dDB uge 6 – Final Exam

Project: DanskeRustneBaner – DRB

--

Søren Løbner, 20050677, lobner@daimi.au.dk

Oktober 2007

Contents

Database projekt

Opgaven lyder på at designe en databasestruktur til brug i et baneselskab ved navn DRB, DanskeRustneBaner. Denne opgave er stillet som obligatorisk del af kurset dDB på Datalogisk institut, DAIMI, ved Århus Universitet i efteråret 2007.

Initielt design

Som en begyndelse designede vi vores struktur vha. et Entitets og Relations Diagram, E/R diagrammet. Dette dannede dermed grundlaget for konstruktionen af databasen i SQL.

Og vi har designet vores E/R diagram for selskabet "Danske Rustne Baner" efter overvejelser om hvordan de forskellige entiteter skal arrangeres. Det har givet en struktur som vi mener er optimal mht. de krav som vi er givet. Sektionerne i E/R diagrammet er delt op efter *Trains, Wagons and Seats, Connections, Passengers, E-Tickets and Seat Reservations, Railway Stations*

E/R-diagrammet er lavet i samarbejde med: Anders Halager, Anders Riggelsen og Troels Hansen. Denne gruppe blev efterfølgende splittet op, da vi ikke måtte arbejde 4 mand i samme gruppe. Hvorefter jeg udfærdigede de øvrige ugers gruppearbejde selvstændigt, med undtagelse af uge 5, hvor Anders Riggelsen gav mig en hånd med programmerings delen.

Trains, Wagons and Seats

Som udgangspunkt mener vi at *seat*-entiteten udgør et centralt punkt i strukturen. Da det er disse som systemet i sidste ende skal kunne foretage optimal tildeling af, til kunder som køber billetter med reservation. Vi mener også at sædet er centralt på den måde, at det ligesom er *wagon*, som er "bygget" uden om sædet. Det viser vi ved at vognen indeholder en many-one relation *is-in* til sæderne.

Da redundans så vidt muligt forsøges undgået, har vi lavet et sæde, som har *is-a* relationer til de forskellige sædeklasser *economy, firstclass* og *sleep.* Disse klasser binder sig også udelukkende til sædet, og ikke til vognen som de er placeret i.

Sædet har attributterne *letter* og *row* som viser sædets placering i toget. Toget som entitet har vi i vores model helt undladt, da vi mener at det er overflødigt og vil skabe redundans, i og med det kan representeres lige så godt i form af et sæt af vognenes *ID*-attributter. Og passagerer som køber en billet, køber udelukkende retten til at rejse en strækning, plus muligvis en plads. Men ikke specifikt hvilken vogn de skal sidde i. Det bliver tildelt automatisk, da alle vognene i startkonfigurationen ikke nødvendigvis er dem som ankommer til destinationen.

Connections

En connection er en forbindelse fra en station til en anden. Med en many-one relation *RunsOn* til wagons representeres vognenes strækning som en connection entitet. Den har *Departure* og *Arrival* tidspunkter og en *EndPoints* relation til hvilke stationer der stoppes ved.

De tre *Price* attributter representere priserne på de forskellige sædeklasser.

Passangers

Passengers entiteten indeholder kundedata om passagerene i DRB. Passengers entiteten befinder sig under E-Tickets, og har attributterne ID, Name, Age og OrderNo.

E-Tickets and Seat Reservations

Vores billetter knytter sig til en afstand. Denne afstand består af to stationer med eller uden mellemliggende stationer. Den samlede pris består af prisen for hver enkelt afstand, som funktion af billettypen. Billettypen kan variere fra station til station. F.eks. kan en passager bestille på en tur der strækker sig fra kl. 20 til 04 næste morgen, bestille et *1stClass-seat* for de *Connections* der rejses mellem i tidsrummet 20-23 og en *Sleep-seat* for *Connections* i tiden 23 til 04.

