2023

Databasimplementation

DATABASKONSTRUKTION (IT354G)

WILLIAM LINDHOLM (A22WILLI)

Innehållsförteckning

1. Antaganden

Under uppgiftens gång har en del oklarheter trätt fram, under denna rubrik ligger dessa listade.

- I det mottagna IE-diagrammet stod det att en agent endast skulle ha ett attribut som lagrar dess ursprungliga namn, ”Unamn”, samtidigt står det i beskrivningen för agent att det ursprungliga namnet ska lagras med förnamn och efternamn separat. Därför skapas istället två unika attribut för agentens riktiga namn; förnamn och efternamn.
- Agenten ska enligt diagrammet identifieras av ett namn och en siffra. I rapporten står det istället att agenten ska identifieras av ett kodnamn. Därför implementerades ett unikt attribut; Kodnamn för varje agent som innehåller både dess bokstav och siffra, ex: ”b23”.
- Vid implementering av rapport, kommer varken arvet eller rader att realiseras. För att det därför fortfarande ska gå att lagra rapporter med information skapas istället ett attribut; innehåll, som är en lång textsträng.
- Eftersom vissa avgränsningar gjorts, och endast en begränsad del av den databasen kommer realiseras går det inte gå att göra följande.
    - Hämta observationer från handläggare, se vilken observation en incident hänger ihop med eller se vilka hjälpmedel en viss operation eller fältagent är tilldelade.
    - Lagra rapporter av typerna avslag och uppföljning.
    - Lagra rader för rapporter.
- På grund av den stora mängden härledda attribut, rättigheter och begränsningar som måste beaktas vid insättning av data i databasen, är det orealistiskt att genomföra alla implementationer i denna prototyp. De funktioner som har implementerats kommer att förklaras mer ingående senare i senare avsnitt.

2. Databasimplementation

Vid databasens implementering valdes en sektion som omfattar agenter och deras relationer till olika operationer och incidenter. För en tydligare överblick kan man se IE-diagrammet som representerar denna sektion i bilaga A. Utifrån detta diagram designades de nödvändiga tabellerna som finns presenterade i bilaga B.

2.1 Datatyper

När en databas implementeras är det viktigt att använda lämpliga och genomtänkta datatyper. Olämpliga datatyper kan kraftigt öka lagringsbehovet eller helt underminera databasens tänkta funktionalitet. I den databasen som ska implementeras under loppet av denna uppgift finns många attribut där valet av datatyp kräver särskilt övervägande.

En agents kodnamn kommer alltid vara två eller tre tecken långt, då det är tecken/text som ska lagras, är det endast datatyperna char och varchar som är aktuella. Skillnaden mellan dessa är att typen varchar dynamiskt kan ändra längden tecken som lagras, därmed är denna väl optimerad för attribut där längden kan variera. Då det i detta fall alltid kommer vara antingen 2 eller 3 tecken, blir en char(3) mest effektivt.

Varpå Terrängtabellen delats upp utefter denormaliseringstekniken ”codes”, krävs en datatyp som håller ett nummer motsvarande ett terrängnamn. Då det finns ett ytterst begränsat antal olika terrängtyper behövs endast en begränsad mängd heltal lagras, därför används typen TINYINT, som har ett intervall på -128 till 127. Då det i detta fall blir enklast att lagra terrängtyper med positiva heltal, används istället datatypen ”TINYINT UNSIGNED” som har ett intervall på 0 till 255.

När lön lagras; ett numeriskt värde som ofta inkluderar decimaler och kräver hög precision, används datatypen ”DECIMAL”. Detta val gjordes eftersom användningen av float eller double kan leda till avrundningsskillnader, vilket riskerar att ge en inkorrekt lön. Även om dessa fel kan vara mycket små, är det något som företag eller myndigheter inte kan kompromissa med. På grund av den högre precisionen kräver decimal-datatypen mer lagringsutrymme och ökar även beräkningskomplexiteten. (MySQL, n.d.-b).

2.2 Begränsningar

2.2.1 begränsning av tomma värden

För att kontrollera de relationer som befinner sig i IE-diagrammet krävs ibland användning av ”NOT NULL”, som begränsar ett attribut på så sätt att det inte kan innehålla ”NULL”-värden. Denna kolumn måste då fyllas med ett värde och kan inte lämnas tom. Därav valdes bland annat operationstabellens främmande identifierande nyckelattribut för tabellen incident för att implementera en ”NOT-NULL”-begränsning. Detta eftersom en operation enligt det tillhandahållna IE-diagrammet inte ska få existera utan att tillhöra en incident. Se kod nedan i figur 1 för implementation.

```
CREATE TABLE Operation (
    ...
    IncidentName      VARCHAR(64) NOT NULL,
    IncidentNumber    INT NOT NULL,
    ...
) ENGINE = INNODB;
```

Figur 1: Betoning av implementation av NOT NULL i tabellen Operation.

2.2.2 Anpassade begränsningar

Ofta finns det specifika krav för hur inmatade data ska se ut, enligt uppgiftstexten ska agentens kodnamn bestå av en bokstav följt av en eller två siffror, exempelvis: ’D56’ (Gustavsson, 2005). Vid sådana tillfällen kan det vara lämpligt att använda en ”check constraint” inmatad data kontrolleras av ett förbestämt villkor.

```
CHECK (CodeName RLIKE '^[a-zA-Z][0-9]{1,2}$'),
CHECK (LENGTH(FirstName) > 0 AND LENGTH(LastName) > 0),
CHECK (Salary >= 0),
```

Figur 2: Begränsningar vid insättning av data i tabellen Agent.

I exemplet som visas i figur 2 används flera "check constraints" för att garantera dataintegriteten. Först och främst ser villkoret ”CodeName RLIKE '^[a-zA-Z][0-9]{1,2}$’" till att agentens kodnamn följer det specificerade formatet. Här används ett reguljärt uttryck (”Regex”) för att matcha teckenmönstret. Detta säkerställer att kodnamnet inleds med en bokstav och sedan följs av en eller två siffror. Dessutom finns det en annan begränsning som säkerställer att både förnamn och efternamn har en längd större än 0, vilket innebär att dessa fält inte kan vara tomma (fyllas med textsträngar av längden 0). Detta är viktigt för att alltid ha identifierbar information om varje agent.

En ytterligare begränsning ser till att lönen (Salary) som anges för en agent alltid är ett icke-negativt värde. Detta är logiskt eftersom en lön inte kan vara negativ. Detta kommer dock aldrig vara ett problem då lönen definieras som ett osignerat decimalvärde, vilket innebär att lönen rent praktiskt inte kan vara negativ, detta är mer en förtydning för uppgiftens skull.

2.2.3 Unika värden

Ibland bör alla värden som matas in i ett visst attribut på en tabell vara helt unika, det ska alltså inte finnas, exempelvis, två agenter med kodnamnet ’D56’. För att styrka denna begränsning används syntaxen ”UNIQUE” inom SQL. Om man försöker lägga till eller ändra en post så att den bryter mot en UNIQUE-begränsning, kommer databassystemet att generera ett fel och förhindra operationen. Inom det aktuella databassystemet som designas har denna begränsning implementerats på flera ställen. De tabeller som har attribut med en ”UNIQUE”-constraint är följande. Agent(Kodnamn), Terräng(TerrängKod) och Incident(IncidentNr).

När man använder denna begränsning bör man hantera den med omsorg, eftersom den kan påverka de relationer som definierades under den ursprungliga databasdesignen. Om denna

begränsning exempelvis tillämpas på en tabell som representerar en många-till-många-relation, kan relationens natur förändras till en annan relationstyp. Ex: 1-N eller 1-1.

I databasen har begränsningen "UNIQUE" särskilt implementerats för kolumnen "IncidentNumber" i incidenttabellen. Denna begränsning säkerställer att varje incident har ett unikt identifieringsnummer, vilket eliminerar risken för rader med samma identifierande värden. Utan denna unika begränsning skulle det vara möjligt för flera incidenter att dela samma nummer, vilket skulle komplicera databasförfrågningar och riskera att felaktig data hämtas. Här togs även beslutet att inte använda en unik-begränsning på incidentens namn, trots att detta också är en primärnyckel, detta för att det räcker med ett unikt identifierande värde per tabell. Efter ett par år i drift skulle det även kunna bli komplicerat att hitta unika namn för alla incidenter.

Genom att tillämpa alla dessa typer av begränsningar som tagits upp under kapitel 2.2 går det säkerställa att alla data som matas in i databasen uppfyller de krav och standarder som fastställts i databasbeskrivningen. Detta minskar i sin tur risken för felaktig inmatning, samt en stor förbättring av databasens dataintegritet.

3. Denormalisering

För att minska mängden operationer som krävs för att hämta korrekt data, kan denormalisering användas för att på olika sätt sammansätta olika tabeller. Detta kräver oftast en ökad mängd lagring i och med att redundant data kan förekomma. I dagens läge är detta oftast inte ett problem då beräkningskraft är dyrare än lagring.

3.1 Merge

Den första typen av denormalisering som använts är ”merge”, denna teknik innebär att två tabelles sammanslås utan att de nödvändigtvis modifieras. Här valdes tabellerna region och incident, detta eftersom det från agentens sida, är onödigt att genomföra en join-operation på 3 olika tabeller för att hämta terrängen på en operation som agenten är med i. Med denormaliseringen minskas denna operationen till en ”join” på två tabeller.

Denormaliseringen genomfördes genom att kombinera alla attribut från både "region" och "incident". Den resulterande sammanslagna tabellen behöll namnet "Incident", medan den nu överflödiga tabellen "region" raderades. Efter detta insågs det att kopplingen mellan "operation" och "region" fortfarande hänvisade till den borttagna "region"-tabellen. Eftersom informationen som tidigare fanns i "region" nu finns i "incident", skapades ett nytt främmande nyckelattribut i "operation" som pekar mot "incident". "Incident"-tabellens primärnycklar fortsätter att vara IncidentNamn och IncidentNr. Här nedanför i figur 1 presenteras koden för den sammanslagna "incident"-tabellen.

```
CREATE TABLE Incident (
            RegionName              VARCHAR(32) NOT NULL,
            Terrain                 TINYINT UNSIGNED NOT NULL,
            IncidentName            VARCHAR(64) NOT NULL,
            IncidentNumber          INT UNIQUE NOT NULL,
            Location                VARCHAR(128),
            PRIMARY KEY             (IncidentName, IncidentNumber),
            FOREIGN KEY (Terrain) REFERENCES Terrain(TerrainCode)
) ENGINE = INNODB;
```

Figur 3: Implementation av incidenttabell.

