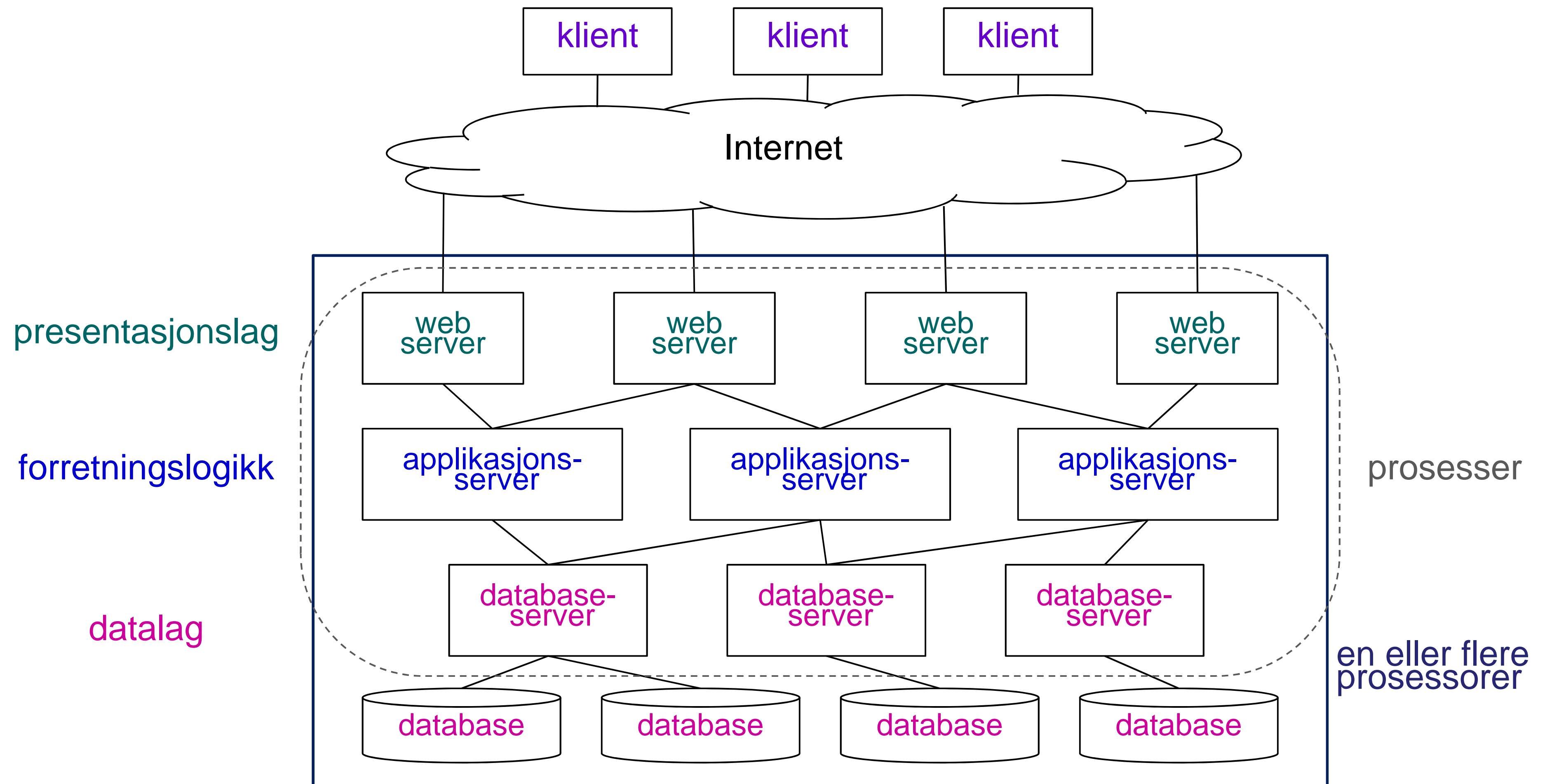
UNIVERSITETET
I OSLO

Systemaspekter ved SQL

© Institutt for Informatikk

3-lagsarkitektur i webapplikasjoner



SQL-omgivelsen

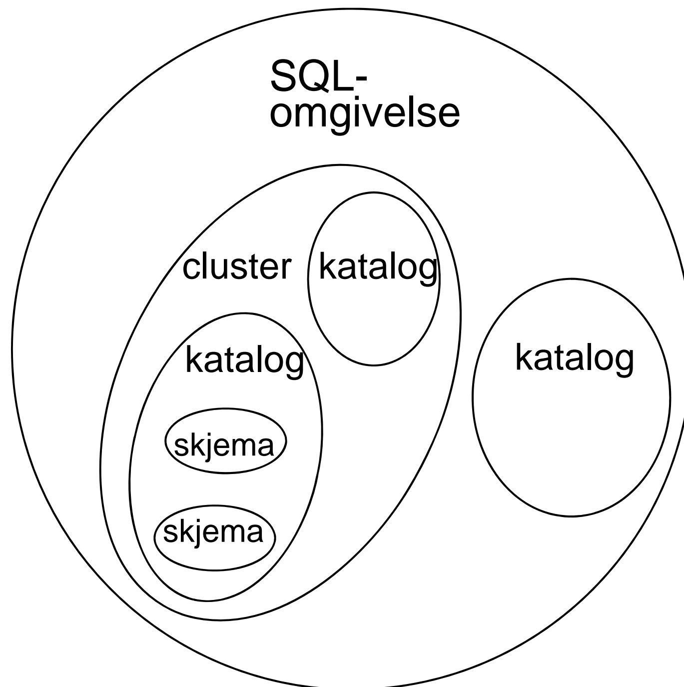
- Et rammeverk som data kan eksistere under og hvor SQL-operasjoner på dataene kan eksekveres
  - I praksis: Installasjon av DBMS på en gitt plattform, dvs. en samling av maskiner m/tilhørende nettverk
- **Skjema**: Samling av tabelldefinisjoner
  - Hver tabelldefinisjon omfatter views, assertions, triggere, stored procedures mm.
  - Utgjør et *navnerom*; tabellnavn mm. kan gjenbrukes mellom skjemaer
- **Katalog**: Samling av skjemaer
  - Skjemanavn må være entydige innen katalogen
  - Hver katalog inneholder et spesialskjema med informasjon om alle skjemaer i katalogen
