
LÄR DIG DATABASER OCH SQL FRÅN GRUNDEN

Adrian Krsmanovic
Robert Westman
William Blennow
Linus Rundberg Streuli
Antonio Prgomet

Första upplagan

Innehållsförteckning

Förord	vii
Bokens GitHub	vii
Bokens syfte och målgrupp	vii
Programvara som används i boken	viii
Bokens utformning	viii
Språk	x
Bokens disposition	xi
1 Introduktion till databaser och SQL	1
1.1 Informationsteknik (IT)	1
1.2 Databaser, databashanterare och SQL	3
1.3 Data och information	4
1.4 Övningsuppgifter	7
2 Databasteori	9
2.1 Tre abstraktionsnivåer för databaser	9
2.2 SQL-språkets kategorisering	10
2.3 Grundläggande teori för relationsdatabaser	11
2.3.1 Tabeller	12
2.3.2 Transaktioner	14
2.3.3 CRUD-flödet	15
2.3.4 Begränsningar	15
2.3.5 Nycklar	16
2.3.6 Tabellrelationer	19
2.4 ACID	21

2.4.1	Atomicity	21
2.4.2	Consistency	22
2.4.3	Isolation	23
2.4.4	Durability	24
2.5	Normalisering	25
2.5.1	Onormaliserad form (UNF)	26
2.5.2	Första normalformen (1NF)	26
2.5.3	Andra normalformen (2NF)	26
2.5.4	Tredje normalformen (3NF)	27
2.6	Optimering	29
2.6.1	Indexering	29
2.6.2	<i>Query</i> -optimering	31
2.6.3	<i>Load</i> -balansering	31
2.6.4	<i>Cache</i> -hantering	32
2.7	Designval	35
2.7.1	STAR	36
2.7.2	Snowflake	37
2.8	ETL	39
2.8.1	Data Warehouse	40
2.8.2	Data Lakes	41
2.8.3	Data Lakehouses	41
2.9	Molntjänster/on-premises	42
2.9.1	Molntjänster	42
2.9.2	On-premises	44
2.10	Big Data	44
2.11	Övningsuppgifter	46
3	Grundläggande SQL	49
3.1	Syntax	49
3.2	Konventioner	53
3.3	CRUD	54
3.3.1	Create	54
3.3.2	Read	58
3.3.3	Update	59
3.3.4	Delete	60
3.4	Introduktion av databasen Köksglädje	61
3.4.1	Beskrivning av databasen Köksglädje	61

3.4.2	Entity-Relationship-diagram	63
3.4.3	Inläsning av databasen Köksglädje	65
3.5	Queries	66
3.5.1	Joins	67
3.5.2	Select Top och Limit	75
3.5.3	Order By	77
3.5.4	Group By och Having	79
3.5.5	Case When	81
3.6	Funktioner	82
3.7	Vyer	83
3.8	Lagrade procedurer	85
3.9	Indexering	89
3.10	Exempel på några användbara queries	90
3.11	Övningsuppgifter	96
4	Mer om SQL	99
4.1	Mer avancerade queries	99
4.1.1	Subqueries	100
4.1.2	Common Table Expression (CTE)	103
4.2	Mer avancerade SQL-funktioner	106
4.2.1	Window Functions	106
4.2.2	User Defined Functions (UDF)	108
4.2.3	Textsök i SQL	111
4.3	Triggers	113
4.4	Övningsuppgifter	115
5	Databasadministration	117
5.1	Roller och säkerhet	117
5.1.1	Organisationsroller och behörigheter	118
5.1.2	Behörighetsstrategier	120
5.1.3	Skapa, tilldela och ändra roller	121
5.1.4	General Data Protection Regulation (GDPR)	127
5.2	Backupar och hantering av störningar	130
5.2.1	Backupar	130
5.2.2	Hantering av störningar	137
5.2.3	Retentionspolicy	140
5.3	Underhåll	144

5.3.1	Prestanda	145
5.3.2	Defragmentering	145
5.3.3	Regelbundna uppdateringar	146
5.3.4	Uppdatera programvara	146
5.3.5	Datavalidering/bearbetning	147
5.3.6	Diskutrymme/minne	148
5.4	Monitorering och loggfiler	148
5.4.1	Prestanda	149
5.4.2	Tillgänglighet	149
5.4.3	Händelser	150
5.4.4	Live-loggning och larm	151
5.5	Övningsuppgifter	153
6	NoSQL	155
6.1	NoSQL databastyper	156
6.1.1	Nyckel-värde-databaser	156
6.1.2	Dokumentdatabaser	159
6.1.3	Bredkolumndatabaser.	164
6.1.4	Grafdatabaser	167
6.2	Hantering av relationell data i NoSQL	172
6.3	Övningsuppgifter	173
7	Data Management	175
7.1	Datastyrning	176
7.2	Metadata	176
7.3	Data-livscykeln	177
7.3.1	Insamling	177
7.3.2	Bearbetning	178
7.3.3	Lagring	178
7.3.4	Användning	178
7.3.5	Arkivering	179
7.3.6	Destruktion	179
7.4	Datakvalitet	179
7.4.1	Dimensioner av datakvalitet	180
7.5	Roller och ansvarsområden	181
7.6	Datasäkerhet	182
7.7	Användbar data	182

7.8	Övningsuppgifter	184
8	Två avslutande exempel där SQL används	185
8.1	Utforskning av en databas	185
8.2	Full-stack applikation	196
8.3	Övningsuppgifter	205

Förord

I detta förord till boken kommer vi presentera bokens *GitHub*-sida, syfte och målgrupp, programvara som används, hur boken är utformad, några kommentarer kopplat till språket samt bokens disposition.

Bokens GitHub

Boken har en tillhörande *GitHub*-sida där material kopplat till boken finns uppladdat. Se följande länk:

https://github.com/AntonioPrgomet/laer_dig_databaser_och_sql_1uppl

Bokens syfte och målgrupp

Bokens syfte är att lära ut grunderna i databaser och “Structured Query Language” (SQL). Därför kan den användas för självstudier och i kurser av olika slag inom exempelvis yrkeshögskolan, företag eller andra organisationer. Boken lär ut databaser och SQL från grunden innebärande att inga förkunskaper inom dessa områden förutsätts. Även de som har vissa förkunskaper kan med fördel läsa boken för att få en gedigen grund.

Programvara som används i boken

Det finns flertalet olika databashanterare där SQL används. Två av de vanligast förekommande är “MySQL” och “Microsoft SQL Server” (MSSQL). I praktiken är skillnaderna små och den läsare som behärskar det ena kan med enkelhet lära sig det andra på kort tid. I boken visar vi kod för både MySQL och MSSQL direkt efter varandra i de fall det finns skillnader. I de fall koden är densamma i båda system så visas endast ett kodexempel som gäller för båda databashanterarna.

Det finns olika grafiska användargränssnitt för MySQL, vi har använt “MySQL Workbench” som är den mest populära. Det kan laddas ned gratis här: <https://dev.mysql.com/downloads/workbench/>

MSSQL är också gratis att ladda ned och det kan göras här: <https://www.microsoft.com/en-us/sql-server/sql-server-downloads>

I Kapitel 6 använder vi programmeringsspråket Python för att demonstrera bruket av NoSQL-databaser. I Avsnitt 8.2 använder vi Python för att utveckla en applikation som använder SQL för att lagra och redovisa data. För dessa delar av boken förutsätts det att läsaren behärskar grundläggande Python. Den läsare som inte har erfarenhet av Python kan lära sig det genom att se en video som finns på bokens *GitHub*-sida. Den läsare som vill fördjupa sig ytterligare kan till exempel läsa boken “Lär dig Python från grunden” som är författad av Linus Rundberg Streuli och Antonio Prgomet.

Bokens utformning

I boken använder vi två sorters informationsrutor som ser ut enligt följande.

i Såhär ser en allmän informationsruta ut. I de här rutorna skriver vi korta fördjupande förklaringar till vissa begrepp och koncept.

! Det här är en viktig-ruta. De här rutorna använder vi för att sätta fokus på viktiga distinktioner eller detaljer som kan vara svåra att upptäcka men som kan ha stor betydelse.

Ofta är bokens kodexempel annoterade. I högra kanten förekommer siffror som motsvarar en förklarande text under kodexemplet. Efter den förklarande texten följer oftast resultatet av själva koden. Nedan visas ett kodexempel där vi skriver ut två kolumner från en tabell som heter "Customers".

```
SELECT CustomerID,      ①  
       FirstName        ②  
FROM Customers;        ③
```

- ① Kolumnen "CustomerID" väljs.
- ② Kolumnen "FirstName" väljs.
- ③ Kolumnerna väljs från tabellen som heter "Customers".

CustomerID	FirstName
1	Greta
2	Zara
3	Dragan
4	Lars
5	Greta
6	Karin
7	Sven
8	Johan
9	Hussein
10	Ruth

Varje kapitel avslutas med övningsuppgifter vars syfte är att bidra till att läsaren befäster de kunskaper som gås igenom i kapitlet. Därför är uppgifterna av standard-karaktär snarare än komplex problemlösning.

I boken används tre typer av data när kod demonstreras.

- Enkel exempeldata. Denna typ av data är väldigt enkel eftersom fokus ska vara på att se vad koden gör och det underlättas när datan är enkel. Se exempelvis Avsnitt 3.5.1 där vi skapar tabellerna "Students" och "Classes".
- En pedagogiskt utvecklad databas som heter "Köksglädje". Det är en exempeldatabas för en fiktiv svensk butikskedja som säljer köksutrustning. Den kan nås på bokens *GitHub*-sida och introduceras i Avsnitt 3.4.
- En exempeldatabas för det fiktiva företaget "Wide World Importers" som heter "WideWorldImporters" och är utvecklad av Microsoft. Denna databas är mer komplex och syftet är att läsaren ska se hur det kan se ut i verkligheten. Denna databas används i Avsnitt 8.1. Även denna databas kan nås på bokens *GitHub*-sida.

Språk

Den här boken är skriven på svenska. Bokens kodexempel är däremot på engelska eftersom det är en starkt rådande praxis världen över att kod generellt sett skrivs på engelska. För vissa koncept använder vi de engelska begreppen i den löpande texten då de i praktiken används även när vi talar svenska. Ett sådant exempel är *query*. I de fall engelska begrepp används så är dessa kursiverade med undantag från bokens innehållsförteckning.

I Sverige används kommatecken som decimaltecken och punkt som tusentalavgränsare medan andra länder såsom USA gör tvärtom, det vill säga använder punkt som decimaltecken och komma som tusentalavgränsare. Ett-hundra-två-tusen-komma-ett hade i Sverige skrivits som 100.000,1 medan det i USA hade skrivits som 100,000.1. Vi har i denna bok av konsistensskäl valt att använda den amerikanska standarden eftersom punkt är vad som används som decimalavgränsare i databashanterare såsom MySQL och MSSQL.

Bokens disposition

Denna bok går igenom databaser och SQL från grunden. Det innebär att inga förkunskaper inom databaser eller SQL förutsätts. I Kapitel 1 ges en kontext och introduktion för databaser och SQL. I Kapitel 2 "Databasteori" går vi igenom grundläggande teori om databaser. I Kapitel 3 "Grundläggande SQL" går vi igenom grunderna för programmeringsspråket "Structured Query Language" (SQL) igenom, vilket används för att hantera data i relationella databaser. Kapitel 4 "Mer om SQL" är en direkt fortsättning på Kapitel 3 där vi fördjupar oss i mer avancerade tillvägagångssätt inom SQL. Oavsett om vi står inför skapandet av en databas eller om det redan finns en befintlig så behöver den administreras. Det är ämnet för Kapitel 5, "Databasadministration". Namnet NoSQL refererar till "non-SQL" eller "non-relational" och refererar till den typ av databaser som inte är relationella. Som exempel kan vi tänka på en databas som kan lagra diverse dokument. En överblick över denna typ av databaser ges i Kapitel 6 "NoSQL". För många organisationer är data en mycket viktig och värdefull resurs och behöver därför förvaltas. Detta är ämnet för Kapitel 7, "Data Management". Två större exempel där SQL används presenteras i Kapitel 8 som är bokens avslutande kapitel.

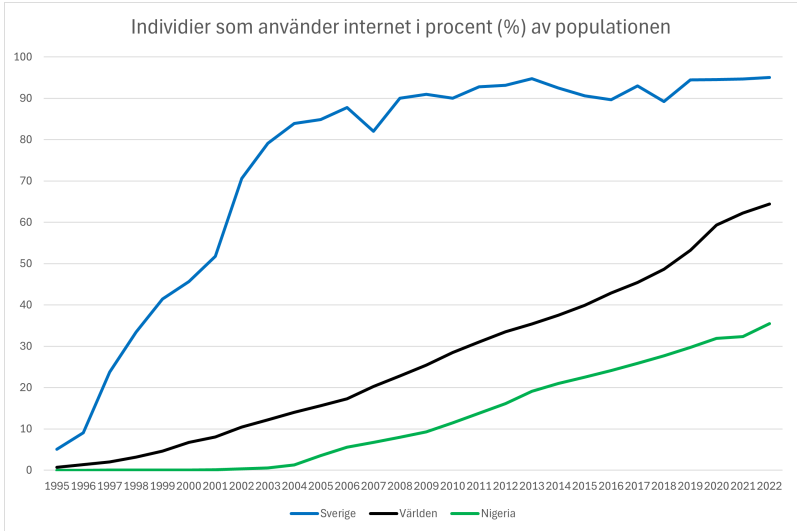
Kapitel 1

Introduktion till databaser och SQL

I detta, bokens första kapitel, kommer vi få en introduktion till databaser och SQL.

1.1 Informationsteknik (IT)

Informationsteknik (IT) har i modern tid genomgått en omfattande utveckling och blivit en grundpelare i dagens samhälle. Som exempel kan vi kolla på antalet individer som använder internet i procent (%) av populationen, se Figur 1.1. Exempelvis så använde cirka 0.7% av världsbefolkningen internet år 1995, en siffra som ökat till cirka 64% år 2022.



Figur 1.1: I Sverige har antalet individer som använder internet i procent (%) av populationen gått från cirka 5% år 1995 till cirka 95% år 2022. Motsvarande siffror för världsbefolkningen är 0.7% och 64%, för Nigeria gäller siffrorna 0% och 35%. Datan är hämtad från World Bank: <https://data.worldbank.org/indicator/IT.NET.USER.ZS>

i Samhället är idag i många sammanhang starkt beroende av IT system vilket kan medföra stora risker. Detta är något som kan utnyttjas av exempelvis kriminella hackergrupper i utpressningssyfte. Ett exempel är en händelse år 2021 där kassasystemet för matkedjan Coop blev utsatt för en IT attack med konsekvensen att de inte kunde ta betalt av sina kunder. Detta ledde till att Coop temporärt fick stänga ner cirka 800 butiker.

Precis som namnet antyder så kan vi säga att IT handlar om teknik för att hantera information. Exempel på sådan teknik är databaser som kan definieras som "en organiserad samling av information" där vi i modern tid ofta använder datorer för att lagra information i databaser. Innan datorns intåg kunde en organiserad samling information exempelvis vara ett arkiv med fysiska dokument i pappersform. I avsnittet som följer går vi djupare in på databaser.

1.2 Databaser, databashanterare och SQL

Vi kan kategorisera databaser som relationella och icke-relationella.

I relationella databaser är data lagrad i tabeller som består av rader och kolumner. Mellan tabellerna finns det relationer som gör att dessa kan sammankopplas, exempelvis för att skapa en ny tabell bestående av utvalda kolumner från andra tabeller. För att interagera med en databas så används det som kallas för en databashanterare. För relationella databaser är två vanligt förekommande databashanterare MySQL och Microsoft SQL Server (MSSQL). I dessa databashanterare så används programmeringsspråket "Structured Query Language" (SQL) med vars hjälp vi bland annat kan hämta och modifiera data. SQL uttalas bokstav för bokstav ("S-Q-L") eller ibland som i engelskans "sequel". Beroende på vilken databashanterare som används så kan SQL-kod se annorlunda ut, men som läsaren kommer se i Kapitel 3 skrivs i många fall identisk eller liknande kod i MySQL och MSSQL.

Icke-relationella databaser kallas också för NoSQL-databaser. Namnet NoSQL refererar till "non-SQL" eller "non-relational" och refererar till den typ av databaser som inte är relationella. Som exempel kan vi tänka en databas som kan lagra diverse dokument eller bilder. NoSQL-databaser kommer vi lära oss om i Kapitel 6.

Databaser används för att samla och lagra data och information. Just data och information är ämnet för nästa avsnitt.

1.3 Data och information

Två begrepp som har en nära koppling är "data" och "information". Vi kan säga att information är data som blivit bearbetad och/eller tolkad för att vara meningsfull för användaren. Data är alltså råmateriet som bearbetas och/eller tolkas för att skapa information. Här är ett exempel på vad en person hade kunnat säga till en kollega på ett företag och som visar på distinktionen mellan data och information. "Här är en försäljningsrapport för föregående år [Information]. Försäljningsrapporten är baserad på datan i vår databas [data]." I många sammanhang används de två begreppen synonymt och beroende på kontext och/eller behov så kan det vara mer eller mindre användbart att faktiskt särskilja de två begreppen.

Enligt Statista (<https://www.statista.com/statistics/871513/worldwide-data-created/>) så har mängden data/information som skapats, lagrats, kopierats eller konsumerats per år i världen ökat exponentiellt. År 2010 rörde det sig om cirka 2 zettabytes data. Tio år senare, år 2020, rörde det sig om cirka 64 zettabytes data. År 2025 förväntas den siffran vara cirka 181 zettabytes.

Data kan kategoriseras som strukturerad, ostrukturerad eller semi-strukturerad. Strukturerad data är sådan data som blivit organiserad och standardiserad utifrån en definierad struktur. Exempelvis kan ett företag ha en kundtabell där varje kund har ett förnamn, efternamn och ålder där det blivit standardiserat att förnamn och efternamn är en textsträng medan ålder ska vara ett heltal som är större än eller lika med noll. Ostrukturerad data är sådan data som inte har en inneboende organiserad struktur som exempelvis bilder, mejl eller

textdokument. För att kunna identifiera och organisera ostrukturerad data kan den märkas med en etikett och får då en viss struktur, då kan den sägas vara semi-strukturerad. Om vi tänker oss att vi har två dokument som innehåller text och vet att Johanna skrivit det ena och Björn det andra. Då hade vi kunnat se de två dokumenten som en samling av ostrukturerad data. Om vi på det dokumentet som Johanna skrivit sätter en post-it lapp och skriver "Johanna" och på det dokumentet som Björn skrivit sätter en post-it lapp och skriver "Björn" så har vi infört en viss struktur och datan kan då betraktas som semi-strukturerad. För strukturerad data används primärt relationella databaser och för ostrukturerad och semi-strukturerad data används primärt NoSQL-databaser. Generellt sett används strukturerad data i mycket större utsträckning än ostrukturerad data eftersom den är lättare att använda. Det finns dock uppskattningar på att cirka 80% av all data som existerar är ostrukturerad.

