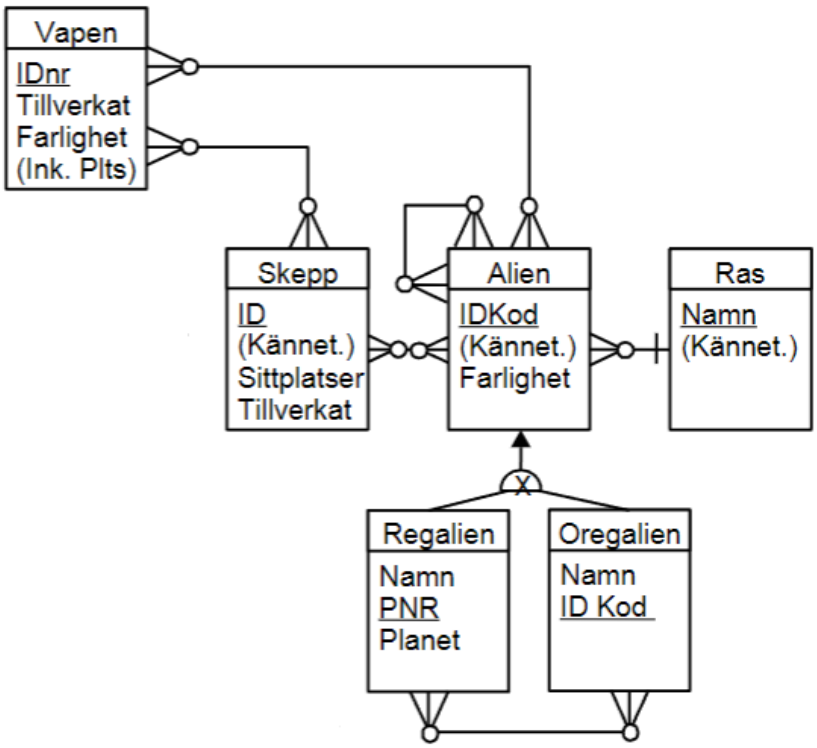
Databasimplementation

Intoduktion

För denna uppgift har den översta, högra delen av EER-Schemat använts, det vill säga; *Vapen, Skepp, Alien, Ras, Regalien* samt *Oregalien*. Se bild nedan. Tanken är att dokumentera vilka utomjordingar som har tillgångar som kan vara av risk eller annat meningsfullt värde att ha kunskap om, speciellt vid eventuell incident eller liknande, men ett stort fokus ligger även på att kunna gömma/hemlighetsstämpla information samt att kunna radera utomjordingar från databasen där uppgifter har utgått eller liknande.



Datatyper: CHAR

Under implementationen har varierande datatyper använts, bland annat används datatypen CHAR för både en aliens *IDkod*, personnummer (*pnr*) av registrerade aliens samt en oregistrerad aliens *införelsedatum* i databasen. Detta känns naturligt då de bland annat alltid ska ha en fast längd men även för att det blir enkelt att kunna bestämma formatet för kolumnen. Eftersom storleken på datan är fast och känt, kommer det alltid att tilldelas utrymme för den maximala längden och i det fall då data behöver hämtas kommer det ofta vara snabbare än att hämta data från en VARCHAR kolumn eftersom det inte finns något behov av att ta hänsyn till variabla längder. Detta är speciellt fördelaktigt när det kommer till frekvent hämtning av stora datamängder.

Datatyper: TINYINT + UNSIGNED

Datatypen TINYINT används i fältet *id* under *farlighet*stabellen eftersom detta fält *alltid* kommer att ha en ganska liten mängd rader då dessa ska beskriva vilka farlighetsnivåer som finns, då dessa idag enbart är 8 räcker detta. Även om man en dag skulle vilja utöka antalet nivåer finns det utrymme för detta. Att använda TINYINT för detta fält är fördelaktigt eftersom datatypen är kompakt och enbart kräver 1 byte av lagring. Detta medför även en bättre query prestanda eftersom det krävs mindre

minne för att hämta data från databasen. Med TINYINT används även UNSIGNED eftersom detta fält inte ska kunna vara negativ. Det är inte så intuitivt att ha ett id-fält som är negativt.