- **Cluster**: Samling av kataloger
  - Hver bruker har et assosiert cluster som er mengden av kataloger tilgjengelige for brukeren
    Dvs at clusteret definerer en bestemt brukers «Database»

Begrepsbruk i Postgres

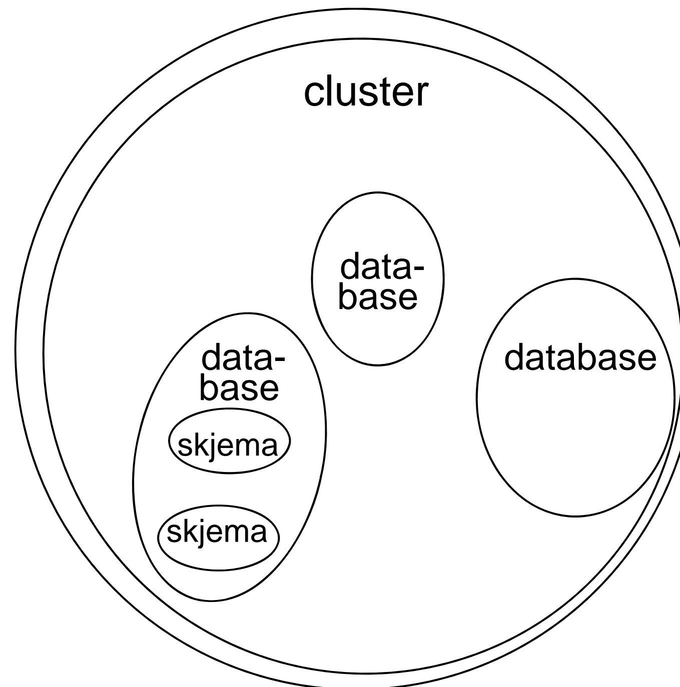
- **Skjema**: Samling av tabelldefinisjoner
  - Brukere kan aksessere data på tvers av skjemaer (i henhold til rådende aksessrettigheter)
- **Database**: Svarer til det som i standarden kalles **katalog**
  - Brukere kan ikke aksessere data på tvers av databaser
  - Skjemaer som fins i alle databaser:
    - public
    - brukerdefinerte skjemaer
    - pg_catalog
- **Cluster**: Omfatter alle databaser som håndteres av en og samme Postgres-serverinstans

