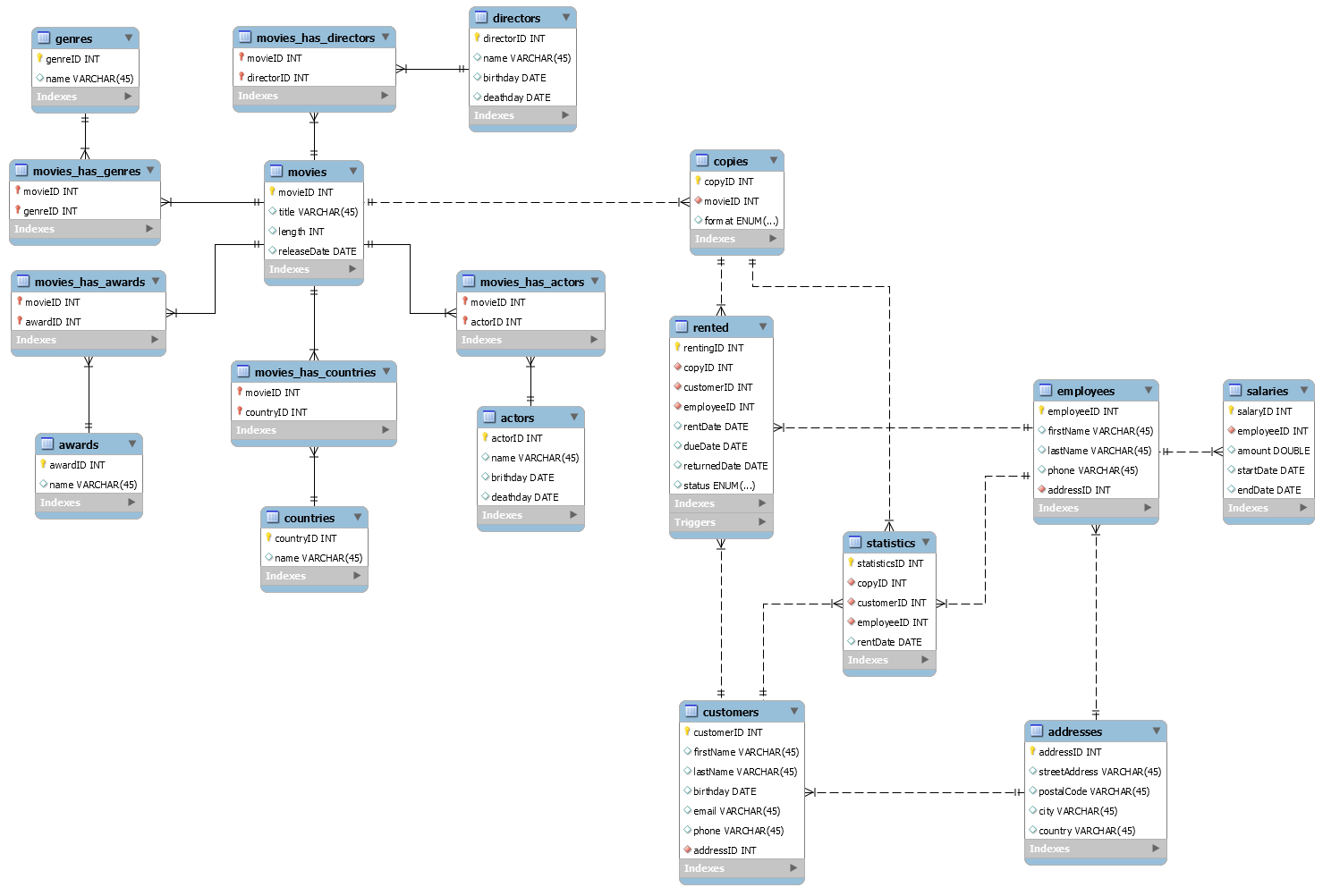
Slutprojekt

Informationsarkitektur och databasutveckling

I det här dokumentet kommer jag att förklara och motivera de val jag gjort när jag designat och skapat min databas. Jag kommer att ta upp vilka fördelar som jag anser att min design har, samt om den medför några nackdelar, och dessutom förklara vilken storage engine jag har valt och varför.

# Fördelar och nackdelar med min design

Innan jag förklarar vad som är bra och mindre bra med min design kommer jag att förklara vad min design är, hur jag har valt att strukturera min modell och databas. Nedan finns en bild på min modell, det vill säga alla entiteter (tabeller), deras attribut (kolumner), samt relationerna mellan dessa (främmande nycklar).