3.2 Codes

Därefter påbörjades den andra typen av denormalisering; ”Codes”, där ofta förekommande textsträngar, exempelvis företagsnamn eller stadsnamn byts ut mot ett motsvarande nummer. Ett perfekt tillfälle att tillämpa denna teknik finns i denna databas i regionstabellen, där regioner har en terräng. Här består terrängskolumnen av ofta upprepade textsträngar, såsom: tundra, öken eller skog. För att åstadkomma detta mönster måste en separat tabell för alla terränger lagras, se figur 3.

```
CREATE TABLE Terrain (
            TerrainCode          TINYINT UNSIGNED UNIQUE NOT NULL,
            TerrainName           VARCHAR(64),
            PRIMARY KEY           (TerrainCode)
) ENGINE = INNODB;
```

Figur 4: Implementation av terrängtabell.

I figur 4 representeras terrängen som ett icke-signerat "tinyint"-värde, vilket innebär ett heltal i intervallet 0 till 255. Detta innebär att det kan finnas upp till 255 olika terrängtyper, vilket anses vara en praktisk begränsning. Tabellen identifieras då av denna siffra, som därefter möjliggör hämtning av siffrans motsvarande terrängnamn. I figur 3, kan man sedan se motsvarande lagring av terräng, där endast en tinyint lagras, istället för en; i detta fall varchar(64) som markant minskar denna tabells radstorlek.

Denna teknik medför dock större krav på beräkningskraft då en ”join” måste genomföras varje gång en incidents terrängnamn ska hämtas. Detta spelar dock ingen roll eftersom bedömningen har gjorts att terrängtypen väldigt sällan hämtas, och platsbesparingen är därför mer värdefull.

3.3 Vertical split

I databasen har en ”vertical split” implementerats för tabellen rapport, där rapporter kan arkiveras till tabellen ”ArchivedReports”, detta möjliggör separat hantering av dessa. Exempelvis hade ett index kunnat implementeras på ”Report”-tabellen för att möjliggöra snabbare sökning på aktiva rapporter. Dessutom kan de arkiverade rapporterna i framtiden lagras på långsammare/billigare diskar. Storleken på indexen för de aktiva rapporterna minskar också, vilket gör att denna tabell snabbare och mer effektivt kan modifieras.

Den vertikala delningen genomfördes på just rapporterna eftersom dessa innehåller stora datamängder, samt att agenterna kan skapa väldigt många rapporter, detta bidrar till att fördelen med att lagra endast de aktiva rapporterna väger upp den ökade komplexiteten med att behöva flytta rapporter till arkivet när de inte längre är aktiva. Rent praktiskt var det även enkelt att implementera en vertikal delning här eftersom det inte finns någon annan tabell som är länkad till rapporterna, därav kommer det inte bli några problem med ”främmande nyckelintegritet om en rapport flyttas från ”Report” till ”ArchivedReport”. Se figur 5 för implementation av tabellen ”ArchivedReport”.

```
CREATE TABLE ArchivedReport (
            DateCreated             DATETIME NOT NULL,
            Author                  CHAR(3) NOT NULL,
            Title                   VARCHAR(64) NOT NULL
            Content                 VARCHAR(1024),
            IncidentName            VARCHAR(64),
            IncidentNumber          INT,
            PRIMARY KEY             (DateCreated, Title),
            FOREIGN KEY             (Author) REFERENCES
Agent(CodeName),
            FOREIGN KEY             (IncidentName, IncidentNumber)
            REFERENCES Incident     (IncidentName, IncidentNumber)
) ENGINE = INNODB;
```

Figur 5: Implementation av tabellen "ArchivedReport".

3.4 Horizontal split

I den databas som designats i denna uppgift har varje fältagent en kompetens och en specialitet, dessa lagras som långa textsträngar för varje Agent, oberoende av deras typ. Detta innebär att även om en agent är en handläggare, som inte ska ha någon kompetens eller specialitet, kommer dessa kolumner att lagras. Även om detta inte var fallet, att det inte fanns ett arv av denna struktur i agent-tabellen, är det sällan en fältagents specialitet och kompetens behöver hämtas. Det innebär att det är onödigt att dess storlek ska ökas, samt att onödig data ska läsas varje gång agent-tabellen hämtas. Därför är detta ett perfekt tillfälle att implementera en horisontell delning. Där dessa stora attribut lagras i en separat tabell, som identifieras av dess koppling till en agent.

```
CREATE TABLE FieldAgentAttributes (
            AgentCodeName           CHAR(3) UNIQUE NOT NULL,
            Specialty               VARCHAR(256),
            Competence              VARCHAR(256),
            CHECK       (Specialty IS NULL OR LENGTH(Specialty) > 0),
            CHECK       (Competence IS NULL OR LENGTH(Competence) > 0),
            PRIMARY KEY (AgentCodeName),
            FOREIGN KEY (AgentCodeName) REFERENCES Agent(CodeName)
) ENGINE = INNODB;
```

Figur 6: Implementation av tabellen "FieldAgentAttributes".

Med denna implementation, som visas i figur 4, kommer endast de agenter som har en specialitet eller kompetens inlagd att lagras. Därav undviks alla potentiella null-värden som hade skapats i agent-tabellen, och radstorleken minskas även markant eftersom 2 varchar-variabler med längden 256 tecken flyttats från agent. Som följd av detta behöver dock en join-operation genomföras varje gång en agents specialitet eller kompetens ska hämtas, men i detta fall väger fördelarna mycket tyngre än nackdelarna.

4. Indexering

För att snabba upp förfrågningar mot en databas kan indexering användas. När ett index på en kolumn skapas, sorteras denna kolumn, vilket bidrar till snabbare hämtning, till bekostnad av långsammare insättning. Vid insättning, måste allt innehåll tas i åtanke för att kunna placera den nya raden på rätt ställe. Men vid hämtning kan en mycket snabbare sökalgoritm; ”binary search” användas, som har den mycket lägre tidskomplexiteten O(log N).

När data är indexerad och sorterad kan sökningar genomföras med algoritmer som binärsökning vilket delar upp datamängden i hälften vid varje steg tills den hittar det önskade värdet. Detta ger en tidskomplexitet på O(log N), vilket innebär att söktiden ökar logaritmiskt i förhållande till datamängden. I kontrast tar en sökning i icke-sorterad data i genomsnitt O(N) tid, eftersom varje post potentiellt måste kontrolleras linjärt växer (GeeksforGeeks, 2023-b). Därför blir prestandaökningarna exponentiella med sorterad data jämfört med icke-sorterad, särskilt när datamängden växer (GeeksforGeeks, 2023-a).

Det finns även flera olika typer av indexeringar, varje gång en primärnyckel skapas, skapas automatiskt ett index för detta attribut. Likaså vid skapande av främmande nycklar. Detta är för att öka prestandan vid ”join”-operationer.

På grund av de prestandaförsämringar som följer med indexering vid insättning av data, bör endast fält som sökningar ofta genomförs på indexeras. I den aktuella databasen har därför ett "Composite index" skapats för Agentens för- och efternamn. Med denna typ av index kan upp till 16 kolumner kombineras. (MySQL, n.d.-a). Dessa index optimerar även prestandan vid operationer som sortering av all data i en tabell. Om man exempelvis vill hämta alla agenter i alfabetisk ordning efter hela namnet, är det fördelaktigt att ha index på både förnamn och efternamn, eftersom dessa tillsammans bestämmer sorteringen. Nedan; i figur 7, finns implementationen av det diskuterade indexet.

Detta index kommer vara användbart i den slutgiltiga implementationen av detta system, då gruppledare ofta kommer söka efter agenter på deras riktiga namn, eftersom det för människor är lättare att komma ihåg namn, än slumpmässigt genererade bokstäver och siffror.

```
CREATE INDEX Name ON Agent(FirstName, LastName) USING BTREE;
```

Figur 7: Implementation av Index för Agent.

”BTREE” användes sedan då det är en effektiv algoritm för att lagra flera olika typer av data, text eller siffror. Trots att ”BTREE” är standardalgoritmen som används i MySQL (Zawodny & Balling, 2004) definieras denna explicit för att följa den praxis som förekommer i kursens föreläsningar.

Därefter skapades även ett index för de terrängtyper (Terrain(TerrainName)) som befinner sig i terrängtabellen för att effektivisera förfrågningar som hämtar terrängnamn, indexet passar perfekt i denna tabell eftersom det kommer hålla samma statiska data under hela sin ”livstid” då det finns ett begränsat antal terrängtyper. Eftersom tabellen huvudsakligen kommer att innehålla statisk data, givet det begränsade antalet terrängtyper, kommer de flesta operationer att vara hämtningar. Även om indexering kan göra insättningar något långsammare och öka lagringsbehovet, är dessa nackdelar marginella då insättningar kommer genomföras mycket sällan. Prestandaökningarna blir också små eftersom prestandaökningarna ökar exponentiellt jämfört med icke-sorterad data beroende på mängden data. Men likaså kommer det bli prestandaökningar. Se implementation nedan i figur 8.

```
CREATE INDEX TerrainTypes ON Terrain(TerrainName);
```

Figur 8: Implementation av index i Terrängtabell.

Avslutningsvis bör man ha i åtanke att även om indexering är möjlig på alla attribut i en tabell, bör man undvika att lägga till för många då det ökar indexstorleken och kan påverka prestandan. Valet av kolumner bör baseras på vanliga förfrågningar. Indexering förbättrar sökeffektiviteten men kräver mer lagringsutrymme, så det är viktigt att balansera prestanda med lagringsbehov.