För de flesta organisationer är data ofta en mycket viktig resurs som bland annat används för rapportering, analys och diverse beslutsprocesser. Som exempel kan nämnas att företag kan skapa och använda rapporter för att se hur dess försäljning utvecklas. Dessa rapporter kan därefter analyseras och beroende på analysens utfall kan olika åtgärder såsom till exempel prisjustering eller produktutveckling påbörjas för att företaget ska vara konkurrenskraftigt på sin marknad. Data behövs även för att uppfylla juridiska krav. Exempelvis behöver en bank ha kunskap och kännedom om dess kunder och vad de använder lånade pengar till. Detta för att försvåra och förhindra att banker och dess verksamhet utnyttjas för penningtvätt eller finansiering av terrorism, men även för att veta vilka risker de tar. Precis som ekonomiska, mänskliga eller naturliga resurser såsom skog och olja har ett värde och behöver förvaltas så är data också en resurs som har ett värde och behöver förvaltas. Fraserna "Data is the new gold" och "Data is the new oil" konkretiserar det faktum att data är en värdefull resurs.

Det är ingen överdrift att säga att stater, företag, organisationer och individer idag i hög utsträckning är väldigt beroende av data för att kunna verka effektivt eller ens verka alls. Vi ger några exempel nedan.

- Stater lagrar och använder data om medborgare såsom ålder,

ägarbesiddelse och skatt. Föreställ dig vad som hade hänt ifall information om vem som till exempel äger vilka fastigheter hade gått förlorad eller om de system som hanterar data kopplat till hur mycket skatt olika individer och organisationer ska betala kollapsar.

- Företag lagrar och använder data/information om exempelvis kunder, försäljning och anställda. Denna data kan användas för att skapa rapporter och analyser som i sin tur kan ligga till grund för hur företaget ska agera.
- Individer använder dagligen pengar för att genomföra diverse transaktioner såsom att köpa mjölk eller boka en tågbiljett via internet. Många betalningar görs via t.ex. bankkort vilket innebär att pengar flyttas mellan konton som helt enkelt är lagrad information. När en tågbiljett bokas så sparas exempelvis information om biljettköpet och kunden som köpt biljetten i tågbolagets databaser.

1.4 Övningsuppgifter

1. Vad står förkortningen “IT” för?
2. Förklara begreppen databas, databashanterare och SQL.
3. Förklara begreppen “data” och “information”.

Kapitel 2

Databasteori

I detta kapitel presenteras grundläggande teori för databaser.

Vi börjar med att i Avsnitt 2.1 presentera tre abstraktionsnivåer för databaser.

I Avsnitt 2.2 till och med Avsnitt 2.7 presenteras databasteori där det är underförstått att vi syftar på relationella databaser. Icke-relationella databaser kommer behandlas i Kapitel 6.

Avsnitt 2.8 till och med Avsnitt 2.10 är allmän teori relevant för alla typer av databaser, såväl relationella som icke-relationella.

I nästa kapitel kommer vi använda SQL-programmering för att praktiskt arbeta med relationella databaser. Innehållet i detta kapitel tenderar att bli mer konkret efter det. Därför är det en god idé att kolla igenom detta kapitel igen efter att Kapitel 3 har arbetats igenom.

2.1 Tre abstraktionsnivåer för databaser

I detta avsnitt går vi igenom de tre abstraktionsnivåerna som kan ge oss en förståelse för hur databassystem är uppbyggda och fungerar.

Vi kan betrakta en databas ur tre olika abstraktionsnivåer. Dessa är den “fysiska nivån”, den “logiska nivån” och den “externa nivån”.

Den fysiska nivån – refererar till hur data fysiskt lagras. Exempelvis på diskar eller i filer på en server. När vi senare diskuterar huruvida data ska lagras via molnet eller *on-prem* (Avsnitt 2.9), indexering (Avsnitt 2.6.1) och lagring (Avsnitt 5.3.6) så befinner vi oss på den fysiska nivån.

Den logiska nivån – refererar till de datatyper och relationer som finns i databasen. När vi senare diskuterar exempelvis nycklar (Avsnitt 2.3.5), relationer mellan tabeller (Avsnitt 2.3.6) och hur data är organiserad (Avsnitt 2.5) så befinner vi oss på den logiska nivån.

Den externa nivån – refererar till den data som en användare ser eller kan använda. När vi senare diskuterar tabeller (Avsnitt 2.3.1), vyer (Avsnitt 3.7) eller vilka delar av en databas en användare har behörighet till (Avsnitt 5.1.1) så befinner vi oss på den externa nivån.

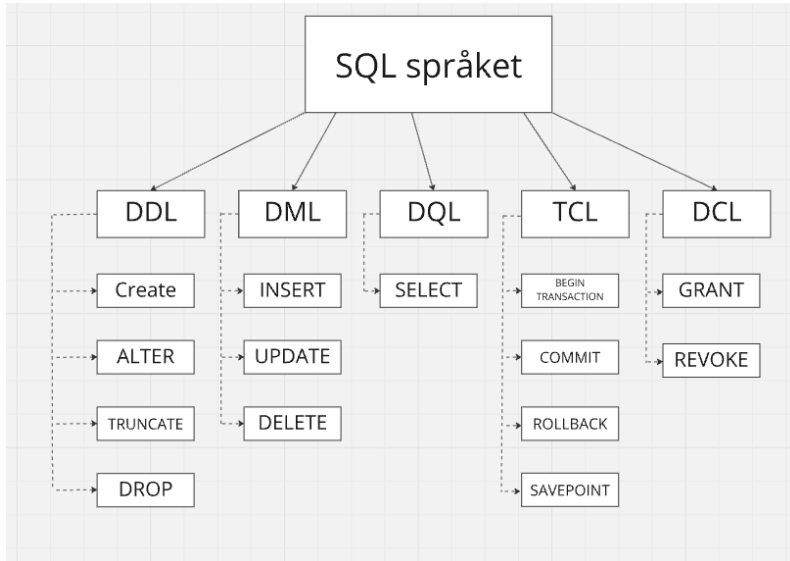
2.2 SQL-språkets kategorisering

SQL-kommandon kan delas in i kategorier. Varje kategori har ett specifikt syfte i SQL. Genom att känna till denna kategorisering får vi en mer explicit förståelse hur SQL kan användas.

- **Data definition language (DDL)** - DDL består av kommandon som används för att definiera och hantera en databas. Exempelvis `CREATE` och `DROP` som skapar, ändrar och raderar databasens struktur.
- **Data manipulation language (DML)** - DML består av kommandon som manipulerar data. Exempelvis `INSERT` och `DELETE` som förändrar innehållet i tabeller.
- **Data query language (DQL)** - DQL består endast av ett kommando: `SELECT`. Det används för att läsa ut data som operationer därefter kan utföras på.
- **Transaction control language (TCL)** - TCL består av kommandon som utför transaktioner, exempelvis `COMMIT` och

ROLLBACK. Vad transaktioner är går vi igenom i Avsnitt 2.3.2.

- **Data control language (DCL)** - DCL består av kommandon som hanterar rättigheter i en databas. Exempelvis `GRANT` som ger användare valda rättigheter i en databas.



Figur 2.1: Kategorisering av SQL-språket.

2.3 Grundläggande teori för relationsdatabaser

I det här avsnittet börjar vi med att gå igenom tabeller som är en central del av relationsdatabaser. Vidare går vi igenom vad en transaktion är och hur de utförs med hjälp av CRUD-flödet, hur begränsningar sätts och hur tabellrelationer etableras genom nycklar.

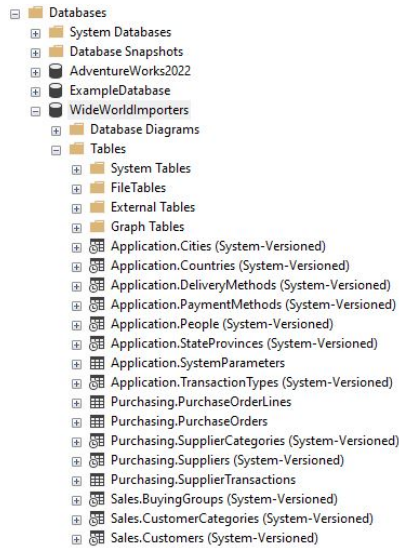
2.3.1 Tabeller

Varje tabell i en relationell databas ska ha ett unikt namn och vid namngivning av tabeller eftersträvas deskriptiva och lättförståeliga namn. Tabeller är sällan ensamstående, utan är ofta en del av en större samling av tabeller. I MSSQL kallas en sådan samling för databas-schema. Ett schema hade kunnat heta "Customers" som innehåller tabeller kopplat till ett företags kunder. En tabell som heter "CustomerInfo" hade kunnat ha kolumner såsom förnamn, efternamn och ålder. En annan tabell som heter "CustomerAddress" hade kunnat ha kolumner såsom address, postnummer, stad och land. En tredje tabell "CustomerContact" hade kunnat ha kolumner såsom mobilnummer och mejladress. I MSSQL är alltså strukturen Databas → Schema → Tabell. Notera att MySQL inte har samma struktur, där benämns hela databasen som "schemat", strukturen i MySQL är alltså Databas → Tabell.

I Figur 2.2 ser vi att databasen "WideWorldImporters" används. I databasen finns det olika scheman såsom "Application", "Purchasing" och "Sales" som innehåller tabeller. Exempelvis har "Application"-schemat en tabell som heter "Cities" och namnges då på formen "Application.Cities".

Varje rad i en tabell representerar en "post" medan kolumner innehåller de attribut som kännetecknar posten. Varje kolumn har en specifik datatyp, vilket säkerställer att värden i kolumnen konsekvent är av samma datatyp. Se Figur 2.3 för ett praktiskt exempel på hur en tabell innehållande data om bilar kan se ut.

Några vanligt förekommande datatyper är heltal, decimaltal, textsträngar av obestämd längd, textsträngar av bestämd längd och datum. Se Tabell 2.1 för hur ett antal vanliga datatyper benämns i olika databashanterare. De två databashanterarna PostgreSQL och SQLite kommer vi inte gå in på i denna bok. I tabellens andra kolumn har vi skrivit MSSQL/MySQL eftersom de datatyperna vi presenterar benämns på samma sätt i MSSQL och MySQL.



Figur 2.2: I MSSQL kan det finnas olika databaser. En relationell databas kan ha olika scheman där relaterade tabeller är samlade. Hierarkin är alltså Databaser → Scheman → Tabeller. I MySQL är hierarkin Databaser → Tabeller där databaser benämns “Schema”.

	A	B	C	D	E	F	G	H
1	pris	bränsle	växellåda	milttal	modellår	bilty	drivning	hästkrafter
2	229900	miljöbränsle/hybrid	automat	1994	2022	halvkombi	tvåhjuldriven	116
3	269900	miljöbränsle/hybrid	automat	0	2023	halvkombi	tvåhjuldriven	117
4	189900	bensin	automat	7609	2017	suv	fyrhjuldriven	116
5	419900	miljöbränsle/hybrid	automat	6500	2021	suv	fyrhjuldriven	306
6	254000	miljöbränsle/hybrid	automat	689	2021	halvkombi	tvåhjuldriven	123
7	304900	miljöbränsle/hybrid	automat	1651	2022	suv	fyrhjuldriven	117
8	190000	miljöbränsle/hybrid	automat	4244	2021	kombi	tvåhjuldriven	124
9	229900	miljöbränsle/hybrid	automat	3692	2022	suv	fyrhjuldriven	306
10	419900	bensin	manuell	1415	2020	halvkombi	fyrhjuldriven	262

Figur 2.3: En tabell innehållande data om bilar.

Tabell 2.1: Några vanligt förekommande datatyper.

Beskrivning	MySQL/MSSQL	PostgreSQL	SQLite
Textsträng, bestämd längd	CHAR(____)	CHAR	CHAR
Textsträng, obestämd längd	VARCHAR(____)	VARCHAR	TEXT
Heltal	INT	INTEGER	INTEGER
Stora heltal	BIGINT	BIGINT	BIGINT
Små heltal	SMALLINT	SMALLINT	SMALLINT
Decimaltal	FLOAT	REAL	REAL
Datum	DATE	DATE	DATE
Datum och tid	DATETIME	TIMESTAMP	TEXT

2.3.2 Transaktioner

En transaktion är en enhetlig operation bestående av ett eller flera SQL-kommandon. Syftet är att dessa operationer ska exekveras tillsammans, som en enhet. Detta för att alla operationer som utförs i databasen ska vara konsekventa och pålitliga.

I Kapitel 3 kommer vi lära oss att skriva och förstå SQL-kod, men vi kan redan nu se ett exempel på hur vi skulle kunna rätta till ett stavfel i en av våra tabeller. Koden nedan uppdaterar tabellen “Customers” och ersätter värdet “Stokholm” med “Stockholm” i kolumnen “City” eftersom det är felstavat.

```
UPDATE Customers
SET City = 'Stockholm'
WHERE City = 'Stokholm';
```

Denna SQL-*query* bestod av tre delar; UPDATE, SET och WHERE. Tillsammans uppdaterar de informationen i tabellen “Customers”. Utförs detta inom ramen för en transaktion säkerställs att hela operationen utförs som en helhet. Detta innebär att om ett fel uppstår under uppdateringen kommer inga ändringar utföras i databasen.

Query är ett återkommande begrepp genom hela boken. En *query* är en förfrågan som ställs till databasen att hämta, uppdatera, info-

ga eller ta bort data. Många interaktioner med databasen sker via *queries*.

Transaktionen säkerställer också att uppdateringen inte krockar med andra eventuella transaktioner som arbetar med samma data. Detta kan ske när flera personer arbetar samtidigt vilket ofta är fallet i företag och organisationer. Den ser till att uppdaterad information sparas i databasen innan nästa transaktion försöker hantera samma data.

2.3.3 CRUD-flödet

CRUD är en akronym för *Create*, *Read*, *Update* och *Delete*. Begreppet innefattar de grundläggande operationer som kan utföras på data i en relationell databas.

- *Create*-momentets syfte är att lägga till rader i en tabell.
- *Read*-momentets syfte är att läsa ut data från en tabell.
- *Update*-momentets syfte är att uppdatera existerande data i en tabell.
- *Delete*-momentets syfte är att ta bort rader från en tabell.

Operationerna kan påverka en eller flera rader åt gången, beroende på hur vi utformar vår *query* till databasen. Praktisk tillämpning av CRUD går igenom i Avsnitt 3.3.

2.3.4 Begränsningar

Begränsningar, eller *constraints*, dikterar de regler en tabell eller en kolumn följer. Krockar en operation med en begränsning avslutas transaktionen och ett felmeddelande genereras. Begränsningar på tabell-nivå inkluderar hela tabellen medan begränsningar på kolumn-nivå endast inkluderar angiven kolumn.

Exempel på vanligt förekommande begränsningar:

- **NOT NULL** – dikterar att data i kolumnen måste ha ett giltigt värde som inte är NULL.

- **UNIQUE** – data i kolumnen måste vara unik, inga duplikat får förekomma.
- **PRIMARY KEY (PK)** – inkluderar både **NOT NULL** och **UNIQUE** begränsningar. En PK ska unikt kunna identifiera varje rad i en tabell.
- **FOREIGN KEY (FK)** – kolumner med FK kan identifiera data i andra tabeller genom sin koppling till den andra tabellens PK eller kolumner som har en **UNIQUE** begränsning.
- **CHECK** – värdet i en vald kolumn måste stämma överens med de tillstånd eller villkor kolumnen tilldelats via **CHECK**-begränsningen. Om exempelvis en tabell innehåller fotbollsspelare hade en kolumn som heter “Position” där de tillåtna värdena är “Målvakt”, “Försvarare”, “Mittfältare” och “Anfallare” varit lämpligt.
- **DEFAULT** – anger ett standardvärde för kolumnen. Kolumnen använder sig av det angivna standardvärdet om inget annat anges.

2.3.5 Nycklar

Nycklar är ett attribut som bör finnas i varje tabell, främst för att skapa relationer mellan tabeller men även för att främja dataintegritet. Det finns totalt sex olika nyckeltyper, de vanligast förekommande är **PRIMARY KEY** och **FOREIGN KEY**.

De sex nycklarna är:

- **Primärnycklar (PRIMARY KEY)** – den vanligaste typen av nyckel, agerar unik identifierare åt tabellen. Den kolumn eller de kolumner som väljs till PK får enbart innehålla unika värden som är skilda från **NULL**. Varje tabell kan endast ha en primärnyckel. Primärnyckeln kan dock bestå av fler än en kolumn. Nedan följer två kodexempel för att förtydliga. I det första kodexemplet väljer vi en kolumn som primärnyckel. I det andra kodexemplet väljer vi två kolumner som primärnyckel.

```
CREATE TABLE Customers (                                ①
    CustomerID INT PRIMARY KEY,                          ②
    FirstName VARCHAR(50),                                ③
    LastName VARCHAR(50)
);
```

- ① Tabellen “Customers” skapas.
- ② Kolumnen “CustomerID” skapas och tilldelas datatypen INTEGER. Den utses även till primärnyckel.
- ③ Kolumnen “FirstName” skapas och tilldelas datatypen VARCHAR med maxlängd 50.

```
CREATE TABLE OrderDetails (                             ①
    OrderID INT,
    ProductID INT,
    Quantity INT,                                         ②
    PRIMARY KEY (OrderID, ProductID)                     ③
);
```

- ① Tabellen “OrderDetails” skapas.
 - ② Kolumnerna “OrderID”, “ProductID” och “Quantity” skapas och tilldelas datatypen INTEGER.
 - ③ Tillsammans skapar kombinationen “OrderID” och “ProductID” en sammansatt primärnyckel. Detta för att en produkt eller en order kan återkomma, men kombinationen av dessa kommer alltid vara unik.
- Främmande nycklar (FOREIGN KEY) – vanligtvis är det en främmande nyckel som skapar relationen till en annan tabell genom att referera till den primärnyckel som existerar i den första tabellen.
 - Sammansatt nyckel (*Composite key*) – när primärnyckeln innehåller fler än en kolumn kallas det för en sammansatt nyckel. Detta är användbart när ingen enskild kolumn har unika attribut, men när en kombination av kolumner har det.

- Kandidatnyckel (*Candidate key*) – alla kolumner, eller kombinationer av kolumner, som unikt kan identifiera rader i en tabell är en kandidatnyckel. När en primärnyckel valts kvarstår övriga kandidatnycklar som alternativa unika identifierare. För att en kolumn ska kunna vara en kandidatnyckel måste den innehålla unika värden.
- Alternativnyckel (*Alternate key*) – en kandidatnyckel som för tillfället inte är primärnyckel.
- Supernyckel (*Super key*) – en uppsättning av en eller flera kolumner som unikt identifierar en rad. Alla primärnycklar är även supernycklar, eftersom en primärnyckel alltid unikt identifierar en rad. Alla supernycklar är dock inte primärnycklar, då en supernyckel inte behöver vara minimalistisk, vilket är ett krav för en primärnyckel. Se nedanstående exempel.

