

Om foredragsholderen

Svenne Krap
svenne@kracon.dk

- Cand.Merc
- Selvstændig programmør, DBA og SysAdmin
- Arbejdet med databaser i ti år (Mysql, DB2, Oracle, SQL Server og PostgreSQL)
- Arbejdet med PostgreSQL siden 1999

På dette foredrag

- Databaser generelt (7 slides)
- PostgreSQL (12 slides)
- Første gang (11 slides)
- Hands on (24 slides + live session)
- Afslutning

Databaser generelt

Hvad er en RDBS?

En database (RDBS) er konceptuelt set en black-box storage engine, der styres via et kunstigt sprog (SQL).

Databaser har to typer kommandoer, administrationskommandoer (DDL) og brugskommandoer (DML).

De typiske (DML) er *select*, *insert*, *update* og *delete*.

Hvad er en RDBS (2)

Et antal homogene data (kaldet rækker) har et antal målepunkter (kaldet felter) og samles i en tabel, et antal tabeller samles i en database.

Først defineres datas “udseende” med *create table (DDL)*, hvorefter de kan behandles med DML-kommandoer, fx. indsættes med *insert* .

Eksisterende data kan opdateres med *update* eller slettes med *delete*.

Endeligt kan data forespørges med *select*.

Hvad er en RDBS (3)

Det er vigtigt at holde sig for øje, at SQL tænker i mængder, hvilket betyder at en forespørgsel sagtens kan være resultatløs men korrekt.

Forestil jer: vælg alle deltagere (i rummet her), der er højere end 40 meter.

Eller: Vælg alle tal som er lige og som er ulige

Der er ikke noget galt med forespørgslerne, men der er (formodentligt) ikke nogen der opfylder kriteret. Dvs databasen ville svare tilbage “ok, ingen fundet.”

Hvad er en RDBS (4)

Husk at selvom data typisk vises i en grid-control, har databaser konceptuelt meget lidt med regneark at gøre!

For databaser er der et koncept af “aktuel række”, og den kan ikke referere til rækker før eller efter sig selv (men til sine egne felter og andre forespørgsler).

Den magiske select

SELECT *felter* FROM *kilder* WHERE *betingelser*
ORDER BY *sortering*

Der sker nu (konceptuelt):

- Saml kilder
- Eliminer rækker der ikke opfylder betingelser
- Kopier / Skab felter til resultat
- Sorter resultat

Hvordan bliver man god til SQL

- 1) Tænk i mængde sprog fx. “giv mig alle dem”, “hvor det (ikke) gælder at” osv
- 2) Tænk i samme rækkefølge som databasen (kilder, begrænsninger, felter og sortering)
- 3) Tænk på at der typisk er mere end en måde at udtrykke det samme på. (også med hensyn til hastighed)
- 4) Divide and conquer, prøv ikke at løse hele “problemet” (forespørgslen) i et forsøg, men tag den bid for bid – alle databaser har gode interaktive værktøjer!
- 5) Lær og brug profiling værktøjerne (EXPLAN/EXPLAIN ANALYZE i PostgreSQL)
- 6) Lær og brug så mange features som muligt i SQL (der er en årsag til de findes!)
- 7) “Undgå” gui-værktøjerne – *Easy is the worst enemy of skilled*
- 8) Øvelse gør mester

Termer brugt i dette foredrag

Nogle database termer har forskellig mening ved forskellige leverandører, i dette foredrag betyder:

Server = fysisk maskine

Instans = een kørende udgave på en server lyttende til en given port på en given IP-adresse (flere instanser kan køre på samme server)

Database = en samling af data kørende i een instans.

Flere databaser kan køre simultant i en instans.

Schema = namespaces i en given database

PostgreSQL

PostgreSQL historie

- 1977 Berkeley: Ingres (nu Computer Ass.)
- 1986 Berkeley: “post Ingres” (Postgres)
- Kommercielt Illustra -> Informix (nu, IBM)
- 1994: RIP Postgres
- 1995 Berkeley: SQL engine -> Postgres95
- 1996 Udvikling uden for Berkeley
- PostgreSQL 6.0 (1996)
 - Idag, PostgreSQL 8.2.4 (8.3 i feature freeze)

PostgreSQL filosofi

- Datasikkerhed (ACID)
 - Atomicity (transactions)
 - Consistency (foreign keys)
 - Isolation (transactions)
 - Durability
- Korrekthed
 - Count(*)
- Stabilitet
- Hastighed

