Databaseprogrammering

- "begrænsninger" og triggere
  – defineres som del af "skema" og vedligeholdes af DBMS

- Transaktioner og procedure'
  – programmeres i (f.eks.) Embedded SQL, PL/SQL
  – flerbrugeraspektet; `COMMIT`, `ROLLBACK`, m.v.

F.eks. opdateringsrutine

At opdatere database for at afspejle begivenhed W i verden:

> Udføre proceduren:
> gør X, gør Y og hvis Z gør *Æ* ellers gør *Ø*

Hjulpet af automatisk trigger:

> Hmmm, når *Ø*, så må jeg gøre *Å*!

Eksempler på begrænsninger i SQL: Nøgler

Nøgle? Et eller flere attributter, som identificerer tupel.

Flere alternative måder: Eksempler

```
CREATE TABLE Moviestar (
  name CHAR(30) PRIMARY KEY,
  address VARCHAR(256) UNIQUE
  gender CHAR(1),
  birthdate DATE
);
```

Der genereres automatisk indeks på `UNIQUE` og `PRIMARY KEY`.
Forskellen: `PRIMARY` kan også bruges andetsteds som fremmednøgle (senere).

Beslægtet begrænsning i SQL

```
CREATE TABLE ... (
  ...
  gender CHAR(1) NOT NULL,
  ...
);
```

`NOT NULL` underforstået ved `UNIQUE` og `PRIMARY KEY`.

Hvordan gennemtvinges sådanne begrænsninger?

Forsøg på opdatering, som strider mod begrænsninger **afvises** af DBMS (fejlmeddelelse).
– enkelt at håndtere fordi kun én relation
– reduceres til problem ved én tupel!

Referentiel integritet og fremmednøgler

Eksempel:

```
CREATE TABLE MovieExec (
   ...
   cert# INT PRIMARY KEY,
   ...
);

CREATE TABLE Studio (
  namechar(30) PRIMARY KEY,
  address VARCHAR(255),
  presC# INT REFERENCES MovieExec(cert#)
  );
```

Hvordan gennemtvinges sådanne begrænsninger? – involverer to relationer

Default: **Afvis** den ændring, som skaber problem, f.eks.

- Indsættelse af `Studio`-tupel med `presC#` = "ukendt" `cert#`
- Sletning af `MovieExec`-tupel hvis `cert#` er i brug.
- "`UPDATE`" med tilsvarende konsekvenser.

Alternativt "CASCADE" eller "SET NULL"

Eksempel

```
CREATE TABLE MovieExec (
   ...
   cert# INT PRIMARY KEY,
   ...
);

CREATE TABLE Studio (
  name CHAR(30) PRIMARY KEY,
  address VARCHAR(255),
  presC# INT REFERENCES MovieExec(cert#)
     ON DELETE SET NULL
     ON UPDATE CASCADE
  );
```

Begrænsning gennemtvinges ved

1. Accepter ændring på `MovieExec` hvis overhovedet muligt.

2. Ret tupler i `Studio` så pengene kommer til at passe!

F.eks.

- Slettes en `MovieExec`-tupel, `NULL`-stilles `cert#`-attributter for alle berørte `Studio`-tupler.

- Ændres en given `MovieExec`-tupel vha. "`UPDATE`" så `cert#` går fra c_1 til c_2, så ændres alle `Studio`-tupler som bruger c_1 tilsvarende.

Hvad betyder:

`ON DELETE CASCADE`

Interesserede kan prøve efter i ORACLE om

`CASCADE` virker rekursivt, og hvad der sker ved cirkularitet?

Andre "simple constraints"

Eksempelvis "`CHECK`":

```
CREATE TABLE Moviestar (
  name CHAR(30),
  address VARCHAR(256),
  gender CHAR(1) CHECK (gender IN ('F', 'M')),
  birthdate DATE,
  salary INT CHECK (salary >1000000),
  ...
  strange-attr INT CHECK (strange-attr IN (SELECT ... FROM Rel WHERE ...))
);
```

Afviser de ændringer til `Moviestar`, som ville skabe problemer, **men opdager ikke** hvis `Rel` ændres uheldigt!

Tupel-constraints

... som kan referere til flere attributter i tupel, f.eks.

```
CREATE TABLE Moviestar (
  name CHAR(30),
  address VARCHAR(256),
  gender CHAR(1) CHECK (gender IN ('F', 'M')),
  birthdate DATE,
  salary INT CHECK (salary >1000000),
  CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

Semantik som ovenfor.

Den ultimative "constraint": `ASSERTION`

Kan udtrykke enhver begrænsning, vi kan udtrykke i relationel algebra.