Railway Stations

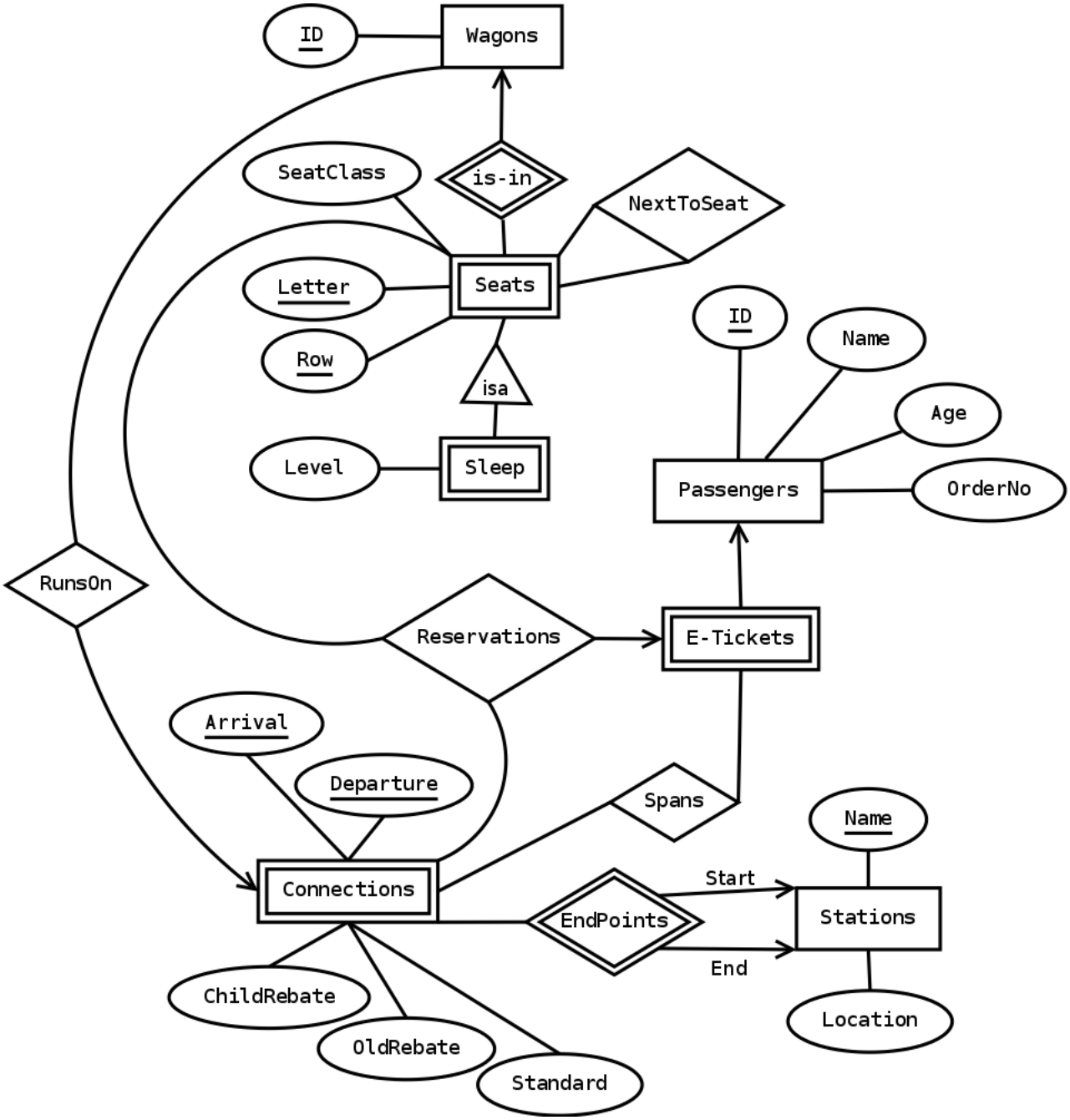
Igen har vi som i tilfældet med *Passenger*-entiteten undladt denne. Da den så at sige er representeret i entitetssættet bestående af *Wagons* og *Connections*. Et tog bliver som oftest delt ved rejser som spænder over flere stationer. Derfor knytter den indirekte *schedule* sig til *Wagons-Connections* relationen. Med tilhørende *Departure* og *Arrival* tider.

Konstruktion af E/R diagram

Tilføjelse af constraints

Disse findes i 5 former; Key, Single-value, Referential integrity, Domain og General constraints.

En tilføjelse af disse til E/R diagrammet ser ud som følger:



Endeligt E/R diagram

Som det ses, så er key constraints de understregede attributter i E/R diagrammet. Disse, og så Referential integrity constraints, som er de relationships som forbinder de forskellige entity-sets, de er de eneste constraints jeg mener jeg kan angive direkte i mit E/R diagram.

I E/R diagrammet er weak entity-sets også markerede. De kommer i betragtning når nu vi skal lave skemaerne for relationerne lidt senere.