PostgreSQL facts 'n' features

- BSD Licens (og kommercielle søskende)
- “Alle” POSIX platforme + Win32
- Det meste af SQL92 og SQL99
- MVCC (multi version concurrency control)
 - VACUUM, men ingen undolog
- Native BLOBS og CLOBS (TOAST)
- Interfaces: C, C++, .NET, Perl, Python, Ruby, Tcl, JDBC, ODBC osv.
- PLs: Java, Perl, Python, Ruby, Tcl, C/C++, PL/PgSQL, R
- Unicode og Locale*
- Max
 - Tablesize: 32TB
 - Rowsize: 1,6TB
 - Fieldsize: 1GB
 - Columns: 250-1600

PostgreSQL facts 'n' features (2)

- SQL 92/99 support:
- Subqueries (også i from-klausuler)
- read committed & serializable transaction levels
- Information schema
- (multi-column) Pkeys, Fkeys (restrict/cascade), check constraints, unique & not nulls
- Sequences
- LIMIT/OFFSET
- Partial og Functional indexes
- Table inheritance
- Async notification

Fremtiden

8.3 (lige om hjørnet):

- **Tsearch2 (full-text) i core**
- Asynkron commit
- **Updateable cursors**
- XML
- Masser af performance ting

Senere:

- Fuld locale support
- Replikering i core

Politik:

- Major release = 18 mdr., 8.3 = 12ish mdr., 8.4+ ukendt
- Kun nye features i major releases (kræver dump/restore)!

Hvem bruger PostgreSQL?

- Mig :)
- BASF
- .ORG
- Universiteter (Berkeley, Alabama, NSW, Oslo, Sydney, Prag)
- U.N.
- US CDC
- US Dep. of Labor
- Creative Commons
- IMDB
- Macworld
- Debian
- Sourceforge
- Fujitsu
- Sun
- Cisco
- Skype

PostgreSQL vs. andre

- MVCC er pessimistic version control
- Transaction levels kan være strictere end forespurgt
- Ingen planner hints (men sofistikeret statistisk optimizer)
- Aldrig uden for transactions (dog auto trans. wrap)
- PL/pgsql ligner PL/SQL (oracle)
- SQL syntax (MySQL: pas på string concat.)
- Ingen storage engines, ingen memory tables
- Contrib og andet eksternt udviklet
 - Tsearch2
 - PostGIS
 - Slony

PostgreSQL vs. andre (2)

- Ingen cross database selects! (dblink findes dog)
- Meget udvidbar (types, aggregates, languages, operators, conversions, rules)
- Kun to filer (pg_hba.conf, postgresql.conf)
- Alle “catalog data” ligger i basen (pg_...) og kan modificeres via SQL.
- “from-less” selects
- *as* i column alias er ikke optionelt
- Længdeangivelse på varchar ej nødvendig
- Count(*)
- Normalt ingen explicit håndtering af CLOB eller BLOB pga. TOAST

Performance

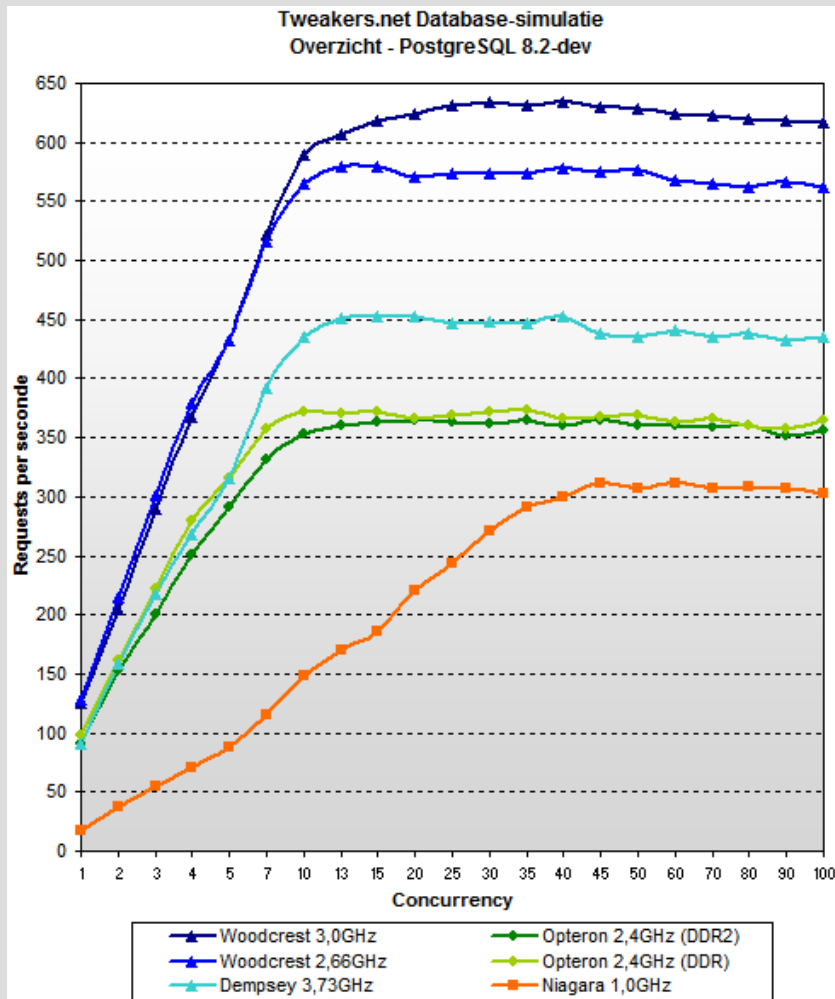
Generelt god, skaleringsproblemer ved store systemer (16+ CPUer, store disk systemer).