I tidigare exempel skapade vi en tabell där två kolumner bildade en sammansatt primärnyckel, dessa två kolumner är också en supernyckel. Vi skulle teoretiskt sett kunna lägga till ännu en kolumn till den sammansatta primärnyckeln på följande sätt.

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID, Quantity) ①
);
```

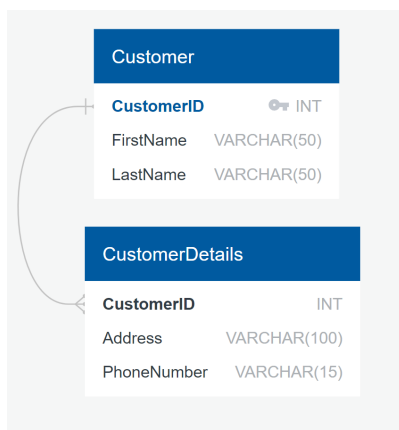
- ① “OrderID”, “ProductID” och “Quantity”-kolumnerna tilldelas primärnyckel-status och bildar en supernyckel.

Den tredje kolumnen “Quantity” tillför ingen unikhet i primärnyckeln då de första två kolumnerna redan unikt identifierar varje rad i tabellen. Detta bryter mot kravet att en primärnyckel ska hållas minimalistisk.

2.3.6 Tabellrelationer

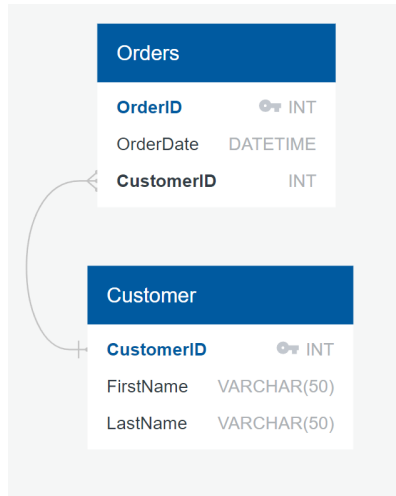
Nycklar används för att skapa relationer mellan tabeller. Nedan listas de relationer som finns.

One-to-one – när den främmande nyckeln är unik i sin tabell utgör den en “one-to-one”-relation med sin kopplade primärnyckel. Exempelvis har varje kund endast ett “CustomerID” i både “Customer” och “CustomerDetail”-tabellerna, se Figur 2.4.



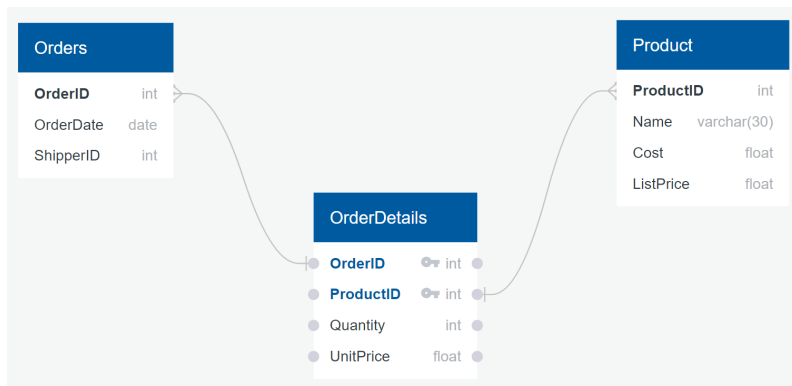
Figur 2.4: Varje enskilt “CustomerID” är unikt i båda tabellerna.

One-to-many – när den främmande nyckeln inte är unik i sin tabell handlar det om en “one-to-many”-relation. Exempelvis kan varje unikt “CustomerID” kopplas till flera “OrderID”, se Figur 2.5 för ett exempel.



Figur 2.5: Varje "CustomerID" kan förekomma upprepade gånger i "Orders"-tabellen, eftersom en kund ofta kan lägga flera ordrar över tid.

Many-to-many – när flera rader i en tabell har en relation med flera rader i en annan tabell. För att skapa den här typen av relation måste en ny tabell skapas mellan de två tabeller som ska kopplas samman. Exempelvis kan samma person ha många ordrar i tabellen "Orders" där den handlat samma föremål från tabellen "Products" flera gånger. Då behöver en ny tabell skapas mellan de två, "OrderDetails" där vi anger "OrderID" och "ProductID" som primärnycklar till de första två tabellerna. Nu har vi indirekt skapat en "many-to-many"-relation. Se Figur 2.6.



Figur 2.6: Exempel på en “many-to-many”-relation.

2.4 ACID

ACID är en akronym för *Atomicity*, *Consistency*, *Isolation* och *Durability*. Det här är de grundpelare som en databastransaktion vilar på. Om ACID följs säkerställs det att databasen fortsatt är i korrekt skick även efter att data uppdaterats, hämtats eller förändrats.

2.4.1 Atomicity

Atomicity innebär att varje transaktion som förändrar data ska utföras i sin helhet, annars exekveras inte transaktionen. En operation ska alltså inte kunna delvis genomföras, då det kan medföra dataförlust eller datakorruption. Datakorruption innebär att data ändras på ett felaktigt sätt. Om en händelse inte kan genomföras i sin helhet ska inga förändringar ske i databasen. Detta kallas vanligtvis för *all-or-nothing*-regeln. Moderna databashanterare som MSSQL och MySQL sköter *atomicity* automatiskt.

För mer explicit kontroll kan *atomicity* upprätthålls genom exempelvis loggfiler. När en transaktion utförs loggas alla operationer och misslyckas den ska databasen återställas till det ursprung den befann sig i innan transaktionen utfördes. `BEGIN TRANSACTION` (I MySQL an-

vänds enbart **BEGIN** eller **START TRANSACTION**) och **ROLLBACK/COMMIT TRANSACTION** kan användas för att specificera var en transaktion börjar och slutar. **SAVEPOINT** kan användas för att skapa en punkt att återgå till, snarare än att återställa hela transaktionen om något skulle gå fel. Nedan följer ett exempel på hur det kan se ut.