Andre constraints såsom bl.a. domain constraints er betingelsen om, hvilken type indhold attributterne har. Sådan som, at 'Passengers(age)' attributten, og de forskellige 'Connections(*Rebate)' attributters indhold er, og kan kun representeres ved integers og ikke tekst. Rebate er en procent-sats, som trækkes fra prisen for den valgte SeatClass, over den pågældende Connection.
Single value constraints binder sig til unikke attributter, det er f.eks alle key-attributterne. De må selv sagt kun optræde én enkelt gang. En single value constraint, er også OrderNo-attributten til Passengers entiteten. Der er et unikt ordre nummer pr. billet bestilling, men den er ikke en key, da den ikke unikt identificerer en passager i vores database.

Yderligere så har E/R diagrammet fået disse ændringer. Som nævnt, så er Passengers-entiteten er nu inkluderet, til at indeholde passagerdata på tværs af bestillinger. Seats entity-sets har fået en NextToSeat, så nabosæder kan lokaliseres, en SeatClass som fortæller om det er et 1. klasse eller standard sæde. En ISA relation til Sleep entiteten, fortæller om 1. klasses eller standard sædet er en soveplads, og i hvilket niveau den befinder sig.

Fra E/R til Relationelt design.

Ud fra det producerede E/R diagram, er det nu muligt, at danne de skemaer som beskriver databasestrukturen. Disse skemaer formes ud fra de i E/R diagrammet konstruerede entity-sets og relationships. Vi har givet tre fremgangsmåder til at konvertere disse diagrammer. Den første er E/R-style konvertering, så er det den Objekt Orienterede tilgang, og sidst er det NULLs metoden.

Da jeg har et E/R diagram, har jeg valgt at foretage en E/R-style konvertering. Det gør jeg ved at danne relationer ud fra mine entity-sets og mine relationships, i det omfang det behøves. NULLs metoden synes jeg ikke virker overskuelig, da alle mine attributter og entity-sets ville skulle vises i en enkelt relation og OO-tilgangen ville, hvis vi har $n+1$ entity-sets, i værste fald producere 2^n relationer. Da vi skal minimere pladsen og undgå redundans, valgte jeg derfor E/R-style konvertering.

Under hver relation vil jeg liste de enkelte constraints, hvis der er nogle knyttet dertil. Som tidligere nævnt, er alle attributter som er Keys, også Single Value constraints, så dem vil jeg ikke liste.

Skemaer dannet fra E/R

- **Wagons(ID)**

- **Seats(Letter, Row, SeatClass, isaSleep, wagon_id)**
  - **SeatClass: er enten '1. klasse' eller 'standard'**
  - **isaSleep: Boolean om det er en soveplads**
  - **wagon_id: er togvognens unikke ID**

- **NextToSeat(r_Letter, r_Row, r_wagon_id, l_Letter, l_Row, l_wagon_id)**
  - **r_: er sædet til venstre for**
  - **l_: er sædet til højre for**

- **Sleep(Level, SeatLetter, SeatRow, wagon_id)**
  - **Level: er '1' eller '0'**

- **Connections(Departure, Arrival, ChildRebate, OldRebate, Std.Rebate, StartName, EndName, wagon_id)**

- **Spans(Departure, Arrival, StartName, EndName, E-TicketID)**

- **RunsOn(wagon_id, Departure, Arrival, StartName, EndName)**

- **E-Ticket(E-TicketID)**

- **Passengers(ID, Name, Age, OrdreNo)**
  - **Age: er et positivt tal**

- **Reservations(PassengerID, Departure, Arrival, StartName, EndName, SeatLetter, SeatRow, SeatClass, isaSleep, wagon_id)**
  - **isaSleep: Boolean om det er en soveplads**

Udformning af relationer

Functional Dependancies

Ved at bestemme functional dependancy (FD) i en relation, kan denne optimeres og redundans kan minimeres. FD er når en eller flere attributter funktionelt definerer en eller flere andre attributter. Det kan udtrykkes som: $\{A_1, A_2, \ldots A_n\} \rightarrow \{B_1, B_2, \ldots B_n\}$

Ikke-trivielle FD's er alle entiteters keys og deres dertilhørende attributter. Som f.eks.:

Stations: { Name } → { Location }
Da ingen stationer har det samme navn, bestemmes Name til vores key.

Seats: { Row, Letter, WagonID } → { SeatClass }
Row og Letter definere ikke alene hvilken type sæde der er tale om. Men så snart at WagonID kommer med, så bliver SeatClass entydigt bestemt derudfra. Dette er da definitionen på en komplet-ikke-triviel FD.