Datatyper: DATETIME

Även datatypen DATETIME har använts och det förekommer bland annat i fälten *logg_tid* och i *logg_datum* under tabellen *Ras_Logg* respektive *Alien_Hemlighetstämplande_Logg* som används för dokumentationsföring av raser och utomjordingar som för en eller annan anledning fått information om sig hemligstämplade. Meningen är att man ska kunna gå tillbaka och se när något ändrades, tabellen agerar alltså lite som en historik. Med detta blir det enkelt att logga saker, samt att det blir lätt att hitta, om man kommer ihåg ungefär vilken period som ändringen utfördes.

CONSTRAINT: CHECK

I denna uppgift har CHECK använts för att sätta regler samt för att säkerställa att databasen fungerar som förväntat. CHECK används på många ställen, bland annat på *IDkod*, *pnr*, och *införelsedatum* som tidigare nämnts, för att kontrollera att fälten har giltiga data i exempelvis ett korrekt format vid insättning av nya rader. Detta medför en mindre risk för fel vid datamanipulation samt kan göra det enklare att tillföra konsistens i databasen. Speciellt vid stora och komplexa databaser kan det vara av nytta att använda sig av CHECK för att säkra dataintegriteten och hanteringen. För just denna databas finns fördelen främst vid hanteringen av relationer mellan olika tabeller samt att data blir sammanhängande.

CONSTRAINT: UNIQUE

UNIQUE används för att säkerställa att ett värde i en kolumn alltid kommer att vara unikt och inte finnas i flera olika rader trots att det inte är en primärnyckel. UNIQUE har använts väldigt varsamt, enbart i *farlighets*tabellen samt i *ras*tabellen. Hela poängen med en farlighetsgrad är att de i sig ska vara ensamma och beskrivande för just den graden. Det finns ingen mening med att ha fler än en grad av ”harmlös” eftersom de representerar samma sak. Detsamma gäller namnet för en ras, det finns ingen mening att ha flera rader som beskriver samma ras, med samma namn. Det är en representation varav det enbart behövs 1 av.

CONSTRAINT: NOT NULL

NOT NULL finns även den på många ställen, men den som är mest kritiskt är nog den i tabellen *Vapen* där den finns för att säkerställa att ett vapen är kopplad mot ett skepps id eller ett aliens id, detta är på grund av kravet att ett vapen **måste** ha en ägare, men **inte** kan ägas av **både** en alien och ett skepp samtidigt. NOT NULL constrainten har använts i samband med en check för att kontrollera att det alltid finns ett fält som är NULL och ett som inte är NULL. Sammanfattningsvis kan inte både skepp id och alien id fälten vara NULL samtidigt, och de kan inte heller vara NOT NULL samtidigt.

Denormalisering

Idag är det större efterfrågan på snabb hämtning av data, snarare än att effektivisera utrymmet, därför har denormaliseringstekniker använts för att optimera databasen.

Merging

När databasen började skrivas skapades olika tabeller för Ras och för Alien, dessa såg ut som följande:

```
CREATE TABLE Ras(
    rasID      SMALLINT AUTO_INCREMENT,
    namn        VARCHAR(30) UNIQUE,
    PRIMARY KEY (rasID)
);

CREATE TABLE Alien(
    IDkod       CHAR(25),
    farlighet   TINYINT UNSIGNED DEFAULT 4,
    rasID       SMALLINT DEFAULT 1,
    PRIMARY KEY (IDkod),
    FOREIGN KEY (farlighet) REFERENCES Farlighet (id),
    FOREIGN KEY (rasID) REFERENCES Ras (rasID)
);
```