5. Vyer

Under skapandet av databasen hittas ofta många samband, potentiella härleda attribut och generell data som enkelt kan tas fram med en operation. Då överflödiga tabeller som kan introducera onödigt mycket redundant data, helst bör undvikas kan man istället skapa vyer. Vid en vanligt selektion agerar dessa precis som vanliga tabeller, men i bakgrunden genomför de oftast någon typ av operation, exempelvis en ”join”. I uppgiften har ett antal vyer skapats, i denna rapport kommer dock endast två förklaras i detalj. En vy kommer alltså aldrig att spara data, så varje gång en vy tillkallas, kommer den operation vyn implementerar att köras.

För att skapa en vy används SQL-kommandot ”CREATE VIEW” följt av vyns namn, sedan avslutas kommandot med ”AS” följt av den operation som ska köras vid hämtning av vyn.

5.1 Simplification (Report)

"Simplification" syftar till att sammanföra data från två eller flera tabeller för att visa sammansatt information som inte kan hämtas från en enskild tabell. Dock undviks komplexa operationer, såsom att beräkna medelvärden. Vyn, benämnd "AllReports" (se figur 9), sammanför data från "Report" och "ArchivedReport" med hjälp av en "UNION ALL"-operation. Denna vy ger en helhetsbild av alla sparade rapporter och är särskilt värdefull för handläggare som behöver identifiera lämpliga gruppledare för olika uppgifter. Att kunna granska både aktuella och äldre rapporter kan vara avgörande när specifika situationer uppstår.

```
CREATE VIEW AllReports AS

SELECT *

FROM Report

UNION ALL

SELECT *

FROM ArchivedReport;
```

Figur 9: Simplification vy (AllReports).

I koden presenterad i figur 9 används operationen "UNION ALL". Denna operation kombinerar data från båda tabellerna, men på ett annorlunda sätt jämfört med en JOIN-operation. Medan en JOIN placerar data horisontellt bredvid varandra, arrangerar "UNION ALL" data vertikalt, så att de läggs till direkt efter varandra.

Den främsta anledningen till att denna tabell skapades är för att det på grund av den vertikala delningen som införts på tabellen ”Report” numer inte går att hämta alla rapporter på ett enkelt sätt.

5.2 Simplification (Fieldagent)

Eftersom det inte på ett enkelt sätt går att hämta alla fältagenter, samt dess härledda attribut, skapas även en vy för detta. (se figur 10)

```
CREATE VIEW FieldAgents AS

SELECT Agent.CodeName, Firstname, LastName, Salary, Specialty,
Competence, AvgSuccessRate

FROM Agent, FieldAgentAttributes, AgentRates

WHERE Agent.IsFieldAgent

AND Agent.CodeName = FieldAgentAttributes.AgentCodeName

AND Agent.CodeName = AgentRates.CodeName;
```

Figur 10: Simplification vy (FieldAgents).

Denna vy samlar all information kopplad till en fältagent baserat på de attribut som definieras för en fältagent i relationsdatamodellen i bilaga A. Dock exkluderas fältagentens uppdrag eftersom en fältagent kan ha flera uppdrag, detta implementeras istället som en egen vy kallade ”AgentOperations”, denna vy kommer dock inte förklaras i detalj i denna rapport.

Vyn utför en "join"-operation med en annan vy, nämligen "AgentRates", som beräknar medelvärdet för lyckade operationer för alla agenter. Vidare filtreras agenter som inte är fältagenter bort med hjälp av villkoret "WHERE Agent.IsFieldAgent". Istället för att hämta all data väljer vi specifikt ut de kolumner vi vill visa, för att undvika duplicering av kolumner med identiska värden. Detta beror på att tabellerna ”Agent”, ”FieldAgentAttributes” och ”AgentRates” alla har en kolumn för agentens kodnamn.

Denna tabell är främst tänkt för att förbättra gruppledarens möjlighet att enkelt inspektera de agenter som finns tillgängliga. Genom att visa all relevant data på ett och samma ställe kommer hämtningen av fältagenter dramatiskt underlättas i framtiden.

5.3 Specialization (AgentRates)

”Specialization” är en typ av vy som kan genomföra operationer för att exempelvis bestämma medelvärde, summa eller maxtal, och visa detta som en kolumn i en vy. Dessa vyer är ofta till för specifika scenarion. Det skulle även kunna användas för att ge en användare tillgång till delar av en större vy.

Då varje typ av agent har det härledda attributet ”Rate” som innehåller dess genomsnittliga resultat av slutförda operationer, är detta en vy som är optimal att implementera. Se kod i figur 11.

```
CREATE VIEW AgentRates AS
SELECT
    OpIn.CodeName,
    AVG(Op.SuccessRate) AS AvgSuccessRate
FROM OperatesIn OpIn, Operation Op
WHERE
    OpIn.OperationName = Op.OperationName
    AND OpIn.StartDate = Op.StartDate
GROUP BY OpIn.CodeName;
```

*Figur 11: Implementation av ”AgentRates” vy (*specialization).

Denna vy baseras på SQL-funktionen "AVG", som står för "average" på engelska, vilket betyder snitt eller medelvärde på svenska. Funktionen beräknar medelvärdet av kolumnen "SuccessRate" för varje agent. För att säkerställa att varje agent representeras på en egen rad används "GROUP BY"-operationen. Genom denna operation får varje agent en unik rad, och medelvärdet beräknas för alla operationer där agenten deltagit. Resterande kod, ”WHERE” används sedan för att genomföra den ”join”-operation som möjliggör datahämtning från både agent- och operationstabellen.

Denna vy är användbar både för handläggare och gruppledare, eftersom båda typer av agenter kommer använda denna vy för att avgöra vilka agenter som presterar bra. De kommer dock oftast använda denna tabell indirekt genom att hämta data från någon av de vyer som använder sig av ”AgentRates” för att visa agenternas snittresultat. Exempelvis ”FieldAgents”, ”GroupLeaders” eller ”Managers”.

6. Stored procedures

”Stored procedures” är fördefinierade funktioner som kan exekveras för att hämta, radera eller modifiera data. Till skillnad från vyer kan dessa procedurer ta emot inparametrar. Till exempel kan en procedur hämta alla operationer som börjar med bokstaven 'g' eller radera alla incidenter med färre än ett visst antal agenter. Ur ett säkerhetsperspektiv är de särskilt värdefulla eftersom de är immuna mot SQL-injektioner; då den operation som ska utföras redan är fastställd. I en hypotetisk inloggningsfunktion skulle det exempelvis vara möjligt att logga in med namnet ”xxx OR 1 = 1” utan risk. Dessutom reducerar procedurer den komplexitet som annars kan uppstå när man ställer frågor till servern, eftersom de kan vara fördefinierade.

För att skapa en procedur används kommandot ”CREATE PROCEDURE” följt av procedurens namn, samt dess in- och utparametrar. Sedan skrivs koden innanför ett ”BEGIN” och ”END” kommando. Då ”standardavslutaren” i SQL är semikolon (;) och en procedur kan använda flera sammanlänkade operationer, krävs det att en ”DELIMMITER” används, för att temporärt förändra vad som används för att avsluta proceduren, då semikolon används inuti proceduren. I detta fall ändrades avslutningskommandot till ”//”. För en konkret implementation se figur 12.

I följande kapitel kommer 4 fördefinierade procedurer förklaras mer ingående.

6.1 Procedur 1: Datahämtning

Den första proceduren som skapades var en enkel procedur som hämtar alla agenters för och efternamn i alfabetisk ordning, sorterad på efternamn följt av förnamn. Se figur 12.

```
CREATE PROCEDURE GetAgentNamesAlphabetically()
BEGIN
    SELECT FirstName, LastName, CodeName
    FROM Agent
    ORDER BY LastName, FirstName;
END//
```

Figur 12: Enkel procedur (hämta alla agenters för- och efternamn i alfabetisk ordning).

Den kod som visas i figur 12, implementerades för att användare som saknar rättigheter att se all data kring en agent, som kan vara känslig, ändå kan se deras namn. Fältagenter bland annat, ska inte kunna se vilka kompetenser, specialiteter, samt vilken lön andra agenter har, därför får de istället rättighet till denna procedur, som endast hämtar relevant data. Det går såklart även se deras kodnamn, så man vet vilken agent som har vilket namn.

6.2 Procedur 2: Datahämtning med parameter

Nästa procedur används för att hämta operationer som skett mellan två datum, vilket sker genom att proceduren accepterar två parametrar. Den första parametern tar in det tidigaste startdatumet en operation får ha, den andra tar in det senaste slutdatumet. Denna procedur kan användas av alla agenter, men kanske främst av gruppledare, ifall de vill hitta tidigare operationer. För att sedan slippa leta igenom alla rapporter, kan man filtrera efter datum. Se kod i figur 13.

```
CREATE PROCEDURE GetOperationsInRange(IN _StartDate DATE, IN
_EndDate DATE)

BEGIN

    SELECT *

    FROM Operation Op

    WHERE Op.StartDate > _StartDate

    AND Op.EndDate IS NOT NULL AND Op.EndDate < _EndDate;

END//
```

Figur 13: Implementation av procedur med parametrar.

I första raden i koden, där själva proceduren deklareras skapas in parametrar innanför ”parametersektionen”, genom att skriva ”IN _parameternamn”. För denna databas föregås alla parameternamn med ett understreck (_) för att enkelt kunna skilja på parametrar och attribut inom procedurerna.

6.3 Procedur 3: Procedur med villkor

Nästa procedur används främst för att kontrollera antalet operationer som en agent får vara med i den senare delen kring ”triggers” genom att returnera det antal operationer som en given agent för tillfället är med i. Detta sker genom att användaren eller funktionen som vill hämta antalet aktiva operationer en agent är delaktig i, specificerar agentens kodnamn som parameter. Sedan hämtas svaret genom en ”OUT” parameter. Därför måste en variabel deklareras innan proceduren exekveras, som sedan kan hålla svaret. Se implementation i figur 14.

Då en agent ska kunna, under sin totala arbetstid hos PUCKO, kunna vara med i så många operationer som krävs, skapades denna funktion för att returnera de operationer som en agent för närvarande är delaktig i. Alltså operationer med ett slutdatum någon gång i framtiden. Således sorterar denna procedur även ut gamla operationer genom att jämföra slutdatumet, med dagens datum.