Connections: { Departure, Arrival, StartPoint, EndPoint } → { ChildRebate, OldRebate, Std.Rebate}
For hver strækning og tidspunkt vil der være en unik pris for alle sæderne, så der vil ikke kunne findes flere entries i databasen med de tidspunkter og start/slut stationer som vil give forskellige priser. Derfor er det en FD.

Passengers: { ID } → { Name, Age, OrderNo }
ID vil som key funktionelt definere de øvrige attibutter i Passengers-entiteten.

Boyce-Codd Normal Form

Ved en gennemgang af mine skemaer mener jeg ikke at kunne finde nogen som bryder med BCNF formen. De FD'er som er listet oven over har alle en ikke-triviel venstreside som er en superkey. Det vil sige, at venstresiden funktionelt bestemmer højresiden.

Gruppearbejde i relationel algebra

Jeg kunne ikke få det LaTeX til at virke ordentligt. Men jeg fik at vide at pseudo SQL også ville blive godkendt. Så Anders Riggelsen og mig fik flg. opstillet:

- *How many seats are still available (i.e. not booked) on today's 10am direct connection between Vejle and Fredericia?*

```
A(WagonID) :=
SELECT wagon_id
FROM Connection
WHERE
      depart LIKE '%10am' AND
      start = "Vejle" AND
      end = "Fredececia"

      B(row, letter, id) := natural join (Seats, A)

      C(row, letter, id) := natural join (Reservations, A)

Answer(trains available) := count(B - C)
```

De ledige sæder er derfor fundet som B – C

2) *"Find all ways together with their price to reach Aalborg from Frederikshavn today with at most one stop in between."*

```
SELECT  (A.price + B.price), A.start, B.start, B.end
FROM Connections AS A, Connections AS B
WHERE
      A.start = "Frederikshavn" AND
      A.end = B.start AND
      B.end = "Aalborg" AND
      A.depart = $today AND
      B.arrive = $today
```

3) *When should I leave Aarhus the latest if I want to be in Randers before 2PM today with at most one stop?*

```
SELECT max(A.depart)
FROM Connection AS A, Connection AS B
WHERE
      A.start = ″Aarhus″ AND
      A.end = B.start AND
      B.end = ″Randers″ AND
      B.arrive < 2pm today
```

4) *"Find 2 neighbouring seats (if possible) all the way along the following route: Aarhus-Vejle-Kolding anytime today provided the waiting time in Vejle is less than 1 hour."*

```
W(id) :=
    SELECT WagonID
    FROM Runs-On as A,Runs-On AS B
    WHERE
    A.start = ″Århus″ AND
    A.slut = B.start = ″Vejle″ AND
    B.slut = ″Kolding″ AND
    (B.depart - A.arrive) < 1 hour

      Seats(row1, letter1, id1, row2, letter2, id2)
          := natural join (W, Close-To)

      Reserved(letter, row, id) :=
            SELECT (letter, row, id) FROM Reservations
            WHERE
            `start` IN {″Århus″, ″Vejle″} AND
            `end` IN {″Vejle″, ″Kolding″}

      Answer(row1, letter1, id1, row2, letter2, id2) :=
            SELECT * FROM Pairs, Reserved
            WHERE
            Reserved <> (Pairs.row1, Pairs.letter1, Pairs.id1) AND
            Reserved <> (Pairs.row2, Pairs.letter2, Pairs.id2)
```

Generering af SQL

Fra relationships til relations.

Nu når vi har fået vores skemaer på plads, kan SQL udtrykkene til konstruktion af relationerne laves. Det foregår ved at vi tager en 1-til-1 oversættelse af skemaerne, og sørger for at indarbejde de krævede constraints. De understregede konverteres til Keys eller Uniques, som er single value constraints. Da der udelukkende kan eksistere unikke tupler af disse.
Unique dog med undtagelsen af **NULL** værdier, da disse godt kan optræde for denne type.

Til stort set alle attributer er tilføjet en constraint med **NOT NULL** som ikke tillader indsættelse af en tuppel hvor disse attributer mangler. Enkelte steder, som f.eks. **level** attibutten, må gerne være **NULL**. For enkelte relationer er der således **CHECK**'s af om disse constraints er overholdt. Teoretisk virker de ligesom assertions, ved at værdier som ikke evaluere til **TRUE**, ikke tillades at blive indsat i relationen. I vores DBMS, som er MySQL med phpMyAdmin som frontend, lader disse constraints dog ikke til at blive overholdt. Ikke destomindre er de med.