```
BEGIN TRANSACTION;                                ①
INSERT INTO Orders (OrderID, CustomerID) VALUES (5,
    ↪ 10);                                           ②
SAVEPOINT SavePoint1;                              ③
INSERT INTO OrderDetails (OrderID, ProductID,
    ↪ Quantity) VALUES (1, 101, 10p);              ④
ROLLBACK TO SavePoint1;                            ⑤
COMMIT;                                             ⑥
```

- ① Transaktionen startar.
- ② Värdena 5 och 10 läggs till i kolumnerna “OrderID” och “CustomerID”.
- ③ “Savepoint1” skapas, en punkt dit transaktionen kan rullas tillbaka vid eventuella fel.
- ④ Värdena 1, 101 och felskrivningen “10p” försöker läggas till i kolumnerna “OrderID”, “ProductID” och “Quantity”.
- ⑤ “Savepoint1” specificeras som den punkt transaktionen ska rullas tillbaka till om ett fel uppstår. Eftersom “Quantity”-kolumnen inte accepterar värdet “10p” kommer den här transaktionen rullas tillbaka.
- ⑥ Eftersom transaktionen rullades tillbaka till “Savepoint1” utförs och sparas enbart de ändringar som gjordes innan den punkten.

2.4.2 Consistency

Consistency innebär att tabeller alltid ska befinna sig i ett giltigt tillstånd efter varje transaktion.

Consistency upprätthålls med bland annat begränsningar, *triggers* och lagrade procedurer som går igenom i Avsnitt 3.8.

2.4.3 Isolation

Isolation innebär att flera händelser ska kunna ske samtidigt utan att de olika operationerna påverkar varandra. Även om flera transaktioner sker parallellt ska det verka som att den enskilda operationen är den enda som pågår. Detta motverkar dataförlust och datakorruption.

Det här uppnås genom låsning. Databasen låser rader, tabeller eller andra resurser för att förhindra påverkan från flera håll samtidigt. SQL gör detta automatiskt åt oss. I Tabell 2.2 listar vi de isoleringsnivåer som finns samt hur de transaktionsproblem som kan uppstå är möjliga eller ej på respektive nivå.

Tabell 2.2: Olika isoleringsnivåer. Högre isoleringsnivåer möjliggör färre problem.

Isolerings-nivå	<i>Dirty read</i>	<i>Non-repeatable read</i>	<i>Phantoms</i>
Read uncommitted	Möjlig	Möjlig	Möjlig
Read committed	Ej möjlig	Möjlig	Möjlig
Repeatable read	Ej möjlig	Ej möjlig	Möjlig
Serializable	Ej möjlig	Ej möjlig	Ej möjlig

Dirty read - uppstår när en transaktion läser data som inte blivit sparad. Exempelvis kan Transaktion 1 ändra data, men innan ändringarna sparats läser Transaktion 2 samma data. Transaktion 1 avbryts och Transaktion 2 har nu felaktig data eller data som inte borde existera.

Non-repeatable read - uppstår när en transaktion läser av samma rad flera gånger och får olika värden. Skiljer sig från *dirty reads* då Transaktion 1 måste ha sparat sina ändringar för att Transaktion 2 ska få ett annat värde mellan läsningar.

Phantoms - uppstår när två identiska transaktioner returnerar olika resultat. Detta beror på att den underliggande datan ändrats.

- **Read Uncommitted** tillåter *dirty reads*, *non-repeatable reads*, och *phantoms*. Detta är den lägsta isoleringsnivån där transak-

tioner kan läsa osparade ändringar gjorda av andra transaktioner.

- ***Read Committed*** förhindrar *dirty reads*. Det här är den vanligaste standardinställningen. Transaktioner läser endast data som blivit sparad, men data kan fortfarande ändras mellan läsningar, vilket leder till *non-repeatable reads*.
- ***Repeatable read*** förhindrar både *dirty reads* och *non-repeatable reads*. Denna nivå säkerställer att om en transaktion läser en rad, kommer ingen annan transaktion att kunna ändra den raden innan den första transaktionen är avslutad. Nya rader kan dock fortfarande tillkomma, vilket orsakar *phantoms*.
- ***Serializable*** är den högsta isoleringsnivån där *dirty reads*, *non-repeatable reads*, och *phantoms* är förhindrade. Denna nivå ser till att transaktioner är helt isolerade, vilket simulerar exekvering av transaktionerna sekventiellt.

Notera att högre nivåer är mer resurskrävande då de utför många låsningar och blockeringar samtidigt. Den höga låsningsgraden kan även skapa väntetid vid dataförfrågningar eller *deadlocks* där två transaktioner väntar på att den andres låsning ska släppa. I MySQL är "*repeatable read*" standardinställningen. I MSSQL är "*read committed*" standardinställningen.

2.4.4 Durability

Durability innebär att en transaktion som ägt rum ska sparas permanent. Även om exempelvis strömmen bryts eller om servern kraschar. Data ska lagras på säkra fysiska platser och *backuper* ska finnas redo att återställas ifall något sker.

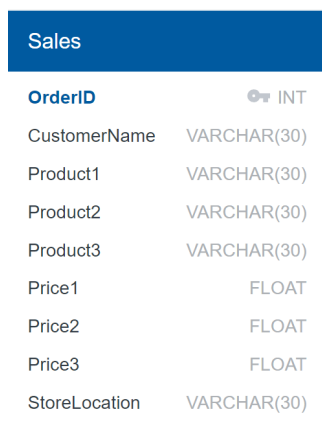
Innan en ändring utförs i databasen skrivs en loggfil (*Write-Ahead Logging*, WAL) som beskriver den ändring som är på väg att utföras. Detta sker automatiskt i MSSQL och MySQL. Loggfilen kan användas för att återskapa databasen i rätt tillstånd, förutsatt att loggfilen lagrats på en säker plats.

2.5 Normalisering

För att en relationell databas ska fungera effektivt och minimera risken för oväntade fel tillämpas normalisering. De vanligast förekommande normaliseringsformerna kallas onormaliserad form, eller på engelska *unnormalized form*, (UNF), första normalformen (1NF), andra normalformen (2NF) och tredje normalformen (3NF). Vi kommer beskriva dem i kommande underavsnitt. Det är utanför denna bokens omfattning men för fullständighetens skull nämner vi att det finns fler normalformer; EKNF, BCNF, 4NF, ETNF, 5NF, DKNF, 6NF.

Att normalisera en databas innebär att tabeller bryts ner till sina minsta beståndsdelar. Detta skapar därmed fler tabeller där fler relationer måste skapas, men säkerställer datans pålitlighet. Notera att fler tabeller också innebär att data måste hämtas från fler källor i databasen. Detta kan vara resurskrävande för systemet och resultera i längre svarstider.

I Figur 2.7 ser vi ett exempel på en tabell. Denna tabell kommer vi använda i följande fyra underavsnitt för att förklara UNF, 1NF, 2NF och 3NF.



Sales	
OrderID	INT
CustomerName	VARCHAR(30)
Product1	VARCHAR(30)
Product2	VARCHAR(30)
Product3	VARCHAR(30)
Price1	FLOAT
Price2	FLOAT
Price3	FLOAT
StoreLocation	VARCHAR(30)

Figur 2.7: Tabellens utgångspunkt, UNF.

2.5.1 Onormaliserad form (UNF)

Den onormaliserade formen är ofta startpunkten för ett databasschema. I den här formen lagras data utan riktlinjer för struktur, vilket kan leda till redundans och ineffektiv tabell-design. Som vi ser i Figur 2.8 kan en onormaliserad tabell innehålla upprepande kolumner för varje post.

I Figur 2.8 lämnas celler tomma om en kund handlar färre än tre varor, eftersom det finns tre “Product”-kolumner. Den här datan kommer vara svår att uppdatera, då samma produkt kan hamna i olika kolumner.

OrderID	CustomerName	Product1	Product2	Product3	Price1	Price2	Price3	StoreLocation
1	Kjell	TV	Soundbar	HDMI-kabel	12999	2499	249	Stockholm
2	Lena	Laptop	Musmatta	NONE	13799	159	NONE	Malmö
3	Hans	AC	NONE	NONE	4359	NONE	NONE	Göteborg

Figur 2.8: Hur tabellen ser ut i praktiken.

2.5.2 Första normalformen (1NF)

Första normalformen säger att:

- Varje post innehåller atomiska värden som inte kan delas upp ytterligare. I exemplet innebär detta att produkt och pris enbart ska representeras i en kolumn var.

För att transformera tabellen i enlighet med 1NF tar vi bort de överflödiga produkt och priskolumnerna. Se Figur 2.9.

Eftersom varje order kan innehålla flera produkter har vi använt “OrderID” och “Product”-kolumnerna för att skapa en sammansatt nyckel.

2.5.3 Andra normalformen (2NF)

Andra normalformen säger att:

- 1NF följs.

Sales1NF	
OrderID	int
CustomerName	varchar(30)
Product	varchar(30)
Price	float
StoreLocation	varchar(30)

Figur 2.9: Tabellen är nu atomisk i enlighet med 1NF.

- Alla kolumner som inte är nyckel-kolumner ska vara beroende av den primärnyckel som finns i tabellen.

Det här följs inte i Figur 2.9, eftersom "StoreLocation"-kolumnen är beroende av kolumnen "OrderID" men inte av kolumnen "Product". Detta eftersom en order är kopplad till en specifik butik, men en specifik produkt är inte det.

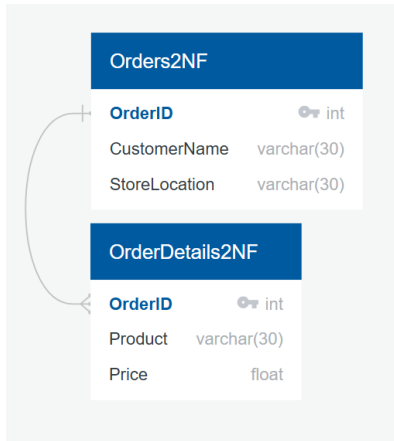
För att transformera tabellen enligt 2NF bryter vi ner tabellen till två tabeller, "Orders" och "OrderDetails". I Figur 2.10 ser vi att alla attribut nu är beroende av respektive nyckel.

2.5.4 Tredje normalformen (3NF)

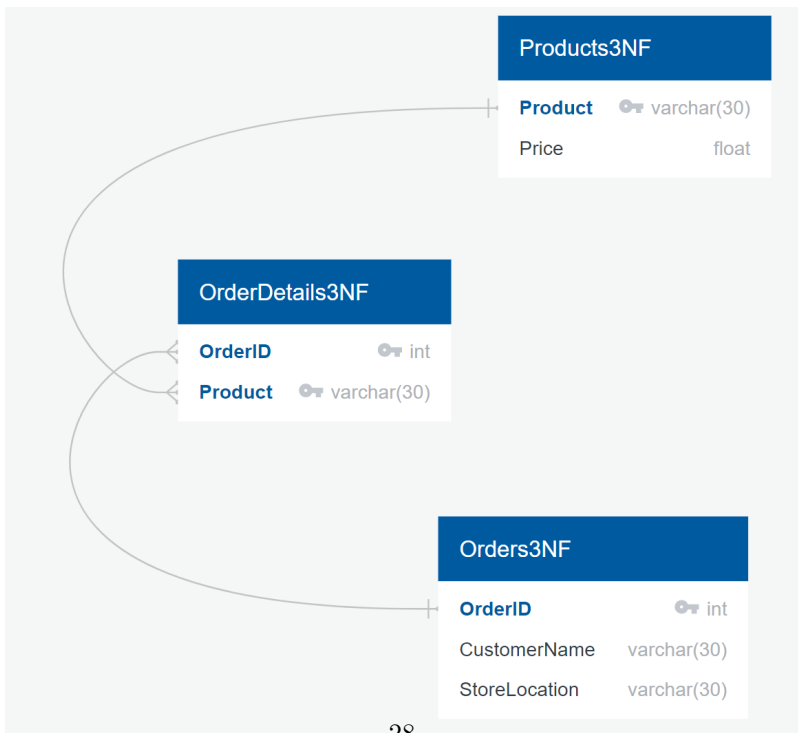
Tredje normalformen säger att:

- 2NF följs.
- Alla kolumner som inte är nyckel-kolumner är direkt beroende av de nyckel-kolumner som finns i tabellen men inte av varandra.

Vi ser i Figur 2.10 att "Price" i högsta grad är beroende av "Product", vilket då bryter mot regeln för 3NF. För att transformera tabellen enligt 3NF bryter vi ut ytterligare en tabell, "Products". Se Figur 2.11.



Figur 2.10: Normalisering i enlighet med 2NF



Figur 2.11: Ytterligare normalisering i enlighet med 3NF.

Nu är alla icke-nyckelkolumner enbart beroende av nyckelkolumnerna i tabellen och kraven för 3NF är uppfyllda.

Det här är ett förenklat exempel. Denna struktur skulle tids nog introducera redundans i “Orders”-tabellen. Om en kund handlar på flera “StoreLocation” kommer denna kunds namn dyka upp flera gånger i “CustomerName”-kolumnen. Detta tar upp onödig lagringsplats och kan försvåra uppdateringar i kolumnen. När redundans uppstår får stegen gås igenom igen och fler tabeller får brytas ut vid behov. Exempelvis skulle en “Customers”-tabell göra det här schemat mer effektivt, då det skulle innebära att tabellen “Orders” använder “CustomerID” för att hämta information om en kund i “Customers”-tabellen. Detta motverkar att samma namn dyker upp flera gånger i “Orders”-tabellen och gör det lättare att uppdatera kundinformation.

2.6 Optimering

För att en databas ska fungera så effektivt som möjligt behöver hänsyn till vissa faktorer tas. Nedan kommer vi titta närmare på indexering, optimering av *queries*, *load*-balansering samt hur systemet använder *cache*-minnet för att slippa hämta samma information flera gånger.

Avsnitt 2.6.1 om indexering är ett centralt koncept som ofta används och diskuteras. Avsnitt 2.6.2 till och med Avsnitt 2.6.4 är mer fördjupande avsnitt och kan hoppas över om läsaren inte är intresserad.

2.6.1 Indexering

Indexering används för att hitta rader som uppfyller ett visst kriterium i en specifik kolumn. Genom att använda ett index kan systemet direkt söka bland de värden som är lagrade i indexet och därmed snabbt hitta de rader som matchar. Utan indexering måste systemet gå igenom tabellen rad för rad för att hitta matchande rader, vilket gör sökningen långsammare.

Notera att primärnycklar automatiskt utgör ett index. Om inget index finns och datan är utspridd slumpmässigt kallas detta en *heap*.

När indexering tillämpas på en kolumn kan systemet snabbare söka igenom kolumnen och utföra exempelvis de *joins* som demonstreras i Avsnitt 3.5.1. Görs en sökning eller filtrering efter alla personer som är över 18 år i en kundtabell hade resultatet returnerats snabbare om ålders-kolumnen är indexerad. Anledningen till att indexering inte alltid tillämpas beror på att `INSERT`-, `UPDATE`-, och `DELETE`-operationer tar längre tid när de utförs på indexerade kolumner, eftersom påverkade index måste uppdateras varje gång.

En avvägning av vad som är viktigast måste därför göras beroende på vad tabellen används till. Om vi exempelvis har en tabell som primärt används för att lagra information är det viktigare att det går snabbt att lägga till fler rader än att utföra sökningar. I det fallet är indexering inte att föredra.

De vanligaste indextyperna är klustrade (`CLUSTERED INDEX`) samt icke-klustrade index (`NONCLUSTERED INDEX`).

I ett klustrat index lagras data fysiskt enligt den nyckel vi valt att använda oss av. Den faktiska ordningen i tabellen ändras alltså baserat på det index som specificerats. Detta gör att varje tabell högst kan ha ett klustrat index.

Icke-klustrad indexering skapar en separat kopia av tabellen inklusive indexerade kolumnvärden som pekar på de rader som lagrar data i den ursprungliga tabellen. Ordningen i tabellen ändras därmed inte, utan blir enbart refererad till den plats där datan är lagrad. En tabell kan ha flera icke-klustrade index.

Huruvida klustrade eller icke-klustrade index ska användas beror på tillämpningsområdet. Klustrade index är att föredra när data ofta läses ut i sorterad ordning eller vid intervallsökningar, eftersom den fysiska sorteringen av tabellen möjliggör snabb åtkomst. Däremot kan de medföra högre resursanvändning vid uppdateringar, eftersom tabellens fysiska ordning kan behöva ändras.

Icke-klustrade index kan förbättra sökprestanda för specifika frågor utan att ändra den fysiska ordningen i tabellen. De är också att föredra när tabeller ändras ofta, då de generellt har mindre påverkan på prestanda vid tillägg och uppdateringar jämfört med klustrade index.

2.6.2 Query-optimering

Syftet med *query*-optimering är att minska mängden resurser systemet använder för att hämta information. Optimeringen sker automatiskt via *query-plans*. I korthet innebär det att systemet försöker optimera den operation vi utför. Vi kan påverka optimeringen genom att deklarerar index, specificera *query-hints* när en förfrågan görs eller genom att använda kommandot `UPDATE STATISTICS`. Det är utanför denna bokens omfattning att gå in på detaljer men vi nämner det kort i nästa stycke.

Query-hints används för att skriva över de instruktioner som *query-plan* automatiskt följer. `UPDATE STATISTICS`-kommandot används för att uppdatera den statistik systemet internt använder sig av när det optimerar vår *query*. Det kan vara användbart i samband med att vi “batchar” in data från ett annat system.

i En *batch* (“sats” eller “omgång”) innebär att en större mängd data eller operationer bearbetas samtidigt, snarare än att de hanteras en åt gången. Det är en vanlig metod inom databehandling där uppgifter utförs i grupper för att effektivisera processer.

2.6.3 Load-balansering

Syftet med *load*-balansering är att minska trycket på en enskild server när *queries* görs. Detta uppnås genom att trafiken slussas till flera servrar istället för en. Detta sker inte automatiskt i MSSQL och måste implementeras manuellt om det ska användas. *Load*-balansering har likheter och implementeras ofta i samband med återställningsstrategier, vi återkommer till dessa i Avsnitt 5.2.

i En *server* är en dator som förmedlar information eller tillhandahåller tjänster till andra datorer (klienter) i ett nätverk. Klienterna skickar förfrågningar till servern som svarar genom att utföra en tjänst, exempelvis leverera en fil eller bearbeta en *query*. En server finns vanligtvis i ett *server-room* tillsammans med många fler servrar. En hel byggnad full med *server-rooms* kallas *data-center*.

Det är utanför denna bokens omfång att gå in på detaljer men vi nämner kort två exempel på algoritmer för hur denna fördelning kan ske:

- *Round-robin* - varje tillgänglig server blir i turordning tilldelad en del av den inkommande trafiken. När alla servrar blivit tilldelad en del börjar processen om.
- *Least-connection* - inkommande trafik blir riktad mot den server som har minst antal aktiva anslutningar.

2.6.4 *Cache*-hantering

Caching innebär att en kopia eller en referens till den data vi vill använda sparas i systemets interna minne. Detta kan ske både på hårdvaru- och mjukvarunivå, vilket förbättrar prestanda och möjliggör snabbare åtkomst till data. Hårdvara syftar på de fysiska delar som en maskin består av, exempelvis en hårddisk eller en processor. Mjukvara syftar på de program eller applikationer som körs via hårdvaran, exempelvis operativsystem.

I mjukvara används *cache* för att temporärt lagra data som applikationer ofta använder. När en användare begär data, exempelvis en databasfråga eller en webbsidehämtning kan programvaran spara en kopia av den begärda datan i *cache*-minnet. Nästa gång samma begäran görs kan systemet leverera informationen direkt från *cache*, vilket är snabbare än att genomföra samma operation från början igen.

I hårdvara används *cache* främst på processornivå. Processorer har olika nivåer av *cache*-minnen (L1, L2, L3) där L1 är minst men snabbast och L3 är störst men långsammast. Dessa minnen är effektiva

och kan hålla de mest frekvent använda instruktionerna och data nära sina processorkärnor. Detta minskar behovet att hämta data från det långsammare *RAM*-minnet eller hårddiskarna.

i Exempelvis vid inköp av en persondator är det nyttigt att förstå olika minnestyper och deras funktioner. Nedan följer förklaringar.

- *Cache* - generellt sett är *cache*-minnet något som vanliga användare fokuserar på. Däremot kan det vara viktigt för användare som behöver snabb åtkomst till frekvent använd data, såsom vid större matematiska beräkningar eller komplexa simuleringar. *Cache*-minnet är mycket snabbt och används för att tillfälligt lagra data som processorn ofta använder.
- *RAM* - *RAM*-minnet hanterar korttidslagring av data och påverkar de flesta användare. När ett program körs används *RAM* för att lagra den information som programmet behöver för att fungera effektivt. Ju mer *RAM*-minne som finns tillgängligt, desto fler program kan köras samtidigt utan att systemet blir långsamt.
- Hårddisk - hårddisken är en lagringsenhet där data sparas långsiktigt. Att läsa eller skriva data till en traditionell hårddisk är relativt långsamt eftersom den har mekaniska delar. En *solid state drive* eller *SSD* är snabbare, då den saknar rörliga delar. Ju större hårddisk eller *SSD*, desto mer data kan lagras.

Anledningen till att *cache*-minne inte används för all typ av lagring är främst att information ska hållas så nära processorkärnan som möjligt, då det ger lägst svarstid. Ett större minne skulle därmed innebära längre svarstider. Medan *RAM*-minnet kan vara tiotals *gigabyte* och disk-utrymmet kan nå flera *terabyte* är *cache*-minnet vanligtvis enbart ett fåtal *megabyte*. Mängden information som lagras i *cache*-minnet är således begränsad och reserverad för data som kräver snabb åtkomst. Se Tabell 2.3 för mer information kring informationsenheter.

Tabell 2.3: Informationsenheter samt den storlek de representerar.

Informationsenhet	Storlek
Bit	Binär siffra, 1 eller 0
Byte	8 bit
Kilobyte	1024 byte
Megabyte	1024 kilobyte
Gigabyte	1024 megabyte
Terabyte	1024 gigabyte
Petabyte	1024 terabyte
Exabyte	1024 petabyte
Zettabyte	1024 exabyte

När en *query* utförs tittar systemet först i *cache*-minnet. Finns den sökta datan där returneras den direkt, annars utförs *queryn* i sin helhet.

Ett par exempel på *caching* som sker automatiskt i MSSQL och MySQL:

- *Query-caching* - lagrar resultat från en *query*. Om en identisk *query* körs flera gånger finns alltså resultatet redan lagrat. Detta är mindre effektivt om datan ändras ofta.
- *Rad-caching* - lagrar resultat på rad-nivå. Om en rad frekvent förekommer i våra resultat kan den sparas i *cache*-minnet för snabbare åtkomst.
- Databasens *buffer-cache* - målet med *buffer-cachen* är att lagra både sådan data vi hämtar ofta och data vi nyligen hämtat. En del av serverns *RAM*-minne är dedikerat till den här processen, vilket gör det betydligt snabbare än att hämta datan från hårddisken.

2.7 Designval

När en relationell databas skapas är det viktigt att strukturera tabeller på ett ändamålsenligt sätt för att optimera datahämtning och analys. Detta uppnås ofta genom att kategorisera tabeller som antingen “*fact*”-tabeller eller “*dim*”-tabeller.

Fact-tabeller innehåller kvantitativ data som representerar händelser eller transaktioner, såsom individuella kundordrar, antal klick på en annons eller betyg för produkter. Förutom mätvärden, som antal eller belopp, inkluderar *fact*-tabeller också nycklar som refererar till *dim*-tabeller. Detta gör det möjligt att analysera data, exempelvis för att identifiera den kund som handlar mest eller vilken produkt som har högst betyg.

Dim-tabeller innehåller deskriptiv information som ger viktig kontext till *fact*-tabellerna, såsom kund-, annons- eller produktnamn. Dessa tabeller kan användas för att filtrera eller gruppera de mätvärden som finns i *fact*-tabeller.

Som exempel kan vi tänka oss en “TransactionDetails”-*fact*-tabell. Den innehåller bland annat kolumnerna “Quantity”, “PriceAtPurchase”, “TotalPrice” och “ProductID”. För att *fact*-tabellen ska kunna utföra beräkningar på produktnivå, såsom “Quantity” multiplicerat med “PriceAtPurchase” och på så sätt ta fram “TotalPrice” per produkt, krävs det att *fact*-tabellen är kopplad till en *dim*-tabell. I detta exempel skulle *dim*-tabellen vara “Products”. Fördelen med detta är att när en produkts information behöver uppdateras, behöver uppdateringen endast ske en gång - i den rad produkten beskrivs i “Products”-tabellen. Detta säkerställer också att *fact*-tabellen använder korrekt information varje gång en beräkning utförs.

De två vanligaste databas-scheman kallas *STAR* och *Snowflake*, på grund av deras utseenden. Gemensamt för dessa är att centralt finns en eller flera *fact*-tabeller och från dem byggs *dim*-tabeller ut i ett stjärn- eller snöflingemönster. Tanken är att *fact*-tabellen effektivt ska kunna referera till *dim*-tabeller för att kunna utföra aggregeringar och beräkningar.

2.7.1 STAR

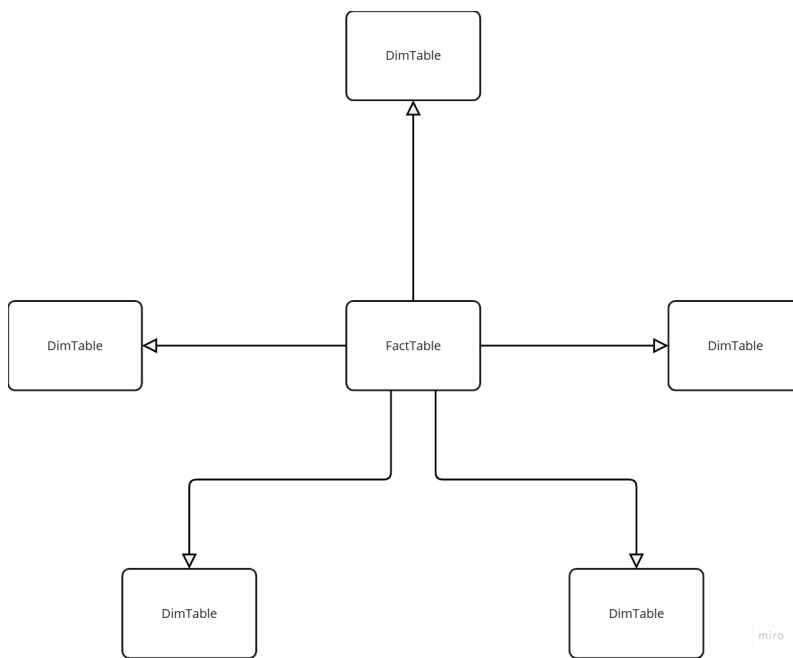
STAR-schemat har en enkel arkitektur där *fact*-tabellen befinner sig centralt och *dim*-tabeller placeras direkt angränsande, spridda runt om. Den här schema-typen är ofta delvis normaliserad. Det innebär att *dim*-tabeller kan innehålla mer information än bara dess referensnyckel och redundans kan uppstå. Exempel på detta är om en “Customers”-tabell har en “City”-kolumn. Eftersom flera kunder kan komma från samma stad skulle staden refereras till onödigt många gånger.

Fördelar:

- Enkelt att förstå och implementera.
- *Queries* blir simplare eftersom mängden *dim*-tabeller är begränsad.
- *Queries* blir snabbare och lättare att förstå.

Nackdelar:

- Datakvaliteten kan variera på grund av redundans.
- Tar upp mer minne än *Snowflake*.
- Möjligheterna för avancerade *Queries* är begränsade.



Figur 2.12: Ett exempel på ett STAR-schema med en fact-tabell i mitten och dim-tabeller spridda runtom.

2.7.2 Snowflake

Snowflake-schemat är en utveckling av *STAR*-schemat där *dim*-tabeller bryts ner ytterligare. Därmed liknar schemat en snöflinga till utseendet. Antag att vi har *dim*-tabellen "Product" som innehåller all information om produkter. Tabellen skulle kunna brytas ner till exempelvis "ProductCategory", "ProductDetails" och "ProductColor".

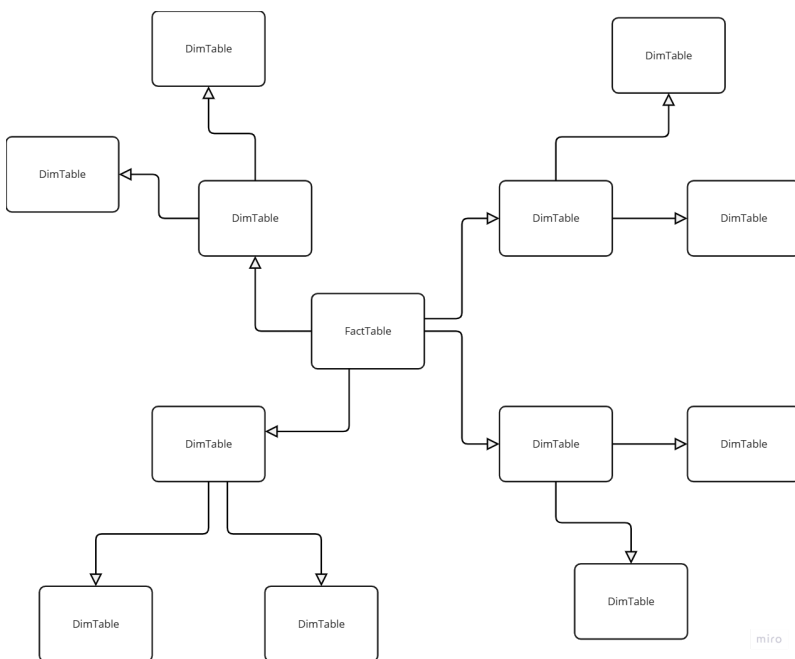
Ovanstående är ett exempel på normalisering, vilket kan göra schemat mer komplext att arbeta med och underhålla. Men i vissa fall kan det ge bättre datakvalitet och minska lagringsutrymmet som krävs.

Fördelar:

- Skalbart, då alla tabeller är minimalistiska och därmed lätta att förstå. Det är därför lätt att utöka schemat.
- Gör det lättare att hämta stora mängder information.
- Förstärker datakvalitet.

Nackdelar:

- Att hämta information från många olika tabeller kan bli både komplicerat och resurskrävande vilket kan göra att *queries* tar lång tid att exekvera.
- Kan vara mycket arbete att implementera, eftersom tabeller bryts ner i flera beståndsdelar vilket resulterar i fler tabeller.



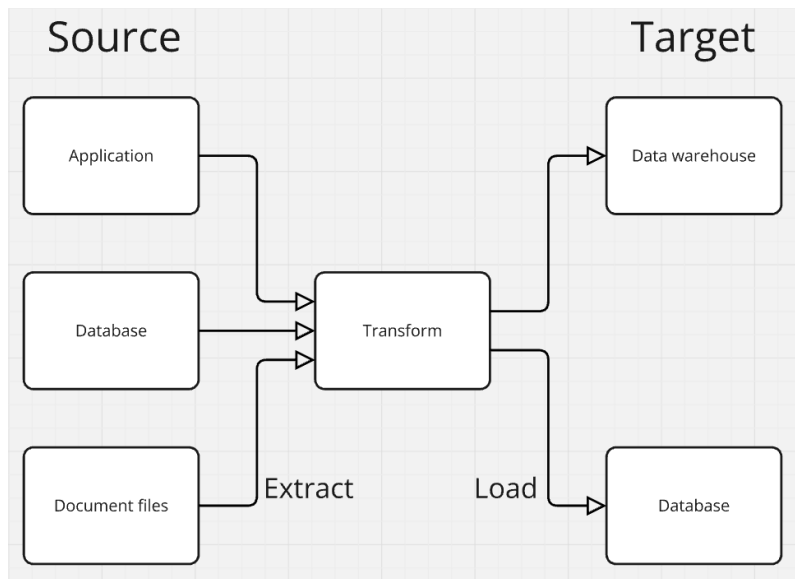
Figur 2.13: Ett exempel på ett Snowflake-schema med en fact-tabell i mitten och flera lager av dim-tabeller runt om.

Skulle databasen innehålla väldigt stora mängder data kan vi istället

titta på en icke-relationell databas-lösning, då dessa ofta är lättare att skala upp än relationella databaser. Icke-relationella databaser behandlas i Kapitel 6.

2.8 ETL

ETL står för *Extract-Transform-Load* och beskriver en process där data extraheras från en *input*-källa, transformeras och laddas in till en *output*-datakälla. Ett vanligt förekommande begrepp är ”*ETL Pipeline*”, vilket beskriver datas färd från insamling och transformering till den slutdestination där den ska användas. Se Figur 2.14 för ett visuellt exempel. Till vänster i bilden ser vi de källor data kan hämtas från, i mitten utförs någon form av datatransformation och sedan skickas datan vidare till en destination till höger.



Figur 2.14: ETL-visualisering.

Extract – i det här steget hämtas data in från en eller flera källor

såsom applikationer, databaser eller dokumentfiler. Datan skickas sedan vidare till en temporär lagringsplats där nästa steg, *transform*, genomförs.

Transform – i det här steget bearbetas data för att säkerställa att den anländer till slutdestinationen som är “load”-steget i önskat format. I *transform* genomförs också *data cleaning*, vilket är ett vedertaget begrepp. Här hanteras exempelvis saknad data, dubletter och inkonsekventa dataformat. Ett exempel på inkonsekvent data kan vara att datum från en källa är skrivna på formatet YYYY-MM-DD medan de från en annan källa är skrivna på formatet MM/DD/YYYY.

Load – i det sista steget flyttas den färdiga datan till dess slutdestination. Vart datan hamnar beror på kontext. Ett stort multinationellt företag har andra krav och behov på datalagring än en privatperson. I enklare fall kan data lagras i dokumentfiler. Om den används frekvent eller om det konstant tillkommer ny data kan den lagras i en SQL-databas. Större mängder data kan förvaras i ett *data warehouse* eller en *data lake*. Dessa koncept beskrivs i kommande två avsnitt. Data som är redo att användas skulle exempelvis också direkt kunna skickas till *Business-Intelligence*-applikationer. BI-applikationer har dock ofta en underliggande databas som även lagrar den data som används för att skapa rapportering eller analyser och prediktioner.

2.8.1 Data Warehouse

Ett *data warehouse* kan i sitt grundutförande beskrivas som en lagringsplats för större mängder strukturerad data. Syftet är att centralisera data från flera källor. Ska diverse funktionalitet såsom analys eller maskininlärning tillämpas så måste den funktionaliteten kopplas på manuellt.

Värt att notera är att större aktörer som Amazon ofta byggt in funktionalitet i sina *data warehouse*-lösningar. Detta är en stor anledning till att *data warehouses* blivit populära. Via dessa tjänster kan funktionalitet samlas på en plats. Data kan hämtas in och analyseras, därefter kan insikter appliceras, allt under samma “tak”.

Ofta är *data warehouses* strukturerade med stöd och funktionalitet

för SQL. Det gör dem till en vanlig lösning för rapportering och analys av data.

Exempel på *data warehouse*-helhetslösningar är Snowflake, Google BigQuery, Amazon Redshift och Microsoft Azure Synapse Analytics. Dessa exempel är moln-baserade lösningar.

En möjlig nackdel med *data warehouses* är att både implementering samt underhåll kan vara resurskrävande om en organisation själv utvecklar det. Moln-lösningar blir allt mer vanligt förekommande och kan vara mycket mindre resurskrävande att implementera och underhålla.

2.8.2 Data Lakes

Data lakes är utformade för att lagra all typ av data, oavsett om den är strukturerad eller ostrukturerad. Detta gör att NoSQL-lösningar är vanligare i *data lakes*, mer om NoSQL i Kapitel 6. Ostrukturerad data kan exempelvis vara dokument, ljudfiler eller bilder. Det betyder att en *data lake* kan lagra fler varianter av data än exempelvis ett *data warehouse*, som främst behandlar strukturerad data. Detta har resulterat i att *data lakes* ökat i popularitet på senare tid då de gör det möjligt att lagra stora mängder rådata.

Det finns ett antal tredjepartsapplikationer som erbjuder *data lake*-tjänster. Exempelvis Lake Formation, Azure Data Lake och Google BigLake.

Data som anländer till en *data lake* behöver hanteras i samband med att den ska användas. Görs inte detta riskerar vi att få en *data lake* som är för stor och oorganiserad för att använda, detta brukar då kallas för en "*data swamp*".

2.8.3 Data Lakehouses

På senare tid har större aktörer som Amazon och Google börjat implementera funktionalitet som traditionellt har varit kopplat till *data warehouses* i *data lakes* och vice versa. Gradvis har detta gjort distinktionen mellan de två mer svårtolkad. Vi kommer inte gå in närmare

på det i den här boken men det kan vara bra att ha med sig den informationen.

2.9 Molntjänster/on-premises

IT-tjänster för datahantering kan delas in i molntjänster och *on-premise* som ofta benämns som *on-prem*-lösningar. Molntjänster är IT-tjänster som tillhandahålls via ett nätverk, vanligtvis internet. Molntjänsterna kan erbjuda skalbara lösningar med mindre behov av tekniskt underhåll från organisationens sida, men kan också innebära mindre kontroll i aspekter som säkerhet och dataintegritet. *On-prem* innebär att tjänsterna körs lokalt inom organisationen, vilket möjliggör mer personliga lösningar men kan innebära höga kostnader för uppstart och förvaltning. Valet mellan dessa två alternativ beror ofta på organisationens storlek, budget, säkerhets- samt skalbarhetsbehov.

Vanligt förekommande molntjänster erbjuds bland annat av Microsoft Azure, Amazon Web Services och Google Cloud Platform. Nedan tittar vi närmare på de två lösningarna.

i Molntjänster har revolutionerat hur företag hanterar data-lagring och drift genom att erbjuda kostnadseffektiv och skalbar infrastruktur på ett lättillgängligt sätt. Detta har möjliggjort att verksamheter snabbt kan expandera, utnyttja stora mängder data och anpassa resurser baserat på behov.

2.9.1 Molntjänster

Molntjänster är tillgängliga för både privatpersoner och organisationer och är skalbara för att möta användares specifika krav. Det här gör det möjligt att skala upp eller skala ner projektets storlek utifrån behov.

Eftersom kostnader generellt baseras på hur tjänster används är det här en flexibel och effektiv lösning. Användare av molntjänster behöver exempelvis inte själva sätta upp egna servrar och underhålla dessa.

Om mer kapacitet behövs kan detta dessutom enkelt skalas upp och behövs mer funktionalitet kan också implementeras. Generellt sett mot en tillkommande kostnad.

Värden för molntjänsten står också för driftsäkerheten. Vanligtvis befinner sig de projekt man arbetar med i ett datacenter. För att garantera säkerheten för projektet är detta datacenter ett av flera, vilka agerar *backup* till varandra. Vid den händelse att ett strömavbrott, en naturkatastrof eller annan fysisk orsak gör att ett datacenter går *offline* eller förstörs så tar ett annat center över och agerar värd. För att försäkra att inte flera center går *offline* samtidigt brukar de separeras geografiskt genom att placeras i olika regioner, länder eller till och med kontinenter. Endast om alla datacenter går *offline* samtidigt bryts anslutningen helt. Det är ytterst osannolikt att det sker men är teoretiskt möjligt.

Övrig säkerhet sköts också av värden, exempelvis fysisk säkerhet vid datacenter och kryptering för att skydda organisationens filer. Personlig säkerhet som användarbehörighet och datalagringsstrategier sköts av kunden. Säkerhetsfrågan är ofta central när det gäller molntjänster, särskilt för verksamheter där data absolut inte får äventyras.

Ett tydligt exempel på sådan verksamhet är rättsväsendet. Om känsliga dokument från domstolsförhandlingar och pågående utredningar skulle läcka ut kan det påverka både rättsfall och enskilda individers säkerhet.

Energisektorn är ett annat tydligt exempel, där exempelvis elnät och vattenförsörjning ingår. En fientlig aktör som får tillgång till dessa system skulle potentiellt kunna orsaka stora störningar i samhällsviktiga funktioner i samband med exempelvis krig eller utpressning av hacker-grupper. Detta har konsekvenser för både den nationella säkerheten och för individuella medborgare.

i År 2017 avslöjades att Transportstyrelsen hade exponerat känslig information till icke säkerhetskontrollerad personal utomlands. I stort sett hela Transportstyrelsens IT-miljö gjordes tillgänglig, bland annat det svenska registret över körkort (inklusive bilder), känslig information om infrastruktur och agenter med skyddad identitet. Detta dataläckage resulterade i att generaldirektören för Transportstyrelsen samt två ministrar från regeringen avgick vilket indikerar allvaret i händelsen.

2.9.2 On-premises

On-prem innebär att alla IT-tjänster och system finns lokalt inom organisationens egna fysiska infrastruktur, till exempel i en serverhall. Det här ger organisationen mer kontroll över tjänsterna, vilket kan vara fördelaktigt för just säkerhet och dataintegritet. Men det innebär också att organisationen själv måste tillhandahålla både hårdvara och mjukvara.

Uppstartskostnaden för hårdvaran och infrastruktur så som servrar, nätverks- och lagringsenheter kan vara hög. Systemet ska också både underhållas och vara redo för expansion vid behov. Kostnaden är också konstant oavsett till vilken grad systemet nyttjas. *On-prem*-lösningar ger alltså mer kontroll, men kräver ofta signifikant mer investeringar vid uppstart.

2.10 Big Data

Ett begrepp som ligger utanför denna bokens omfång men som ändå kan vara bra att ha hört talas om är ”*big data*”. *Big data* refererar till samlingar av data som är så stora att vanliga system inte kan hantera dem. Begreppet omfattar både strukturerad och ostrukturerad data som vanligtvis växer exponentiellt över tid, exempelvis innehåll på sociala medie-plattformar som konstant tar emot ny data.

Med tiden har verktyg med syfte att hantera *big data* utvecklats. Bland annat kan det användas i AI-tillämpningar som maskininlär-

ning eller andra mer avancerade analysmetoder. Populära tjänster för detta är Apache Hadoop, Spark och Kafka. Dessa verktyg är grundläggande inom *big data* och används för att lagra, bearbeta och analysera stora datamängder. De används som fundamentala ramverk för molnbaserade helhetslösningar som Google BigQuery, Amazon EMR och Microsoft Azure HDInsight.

2.11 Övningsuppgifter

1. Beskriv kortfattat de tre abstraktionsnivåerna i en databas och vad de representerar.
2. Beskriv vad följande datatyper representerar.
 - a) INTEGER
 - b) FLOAT
 - c) CHAR
 - d) VARCHAR(__)
 - e) DATE
 - f) DATETIME
3. Förklara begreppen transaktion och “*all-or-nothing*”-regeln.
4. Beskriv akronymen CRUD. Vad innebär de olika momenten?
5. Beskriv följande begränsningar.
 - a) UNIQUE
 - b) NOT NULL
 - c) PRIMARY KEY
 - d) CHECK
 - e) DEFAULT
6. Beskriv följande relationer.
 - a) “*One-to-one*”
 - b) “*One-to-many*”
 - c) “*Many-to-many*”
7. Beskriv kortfattat akronymen ACID samt vad de olika momenten innebär.
8. Beskriv syftet med normalisering.

9. Vad händer med tabeller ju högre grad av normala former som tillämpas?
10. Förklara kortfattat vad indexering är och hur det relaterar till prestanda.
11. Ge exempel på vad som bör lagras i en *fact* respektive *dim*-tabell.
12. Beskriv skillnaden mellan ett STAR-schema och ett Snowflake-schema. Beskriv även för och nackdelar.
13. Beskriv kortfattat ETL-processen och vad varje steg innebär.
14. Beskriv kortfattat vad ett *data warehouse* är.
15. Beskriv hur en *data lake* skiljer sig från ett *data warehouse*.
16. Beskriv fördelar och nackdelar med molnlösningar kontra *on-prem* lösningar.

Kapitel 3

Grundläggande SQL

Detta kapitel introducerar programmeringsspråket *Structured Query Language* (SQL) som används för att interagera med relationella databaser. Detta innefattar grundläggande syntax och konventioner, *CRUD*-flödet, *queries* samt funktioner, vyer, lagrade procedurer och index. I Avsnitt 3.4 introducerar vi databasen “Köksglädje” som kommer användas på ett flertal ställen i boken.

i I de flesta fall kommer de kodexempel vi demonstrerar vara likadana för Microsoft SQL Server (MSSQL) och MySQL. I de fall där det skiljer sig kommer vi att demonstrera hur koden ser ut i både MySQL och MSSQL.

3.1 Syntax

Programmeringsspråket SQL är uppdelat i ett antal språkliga element. Dessa är bland annat:

- *Query* är en samling instruktioner som ges till en databas. En *query* består av ett eller flera *statements*.

- *Statement* är den instruktion som ges till databasen. Exempelvis `SELECT`, `DELETE`, `UPDATE`, `CREATE`, `DROP` och `ALTER`. Flera *statements* i en *query* separeras med semikolon “;”.
- *Clause* är en beståndsdel av ett *statement*. Det kan exempelvis filtrera eller begränsa resultatet. Nedan listas några *clauses*.

`SELECT` används för att specificera vilka kolumner som ska läsas ut.

`FROM` används för att ange vilken tabell kolumner finns i.

`WHERE` används för att styra vilka rader som är en del av resultatet från en *query*, exempelvis vilka rader som ska läsas ut eller vilka rader som ska uppdateras.

`ORDER BY` används för att sortera ett resultat.

- *Expressions* är en kombination av operatorer och symboler som returnerar ett värde eller *result set*. Ett *result set* är den data som returneras av en *SQL-query*, exempelvis en tabell från en *SELECT-query*. Se Tabell 3.1 för olika operatorer.
- *Predicate* är ett *expression* som returnerar sant, falskt eller okänt. *Predicate* kan användas för att diktera villkoret i en *clause*. Exempelvis `WHERE Amount > 100` eller `WHERE AGE BETWEEN 20 AND 30`.
- *Keywords* är definierade ord i SQL. Det finns reserverade *keywords* som exempelvis `SELECT`, `GROUP` och `CREATE`. Det finns även icke reserverade *keywords* som exempelvis `COLUMN`, `DAY` och `MONTH`.
- *Identifiers* är namn på objekt i en databas, exempelvis tabeller eller kolumner. I de fall en *identifier* skulle ha samma namn som ett reserverat *keyword* behöver det omges av dubbla citations-tecken. Generellt sett bör det undvikas att skapa en *identifier* med samma namn som ett *keyword*.
- *Insignificant whitespaces* eller blanka tecken som exempelvis mellanslag ignoreras i *SQL-queries*. Detta gör att SQL-kod kan formateras på ett läsvänligt sätt. I kodblocket nedan ser vi att

raderna är separerade med tomma rader utan att funktionaliteten påverkas, det påverkar dock läsbarheten negativt i detta fall.