Eftersom det med denna procedur är möjligt att mata in ett kodnamn för en agent som inte existerar, därav måste användaren meddelas om att något gått fel, istället för att ge ett svar som då skulle bli 0. Detta felmeddelandet implementeras genom att signalera en felkod, följt av ett felmeddelande. För ohanterade användardefinierade undantag används felkoden ”45000” (MySQL, n.d.-c).

Då felkoden bör ge användaren ett meddelande som visar precis vad som gått fel, är det en bra idé att beskriva vilken agent som inte kan hittas. Då det i MySQL inte går att konkatenera text direkt i felmeddelandet, skapas en variabel som håller meddelandet genom funktionen ”SET” följt av ett meddelande som beskriver både vad som gått fel, och vilken agent som inte kunde hittas. För att skapa detta meddelandes konkateneras texten genom funktionen ”CONCAT()”.

Förutom att upprätthålla de begränsningar en agent ska ha i relation till operationer, kan denna funktion användas av gruppledare, för att se vilka agenter som för närvarande inte är delaktiga i en operation, vilket kommer gå snabbare än att leta efter agenter som inte finns med i ”AgentOperations” vyn, där alla agenter samt dess operationer listas.

```
CREATE PROCEDURE GetNumberOfActiveOperations(IN _Agent CHAR(3),
OUT _NumOfActiveOperations SMALLINT UNSIGNED)

BEGIN

  IF (SELECT COUNT(*) FROM Agent WHERE CodeName = _Agent) = 0

  THEN SET @message = CONCAT('Agent: ', _Agent, ' does not
    exist.');

  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message;

  END IF;


  SELECT COUNT(OpIn.OperationName) AS SumOfOperations INTO
    _NumOfActiveOperations

  FROM OperatesIn OpIn

  JOIN Operation Op ON OpIn.OperationName = Op.OperationName AND
    OpIn.StartDate = Op.StartDate

  WHERE OpIn.CodeName = _Agent AND (Op.EndDate IS NULL OR
    Op.EndDate > CURDATE());

END//
```

Figur 14: Implementation av ”GetNumberOfActiveOperations” procedur.

6.4 Procedur 4: Förflyttning av data i vertikal split

I kapitel 3.3 genomfördes en vertikal delning på tabellen ”Report”, vilket i detta fall innebär att rapporter ska kunna arkiveras, potentiellt till långsammare och billigare lagringsutrymme, eller en tabell med färre indexerade kolumner. Detta kräver då att det finns en procedur som kan genomföra denna förflyttning av data. Se kod i figur 15.

```
CREATE PROCEDURE ArchiveReport(IN _DateCreated DATETIME, IN _Title
VARCHAR(64))

BEGIN

    INSERT INTO ArchivedReport (DateCreated, Author, Title,
      Content, IncidentName, IncidentNumber)

    SELECT R.DateCreated, R.Author, R.Title, R.Content,
      R.IncidentName, R.IncidentNumber

    FROM Report R

    WHERE R.DateCreated = _DateCreated AND R.Title = _Title;


    DELETE FROM Report

    WHERE Report.DateCreated = _DateCreated AND Report.Title =
      _Title;

END //
```

Figur 15: Implementation av procedur ”ArchiveReport”.

Då en rapport i databasen identifieras av både det datum den skapades, samt dess titel, krävs två inparametrar för att möjliggöra lokalisering av rätt record. Därefter måste den data som ska flyttas först skapas i arkivtabellen, att den data som ska flyttas först kopieras in i den nya tabellen innan en radering genomförs är ett medvetet val för att förhindra dataförlust om denna operation skulle fallera. Till sist, som tidigare nämnt, raderas den data som nu finns på två ställen; i både ”Report” och ”ArchivedReport”.

Det finns för närvarande inte något automatiskt system som kör arkiveringsfunktionen, detta skulle optimalt implementeras genom ett skript som exempelvis körs en gång om dagen, och förflyttar alla rapporter äldre än ett år utefter de krav/önskemål som beskrivits i uppgiftsbeskrivningen. På grund av detta är funktionen inte tänkt att direkt användas av någon person eller grupp. Indirekt vid full implementering kommer det dock att drastiskt rensa upp bland rapporterna och möjliggöra snabbare hämtning, och sökning av de aktuella rapporterna som en användare antagligen vill hitta.

7. Triggers

Triggers är automatiska funktioner som aktiveras vid insättning, uppdatering eller radering av data i en tabell. Med hjälp av dessa kan man till exempel genomföra mer avancerade valideringar vid datainsättningar, ta bort relaterad data eller uppdatera främmande nycklar när befintlig data ändras. I en trigger kan man sedan komma åt den data som modifierats, skapats eller raderats genom SQL-kommandona ”OLD” eller ”NEW” beroende på vilken typ av operation som triggern lyssnar på.

7.1 Loggning

Loggning är ett viktigt verktyg, framförallt när det handlar om känslig data, såsom löner, personlig information, men kanske framförallt hemliga agenter. Därför loggas alla förändringar som genomförs på agenttabellen i en separat agentloggningstabell, där operationen som genomförts, användaren som genomfört operationen, agenten i fråga, samt tiden det hände loggas. I figur 16 finns en trigger som reagerar på ”inserts” i agenttabellen.

```
CREATE TRIGGER AgentLogInsert AFTER INSERT ON Agent

FOR EACH ROW BEGIN

  INSERT INTO AgentLog (Operation, UserName, CodeName,
   OperationTime) VALUES

  ('INSERT', USER(), NEW.CodeName, NOW());

END//
```

Figur 16: Loggningstrigger (insert) Agent.

Denna loggning sker främst för att hålla koll på vem som ändrat och när ändringar sker i en tabell. Då exempelvis systemadministratörer i detta fall, har rättighet att ändra i alla tabeller i databasen skulle de kunna ändra lönen på en agent. Ifall detta inte var en ”godkänd” operation, måste det finnas någon som tar ansvar för det fel som begåtts, i vilket fall loggningstabellen kan vara ett viktigt verktyg.

Eftersom vi inte endast vill logga insättningar av data kan koden i figur 16 även modifieras till att logga ”UPDATE” och ”DELETE” genom att byta ut ”AFTER INSERT” i deklarationen till ”AFTER UPDATE” och ”AFTER DELETE”. Därefter bör även det värde som sätts in i ”operations”-kolumnen i loggtabellen uppdateras för att reflektera den operation som de faktiskt reagerar på. Båda dessa triggers finns implementerade i databasen, de kan hittas under den fulla databasimplementationen i bilaga C.

7.2 Trigger med många funktioner

I en relationsdatabas finns det ofta tabeller som bygger på att andra tabeller existerar, denna databas är inget undantag. Skulle en incident raderas, kommer flera andra tabeller innehålla rader med främmande nycklar som pekar på data som inte längre existerar. I detta fall kommer operationer, som identifieras med hjälp av incident, inte längre att stämma. Och de rapporter som skrivits kring en incident som tagits bort kommer inte heller att fungera korrekt. Främmande nyckel integriteten har då förlorats.

På grund av detta fenomen implementeras en trigger som automatiskt rensar rapporter och operationer som tillhör en borttagen incident. Se implementation i figur 17.

```
CREATE TRIGGER IncidentCleanup BEFORE DELETE ON Incident

FOR EACH ROW BEGIN

    DELETE FROM Report

    WHERE IncidentName = OLD.IncidentName AND IncidentNumber =
      OLD.IncidentNumber;


    DELETE FROM ArchivedReport

    WHERE IncidentName = OLD.IncidentName AND IncidentNumber =
      OLD.IncidentNumber;


    DELETE FROM OperatesIn

    WHERE IncidentName = OLD.IncidentName AND IncidentNumber =
      OLD.IncidentNumber;


    DELETE FROM Operation

    WHERE IncidentName = OLD.IncidentName AND IncidentNumber =
      OLD.IncidentNumber;

END//
```

Figur 17: Trigger som raderar data tillhörande borttagna incidenter.

Denna trigger är i grunden ganska enkel: den utför samma ”delete”-operation på samtliga tabeller där data relaterad till en incident finns. Eftersom en radering genomförs finns det inga nya värden, och "OLD" används för att referera till det värde som ska raderas. Ordningen i vilken operationerna genomförs är kritisk. Tabellen "OperatesIn" bör rensas före "Operation" eftersom den förstnämnda innehåller ett främmande nyckelattribut som pekar på vilken operation en agent deltar i. Om en del av processen oväntat skulle misslyckas är det mycket viktigt att operationerna sker i rätt ordning för att bibehålla integriteten av de främmande nycklarna. Därför sker all radering i dessa tabeller innan en incident tas bort. Annars riskerar man att det finns poster kvar som fortfarande hänvisar till en incident om en operation fallerar.

Framförallt kommer detta gynna handläggare, då de är de enda med rättighet att göra förändringar och kompetens att ändra i incidenttabellen. Men kanske framförallt är det användbart vid det tillfälle en incident behöver mörkas. Då kan snabbt alla spår av att denna incident någonsin ägt rum raderas, eftersom alla rapporter och operationer som varit bundna till denna incident automatiskt följer med.

7.2 Trigger för validering av data

Vid tilldelning av operationer till en agent finns vissa begränsningar kring antal. En agent får endast vara delaktig i max 3 operationer om alla dessa befinner sig i samma region. Eftersom denna mer komplicerade begränsning kräver flera olika tabeller och operationer för att validera, går det inte använda en simpel ”CHECK”. Istället används därför en trigger.

För att genomföra denna validering måste det antal operationer som agenten är med i beräknas innan insättningen vilket innebär att själva triggern bör reagera innan insättningen sker, därav kommer det bli en ”BEFORE INSERT”. Vidare så lagras inte en agents operationer i agenttabellen, triggern bör därför ske på tabellen ”OperatesIn”. Därefter måste även den region som agenten för tillfället är aktiv i hämtas.

