

Air Crash Booking System

Eksamensopgave i **Databaser (dDB)**, Eo6
Vejleder: Louis Salvail

Afleveret 27. oktober 2006 af:
Jens Gram Pedersen, 20041039, mail@jensgram.dk

28 nummererede sider

INDHOLDSFORTEGNELSE

INDLEDNING	1
E/R-DAIGRAM & ENTITETSSÆT	2
Overvejelser i forbindelse med modellen.....	4
CONSTRAINTS, DATABASE-SCHEMA & DOMÆNER	6
Nøgler.....	6
Single-Value begrænsninger.....	6
Referentiel integritet.....	6
Andre begrænsninger.....	6
Relationelt database-schema & domæner.....	7
FD'ER & RELATIONEL ALGEBRA	8
Funktionelle afhængigheder.....	8
Relationel algebra (opgaver fra ugeseddel).....	9
SQL, INDEKSER & SIKRING MOD OVERBOOKING	12
Oprettelse af tabeller og constraints i SQL.....	12
Indekser.....	15
Sikring mod overbooking.....	15
Oversættelse af relationel algebra til SQL (opgaver fra ugeseddel).....	16
Yderligere opgaver fra ugesedlen.....	17
JAVA-INTERFACET, PROBLEMER & TRANSAKTIONER	19
Brug af Java-interfacet.....	19
Problemer.....	19
Brug af transaktioner.....	20
DISKUSSION AF DESIGN & ÆNDRINGER	22
Løbende ændringer & revideringer.....	22
Assertions & triggers.....	22
Brugere & autorisationer.....	25
KONKLUSION & DISKUSSION	27
REFERENCER	28

INDLEDNING

Denne opgave omhandler det endelige system, der er udarbejdet i løbet af kurset. Opgavens første del er en sammenskrivning af de løbende aflevering ud fra den endelig databasemodel. De overordnede afsnit er derfor baseret på de løbende afleveringer, men tilrettet til at afspejle den endelige model.

Opgavens anden del indeholder behandling af de problemstillinger, der blev præsenteret i forbindelse med eksamensopgaven.

Air Crash Booking System er et databasebaseret system, der kan håndtere flyafgange, passagerer, lufthavne, reservation, sæde-bookinger m.v. Dertil hører en Java-applikation, der muliggør bestilling af billetter, oprettelse af nye afgange, samt yderligere administration.

Air Crash er et fiktivt luftfartsselskab, der tilbyder rejser i mindre fly. Det betyder, at der er plads til mellem 5 og 20 personer på de forskellige afgange. Ikke desto mindre skal systemet kunne håndtere *frequent flyers*, elektroniske billetter etc. De specifikke krav diskuteres sammen med den følgende gennemgang af systemet.

E/R-DIAGRAM & ENTITETSSÆT

Gennem arbejdet med *Air Crash* systemet, er den endelige databasestruktur endt som illustreret på nedenstående E/R-diagram:

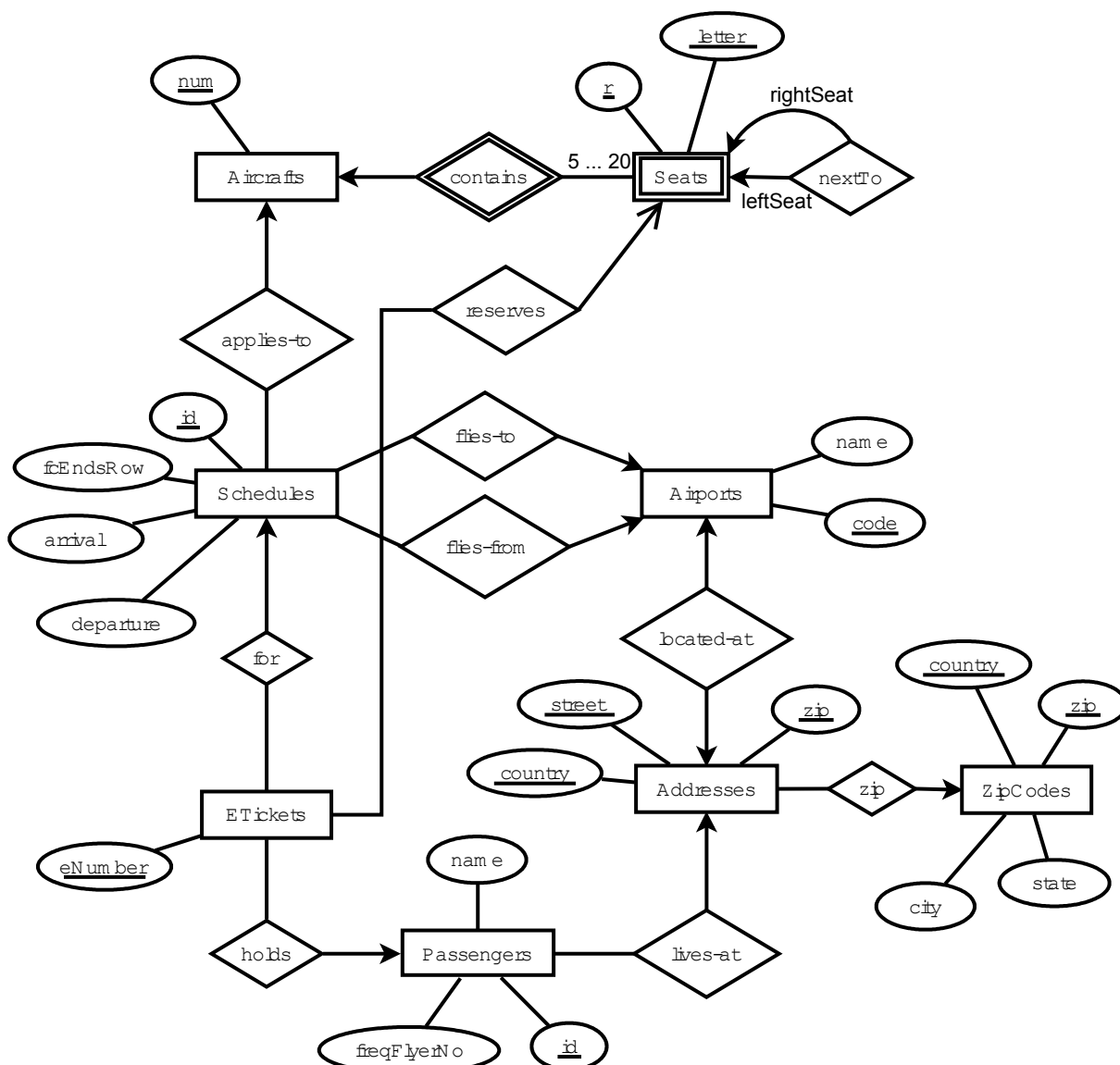


Illustration 1: E/R-diagram for Air Crash Booking System. Relationer markeret med udfyldte pile angiver én-til-mange (og én-til-én) relationer, hvor der påkræves referentiel integritet. Mere herom senere. Relationen mellem ETickets og Seats kræver blot en "maksimalt én" relation og piler er derfor ikke udfyldt.