Kører single-threaded.. dvs. multi-core/cpu kan kun udnyttes ved tilstrækkeligt antal simultane queries (kommerciel Bizgres MPP)

Ikke så mange knobs and switches i opsætning, overlader så meget caching som muligt til O/S.

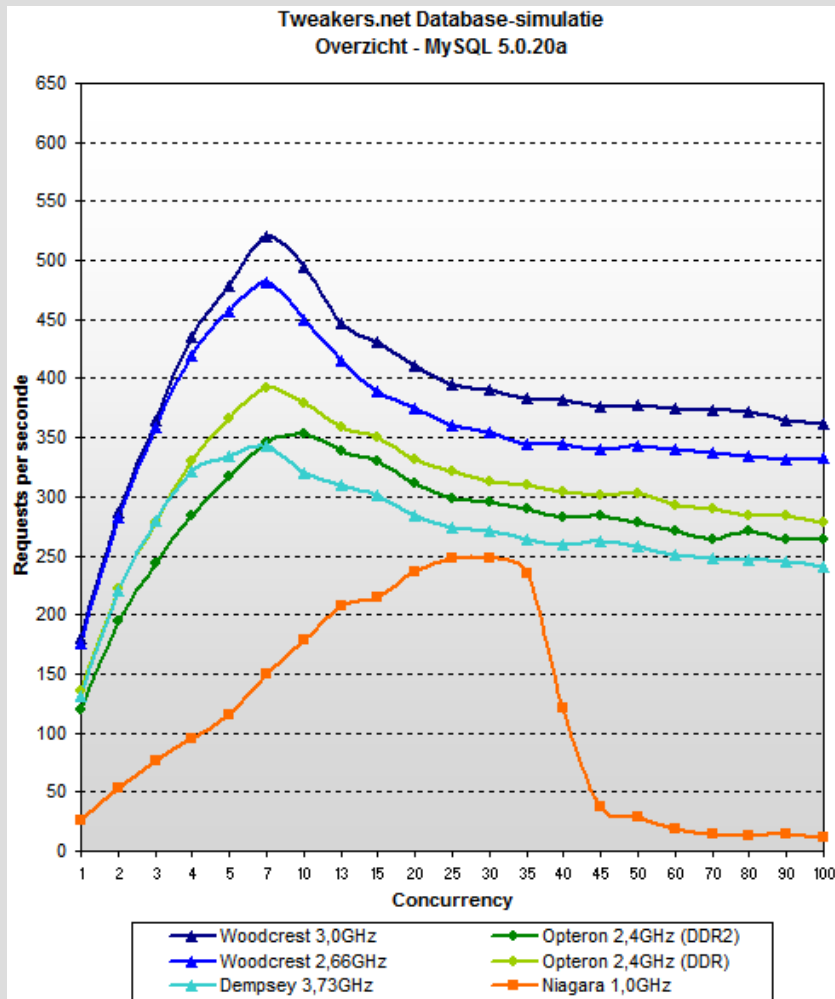
Ingen planner hints

Performance (2)



- For x86 lineær skalering til ca. 10 concurrent queries, herfra flat
- Niagara længere lineær skalering, dog væsentlig lower throughput (single threading)
- Ved concurrency > 10 på woodcrest 3.0 = 600 rps

Performance (3)



- Skalærer lineært optil ca. 5 i concurrency derpå ydelsesfald!
- På intet tidspunkt over 525ish på 3.0 Woodcrest

Performance (4)

Første industry standard benchmark (Jun 2007):

778.14 SPECjAppServer2004 JOPS@Standard

Næste benchmark (Jul 2007):

813.73 SPECjAppServer2004 JOPS@Standard

Oracle på lignende* hardware: 1000ish

Første gang

Installation og upgrade

Installation:

- Installer pakke
- Initdb (eller re-initdb)
- Tun *postgresql.conf* og *pg_hba.conf*
- Tænd
- Profit...