Vid första anblick ser det ut som två block av olika sammankopplade tabeller som länkas samman av en relation. Detta är för att jag har valt att på ena sidan (vänster på bilden) samla all information om filmerna, i en mer abstrakt bemärkelse. Här sparas en rad per film i tabellen *movies* där varje får ett eget id, en titel, speltid, och releasedatum. Sedan har jag en uppsättning många-till-många-relationer för skådespelare, regissör, genre, land, och utmärkelser, då varje film kan ha noll, en, eller flera av alla dessa (i teorin), och varje skådespelare, regissör, genre, land, och utmärkelse kan förekomma i noll, en, eller flera filmer. I databasen kopplas alla dessa ihop med hjälp av deras respektive id som ligger som primärnycklar. Detta minimerar redundansen och på samma gång ökar databasens integritet.

Den högra delen av bilden innehåller tabeller som har med själva uthyrningen att göra. Till att börja med har vi tabellen *copies* som är ett register över de fysiska kopior av alla filmer som företaget äger. Varje kopia får ett unikt id, och kopplas till vilken film det är med filmens unika id, samt så lagras vilket format den fysiska kopian är i, DVD eller BLU-RAY. På så vis kan företaget äga flera kopior av samma film, utan att behöva spara samma film flera gånger i film-tabellen, till och med i olika format. Varje gång en film hyrs ut läggs en ny rad till i tabellen *rented* där id-numret för vilken fysisk kopia som hyrs sparas, samt id-nummer för kunden som hyr och den anställda som hyr ut. Dessutom sparas dagens datum för när hyrningen sker, och återlämningsdatumet sätts till fyra dagar efter uthyrningsdatumet. Återlämningsdatumet lämnas som null tills det att filmen återlämnas, och en status indikerar även om filmen är uthyrd, återlämnad i tid, eller återlämnad sent. Denna kolumn är av datatypen ENUM med de tillåtna värdena RENTED, ON TIME, och RETURNED LATE. Dessa tre används i min databas, men det finns även ett till tillåtet värde jag lade till från början, nämligen LATE. Min tanke med det var att om databasen faktiskt låg på en server som var online skulle en procedur kunna köras t.ex. en gång per dygn, och uppdatera statusen på uthyrda filmer från RENTED till LATE om de var efter återlämningsdatumet, men jag har inte implementerat detta då det var utanför uppgiftens scope. Denna funktion är förstås egentligen inte nödvändig då man lika gärna kan jämföra datumen, men min tanke var att det skulle underlätta för de anställda att snabbt kunna se vilka filmer som är försenade.

Tabellerna *customers* och *employees* innehåller liknande attribut, med vissa skillnader. Namn och kontaktuppgifter, och ett id som refererar till en adress. *Address* är en egen tabell då flera personer kan bo på samma adress, och att spara dem på det här sättet och sen referera till dem minskar igen redundansen och risken för motsägelser. Jag har sedan valt att spara alla anställdas löner i en separat tabell där jag sparar summan tillsammans med den anställdas id, ett startdatum och ett slutdatum. På så vis, när en anställd får en löneförändring läggs en ny rad till i tabellen med den nya summan, och ett nytt startdatum, medan den gamla aktuella raden får ett slutdatum. Att spara det på det här viset bibehåller en lönestatistik för varje anställd, vilket skulle varit svårare om det sparats direkt i *employees*-tabellen. Till sist finns tabellen *statistics* där nya rader automatiskt läggs till när en uthyrning sker, med en mindre mängd data än *rented*, där man ser vilken film som hyrts av vilken kund, av vilken anställd, samt datum för uthyrningen.

Vad jag anser är bra med min design är att den är strukturerad för att minimera förekomsten av redundant data så långt det är möjligt, och att den samtidigt är enkel att expandera efter behov. Skulle man t.ex. vilja lägga till mer information att koppla till filmerna, t.ex. vilka språk som förekommer i dem, om de tillhör något delat universum med andra filmer, såsom Marvel, DC, osv., vem/vilka som är producenter, eller liknande, så är det bara att lägga till en sådan tabell och göra en ny många-till-många-relation till *movies* och resten av databasen kommer att fortsätta fungera som om inget har hänt, utöver att de views som finns behöver uppdateras för att visa den nya informationen.