```
UPDATE Products

SET Price = Price - 50

WHERE ProductName = 'Bicycle';

SELECT *
FROM Products;
```

I ovanstående kodblock ser vi några exempel på SQL-element.

- UPDATE, SET och WHERE är alla tre *keywords*, UPDATE är även ett *statement* medan SET och WHERE är *clauses*.
- UPDATE följt av “Products” som är en *identifier* specificerar att tabellen “Products” ska uppdateras.
- SET följt av “Price” som också är en *identifier* specificerar att kolumnen “Price” ska uppdateras. “Price - 50” är ett *expression* som anger att priset ska sänkas med 50.
- WHERE filtrerar vilka rader som ska uppdateras. I detta fall används ett *predicate* för att endast uppdatera rader där “ProductName” är “Bicycle”.
- Mellan raderna finns *insignificant whitespaces* som inte påverkar koden. I detta fallet påverkar det dock läsbarheten negativt.
- Sammansatt är all kod ett statement och i detta fall består vår *query* av två *statements*. När vi utför *queryn* sänks priset för “Bicycle” med 50 kr i tabellen “Products”. Detta är en följd av det första *statementet*. I det andra *statementet* läser vi ut hela tabellen genom att skriva **SELECT * FROM PRODUCTS;**. Notera att våra två statements separeras genom semikolon “;”.

Notera, vi har här skrivit kod för att demonstrera koncepten och använder alltså ingen databas.

Tabell 3.1: Några vanligt förekommande operatorer i SQL.

Kategori	Operator	Beskrivning
Aritmetiska	+	Addition
Aritmetiska	-	Subtraktion
Aritmetiska	*	Multiplikation
Aritmetiska	/	Division
Aritmetiska	%	Modulo
Jämförelser	=	Lika med.
Jämförelser	>	Större än.
Jämförelser	<	Mindre än.
Jämförelser	>=	Större än eller lika med.
Jämförelser	<=	Mindre än eller lika med.
Jämförelser	<>	Inte lika med.
Logiska	ALL	SANT om alla jämförelser i en uppsättning är SANT.
Logiska	AND	SANT om båda booleska uttrycken är SANT.
Logiska	ANY	SANT om någon av jämförelserna i en uppsättning är SANT.
Logiska	BETWEEN	SANT om operanden är inom ett visst intervall.
Logiska	EXISTS	SANT om en <i>subquery</i> innehåller några rader.
Logiska	IN	SANT om operanden är lika med ett av uttrycken i en lista.
Logiska	LIKE	SANT om operanden matchar ett mönster.
Logiska	NOT	Vänder värdet av en annan boolesk operator.
Logiska	OR	SANT om något av de booleska uttrycken är SANT.
Logiska	SOME	SANT om några av jämförelserna i en uppsättning är SANT.

3.2 Konventioner

Generellt sett bör SQL-koden vi skriver följa konventioner.

Det finns allmänt accepterade konventioner såsom att kommentarer ska användas där det är nödvändigt och i de fallen ska vi kommentera “varför vi gör något” och inte “vad vi gör”. Detta eftersom “vad vi gör” kan utläsas från själva koden och blir alltså onödig upprepning. För att kommentera SQL-kod används två bindestreck “--” följt av kommentaren. Om kommentaren sträcker sig över flera rader används “/*” för att påbörja kommentaren och “*/” för att avsluta den.

Alla nyckelord bör skrivas med versaler, trots att nyckelord i SQL inte är skifteslägeskänsliga. Det innebär att även om exempelvis nyckelordet `SELECT` skrivs som `sEleCT` så fungerar fortfarande koden, men det följer inte konventionen, försämrar läsbarheten och bör därför inte skrivas på det sättet.

Identifiers ska vara deskriptiva och namngivning ska vara konsekvent.

Insignificant whitespaces ska användas för att formatera koden på ett sätt som gör den enklare att läsa och tyda.

Det existerar även olika konventioner för olika SQL-varianter. I MS-SQL används *PascalCase* för namngivning av tabeller och kolumner. Vi skriver exempelvis “EnStorTabell”. I MySQL används *snake_case* för namngivning av tabeller och kolumner, vi skriver exempelvis “en_stor_tabell”.

Den intresserade läsaren kan söka på internet för att läsa mer om konventioner.

! I kodexemplen kommer vi följa konventionerna för MSSQL. I de fall koden för MSSQL och MySQL skiljer sig åt demonstrerar vi kod för båda databashanterarna och följer respektive databashanterares konventioner.

3.3 CRUD

I detta avsnitt kommer vi att demonstrera hur vi utför *Create*, *Read*, *Update* och *Delete*-momenten i *CRUD*-flödet.

3.3.1 Create

Create-momentet innefattar operationen `INSERT INTO`, som används för att lägga till rader i en tabell. Skapandet av databaser och tabeller inkluderas inte i *CRUD*-flödet. Det kommer ändå att demonstreras i detta avsnitt eftersom det ligger nära till hands att utföra det här. Nyckelordet `CREATE` används för att skapa databaser och tabeller. Som vi senare i detta kapitel kommer att demonstrera kan `CREATE` även användas för att skapa andra objekt, exempelvis vyer och lagrade procedurer.

Vi börjar med att skapa en databas genom ett *CREATE-statement*. Vi skriver nyckelordet `CREATE` och specificerar att det är en databas med namnet “FruitStand”.