```
CREATE ASSERTION PresidentsAreRich CHECK
    (NOT EXISTS
         (SELECT *
          FROM Studio, MovieExec
          WHERE presC# = cert# AND netWorth < 10000000
         )
    );
```

- Checkes hver gang en indgående relation ændrer sig
  – her: `Studio` og `MovieExec`
- Afviser enhver opdatering, som ville give problemer;
  dvs. **ingen mulighed for automatisk reparation** a la `CASCADE`
- Er altid overholdt (modsat de forrige "`CHECK`s).
- (Potentielt) meget dyr at evaluere
  – her: det uskyldige komma mellem `Studio` og `MovieExec`!!!

Bør bruges med omtanke — nyttig til testformål og eksperimenter.

En pragmatisk efterligning + udvidelse: Triggers...

Triggers

- Startes ved begivenhed+betingelser *angivet af programmøren*,
- Kan udføre vilkårlige database-modifikationer
  – relationen involveret i begivenheden + alle andre relationer
- Problem med uendelige løkker:
  Begivenhed → Trigger1 → Trigger2 → Trigger1 → ...
  F.eks. Oracle forbyder triggere med "gensidige ændringer"

Eksempel fra bogen i SQL3 syntaks:

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON movieExec
REFERENCING
    OLD AS OldTuple,
    NEW AS NewTuple
WHEN(OldTuple.netWorth > NewTuple.netWorth)
    UPDATE MovieExec
    SET netWorth = OldTuple.netWorth
    WHERE cert# = NewTuplw.cert#
FOR EACH ROW
```

NB: Oracle PL/SQL's syntaks en smule anderledes (senere)
Betydning:

- Begivenhed: `UPDATE OF netWorth ON movieExec`
  Alternativer: `INSERT ON`, `DELETE ON`
- Betingelse: `WHEN(OldTuple.netWorth > NewTuple.netWorth)`
- Tidspunkt for handlingen relativt til begivenheden: `AFTER`
  Alternativer: `BEFORE`, `INSTEAD OF`
- Den udløsende begivenheds omfang: `FOR EACH ROW`, "row-level trigger"
  Alternativt: (ingenting), "statement-level trigger"
  – her kan refereres `OLD_TABLE AS` ..., `NEW_TABLE AS` ...

Overvej: Hvad synes vi om disse triggers som sprogdesign??

Transaktioner, procedurer, applikationer

... programmeres f.eks. i

- SQL
- PL/SQL – Procedurel udvidelse af SQL
- Embedded SQL — indlejret i C vha. præcompiler
- JDBC — Java bibiliotek som "interfacer" til Oracle
- eller lidt af hvert af det hele ...

RDBMS, f.eks. Oracle, gør hvad det kan for at give hver "klient-proces" indtryk af at være alene i verden.

Dvs. programmøren behøver kun i ringe grad skrive kode, som synkroniserer forskellige processer.

RDBMS stiller "højniveauværktøj" så som `COMMIT` og `ROLLBACK` til rådighed.

Eksempel i Embedded SQL

```
void getStudio() {
    EXEC SQL BEGIN DECLARE SECTION;
        char studioname[15], studioAddr[50];
        char SQLSTATE[6];
    EXEC SQL END DECLARE SECTION:
    /* print request that studio name and address be
       entered and read response into variables
       studioName and studioAddr */
    EXEC SQL INSERT INTO Studio(name, address)
       VALUES(:studioName, :studioAddr)
}
```

Syntaks for delte variable med ":" inde i SQL-udtryk.

NB: `SQLSTATE` er en særlig variabel som fortæller om foregående operation gik godt (jvf. `CHECK`, `TRIGGER` osv.).
Mon JDBC bruger exceptions? [Ved det ikke – oplagt mulighed].

Introduktion til sproget PL/SQL

Integreret del af ORACLE:
Udvider SQL med blokke, procedurer, control-flow, variable, exceptions osv.
Kan kaldes direkte via SQL-klient eller aktiveres fra applikationsprogram.

Blokke:

```
DECLARE
/* Declarative section: variables, types, and local subprograms. */
BEGIN
/* Executable section: procedural and SQL statements go here. */
EXCEPTION
/* Exception handling section: error handling statements go here. */
END;
```

Eller helt ned til:

```
BEGIN
/* Executable section: procedural and SQL statements go here. */
END;
```

Modsat Java har PL/SQL blokke i blokke i blokke ...