Upgrades

- Indenfor major release (8.2.1->8.2.4) : installer pakke
- Fra en major til en anden:
 - 1) Installer ny version parallelt
 - 2) DUMP gammel data med NY *pg_dump* !
 - 3) Import dump i ny (*psql < dump.sql*)

Vedligehold

Problem: *Pessimistic storage strategy = dead rows*

Løsning: *VACUUM (FULL/ANALYZE/FREEZE)*

Ny løsning: *Auto vacuum daemon*

I praksis gør man tit begge dele

Ellers...

... glem den findes og brug den!

Brug

Værktøjer:

- `psql` (cli)
- PhpPgAdmin (web)
- PgAdminIII (gui)
- Shelltools (`createuser`, `createlang`)
- Egne tools (husk `psql -E`)

Jeg bruger primært *psql* (+ *egne scripts* + *vcs*), men afhængigt af opgave også de andre ...

HUSK: PgAdmin3 er IKKE til at vurdere perf. med (net latency + sløv gridcontrol tælles med!)

Sikkerhed

- Brugere laves direkte i sql (CREATE ROLE <navn> with password '<password>' LOGIN), adgang til baser styres via PG_HBA.CONF (husk HUP postmaster) samt evt. CONNECT-privilege.

To adgangsveje:

- Lokale unix-sockets
- TCP/IP (med/uden SSL)

To “modes”:

- Trust
- Password/MD5

Første gang

- Næsten altid re-initdb... tegnsæt/locale
- Default user “postgres”
- Default database “template1”
- Dvs. *psql template1 postgres*

- Første to “normale” kommandoer

1) `create role sk with password 'secret' login;`

2) `create database sk with owner sk;`

Hvis min unixacc. er sk kan jeg nu logge på via
“psql” eller “psql database-navn” da default via
unix-sockets er trust.

Performance i 5 simple steps *

- 1) ALTID, ALTID nyeste, stable version!
 - 6.x, 7.x, 8.0.x, 8.1.x (snart +8.2) er gammelt l*rt !
 - ca. 20 % performance ekstra for hver major release siden 7.0
 - Meget lidt backportes (af core team)
- 2) Shared_buffers og Effective_cache_size til hhv. 0,25 og 0,50 gange system memory (husk sysctl)
- 3) ANALYZE (eller bedre VACUUM ANALYZE) tabeller tit – og altid ved problemer med en tabel!
- 4) log_min_duration_statement for at finde sløve queries
- 5) EXPLAIN og EXPLAIN ANALYZE dine queries

En RL-optimering

På en kundes system loadede en PHP-side på 20 sekunder. Siden indeholdt tre queries, hvoraf to afsluttede sub-sekund.

Den sidste leverede det rette resultat, men *langsomt*.

Først lidt statistik:

```
> select count(*) from runs;
count
-----
  1492
(1 row)
```

```
> select count(*) from company_run;
count
-----
109093
(1 row)
```

En RL-optimering (2)

```
explain analyze select r.*, (select count(*) from company_run where run_id = r.id) as leadcount from runs r where
gone_through = true and not exists (select null from runs r2 where gone_through = false and r2.zipcode = r.zipcode)
and not exists(select null from runs r3 where r3.zipcode = r.zipcode and r3.ts > r.ts) order by id;
```

QUERY PLAN

```
-----
Index Scan using runs_pkey on runs r (cost=0.00..707402.59 rows=225 width=37) (actual time=89.504..19754.498 rows=699
loops=1)
Filter: (gone_through AND (NOT (subplan)) AND (NOT (subplan)))
SubPlan
-> Seq Scan on runs r3 (cost=0.00..49.37 rows=1 width=0) (actual time=0.201..0.201 rows=1 loops=1484)
    Filter: ((zipcode = $1) AND (ts > $2))
-> Seq Scan on runs r2 (cost=0.00..45.64 rows=1 width=0) (actual time=0.321..0.321 rows=0 loops=1490)
    Filter: ((NOT gone_through) AND (zipcode = $1))
-> Aggregate (cost=2513.63..2513.64 rows=1 width=0) (actual time=27.117..27.119 rows=1 loops=699)
    -> Seq Scan on company_run (cost=0.00..2513.03 rows=241 width=0) (actual time=17.820..26.992 rows=95
loops=699)
        Filter: (run_id = $0)
Total runtime: 19756.076 ms
(11 rows)
```