Första steget för att optimera databasen har gjorts genom att slå ihop tabellerna Ras och Alien till en enda. Detta kallas för 'Merging'. Detta känns naturligt då relationen mellan Ras och Alien är 1:N och en främmande nyckel elimineras, på så sätt blir det mindre komplext med färre främmande nycklar att hantera. Resultatet blev följande:

```
CREATE TABLE Alien(
    IDkod       CHAR(25),
    farlighet   TINYINT UNSIGNED DEFAULT 4,
    rasID       SMALLINT DEFAULT 1,
    ras_namn    VARCHAR(30),
    PRIMARY KEY (IDkod),
    FOREIGN KEY (farlighet) REFERENCES Farlighet (id)
);
```

Codes

Tabellen Farlighet hade från början enbart en kolumn med strängar för olika farlighetgrader. Genom att 'kodifiera' tabellen kan man i stället hämta en TINYINT som kan ge bättre responstid i jämförelse med att hämta en lång sträng.

```
CREATE TABLE Farlighet(
    id          TINYINT UNSIGNED AUTO_INCREMENT,
    grad        VARCHAR(16) UNIQUE,
    PRIMARY KEY (id)
);
```

Vertical split

Vertikal split gör man för att få en mindre tabell men högre prestanda, och för de få gånger som man behöver mer information är det värt att vänta lite längre för. I den här uppgiften har spliten gjorts på tabellen *Vapen.* Den splittade tabellen med färre kolumner heter *Vapen_Ägare* då tanken är att man kanske redan vet vilket vapen som man vill leta upp och enbart hitta dess ägare, då går det fortare med en tabell som innehar färre kolumner och därmed också mindre data att hämta. Resultat blir följande:

```
CREATE TABLE Vapen(
    vapen_IDnr  INT,
    tillverkat  DATE,
    farlighet   TINYINT UNSIGNED DEFAULT 4,
    alien_IDkod VARCHAR(25),
    skepp_id    INT,
    PRIMARY KEY (vapen_IDnr),
    FOREIGN KEY (farlighet) REFERENCES Farlighet (id),
    CONSTRAINT FOREIGN KEY (skepp_id) REFERENCES Skepp (id),
    CONSTRAINT FOREIGN KEY (alien_IDkod) REFERENCES Alien (IDkod),

    -- CHECK som kollar att så att ett fält är tomt.
    CONSTRAINT chk_vapen_null
    CHECK ( alien_IDkod IS NULL OR skepp_id IS NULL ),

    -- CHECK som kollar att ett fält INTE är tomt.
    CONSTRAINT chk_vapen_not_null
    CHECK ( alien_IDkod IS NOT NULL OR skepp_id IS NOT NULL )
);
```

```
CREATE TABLE Vapen_Ägare(
    vapen_IDnr  INT,
    alien_IDkod VARCHAR(25),
    skepp_id    INT,
    PRIMARY KEY (vapen_IDnr),
    CONSTRAINT FOREIGN KEY (skepp_id) REFERENCES Skepp (id),
    CONSTRAINT FOREIGN KEY (alien_IDkod) REFERENCES Alien (IDkod),

    -- CHECK som kollar att så att ett fält är tomt.
    CONSTRAINT chk_vapen_ägare_null
    CHECK ( alien_IDkod IS NULL OR skepp_id IS NULL ),

    -- CHECK som kollar att ett fält INTE är tomt.
    CONSTRAINT chk_vapen_ägare_not_null
    CHECK ( alien_IDkod IS NOT NULL OR skepp_id IS NOT NULL )
);
```

I *Vapen_Ägare* är kolumnerna tillverkat och farlighet tydligt uteblivet.