SQL-omgivelsen

standardterminologi



Postgres-terminologi



Systemkatalogen

- Inneholder informasjon om systemet (metadata)
  - systemet bruker systemkatalogen til bokholderi
  - definert via vanlige tabelldefinisjoner
    - brukere kan gjøre queries mot systemkatalogen
- Systemkatalogen i Postgres: pg_catalog
  - Gjenfinnes som et skjema i hver database
  - Noen systemtabeller inneholder informasjon som gjelder hele clusteret, ikke bare den enkelte database
    Eks.: pg_database(datname, datdba, encoding, ...)
    - fins som en tabell i samtlige pg_catalog-skjemaer
    - inneholder ett tuppel for hver database

Skjemaer og kataloger i Postgres

create schema <skjemanavn>;
<elementdeklarasjoner kvalifisert med skjemanavn>;
drop schema skjemanavn;

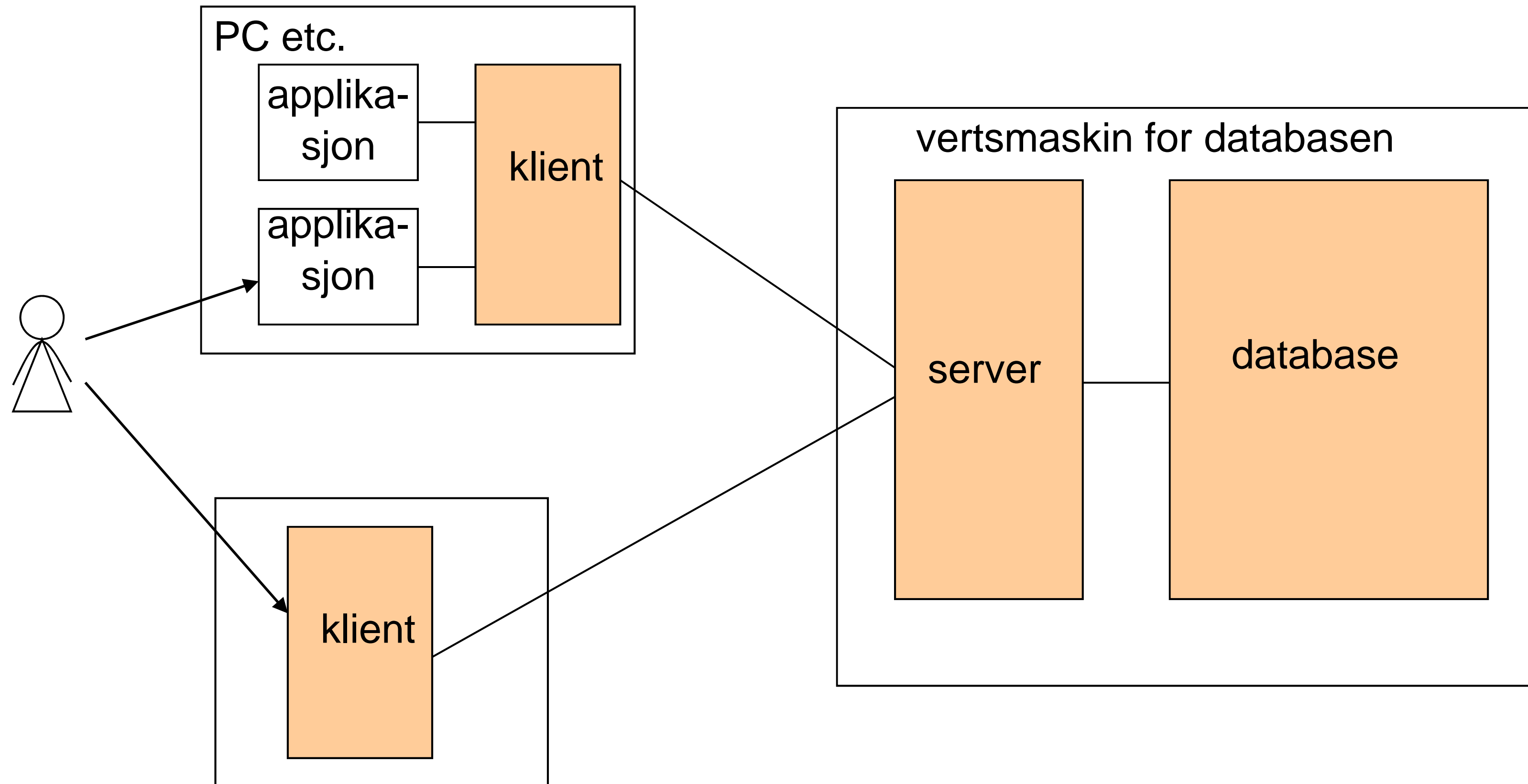
create database dbnavn [**owner** rollenavn];[†]
drop database dbnavn;[†]