```
CREATE DATABASE FruitStand;
```

①

- ① Nyckelorden `CREATE DATABASE` används för att skapa databasen “FruitStand”.

i Semikolon “;” används för att separera flera *statements* i en *query*. Har vi endast ett *statement* behöver vi inte använda ett semikolon för att avsluta vårt *statement*, men det är en god vana att göra det.

Vi kan nu ange att det är i vår nyskapade databas vi vill utföra kommande operationer. Detta gör vi genom nyckelordet `USE` följt av namnet på databasen.

```
USE FruitStand;
```

①

- ① `USE` används för att specificera att kommande *queries* ska exekveras i databasen “FruitStand”.

Nyckelordet `CREATE` används även för att skapa tabeller i databaser. I samma *CREATE-statement* anges även kolumnerna som ska skapas i tabellen.

Vi skriver nyckelorden `CREATE TABLE` följt av önskat namn på tabellen. I samma *statement* anger vi kolumners namn och kolumners datatyp samt eventuella nycklar och begränsningar.

```
CREATE TABLE ProductsTable( ①
    ProductID INT PRIMARY KEY, ②
    ProductName VARCHAR(50) UNIQUE, ③
    Amount INT, ④
    Price INT ⑤
);
```

- ① Nyckelorden `CREATE TABLE` följs av “ProductsTable” för att skapa en tabell med namnet “ProductsTable”.
- ② Kolumnen “ProductID” skapas med datatypen `INTEGER`, och `PRIMARY KEY` anges för att göra kolumnen till tabellens primärnyckel.
- ③ Kolumnen “ProductName” skapas med datatypen `VARCHAR` där maximalt 50 tecken får användas. Även en *unique constraint* appliceras.
- ④ Kolumnen “Amount” skapas med datatypen `INTEGER`.
- ⑤ Kolumnen “Price” skapas med datatypen `INTEGER`.

i Används inte `USE` för att specificera vilken databas vi vill arbeta i kan vi skriva följande:
“`CREATE TABLE FruitStand.dbo.ProductsTable(...)`”.
Vi har specificerat att i databasen “FruitStand” under schemat “dbo” ska tabellen “ProductsTable” skapas. “dbo” är en akronym för “database owner” och är ett schema som automatiskt skapas i MSSQL om inte användaren strukturerar tabeller enligt egna definierade scheman. I MySQL används databasnamn för att separera tabeller snarare än scheman. Notera att databaser i MySQL benämns “schema” vilket kan vara en förvirrande aspekt när man växlar mellan MSSQL och MySQL.

För att lägga till rader i vår nyskapade tabell skriver vi ett `INSERT INTO-statement`. Vi använder oss av nyckelorden `INSERT INTO` följt av vilken tabell vi vill lägga till rader i. Därefter används nyckelordet `VALUES` följt av önskade värden för raden.

```
INSERT INTO ProductsTable          ①  
VALUES (1, 'Apples', 10, 20);      ②
```

- ① “ProductsTable” anges efter `INSERT INTO` som den tabell där rader ska läggas till.
- ② Önskade värden för alla kolumner specificeras i en parentes efter nyckelordet `VALUES`.

Vi kan även lägga till fler rader i ett *statement*. I koden nedan lägger vi till två rader.

```
INSERT INTO ProductsTable  
VALUES (2, 'Pears', 20, 25),          ①  
       (3, 'Bananas', 25, NULL);
```

- ① Parenteserna separeras med ett kommatecken.

Eftersom vi inte satte någon *constraint* på “Price”-kolumnen när vi

skapade tabellen kunde värdet sättas till NULL. Skulle vi försöka lägga till en ny rad med ett produktnamn som redan existerar i tabellen genereras ett felmeddelande eftersom vi satte begränsningen *unique* för kolumnen “ProductName”.

```
INSERT INTO ProductsTable  
VALUES (4, 'Apples', 30, 12);
```

①

- ① “Apples” som redan existerar för kolumnen “ProductName” anges som önskat värde.

Ovanstående kodexempel genererar följande felmeddelande:

Msg 2627, Level 14, State 1, Line 1 Violation of UNIQUE KEY constraint 'UQ_Products'. Cannot insert duplicate key in object 'dbo.ProductsTable'. The duplicate key value is (Apples).

Vi gjorde kolumnen “ProductID” till vår *primary key* i tabellen vilket innebär att kolumnen endast får ha unika attribut som är skilda från NULL. Försöker vi lägga till en rad där “ProductID” är NULL får vi återigen ett felmeddelande.

```
INSERT INTO ProductsTable  
VALUES (NULL, 'Bananas', 30, 12);
```

Vi får följande felmeddelande:

Msg 515, Level 16, State 2, Line 1 Cannot insert the value NULL into column 'ProductID', table 'FruitStand.dbo.ProductsTable'; column does not allow nulls. INSERT fails.

Nedanstående kod returnerar ett felmeddelande eftersom värdet “3” redan existerar i kolumnen “ProductID”. Felet uppstår eftersom tabellens *primary keys* kräver unika attribut.

```
INSERT INTO ProductsTable  
VALUES (3, 'Oranges', 30, 12);
```

Msg 2627, Level 14, State 1, Line 1 Violation of PRIMARY KEY constraint 'PK_Products'. Cannot insert duplicate key in object 'dbo.ProductsTable'. The duplicate key value is (3).

3.3.2 Read

Read-momentet består av operationen **SELECT** som läser ut data från en tabell.

För att utföra operationen skriver vi nyckelordet **SELECT** följt av vilka kolumner vi vill läsa ut och i vilken tabell kolumnerna finns.

```
SELECT * FROM ProductsTable; ①
```

- ① Genom att använda stjärnan “*” så väljs alla kolumner från “ProductsTable”-tabellen.

ProductID	ProductName	Amount	Price
1	Apples	10	20
2	Pears	20	25
3	Bananas	25	NA

Skulle vi endast vara intresserade av att läsa ut specifika kolumner kan vi specificera detta i ett *SELECT-statement*.

```
SELECT ProductName,  
       Price  
FROM ProductsTable; ①
```

- ① Endast “ProductName” och “Price”-kolumnerna läses ut.

ProductName	Price
Apples	20
Pears	25
Bananas	NA

3.3.3 Update

Update-momentet består av operationen `UPDATE` som används för att uppdatera rader i en tabell.

Följande kodexempel demonstrerar hur `UPDATE` används för att uppdatera rader i en tabell.

```
UPDATE ProductsTable      ①
SET Price = 25             ②
WHERE ProductName = 'Apples'; ③
```

- ① `UPDATE` används för att specificera att det är i “ProductsTable”-tabellen som operationen ska utföras.
- ② `SET` uppdaterar “Price”-kolumnen till 25 för alla rader som matchar villkoret i kommande *WHERE-clause*.
- ③ `WHERE` används för att specificera att enbart rader där “ProductName”-kolumnen har värdet “Apples” ska uppdateras.

Vi skriver ett *SELECT-statement* för att läsa ut raderna och verifiera att uppdateringen har utförts enligt förväntan.

```
SELECT Price,
       ProductName
FROM ProductsTable
WHERE ProductName = 'Apples';
```

Price	ProductName
25	Apples

Vi ser att produkten “Apples” nu har priset 25 precis som vi önskade.

3.3.4 Delete

Delete-momentet består av en operation, **DELETE**, som används för att radera rader i en tabell. Vi kommer i detta avsnitt även demonstrera **DROP** eftersom det ligger nära till hands. **DROP** används för att radera bland annat tabeller och databaser.

Vi använder nyckelordet **DELETE** följt av den tabell vi vill radera rader i. Vi behöver också specificera vilken eller vilka rader vi vill radera genom en *WHERE-clause*. Använder vi inte en *WHERE-clause* raderas alla rader i tabellen.

```
DELETE FROM ProductsTable ①  
WHERE ProductName = 'Apples'; ②
```

- ① “ProductsTable” anges som tabellen där rader ska raderas.
- ② En *WHERE-clause* används för att specificera att rader där kolumnen “ProductName” har värdet “Apples” ska raderas.

För att verifiera att raden tagits bort kan vi skriva ett *SELECT-statement*.

```
SELECT * FROM ProductsTable  
WHERE ProductName = 'Apples';
```

ProductID	ProductName	Amount	Price
-----------	-------------	--------	-------

Vi ser att raden med produktnamnet “Apples” har raderats precis som förväntat.

För att radera en hel tabell skriver vi ett *DROP-statement*. Vi använder nyckelorden **DROP TABLE** följt av namnet på den tabell vi vill radera.

```
DROP TABLE ProductsTable; ①
```

- ① “ProductsTable” specificeras efter **DROP TABLE** för att raderas.

Vi kan också radera en hel databas genom ett *DROP-statement*. Vi specificerar att det är en databas med nyckelorden `DROP DATABASE` följt av namnet på den databas vi vill radera.

```
DROP DATABASE FruitStand;
```

3.4 Introduktion av databasen Köksglädje

I detta avsnitt introduceras exempeldatabasen “Köksglädje” som vi kommer använda ett flertal gånger i boken. Databasen finns tillgänglig på bokens *GitHub*-sida. Vi börjar med att beskriva databasen i ord och genom ett *Entity-Relationship*-diagram (ER-diagram). Därefter läser vi in databasen och gör några utskrifter från den.

3.4.1 Beskrivning av databasen Köksglädje

Databasen “Köksglädje” är en exempeldatabas för en fiktiv svensk butikskedja som säljer köksutrustning. All data som representerar priser och transaktioner är i svenska kronor, SEK. I databasen “Köksglädje” finns det totalt 8 tabeller.

”CustomerContactLog”-tabellen har information relaterad till kampanjutskick mot kund och innehåller kolumnerna:

- ContactLogID som är tabellens primärnyckel.
- CustomerID som är varje kunds unika kundnummer.
- CampaignID som är varje kampanjs unika nummer.
- ContactDate som innehåller datum för eventuella kampanjutskick.

”Customers”-tabellen har information relaterat till specifika kunder och innehåller kolumnerna:

- CustomerID som är tabellens primärnyckel.
- FirstName
- LastName

- Email
- ApprovedToContact som visar om en kund får kontaktas.
- ActiveMember som visar om en kund just nu är medlem.
- JoinDate

”MarketingCampaigns”-tabellen har information om existerande kampanjer och innehåller kolumnerna:

- CampaignID som är tabellens primärnyckel.
- CampaignName som visar en kampanjs namn.
- StartDate
- EndDate
- DiscountPercentage
- CategoryID

”ProductCategories”-tabellen har information om de olika produktkategorier som finns och innehåller kolumnerna:

- CategoryID som är tabellens primärnyckel.
- CategoryName
- Description som beskriver de olika produktkategorierna.

”Products”-tabellen har information relaterad till de produkter som säljs och innehåller kolumnerna:

- ProductID som är tabellens primärnyckel.
- ProductName
- Description
- Price som visar produktens försäljningspris.
- CostPrice som visar produktens inköpspris.
- CategoryID

”Stores”-tabellen som har information relaterad till varje butik och innehåller kolumnerna:

- StoreID som är tabellens primärnyckel.
- StoreName
- Location
- ManagerName
- ContactNumber

"TransactionDetails"-tabellen har detaljerad information om varje transaktion och innehåller kolumnerna:

- TransactionDetailID som är tabellens primärnyckel.
- TransactionID
- ProductID
- Quantity
- PriceAtPurchase
- CampaignID
- TotalPrice

"Transactions"-tabellen har generell information om varje transaktion och innehåller kolumnerna:

- TransactionID som är tabellens primärnyckel.
- CustomerID
- StoreID
- TransactionDate
- TotalAmount

Notera att exempelvis "TotalPrice" från tabellen "TransactionDetails" representerar den totala summan per produkt ifall en kund köpt flera produkter. Det vill säga "PriceAtPurchase" multiplicerat med "Quantity". Detta kan jämföras med "TotalAmount" från "Transactions"-tabellen, som i stället visar totalbeloppet för en hel transaktion som kan innehålla flera produkter. "TransactionDetails"-tabellen innehåller alltså detaljerad information, medan "Transactions"-tabellen innehåller övergripande information. Skulle en transaktion från "Transactions"-tabellen innehålla fyra olika produkter så beskrivs dessa i detalj i "TransactionDetails"-tabellen. Denna distinktion är viktig att göra då varje produkt kan ha varierande pris, beroende på om en kampanj är aktiv eller inte. Denna information hämtas då från "MarketingCampaigns"-tabellen.

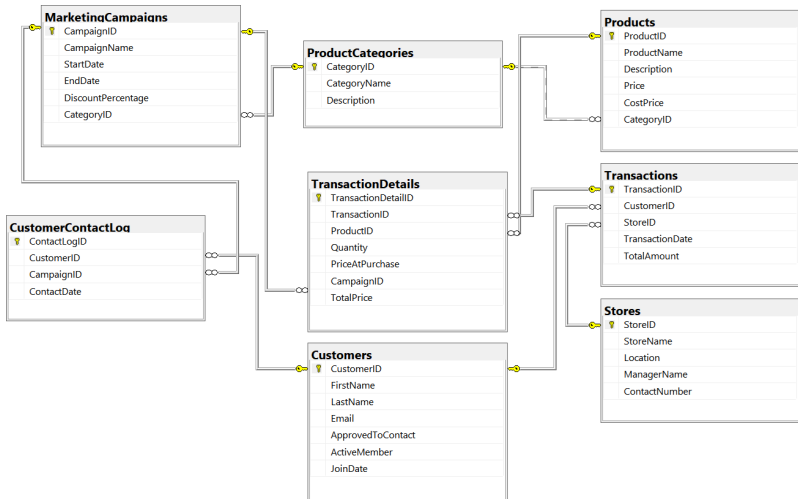
3.4.2 Entity-Relationship-diagram

Entity-Relationship-diagram benämns ofta som "ER-diagram" och används för att visa strukturen i en databas. I diagrammet visas tabeller (*entity*) och deras tillhörande kolumner samt relationerna (*relations*-

hip) mellan tabellerna.

För att generera ER-diagram finns det inbyggda verktyg i både MySQL och MSSQL. I MySQL klickar vi på “Database” och sedan “Reverse engineer” för att lägga till de tabeller vi önskar i vårt diagram. I MSSQL använder vi “Object Explorer” och högerklickar på “Database Diagrams” för att sedan välja “New Database Diagram”. Andra verktyg som kan användas för att skapa ER-digram är “Lucidchart”, “SQL Database Modeler” eller “Quick Database Diagrams”. Du kan snabbt lära dig hur du till exempel använder “Quick Database Diagrams” genom att gå till deras hemsida: <https://www.quickdatabasediagrams.com/>. De flesta av de ER-diagram som visades i Kapitel 2 skapades med hjälp av “Quick Database Diagrams”. Exempelvis Figur 2.6.

I Figur 3.1 ser vi ett ER-diagram för databasen “Köksglädje”. Vi kan se vilka tabeller som finns i databasen och respektive tabells kolumner. Vi ser även relationerna mellan tabellerna.



Figur 3.1: ER-diagram för “Köksglädje” som skapats i MSSQL.

3.4.3 Inläsning av databasen Köksglädje

För att kunna använda databaser i databashanterare såsom *MySQL* eller *MSSQL* behöver vi först importera dem. I bokens tillhörande *GitHub*-sida hittar vi databasfilerna och vi förklarar nu hur de importeras för respektive databashanterare.

I *MySQL* utgår vi från “Local instance”-fliken, klickar på “Server” och väljer “Data Import”. Vi väljer “Import from Self-Contained File” och anger sökvägen till filen. Sedan klickar vi på “Start Import”, när importen är slutförd klickar vi på uppdatera-knappen och vi kan se vår databas “Köksglädje” i programmet.

I *MSSQL* högerklickar vi på “Databases”, trycker på “Restore Database”, “Device” och därefter på de tre punkterna “...”, “Add” och lokaliserar den plats vi har sparat *.bak* filen på. Slutligen trycker vi på OK och slutför inläsningen av databasen.

När vi har läst in databasen “Köksglädje” i vår databashanterare kan vi göra några utskrifter från den för att inspektera datan.

Vi börjar med att ange att det är i databasen “Köksglädje” vi vill utföra kommande operationer. Detta gör vi genom nyckelordet *USE* följt av “Köksglädje”.

```
USE Köksglädje;
```

I nedanstående kodexempel skriver vi ett *SELECT-statement* för att se vilka de olika produktkategorierna är.

```
SELECT CategoryID,  
       CategoryName  
FROM ProductCategories; ①
```

- ① Kolumnerna “CategoryID” och “CategoryName” kolumnerna läses ut.

CategoryID	CategoryName
1	Köksknivar
2	Köksmaskiner
3	Grytor/Kastruller
4	Bakredskap
5	Köksredskap
6	Förvaring
7	Stekpannor
8	Övrigt

I nedanstående kodexempel använder vi en *WHERE-clause* på “CategoryID”-kolumnen för att se namn och pris för alla produkter i kategorin “Köksknivar”.