Horizontal split

Denna teknik har använts på en tabell som heter *Alien_Hemligstämplande_Logg*. Här görs det för att plocka ut kolumnen för kommentarer, då det kanske inte är alla hemligstämplande som är signifikanta så att det behövs ytterligare kommentar på detta, och när det behöver lagras en kommentar finns en egen tabell för detta med en främmande nyckel tillbaka till den originella tabellen för att knyta den mot ett specifikt hemligstämplande.

```
CREATE TABLE Hemligstämplat_Logg_Kommentar(
    loggID      SMALLINT AUTO_INCREMENT,
    kommentar   VARCHAR(255),
    PRIMARY KEY (loggID),
    FOREIGN KEY (loggID) REFERENCES Alien_Hemligstämplade_Logg(loggID)
);
```

Indexering

Index gör det snabbare att söka upp något, men det gör det även långsammare för att uppdatera en databas. Därför behöver index alltså skapas varsamt. För denna databas har ett index skapats för kolumnen *ras_namn* i tabellen *Alien* för anledningen att det har varit stor fokus i att skapa procedurer för att hemligstämpla raser hos aliens. Det vill säga, information om raser, exempelvis dess id samt dess namn hämtas ofta. RasID loggas redan som integers, vilket gör det lite snabbare redan, det som i stället hade gynnats av indexering är rasens namn som också hämtas varje gång en hemligstämpling sker. Eftersom rasens namn lagras i en VARCHAR av okänt längd, kan det verkligen tjäna på att indexeras.

```
CREATE TABLE Alien(
    IDkod       CHAR(25),
    farlighet   TINYINT UNSIGNED DEFAULT 4,
    rasID       SMALLINT DEFAULT 1,
    ras_namn    VARCHAR(30),
    PRIMARY KEY (IDkod),
    FOREIGN KEY (farlighet) REFERENCES Farlighet (id)
);

CREATE INDEX alien_rasnamn_index ON Alien (ras_namn ASC) USING BTREE;
```

Ett index har även skapats på kolumnen *logg_datum* i tabellen *Alien_Hemligstämplande_Logg* av den anledningen att denna kolumn varken är en primär- eller främmande nyckel. Genom att indexera på *logg_datum* kan man hämta data som loggades vid specifika datum. Detta leder till en mer effektiv ”historik” som kan observeras på ett mer strukturerat sätt.

```
CREATE TABLE Hemligstämplat_Logg(
    loggID      SMALLINT AUTO_INCREMENT,
    logg_datum  DATETIME DEFAULT NOW(),
    IDkod       CHAR(25),
    rasID       SMALLINT,
    ras_namn    VARCHAR(30),
    ras_kännetecken VARCHAR(255) NOT NULL,
    PRIMARY KEY (loggID)
);

CREATE INDEX hemlig_logg_index ON Alien_Hemligstämplade_Logg (logg_datum
ASC) USING BTREE;
```

Vyer

Det har även skapats ett antal vyer för att förenkla användandet av databasen. Alien_Personnummer_view kan användas för att få en simplifierad överblick över alla aliens, både på registrerade aliens och oregistrerade aliens. Det som gör denna vy speciell är att denna tabell innehåller de registrerade aliens personnummer och de oregistrerade aliens införelsedatum under en gemensam kolumn kallat ”Personnummer”. Vad som är bra med detta är att man får relevant information över alla aliens som finns i databasen men att man enkelt ser skillnaden mellan de registrerade och oregistrerade aliens på hur långt värdet är i ’Personnummer’ kolumnen. Är den ”kortare” är det en registrerad alien, medan om den är ”längre” så är det en oregistrerad alien.

```
CREATE VIEW Alien_Personnummer_view AS
SELECT IDkod, namn, pnr AS 'Personnummer', hemplanet
FROM Registrerad_Alien
UNION
SELECT IDkod, namn, införelsedatum AS 'Personnummer', NULL as 'hemplanet'
FROM Oregistrerad_Alien;
```

I Offentliga_Raser_view kan en användare med lägre auktorisation få tillgång till en vy över aliens som inte är hemligstämplade, i stället för den kompletta tabellen över alla aliens. Det finns även en ”motsatsvy” som ger tillgång till alla aliens som *är* hemligstämplade, som kan göra det enklare för att få en överblick över den samlingen.

```
CREATE VIEW Offentliga_Raser_view AS
SELECT IDkod, ras_namn
FROM Alien
WHERE ras_namn <> 'HEMLIGSTÄMPLAT' OR ras_namn IS NULL
GROUP BY ras_namn;

CREATE VIEW Hemliga_Aliens_view AS
SELECT IDkod, ras_namn
FROM Alien
WHERE ras_namn = 'HEMLIGSTÄMPLAT'
GROUP BY ras_namn;
```