Systemet baserer sig således på 8 entitetssæt:

- **Aircrafts:** Dette entitetssæt indeholder et sæt af fly, der udgør *Air Crash*'s samlede flåde. Hvert fly har et unikt nummer, der kunne være på formen "YZ 192" etc. Dette entitetssæt kunne desuden indeholde mere specifikke flydata (model, fabrikant, maksimal kapacitet), men disse aspekter har jeg ignoreret i denne sammenhæng.
- **Seats:** Idéen bag dette entitetssæt er, at en entitet i Seats modsvarer et konkret sæde i et af selskabets fly (med bogstav og række¹). Det vil sige, at et sæde, der findes i virkeligheden

¹ ROW er et reserveret ord i Oracle, så række-attributten har fået navnet "r".

også findes én gang i databasen. Sæder er en svag entitet, da en del af nøglen gives gennem relationen til Aircrafts (“contains” kræver referentiel integritet, da ethvert sæde nødvendigvis skal knytte sig til et fly – ellers kan det ikke identificeres og det vil desuden ikke give mening at have et sæde, der ikke er knyttet til et fly). Jeg har valgt at lade hvert Seat indgå i en relation (“nextTo”), der knytter et sæde til nabosædet (-erne). Man kunne også have beregnet, hvilket sæder, der er ved siden af et givent sæde via bogstaverne, men jeg finder den valgte modellering mere alsidig og “korrekt”. Et sæde, der ikke indgår i en nextTo-relation som eksempelvis *rightSeat* må nødvendigvis være en vinduesplads i venstre side af flyet.

- **Addresses:** En adresse er et sæt bestående af gade (dækker over vejnavn og husnummer), land og postnummer. Adresser er internationale, hvorfor et ikke-dansk element – stat – er medtaget. Oprindeligt var by og stat attributter på Addresses-sættet, men gennem applicering af principperne for BCNF (Boyce-Codd Normal Form) blev det tydeligt, at det kunne give anledning til anomalier (by og stat afhænger af postnummer og land, jf. afsnittet “Funktionelle afhængigheder”, s. 8. I Addresses fungerer gade, postnummer og land som nøgle (jeg forudsætter, at “Lufthavnsboulevarden 6” i 2770, DK er unik). Grunden til, at adresser er et selvstændigt entitetssæt er, at en adresse ikke er atomisk og desuden kan genbruges.
- **ZipCodes:** Som nævnt ovenfor kan postnummer og land angive by og stat (forudsat, at et postnummer ikke dækker over flere stater i samme land). ZipCodes blev introduceret for at fjerne anomalier og mindske redundans i databasen.
- **Airports:** En lufthavn har et almindeligt navn (dagligdags navn, ex: “Kastrup Lufthavn”) og en unik kode. Denne kode (ex: “DK-KBH”) fungerer derfor som nøgle. Jeg formoder, at der kun findes én lufthavn pr. adresse, hvorfor relationen til entitetssættet Addresses er en én-til-én relation.
- **Passengers:** En passager har et navn og et unikt ID, som fungerer som nøgle. Dette ID er blot et nummer. Ligesom lufthavne har en relation til en adresse, gør det samme sig gældende for passagerer. Her gælder det dog, at relationen er én-til-mange, da forskellige passagerer kan bo på samme adresse. Attributten *freqFlyerNo* behandles i det følgende afsnit.
- **Schedules:** I denne model er en “Schedule” en konkret flyafgang – eksempelvis “Fly YZ 192 DK-KBH->UK-HEAT med afgang 27. okt. '06 kl. 8:15”. Den enkelte afgang har et unikt ID, der benyttes som nøgle. Afgang- og ankomsttidpunkter er *timestamps* (dato og tid), men udgangspunkt og destination er relationer til konkrete lufthavns-entiteter. I første

omgang påkrævede relationen til Aircrafts ikke referentiel integritet, da jeg forestillede mig, at en afgang kunne være planlagt uden at være knyttet til et fly. Det gjorde det dog umuligt at sikre en afgang mod overbooking. Da jeg skønnede, at det var vigtigere, påkræves der nu referentiel integritet på alle Schedule's relationer. Attributten *fcEndsRow* benyttes til at angive, hvor i flyet der skelnes mellem økonomi- og første klasse. Jeg har vedtaget, at *fcEndsRow* er -1, hvis alle rækker er første klasse og 0, hvis alle rækker er økonomiklasse². Attributten er medtaget for at kunne ændre skellet mellem økonomi- og første klasse fra afgang til afgang. Alternativt kunne man have tilføjet en boolsk attribut på Seats-sættet (ex: *isEconomy*), men så ville klassen være knyttet direkte til sædet og derved ikke kunne ændres fra afgang til afgang!

- **ETickets:** En e-billet har et unikt nummer og knytter sig med nødvendighed til en afgang (Schedule) og en passager (Passengers). Desuden har ETickets en relation til Seats. Denne relationen kræver ikke referentiel integritet, da en E-ticket uden et tilknyttet i dette system betragtes som en booking på en afgang. En E-ticket, som *er* tilknyttet et sæde fungerer som en konkret reservation og dermed som *boarding pass*.

Overvejelser i forbindelse med modellen

Et luftfartsselskab er noget, der findes i den fysiske verden. Derfor er det også relativt let at være tro overfor virkeligheden, hvilket jeg har forsøgt i forbindelse med udarbejdelsen af modellen for *Air Crash* systemet. Som omtalt er entitetssættet Addresses indført for at undgå redundans, hvis flere passagerer bor på samme adresse. Det giver derimod ikke mindre redundans i henhold til Airports, da relationen hér er én-til-én, og det derfor havde været lige så effektivt at lade attributterne fra Addresses være attributter direkte i Airports. Alligevel har jeg valgt at være konsekvent og lade al adresseinformation være i Addresses og ZipCodes (i henhold til BCNF, som nævnt ovenfor).

Ved entitetssættet Passengers benytter jeg attributten *freqFlyerNo*. Alternativt kunne man have benyttet en "isa"-relation for at skelne almindelige passagerer og *frequent flyers*:

Da systemet ikke skal kunne andet end at skelne passagerer fra *frequent flyers*, finder jeg dog den valgte løsning mest anvendelig.

Attributten *freqFlyerNo* indeholder et nummer for *frequent flyers*, mens den blot er NULL ved

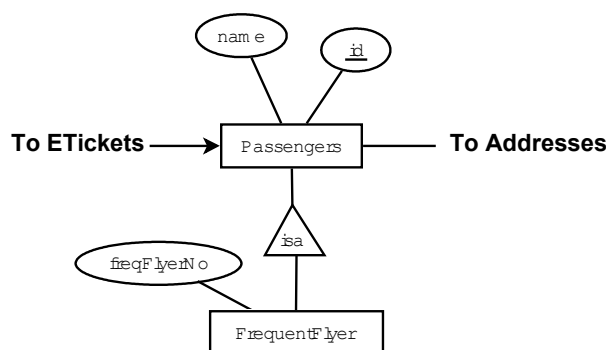


Illustration 2: Alternativ løsning med "isa"-relation.

² Se desuden afsnittet "Single-Value begrænsninger", s. 6.

almindelige passagerer. Man kan således sige, at “Use null values”-strategien er benyttet til “konvertering” fra subklasser til ét entitetsæt (Garcia-Molina *et al.* 2002, p. 76).

Modellen er i alle tilfælde forsøgt holdt så simpel som muligt. Det ses eksempelvis ved, at jeg alle steder har kunnet nøjes med binære relationer. Desuden er alle attributter simple, atomiske datatyper (jf. afsnittet “Relationelt database-schema & domæner”, s. 7 for valgte datatyper). Entitetsættet Addresses er eksempelvis fremkommet, da en adresse er kompleks og derved ikke bør behandles som en simpel datatype (én streng med værdierne fra alle attributter).

Der er forsøgt valgt løsninger, der understøtter et fleksibelt bookingscenarie, samt rekonfiguration af sæder m.v.

CONSTRAINTS, DATABASE-SCHEMA & DOMÆNER

Med udgangspunkt i ovenstående E/R-diagram er der flere simple *constraints* i modellen.

Nøgler

For de 7 “stærke” entitetssæt er nøglerne valgt, så de entydigt kan identificere enhver entitet. Entitetssættet Seats er svagt og kan derfor først unikt identificeres gennem relationen til Aircrafts (via nøglen i Aircrafts). Derfor er relationen *contains* omringet af dobbeltramme (der er tale om et *supporting relationship*).