```
SELECT ProductName,
       Price
FROM Products
WHERE CategoryID = '1';
```

①

- ① *WHERE* används för att filtrera datan genom villkoret “CategoryID = 1”.

ProductName	Price
Kockkniv	999
Brödkniv	749
Santokukniv	1199
Skalkniv	299
Grönsakskniv	499

3.5 Queries

I detta avsnitt kommer vi att demonstrera olika typer av *queries*.

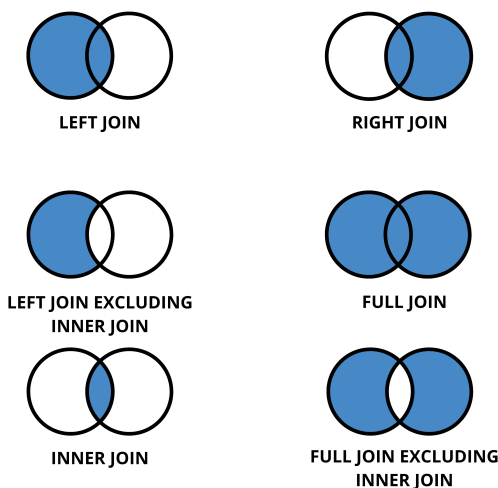
3.5.1 Joins

JOIN-clauses i SQL används för att kombinera kolumner från en eller flera tabeller till en ny tabell. Oftast används två eller flera tabeller men det är även möjligt att göra det som kallas en *SELF JOIN*. Detta innebär att en tabell utför en *join* på sig själv. Vi går inte in på detta utan nämner endast det för den intresserade läsaren.

I detta avsnitt kommer följande sex *joins* att demonstreras:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- LEFT JOIN EXCLUDING INNER JOIN
- FULL JOIN EXCLUDING INNER JOIN

Se Figur 3.2 för en grafisk representation över dessa *joins*.



Figur 3.2: De sex nämnda joins.

Vi kommer nu att skapa två exempeltabeller, “Students” och “Classes”, för att demonstrera *joins*. I koden följer vi namngivningskonventionen för MSSQL som är *PascalCase*. I MySQL hade namngivningskonventionen följt *snake_case* istället.

```
CREATE DATABASE JoinDemo; ①

USE JoinDemo;

CREATE TABLE Students(
    ClassID INT,
    LastName VARCHAR(30)
); ②

CREATE TABLE Classes(
    ClassID INT,
    ClassName VARCHAR(30)
); ③

INSERT INTO Students
    VALUES(1, 'Berg'),
           (1, 'Johansson'),
           (2, 'Nilsson'),
           (3, 'Sten'),
           (NULL, 'Doe'); ④

INSERT INTO Classes
    VALUES(1, 'Mathematics'),
           (2, 'Chemistry'),
           (3, 'History'),
           (4, 'Biology'); ⑤
```

- ① Databasen “JoinDemo” skapas.
- ② Tabellen “Students” skapas i “JoinDemo”-databasen.
- ③ Tabellen “Classes” skapas i “JoinDemo”-databasen.
- ④ Rader för “Students”-tabellen läggs till.
- ⑤ Rader för “Classes”-tabellen läggs till.

Nu när tabellerna är skapade kan vi läsa ut dem för att inspektera resultatet.

```
SELECT * FROM Students;
```

ClassID	LastName
1	Berg
1	Johansson
2	Nilsson
3	Sten
NA	Doe

```
SELECT * FROM Classes;
```

ClassID	ClassName
1	Mathematics
2	Chemistry
3	History
4	Biology

Vi demonstrerar nu de olika *joins* med våra två skapade tabeller.

INNER JOIN demonstreras nedan.

```
SELECT Students.ClassID,           ①  
       Students.LastName,         ②  
       Classes.ClassName          ③  
FROM Students INNER JOIN Classes  ④  
     ON Students.ClassID = Classes.ClassID; ⑤
```

- ① Från tabellen “Students” läses kolumnen “ClassID” ut.
- ② Från tabellen “Students” läses kolumnen “LastName” ut.

- ③ Från tabellen “Classes” läses kolumnen “ClassName” ut.
- ④ **FROM** specificerar att det är från tabellen “Students” vi läser ut kolumner. **INNER JOIN** specificerar att vi vill utföra en *inner join* med tabellen “Classes”.
- ⑤ Nyckelordet **ON** används för att specificera att kolumnen “ClassID” ska användas för att matcha rader i de två tabellerna.

ClassID	LastName	ClassName
1	Berg	Mathematics
1	Johansson	Mathematics
2	Nilsson	Chemistry
3	Sten	History

! Om vi endast skriver **JOIN** kommer en **INNER JOIN** att utföras. Vi rekommenderar att det explicit skrivs **INNER JOIN** för ökad läsbarhet.

LEFT JOIN demonstreras nedan.

```
SELECT Students.ClassID,
       Students.LastName,
       Classes.ClassName
FROM Students LEFT JOIN Classes
       ON Students.ClassID = Classes.ClassID; ①
```

- ① **LEFT JOIN** utförs.

ClassID	LastName	ClassName
1	Berg	Mathematics
1	Johansson	Mathematics
2	Nilsson	Chemistry
3	Sten	History
NA	Doe	NA

i I SQL blir den tabell som anges efter **FROM** den vänstra tabellen. Den tabellen som anges efter **JOIN** blir den högra.

RIGHT JOIN demonstreras nedan.

```
SELECT Students.ClassID,  
       Students.LastName,  
       Classes.ClassName  
FROM Students RIGHT JOIN Classes  
      ON Students.ClassID = Classes.ClassID; ①
```

① **RIGHT JOIN** utförs.

ClassID	LastName	ClassName
1	Berg	Mathematics
1	Johansson	Mathematics
2	Nilsson	Chemistry
3	Sten	History
NA	NA	Biology

i I praktiken är **LEFT JOIN** mer vanligt förekommande än **RIGHT JOIN**. Vill vi utföra en **RIGHT JOIN** kan vi helt enkelt ändra ordningen på tabellerna och därefter göra en **LEFT JOIN**.

FULL JOIN är inte tillgängligt i MySQL. I MySQL behöver vi därför använda en kombination av **LEFT JOIN** och **RIGHT JOIN** samt en **UNION**. En **UNION** används för att kombinera resultaten från två eller flera **SELECT-statements**.

I MySQL ser koden ut enligt följande:

```
SELECT students.class_id,  
       students.last_name,
```

```

        classes.class_name
FROM students
LEFT JOIN classes                                ①
    ON students.class_id = classes.class_id
UNION                                           ②
SELECT students.class_id,
        students.last_name,
        classes.class_name
FROM students
RIGHT JOIN classes                               ③
    ON students.class_id = classes.class_id

```

- ① LEFT JOIN används för att läsa ut alla rader från den vänstra tabellen “students”.
- ② UNION används för att kombinera *SELECT-statements*.
- ③ RIGHT JOIN används för att läsa ut alla rader från den högra tabellen “classes”.

I MSSQL utförs FULL JOIN enligt följande:

```

SELECT Students.ClassID,
        Students.LastName,
        Classes.ClassName
FROM Students FULL JOIN Classes
    ON Students.ClassID = Classes.ClassID;    ①

```

- ① FULL JOIN utförs.

ClassID	LastName	ClassName
1	Berg	Mathematics
1	Johansson	Mathematics
2	Nilsson	Chemistry
3	Sten	History
NA	Doe	NA
NA	NA	Biology

LEFT JOIN EXCLUDING INNER JOIN demonstreras nedan.

```
SELECT Students.ClassID,  
       Students.LastName,  
       Classes.ClassName  
FROM Students LEFT JOIN Classes  
      ON Students.ClassID = Classes.ClassID  
WHERE Classes.ClassID IS NULL; ①
```

- ① Endast rader där det saknas ett värde i den högra tabellen läses ut.

ClassID	LastName	ClassName
NA	Doe	NA

i RIGHT JOIN EXCLUDING INNER JOIN kommer inte att demonstreras. Detta eftersom den kan utföras på mostvarande sätt som en LEFT JOIN EXCLUDING INNER JOIN men för den högra tabellen. Alternativt så “byter vi plats” på den vänstra och högra tabellen och utför en LEFT JOIN EXCLUDING INNER JOIN därefter. Den metoden är mer vanligt förekommande i praktiken.

FULL JOIN EXCLUDING INNER JOIN demonstreras nedan. Eftersom FULL JOIN inte är definierat i MySQL behöver vi återigen använda en kombination av LEFT JOIN och RIGHT JOIN samt UNION.

I MySQL ser koden ut enligt följande:

```
SELECT students.class_id,  
       students.last_name,  
       classes.class_name  
FROM students  
LEFT JOIN classes
```

```

        ON students.class_id = classes.class_id
WHERE classes.class_id IS NULL                                ①
UNION
SELECT students.class_id,
        students.last_name,
        classes.class_name
FROM students
RIGHT JOIN classes
        ON students.class_id = classes.class_id
WHERE students.class_id IS NULL;                               ②

```

- ① Enbart rader där det saknas ett värde i den högra tabellen läses ut.
- ② Enbart rader där det saknas ett värde i den vänstra tabellen läses ut.

I MSSQL ser koden ut enligt följande:

```

SELECT Students.ClassID,
        Students.LastName,
        Classes.ClassName
FROM Students FULL JOIN Classes
        ON Students.ClassID = Classes.ClassID
WHERE Students.ClassID IS NULL
        OR Classes.ClassID IS NULL;

```

ClassID	LastName	ClassName
NA	Doe	NA
NA	NA	Biology

Aliasing kan användas för att ge tabeller i ett *statement* ett alias. Detta kan göra koden mer läsbar. Det är valfritt vilket alias som används men "A" och "B" är vanligt förekommande om två tabeller används. Används tre tabeller kallas den tredje för "C" och så vidare. Det förekommer också att förklarande eller deskriptiva alias används.

```

SELECT A.ClassID,
       A.LastName,
       B.ClassName
FROM Students AS A
INNER JOIN Classes AS B
      ON A.ClassID = B.ClassID;

```

①

②

① “Students”-tabellen tilldelas aliaset “A”.

② “Classes”-tabellen tilldelas aliaset “B”.

ClassID	LastName	ClassName
1	Berg	Mathematics
1	Johansson	Mathematics
2	Nilsson	Chemistry
3	Sten	History

Resultatet blir det samma som den tidigare demonstrerade INNER JOIN men *queryn* är skriven med *aliasing*.

i Vi behöver inte använda nyckelordet **AS** för att ge en tabell ett alias. Det är dock en god vana att använda det eftersom det gör koden mer läsbar. Om vi inte använder **AS** skriver vi endast tabellen och alias enligt följande: **FROM Students A** och **INNER JOIN Classes B**.

3.5.2 Select Top och Limit

I tabeller med ett stort antal rader kan **SELECT**-operationen vara tidskrävande. Då kan det vara användbart att skriva ut ett begränsat antal rader. I MSSQL används nyckelorden **SELECT TOP** följt av antal rader som ska läsas ut. I MySQL används nyckelordet **LIMIT**.

I nedanstående kodexempel och de resterande i detta kapitel använder vi databasen “Köksglädje”.

Vi skriver en *query* för att läsa ut de tio första raderna från databasens “Transactions”-tabell.

I MySQL ser koden ut enligt följande:

```
SELECT transaction_id,  
       transaction_date,  
       total_amount  
FROM transactions  
LIMIT 10; ①
```

- ① Nyckelordet LIMIT följt av 10 anges i slutet av *queryn* för att begränsa antalet rader som läses ut.

I MSSQL ser koden ut enligt följande:

```
SELECT TOP 10 TransactionID,  
       TransactionDate,  
       TotalAmount  
FROM Transactions; ①
```

- ① SELECT TOP 10 används för att endast läsa ut de tio första raderna.

TransactionID	TransactionDate	TotalAmount
1	2021-05-04	2544
2	2021-05-04	799
3	2021-05-04	3295
4	2021-05-05	1598
5	2021-05-05	3195
6	2021-05-06	3347
7	2021-05-06	4996
8	2021-05-07	1497
9	2021-05-08	6047
10	2021-05-09	1998

3.5.3 Order By

ORDER BY sorterar resultatet av en *query* i stigande eller fallande ordning.

Vi skriver en *query* för att från tabellerna “Transactions” och “Stores” i databasen “Köksglädje” läsa ut transaktionsdatum, den totala summan samt namnet på butiken där transaktionen skedde. I slutet av *queryn* används nyckelorden ORDER BY för att specificera att vi vill sortera listan på ett visst sätt. I detta fallet vill vi sortera med avseende på “TotalAmount”.

```
SELECT A.TransactionDate,  
       A.TotalAmount,  
       B.StoreName  
FROM Transactions AS A  
INNER JOIN Stores AS B  
      ON A.StoreID = B.StoreID  
ORDER BY A.TotalAmount;
```

①

① *Queryn* sorteras med avseende på “TotalAmount”-kolumnen.

TransactionDate	TotalAmount	StoreName
2023-06-30	84.15	Köksbutiken Stockholm
2023-07-08	84.15	Matlagningshörnan Stockholm
2023-08-20	84.15	Matlagningshörnan Stockholm
2023-09-01	99.00	Köksredskap Göteborg
2023-10-04	99.00	Grytor & Stekpannor Göteborg
2023-12-28	99.00	Stockholm Gourmet
2023-03-19	99.00	Köksbutiken Stockholm
2022-10-07	99.00	Stockholms Knivspecialist
2022-12-23	99.00	Stockholms Knivspecialist
2023-02-17	99.00	Matlagningshörnan Stockholm

Eftersom vi inte specificerade nyckelorden ASC eller DESC sorteras det enligt standard i stigande ordning. Önskar vi sortera i fallande ordning specificerar vi detta med nyckelordet DESC.

```

SELECT A.TransactionDate,
       A.TotalAmount,
       B.StoreName
FROM Transactions AS A
INNER JOIN Stores AS B
      ON A.StoreID = B.StoreID
ORDER BY A.TotalAmount DESC;

```

①

① DESC används för att sortera i fallande ordning.

TransactionDate	TotalAmount	StoreName
2022-05-14	8995.0	Malmö Gourmetkök
2022-02-20	8794.0	Köksredskap Göteborg
2022-04-15	8695.0	Stockholm Gourmet
2022-10-24	8695.0	Stockholms Knivspecialist
2021-07-07	7826.4	Köksbutiken Stockholm
2023-12-09	7695.2	Matlagningshörnan Stockholm
2022-11-30	7646.0	Grytor & Stekpannor Göteborg
2021-06-11	7645.5	Malmö Gourmetkök
2022-10-28	7545.0	Köksredskap Göteborg
2022-03-02	7496.0	Grytor & Stekpannor Göteborg

Vi kan kombinera `SELECT TOP` eller `LIMIT` med `ORDER BY` för att skapa en topplista. Vi skriver en *query* för att läsa ut de fem första “TransactionID” och “TotalAmount”. Vi sorterar tabellen med avseende på “TotalAmount”-kolumnen i fallande ordning för att läsa ut de transaktioner med högst “TotalAmount”.

I MySQL ser koden ut enligt följande:

```

SELECT transaction_id,
       total_amount
FROM transactions
ORDER BY total_amount DESC
LIMIT 5;

```

I MSSQL ser koden ut enligt följande:

```
SELECT TOP 5 TransactionID,  
              TotalAmount  
FROM Transactions  
ORDER BY TotalAmount DESC;
```

TransactionID	TotalAmount
408	8995.0
317	8794.0
375	8695.0
591	8695.0
65	7826.4

Resultatet blir en tabell med de fem transaktioner som har högst “TotalAmount”.

3.5.4 Group By och Having

GROUP BY är en funktion som möjliggör gruppering av data och aggregerade beräkningar för dessa grupper.

Vi skriver en *query* för att summera “TotalAmount” för respektive “StoreName”.

```
SELECT SUM(A.TotalAmount) AS TotalSales,      ①  
       B.StoreName                               ②  
FROM Transactions AS A  
INNER JOIN Stores AS B  
      ON A.StoreID = B.StoreID  
GROUP BY B.StoreName;                          ③
```

- ① SUM används för att summera “TotalAmount” och värdet sparas i en ny kolumn som vi namnger till “TotalSales”.
- ② “StoreName” läses ut.

- ③ Den aggregerade summan “TotalSales” beräknas för varje “Store-Name”.

TotalSales	StoreName
277181.9	Grytor & Stekpannor Göteborg
304157.3	Köksbutiken Stockholm
146306.8	Köksproffset Malmö
218076.5	Köksredskap Göteborg
176285.1	Malmö Gourmetkök
220362.1	Matlagningshörnan Göteborg
290975.6	Matlagningshörnan Stockholm
298103.9	Stockholm Gourmet
302088.5	Stockholms Knivspecialist

Vi ser att den totala försäljningen för varje grupp läses ut. I detta fall för varje butik. Exempelvis ser vi på den andra raden att “Köksbutiken Stockholm” har sålt för 304,157.3 kr.

För att filtrera resultat av en aggregerad kolumn används nyckelordet **HAVING**.

```
SELECT SUM(A.TotalAmount) AS TotalSales,
       B.StoreName
FROM Transactions AS A
INNER JOIN Stores AS B
      ON A.StoreID = B.StoreID
GROUP BY B.StoreName
HAVING SUM(A.TotalAmount) > 200000; ①
```

- ① **HAVING** används för att filtrera resultat så att endast rader där den totala försäljningen överstiger 200,000 kr inkluderas.

TotalSales	StoreName
277181.9	Grytor & Stekpannor Göteborg
304157.3	Köksbutiken Stockholm

TotalSales	StoreName
218076.5	Köksredskap Göteborg
220362.1	Matlagningshörnan Göteborg
290975.6	Matlagningshörnan Stockholm
298103.9	Stockholm Gourmet
302088.5	Stockholms Knivspecialist

3.5.5 Case When

CASE WHEN används för att gå igenom olika villkor och returnera ett värde när det första villkoret uppfylls.

I kodexemplet nedan använder vi **CASE WHEN** för att skapa en kolumn som anger ifall en butik har sålt för över 200,000 kr. Om den har gjort det sätts värdet 1, annars 0.

```
SELECT SUM(A.TotalAmount) AS TotalSales,
       B.StoreName,
       CASE WHEN SUM(A.TotalAmount) > 200000 THEN
         ↪ 1 ELSE 0 END AS Over200 ①
FROM Transactions AS A
INNER JOIN Stores AS B
      ON A.StoreID = B.StoreID
GROUP BY B.StoreName;
```

- ① **CASE WHEN** används för att skapa kolumnen “Over200”. Har butiken sålt för över 200,000kr sätts värdet 1. Nyckelordet **ELSE** används för att specificera att om villkoret inte uppfylls sätts värdet 0.

TotalSales	StoreName	Over200
277181.9	Grytor & Stekpannor Göteborg	1
304157.3	Köksbutiken Stockholm	1
146306.8	Köksproffset Malmö	0
218076.5	Köksredskap Göteborg	1

TotalSales	StoreName	Over200
176285.1	Malmö Gourmetkök	0
220362.1	Matlagningshörnan Göteborg	1
290975.6	Matlagningshörnan Stockholm	1
298103.9	Stockholm Gourmet	1
302088.5	Stockholms Knivspecialist	1

3.6 Funktioner

SQL har flera inbyggda funktioner som kan anropas. Det finns olika kategorier av funktioner. Fyra av dessa är:

- Aggregerings-funktioner som används för att utföra en beräkning över en uppsättning av värden och returnerar ett värde.
- Sträng-funktioner som används för att utföra olika operationer på strängar och returnerar en sträng.
- Matematiska funktioner som används för att utföra olika matematiska beräkningar.
- Datum-funktioner som används för att utföra olika operationer relaterade till datum.

I Tabell 3.2 listas några exempel på funktioner från respektive kategori.

Tabell 3.2: Vanligt förekommande SQL-funktioner.

Kategori	Funktion	Beskrivning
Aggregering	COUNT()	Räknar antalet rader som matchar ett villkor.
Aggregering	SUM()	Returnerar summan av en numerisk kolumn.
Aggregering	MAX()	Returnerar det största värdet i en kolumn.

Kategori	Funktion	Beskrivning
Aggregering	AVG()	Returnerar medelvärde för en numerisk kolumn.
Strängar	UPPER()	Omvandlar en sträng till versaler (stora bokstäver).
Strängar	LOWER()	Omvandlar en sträng till gemener (små bokstäver).
Strängar	LEN()	Returnerar längden av en sträng.
Strängar	CONCAT()	Slår samman två strängar till en.
Matematik	ABS()	Returnerar absolutvärdet av ett tal.
Matematik	LOG()	Returnerar den naturliga logaritmen.
Matematik	ROUND()	Avrundar ett tal till valt antal decimaler.
Datum	DAY()	Returnerar dagen för ett datum.
Datum	MONTH()	Returnerar månaden för ett datum.
Datum	YEAR()	Returnerar året för ett datum.
Datum	DATEDIFF()	Returnerar skillnaden mellan två datum. I MySQL beräknas dagar, i MSSQL kan användaren själv specificera hur det ska beräknas

3.7 Vyer

En vy är en virtuell tabell vars innehåll definieras av en *query*, vilket gör data lättillgänglig i tabellformat. Därmed behöver *queries*, som kan vara komplexa, inte köras om varje gång vi vill ha tillgång till datan. Det tillåter oss också att kunna dela viss utvald information med personer som inte har direkt eller fullständig tillgång till databasen, exempelvis på grund av behörighets- eller säkerhetsskäl.

Vi kan skapa en vy för den totala försäljningen per butik. Vi gör detta genom ett *CREATE-statement* där vi specificerar att det är en vy som ska skapas.