Endre default skjema (navn på elementer i default skjema behøver ikke kvalifiseres med skjemanavn):
set search_path **to** skjemanavn;

Endre inneværende database:
\connect dbnavn

[†]Ikke-standard SQL; svarer til katalog i standard SQL-terminologi

Klienter og tjenere i SQL-omgivelsen



Klient-tjener i SQL

- **SQL-server**: Prosess som bistår i å utføre operasjoner på databaseelementene
  - Svarer til databaseserver i webarkitekturen
- **SQL-klient**: Prosess som hjelper en bruker (person eller applikasjon) i å få kontakt med en server
  - Formidler forespørsel om utføring av en operasjon fra bruker til server og resultat fra server til bruker
  - Svarer til applikasjonsserver i webarkitekturen
- **Modul** i SQL-terminologi = applikasjonsprogram

Impedance mismatch

- Datamodellen til SQL avviker fra datamodellen til tradisjonelle imperative språk
  - Basale datastrukturer:
    - C/C++/Java/..: int, real, char, pekere, arrays, records, klasser, ..
    - SQL: bag
  - Programflyt:
    - C/C++/Java/..: if-then-else, for-løkke, while-løkke, aksess via pekere, ...
    - SQL: Sekvensielt en og en SQL-setning
- Derfor krever inkorporering av SQL i en vanlig programmeringsomgivelse endel apparatur

Aksess til SQL

SQL-standarden krever at enhver SQL-implementasjon tilbyr bruker minst én av følgende:

1. **Generisk SQL-grensesnitt**:
   Bruker kan taste inn SQL-setninger via grensesnittet
   Grensesnittet sørger for eksekvering av setningene
   I Postgres: psql
2. **Embedded SQL**:
   Programmerer kan benytte SQL-relaterte konstruksjoner innkapslet i et vertsspråk
   En preprosessor omformer koden til ren vertsspråkkode
3. **Grensesnitt mot applikasjoner skrevet i andre språk**:
   Programbibliotek tilpasset det enkelte språket som gjør at programmene kan benytte SQL mot databaser
   Kommunikasjonen skjer via parameteroverføring og eventuelt felles variable

