**Attributter Anvendelse**

brugerId brugerens eget valg af identifikation. Skal være unik.

createTs tidspunkt for, hvornår rækken er oprettet eller ændret

password brugerens password i hashed udgave

salt det salt, der skal gøre hash'en forskellig fra andre hashes af samme password

email email-adresse

prim markering af den foretrukne email-adresser

erAdmin om brugeren har administrative rettigheder

lastLogon hvornår har brugeren sidste logget sig på systemet

logonCnt hvor mange gange har brugeren været på

arrangementBeskrivelse en kort forklaring på, hvad arrangement går ud på

arrangementWeb link til web-siden, hvis en sådan findes

arrangementDt hvilken dato afholdes arrangementet

arrangementTid hvad tid på dagen indledes arrangementet

erTilmeldt om bruger er tilmeldt arrangement

spiseSted hvis der er spisning inkluderet, så navn og adresse på spisested

spiseTid tidspunkt for spisning under antagelse af, at det er samme dato som arrangementet

spiser om bruger deltager i spisning

billedAdresse path/filename.extention for uploadede billeder fra arrangementet

kommentar en optionel tekst om billedet

Password og salt gemmes i et antal versioner for at brugeren ikke skal kunne genanvende et password før der er gået et antal generationer.

Da email-adresser er universelt unikke, vil det være en oplagt mulighed at lade adressen indgå som en del af nøglen, men det er valgt, at email-adressen skal være optionel, og det skal være muligt at ændre den.

I nogle RDBMS'er eksisterer en auto-increment værdi, så hver gang der indsættes en række tælles op. Microsoft SQLserver (https://msdn.microsoft.com/en-us/library/ms182776.aspx) har en værdi, der tager udgangspunkt i timestamp, men er garanteret unik. Disse er naturlige valg til identifikation af en række i en tabel.

Hvis ikke der er en datatype med automatisk optælling, kan man konstruere sig frem til en sådan, under forudsætning af, at operationen kan køre atomisk. Altså at der ikke er flere, der kan indsætte eller opdatere samtidigt på samme tabel. Konstruktionen vil se ud som følgende:

Der er en kolonne med navn serialNumber, der indeholder et stigende rækkenummer for tabellen.

Insert into tabelNavn (serialNumber, ....) values((select max(serialNumber)+1, ...);

I en miniature-applikation som nærværende vil det ikke være noget problem at have en lås på hele tabellen, så kun en af gangen kan opdatere i den. I lidt mere anvendte applikationer, vil det normalt ikke være acceptabelt, at låse hele tabellen. Her vil man skulle nøjes med en rækkelås, eller lås den page, der er underopdatering, og så vil ovenstående SQL ikke kunnet garantere en unik værdi.

MySQL har timestampværdier med et-sekunds opløsning som standard. Der er så alligevel mulighed for at oprette op til seks cifre efter som en brøkdel af sekunder; altså en opløsning i mikrosekunder.

https://dev.mysql.com/doc/refman/5.7/en/fractional-seconds.html

Andre RDBMS'er anvender sekunder, millisekunder eller mikrosekunder. Nogle gange er også 100 nanosekunder set (https://www.w3schools.com/sql/sql\_datatypes.asp). I denne webapplikation, er alle indsætninger og opdateringer på tabellerne initieret manuelt. Det vil derfor ikke være forventeligt, at en bruger foretager to ændringer inden for et sekund. BrugerId er unikt og oprettelsestidspunktet (createTs) vil være unikt pr. bruger, når det nu antages, at brugeren ikke kan foretage sig flere handlinger inden timestamp er ændret. Det vil derfor være en mulighed at anvende brugerId og createTs som sammensat nøgle for alle data.

Hvis man forsøger at indsætte to rækker i en tabel, med samme nøgle, opstår der en constraint violation; man kan ikke anvende en given nøgle mere end en gang. Det skal så i givet fald en fejltekst i webbrowseren, så brugeren kan se, at der skal forsøges en gang mere. En mere brugervenlig udgave vil selvfølgeligt være at applikationen i første omgang selv håndterer situationen og f.eks. læser current timestamp nogle gange, og, når timestamp har ændret værdi, indsætter/opdaterer en gang mere.