Single-Value begrænsninger

Alle pile fra relationer i E/R-diagrammet angiver *single-value constraints*, da der hermed menes “maksimalt én” (eller “præcis én” ved udfyldte pile).

For passagerer gælder det, at den mulige NULL-værdi i *freqFlyerNo* er den ene værdi, der kan angive, at den specifikke passager ikke er *frequent flyer*.

Yderligere gælder det, at -1 er den ene værdi, der i Schedules kan angive, at alle sæder er økonomiklasse. Der er således også tale om en *single-value constraint*.

Referentiel integritet

Relationen *nextTo* ved entitetssættet Seats skal have referentiel integritet ved begge rolle (*rightSeat* og *leftSeat*). Det er på den måde altid muligt at afgøre, om to sæder er placeret ved siden af hinanden, samt om et sæde er en vinduesplads (som omtalt er et sæde, der ikke indgår i *nextTo*-relationen med en *rightSeat*-rolle en vinduesplads i flyets venstre side; det er ikke højre sæde for noget andet sæde).

I E/R-diagrammet er referentiel integritet indikeret med udfyldte pile. Således er “*reserves*”-relationen mellem ETickets og Seats eneste relation, der ikke påkræver referentiel integritet (jf. punktet “ETickets”, s. 4).

Andre begrænsninger

Som angivet på E/R-diagrammet kan en Aircraft-entitet have en relation til mellem 5 og 20 sæder.

Relationelt database-schema & domæner

Alle E/R-relationer er transformeret ind i de *schema*'er, der er entitetssæt i E/R-diagrammet. Undtaget herfra er dog *nextTo*-relationen, der har et selvstændigt *schema*. Primærnøgler er skrevet med kapitaler:

<p>Aircrafts (NUM: string)</p> <p>Seats (AIRCRAFTNUM: string, R: int, LETTER: {'A', 'B', 'C', 'D'})</p> <p>Addresses (STREET: string, ZIP: int, COUNTRY: string)</p> <p>ZipCodes (ZIP: int, COUNTRY: string, city: string, state: string)</p> <p>Airports (CODE: string, name: string, addrStreet: string, addrZip: int, addrCountry: string)</p>	<p>Passengers (ID: int, name: string, freqFlyerNo: int, addrStreet: string, addrZip: int, addrCountry: string)</p> <p>Schedules (ID: string, fcEndsRow: int, arrival: timestamp, departure: timetamp, aircraftNum: string, airportFliesFrom: string, airportFliesTo: string)</p> <p>ETickets (ENUMBER: int, passengerId: int, scheduleId: string, seatR: int, seatLetter: {'A', 'B', 'C', 'D'}, aircraftNum: string)</p> <p>nextTo (AIRCRAFTNUM: string, R: int, LLETTER: {'A', 'B', 'C', 'D'}, RLETTER: {'A', 'B', 'C', 'D'})</p>
---	---

FD'ER & RELATIONEL ALGEBRA

Herunder følger funktionelle afhængigheder (*Functional Dependencies*) for alle relationer i modellen, samt løsninger på opgaverne i relationel algebra (ugeseddel 3).

Funktionelle afhængigheder

For den relationelle datamodel har jeg identificeret følgende ikke-trivielle FD'er:

Addresses: Som tidligere omtalt, var by og stat oprindeligt attributter på Addresses-relationen, hvilket gav følgende FD'er³:

- a) street zip country → city state
- b) zip country → city state

Da “zip” og “country” ikke i sig selv er en nøgle for Addresses, blev relationen opdelt i henholdsvis `Addresses (street, zip, country)` og `ZipCodes (zip, country, city, state)`. Det betyder, at den ikke-trivielle FD (b) nu findes i relationen ZipCodes, hvor dens venstreside udgør relationens nøgle. Hermed opfylder relationen BCNF.

Airports:

- c) code → name addrStreet addrZip addrCountry
- d) addrStreet addrZip addrCountry → code name

Venstresiden af (d) er også nøgler for Airports, men *code* er den valgte primærnøgle. Begge nøgler (venstresiderne i (c) og (d)) er minimale og derfor lige gode bud på primærnøgler (jf. Garcia-Molina *et al.* 2002, p. 86).

Passengers:

- e) id → name freqFlyerNo addrStreet addrZip addrCountry
- f) freqFlyerNo → id name addrStreet addrZip addrCountry

FD'en (f) kun gælder, hvis *freqFlyerNo* ikke er NULL. Derfor kan *freqFlyerNo* ikke være en nøgle, og (f) afføder ingen ændringer i databasens *schema*.

Schedules:

- g) id → fcEndsRow arrival departure aircraftNumber airportFliesFrom airportFliesTo
- h) departure aircraftNumber → id fcEndsRow arrival airportFliesFrom airportFliesTo
- i) arrival aircraftNumber → id fcEndsRow departure airportFliesFrom airportFliesTo

Også her gælder det, at venstresiderne i både (h) og (i) kunne være primærnøgler i relationen, men jeg har valgt attributten *id*, da det synes mest logisk.

³ FD'en “state → country” er ikke medtaget, da jeg ikke er sikker på, at navnet på en stat nødvendigvis er unik på verdensplan.

ETickets:

j) eNumber \rightarrow passengerId scheduleId seatRow seatLetter aircraftNumber

k) passengerId scheduleId \rightarrow eNumber seatRow seatLetter aircraftNumber

l) scheduleId seatRow seatLetter aircraftNumber \rightarrow passengerId eNumber

Venstresiden i (k) er en nøgle for relationen. FD'en (l) gælder kun – som (f) ved Passengers – hvis der er tilknyttet et sæde til en given E-Ticket. Derfor kan venstresiden af (l) ikke være en nøgle.

Kun Addresses brød med Boyce-Codd normalformen, hvorfor relationen – som nævnt – blev ændret og udvidet med relationen ZipCodes. Hermed opfylder alle relationer BCNF – og dermed normalformerne til og med 3. niveau.

Relationel algebra (opgaver fra ugeseddel)

A) Antallet af passagerer, der har booket plads på en given afgang (hér: AC-1234), men endnu ikke har reserveret et sæde, kan findes med:

$$COUNT(\sigma_{scheduleId='AC-1234' \wedge seatR IS NULL}(ETickets))$$

Relationen ETickets indeholder netop reservationer uden nødvendigvis at være knyttet til et sæde.

B) For at finde passagerer, der har fløjet med *Air Crash* flere end 10 gange inden for det seneste år, skal man først finde *passengerId* og *scheduleId* for det sidste års afgang. Det gøres via en *natural join* på ETickets og Schedules:

$$D(pId, sId) := \pi_{passengerId, id}(\sigma_{departure > DATE - '0001-00-00' \wedge departure \leq DATE}(ETickets \bowtie_{scheduleId=id} Schedules))$$

Herefter grupperes der på *pId* og antallet af tupler i hver gruppering summeres op:

$$C(pId, ctPass) := \sigma_{ctPass > 10}(\gamma_{pId, COUNT(pId) \rightarrow ctPass}(D))$$

Da vi ønsker specifik passagerinformation, vælger jeg at lave en *natural join* på Passengers:

$$Answer(id, name, ff, as, az, ac, pId, ctPass) := (Passengers \bowtie_{id=pId} C)$$

C) Fly, der ikke er overbookede og flyver fra Viborg (“DK-VIB”) til Odense (“DK-ODE”) inden for de næste 3 dage, findes ved først at finde *scheduleId* og *aircraftNum* på de afgang, der opfylder betingelserne:

$$SA(sId, num) := \pi_{scheduleId, aircraftNum} (\sigma_{airportFliesFrom='DK-VIB' \wedge airportFliesTo='DK-ODE' \wedge DATE < departure \leq DATE + '0000-00-03'} (Schedules))$$