Vyn Nått_Begränsning_view är en speciallösning för att se vilka användare som har nått sin begränsning vid användandet av en procedur. I denna databas finns det exempelvis en gräns på att endast få radera 3 aliens, om inte en administratör eller annan med högre auktoritet säger att man får radera fler. I den situationen är det fördelaktigt att kunna se vilka som behöver en nollställning av sitt användande eller utökning av sin begränsningkvot.

```
CREATE VIEW Nått_Begränsning_view AS
SELECT användare AS 'USER', antal_användningar AS 'ANVÄNDNINGAR',
begränsning 'GRÄNS', procedure_namn
FROM Procedure_Begränsning
WHERE antal_användningar = begränsning;
```

I denna vy kan en användare se medelvärdet på användandet av procedurer. Detta kan vara bra för att exempelvis föra statistik över hur ofta vissa procedurer används, och därefter justera eventuella begränsningar och liknande. Detta är speciellt bra för organisationen att veta vilka procedurer som genomsnittligen används mest.

```
CREATE VIEW AVG_Användning_view AS
SELECT procedure_namn, AVG(antal_användningar)
FROM Procedure_Begränsning
GROUP BY procedure_namn;
```

Stored Procedures

För att få en säkrare databas skapas procedurer för att en användare ska kunna få tillgång till fördesignade ”funktioner” i stället för att denna skapa egna. En användare med tillgång till eventuellt känsliga tabeller kan riskera att manipulera dessa på ett sätt som är oönskat. Det gör det även enklare

för en användare att använda färdiga funktioner i stället för att manuellt gå igenom många steg för samma resultat.

En sådan enkel procedur är nollställ_alla_maxade, som sätter ett värde på noll, i kolumnen ’användningar’ i tabellen Procedure_begränsning när dess värde är ekvivalent eller högre än värdet i ’begränsning’. Denna procedur har skapats för administratörer att nollställa agenter som är speciellt effektiva på sina uppdrag. Genom att sen kalla på proceduren och fånga resultatet i exempelvis @resultat, kan man se vilka som påverkades av proceduren.

```
CREATE PROCEDURE nollställ_alla_maxade (OUT resultat TEXT)
    BEGIN
        -- Hämtar alla som har nått sin maxgräns och sätter in i
'resultat'.
        SELECT GROUP_CONCAT(användare SEPARATOR ', ') INTO resultat
        FROM Procedure_Begränsning
        WHERE användningar >= begränsning;

            -- Det är samma personer som i resultat som kommer att få sina
användningar nollställt.
            UPDATE Procedure_Begränsning
            SET användningar = 0
            WHERE användningar >= begränsning;
    END;

CALL nollställ_alla_maxade(@resultat);
SELECT @resultat;
```

En annan liknande procedure, kallat nollställ_begränsning finns även. Denna tar i stället in två argument, en specifik person som skall påverkas, och vilken begränsning som det berör, för det kan vara så att enbart en viss bestämd procedur ska nollställas för en viss person.

```
CREATE PROCEDURE nollställ_begränsning (IN agent VARCHAR(50), IN kommando
VARCHAR(50))
    BEGIN
        UPDATE Procedure_Begränsning
        SET användningar = 0
        WHERE användare = agent
        AND procedure_namn = kommando;
    END;
```