Attributterne har hver netop en type. Der er ikke nogle af felterne, der skal opdeles, så de er atomiske. Det samme som, at der ikke er nogle repeterende felter i ovenstående, og ved anvendelse af brugerId og createTs som sammensat nøgle, vil alle kolonnerne i en tabel være på 1NF. Det ses dog hurtigt, at der ikke er fuld funktionel afhængighed for alle øvrige kolonner. F.eks. er arrangementTid en attribut, der fortæller om selve arrangementet, men ikke fortæller noget om personen, der har oprettet arrangementet eller har tilmeld sig. Dermed er der ikke en funktionelt afhængighed til nøglen, hvilket er et krav på 2NF.

Generelt kan det siges, at 1NF kræver atomiske værdier og ikke-repeterende grupper. Desuden skal der være en eller flere attributter, der tilsammen entydigt kan identificere en tuple.

2NF kræver, at alle attributter, der ikke indgår i nøglen, skal være fuld funktionelt afhængige af nøglen.

3NF siger, at der ikke må være nogen transitive afhængigheder.

4NF kræver, at der er max en flerværdiet afhængighed i hver relation.

Hvis der er en relation med bilmodeller, motorstørrelser, farver og priser, og vi antager, at der udelukkende er metalfarver. R(model, motor, farve, pris). Det viser sig, at pris ikke varierer med farven, men kun model og motor. For at komme på 2NF skal R splittes i R1(model, motor, farve) og R2(model, motor, pris).

3NF kan illustreres med R(navn, adresse, postnummer, by), hvor by er transitivt afhængig af postnummer. Derfor splittes R til R1(navn, adresse, postnummer) og R2(postnummer, by)

Til 4NF kan vi se på bil-relationerne igen. Der er flere modeller, flere motorer og flere farver. Motor og farve har ingen indbyrdes afhængighed, så derfor bliver R1(model, motor, farve) opdelt i R1a(model, motor) og R1b(model, farve).

Til praktisk anvendelse, er 3NF normalt helt fin; dog finder 4NF også nogle gange god anvendelse, specielt hvis der er tale om større datamængder. Yderligere normalisering er vist mest en teoretisk øvelse.

Ovenstående diskussion udmønter sig i at der skal laves fem tabeller, og at der skal tilføjes et entydigt arrangementId, som vil blive automatisk optalt nummer eller, hvis det ikke findes i pågældende RDBMS’s, current timestamp.

|  |  |  |
| --- | --- | --- |
| **Logon** |  |  |
| brugerId | lastLogon | logonCnt |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **User** |  |  |  |  |  |  |
| brugerId | createTs | password | salt | email | prim | erAdmin |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Arrangement** |  |  |  |  |  |
| arrangementId | arrangementBeskrivelse | createTs | arrangementWeb | arrangementDt | arrangementTid |

|  |  |
| --- | --- |
|  |  |
| spiseSted | spiseTid |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tilmeldinger** |  |  |  |  |
| arrangementId | brugerId | createTs | erTilmeldt | spiser |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Billeder** |  |  |  |  |
| billedAdresse | arrangementId | brugerId | createTs | kommentar |

Ovenstående konstruktion kan også give nogle ganske ”krøllede” SQL’er. Hvis man f.eks. skal finde ud af, hvem der deltager i spisning til et givet arrangement, skal man finde den nyeste række for et givet arrangement pr. bruger. Der er mange dialekter af SQL. En af dem er DB2 for zOS og her ville forespørgslen kunnet formuleres som følgende:

select brugerId

from tilmeldinger

where (arrangementId, brugerId, createTs) in (

select arrangementId, brugerId, max(createTs)

from tilmeldinger

where arrangementId = :hvadErDetForEtArrangementDerSpørgesPå

group by brugerId

)

and spiser = ’True’

Det er dog ikke alle SQL dialekter, der tillader sådan en konstruktion. Som en omgåelse kunne man konkatenere arrangementId, brugerId og createTs til en tekststreng både i where-clausen og i subselecten. Hvis det viser sig, at der ikke er en enkelt konstruktion, der kan løse opgaven, kan man oprette en temporær tabel og anvende den i et select-statement.