Herefter summeres antallet af sæder på de fundne afgang:

$$C1(sId, num, ctSeats) := SA \bowtie_{num, ctSeats} (\gamma_{aircraftNum, COUNT(aircraftNum) \rightarrow ctSeats} (Seats))$$

C1 indeholder nu *sId* (*scheduleId*), *num* (*aircraftNum*) og *ctSeats* (antallet af sæder for hver afgang). På tilsvarende måde tælles antallet af reservationer for hver afgang. I denne omgang kigger jeg på alle Eticket-entiteter, da jeg alligevel skal *joine* med *C1*.

$$C2(sId, ctTick) := \gamma_{scheduleId, COUNT(eNumber) \rightarrow ctTick} (ETickets)$$

C2 indeholder således antallet af reservationer for enhver afgang, der har mindst én Eticket. Det er ligemeget, om en Eticket er knyttet til et sæde, da en booking også vil omfatte et sæde før eller siden og således skal tælles med. Jeg udtrækker nu alle *scheduleId*'er, hvor der stadig er flere sæder end reservationer:

$$Sc(sId) := \pi_{sId} (C1 \bowtie_{ctTick < ctSeats} C2)$$

Dette resultat kan *joines* direkte med Schedules:

$$Answer(id, fcEndsRow, arrival, departure, aircraftNum, airportFliesFrom, airportFliesTo) := (Schedules \bowtie Sc)$$

D) For at finde to ledige sæder ved siden af hinanden – økonomiklasse, afgang AC-1234 – henter jeg først *aircraftNum* og *fcEndsRow* fra Schedules:

$$A(num, fc) := \pi_{aircraftNum, fcEndsRow} (\sigma_{id='AC-1234'} (Schedules))$$

Hermed har jeg mulighed for at finde de Seats, hvis “row” er større end *fc* (det skal jo være på økonomiklasse):

$$S1(n, r, l) := \pi_{aircraftNum, r, letter} (A \bowtie_{num=aircraftNum \wedge fc < r} Seats)$$

Det vil sige, at *S1* nu har en struktur, der svarer til Seats. Det samme vil jeg finde i de ETickets, der er knyttet til et sæde på afgang AC-1234:

$$S2(n, r, l) := \pi_{aircraftNum, r, letter} (\sigma_{scheduleId='AC-1234' \wedge seatR IS NOT NULL} (ETickets))$$

Ledige sæder (*S*) kan nu findes som *S1* EXCEPT *S2*:

$$S(n, r, l) := S1 - S2$$

Herefter *joines* jeg S med *nextTo*-relationen i *leftSeat*-rollen:

$$L(n, r, lLetter, rLetter) := (S \underset{l=lLetter}{\bowtie} nextTo)$$

Resultatet er nu de entiter i S , hvor l er lig $rLetter$ fra L . Her vil ikke være dupletter i form af par af sæder, da det ikke er to Seats der *joines*, men to *joins* med Seats på *nextTo* (jf. Garcia-Molina *et al.* 2002, p. 257).

$$Answer(n, r, lLetter, rLetter) := (S \underset{l=rLetter}{\bowtie} L)$$

I dette resultat vil være alle ledige sædepar på afgang AC-1234, hvis der er nogle.

SQL, INDEKSER & SIKRING MOD OVERBOOKING

I dette afsnit er database-*schema*'et fra s. 7 blevet oversat til SQL. Herudover vises, hvilke indekser jeg har lavet i databasen, samt hvordan en *assertion* kan sikre mod overbooking på en flyafgang. I sidste del findes løsninger på SQL-opgaverne fra ugeseddel 4.

Oprettelse af tabeller og constraints i SQL

I forbindelse med oprettelse af tabellerne i Oracle-databasen, har jeg angivet primærnøgler, fremmednøgler, samt NOT NULL- og CHECK-constraints. Jeg har ikke haft behov for DEFAULT *values*, da jeg forkastede en idé om, at *departure* i Schedules kunne være det nuværende tidspunkt, hvis ikke andet blev angivet. Det virkede for søgt og kom derfor ikke med i den endelige model:

Aircrafts: Her er ingen *constraints*, da en primærnøgle aldrig kan være NULL.

```
CREATE TABLE Aircrafts (
    num CHAR(10) PRIMARY KEY
);
```

Seats: Da alle attributter er nøgler, har jeg ikke benyttet NULL-*constraints*. Attributten *r* ("row") skal være mellem 1 og 20, mens *letter* skal indeholde karakteren A, B, C eller D. Intet fly må have flere end 20 sæder, hvilket nedenstående CHECK-*constraint* sikrer⁴.

```
CREATE TABLE Seats (
    aircraftNum CHAR(10) NOT NULL REFERENCES Aircrafts(num),
    r INT CHECK (r >= 1 AND r <= 20),
    letter CHAR(1) CHECK (letter IN ('A','B','C','D')),
    PRIMARY KEY (aircraftNum, r, letter),
    CHECK (20 >= ALL (SELECT COUNT(*) FROM Seats GROUP BY
        aircraftNum))
);
```

Addresses: Sidste linie angiver en fremmednøgle-*constraint* til ZipCodes.

```
CREATE TABLE Addresses (
    street VARCHAR(255),
    zip INT NOT NULL,
    country VARCHAR(50) NOT NULL,
    PRIMARY KEY(street, zip, country),
    FOREIGN KEY (zip, country) REFERENCES ZipCodes(zip, country)
);
```

⁴ Dette CHECK er ikke medtaget i den aktuelle Oracle-database, da det gav en ret kryptisk fejl. Hverken jeg eller Rune (TA) har været i stand til at identificere fejlen. I stedet er dette CHECK blevet udformet som en *trigger*, jf. afsnittet "Assertions & triggers", s. 22. Problemstillingen behandles desuden i afsnittet "Problemer", s. 19.

ZipCodes: I denne tabel må kun *state* være NULL. Attributten *zip* skal være et tal på maksimalt 6 tegn. Jeg har desuden vedtaget, at det ikke må være tallet 666⁵.

```
CREATE TABLE ZipCodes (
  zip INT CHECK (zip < 1000000 AND zip <> 666 AND zip > 0),
  country VARCHAR(50),
  city VARCHAR(50) NOT NULL,
  state VARCHAR(15),
  PRIMARY KEY(zip, country)
);
```

Airports: En lufthavn skal have et unikt navn, som endvidere ikke må være NULL. Det er ligeledes et krav, at fremmednøgler i “Addresses” eksisterer.

```
CREATE TABLE Airports (
  code CHAR(9) PRIMARY KEY,
  name VARCHAR(30) NOT NULL UNIQUE,
  addrStreet VARCHAR(255) NOT NULL,
  addrZip INT NOT NULL,
  addrCountry VARCHAR(50) NOT NULL,
  FOREIGN KEY (addrStreet, addrZip, addrCountry) REFERENCES
    Addresses(street, zip, country)
);
```

Passengers: Hvis en passager har er *frequent flyer*, skal attributten *freqFlyerNo* være unik. Alle passagerer kan dog have NULL som værdi for denne attribut, hvis de blot er “almindelige” passagerer. Ligesom for Airports gælder det, at fremmednøgler i “Addresses” skal eksistere, samt at *name* ikke må være NULL.

```
CREATE TABLE Passengers (
  id INT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  freqFlyerNo INT UNIQUE,
  addrStreet VARCHAR(255) NOT NULL,
  addrZip INT NOT NULL,
  addrCountry VARCHAR(50) NOT NULL,
  FOREIGN KEY (addrStreet, addrZip, addrCountry) REFERENCES
    Addresses(street, zip, country)
);
```

Schedules: Som omtalt ved punktet “Schedules”, s. 3, kræves der referentiel integritet ved attributten *aircraftNum*, hvorfor den ikke må være NULL. Attributterne *airportFliesTo* og *airportFliesFrom* er fremmednøgler fra tabellen Airports⁶. Ved planlægning af en flyafgang giver det ikke mening ikke at angive tidspunkter for afgang og ankomst. Derfor må disse attributter ikke være NULL – desuden benyttes der et CHECK til at sikre, at ankomsttidspunktet er senere end afgangtidspunktet. Som tidligere omtalt skal der være mulighed for at sætte *fcEndsRow* til -1, men ikke andre negative tal.