En lite mer komplex procedur har även skapats. Denna procedur hemligstämplar en hel ras. Alla befintliga aliens som har rasen kommer att bli hemligstämplade. Information från tabellen som innehåller all information om alien, nämligen *Alien*-tabellen kommer att **flyttas** till tabellen *Hemligstämplat_Logg* för att loggföras. Denna tabell ska såklart vara *hemlig* och inte tillgänglig för de flesta.

```
-- Hemligstämplar alla aliens rasfält som har param_ras_namn samt rasen
självt.
CREATE PROCEDURE hemligstämpla_på_ras_namn(IN param_ras_namn SMALLINT)
    BEGIN
        -- Loggför aliens med rasen som hemligstämplas. --
        INSERT INTO Hemligstämplat_Logg(alien_id, ras_namn)
        SELECT alien_id, ras_namn FROM Alien
        WHERE ras_namn = param_ras_namn;

        -- Sparar information om rasen för återskapande senare.
        UPDATE Hemligstämplat_Logg, Kännetecken_Tillhör_Ras
        SET ras_kännetecken = kännetecken
        WHERE ras_namn = param_ras_namn;

        -- Uppdaterar rasen på aliens med param_ras_namn till
'HEMLIGSTÄMPLAT'.
        UPDATE
            Alien
        SET
            ras_namn = 'Hemligstämplat'
        WHERE
            ras_namn = param_ras_namn;
    END;
```

På andra sidan finns också proceduren avklassificera som gör i princip motsatsen till föregående procedur. Den återger rasen som hemligstämplats till alla alien som påverkades med hjälp av loggen.

```
-- Avklassificerar både ras och alien med ras_namn.
CREATE PROCEDURE avklassificera(IN param_ras_namn SMALLINT)
    BEGIN
        -- Återskapar eventuella kännetecken för rasen.
        INSERT IGNORE INTO Kännetecken_Tillhör_Ras (ras_namn, kännetecken)
        SELECT ras_namn, ras_kännetecken FROM Hemligstämplat_Logg
        WHERE Hemligstämplat_Logg.ras_namn = param_ras_namn;

        -- Återger rasen till alla Aliens med samma ras innan
hemligstämplande.
        UPDATE
            Alien,
            Hemligstämplat_Logg
        SET
            Alien.ras_namn = param_ras_namn
        WHERE
            Alien.alien_id = Hemligstämplat_Logg.alien_id
            AND Hemligstämplat_Logg.ras_namn = param_ras_namn;

    END;
```

Det finns även en avklassificera_alien som tar in ett specifikt alien_id som parameter. Tanken är att efter en hemligstämpling av en alien, om någon gjort fel eller om hemligstämpeln på informationen på en alien har utgått, kan man avklassificera denne. Då passerar man in en alien_id som argument. Skulle det visa sig att id:et är fel, eller inte är hemligstämplat får man även ett **felmeddelande** för att upplysa användaren om detta.

```
-- Avklassificerar en specifik Alien.
CREATE PROCEDURE avklassificera_alien(IN param_alien_id CHAR(25))
    BEGIN
        DECLARE matchning TINYINT;

        -- Kontrollerar att alien_id är hemligstämplat till att börja med.
        SELECT COUNT(*) INTO matchning
        FROM Hemligstämplat_Logg
        WHERE alien_id = param_alien_id;

        -- matchning bör vara antingen 0 ELLER 1, då alien_iden är unik.
        -- resultatet = 0 : alien INTE hemlig.
        -- resultatet = 1 : alien är hemlig.
        IF matchning = 0 THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'ALIEN_ID existerar inte eller är EJ
HEMLIGSTÄMPLAT.';

        ELSE
            UPDATE
                Alien,
                Hemligstämplat_Logg
            SET
                Alien.ras_namn = Hemligstämplat_Logg.ras_namn
            WHERE
                Alien.alien_id = Hemligstämplat_Logg.alien_id
                AND Hemligstämplat_Logg.alien_id = param_alien_id;
        END IF;
    END;
```

Triggers

Triggers har använts flitigt för att säkerställa att både insättning av data och radering sker korrekt. Det används också för att se vilka förändringar som har skett i databasen, vem, vad och när en förändring skedde.