Den første transformation er simpel:

Wagons(ID)

Skemaet konverteres så til en SQL-statement:

```
CREATE TABLE Wagons(
    wagon_id       INT PRIMARY KEY
);
```

Så er det Seats entiteten som konverteres. I denne har jeg i mit E/R diagram og skema en Boolean **isaSleep**, som så har en isa-relation til Sleep entiteten. Denne Sleep entitet har fire attributter, hvor af de tre bare er duplikater fra Seats relationen, og så attributten **level**. Den entitet er tydeligvis redundant, og derfor befandt **level** sig noget smartere under selve Seats entiteten. Og hvis der ikke specificeret en level (hvis den er **NULL**) så er det ikke en soveplads.

Seats(Letter, Row, SeatClass, isaSleep, wagon_id)

- **SeatClass: er enten '1. klasse' eller 'standard'**
- **level: Soveplads niveau 1 eller 0, eller null hvis sæde**
- **wagon_id: er togvognens unikke ID**

Da **ROW** er en kommando i SQL er attributnavnet blevet **_row** og attributten **isaSleep** fra skemaet er byttet ud med **level**.

```
CREATE TABLE Seats(
    _row             INT NOT NULL,
    letter           CHAR(1) NOT NULL,
    wagon_id         INT NOT NULL REFERENCES Wagons(wagon_id),
    UNIQUE(_row, letter, wagon_id),
```

```
    seatclass       CHAR(10) NOT NULL,
    level           INT,
    CHECK(seatclass IN ('1. klasse','Standard')),
    CHECK(level IN ('0','1',NULL))
);
```

Så laver jeg et **INDEX** over sæderne, da de bliver refereret hver gang der bliver lavet et kald til NextToSeat relationen.

```
CREATE UNIQUE INDEX SeatIndex ON Seats(_row, letter, wagon_id);
```

Og så NextToSeat relationen, som dannes ud fra flg. skema:

NextToSeat(r_Letter, r_Row, r_wagon_id, l_Letter, l_Row, l_wagon_id)

- **r_: er sædet til venstre for**
- **l_: er sædet til højre for**

Alle keys er referencer til Seats relationen.

```
CREATE TABLE NextToSeat(
    r_row           INT NOT NULL,
    r_letter        CHAR(1) NOT NULL,
    l_row           INT NOT NULL,
    l_letter        CHAR(1) NOT NULL,
    wagon_id        INT NOT NULL,
    FOREIGN KEY(r_row, r_letter, wagon_id)
                    REFERENCES Seats(_row, letter, wagon_id),
    FOREIGN KEY(l_row, l_letter, wagon_id)
                    REFERENCES Seats(_row, letter, wagon_id),
    CHECK(NOT((r_row = l_row) AND (r_letter = l_letter)))
);
```

Derefter Stations relationen. Den havde jeg glemt at få med i mit skema, men da jeg begyndte at tænke over strukturen, indså jeg at den var nødvendig. Den har et skema som ser sådan he ud:

Stations(name, location)

Og relationen:

```
CREATE TABLE Stations(
    name            VARCHAR(50)  PRIMARY KEY,
    location        VARCHAR(50)  NOT NULL
);
```

Hver gang der skal bruges et startpoint eller endpoint skal der laves opslag i Stations, derfor laves der et **INDEX** dertil.

```
CREATE UNIQUE INDEX StationNameIndex ON Stations(name);
```

Herefter er det skemaet for Connections som har flg. skema.

Connections(Departure, Arrival, ChildRebate, OldRebate, Std.Rebate,
StartName, EndName, wagon_id)

Det er jeg også i stand til at følge, og producere en relation ud fra. Da rabatterne er procentsatser, skal deres værdi befinde sig mellem 0 og 100. Og alle priserne mellem alle byerne er 42,- Jeg er ikke klar over om man logisk kan sammenligne **TIMESTAMP**'s, men da MySQL, som tidligere nævnt, ikke rigtigt retter sig efter disse tjeks, har det ikke noget problem med det.

```
CREATE TABLE Connections(
    departure        TIMESTAMP,
    arrival          TIMESTAMP,
    start_name       VARCHAR(50) REFERENCES Station(Name),
    end_name         VARCHAR(50) REFERENCES Station(Name),
    UNIQUE(departure, arrival, start_name, end_name),
    childrebate      INT NOT NULL,
    oldrebate        INT NOT NULL,
    std_rebate       INT NOT NULL,
    wagon_id         INT NOT NULL,
    CHECK(start_name <> end_name)
    CHECK(departure < arrival),
    CHECK( 0 >= (childrebate AND oldrebate AND oldrebate) <= 100)
);
```