⁵ Dette er udelukkende for at lave en alternativ *constraint* – det har ingen praktisk betydning.

⁶ Jeg overvejede constrainten “CHECK (airportFliesFrom <> airportFliesTo)”. Den blev dog forkastet, da jeg fandt det plausiblet, at *Air Crash* måske også skulle kunne tilbyde rundflyvninger (det er trods alt små fly etc.).

```

CREATE TABLE Schedules (
  id INT PRIMARY KEY,
  fcEndsRow INT CHECK (fcEndsRow >= -1 AND fcEndsRow < 20),
  arrival TIMESTAMP NOT NULL,
  departure TIMESTAMP NOT NULL,
  aircraftNum CHAR(10) NOT NULL REFERENCES Aircrafts(num),
  airportFliesFrom CHAR(9) NOT NULL REFERENCES Airports(code),
  airportFliesTo CHAR(9) NOT NULL REFERENCES Airports(code),
  CHECK (arrival > departure)
);

```

ETickets: For e-billetter gælder det, at fremmednøglerne fra både Aircrafts, Passengers og Schedules skal være opretholdt (*aircraftNum* er en del af nøglen for den svage relation Seats). Som omtalt, kan en e-billet godt findes uden at være tilknyttet et sæde (hvilket sker via attributterne *seatR*, *seatLetter* og *aircraftNum*). Hvis blot én værdi er NULL skal alle 3 attributter dog være det, da det ellers ikke er en gyldig relation til et sæde (nøglen i Seats er minimal). Gennem denne *constraint* bliver det muligt at tjekke om en e-billet gælder som reservation af et sæde via enhver af de 3 nævnte attributter. En anden *constraint* er, at en E-ticket skal være unik mht. *scheduleId* og sæde (kun én e-billet kan knytte sig til et specifikt sæde pr. afgang).

```

CREATE TABLE ETickets (
  eNumber INT PRIMARY KEY,
  passengerId INT NOT NULL REFERENCES Passengers(id),
  scheduleId INT NOT NULL REFERENCES Schedules(id),
  seatR INT,
  seatLetter CHAR(1),
  aircraftNum CHAR(10),
  UNIQUE(scheduleId, seatR, seatLetter, aircraftNum),
  FOREIGN KEY (seatR, seatLetter, aircraftNum) REFERENCES Seats
    (r, letter, aircraftNum),
  CHECK (
    (seatR IS NULL AND seatLetter IS NULL AND aircraftNum IS
     NULL)
    OR
    (seatR IS NOT NULL AND seatLetter IS NOT NULL AND
     aircraftNum IS NOT NULL)
  )
);

```

nextTo: De to sæder, der refereres til i *nextTo*-relationen vil nødvendigvis være placeret i samme fly. Der er derfor ikke grund til at lade *aircraftNum* og *r* ("row") optræde to gange. Gennem en CHECK-*constraint* sikres det, at de to Seat-entiteter, der omfattes af relationen, har "tilstødende" bogstaver (attributten *letter*).

```

CREATE TABLE nextTo (
  aircraftNum CHAR(10) NOT NULL,
  r INT NOT NULL,
  lLetter CHAR(1) NOT NULL,
  rLetter CHAR(1) NOT NULL,
  PRIMARY KEY(aircraftNum, r, lLetter, rLetter),
  FOREIGN KEY (aircraftNum, r, lLetter) REFERENCES Seats
    (aircraftNum, r, letter),
  FOREIGN KEY (aircraftNum, r, rLetter) REFERENCES Seats
    (aircraftNum, r, letter),
  CHECK (
    (lLetter = 'A' AND rLetter = 'B') OR
    (lLetter = 'B' AND rLetter = 'C') OR
    (lLetter = 'C' AND rLetter = 'D')
  )
);

```

Indekser

SQL sørger for, at der oprettes indekser for alle nøgler (både primære og unikke). Ud over disse indekser har jeg fundet det relevant at oprette følgende:

```

CREATE INDEX PassNameIndex ON Passengers(name);
CREATE INDEX SchedDepartIndex ON Schedules(departure);
CREATE INDEX SchedAirportIndex
  ON Schedules(airportFliesFrom, airportFliesTo);
CREATE INDEX ETickPassIndex ON ETickets(passengerId);
CREATE INDEX ETickSchedIndex ON ETickets(scheduleId);

```

Ræsonnementet bag disse indekser er, at de implicerede attributter ofte benyttes ved *queries*, hvor de benyttes som relationer, samt i søgninger. Eksempelvis vil SchedDepartIndex kunne benyttes i *queries*, hvor samtlige afgange i det seneste år skal findes etc.

SchedAirportIndex omfatter to attributter. Attributten *airportFliesFrom* er bevidst nævnt først i definitionen af indekset, da indekset således også kan benyttes som indeks på denne attribut alene (jf. Garcia-Molina *et al.* 2002, p. 296). Jeg har vurderet, at dette vil benyttes mere end et selvstændigt indeks på *airportFliesTo*.

Sikring mod overbooking

For at sikre, at en flyafgang ikke overbookes, kan følgende *assertion* benyttes. Oracle understøtter ikke *assertions*. Det vender jeg tilbage til i afnittet “Assertions & triggers”, s. 22.

```

CREATE ASSERTION NoOverbookCheck CHECK (aircraftNum NOT IN
  (
    SELECT aircraftNum FROM (
      (
        SELECT COUNT(*) AS cntSeats, aircraftNum FROM Seats
          GROUP BY aircraftNum
      ) NATURAL JOIN (
        SELECT COUNT(*) AS cntPass, s.aircraftNum FROM ETickets
          e, Schedules s WHERE e.scheduleId = s.id
          GROUP BY s.aircraftNum
      )
    ) WHERE cntSeats <= cntPass
  )
);

```

Idéen er, at man tæller antallet af sæder på ethvert fly og foretager et *natural join* med de billetter, der er knyttet til det enkelte fly. Da en e-billet ikke nødvendigvis er en reservation af et specifikt sæde, kan der være tupler i ETickets, hvor *aircraftNum* er NULL. Det er derfor nødvendigt at *join* ETickets med Schedules for også at medtælle passagerer, som endnu ikke har reserveret et sæde (de skal jo have et sæde før eller siden).

Oversættelse af relationel algebra til SQL (opgaver fra ugeseddel)

Opgaverne i relationel algebra fra ugeseddel 3 kan oversættes til SQL-queries som følger (se afsnittet “Relationel algebra (opgaver fra ugeseddel)”, s. 9 for grundigere opgavebeskrivelse):

A) Antal passagerer uden sædereservation:

```
SELECT COUNT(*) AS cntPass FROM ETickets WHERE scheduleId =
    'AC-1234' AND seatR IS NULL;
```

! BEMÆRK: I det følgende benytter jeg en lidt speciel notation for at øge læsbarheden. Således er “*subqueries*” navngivne og benyttes i følgende *queries*. De følgende *queries* kan oversættes til korrekt SQL ved at substituere henvisninger til disse *subqueries*.⁷

B) Passagerer, der har fløjet med *Air Crash* flere end 10 gange inden for det seneste år:

```
D := SELECT passengerId pId, scheduleId sId FROM
    (SELECT * FROM ETickets, Schedules WHERE scheduleId = id)
    WHERE departure > CURRENT_DATE - INTERVAL '1' YEAR
    AND departure <= CURRENT_DATE
```

D er hermed en tabel indeholdende alle *passengerId* og *scheduleId* for det seneste år.

```
C := SELECT pId, ctPass FROM
    (SELECT pId, COUNT(pId) AS ctPass FROM D GROUP BY pId)
    WHERE ctPass > 10
```

C indeholder nu ID og antal flyvninger for alle passagerer, der har fløjet flere end 10 gange.

```
SELECT * FROM Passengers, C WHERE id = pId;
```

Passagerinformation findes ved at *join* *C* og *Passengers*.