Inner-loop... $699 * 27,119 \text{ ms} = 18,95 \text{ sekunder!}$

En RL-optimering (3)

```
create index company_run_run_id on company_run(run_id);
```

```
explain analyze select r.*, (select count(*) from company_run where run_id = r.id) as leadcount from runs r
  where gone_through = true and not exists (select null from runs r2 where gone_through = false and r2.zipcode =
    r.zipcode) and not exists(select null from runs r3 where r3.zipcode = r.zipcode and r3.ts > r.ts) order by id;
                                         QUERY PLAN
```

```
-----
Index Scan using runs_pkey on runs r  (cost=0.00..276786.95 rows=225 width=37) (actual time=57.889..968.312
rows=699 loops=1)
  Filter: (gone_through AND (NOT (subplan)) AND (NOT (subplan)))
  SubPlan
    -> Seq Scan on runs r3  (cost=0.00..49.37 rows=1 width=0) (actual time=0.200..0.200 rows=1 loops=1484)
        Filter: ((zipcode = $1) AND (ts > $2))
    -> Seq Scan on runs r2  (cost=0.00..45.64 rows=1 width=0) (actual time=0.311..0.311 rows=0 loops=1490)
        Filter: ((NOT gone_through) AND (zipcode = $1))
    -> Aggregate  (cost=599.78..599.79 rows=1 width=0) (actual time=0.279..0.280 rows=1 loops=699)
        -> Bitmap Heap Scan on company_run  (cost=6.13..599.18 rows=241 width=0) (actual time=0.029..0.163
rows=95 loops=699)
            Recheck Cond: (run_id = $0)
            -> Bitmap Index Scan on company_run_run_id  (cost=0.00..6.07 rows=241 width=0) (actual
time=0.023..0.023 rows=95 loops=699)
                Index Cond: (run_id = $0)
Total runtime: 969.343 ms
(13 rows)
```

Inner-loop nu 699 * 0,280 ms = 195,72 ms!

En RL-optimering (4)

```
create index runs_not_gone_through_zipcode on runs(zipcode) where gone_through = false;

explain analyze select r.*, (select count(*) from company_run where run_id = r.id) as leadcount from runs r
  where gone_through = true and not exists (select null from runs r2 where gone_through = false and r2.zipcode =
    r.zipcode) and not exists(select null from runs r3 where r3.zipcode = r.zipcode and r3.ts > r.ts) order by id
;
```

QUERY PLAN

```
-----
Index Scan using runs_pkey on runs r  (cost=0.00..227127.62 rows=225 width=37) (actual time=9.703..499.578
rows=699 loops=1)
  Filter: (gone_through AND (NOT (subplan)) AND (NOT (subplan)))
  SubPlan
    -> Seq Scan on runs r3  (cost=0.00..49.38 rows=1 width=0) (actual time=0.199..0.199 rows=1 loops=1484)
        Filter: ((zipcode = $1) AND (ts > $2))
    -> Index Scan using runs_not_gone_through_zipcode on runs r2  (cost=0.00..12.28 rows=1 width=0) (actual
time=0.003..0.003 rows=0 loops=1490)
        Index Cond: (zipcode = $1)
    -> Aggregate  (cost=599.78..599.79 rows=1 width=0) (actual time=0.267..0.268 rows=1 loops=699)
        -> Bitmap Heap Scan on company_run  (cost=6.13..599.18 rows=241 width=0) (actual time=0.026..0.153
rows=95 loops=699)
            Recheck Cond: (run_id = $0)
            -> Bitmap Index Scan on company_run_run_id  (cost=0.00..6.07 rows=241 width=0) (actual
time=0.020..0.020 rows=95 loops=699)
                Index Cond: (run_id = $0)
Total runtime: 500.540 ms
(13 rows)
```

En RL-optimering (5)

```
create index runs_zipcode on runs(zipcode) ;
```

```
explain analyze select r.*, (select count(*) from company_run where run_id = r.id) as leadcount from runs r
  where gone_through = true and not exists (select null from runs r2 where gone_through = false and r2.zipcode =
    r.zipcode) and not exists(select null from runs r3 where r3.zipcode = r.zipcode and r3.ts > r.ts) order by id
;
```