E-Tickets er en simpel relation, med kun en ID attribut.

```
CREATE TABLE E_Tickets(
    eticket_id      INT PRIMARY KEY
);
```

Herefter Passengers relationen ud fra skemaet

Passengers(ID, Name, Age, OrdreNo)

- **Age: er et positivt tal**
- **ID: er unikt pr. person.**
- **OrdreNo: er de billetter som er købt i denne persons navn.**

Med flg. relation

```
CREATE TABLE Passengers(
    id              INT NOT NULL PRIMARY KEY,
    name            CHAR(255) NOT NULL,
    age             INT NOT NULL,
    ordreno         INT NOT NULL REFERENCES E_Tickets(eticket_id)
 );
```

Til sidst danner jeg relationerne ud fra mine relationships. Her er alle attributterne referencer fra mine entity sets.

```
CREATE TABLE Spans(
    departure       TIMESTAMP NOT NULL,
    arrival         TIMESTAMP NOT NULL,
    start_name      VARCHAR(50) NOT NULL,
    end_Name        VARCHAR(50) NOT NULL,
    FOREIGN KEY(departure, arrival, start_name, end_name)
                    REFERENCES Connections(depart, arrive, start_name,
```

```
                    end_name),
    eticket_id      INT NOT NULL REFERENCES E_Tickets(eticket_id)
);

CREATE TABLE RunsOn(
    wagon_id        INT NOT NULL REFERENCES Wagons(wagon_id),
    departure       TIMESTAMP NOT NULL,
    arrival         TIMESTAMP NOT NULL,
    start_Name      VARCHAR(50) NOT NULL,
    end_Name        VARCHAR(50) NOT NULL,
    FOREIGN KEY(departure, arrival, start_name, end_name)
                    REFERENCES Connections(depart, arrive, start_name,
                    end_name)
);

CREATE TABLE Reservations(
    eticket_id      INT NOT NULL REFERENCES E_Tickets(eticket_id),
    departure       TIMESTAMP NOT NULL,
    arrival         TIMESTAMP NOT NULL,
    start_Name      VARCHAR(50) NOT NULL,
    end_Name        VARCHAR(50) NOT NULL,
    FOREIGN KEY(departure, arrival, start_name, end_name)
                    REFERENCES Connections(depart, arrive, start_name,
                    end_name),
    _row            INT NOT NULL,
    letter          CHAR(1) NOT NULL,
    wagon_id        INT NOT NULL,
    FOREIGN KEY(_row, letter, wagon_id)
                    REFERENCES Seats(_row, letter, wagon_id)
);
```

Queries:

De queries der blev oprettet som relationel algebra, bliver som følger når de er oversat til SQL:

4) *How many seats are still available (i.e. not booked) on today's 10am direct connection between Vejle and Fredericia?*

```
SELECT COUNT(Connections.wagon_id) AS FreeSeats
FROM Connections
NATURAL JOIN Seats
WHERE
(
_row != (
      SELECT _row
      FROM Reservations
      WHERE start_Name = 'Vejle'
      AND end_Name = 'Fredericia'
      AND departure 10am $TODAY
      AND wagon_id = Connections.wagon_id )
OR letter != (
      SELECT letter
      FROM Reservations
      WHERE start_Name = 'Vejle'
      AND end_Name = 'Fredericia'
      AND departure = 10am $TODAY
      AND wagon_id = Connections.wagon_id )
)
```

Tidsbegrænsningerne er pseudosyntaks i det ovenstående og de efterfølgende, da jeg ikke lige ved hvordan MySQL sammenligner `TIMESTAMP` typer.