Antalet operationer som en agent tillhör, samt den region som agenten opererar i, är i sig komplexa operationer, dessa är därför uppdelade i separata procedurer. Se figur 14 och 18. Dessa kommer dock inte förklaras mer omgående, då procedurer redan förklarats i kapitel 6.

```
CREATE PROCEDURE GetOperationRegion(IN _OpName VARCHAR(128), OUT
  _Region VARCHAR(32))

BEGIN

    SELECT Inc.RegionName INTO _Region

    FROM Incident Inc

    JOIN Operation Op ON Op.IncidentName = Inc.IncidentName AND
      Op.IncidentNumber = Inc.IncidentNumber

    WHERE Op.OperationName = _OpName;

END//
```

Figur 18: procedur som hämtar regionen för en viss operation.

De värden som hämtas av dessa operationer bör även lagras på ett sådan sätt att det enkelt kan användas för nästa del, där själva kravet påtvingas. Därav används SQL:s ”DECLARE” för att deklarera variabler som kan hålla denna information. Se figur 19:

```
DECLARE MaxAllowedOperations INT DEFAULT 3;

DECLARE ExistingRegion VARCHAR(32) DEFAULT NULL;

DECLARE NewRegion VARCHAR(32) DEFAULT NULL;

DECLARE NumOfOperations SMALLINT UNSIGNED DEFAULT NULL;
```

Figur 19: Deklarerade variabler för trigger som validerar tabellen ”OperatesIn”.

Med hjälp av tidigare trigger, för hämtning av region hämtades endast den nya region som användaren försöker lägga till en agent på. Detta värde måste jämföras med den region agenten redan befinner sig i. Vilket innebär att en ny SQL-operation skapas för detta. Till sist hanteras begränsningarna genom att signalera fel om dessa krav inte uppföljs i villkoren i figur 20. För att se hela implementationen av triggern se kod i bilaga C (triggern heter ”ValidateOperatesIn”).

```
IF ExistingRegion IS NOT NULL AND ExistingRegion != NewRegion THEN
  SET @message = CONCAT('Agent already has ongoing operation in
    region:', ExistingRegion);
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message;
END IF;


IF NumOfOperations >= MaxAllowedOperations THEN
  SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Agent can be part of no more than 3
    operations in the same region.';
END IF;
```

Figur 20: Felmeddelanden för trigger ”ValidateOperatesIn”

Från ett användarperspektiv underlättar denna trigger vid tilldelning av operationer till agenter, eftersom det är omöjligt att tilldela för många operationer till en fältagent. Det kan även tjäna fältagenter i form av potentiellt en lättare arbetsbörda då det inte är möjligt för gruppledare att tvinga sina fältagenter exempelvis pendla mellan operationer i flera olika regioner.

8. Rättigheter

Vid design av SQL-databaser går det inte bara att fokusera på datatyper, begränsningar och relationer. Rättighetshantering är minst lika kritisk. Dessa rättigheter bestämmer vilka användare eller grupper som får utföra specifika operationer på databasen, som att läsa, skriva, uppdatera eller ta bort data.

Innan rättigheter kan tilldelas inom SQL, måste man definiera relevanta användare och roller. Detta görs med kommandona ”CREATE USER” eller ”CREATE ROLE”, följt av det önskade namnet på användaren eller rollen.

För att sedan ge specifika rättigheter till en användare eller roll, används kommandot: ”GRANT privilege_type ON database_name.table_name TO 'username'@'hostname';”. Här representerar "privilege_type" den specifika rättigheten som ska ges, till exempel ”INSERT”.

8.1 Roller

Roller representerar en uppsättning rättigheter som kan tilldelas till användare. När en användare tilldelas en specifik roll, erhåller de automatiskt alla rättigheter som associeras med den rollen. Med tanke på att agenter interagerar med systemet på olika nivåer beroende på deras position, är det lämpligt att definiera roller baserade på de olika agenttyperna i systemet: Fältagent, Gruppledare och Handledare.

Först skapades rollen ”FieldAgent” genom kommandot: ”CREATE ROLE FieldAgent;”. Därefter implementerades rollen ”FieldAgent”, som bland annat ska ha rättighet att läsa operationer, incidenter, och rapporter. Dessutom ska fältagenter ha rättighet att skapa och modifiera befintliga rapporter. Dock inte radera dem.

Då en fältagent inte bör ha rättighet att läsa känslig information om sina kollegor har fältagenter inte rättighet att läsa varken vyn ”FieldAgents” eller tabellen ”Agent”. Det finns dock en procedur som hämtar endast data fältagenten har rättighet att läsa, nämligen alla agenters för och efternamn, med sammanhängande kodnamn, eftersom det kan vara värdefullt att veta sina kollegors riktiga namn. Se figur 21 för implementation, procedurens rättighet befinner sig på den sista raden.

```
GRANT SELECT ON PUCKO_DB.OperatesIn TO FieldAgent;

GRANT SELECT ON PUCKO_DB.Operation TO FieldAgent;

GRANT SELECT ON PUCKO_DB.Incident TO FieldAgent;

GRANT SELECT ON PUCKO_DB.AllReports TO FieldAgent;

GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.Report TO FieldAgent;

GRANT SELECT ON PUCKO_DB.AgentOperations TO FieldAgent;

GRANT SELECT ON PUCKO_DB.Terrain TO FieldAgent;

GRANT EXECUTE ON PUCKO_DB.GetAgentNamesAlphabetically TO
  FieldAgent;
```

Figur 21: Rättigheter fältagent.

Här kan det vara värt att notera att ”AllReports” är en vy som fältagenten har rättighet att läsa, eftersom det är en vy, innehåller den inte själv någon data, det är alltså därför inte möjligt att

direkt tilldela modifikationsrättigheter på denna. Detta är något som måste hållas i åtanke vid implementering av webapplikationen eftersom de från användarens håll ser väldigt lika ut.

Denna vy har tilldelats fältagenter eftersom det kan vara svårt att veta vilka rapporter som är aktiva och vilka som har blivit arkiverade, denna vy fyller därför den funktion att den låter användaren. I detta fall en fältagent, enkelt hitta alla rapporter som ligger lagrade, oavsett var de ligger.

Därefter skapades rollen ”GroupLeader” som ska ha rättighet att läsa alla tabeller, samt rättighet att skapa och modifiera operationer, rapporter samt kompetenser och specialiteter för fältagenter. Utöver detta ska gruppledaren även ha rättighet att ta bort fältagenter från operationer. Dessa rättigheter finns konkret implementerade i figur 22.

```
GRANT SELECT ON PUCKO_DB.* TO GroupLeader;

GRANT UPDATE, INSERT ON PUCKO_DB.Operation TO GroupLeader;

GRANT UPDATE, INSERT, DELETE ON PUCKO_DB.OperatesIn TO
  GroupLeader;

GRANT UPDATE, INSERT ON PUCKO_DB.FieldAgentAttributes TO
  GroupLeader;

GRANT UPDATE, INSERT ON PUCKO_DB.Report TO GroupLeader;
```

Figur 22: Rättigheter gruppledare.

Till sist skapades rollen handläggare, som jag valt att översätta till engelskans manager i implementationen. Denna roll ska ha en mer övergripande blick av PUCKOS funktion och har därför inte lika mycket rättigheter när det kommer till detaljer kring vad som händer ute i fält. Därför har handläggaren i denna implementation inte lika mycket rättigheter att läsa data som gruppledaren, dock har handläggaren fler rättigheter när det kommer till modifikation av data. Detta är den enda rollen som har behörighet att skapa, modifiera och radera incidenter. Dessutom har denna roll rättighet att modifiera, skapa och radera i tabellerna terräng och rapport. Se figur 23 för implementation.

```
GRANT SELECT ON PUCKO_DB.GroupLeaders TO Manager;

GRANT SELECT ON PUCKO_DB.Operation TO Manager;

GRANT SELECT, UPDATE, INSERT, DELETE ON PUCKO_DB.Incident TO
  Manager;

GRANT SELECT, UPDATE, INSERT, DELETE ON PUCKO_DB.Report TO
  Manager;

GRANT SELECT ON PUCKO_DB.AllReports TO Manager;

GRANT EXECUTE ON PUCKO_DB.ArchiveReport TO Manager;
```

Figur 23: Rättigheter handläggare.

Eftersom det inte finns ett automatiskt skript som kör den procedur som ska flytta rapporter till de arkiverade rapporterna ges denna rättighet till handläggaren, genom det kommando som befinner sig på sista raden i figur 23. Då det exempelvis inte går att radera, eller visa en

procedur genom de rättighetstyper som används hittills, använder SQL istället rättighetstypen ”EXECUTE” för procedurer.

8.2 Användare

8.2.1 Administratör

Alla storskaliga databaser kräver en administratör, med rättighet och kompetens att modifiera själva funktionen av databasen, i denna implementation innebär det även att administratören får rättighet att läsa och modifiera alla tabeller som möjligtvis kan vara hemliga. För att göra det så säkert som möjligt får endast databasadministratören logga in på plats, detta genomdrivs genom att definiera databasadministratören som en lokal användare genom kommandot: ”CREATE USER 'DatabaseAdmin'@'localhost' IDENTIFIED BY 'Skalbagge#23';” där ’localhost’ begränsar denna användare till lokal inloggning. Därefter tilldelas administratören alla rättigheter av kommandot: ”GRANT ALL PRIVILEGES ON PUCKO_DB.* TO 'DatabaseAdmin'@'localhost';”

8.2.2 Agenter

I denna databas valdes slumpmässiga namn ut för alla användare, de följer dock den praxis som uttryckts i uppgiftsbeskrivningen, där alla agenter har ett användarnamn som börjar med en bokstav följt av en eller två siffror (Gustavsson, 2005).

Som tidigare nämnt skapas en användare genom kommandot ”CREATE USER”, se praktisk implementation av detta i figur 24.

```
CREATE USER 'A1'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT FieldAgent TO 'A1'@'%';
SET DEFAULT ROLE FieldAgent TO 'A1'@'%';

CREATE USER 'D56'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT GroupLeader TO 'D56'@'%';
SET DEFAULT ROLE GroupLeader TO 'D56'@'%';

CREATE USER 'B23'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT Manager TO 'B23'@'%';
SET DEFAULT ROLE Manager TO 'B23'@'%';
```

Figur 24: Skapande av användare.