⁷ Man kunne også eksplicit have kreeret *views* til dette formål, men den valgte syntaks skulle være relativt let at forstå.

C) Fly, der ikke er overbookede og flyver fra Viborg (“DK-VIB”) til Odense (“DK-ODE”) inden for de næste 3 dage. Jeg antager til denne opgave, at den skitserede *assertion* til sikring mod overbooking er implementeret, hvorved opgaven bliver ret simpel (og viser styrken af *constraints* til at sikre sig, at data der ikke er ugyldige data):

```
SELECT aircraftNum FROM Schedules WHERE airportFliesFrom = 'DK-VIB'
AND airportFliesTo = 'DK-ODE'
AND departure > CURRENT_DATE
AND departure <= CURRENT_DATE + INTERVAL '3' DAY;
```

D) At finde to ledige sæder ved siden af hinanden – økonomiklasse, afgang AC-1234 – kræver, først og fremmest, at man finder de økonomiklasse-sæder på afgangen:

```
S1 := SELECT aircraftNum AS n, r, letter AS l FROM
      (SELECT aircraftNum AS num, fcEndsRow AS fc FROM Schedules
       WHERE id = 'AC-1234'), Seats
      WHERE num = aircraftNum AND fc < r
```

Ligeledes skal reserverede sæder på afgangen findes:

```
S2 := SELECT aircraftNum AS n, r, letter AS l FROM ETickets
      WHERE scheduleId = 'AC-1234' AND seatR IS NOT NULL
```

Ledige sæder på økonomiklasse er nu de sæder i *S1*, der ikke indgår i *S2*:

```
S := SELECT S1 EXCEPT8 S2
```

Herefter *joines* først med sæder fra *nextTo*-relationen, hvor sædet har rollen som *leftSeat*:

```
L := SELECT n, r, lLetter, rLetter FROM S, nextTo
      WHERE l = lLetter
```

Ved dernæst at *joine* resultatet af ovenstående (*L*) med *S*, hvor sædet har rollen som *rightSeat*, ender man med ledige sæder på økonomiklassen, der er placeret ved siden af hinanden:

```
SELECT n, r, lLetter, rLetter FROM S, L WHERE l = rLetter;
```

Yderligere opgaver fra ugesedlen

E) For at finde antal afgang og det totale antal solgte billetter for hver lufthavn, hvortil der kan flyves fra Horsens (“DK-HORS”) inden for den næste uge, kan følgende *query* benyttes. Resultatet er sorteret i faldende orden efter antallet af afgange:

```
A := SELECT * FROM Schedules WHERE departure > CURRENT_DATE
      AND departure <= CURRENT_DATE + INTERVAL '7' DAY
      AND airportFliesFrom = 'DK-HORS'
```

A indeholder nu alle afgange i den næste uge, der afgår fra Horsens.

```
B := SELECT airportFliesTo AS destination, COUNT(airportFliesTo)
      AS cntTickets FROM ETickets, A
      WHERE scheduleId = id GROUP BY airportFliesTo
```

A *joines* med *ETickets* for at tælle billetter for hver afgang.

⁸ EXCEPT hedder MINUS i Oracle, men jeg har valgt SQL-navnet for læsbarhedens skyld.

```
| C := SELECT airportFliesTo AS destination, COUNT(airportFliesTo)
      AS cntFlights FROM A GROUP BY airportFliesTo
```

Herefter indeholder *C* alle lufthavnsdestinationer og antallet af afgang dertil. Det endelige resultat findes gennem et *natural join* på *B* og *C* og sorteres efter antallet af afgang i faldende orden:

```
| SELECT * FROM B NATURAL JOIN C ORDER BY cntFlights DESC;
```

F) *Frequent flyers* med mindst 10 flyvninger findes ved at *joine* Passengers og ETickets, hvor *freqFlyerNo* i Passengers er forskellig fra NULL:

```
| SELECT name, COUNT(eNumber) AS cntFlights FROM Passengers, ETickets
      WHERE freqFlyerNo IS NOT NULL AND id = passengerId
      GROUP BY passengerId HAVING COUNT(eNumber) >= 10;
```

JAVA-INTERFACET, PROBLEMER & TRANSAKTIONER

I læsegruppen har vi udviklet et tekstbaseret Java-interface, hvorigennem *Air Crash Booking System* kan benyttes. I det følgende beskrives dette interface, samt de problemer, der har været forbundet med udviklingen deraf. I forbindelse med denne del af projektet, blev der ikke brug for yderligere *assertions* (foruden den føromtalte, jf. “Sikring mod overbooking”, s. 15).

Brug af Java-interfacet

Det var et krav, at interfacet kunne håndtere 3 forskellige brugssituationer:

1. Bestilling af ny billet.
2. Booking af sæde og udskrivning (til skærm) af *boarding pass*.
3. Administration (herunder oprettelse af afgang og håndtering af *frequent flyers*).

Interfacet er baseret på klassen *CmdInterface*, der giver brugeren adgang til systemets 3 facetter. Denne klasse er den eneste med en *main*-metode. Interfacet bruges ved at indtaste nummeret på det ønskede menupunkt, samt data efter forespørgslen fra programmet. Når en handling er udført, vender programmet tilbage til udgangspositionen. Vi har vurderet, at det ekstra arbejde, der er forbundet med *Swing* ikke kunne betale sig, da et tekstbaseret interface giver tilstrækkelige muligheder.

Programmet er meget intuitivt i brug og viser i flere tilfælde gyldige værdier inden indtastning. Dette er dog ikke tilfældet, når brugeren skal indtaste afgang- og billetnumre.

Problemer

Flere steder i databasen er der brugt strenge af typen CHAR (jf. afsnittet “Oprettelse af tabeller og constraints i SQL”, s. 12). Denne datatype kan indeholde en streng af en fast længde, modsat VARCHAR, hvis længde er indholdets maksimale længde. Det viste sig at være et problem, hvis en attribut af typen CHAR indeholder en streng, der er kortere end attributtens definerede længde. I et sådant tilfælde bliver der tilføjet mellemrum til enden af strengen, hvorved eksempelvis *name* i Airports (af typen CHAR(9)) kommer til at indeholde “DK-KBH ”, når man indsætter “DK-KBH”. Interne relationer påvirkes ikke af denne opførsel (da refererende attributter har samme definition), men det er et problem, når brugerinput skal sammenlignes med værdier i databasen. Problemet er blevet løst ved at bruge SQL-funktionen `TRIM()`, der fjerner mellemrum i begge ender af en streng. Alternativt kunne man have ændret typen til VARCHAR.

I administrationsinterfacet opførte Oracle sig ret underligt ved menupunkt 5 (“Find most frequent flyer(s) in period”). I første omgang forsøgte jeg at benytte en *query*, der

indbefattede “... cnt HAVING cnt = MAX(cnt) ...”, men det ville ikke fungere. Idéen var, at alle passagerer med samme antal flyvninger (cnt) skulle vises i stedet for blot den ene passager, der tilfældigvis havde det mindste ID, “første” navn eller andet. Problemet var, at Oracle ikke lod til at fortolke HAVING-delen, hvorfor forespørgslen nu er ændret til en væsentlig længere version.