Embedded SQL

- All SQL-kode i vertsspråket markeres med
  EXEC SQL ...
- Man kan deklarere ***fellesvariable*** (shared variables) for vertsspråket og SQL-koden
  - Disse benyttes til informasjonsutveksling mellom systemene
- Tuplene i resultatet av en query hentes ut via en **cursor**
  - Denne kan gjennomløpe tuplene og legge verdiene i fellesvariable
  - Hvis man vet at resultatet inneholder nøyaktig ett tuppel, kan tuppelets verdier hentes ut direkte (uten bruk av cursor)
- Feilsituasjoner formidles via en spesiell variabel, SQLSTATE
- Postgres ECPG tilbyr embedded SQL i vertsspråket C

ECPG-eksempel

```
void worthRanges() {
  int i, digits, counts[15];
  EXEC SQL SET AUTOCOMMIT TO ON;
  EXEC SQL CONNECT TO mydb USER myname/passwd;
  EXEC SQL BEGIN DECLARE SECTION;
    int worth;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL DECLARE execCursor CURSOR FOR SELECT netWorth FROM MovieExec;
  EXEC SQL OPEN execCursor;
  for (i=0; i<15; i++) counts[i]=0;
  while (1) {
    EXEC SQL FETCH FROM execCursor INTO :worth;
    if (!(strcmp(sqlca.sqlstate, "02000"))) break;
    digits=1;
    while ((worth/=10) > 0) digits++;
    if (digits <= 14) counts[digits]++;
  }
  EXEC SQL CLOSE execCursor;
  for (i=0; i<15; i++) printf("digits = %d: number of execs = %d\n"), i, counts[i]);
  EXEC SQL DISCONNECT;
}
```

Dynamisk SQL

Ved dynamisk SQL suppleres SQL-setninger via tekststrenger
Strengenes innhold kan beregnes dynamisk i vertsspråket

Eks. i ECPG:

```
void insertSquares() {
  EXEC SQL SET AUTOCOMMIT TO ON;
  EXEC SQL CONNECT TO mydb USER myname/passwd;
  EXEC SQL BEGIN DECLARE SECTION;
    int i, j;
    const char *stmt = "insert into square values (?,?)";
  EXEC SQL END DECLARE SECTION;
  EXEC SQL PREPARE mystmt FROM :stmt;
  for (i=0; i<10; i++) {
    j = i*i;
    EXEC SQL EXECUTE mystmt USING (:i, :j);
  }
  EXEC SQL DEALLOCATE PREPARE mystmt;
  EXEC SQL DISCONNECT;
}
```

Grensesnitt mot applikasjoner skrevet i andre språk

- **CLI – Call-Level Interface**
  - De facto standard API for SQL-baserte databaser
  - Programmerer skriver all kode i et vertsspråk (typisk C) og benytter bibliotek av funksjoner for å knytte opp mot og aksessere databasen
  - Databaseaksess oppnås ved å oversende SQL-setninger
  - Liten forskjell fra preprosessert kode fra embedded SQL bortsett fra at koden kan gjøres mindre proprietær (dvs. mindre avhengig av den aktuelle DBMSen)
- **ODBC – Open DataBase Connectivity**
  - Standard API mot DBMSer
  - Bygger på CLI-spesifikasjonene
  - Uavhengig av programmeringsspråk, databasesystemer, operativsystemer
- Postgres tilbyr ikke CLI, men leveres med drivere for ODBC

JDBC – Java Database Connectivity

Javas versjon av CLI

1. Opprett en driver for databasesystemet (installasjons- og implementasjonsavhengig)
2. Opprett en forbindelse til databasen via driveren Forbindelsen tilhører klassen Connection
3. Opprett et statementobjekt,
   plasser inn en SQL-setning,
   bind verdier til SQL-setningens parametre,
   eksekver SQL-setningen,
   gjennomløp resultatet,
   alt via Connection-objektet
4. Lukk forbindelsen

Det fins JDBC-drivere som benytter en ODBC-driver for å koble opp mot databasesystemet (JDBC-ODBC bridge)