QUERY PLAN

```
-----
Index Scan using runs_pkey on runs r  (cost=0.00..166947.67 rows=225 width=37) (actual time=5.665..240.677
rows=699 loops=1)
  Filter: (gone_through AND (NOT (subplan)) AND (NOT (subplan)))
  SubPlan
    -> Bitmap Heap Scan on runs r3  (cost=4.27..10.66 rows=1 width=0) (actual time=0.012..0.012 rows=1
loops=1484)
      Recheck Cond: (zipcode = $1)
      Filter: (ts > $2)
      -> Bitmap Index Scan on runs_zipcode  (cost=0.00..4.27 rows=2 width=0) (actual time=0.007..0.007
rows=3 loops=1484)
        Index Cond: (zipcode = $1)
      -> Bitmap Heap Scan on runs r2  (cost=4.27..10.66 rows=1 width=0) (actual time=0.013..0.013 rows=0
loops=1490)
        Recheck Cond: (zipcode = $1)
        Filter: (NOT gone_through)
        -> Bitmap Index Scan on runs_zipcode  (cost=0.00..4.27 rows=2 width=0) (actual time=0.007..0.007
rows=3 loops=1490)
          Index Cond: (zipcode = $1)
      -> Aggregate  (cost=599.78..599.79 rows=1 width=0) (actual time=0.267..0.268 rows=1 loops=699)
        -> Bitmap Heap Scan on company_run  (cost=6.13..599.18 rows=241 width=0) (actual time=0.026..0.153
rows=95 loops=699)
          Recheck Cond: (run_id = $0)
          -> Bitmap Index Scan on company_run_run_id  (cost=0.00..6.07 rows=241 width=0) (actual
time=0.020..0.020 rows=95 loops=699)
            Index Cond: (run_id = $0)
Total runtime: 241.668 ms
(19 rows)
```

Hands on

Limit/offset

Bruges til at begrænse antallet af rækker, og flytte “vinduet”.

```
select * from sometable order by id limit 10;  
select * from sometable order by id limit 10 offset 10;
```

Husk selects uden *order by* er i tilfældig rækkefølge (typisk disklayout) og bør “konceptuelt” ikke bruges med limit/offset.

Sequences

- Når du skal bruge et autogenerated id-felt
- Er et navngivet objekt der giver næste tal i en række
- Garanti mod overlap, ikke mod huller!
- Skabes via `CREATE SEQUENCE <navn>`
- Kender principielt set kun 2 funktioner:
 - `nextval ('<navn>')`
 - `currval ('<navn>')`
- Kan dog selectes som tabel (mere info, ingen fremføring)
- Pseudotype “serial” sparer lidt tid:
 - `Serial = int4 not null default nextval('<tabelnavn>_seq')`

Sequences DEMO

```
create table test1 (id serial, navn varchar, primary key(id));
NOTICE: CREATE TABLE will create implicit sequence "test1_id_seq" for
serial column "test1.id"
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index
"test1_pkey" for table "test1"
CREATE TABLE
```

```
insert into test1(navn) values ('Kurt');
INSERT 0 1
```

```
insert into test1(id, navn) values (default, 'Jens');
INSERT 0 1
```

```
insert into test1(id, navn) values (null, 'Ole');
ERROR: null value in column "id" violates not-null constraint
```

```
select nextval('test1_id_seq');
nextval
-----
3
(1 row)
```

```
insert into test1(id, navn) values (3, 'Niels');
INSERT 0 1
```

Sequences DEMO (2)

```
create sequence customer_id_seq;
```

```
create table company(company_id int4 not  
null default  
nextval('customer_id_seq'),  
company_name varchar not null, primary  
key(company_id));
```

```
create table person(person_id int4 not  
null default  
nextval('customer_id_seq'), person_name  
varchar not null, employee_of int4 not  
null references company, primary  
key(person_id));
```

Typer og casting

PostgreSQL forsøger selv at finde de rette datatyper og konvertere om muligt.

Nogle gange lykkes det ikke, og der kan man foran tekst angive et type-hint.

Man kan også tvinge en type via CAST-funktionen eller dobbelt-kolon.

Typer og casting DEMO

```
select '3';  
select integer '3';  
select float '3';  
select 8/3;  
select cast(8 as float)/3;  
select 8 / 3::float;  
select ( 20000 * 1.04^50)::int4;
```

```
select '2007-08-30';  
select date '2007-08-30';  
select timestamp '2007-08-30';
```

Tid og dato håndtering

Fire datatyper handler om tid og dato:

- *Date* en simpel dato '2007-08-30'
- *Timestamp* er en dato og tid '2007-08-30 20:45:00'
- *Interval* differencen mellem to Timestamps
- *Time* tid

Date understøtter simpel addering og subtrahering af heltal. Fx. '2007-08-30' + 5 = '2007-09-04'

Timestamp kan adderes og subtraheres af *intervaller*. Fx. '2007-08-30 20:45:00' + interval '5 days' = '2007-09-04 20:45:00'

Tid og dato håndtering (2)

Vigtige funktioner:

- `date_trunc(text,timestamp)` beskærer timestamp til given præcision, fx.

```
select date_trunc('quarter', '2004-08-05 20:50:39'::timestamp);
```

```
date_trunc
```

```
-----
```

```
2004-07-01 00:00:00
```

```
(1 row)
```

- `extract(field from timestamp)` henter (int) værdi fra timestamp

Tid og dato håndtering DEMO

```
select age('1979-04-30 20:50:00'::timestamp);
```

```
select timestamp '1945-05-05' - timestamp '1940-04-09';
```

```
select timestamp '2007-01-01' + interval '200 days';
```

```
select current_timestamp;
```

```
select date_trunc('quarter', current_timestamp);
```

Transaktioner

Transaktioner er krumptappen i ACID-compliant databaser.