Här skapas 3 olika användare, där alla är del av olika roller. För att dessa roller sedan ska börja gälla utan att användaren själv behöver ”aktivera” rollen ändras respektive roll till användarens standardroll.

Rättighetshantering i SQL-databaser är en kritisk säkerhetsåtgärd. Genom att noggrant definiera och tilldela rättigheter säkerställer vi att varje användare endast har tillgång till den

information och de funktioner de behöver för att utföra sina uppgifter. Detta minimerar risken för oavsiktlig eller illvillig skada på databasen.

Referenser:

GeeksforGeeks. (2023-a). Binary search - data structure and algorithm tutorials. https://www.geeksforgeeks.org/binary-search/

GeeksforGeeks. (2023-b). Linear search algorithm - data structure and algorithms tutorials. https://www.geeksforgeeks.org/linear-search/

Gustavsson, H. (2005). Databassystem: Databaskonstruktion Ht 2005. Inlämningsuppgift. https://his.instructure.com/files/885449/download?download_frd=1&verifier=HwCRIQRFThPTy5ltdhXCUlaTSasvCFa2tFJTd979

MySQL. (n.d.-a). MySQL 8.0 Reference Manual :: 8.3.6 multiple-column indexes. https://dev.mysql.com/doc/refman/8.0/en/multiple-column-indexes.html

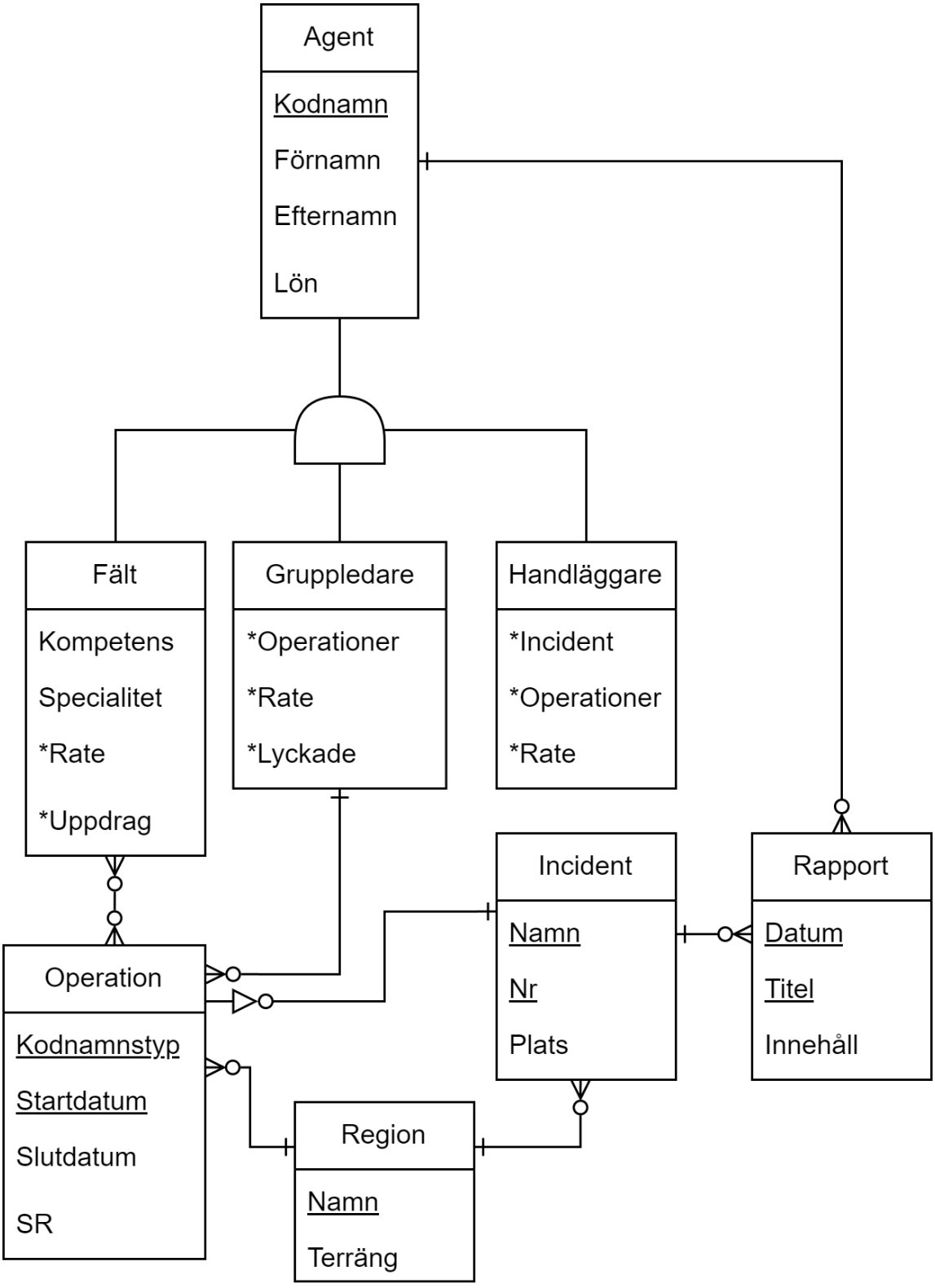
MySQL. (n.d.-b). MySQL 8.0 Reference Manual :: 11.1.3 fixed-point types (exact value) - decimal, numeric. MySQL. https://dev.mysql.com/doc/refman/8.0/en/fixed-point-types.html

MySQL. (n.d.-c). MySQL 8.0 Reference Manual :: 13.6.7.5 SIGNAL Statement. https://dev.mysql.com/doc/refman/8.0/en/signal.html

Zawodny, J. D., & Balling, D. J. (2004). Indexes. In High Performance MySQL. O'Reilly Online Learning. https://www.oreilly.com/library/view/high-performance-mysql/0596003064/ch04.html

Bilagor

Bilaga A:



Bilaga B:

$Agent(\underline{Kodnamn}, Förnamn, Efternamn, Lön, ÄrFältAgent, ÄrGruppLedare, ÄrHandläggare)$

$Operation(\underline{KodnamnsTyp}, Startdatum, Slutdatum, SR, \overline{Region}, \overline{\underline{IncidentNamn, IncidentNr}}, \overline{Gruppleda}$

$OpererarI(\overline{\underline{Kodnamn}}, \overline{\underline{Kodnamnstyp, Startdatum}}, \overline{\underline{IncidentNamn, IncidentNr}})$

$Region(\underline{Namn}, Terräng)$

$Incident(\underline{Namn, Nr}, Plats, \overline{Region})$

$Rapport(\underline{Datum, Titel}, Innehåll, \overline{Författare}, \overline{IncidentNamn, IncidentNr})$