```

CREATE VIEW StoreSales ①
AS
SELECT SUM(A.TotalAmount) AS TotalSales,
       B.StoreName,
       CASE WHEN SUM(A.TotalAmount) > 200000 THEN
         ↪ 1 ELSE 0 END AS Over200000
FROM Transactions AS A
INNER JOIN Stores AS B
      ON A.StoreID = B.StoreID
GROUP BY B.StoreName; ②

```

- ① `CREATE VIEW` används för att specificera att det är en vy som skapas. Vyn tilldelas namnet “TotalSales”.
- ② *Queryn* som vyn ska vara ett resultat av skrivs därefter.

Vi kan nu använda vår vy genom att exempelvis läsa ut datan från vyn enligt koden nedan.

```
SELECT * FROM StoreSales;
```

TotalSales	StoreName	Over200000
277181.9	Grytor & Stekpannor Göteborg	1
304157.3	Köksbutiken Stockholm	1
146306.8	Köksproffset Malmö	0
218076.5	Köksredskap Göteborg	1
176285.1	Malmö Gourmetkök	0
220362.1	Matlagningshörnan Göteborg	1
290975.6	Matlagningshörnan Stockholm	1
298103.9	Stockholm Gourmet	1
302088.5	Stockholms Knivspecialist	1

3.8 Lagrade procedurer

En lagrad procedur är sparad kod som kan återanvändas. Med hjälp av lagrade procedurer kan vi centralisera definitioner. Om den lagrade proceduren används av flera personer säkerställs konsistenta resultat och gemensamma definitioner. Lagrade procedurer går också snabbare att anropa än att skriva om koden som den utför varje gång. Lagrade procedurer kan ha parametrar så användare kan nyttja styrande logik beroende på vilket resultat som önskas. Såväl lagrade procedurer som vyer kan i många sammanhang användas för att hantera samma problem och det är då upp till oss som användare att välja det verktyg vi finner mest ändamålsenligt.

För att skapa en lagrad procedur skriver vi ett *CREATE-statement* där vi specificerar att en lagrad procedur skapas. Vi namnger därefter den lagrade proceduren och skriver koden som den lagrade proceduren ska utföra.

I MySQL ser koden ut enligt följande:

```
DELIMITER // ①

CREATE PROCEDURE get_latest_sales() ②
BEGIN ③
    SELECT transaction_id, ④
           transaction_date,
           total_amount
    FROM transactions
    ORDER BY transaction_date DESC; ⑤
END // ⑥

DELIMITER ; ⑦
```

- ① I MySQL behöver `DELIMITER`-tecknet ändras.
- ② Proceduren tilldelas namet `“get_latest_sales”`.
- ③ `BEGIN` används för att definiera var proceduren startar.
- ④ Kod för att läsa ut kolumnerna `“transaction_ID”`, `“transaction_date”` och `“total_amount”` skrivs.

- ⑤ Proceduren avslutas med “;” vilket är standardvärde för “Delimiter” för att specificera att *queryn* som proceduren ska utföra är avslutad.
- ⑥ *Queryn* avslutas med det *Delimiter*-tecken som specificerades i början av *queryn*.
- ⑦ DELIMITER ändras tillbaka till dess standardtecken.

I MSSQL ser koden ut enligt följande:

```
CREATE PROCEDURE GetLatestSales
AS
BEGIN
    SELECT TransactionID,
           TransactionDate,
           TotalAmount
    FROM Transactions
    ORDER BY TransactionDate DESC
END;
```

①

- ① DELIMITER-tecknet behöver inte ändras och övrig kod är identisk med MySQL, bortsett från konventioner för namngivning.

För att sedan använda den lagrade procedur vi skapade, anropar vi den i en *query*.

I MySQL ser koden ut enligt följande:

```
CALL get_latest_sales();
```

①

- ① CALL används för att anropa proceduren.

I MSSQL ser koden ut enligt följande:

```
EXEC GetLatestSales;
```

①

- ① EXEC används för att anropa proceduren.

TransactionID	TransactionDate	TotalAmount
998	2023-12-30	399.0
999	2023-12-30	798.0
1000	2023-12-30	899.0
997	2023-12-29	1048.0
995	2023-12-28	2495.0
996	2023-12-28	99.0
994	2023-12-27	699.0
993	2023-12-26	2446.0
991	2023-12-23	2395.0
992	2023-12-23	2466.2

För att implementera styrande logik används parametrar. Detta görs genom att skapa en lagrad procedur och ange att vi vill inkludera en eller flera parametrar samt vilken den förväntade datatypen för parametrarna är. Vi kan exempelvis skapa en lagrad procedur där alla transaktioner från den stad vi anger läses ut.

I MySQL ser koden ut enligt följande:

```

DELIMITER //
CREATE PROCEDURE sales_per_city(IN desired_location
    ↪ VARCHAR(30)) ①
BEGIN
    SELECT A.transaction_date,
           A.total_amount,
           B.store_name,
           B.location
    FROM
        transactions AS A
    INNER JOIN
        stores AS B
        ON A.store_id = B.store_id
    WHERE
        B.location = desired_location; ②

```

```
END //

DELIMITER ;
```

- ① IN används för att definiera den ingående parametern, “desired_location” som är av datatyp `VARCHAR` med maximalt 30 tecken.
- ② Det specificeras att endast rader där kolumnen “location” är identiska med den angivna parametern “desired_location” ska läsas ut.

I MSSQL ser koden ut enligt följande:

```
CREATE PROCEDURE SalesPerCity
    @Location nvarchar(30) ①
AS
BEGIN
    SELECT A.TransactionDate,
           A.TotalAmount,
           B.StoreName,
           B.Location
    FROM
        Transactions AS A
    INNER JOIN
        Stores AS B
        ON A.StoreID = B.StoreID
    WHERE B.Location = @Location
END;
```

- ① Parameternamnet “Location” och datatypen `nvarchar(30)` specificeras efter “@”.

För att exekvera den lagrade proceduren anropar vi den och anger ett argument till parametern.

I MySQL ser koden ut enligt följande:

```
CALL sales_per_city('Stockholm');
```

I MSSQL ser koden ut enligt följande:

```
EXEC SalesPerCity @Location = 'Stockholm';
```

TransactionDate	TotalAmount	StoreName	Location
2021-05-04	799	Stockholms Knivspecialist	Stockholm
2021-05-04	3295	Köksbutiken Stockholm	Stockholm
2021-05-06	3347	Matlagningshörnan Stockholm	Stockholm
2021-05-06	4996	Stockholms Knivspecialist	Stockholm
2021-05-08	6047	Stockholms Knivspecialist	Stockholm
2021-05-09	1998	Stockholm Gourmet	Stockholm
2021-05-10	948	Stockholm Gourmet	Stockholm
2021-05-11	3746	Stockholm Gourmet	Stockholm
2021-05-12	2996	Stockholms Knivspecialist	Stockholm
2021-05-19	447	Köksbutiken Stockholm	Stockholm

Som förväntat läser den lagrade proceduren ut alla transaktioner som skett i Stockholm.

3.9 Indexering

I Avsnitt 2.6.1 lärde vi oss om index. Nu ska vi demonstrera hur de kan skapas. För att skapa ett index i SQL skriver vi ett *CREATE-statement* och specificerar att det är ett index vi vill skapa. Vi anger för vilken tabell och vilken kolumn vi skapar indexet.

```
CREATE INDEX idx_FirstName ①  
ON Customers (FirstName); ②
```

- ① Ett index med namnet “idx_FirstName” skapas med ett *CREATE INDEX-statement*.

- ② Indexet skapas för tabellen “Customers” på kolumnen “FirstName”.

i Notera att `CREATE INDEX` skapar ett icke-klustrat index. Processen för att skapa ett klustrat index är densamma, men med nyckelorden `CREATE CLUSTERED INDEX`. I MySQL är en tabells primärnyckel det klustrade indexet per automatik. Skulle ingen primärnyckel finnas används den första unika kolumnen.

Det går att skapa ett index för flera kolumner. Vi anger ytterligare en kolumn i vårt *CREATE-statement*.

```
CREATE INDEX idx_FullName
ON Customers (FirstName, LastName);
```

Indexet är nu skapat och kommer automatiskt att användas av SQL om en *query* involverar kolumnerna. För att ta bort ett index skriver vi ett *DROP-statement* och specificerar att det är ett index vi vill radera.

```
DROP INDEX idx_FullName ①
ON Customers;           ②
```

- ① “idx_FullName” är indexet som ska raderas.
② Indexet finns i tabellen “Customers”.

3.10 Exempel på några användbara queries

I detta avsnitt demonstreras exempel på några användbara *queries*. Vi kommer använda databasen “Köksglädje” för samtliga exempel.

Vill vi läsa ut hur många produkter butikskedjan tillhandahåller använder vi funktionen `COUNT()` i nedanstående kodexempel. Vi får ett

resultat där vi ser antalet rader i tabellen.

```
SELECT COUNT(*) AS TotalProducts
FROM Products;
```

①

- ① COUNT(*) beräknar totalt antal rader i tabellen “Products” och returnerar resultatet i en kolumn som vi namnger till “TotalProducts”.

TotalProducts
40

Vi kan använda DISTINCT vilket är en *clause* för att enbart visa unika värden. Nedanstående kodexempel läser ut alla unika värden för “CategoryID” i “Products”-tabellen, genom att använda DISTINCT.

```
SELECT DISTINCT CategoryID
FROM Products;
```

①

- ① DISTINCT används i tabellen “Products” för att läsa ut unika värden i kolumnen “CategoryID”.

CategoryID
1
2
3
4
5
6
7
8

Vi kan framställa ett mer läsbart resultat, där varje produktkategori med antal tillhörande produkter läses ut. Med hjälp av nyckelorden

INNER JOIN, GROUP BY och ORDER BY gör vi en *query* i nedanstående kodexempel.

```
SELECT B.CategoryID,  
       B.CategoryName,  
       COUNT(A.ProductID) AS ProductCount ①  
FROM Products AS A  
INNER JOIN ProductCategories AS B  
       ON A.CategoryID = B.CategoryID ②  
GROUP BY B.CategoryID,  
         B.CategoryName ③  
ORDER BY ProductCount DESC; ④
```

- ① “CategoryID” och “CategoryName”-kolumnerna läses ut. COUNT() räknar antalet produkter för varje unik kombination av “CategoryID” och “CategoryName”.
- ② “Products”-tabellen tilldelas alias “A”, INNER JOIN utförs med “ProductCategories”-tabellen som tilldelas alias “B”. “CategoryID”-kolumnen användas för att matcha rader i både tabellerna.
- ③ GROUP BY grupperar resultatet med avseende på “CategoryID” och “CategoryName”.
- ④ ORDER BY och DESC sorterar resultatet i fallande ordning med avseende på kolumnen “ProductCount”.

CategoryID	CategoryName	ProductCount
2	Köksmaskiner	8
5	Köksredskap	6
6	Förvaring	5
7	Stekpannor	5
3	Grytor/Kastruller	5
4	Bakredskap	5
1	Köksknivar	5
8	Övrigt	1

Vill vi undersöka försäljningen under 2022 använder vi funktionen

SUM() tillsammans med en *WHERE-clause* där filtreringen görs genom att använda YEAR() funktionen. Se kodexemplet nedan.

```
SELECT SUM(TotalAmount) AS TotalSales2022
FROM Transactions
WHERE YEAR(TransactionDate) = 2022;
```

①
②

- ① SUM() summerar alla värden i kolumnen “TotalAmount” från tabellen “Transactions”, AS används för att ge resultatcolumnen aliaset “TotalSales2022”.
- ② WHERE filtrerar raderna i tabellen “Transactions” med villkoret YEAR(TransactionDate)= 2022.

TotalSales2022
936342.4

Vi kan bygga vidare på det föregående kodexemplet för att läsa ut försäljningen under år 2022 per butik. I nedanstående kodexempel inkluderas butiksnamnen från tabellen “Stores” med hjälp av INNER JOIN, värden aggregeras med hjälp av GROUP BY och resultatet sorteras genom att använda ORDER BY.

```
SELECT B.StoreName,
       SUM(A.TotalAmount) AS TotalSales2022
FROM Transactions AS A
INNER JOIN Stores AS B
      ON A.StoreID = B.StoreID
WHERE YEAR(A.TransactionDate) = 2022
GROUP BY B.StoreName
ORDER BY TotalSales2022 DESC;
```

①
②
③

- ① INNER JOIN-clause kopplar samman “Transactions” med “Stores”.
- ② GROUP BY-clause aggregerar summan per butik.
- ③ ORDER BY-clause sorterar resultatet i fallande ordning.

StoreName	TotalSales2022
Grytor & Stekpannor Göteborg	131118.00
Stockholm Gourmet	128506.85
Stockholms Knivspecialist	124957.85
Köksbutiken Stockholm	115075.15
Matlagningshörnan Stockholm	99771.50
Malmö Gourmetkök	94446.30
Köksredskap Göteborg	91242.70
Matlagningshörnan Göteborg	86174.55
Köksproffset Malmö	65049.55

I nedanstående kodexempel använder vi fyra *JOIN-clauses* för att kombinera flera tabeller. Vi läser ut transaktioner från och med 2023-01-01 och inkluderar kundnamn, butiksnamn, produktnamn, köpta kvantiteter och det totala priset för varje rad.

```

SELECT  B.FirstName AS FN,
        B.LastName AS LN,
        C.StoreName AS SN,
        E.ProductName AS PN,
        D.Quantity AS QTY,
        D.TotalPrice AS TP                                ①
FROM Transactions AS A
LEFT JOIN Customers AS B
    ON A.CustomerID = B.CustomerID
LEFT JOIN Stores AS C
    ON A.StoreID = C.StoreID
LEFT JOIN TransactionDetails AS D
    ON A.TransactionID = D.TransactionID
LEFT JOIN Products AS E
    ON D.ProductID = E.ProductID                            ②
WHERE
    A.TransactionDate >= '2023-01-01';                    ③

```

① Nyckelordet *AS* används för att ge kolumnerna kortare namn så

att tabellen får plats i boken.

- ② **LEFT JOIN-clauses** används för att totalt kombinera fem tabeller.
- ③ **WHERE-clause** filtrerar resultatet så att endast transaktioner från och med 2023-01-01 inkluderas.

FN	LN	SN	PN	QTY	TP
Karin	Nilsson	Malmö Gourmetkök	Matlåda	1	149.0
Karin	Nilsson	Malmö Gourmetkök	Brödlåda	1	599.0
Karin	Nilsson	Malmö Gourmetkök	Visp	1	149.0
Filip	Zetterlund	Stockholms Knivspecialist	Potatispress	1	399.0
Filip	Zetterlund	Stockholms Knivspecialist	Pizzaskärare	1	249.0
Filip	Zetterlund	Stockholms Knivspecialist	Brödrost	1	699.0
Tilda	Carlsson	Köksredskap Göteborg	Pizzaskärare	2	498.0
Tilda	Carlsson	Köksredskap Göteborg	Non-stick Stekpanna	1	559.2
Tilda	Carlsson	Köksredskap Göteborg	Köksassistent	1	4999.0
Tilda	Carlsson	Köksredskap Göteborg	Bakform	1	349.0

3.11 Övningsuppgifter

1. Förklara nedanstående begrepp:
 - a) *Query*
 - b) *Statement*
 - c) *Clause*
 - d) *Keywords*
2. Hur hade en tabell med namnet “customertable” namngivits i MSSQL respektive MySQL?
3. I denna uppgift utförs CRUD-flödet.
 - a) Skapa en databas och namnge den till “Houses”.
 - b) I databasen “Houses” skapa en tabell med namnet “HousePrice”. Den ska bestå av två kolumner, “HouseID” och “HousePrice”. Både kolumnerna ska vara av datatypen `INTEGER`.
 - c) Lägg till två rader i tabellen, en med värdena (1, 4000000) och den andra med (2, 5000000).
 - d) Läs ut samtliga rader från tabellen.
 - e) Uppdatera priset för raden med “ID” 1 till 3500000.
 - f) Radera databasen “Houses”.

Använd databasen “Köksglädje” i resterande uppgifter. Databasen finns på bokens *GitHub*-sida.

4. Läs ut de 10 första raderna från “Transactions”-tabellen.
5. Använd en `INNER JOIN` för att läsa ut de 10 första raderna från “Transactions”-tabellen och inkludera namnet på den butik som transaktionen skett i från “Stores”-tabellen.
6. Använd en `LEFT JOIN` för att läsa ut de 30 första raderna från “Transactions”-tabellen, inkludera också “ApprovedToContact”-kolumnen från “Customers”-tabellen.

7. Använd en **FULL JOIN** för att läsa ut alla rader från “Transactions”- och “Stores”-tabellerna.
8. Använd **ORDER BY** för att läsa ut de 10 dyraste produkterna från “Products” tabellen. Listan ska sorteras med avseende på “Price” i fallande ordning.
9. Skriv en *query* som läser ut “CustomerID” och “LastName”. Använd **GROUP BY** för att beräkna den totala summan varje kund har spenderat.
10. Använd en **CASE WHEN** för att skapa en kolumn som har värdet 1 om en kund har spenderat över 2000 kr och 0 om den har spenderat under 2000 kr.
11. Skapa en vy och namnge den till “CustomerSpending”. Vyn ska visa tabellen som *queryn* i föregående uppgift producerar.
12. Skapa en lagrad procedur som accepterar “CustomerID” som parameter. Den lagrade proceduren ska returnera “CustomerID”, “LastName”, och den totala summan kunden har spenderat.