I kroppen: Ingen DDL-sætninger, kun DML m. udvidelser.

Eksempler på kontrolstrukturer:

```
DECLARE
    a NUMBER;
    b NUMBER;
BEGIN
    SELECT e,f INTO a,b FROM T1 WHERE e>1;
    IF b=1 THEN
        INSERT INTO T1 VALUES(b,a);
    ELSE
        INSERT INTO T1 VALUES(b+10,a+10);
    END IF;
END;
```

Bemærk `SELECT ... INTO`; forudsætter `SELECT` giver netop én tupel.
Hvis ikke, hvad så? En exception? PRØV!

```
DECLARE
    i NUMBER := 1;
BEGIN
    LOOP
        INSERT INTO T1 VALUES(i,i);
        i := i+1;
        EXIT WHEN i>100;
    END LOOP;
END;
```

Andre løkker:

```
WHILE <condition> LOOP
   <loop_body>
END LOOP;
```

```
FOR <var> IN <start>..<finish> LOOP
    <loop_body>
END LOOP;
```

Særligt om typer for PL/SQL-variable

```
DECLARE
    a NUMBER;
    b VARCHAR(7);
    c T1.f%TYPE;
    d T1%ROWTYPE;
```

OBS: Ingen type “Relation”.
I stedet “cursor” som muliggør “algoritmisk” behandling af tuplerne i en relation.

Eksempel:

```
DECLARE
    a T1.e%TYPE;
    b T1.f%TYPE;

    CURSOR T1Cursor IS
        SELECT e, f FROM T1 WHERE e < f
        FOR UPDATE;
BEGIN
    OPEN T1Cursor;
    LOOP
        FETCH T1Cursor INTO a, b;
        EXIT WHEN T1Cursor%NOTFOUND;
        DELETE FROM T1 WHERE CURRENT OF T1Cursor;
        INSERT INTO T1 VALUES(b, a);
    END LOOP;
    CLOSE T1Cursor;
END;
```

Procedurer i PL/SQL

```
CREATE [OR REPLACE] PROCEDURE <name> ( <parameters> ) AS
<local_var_declarations>
BEGIN
    <procedure_body>
END;
```

Parametre specifiseres som `IN` (default?), `OUT` eller `INOUT`;
typer stort set som for PL/SQL-variable, dog `VARCHAR` i stedet for `VARCHAR(7)`.

Eksempel:

```
CREATE TABLE T3 (
    a INTEGER,
    b INTEGER
);
...
CREATE PROCEDURE addtuple3(a NUMBER, b OUT NUMBER)
AS
BEGIN
    b := 4;
    INSERT INTO T3 VALUES(a, b);
END;
...
DECLARE
    v NUMBER;
BEGIN
    addtuple3(10, v);
    ... v ...
END;
```

Variation: funktioner

```
CREATE  [OR REPLACE] FUNCTION <func_name>(<param_list>) RETURN <return_type>
AS ...
```

Triggers i PL/SQL

Forenklet syntaksbeskrivelse:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
    {BEFORE | AFTER | INSTEAD OF}
           {INSERT | DELETE | UPDATE} ON <table_name>
    [FOR EACH ROW [WHEN (<trigger_condition>)]]
    <trigger_body>
```

Eksempel:

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);

CREATE TRIGGER trig1
    AFTER INSERT ON T4
    FOR EACH ROW
    WHEN (NEW.a <= 10)
    BEGIN
        INSERT INTO T5 VALUES(:NEW.b, :NEW.a);
    END trig1;
```

Citat: “The special variables NEW and OLD are available to refer to new and old tuples respectively. Note: In the trigger body, NEW and OLD must be preceded by a colon (”:”), but in the WHEN clause, they do not have a preceding colon!”

Transaktioner og transaktionssstyring

Nødvendigt at tænke på fordi:

- Der er typisk mange brugere, som læser og skriver i de samme tabeller samtidigt.
- Data er tidskrævende at reetable, så robusthed overfor nedbrud er påkrævet.

Idé: Så vidt muligt sørger RDBMS for at give bruger (klient-proces) indtrykket af at være alene om databasen:

Hver bruger kører på sin "virtuelle kopi" af databasen.
Ændringer kræver `COMMIT` for at andre brugere kan se dem.

Hurtigt om transaktioner baseret på OH'er fra Troels Andreasen.

Sikkerhed, beskyttelse og eksplicitte privilegier til brugere til at foretage sig (eller ikke foretage sig) dette og hint.

Læs selv Ullman+Widon, kap 7.3+7.4.