Bilaga C:

```
DROP DATABASE PUCKO_DB;

CREATE DATABASE PUCKO_DB;

USE PUCKO_DB;


/* --- TABLE DEFINITIONS --- */


CREATE TABLE Agent (
		CodeName		CHAR(3) UNIQUE NOT NULL,
   FirstName		VARCHAR(32) NOT NULL,
   LastName		VARCHAR(32) NOT NULL,
   Salary			DECIMAL,
   IsGroupLeader		BOOL DEFAULT 0,
   IsManager		BOOL DEFAULT 0,
   IsFieldAgentBOOL DEFAULT 1,
   CHECK			(CodeName RLIKE '^[a-zA-Z][0-9]{1,2}$'), -- Only accept
ex: 'B23' or 'x5'.
   CHECK			(LENGTH(FirstName) > 0 AND LENGTH(LastName) > 0),
   CHECK			(Salary >= 0),
   PRIMARY KEY		(CodeName)
) ENGINE = INNODB;


CREATE INDEX Name ON Agent(FirstName, LastName) USING BTREE;


-- Denormalization (Horizontal split)
CREATE TABLE FieldAgentAttributes (
		AgentCodeName		CHAR(3) UNIQUE NOT NULL, -- Prevent multiple
FieldAgentAttributes per Agent using unique.
		Specialty		VARCHAR(256),
   Competence			VARCHAR(256),
   CHECK			(Specialty IS NULL OR LENGTH(Specialty) > 0),
```

```
    CHECK                   (Competence IS NULL OR LENGTH(Competence) > 0),
    PRIMARY KEY             (AgentCodeName),
    FOREIGN KEY             (AgentCodeName) REFERENCES Agent(CodeName)
) ENGINE = INNODB;

-- Denormalization (Codes)
CREATE TABLE Terrain (
            TerrainCode             TINYINT UNSIGNED UNIQUE NOT NULL,
    TerrainName                     VARCHAR(64),
    PRIMARY KEY             (TerrainCode)
) ENGINE = INNODB;

CREATE INDEX TerrainTypes ON Terrain(TerrainName);

-- Denormalization (Merge)
CREATE TABLE Incident (
            RegionName              VARCHAR(32) NOT NULL,
    Terrain                         TINYINT UNSIGNED NOT NULL,
            IncidentName VARCHAR(64) NOT NULL,
    IncidentNumber          INT UNIQUE NOT NULL,
    Location                VARCHAR(128),
    PRIMARY KEY             (IncidentName, IncidentNumber),
    FOREIGN KEY             (Terrain) REFERENCES Terrain(TerrainCode)
) ENGINE = INNODB;

-- Denormalization (Vertical split)
CREATE TABLE Report (
            DateCreated     DATETIME NOT NULL,
    Author                          CHAR(3) NOT NULL,
    Title                           VARCHAR(64) NOT NULL,
    Content                         VARCHAR(1024),
```

```
    IncidentName        VARCHAR(64),
    IncidentNumber      INT,
    PRIMARY KEY         (DateCreated, Title),
    FOREIGN KEY                 (Author) REFERENCES Agent(CodeName),
        FOREIGN KEY  (IncidentName, IncidentNumber) REFERENCES Incident(IncidentName,
IncidentNumber)
) ENGINE = INNODB;


-- Denormalization (Vertical split)
CREATE TABLE ArchivedReport (
        DateCreated  DATETIME NOT NULL,
    Author                      CHAR(3) NOT NULL,
    Title                       VARCHAR(64) NOT NULL DEFAULT 'nameless_report',
    Content                     VARCHAR(1024),
    IncidentName        VARCHAR(64),
    IncidentNumber      INT,
    PRIMARY KEY         (DateCreated, Title),
        FOREIGN KEY             (Author) REFERENCES Agent(CodeName),
        FOREIGN KEY  (IncidentName, IncidentNumber) REFERENCES Incident(IncidentName,
IncidentNumber)
) ENGINE = INNODB;


CREATE TABLE Operation (
        OperationName           VARCHAR(128) NOT NULL,
    StartDate           DATE NOT NULL,
    EndDate                     DATE,
    SuccessRate         BIT,
    GroupLeader                 CHAR(3),
    IncidentName        VARCHAR(64) NOT NULL,
    IncidentNumber      INT NOT NULL,
    CHECK                       (EndDate IS NULL OR EndDate >= StartDate),
        PRIMARY KEY  (OperationName, StartDate, IncidentName, IncidentNumber),
```

```
    FOREIGN KEY         (IncidentName, IncidentNumber) REFERENCES Incident(IncidentName,
IncidentNumber),
    FOREIGN KEY         (GroupLeader) REFERENCES Agent(CodeName)
) ENGINE = INNODB;


-- N - M relation (Agent, Operation)
CREATE TABLE OperatesIn (
            IncidentName VARCHAR(64) NOT NULL,
    IncidentNumber      INT NOT NULL,
    OperationName       VARCHAR(128) NOT NULL,
    StartDate           DATE NOT NULL,
    CodeName            CHAR(3) NOT NULL,
    PRIMARY KEY         (IncidentName, IncidentNumber, OperationName, StartDate,
CodeName),
    FOREIGN KEY         (IncidentName, IncidentNumber) REFERENCES Incident(IncidentName,
IncidentNumber),
    FOREIGN KEY         (OperationName, StartDate) REFERENCES Operation(OperationName,
StartDate),
    FOREIGN KEY         (CodeName) REFERENCES Agent(CodeName)
) ENGINE = INNODB;


/* --- LOG TABLES --- */


CREATE TABLE AgentLog (
            Id                                          BIGINT UNSIGNED NOT
NULL AUTO_INCREMENT,
    Operation           VARCHAR(10),
    UserName            VARCHAR(32),
    CodeName            CHAR(3),
    OperationTime       DATETIME,
    PRIMARY KEY         (Id)
) ENGINE = INNODB;
```

```
/* --- STORED PROCEDURES --- */


DELIMITER //


-- Get the number of active operations an agent has.

-- Conditional procedure with error handling

CREATE PROCEDURE GetNumberOfActiveOperations(IN _Agent CHAR(3), OUT
_NumOfActiveOperations SMALLINT UNSIGNED)

BEGIN

		IF (SELECT COUNT(*) FROM Agent WHERE CodeName = _Agent) = 0

			THEN SET @message = CONCAT('Agent: ', _Agent, ' does not exist.');

			SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message;

		END IF;


		SELECT COUNT(OpIn.OperationName) AS SumOfOperations INTO
_NumOfActiveOperations

		FROM OperatesIn OpIn

   JOIN Operation Op ON OpIn.OperationName = Op.OperationName AND OpIn.StartDate =
Op.StartDate

   WHERE OpIn.CodeName = _Agent AND (Op.EndDate IS NULL OR Op.EndDate > CURDATE());

END//


-- Simple procedure

CREATE PROCEDURE GetAgentNamesAlphabetically()

BEGIN

		SELECT FirstName, LastName, CodeName

   FROM Agent

   ORDER BY LastName, FirstName;

END//


CREATE PROCEDURE GetOperationRegion(IN _OpName VARCHAR(128), OUT _Region VARCHAR(32))

BEGIN
```

```
		SELECT Inc.RegionName INTO _Region
    FROM Incident Inc
    JOIN Operation Op ON Op.IncidentName = Inc.IncidentName AND Op.IncidentNumber = Inc.IncidentNumber
    WHERE Op.OperationName = _OpName;
END//

-- Move report from Report to ArchivedReport
-- Procedure that moves tuple in vertical split
CREATE PROCEDURE ArchiveReport(IN _DateCreated DATETIME, IN _Title VARCHAR(64))
BEGIN
    INSERT INTO ArchivedReport (DateCreated, Author, Title, Content, IncidentName, IncidentNumber)
    SELECT R.DateCreated, R.Author, R.Title, R.Content, R.IncidentName, R.IncidentNumber
    FROM Report R
    WHERE R.DateCreated = _DateCreated AND R.Title = _Title;

    DELETE FROM Report
    WHERE Report.DateCreated = _DateCreated AND Report.Title = _Title;
END //

-- get Operations between two dates
-- procedure that selects data
CREATE PROCEDURE GetOperationsInRange(IN _StartDate DATE, IN _EndDate DATE)
BEGIN
		SELECT *
    FROM Operation Op
    WHERE Op.StartDate > _StartDate
    AND Op.EndDate IS NOT NULL AND Op.EndDate < _EndDate;
END//

/* --- TRIGGERS --- */
```

```
-- Log Agent operations
CREATE TRIGGER AgentLogInsert AFTER INSERT ON Agent
FOR EACH ROW BEGIN
        INSERT INTO AgentLog (Operation, UserName, CodeName, OperationTime) VALUES
  ('INSERT', USER(), NEW.CodeName, NOW());
END//

CREATE TRIGGER AgentLogUpdate AFTER UPDATE ON Agent
FOR EACH ROW BEGIN
        INSERT INTO AgentLog (Operation, UserName, CodeName, OperationTime) VALUES
  ('UPDATE', USER(), NEW.CodeName, NOW());
END//

CREATE TRIGGER AgentLogDelete AFTER DELETE ON Agent
FOR EACH ROW BEGIN
        INSERT INTO AgentLog (Operation, UserName, CodeName, OperationTime) VALUES
  ('DELETE', USER(), OLD.CodeName, NOW());
END//

-- Validate that an agent is part of at least one group, fieldagents, managers or groupleaders.
CREATE TRIGGER ValidateAgent BEFORE INSERT ON Agent
FOR EACH ROW BEGIN
        IF NEW.IsGroupLeader = 0 AND NEW.IsManager = 0 AND NEW.IsFieldAgent = 0
                THEN
                        SIGNAL SQLSTATE '45000'
                        SET MESSAGE_TEXT = 'Agent missing type (eg:
FieldAgent).';
  END IF;
END//
```

```
-- Before deleting an incident, also remove all related operations and reports.
CREATE TRIGGER IncidentCleanup BEFORE DELETE ON Incident
FOR EACH ROW BEGIN
   DELETE FROM Report
   WHERE IncidentName = OLD.IncidentName AND IncidentNumber = OLD.IncidentNumber;

   DELETE FROM ArchivedReport
   WHERE IncidentName = OLD.IncidentName AND IncidentNumber = OLD.IncidentNumber;

   DELETE FROM OperatesIn
   WHERE IncidentName = OLD.IncidentName AND IncidentNumber = OLD.IncidentNumber;

   DELETE FROM Operation
   WHERE IncidentName = OLD.IncidentName AND IncidentNumber = OLD.IncidentNumber;
END//

-- Validate that an agent can only be part of Operations in the same region,
-- And at most 3 operations in the same region.
CREATE TRIGGER ValidateOperatesIn BEFORE INSERT ON OperatesIn
FOR EACH ROW BEGIN
                DECLARE MaxAllowedOperations INT DEFAULT 3;
                DECLARE ExistingRegion VARCHAR(32) DEFAULT NULL;
   DECLARE NewRegion VARCHAR(32) DEFAULT NULL;
   DECLARE NumOfOperations SMALLINT UNSIGNED DEFAULT NULL;

   -- Get the region of the new operation the agent is being added to
   CALL GetOperationRegion(NEW.OperationName, NewRegion);

   -- Get the current number of active operations an agent has
   CALL GetNumberOfActiveOperations(NEW.CodeName, NumOfOperations);
```

```
    -- Get the region of an existing operation the agent is part of
    SELECT Inc.RegionName INTO ExistingRegion
    FROM Incident Inc, Operation Op, OperatesIn OpIn
    WHERE Op.OperationName = OpIn.OperationName AND Op.StartDate = OpIn.StartDate AND
Op.IncidentName = Inc.IncidentName AND Op.IncidentNumber = Inc.IncidentNumber
    AND OpIn.CodeName = NEW.CodeName AND (Op.EndDate IS NULL OR Op.EndDate > CURDATE())
    LIMIT 1;

    -- If the agent is part of an operation in a different region, raise an error
    IF ExistingRegion IS NOT NULL AND ExistingRegion != NewRegion THEN
                                SET @message = CONCAT('Agent already has ongoing operation in
region:', ExistingRegion);
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = @message;
                END IF;

    -- If the agent is part already part of 3 operations, raise an error
                IF NumOfOperations >= MaxAllowedOperations THEN
                                SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Agent can be part of no more than 3 operations in the same region.';
    END IF;
END//

CREATE TRIGGER ValidateOperation BEFORE INSERT ON Operation
FOR EACH ROW
BEGIN
    IF NEW.EndDate > DATE_ADD(NEW.StartDate, INTERVAL 5 WEEK) THEN
      SET NEW.EndDate = DATE_ADD(NEW.StartDate, INTERVAL 5 WEEK);
    END IF;
END//



DELIMITER ;
```

```
/* --- VIEWS --- */

-- Specialization
-- Get the average successrate for all agents
CREATE VIEW AgentRates AS
SELECT
   OpIn.CodeName,
   AVG(Op.SuccessRate) AS AvgSuccessRate
FROM OperatesIn OpIn, Operation Op
WHERE
   OpIn.OperationName = Op.OperationName
   AND OpIn.StartDate = Op.StartDate
GROUP BY OpIn.CodeName;

-- Simplification (and specialization)
-- Get all field agents, including their competence, specialization and average successrate.
CREATE VIEW FieldAgents AS
SELECT Agent.CodeName, Firstname, LastName, Salary, Specialty, Competence, AvgSuccessRate
FROM Agent, FieldAgentAttributes, AgentRates
WHERE Agent.IsFieldAgent
AND Agent.CodeName = FieldAgentAttributes.AgentCodeName
AND Agent.CodeName = AgentRates.CodeName;

-- Simplification
-- Get all GroupLeaders
CREATE VIEW GroupLeaders AS
SELECT Agent.CodeName, FirstName, LastName, Salary, AvgSuccessRate
FROM Agent, AgentRates
WHERE IsGroupLeader
AND Agent.CodeName = AgentRates.CodeName;
```

```
-- Simplification
-- Get all managers
CREATE VIEW Managers AS
SELECT Agent.CodeName, FirstName, LastName, Salary, AvgSuccessRate
FROM Agent, AgentRates
WHERE IsManager
AND Agent.CodeName = AgentRates.CodeName;

-- Simplification
CREATE VIEW AllReports AS
SELECT *
FROM Report
UNION ALL
SELECT *
FROM ArchivedReport;

CREATE VIEW AgentOperations AS
SELECT A.CodeName, OpIn.OperationName
FROM Agent A, OperatesIn OpIn
WHERE A.CodeName = OpIn.CodeName;

/* --- GENERATE MOCK DATA --- */

INSERT INTO Terrain (TerrainName, TerrainCode) VALUES
                ('Mountainous', 1),
                ('Plains', 2),
                ('Hills', 3),
                ('Desert', 4),
                ('Forest', 5);
```

```
INSERT INTO Agent (CodeName, FirstName, LastName, Salary, IsGroupLeader, IsManager,
IsFieldAgent) VALUES
        ('A1', 'John', 'Doe', 50000, TRUE, FALSE, TRUE),
        ('B23', 'Jane', 'Smith', 52000, FALSE, TRUE, FALSE),
        ('C45', 'Alex', 'Johnson', 51000, FALSE, FALSE, TRUE),
        ('D56', 'Emily', 'Brown', 50500, TRUE, FALSE, FALSE),
        ('E78', 'Michael', 'Davis', 53000, FALSE, TRUE, TRUE);

INSERT INTO Incident (RegionName, Terrain, IncidentName, IncidentNumber, Location) VALUES
        ('North', 1, 'Heist', 101, 'North Town'),
        ('West', 2, 'Kidnapping', 102, 'South City'),
        ('North', 3, 'Sabotage', 103, 'East Village'),
        ('South', 4, 'Espionage', 104, 'West Point'),
        ('NorthEast', 3, 'Breach', 105, 'Central Hub'),
    ('SouthWest', 2, 'Ambush', 106, 'Mountain Pass'),
        ('MidWest', 4, 'Hijacking', 107, 'River Crossing'),
        ('Central', 1, 'Arson', 108, 'Forest Edge'),
        ('OuterWest', 3, 'Assault', 109, 'Desert Base'),
        ('DeepSouth', 2, 'Theft', 110, 'Island Shore');

INSERT INTO Operation (OperationName, StartDate, EndDate, SuccessRate, GroupLeader,
IncidentName, IncidentNumber) VALUES
        ('Operation Thunder', '2023-01-01', '2023-10-18', 1, 'A1', 'Sabotage', 103),
    ('Operation QuickSilver1', '2023-02-05', '2023-02-18', 0, 'D56', 'Heist', 101),
    ('Operation QuickSilver2', '2023-02-05', '2025-10-18', 0, 'D56', 'Heist', 101),
    ('Operation QuickSilver3', '2023-02-05', '2023-11-18', 0, 'D56', 'Heist', 101),
    ('Operation NightFall', '2023-03-10', '2023-03-30', 1, 'A1', 'Sabotage', 103),
    ('Operation DesertStorm', '2023-04-01', NULL, 1, 'E78', 'Espionage', 104),
    ('Operation ForestGuard', '2023-05-01', '2023-05-10', NULL, 'D56', 'Breach', 105);

INSERT INTO OperatesIn (IncidentName, IncidentNumber, OperationName, StartDate, CodeName)
VALUES
```