Eksempel på bruk av JDBC

```
Connection myCon =
   DriverManager.getConnection(<URL>, <navn>,
   <passord>);
Statement execStat = myCon.createStatement();
ResultSet worths = execStat.executeQuery(
    "select netWorth from MovieExec");
integer worth, digits;
integer[] counts = new integer[15];
while (worths.next()) {
  worth = worths.getInt(1);
  digits = 1;
  while ((worth /= 10) > 0) digits++;
  if (digits <= 14) counts[digits]++;
}
```

Parameteroverføring til queries i JDBC

```
Connection myCon =
  DriverManager.getConnection(<URL>,
    <navn>, <passord>);
PreparedStatement studioStat =
  myCon.createStatement("insert into " +
  "Studio(name, address) values (?,?)");
<les verdier til studioName og studioAddr>;
studioStat.setString(1, studioName);
studioStat.setString(2, studioAddr);
studioStat.executeUpdate();
```

Stored procedures

- Utvidelse av SQL med prosedyralt språk
- I språket kan man definere funksjoner
  - Definisjonene legges inn via generisk SQL-grensesnitt og lagres i selve databaseskjemaet
  - Funksjonene har tilgang til relasjonene direkte eller via SQL-setninger
- Uttrykkskraften er som for tradisjonelle programmeringsspråk
  - Man har tilgang på vanlige språkkonstruksjoner
  - I tillegg kan man bruke SQL-setninger
- Fordeler fremfor å definere funksjonene i en applikasjon (dvs. ved hjelp av embedded SQL eller CLI/ODBC):
  - Mindre kommunikasjonsoverhead mellom klient og server
  - Slipper memory overrun ved store resultatmengder

- Postgres: PL/pgSQL
  - Funksjoner parses og lagres som binærkode første gang funksjonene kalles i en sesjon

DBMS – Database Management System (repetisjon)

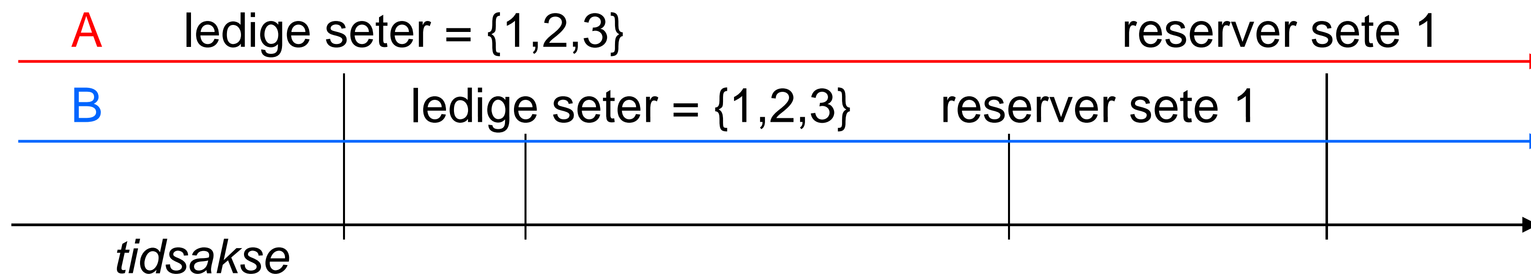
- Spesialisert SW
- Karakteristika:
  - Persistens
  - Transaksjonshåndtering
    - **A** tomicity
    - **C** onsistency preservation
    - **I** solation
    - **D** urability
  - Programmeringsgrensesnitt

Serialiserbarhet

- En eksekvering av en samling funksjoner (hvor hver funksjon kan omfatte en eller flere databaseoperasjoner) er ***seriell*** hvis eksekveringen fullføres fullstendig for én funksjon før neste funksjon eksekveres.
- Eksekveringen er ***serialiserbar*** hvis funksjonseksekveringene er slik at (selv om de rent faktisk kanskje *overlapper* i tid) det *fins en seriell eksekvering* som gir samme totalresultat

Hvorfor serialiserbarhet?