En transaktion sikrer at en gruppe af kommandoer enten alle sker eller slet ikke sker.

Fire modes

- Read dirty: læs data som de er
- read committed: sikrer clean data
- Read repeated: sikrer at data forbliver ens i hele transaktionen
- Serializable: meget strengere, transaktioner failer ved ændrede data og skal nogle gange prøves igen

Transaktioner (2)

Read committed er typisk standard. Kan fx. bruges til delete and re-fill strategier (dermed ingen huller i data availability).

Serializable bruges fx. til bank-konti. Transaktioner kan ikke køre parallelt da de ligger under fælles constraint (kontostand skal være større end 0 fx.)
Serializable fejler hvis forudsætningerne ændrer sig.

Transactions i PGSQL

PostgreSQL kører auto-commit mode som default, kun ganske få strukturelle kommandoer kører uden transaktioner.

Auto-transaction mode disables ved explicit *begin* og explicit *commit / rollback*.

Kun to modes mulige (pga. MVCC) : read committed og serializable. Uanvendelige modes “opgraderes” til strengere (read repeated -> serializable)

Foreign keys

FKKey sikrer at en reference til en anden tabel er gyldig.

Ved indsættelse fejler transaktionen hvis referencen er ugyldig.

Efterfølgende kan der ved ændringer (og sletninger) specificeres en af følgende strategier:

- Restrict (ændring mislykkes)
- Cascade (viderefør ændring)
- Set null
- Set default

Foreign keys DEMO

```
create table companies(id serial,  
name varchar not null, primary  
key(id));
```

```
create table employees (id  
serial,name varchar not null,  
company_id int4 not null  
references company on update  
restrict on delete cascade,  
primary key(id));
```

Domains

Bruges til at sikre ensartede datavalidering :

```
create function test_email(varchar) returns boolean as  
...
```

```
create domain nametype as varchar not null;  
create domain emailtype as varchar check  
(test_email(VALUE));
```

```
create table company (id serial, name nametype, email  
emailtype, ...  
create table person(id serial, name nametype, email  
emailtype...)
```

Nu er:

- name *not null*
- email testes af funktion (men er ikke *not null*)

Partial Indexes

Nogle gange findes data hvor kun et lille subset er interessant hele tiden (`is_current = true`).

Partial indexes kan her både hjælpe på performance (pga. fysisk mindre indexes) og garantere uniqueness på det aktive sæt

Partial Indexes DEMO

```
create table documents(  
id serial,  
filename varchar not null,  
is_current boolean not null default true,  
content varchar,  
primary key(id));  
  
create unique index document_current_filename on documents(filename) where is_current = true;  
  
insert into documents (filename, content, is_current) values ('note.txt', 'husk 1', false);  
insert into documents (filename, content, is_current) values ('note.txt', 'husk 2', false);  
insert into documents (filename, content, is_current) values ('note.txt', 'husk 3', true);  
insert into documents (filename, content, is_current) values ('note.txt', 'husk 4', true);  
update documents set is_current = false where filename = 'note.txt' and is_current = true;  
insert into documents (filename, content, is_current) values ('note.txt', 'husk 4', true);
```

Functional indexes

```
create table users (id serial, alias varchar
  not null, full_name varchar not null, primary
  key(id));
```

```
create index users_alias_ix on users(alias);
```

Nu ønskes case-insensitive search på alias.

```
select * from users where lower(alias) =
  'something'
```

Denne bruger dog ikke index, men dette kan opnås via

```
create index users_alias_cis_ix on
  users(lower(alias));
```

Triggers

Er en mekanisme til at køre en (PL) funktion ved ændringer i data (inserts/updates/deletes).

Triggers kan køre FØR eller EFTER ændringen

Triggers kan være på række eller statement niveau

Triggers kan ændre data, blokere ændringen eller udføre yderligere ting (audit log fx).