('Heist', 101, 'Operation Thunder', '2023-01-01', 'A1'),

('Heist', 101, 'Operation Thunder', '2023-01-01', 'B23'),

('Kidnapping', 102, 'Operation QuickSilver1', '2023-02-05', 'B23'),

('Kidnapping', 102, 'Operation QuickSilver2', '2023-02-05', 'B23'),

('Kidnapping', 102, 'Operation QuickSilver3', '2023-02-05', 'B23'),

('Kidnapping', 102, 'Operation QuickSilver1', '2023-02-05', 'C45'),

('Espionage', 104, 'Operation DesertStorm', '2023-04-01', 'D56'),

('Breach', 105, 'Operation ForestGuard', '2023-05-01', 'E78');

INSERT INTO FieldAgentAttributes (AgentCodeName, Specialty, Competence) VALUES

('A1', 'Explosives Expert', 'Advanced explosives handling and defusal'),

('C45', 'Surveillance Specialist', 'Expert in covert surveillance and intelligence gathering'),

('E78', 'Combat Specialist', 'Skilled hand-to-hand combat and small arms use');

INSERT INTO Report (DateCreated, Author, Title, Content, IncidentName, IncidentNumber) VALUES

('2023-04-15', 'A1', 'Update on Heist', 'Latest details on the heist operation. Successful with minor hitches.', 'Heist', 101),

('2023-06-10', 'B23', 'Kidnapping in West', 'The kidnapping incident at South City has been contained.', 'Kidnapping', 102),

('2023-06-20', 'D56', 'Sabotage Alert', 'Sabotage in East Village has caused significant infrastructure damage.', 'Sabotage', 103),

('2023-07-01', 'A1', 'Espionage in South', 'Espionage activities spotted in West Point. Suspects are being tracked.', 'Espionage', 104),

('2023-08-15', 'C45','Breach in NorthEast', 'A major breach in Central Hub. Cybersecurity teams are on it.', 'Breach', 105);

INSERT INTO ArchivedReport (DateCreated, Author, Title, Content, IncidentName, IncidentNumber) VALUES

('2020-03-05', 'D56', 'Old Heist Report', 'Details on an old heist operation.', 'Heist', 101),

('2019-12-10', 'A1', 'Old Kidnapping Details', 'Report on a kidnapping that happened two years back.', 'Kidnapping', 102),

('2021-01-20', 'B23', 'Sabotage in East - Historical', 'An old sabotage report detailing the damage.', 'Sabotage', 103),

```
    ('2021-02-15', 'A1', 'Espionage History', 'Previous espionage activities in West Point.', 'Espionage', 104),
    ('2021-04-05', 'E78', 'Breach Report - Historical', 'A significant breach happened two years back in Central Hub.', 'Breach', 105);


CALL GetAgentNamesAlphabetically();
/* --- USER PERMISSIONS --- */
/*
SELECT User, Host FROM mysql.user;

-- Drop Users
DROP USER 'DatabaseAdmin'@'localhost';
DROP USER 'A1'@'%';
DROP USER 'C45'@'%';
DROP USER 'D56'@'%';
DROP USER 'B23'@'%';
DROP USER 'E78'@'%';

-- Drop Roles
DROP ROLE FieldAgent;
DROP ROLE GroupLeader;
DROP ROLE Manager;

FLUSH PRIVILEGES;

-- FIELDAGENT ROLE
CREATE ROLE FieldAgent;

GRANT SELECT ON PUCKO_DB.OperatesIn TO FieldAgent;
GRANT SELECT ON PUCKO_DB.Operation TO FieldAgent;
GRANT SELECT ON PUCKO_DB.Incident TO FieldAgent;
GRANT SELECT ON PUCKO_DB.AllReports TO FieldAgent;
```

```
GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.Report TO FieldAgent;
GRANT SELECT ON PUCKO_DB.AgentOperations TO FieldAgent;
GRANT SELECT ON PUCKO_DB.Terrain TO FieldAgent;

-- GROUPLEADER ROLE
CREATE ROLE GroupLeader;

GRANT SELECT ON PUCKO_DB.* TO GroupLeader;
GRANT UPDATE, INSERT ON PUCKO_DB.Operation TO GroupLeader;
GRANT UPDATE, INSERT ON PUCKO_DB.OperatesIn TO GroupLeader;
GRANT UPDATE, INSERT ON PUCKO_DB.FieldAgentAttributes TO GroupLeader;
GRANT UPDATE, INSERT ON PUCKO_DB.Report TO GroupLeader;

-- MANAGER ROLE
CREATE ROLE Manager;

GRANT SELECT ON PUCKO_DB.GroupLeaders TO Manager;
GRANT SELECT ON PUCKO_DB.Operation TO Manager;
GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.Incident TO Manager;
GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.Report TO Manager;
GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.AllReports TO Manager;
GRANT SELECT ON PUCKO_DB.ArchivedReport TO Manager;
GRANT SELECT, UPDATE, INSERT ON PUCKO_DB.Terrain TO Manager;

-- DATABASE ADMIN
CREATE USER 'DatabaseAdmin'@'localhost' IDENTIFIED BY 'Skalbagge#23';
GRANT ALL PRIVILEGES ON PUCKO_DB.* TO 'DatabaseAdmin'@'localhost';

-- FIELDAGENTS
CREATE USER 'A1'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT FieldAgent TO 'A1'@'%';
```

```
SET DEFAULT ROLE FieldAgent TO 'A1'@'%';

CREATE USER 'C45'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT FieldAgent TO 'C45'@'%';
SET DEFAULT ROLE FieldAgent TO 'C45'@'%';

-- GROUPLEADERS
CREATE USER 'D56'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT GroupLeader TO 'D56'@'%';
SET DEFAULT ROLE GroupLeader TO 'D56'@'%';

-- MANAGERS
CREATE USER 'B23'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT Manager TO 'B23'@'%';
SET DEFAULT ROLE Manager TO 'B23'@'%';

CREATE USER 'E78'@'%' IDENTIFIED BY 'Skalbagge#23';
GRANT Manager TO 'E78'@'%';
SET DEFAULT ROLE Manager TO 'E78'@'%';

FLUSH PRIVILEGES;
```