- Ved flere samtidige prosesser på samme dataelement kan resultatet av en ikke-serialiserbar eksekvering være uforutsigbart
- **Eks**: Flyreservasjon =
  finn ledige seter; reserver et sete;



Atomisitet

- En eksekvering er ***atomær*** hvis vi er garantert at hvis det ikke er mulig å gjennomføre alle deler av eksekveringen, så vil resultatet av eksekveringen være som om den aldri var blitt påbegynt

Hvorfor atomisitet

- Selv med serialiserbarhet kan vi få uventede situasjoner – hvis en funksjonseksekvering blir avbrutt (f.eks. kræsjer).
  Dette kan skje også om funksjonen består av bare én databaseoperasjon fordi underliggende hardware eller software (maskininstruksjoner) kan bestå av flere deler.
- **Eks**: **update** R **set** v1=x1, v2=x2;
  = for hvert tuppel: skriv x1 i v1; skriv x2 i v2;

  Katastrofalt selv hvis bare ett tuppel:

  avbrudd

Transaksjoner

- **Transaksjon**: Samling av databaseoperasjoner (en eller flere) som vi krever at skal eksekveres atomært
  - Enten må alle delene av operasjonene lykkes, eller så må databasen settes tilbake slik den så ut før operasjonene ble påbegynt
  - I tillegg forlanger SQL-standarden at transaksjoner eksekveres på en serialiserbar måte (med mindre et annet ***isolasjonsnivå*** er angitt)

Transaksjonsbegrepet i det generiske SQL-interfacet

- Hver SQL-setning er en transaksjon i seg selv
- Hvis en SQL-setning utløser triggere, er disse også en del av samme transaksjon
- I Postgres psql kan vi samle flere kommandoer i en transaksjon ved **begin + commit**:

=> **begin**; (eller **start transaction**;)
=> ...
=> ... (SQL-setninger o.a. som inngår i transaksjonen)
=> **commit**; (eller **rollback** hvis ikke alt gikk som planlagt)

Standard SQL: **start transaction**, **commit**, **rollback**

Transaksjoner i embedded SQL

- Hver SQL-setning er automatisk en transaksjon (jf. transaksjoner i det generiske SQL-interfacet)
  - I Postgres ECPG er det i default modus *ikke* slik: Alle enkeltstående SQL-setninger må committes *eksplisitt* med EXEC SQL COMMIT
- En samling SQL-setninger og setninger i vertsspråket kan markeres som å tilhøre en transaksjon:
  - En transaksjon initieres ved

    EXEC SQL START TRANSACTION;
  - Transaksjonen avsluttes ved en av SQL-setningene

    EXEC SQL COMMIT;

    EXEC SQL ROLLBACK;
  - I Postgres ECPG kan i default modus den innledende EXEC SQL START TRANSACTION sløyfes

commit og rollback

- **commit**: Markerer at transaksjonen var vellykket
  - Fra **start transaction** til **commit** er alle endringer i databasen *tentative*, de kan senere omgjøres
  - Ved **commit** gjøres endringene *varige* – de ***committes***†
- **rollback**: Markerer at transaksjonen feilet
  - Alle endringer i databasen siden **start transaction** *omgjøres* – det foretas en ***rollback***

†Unntak: Hvis transaksjonen utløste krav om sjekking av skranker som er angitt som **deferred**, vil denne sjekkingen først bli utført. Hvis skrankene ikke holder, vil det bli foretatt en rollback på tross av **commit**-instruksjonen

Når kan parallelle prosesser gi problemer?

- **Lesing** forårsaker aldri problemer
  - Hvis alle transaksjoner i en eksekvering bare foretar leseoperasjoner, kan vi tillate parallelle utførelser av dem
  - Vi hjelper systemet med å optimalisere ved å markere en transaksjon som en ren lesetransaksjon:
    start transaction read only;
- Ved **skriving** til databasen kan vi få brudd på serialiserbarhetskravet
  - Vi kan markere en transaksjon som en (lese- og) skrive-transaksjon med
    start transaction read write;

Hva slags problemer?