Triggers DEMO

```
create table login (id serial, username varchar, password varchar, valid_from
    timestamp not null default current_timestamp, valid_until timestamp, primary
    key(id));
```

```
create or replace function login_password_scrambler() returns trigger as $$
begin
NEW.password = md5(NEW.password);
return NEW;
end;
$$ language plpgsql;
```

```
create trigger login_password_scrambler_trg before update or insert on login for
    each row execute procedure login_password_scrambler();
```

```
create or replace function login(username varchar, password varchar) returns
    setof login as $$ select * from login where username = $1 and password =
    md5($2) and (valid_until is null or valid_until > current_timestamp) ; $$
language sql;
```

```
insert into login(username,password) values ('sk','password');
```

```
select * from login;
```

```
select * from login('sk','password');
```

Triggers DEMO (2)

```
create or replace function password_generator()  
  returns varchar as $$ select varchar  
  'secret'; $$ language sql;
```

```
create or replace function new_password(id  
  int4) returns varchar as $$  
declare holder varchar;  
begin  
select into holder password_generator();  
update login set password = holder where id =  
  $1;  
return holder;  
end;  
$$ language plpgsql;
```

Subqueries

Laver et “query i query”.

Tre typer:

- where

```
select * from users where company_id in (select id from companies  
where ...)
```

- from

```
select id, 1- pct as rebate, value as list_price, pct * value as  
your_price from (select id, 0.9 + random() * 0.1 as pct, value from  
goods)
```

- resultat

```
select id, (select sum(value) from subtable st where st.id =  
mt.id) from maintable mt
```

Set-returning functions

Demo

```
create type mytype as (id int4, id2 int4);
```

```
create or replace function myid(int4) returns setof  
mytype as $$ select * from generate_series(1,$1) a ,  
generate_series(1,$1) b; $$ language sql;
```

```
select *, cid * id * id2 as result from (select cid,  
(myid(cid)).* from generate_series (2,4) as c(cid)) x  
order by result desc;
```

```
select * from myid(8) where id2 < id;
```

Schemas

Er en måde at strukturere sine data på (tænk “stier”).

Vigtige kommander:

```
create schema rapporter;  
set search_path = public, rapporter;  
create table schema.table  
select * from schema.table;
```

Default search_path er “\$user, public”

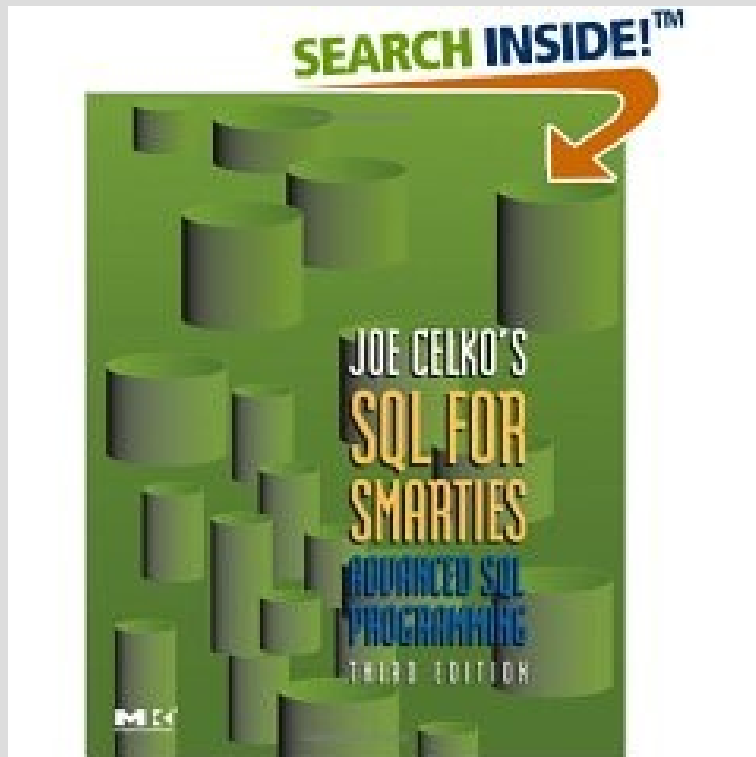
Afslutning

Interessante urls

- <http://www.postgresql.org/docs/8.2/interactive/index.html>
- <http://www.postgresql.org>
- <http://pgfoundry.org/> (sourceforge for PG projekter)
- <http://www.planetpostgresql.org> (PostgreSQL blog)
- <http://www.varlena.com> (PostgreSQL General Bits)
- <http://www.postgresql.org/docs/techdocs> (technical docs)

Videre herfra?

For avanceret og hurtig sql generelt:



For PostgreSQL:

<http://www.postgresql.org/docs/8.2/interactive/>
(rtfm)

Især:

- II The SQL Language
 - 8. Datatypes
 - 9. Functions and Operators
 - 12. Concurrency Control
- V Server Programming
 - 33. Extending SQL
 - 37. PL/pgSQL
- VI Reference
 - I. SQL Commands

Tak for idag

Spørgsmål?