Som omtalt i afsnittet “Oprettelse af tabeller og constraints i SQL”, s. 12, var vi ved oprettelse af tabellen Seats tvunget til at fjerne den *constraint*, der skulle sikre, at intet fly havde flere end 20 sæder:

```
CREATE TABLE Seats (
  aircraftNum CHAR(10) NOT NULL REFERENCES Aircrafts(num),
  r INT CHECK (r >= 1 AND r <= 20),
  letter CHAR(1) CHECK (letter IN ('A', 'B', 'C', 'D')),
  PRIMARY KEY (aircraftNum, r, letter),
  CHECK (20 >= ALL (SELECT COUNT(*) FROM Seats GROUP BY
    aircraftNum))
);
```

Tanken var, at *constraint*'en skulle være sand så længe ethvert fly har 20 eller færre sæder, men Oracle vil ikke tillade denne *constraint*. Løsningen er blevet, at denne *constraint* nu er udformet som en *trigger*, jf. afsnittet “Assertions & triggers”, s. 22.

Brug af transaktioner

Der er to aspekter i en transaktion: Serialiserbarhed og atomicitet. Et sæt af operationer siges at udføres serielt, hvis de udføres fuldstændigt før enhver anden operation. Serialiserbarhed er således et udtryk for, at operationer opfører sig som serielle handlinger, selv om der er overlap i tiden. Atomicitet betyder, at et sæt af operationer enten skal udføres komplet eller slet ikke. Således vil atomicitet sikre, at en operation, der består af to eller flere *queries* aldrig afbrydes og derved efterlader databasen i en inkonsistent tilstand. Konkret sikres atomicitet ved, at operationerne udføres lokalt og derefter *commit*'es.

Man kan vælge at slække på kravet til serialiserbarhed for en transaktion, hvis der er tale om operationer, der ikke ændrer data i databasen (såkaldte *read-only* operationer). Det er i dette tilfælde op til SQL-systemet at afgøre, hvorvidt dette slækkede krav kan benyttes til at tillade parallelle handlinger (Garcia-Molina *et al.* 2002, p. 403).

Enhver *query* er automatisk en transaktion og udføres derfor serielt og atomisk. Af samme grund har der kun været brug for at benytte transaktioner eksplicit i de tilfælde, hvor en funktion benytter sig af flere *queries*. Et sådant tilfælde findes i administrationsinterfacet (menupunkt 0, “Schedule a new flight”), hvor afgangens ID beregnes ved at lægge én til det samlede antal af afgang. I denne sammenhæng er det vigtigt, at det samlede sæt af *queries*

behandles serielt, så man ikke risikerer, at det fundne ID er ugyldigt. Det kunne blive tilfældet i den situation, hvor to brugere forsøger at oprette nye afgang på samme tidspunkt – og derved kunne få genereret samme ID.

I de tilfælde, hvor vi har brugt transaktioner er de af typen *read/write*, da det netop er den serielle opførsel, vi agter at sikre. Da vores transaktioner består af en læsning og derefter en skrivning er det derimod ikke så vigtigt, at der er atomicitet (skrivningen foregår jo som det sidste). Transaktionstypen *read/write* er standardtypen for transaktioner, da det er den strengeste. Det samme gælder *isolation levels*, hvor udgangspunktet er *serializable*. Det betyder, at *dirty reads* ikke er tilladt – det vil sige, at en *query* i transaktionen ikke kan læse data, der ikke er blevet *committed*.

DISKUSSION AF DESIGN & ÆNDRINGER

Løbende ændringer & revideringer

Databasen til *Air Crash* systemet har naturligvis udviklet sig i takt med, at der kom nye krav dertil. Som udgangspunkt har selve strukturen været relativt stabil med entitetsættene Aircrafts, Seats, Schedules, Airports, ETickets, Passengers og Addresses. I forbindelse med udryddelse af anomalier og redundans blev det klart, at et selvstændigt entitetsæt til ZipCodes ville være den korrekte løsning.

Modelleringen af sædekonfiguration har være emne for en del mindre justeringer i takt med de stigende krav til de mulige forespørgsers kompleksitet. I første omgang havde vi i gruppen modelleret to relationer til og fra Seats for at kunne finde nabosæder. Det viste sig hurtigt meget redundant, hvorefter vi endte med *nextTo*-relationen, der fungerer tilfredsstillende til de nuværende systemkrav.

I takt med de stigende krav til datakonsistens, er flere og flere relationer mellem entitetsæt kommet til at påkræve referentiel integritet. Således er det nu kun *reserves* mellem ETickets og Seats, der ikke kræver referentiel integritet. Dette skyldes, at en ETicket både tjener som booking på en afgang og som konkret sædereservation (*boarding pass*).

Da ændringerne er blevet foretaget løbende har det været overkommeligt. Til gengæld har det ofte medført, at modeller, relationel algebra og *queries* fra tidligere uger også måtte revideres. Det er klart, at man i dag har et større indblik i, hvordan man starter “rigtigt”, men jeg ser ikke behov for gennemgribende ændringer som sådan, da modellen virker logisk og relativt simpel. Denne påstand mener jeg underbygges af, at jeg i meget få tilfælde har behov for flere selvstændige *queries* – langt de fleste opgaver kan klares med ren SQL og *subqueries*.

Man kan indvende, at *aircraftNum* er redundant i ETickets, da en e-billet altid er knyttet til en afgang (der påkræves referentiel integritet fra ETickets til Schedules). Derfor kan man altid finde *aircraftNum* via Schedules, men jeg har valgt at beholde attributten, da den tjener som en del af nøglen for det potentielle sæde – ikke direkte som indikator for e-billetens tilknyttede fly.

Assertions & triggers

Som nævnt i afsnittet “Sikring mod overbooking”, s. 15, havde vi i gruppen udarbejdet følgende *assertions* til at sikre ethvert fly mod overbooking:

```

CREATE ASSERTION NoOverbookCheck CHECK (aircraftNum NOT IN
(
  SELECT aircraftNum FROM (
    (
      SELECT COUNT(*) AS cntSeats, aircraftNum FROM Seats
      GROUP BY aircraftNum
    ) NATURAL JOIN (
      SELECT COUNT(*) AS cntPass, s.aircraftNum FROM ETickets
      e, Schedules s WHERE e.scheduleId = s.id
      GROUP BY s.aircraftNum
    )
  ) WHERE cntSeats <= cntPass
);

```

Da Oracle ikke understøtter *assertions* må den omskrives til en *trigger*. En *trigger* er langt mere specifik end en *assertion*, da den omfatter enten *update*, *insert* eller *delete*. I modsætning hertil evalueres en *assertion* ved enhver modifikation af de implicerede tabeller.

For at slippe for en masse gentagen SQL, vil jeg først oprette to *views*, der kan gøre det følgende mere overskueligt. Først og fremmest et *view*, der blot indeholder antallet af sæder:

```

CREATE VIEW AcSeatCnt AS
  SELECT COUNT(*) AS cnt FROM Seats GROUP BY aircraftNum;

```

Dernæst et *view*, der indeholder overbookede fly (og derfor altid bør være tom):

```

CREATE VIEW OverbookedAcs AS
  SELECT aircraftNum FROM (
    (
      SELECT COUNT(*) AS cntSeats, aircraftNum FROM Seats
      GROUP BY aircraftNum
    ) NATURAL JOIN (
      SELECT COUNT(*) AS cntPass, s.aircraftNum
      FROM ETickets e, Schedules s WHERE e.scheduleId = s.id
      GROUP BY s.aircraftNum
    )
  ) WHERE cntSeats < cntPass
;

```

Første mulighed for at overbooke et fly opstår, hvis man forsøger at slette et sæde. Derfor genindsættes et sæde, hvis det kommer til at betyde, at flyet har færre sæder end der er reservationer:

```

CREATE TRIGGER DeleteSeatTrigger
AFTER DELETE ON Seats
REFERENCING
  OLD TABLE AS oldT
FOR EACH STATEMENT
WHEN (5 > ANY (SELECT cnt FROM AcSeatCnt)
  OR EXISTS (SELECT * FROM OverbookedAcs))
  INSERT INTO Seats (SELECT * FROM oldT
  WHERE (aircraftNum, r, letter) NOT IN Seats);

```

Denne *trigger* sørger desuden for, at intet fly kan ende med at have færre end 5 sæder (jf. afsnittet "Problemer", s. 19).