5) *Find all ways together with their price to reach Aalborg from Frederikshavn today with at most one stop in between.*

```
SELECT Depart.start_name AS _FROM,Arrive.start_name AS
_STOP,Arrive.end_name AS _TO
FROM Connections AS Depart, Connections AS Arrive
WHERE Depart.start_name = 'Aalborg' AND
      Depart.end_name = Arrive.start_name AND
      Arrive.end_name = 'Frederikshavn' AND
      Depart.departure = $TODAY AND
      Arrive.arrival = $TODAY AND
      Depart.departure < Arrive.arrival AND
      Depart.wagon_id = Arrive.wagon_id
```

6) *When should I leave Aarhus the latest if I want to be in Randers before 2PM today with at most one stop?*

```
SELECT max(A.departure)
FROM Connection AS _FROM, Connection AS _TO
WHERE _FROM.start = ”Aarhus”
AND _FROM.end = _TO.start
AND _TO.end = ”Randers”
AND _TO.arrival < 2pm $TODAY
```

7) *Find 2 neighbouring seats (if possible) all the way along the following route: Aarhus-Vejle-Kolding anytime today provided the waiting time in Vejle is less than 1 hour.*

```
SELECT *
FROM Connections as A, Connections AS B
WHERE
      A.start = ”Århus” AND
      A.slut = B.start = ”Vejle” AND
      B.slut = ”Kolding” AND
      (B.departure - A.arrival) < 1 hour
      .
      .
      .
      .
      _
```

Nummer 4 her kunne jeg simpelthen ikke få til at lykkes. Men det jeg gerne ville var, at finde alle sæderne fra i alle vognene fra Århus til Vejle. Og dernæst joine med relationen NextToSeat for at få de sæder som er naboer dertil. Dernæst ville jeg finde ud af hvilke, hvis nogen, der var optagede, og vælge dem fra.

Nye Queries

1) *Give the total number of tickets sold for each destination reachable from Vejle during the next 7 days. Sort them in decreasing order of price and include only destinations that are not fully booked.*

```
SELECT Reservations.start_Name,COUNT(*) AS n
FROM Connections
INNER JOIN Reservations
USING (wagon_id, departure, arrival, start_Name, end_Name)
WHERE Reservations.start_Name = 'Vejle'
AND n < 4
GROUP BY start_Name
```

Give a list of stations with their full names, their total number of arrivals, and total number of departures during next week. Order them in lexicographical order.

```
SELECT DISTINCT start_name AS
station,
   (SELECT COUNT(*) AS number_of_departures
   FROM `Connections`
   WHERE start_name = station) AS
   starts_from,
       (SELECT COUNT(*) AS number_of_arrivals
       FROM `Connections`
       WHERE end_name = station) AS
       ends_at
FROM Connections
ORDER BY station
```

Endeligt produkt

Ændringer i det implementerede

Jeg havde en del problemer med at følge mit design fra E/R diagrammet. Det betød at jeg måtte lave nogle ændringer hist og her. Blandt andet så har jeg i mit E/R diagram, relationships for hvilke jeg har oprettet skemaer, og senere **`CREATE TABLE`** statements. Men i DRB.java bruger jeg slet ikke disse relationer.

I det hele taget var det meget problematisk, at skulle holde sig til den teoretiske lære om databaser, når man kunne se at en anden og mere ukonventionel tilgang ville være hurtigere men fungere ligeså fint, i hvert fald mht. vores relativt begrænsede system. Men der måtte man så bide i skeen.

Bekostninger

Og det var først i selve implementeringsdelen at det gik op for mig, hvilke hjørner der skulle skæres, for at det hele kunne gå op i en højere enhed. Det blev bl.a. på bekostning af en ordentlig sti-finder algoritme. For til at finde en optimal rute mellem de forskellige stationer, havde jeg tænkt mig at lave en form for implementation af Dijkstras SSSP algoritme, men det endte, pimært pga. tidspres, til at blive den bedst fungerende løsning som jeg kunne få til at virke.

Næste gang

Nu beder opgaven mig om at reflekterer over hvad jeg ville gøre anderledes, hvis eller når denne opgave kunne blive stillet igen. Og her må jeg sige, at jeg i alt for høj grad, denne gang, har fokuseret på den forestående opgave i de enkelte uger.

Her synes jeg det ville være en langt større fordel, tidligt at overveje hvordan man rent praktisk ville implementerer de forskellige dele i opgaven. Fordi det at vi sidder og arbejder med noget teori i 3-4 uger, hvor efter vi får en aflevering til at få det hele implementeret, det gør at det bliver en smule presset og ustruktureret.

Transactions

For at sikre mig at mit program er serialized, altså at alle tråde i udførslen overholder concurrency, samtidighed i udførsel, så låser jeg de funktioner og metoder, som ændrer på tilstanden, ind i afsnit som gør at den eller de transaktioner som foregår i mellem disse låse, den først bliver udført når jeg er sikker på at den kan udføre hele koden.