Först och främst, om man vill lägga till en alien som en registrerad eller oregistrerad måste man vanligtvis först lägga till information i *Alien* tabellen, eftersom registrerade och oregistrerade aliens är en subtyp av alien. Här finns dock en trigger som gör att en läggs till i Alien tabellen automatiskt innan, om den inte redan finns med.

```
CREATE PROCEDURE addera_alien(IN new_id CHAR(25))
    BEGIN
        INSERT IGNORE INTO Alien (alien_id) VALUES (new_id);
    END;

CREATE TRIGGER addera_registrerad
    BEFORE INSERT
    ON Registrerad_Alien
    FOR EACH ROW
    BEGIN
        DECLARE ny_ras VARCHAR(30);
        DECLARE sparade_raser SMALLINT;

        CALL addera_alien(NEW.alien_id);
        IF NEW.hemplanet IS NOT NULL OR NEW.hemplanet <> '' THEN
            UPDATE Alien
            SET ras_namn = CONCAT(NEW.hemplanet, 'ian')
            WHERE Alien.alien_id = NEW.alien_id;
        END IF;
    END;

CREATE TRIGGER addera_oregistrerad
    BEFORE INSERT
    ON Oregistrerad_Alien
    FOR EACH ROW
    BEGIN
        CALL addera_alien(NEW.alien_id);
    END;
```

I samband med detta triggas även en annan trigger, logga_insättning_alien, som gör en logg på detta när en alien läggs till i Alien tabellen, som då sker automatiskt när man gör en insättning på någon av subtyperna. Detta betyder att vilket man än väljer, att göra en insättning på Alien, registrerad_alien eller oregistrerad_alien så kommer den alltid att finns med i en logg.

```
CREATE TRIGGER logga_insättning_alien
    BEFORE INSERT
    ON Alien
    FOR EACH ROW
    BEGIN
        INSERT INTO Trigger_Logg (trigger_typ, data, användare)
            VALUES ('INSÄTTNING', CONCAT('Alien: ', NEW.alien_id),
CURRENT_USER);
    end;
```

Även här finns det en motsats, en loggning för radering av aliens, den fungerar på liknande sätt.

```
CREATE TRIGGER logga_raderingar_alien
    BEFORE DELETE
    ON Alien
    FOR EACH ROW
    BEGIN
        INSERT INTO Trigger_Logg (trigger_typ, data, användare)
            VALUES ('RADERING', CONCAT('Alien: ', OLD.alien_id),
CURRENT_USER);
    END;
```

Dessa loggningar sätts sedan in i en tabell som heter Trigger_Logg. Denna tabell kommer att innehålla alla insättningar och raderingar på aliens. Just Alien tabellen var valts då det är mycket som bygger på den, de flesta kopplingar görs mot Alien, och tabeller som ärver, ärver från just Alien.

```
CREATE TABLE Trigger_Logg (
    logg_id INT AUTO_INCREMENT,
    loggtid TIMESTAMP DEFAULT CURRENT_TIMESTAMP(),
    trigger_typ ENUM ('RADERING', 'INSÄTTNING', 'UPPDATERING'),
    data    VARCHAR(32) NOT NULL,
    användare   VARCHAR(50) NOT NULL,
    PRIMARY KEY (logg_id)
);
```

Liknande typer av triggers finns även på Vapen, dock är dessa inte loggade. *Vapen*tabellen är speciell eftersom en split har gjorts på den, därför finns dock en trigger för loggning av uppdateringar på tabellen, det vill säga om ett vapen byter ägare, måste detta loggföras, och sedan uppdateras den utsplittade tabellen Vapen_Ägare.

```
CREATE TRIGGER logga_uppdatering_vapen
    AFTER UPDATE
    ON Vapen
    FOR EACH ROW
    BEGIN
        INSERT INTO Trigger_Logg (trigger_typ, data, användare)
            VALUES ('UPPDATERING', CONCAT('Vapen: ', OLD.vapen_id),
CURRENT_USER);

        UPDATE Vapen_Ägare
        SET Vapen_Ägare.skepp_id = NEW.skepp_id,
            Vapen_Ägare.alien_id = NEW.alien_id
        WHERE Vapen_Ägare.vapen_id = NEW.vapen_id;
    END;
```