- Tre fenomener som fører til brudd på serialiserbarhet (i henhold til ANSI SQL-92):
  - Dirty read
  - Nonrepeatable read (= fuzzy read)
  - Phantom read

  (Dette er ikke en komplett liste; senere er det identifisert flere anomaliteter)

Dirty read

- **Dirty data** = data skrevet av en transaksjon som ennå ikke har gjort commit
- **Dirty read** = en transaksjon leser data skrevet av en parallell ikke-committet transaksjon
  - Man risikerer at transaksjonen som foretok skrivingen, senere gjør en rollback slik at dataene er ugyldige

Nonrepeatable read

- En transaksjon leser på nytt et dataelement den alt har lest, men opplever at dataene er endret

  Dette kan skyldes at dataene i mellomtiden er blitt modifisert av en annen transaksjon som committet mellom de to leseoperasjonene

  - Merk at dette kan skje selv om systemet garanterer at transaksjonen ikke kan lese dirty data
    (den andre transaksjonen ble f.eks. påbegynt etter den første les-operasjonen og avsluttet før den andre)

Phantom read

- En transaksjon eksekverer på nytt en query den alt har eksekvert, men opplever at noen nye tupler er kommet til (men de tuplene man har fått tilslag på, vil ikke forsvinne og ikke endres)

  Dette kan skyldes at en annen transaksjon som nylig har committet, har lagt til tupler

  – Merk at dette kan skje selv om systemet garanterer at transaksjonen ikke kan oppleve nonrepeatable reads (på tuppelnivå)

Er ikke-serialiserbarhet **alltid** et problem?

- Noen ganger er det *ikke* kritisk om systemet tillater at ett eller flere av disse fenomenene kan opptre
  - Kanskje er resultatet av transaksjonen likevel tilfredsstillende
  - Kanskje kan man leve med at dataene av og til må forkastes i etterhånd
- Å tillate avvik fra serialiserbarhet vil øke gjennomstrømmingen av transaksjoner i systemet

ANSI SQL-92 isolasjonsnivåer

- ***Isolasjonsnivået*** til en transaksjon angir i hvilken grad serialisering kan fravikes i transaksjonen
- Isolasjonsnivået til en transaksjon påvirker bare *denne* transaksjonens virkelighetsoppfatning
- Mulige isolasjonsnivåer i SQL:
  - **serializable** = det fins en eksekvering som gir samme resultat og som er seriell
  - **repeatable read** = phantom read
  - **read committed** = nonrepeatable read
  - **read uncommitted** = dirty read

Isolasjonsnivå	**dirty read**	**nonrepeatable read**	**phantom read**
read uncommitted	mulig	mulig	mulig
read committed	umulig	mulig	mulig
repeatable read	umulig	umulig	mulig
serializable	umulig	umulig	umulig

Hvordan sette isolasjonsnivå

start transaction [*transaction_mode*]

hvor *transaction_mode* er en kommaseparert liste av elementene

- **isolation level** {**serializable** | **repeatable read** |
  read committed | **read uncommitted** }
- **read write** | **read only**

Eksempel:

start transaction read only, **isolation level read committed**;

- Default aksessmodus ved de forskjellige isolasjonsnivåene:
  - serializable: read-write
  - repeatable read: read-write
  - read committed: read-write
  - read uncommitted: read-only

Isolasjonsnivåer i Postgres

- I Postgres og de fleste moderne systemer brukes **snapshot isolation** som default isolasjonsnivå
  - Styrke mellom **read committed** og **serializable**
  - Mer effektiv å håndheve enn de tradisjonelle isolasjonsnivåene
  - For de fleste praktiske formål er **snapshot isolation** tilstrekkelig
- Postgres vs. ANSI SQL-92 isolasjonsnivåer:
  - **read committed** (default nivå) er egentlig **snapshot isolation**
  - **read uncommitted** reduseres til **read committed**
  - **repeatable read** reduseres til **serializable**
  - **serializable** garanterer *ikke* fullstendig serialiserbarhet, det fins sære unntak