08-28]

Bjurén, Johan. (2023) Views. [Video]. https://his.instructure.com/courses/6983/pages/databas [2023-08-28]