Dessutom samlas alla uthyrningar i en enda tabell, vilket underlättar beräkningar. Däremot krävs det fler joins, även för enklare frågor, eftersom mycket av informationen är spridd i olika tabeller. Det går t.ex. inte att se vilken film som hyrts direkt i *rented*-tabellen, utan bara vilken kopia, så då måste man joina till *movies* genom *copies* för att kunna se detta. Men fördelen att kunna äga flera kopior av samma film utan att dubbelspara filmer överväger denna negativa aspekt enligt mig. Denna strukturering har påverkat hur jag skapade mina stored procedures *sp\_rent\_movie* och *sp\_return\_movie*. Vid uthyrning anger man vilken film som ska hyras ut baserat på filmens id, så kommer proceduren att ta reda på om det finns någon ledig kopia av filmen, eller om alla kopior av den valda filmen är uthyrd. Hittar den en ledig kopia kommer den att lägga till den i *rented*-tabellen, annars kommer den meddela att ingen ledig kopia finns. Vid återlämning av film anger man kopians id istället för filmens id, då systemet måste veta exakt vilken kopia det är som kommer tillbaka, till skillnad från vid uthyrningen, då det inte spelar någon roll vilken kopia det är, så länge det är samma film i samma format. Jag valde att tillåta företaget att specificera om kopian är i DVD-format eller Blu-ray-format, då vissa föredrar Blu-ray framför DVD, medan andra kanske bara har en DVD-spelare.

En nackdel med min design och struktur har att göra med hur jag sparar personer. Jag har fyra tabeller som sparar personer av olika slag, *actors*, *directors*, *customers*, och *employees*. Detta medför mindre problem för dataintegriteten. Exempelvis är en av filmerna jag lade till i databasen en dokumentär, där regissören (sparad i en tabell) också förekommer i filmen (som en ”skådespelare”). Detta betyder att samma person är sparad på två olika platser i databasen, vilket skulle kunna orsaka problem om någon sparad data skulle ändras och behöva uppdateras. Dock innehåller dessa två tabeller endast namn, födelsedag, och dödsdag, och dessa tre ändras rätt så sällan. När det kommer till *customer* och *employee* är problemen något större. Databasen tillåter t.ex. inte anställda att hyra filmer, utan de måste då registrera sig som kunder, och då förekomma med i stort sett samma attribut på två platser i databasen, dåligt för lagringsutrymme och dataintegritet. Vad jag egentligen borde ha gjort för att komma runt dessa problem är att ha haft en tabell där alla personer lagrades, skådespelare, regissörer, kunder, och anställda. Därifrån kunde jag sedan koppla deras personliga id till de fyra tabellerna jag har nu, eller möjligtvis kunna koppla dem direkt till t.ex. kopplingstabellerna för regissör och skådespelare. De separata tabellerna för anställd och kund hade varit kvar, då olika extra information behöver sparas om dessa två roller.

Ännu en nackdel med min design och struktur är att jag inte implementerat någon struktur för betalning av uthyrningarna. Jag valde att inte ta med det i min databas, då det finns en så stor mängd sätt att genomföra betalningar, med kreditkort, fakturor, Swish, osv. och det kändes som ett stort tillägg och till viss del utanför uppgiftens scope att skapa en filmuthyrningsdatabas. Dock skulle det troligtvis vara relativt enkelt att implementera detta genom att skapa relevanta betalningstabeller, och sedan koppla in dem i de existerande tabellerna, och möjligtvis till varje uthyrning koppla en instans av en betalning.

# Storage Engine

Den storage engine jag har valt för alla mina tabeller är den som är standard i MySQL just nu, InnoDB. Från det jag har läst om MyISAM verkar det som att det är en snabbare storage engine än InnoDB när det kommer till läs- och skriv-operationer. Utöver det kan jag inte se några fördelar med MyISAM över InnoDB när det komemr till den databas jag har skapat. Om du vill göra en uppdatering i någon tabell i MyISAM låser den hela tabellen tills du är klar med uppdateringen, medan InnoDB endast låser de rader du kommer att påverka, vilket är fördelaktigt om t.ex. flera kunder försöker ändra på sina uppgifter samtidigt, då väntetiderna blir kortare. InnoDBs stöd för transaktioner som MyISAM inte har ser jag ingen större poäng med att använda så som databasen ser ut idag, då inga komplicerade transaktioner sker. Skulle den däremot utökas och implementera någon slags credit-system där man kan ladda på sitt konto med pengar och sedan använda dem för att hyra filmer med skulle det vara en bra idé att göra det med transaktioner. Den största anledningen varför InnoDB är det bästa valet för min databas är för att den stödjer integritet och restriktioner för främmande nycklar. Då min databas är uppdelad i många olika tabeller med mycket relationer mellan dem är det viktigt att de olika främmande nycklarna behåller sina referenser och att man kan välja vad som ska hända när en viss post ska tas bort från en tabell som refereras till i en annan tabell.