create table sel (arrangementId char(…), brugerId char(…), createTs Timestamp);

insert into sel

select arrangementId, brugerId, max(createTs)

from tilmeldinger

where arrangementId = :hvadErDetForEtArrangementDerSpørgesPå

group by brugerId

;

select brugerId

from tilmelding t, sel s

where t.arrangementId = s.arrangementId

and t. brugerId = s.brugerId

and t.createTs = s.createTs

and t.spiser = ’True’

;

drop table sel;

Det bliver til en længere historie; dog har flere SQL varianter andre muligheder, der gør forespørgslen mindre klodset. Den dyreste mulighed er at tage resultatet af subselecten ind i et array i programmet, og så derefter slå op i tabellen tilmelding med hver enkelt række i arrayet. Der bliver meget transport og initiering af forespørgsel i databaseserveren.

Man kan argumentere for en yderligere normalisering, da man kan risikere redundans. F.eks. er User-tabellen ikke i 4NF. Det medfører, at en ændring i f.eks. email-adressen, giver en ekstra række i tabellen, men de samme data som i den nyeste række, bortset altså fra email-adressen.

<https://www.ischool.utexas.edu/~wyllys/DMPAMaterials/normstep.html> 🡨 brug denne til ref

(<http://www.stjerne.biz/wp-content/uploads/2011/11/Database-design.pdf>)

Opsplitningen vil give tre tabeller, hvor password og salt hører sammen i den ene tabel, email-adressen hhv. erAdmin kommer i to andre tabeller. Hvis ikke der var interesse for historikken på Admin-tabellen, så ville createTs være overflødigt. Dermed ville der være brugerId som nøgle. Da erAdmin er en boolsk værdi, ville det ikke være nødvendigt at have den som selvstændig attribut, og derfor ville Admin-tabellen kun bestå af brugerId.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **UserPassword** | |  |  |  | **Email** |  |  |  | **Admin** |  |  |
| brugerId | createTs | password | salt |  | brugerId | createTs | email |  | brugerId | createTs | erAdmin |

Af praktiske grunde accepteres redundans, og normalisering ud over 3NF (og evt. Boyce Codd) vil ikke blive gennemført.

Auto-increment konstruktionen blev fundet på mysql.com. Den varierer meget mellem de forskellige database systemer. Samme sted blev også fundet en boolean datatype, men den gav syntaxfejl ved import i MySQL-Workbench; derfor blev datatypen ændret til char(1), der så f.eks. kan antage værdierne ’T’ og ’F’. Det vil blive styret i programmet, der skal læse og skrive på tabellerne, ved hjælp af to konstanter. I C# bliver det f.eks. til:

public const char booleanTrue=’T’;

public const char booleanFalse=’F’;

https://dev.mysql.com/doc/refman/5.7/en/numeric-type-overview.html

Herunder er så en MySQL flavoured DDL for oprettelse af en database med de fem tabeller, og et EER-diagram importeret i MySQL-Workbench.

create database VoresJazzklub;

use VoresJazzklub;

create table Logon(brugerId varchar(255) not null, lastlogon timestamp not null, logonCnt integer not null, primary key (brugerId));

create table Users(brugerId varchar(255) not null, createTs timestamp(6) not null, password char(128) not null, salt char(128) not null, email varchar(255) not null, prim char(1) not null, erAdmin char(1) not null, constraint pk\_user primary key (brugerId, createTs));

create table Arrangement(arrangementId BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT UNIQUE, arrangementBeskrivelse varchar(255) not null, createTs timestamp(6) not null, arrangementWeb varchar(255) not null, arrangementDt date not null, arrangemnetTid time not null, spiseSted varchar(255) not null, spiseTid time not null, primary key(arrangementId));

create table Tilmeldinger(arrangementId BIGINT not null, brugerId varchar(255) not null, createTs timestamp(6) not null, erTilmeldt char(1) not null, spiser char(1) not null, constraint pk\_tilmeldinger primary key(arrangementId, brugerId, createTs));

create table Billeder(billedAdresse varchar(255) not null, arrangementId BIGINT not null, brugerId varchar(255) not null, createTs timestamp(6) not null, kommentar varchar(255) not null, primary key(billedAdresse));