En annan trigger finns på skepp. Denna bestämmer två saker; om ett skepp registreras utan en tillverkningsplanet, sätts denna automatiskt till ’Okänt’. Två, den säkerställer att ett skepps antal sittplatser finns inom spannet av 1-5000. Skulle det vara så att de registrerade sittplatserna hamnar utanför detta spann, avbryts insättningen och ett felmeddelande ges till användaren.

```
CREATE TRIGGER chk_skepp
BEFORE INSERT
ON Skepp
FOR EACH ROW
BEGIN
    -- Det finns för lite information för att veta var ett skepp är
tillverkat,
    -- därför sätts tillverkningsplaneten som OKÄNT i det fall då fältet är
tomt.
    IF NEW.tillverkningsplanet IS NULL OR '' THEN
        SET NEW.tillverkningsplanet = 'OKÄNT';
    END IF;

    -- CONSTRAINTS som ser till att sittplatser är mellan 1-5000.
    IF NEW.sittplatser IS NULL OR NEW.sittplatser = '' THEN
        SET NEW.sittplatser = FLOOR(1 + RAND() * 5000);
    ELSEIF NEW.sittplatser > 5000 OR NEW.sittplatser < 1 THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'SITTPLATSER must be between 1 and 5000.';
    END IF;
END;
```

Rättigheter

För att få bättre koll på användare av databasen skapas en tabell för att hålla denna information. Där finns användarnamn, och lösenord, samt en roll för att kunna säkerställa att rätt rättigheter ges i enlighet med rolltypen.

En användare av typen ’agent’ är en ’vanlig’ användare som finns i organisationen och får många rättigheter som har med att hemligstämpling av aliens och liknande. En användare av typen ’administratör’ får utöver de rättigheter som en sedvanlig agent har, mer förvaltande rättigheter, bland annat tillgång till nollställning och utökande av begränsningar på procedurer.

```
-- Skapar olika USERS för databasen med specifika rättigheter beroende på
USER typ.
CREATE USER IF NOT EXISTS 'a21liltr_agent'@'%' IDENTIFIED BY 'foo';
CREATE USER IF NOT EXISTS 'a21liltr_administratör'@'%' IDENTIFIED BY 'bar';

INSERT INTO användare (användarnamn, lösenord, roll) VALUES ('agent',
'foo', 'agent'),
                                                            ('administratör',
'bar', 'administratör');

-- Rättigheter till en "vanlig" agent.
GRANT EXECUTE ON PROCEDURE a21liltr.radera_alien TO 'a21liltr_agent'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.radera_skepp TO 'a21liltr_agent'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.avklassificera TO 'a21liltr_agent'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.avklassificera_alien TO
'a21liltr_agent'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.hemligstämpla_på_ras_namn TO
'a21liltr_agent'@'%';
GRANT SELECT ON a21liltr.Alien_Personnummer_view TO 'a21liltr_agent'@'%';
GRANT SELECT ON a21liltr.Offentliga_Raser_view TO 'a21liltr_agent'@'%';
GRANT SELECT ON a21liltr.Hemliga_Aliens_view TO 'a21liltr_agent'@'%';
GRANT SELECT ON a21liltr.Kännetecken_Entitet_view TO 'a21liltr_agent'@'%';


-- Rättigheter till en agent (administratör) med högre auktoritet.
GRANT EXECUTE ON PROCEDURE a21liltr.radera_alien TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.radera_skepp TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.avklassificera TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.avklassificera_alien TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.hemligstämpla_på_ras_namn TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.nollställ_begränsning TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.nollställ_alla_maxade TO
'a21liltr_administratör'@'%';
GRANT EXECUTE ON PROCEDURE a21liltr.ändra_begränsning TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.Alien_Personnummer_view TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.Offentliga_Raser_view TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.Hemliga_Aliens_view TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.Kännetecken_Entitet_view TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.Nått_Begränsning_view TO
'a21liltr_administratör'@'%';
GRANT SELECT ON a21liltr.AVG_Användning_view TO
'a21liltr_administratör'@'%';

-- Laddar om så att användarna får sina rättigheter.
FLUSH PRIVILEGES;
```