Dernæst kan et fly blive overbooket, hvis man ændrer *aircraftNum* på et sæde (det er måske ikke så realistisk, at man forsøger at flytte et sæde fra ét fly til et andet – denne *trigger* er dog nødvendig for til enhver tid at kunne sikre en konsistent database). I dette tilfælde sørger jeg desuden for, at ethvert fly altid har mellem 5 og 20 sæder (jf. ovenfor):

```
CREATE TRIGGER UpdateSeatAcTrigger
AFTER UPDATE OF aircraftNum ON Seats
REFERENCING
    OLD TABLE AS oldT,
    NEW TABLE AS newT
FOR EACH STATEMENT
WHEN (5 > ANY (SELECT cnt FROM AcSeatCnt)
      OR 20 < ANY (SELECT cnt FROM AcSeatCnt)
      OR EXISTS OverbookedAcs)
BEGIN
    DELETE FROM Seats WHERE (aircraftNum, r, letter) IN newT;
    INSERT INTO Seats (SELECT * FROM oldT);
END;
```

For nu at afrunde sikringen af, at ethvert fly har mellem 5 og 20 sæder, vælger jeg også at oprette en *trigger*, der gælder ved indsættelse af et sæde:

```
CREATE TRIGGER InsertSeatTrigger
AFTER INSERT ON Seats
REFERENCING
    NEW ROW AS newR
FOR EACH ROW
WHEN (20 < ANY (SELECT cnt FROM AcSeatCnt))
    DELETE FROM Seats s WHERE s.aircraftNum = newR.aircraftNum
    AND s.r = newR.e AND s.letter = newR.letter;
```

Hvis en afgang skal befordres af et andet fly, er det også nødvendigt at tjekke for overbooking. Det gøres med en *trigger* på ændring af *aircraftNum* i Schedules:

```
CREATE TRIGGER UpdateScheduleAcTrigger
AFTER UPDATE OF aircraftNum ON Schedules
REFERENCING
    OLD TABLE AS oldT,
    NEW TABLE AS newT
FOR EACH STATEMENT
WHEN (EXISTS (SELECT * FROM OverbookedAcs))
BEGIN
    DELETE FROM Schedules
    WHERE (id, fcEndsRow, arrival, departure, aircraftNum,
           airportFliesFrom, airportFliesTo) IN newT;
    INSERT INTO Schedules (SELECT * FROM oldT);
END;
```

Desuden er det nødvendigt at tjekke ved indsættelse i ETickets:

```
CREATE TRIGGER InsertETicketTrigger
AFTER INSERT ON ETicket
REFERENCING
    NEW ROW AS newR
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM OverbookedAcs))
    DELETE FROM ETickets e WHERE e.eNumber = newR.eNumber;
```

Sidste scenario omfatter, at *scheduleId* ændres for en e-billet. Også i dette tilfælde vælger jeg at genoprette databasen som den var før ændringen:

```

CREATE TRIGGER UpdateETicketSchTrigger
AFTER UPDATE OF scheduleId ON ETickets
REFERENCING
    OLD TABLE AS oldT,
    NEW TABLE AS newT
FOR EACH STATEMENT
WHEN (EXISTS (SELECT * FROM OverbookedAcs))
BEGIN
    DELETE FROM ETickets WHERE (eNumber, passengerId, scheduleId,
        seatR, seatLetter, aircraftNum) IN newT;
    INSERT INTO ETickets (SELECT * FROM oldT);
END;

```

Alt i alt skulle det nu betyde, at databasens tilstand altid er tilfredsstillende mht. antal sæder i ethvert fly, samt sikring mod overbooking. Bemærk, at ovenstående *triggers* er skrevet efter SQL-specifikationen – ikke efter Oracle's syntaks, der indeholder mindre variationer.

Brugere & autorisationer

Hvis *Air Crash* systemet skulle have 3 brugertyper, fordelt som:

- a) Rejseselskaber, der skal kunne bestille billetter,
- b) Ansatte, der skal kunne udstede *boarding passes*, og
- c) Administratorer, der kunne oprette afgang og håndtere *frequent flyers*,

vil det være oplagt at give dem forskellige rettigheder i systemet. Med det nuværende system vil disse rettigheder (*privileges*) skulle gives som:

- **SELECT**

- a) Passengers, ETickets, Schedules og Airports. Dertil kommer *view*'et nsFree, der afføder nsRightIsfree, nsLeftIsfree, nsThisIsFree, nsAll, nsThisIsUsed, nextTo og Seats.
- b) ETickets og Schedules. Dertil kommer *views*: allInfo, nsFree og nsNotFree, der afføder numbers, passName, fliesToName, fliesFromName, Passengers, Airports, nsRightIsfree, nsLeftIsfree, nsThisIsFree, nsAll, nsThisIsUsed, nextTo, Seats, nsNotFree og alle aw*-*views*.
- c) Airports, Aircrafts, Schedules, Passengers, ETickets.

- **INSERT**

- a) ETickets.
- b) (ingen rettigheder)
- c) Schedules.

- **UPDATE**
 - a) (ingen rettigheder)
 - b) ETickets.
 - c) Passengers.
- **REFERENCES** (gælder for alle brugertyper): Addresses (via Airports og Passengers); Passengers, Schedules og Seats (via ETickets); Airports og Aircrafts (via Schedules).
- **TRIGGER**: Hvis man tolker administratorgrupper som ansatte i *Air Crash*, der skal bruge Admin-interfacet, skal ikke engang denne gruppe have rettigheder til at oprette *triggers*. I denne sammenhæng er det en opgave for databaseadministratoren. De relationer, der omfattes af en *trigger* skal være tilgængelige for den bruger, der opretter den, men ikke for brugere, som udfører en handling, der “vækker” en *trigger* (Garcia-Molina *et al.* 2002, p. 411).

Angivelsen af ovenstående rettigheder tager udgangspunkt i den aktuelle implementation med de funktionaliteter, der kræves i hvert af de 3 moduler.

KONKLUSION & DISKUSSION

Gennem arbejdet med *Air Crash Booking System*, har vi i gruppen lavet en databasemodel, der kan løse de givne fordringer. Derudover mener jeg, at den endelige løsning er relativt fleksibel og i høj grad afspejler virkeligheden.

I løbet af kurset er der sket flere ændringer i modellen – efterhånden som vi er blevet klogere og problemerne er blevet mere komplekse. Denne udvikling er behandlet ovenfor.

Der er benyttet *constraints* til at sikre, at databasen altid er i en gyldig tilstand. Desuden benytter vi i Java-interfacet transaktioner i de tilfælde, hvor flere *queries* skal udføres serielt, samt komplet eller slet ikke. Også hér er målet at sikre konsistens i databasens informationer.

Det er min intention, at denne rapport skal vise motivationen og rationalet bag beslutninger i forbindelse med databasens design, samt de metoder, der er benyttet til at sikre konsistens.

Alle *queries* og udtryk i relationel algebra er kommenteret. Dette er ligeledes for at tydeliggøre ræsonnementet bag hvert skridt.

Alt i alt er jeg tilfreds med resultatet, da det opfylder de stillede krav. Desuden finder jeg modellen logisk og i overensstemmelse med mine tanker om, hvordan et bookingsystem bør fungere.

REFERENCER

Garcia-Molina, H., J. D. Ullman & J. Widom. (2002). *Database Systems: The Complete Book*. Upper Saddle River, NJ: Prentice Hall, Inc.