Det ser ud som følger:

```
con.setAutoCommit(false);
con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

Herefter er alle funktioner og metoder som befinder sig herimellem sikret alt fra fejl i input data, til udefrakommende hændelser som strømafbrud. Det gør jeg ved, at alt SQL som bliver udført herimellem, som tidligere nævnt, det først bliver udført, når jeg kalder `commit()` på min connection.

```
con.commit();
con.setAutoCommit(true);
```

Som jeg også nævnte, så vedrører dette kun transaktioner som har en ændrende virkning på mine data. En såkalt mutable funktion eller metode. Det har jo ingen konsekvens hvorvidt jeg læser hvor mange pladser der er ledige i toget fra Kolding til Århus i eftermiddag, og den transaktion ikke bliver fundstændigt gennemført. Det eneste der sker, er at man får noget korrupt data, hvilket mit UI så fortæller. Men data i databasen bliver jo kun læst, og intet er rørt ved.

Isolationlevel

Som det kan læses ovenover, så viser `TRANSACTION_READ_COMMITTED` mit valg af isolationlevel. Det forhindrer at det bliver foretaget dirty reads, som er læsning af data som endnu ikke er fuldt comittet. Mht. serializabillity så har jeg ikke behov for at specificere `SERIALIZABLE` da det er default i SQL.

Assertations, Triggers og References

Som man kan læse af mine skema-deklarationer i afsnit 5, så var det planen at konstruere mine assertations i form af **REFERENCES** og **CHECKS** for at opretholde integritet og konsistens, og i form af **TRIGGERS**, og i et andet DBMS ville dette også klart have været at foretrække, da især **TRIGGERS** letter modifikationer i databasen overordentligt meget.

Men MySQL understøtter efter min bedste overbevisning ikke **REFERENCES**. Og MySQL version 4 har heller ikke **TRIGGERS**, det er først kommet med fra version 5.

Jeg har angivet begge disse, i mine skemaer, men det havde ingen indflydelse på datastrukturen. Og der var i MySQL ingen steder de kunne angives manuelt. Så, efter at have hørt det samme fra andre grupper, gik jeg fra det igen. Jeg rådførte mig derefter med Martin Geisler, og fik besked på at jeg så bare skulle angive hvorledes jeg ville have gjort det.

Og brugen af **REFERENCES** kan man, som nævnt se ud fra mine skemaer som er vist i afsnit 5. **TRIGGERS** kunne have været brugt adskillige steder i mit program, som f.eks. i billetbestillingen, hvor jeg foretager flere transaktioner i samme metode. **CHECKS** har jeg lavet i form af kontrol med datatypen på input, igennem javas logiske operatorer og if-betingelser.

Transaktioner er selvfølgeligt isolerede, men som i stedet for at være afledt af nye database connections kald, godt kunne have været instansieret af **TRIGGERS**. Eksembelvis bruger jeg i billetbestillingen under menupunkt 1.3 hele seks instansierede **PreparedStatement** kald. Disse kunne uden tvivl have været reduceret med **TRIGGERS**.

Brugere og rettigheder

Af ovenstående grunde kan jeg ikke oplyse privilegierne for brug af **REFERENCES** og **TRIGGERS**.

Rettigheder for **SELECT**, **INSERT** og **UPDATE** for de givne grupper er listet herunder:

DRB Administrator

Administratoren er den højeste autorietet i systemet, og har derfor tilladelse til at modificere alle relationerne. Det har administratoren af den grund, at vedkommende skal kunne debugge og teste systemet, ligesom vedkommende skal kunne servicere de øvrige medarbejde i DRB.

De ville altså have **SELECT**, **INSERT** såvel som **UPDATE** for flg. relationer:

```
Connections,      RunsOn,
E_Tickets,        Seats,
NextToSeat,       Spans,
Passengers,       Stations,
Reservations,     Wagons
```

DRB Medarbejdere

Medarbejdere har derimod kun rettigheder til at udføre **`INSERT`** og **`UPDATE`** i nedenstående:

`E_Tickets, Passengers, Reservations`

Herudover har de rettigheder til at kunne udføre **`SELECT`** på alle relationerne.

Passengere

Passagerene har den laveste autoritet, og har kun rettighed til at kunne bestille billetter, og oprette deres egen kundedata(markeret som `**` omkring relationerne) i Passengers-relationen. Så derfor har de de rettigheder til de samme relationer som medarbejderene har, men de har udelukkende rettighed til at modificere tupler, som indeholder deres unikke `id` attribut.

`*E_Tickets*, *Passengers*, *Reservations*`

-- Søren Løbner