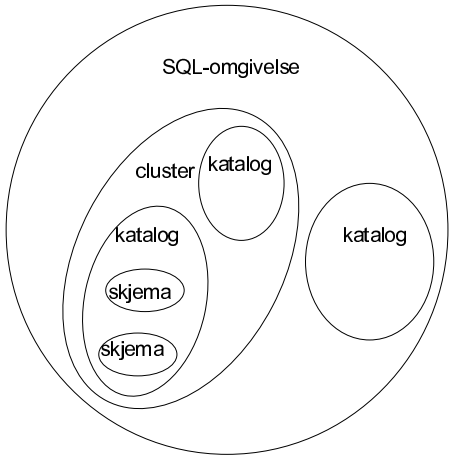
UNIVERSITETET I OSLO

Systemaspekter ved SQL

SQL-omgivelse

- 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
- **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»

SQL-omgivelser



Skjema

create schema <skjemanavn> <elementdeklarasjoner>

Eks:

```
create schema MovieSchema
  create table MovieStar(
    name char[30] primary key,
    address varchar[255],
    gender char[1],
    birthdate date
  );
```

Veksling mellom skjemaer:

```
set schema MovieSchema;
```

Katalog

Veksling mellom kataloger:

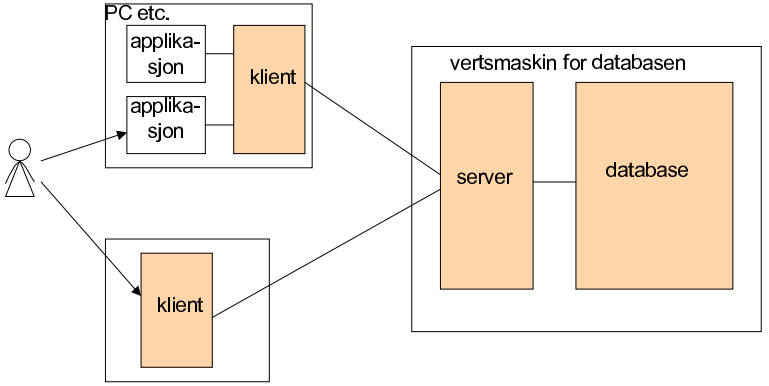
set catalog <katalognavn>

- For kataloger fins ingen create-setning. Ved navnekonflikter kan skjemaelementer kvalifiseres med skjema- og katalognavn slik:

  MovieCatalog.MovieSchema.MovieStar

  På denne måten kan også elementer utenfor nåværende katalog/skjema navngis

Klienter og tjenere i SQL-omgivelsen



Klient-tjener i SQL

- **SQL-server**: Prosess som bistår i å utføre operasjoner på databaseelementene
- **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
- Opprettelse av forbindelse mellom klient og server:
  connect to <servernavn> **as** <connectionnavn>
  authorization <navn og passord>
- Veksling mellom forbindelser:
  set connection <connectionnavn>
- Terminering av en forbindelse:
  disconnect <connectionnavn>
- **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
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 innkapslet i vertsspråket, markeres med
  EXEC SQL ...
- Man kan deklarere ***fellesvariable*** (shared variables) for vertsspråket og SQL-koden
- Disse benyttes til informasjonsutveksling mellom systemene
- Feilsituasjoner formidles via en spesiell variabel, SQLSTATE
- 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)

Eksempel embedded SQL

```
void worthRanges() {
  int i, digits, counts[15];
  EXEC SQL BEGIN DECLARE SECTION;
    int worth;
    char SQLSTATE[6];
  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(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]);
}
```


Deklarasjon av fellesvariable

EXEC SQL BEGIN DECLARE SECTION;

<felles variable; syntaks avhenger av vertsspråket>

EXEC SQL END DECLARE SECTION;

- I vertsspråkets setninger benyttes fellesvariablene som andre programvariable
- I SQL-setninger kan fellesvariablene benyttes som om de er konstanter
  Fellesvariable prefikses med ':'

SQL-setninger som ikke er queries

```
EXEC SQL <SQL-setning>
```

Eks:

```
EXEC SQL INSERT INTO
    Studio(name, address)
    VALUES (:studioName, :studioAddr)
```

der studioName og studioAddr er fellesvariable

Select som returnerer ett tuppel

```
EXEC SQL SELECT <projeksjonsliste>
  INTO <liste av fellesvariable,
        antall og type må stemme med projeksjonsliste>
  FROM <liste av relasjoner>
  WHERE <betingelse>
```

Eks:

```
EXEC SQL SELECT netWorth
  INTO :presNetWorth
  FROM Studio, MovieExec
  WHERE presC# = cert# AND
        Studio.name = :studioName;
```


Generell query

1. Cursordeklarasjon
   EXEC SQL DECLARE <cursornavn>
   [SCROLL] CURSOR FOR <query>
2. Initialisering av cursor
   EXEC SQL OPEN <cursornavn>
3. Innhenting av neste tuppel til cursor
   (gjøres så mange ganger man vil)
   EXEC SQL FETCH [<mode>]
   FROM <cursornavn>
   INTO <liste av fellesvariable>
   hvor <mode> er en av NEXT, PRIOR, FIRST, LAST, ...
   (<mode> benyttes hvis og bare hvis SCROLL CURSOR)
4. Lukk cursor
   EXEC SQL CLOSE <cursornavn>

Dynamisk SQL

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

```
EXEC SQL PREPARE <var> FROM <expr>;
EXEC SQL EXECUTE <var>;
```

Alternativt:

```
EXEC SQL EXECUTE IMMEDIATE <var>;
```

Eks:

```
void readQuery() {
  EXEC SQL BEGIN DECLARE SECTION;
    char *query;
  EXEC SQL END DECLARE SECTION;
  <les inn en streng og få query til å peke på første tegn i strengen>;
  EXEC SQL EXECUTE IMMEDIATE :query;
}
```

Stored procedures

- PSM – Persistent Stored Modules – er et imperativt programmeringsspråk
- En ***modul*** i PSM er en funksjons- eller prosedyredeklarasjon o.a.
- Modulene lagres i selve databaseskjemaet Modulene har tilgang til relasjonene direkte eller via SQL-setninger
- SQL utvidet med PSM gir oss et språk med samme uttrykkskraft som tradisjonelle programmeringsspråk

Eksempel PSM

```
create procedure Mean (
  in s char[15],
  out mean real
)
declare Not_Found condition for sqlstate '02000';
declare movieCursor cursor for select length from Movie where studioName = s;
declare newlength integer;
declare movieCount integer;
begin
  set mean = 0.0;
  set movieCount = 0;
  open movieCursor;
  movieLoop: loop
    fetch movieCursor into newLength;
    if Not_Found then leave movieLoop end if;
    set movieCount = movieCount + 1;
    set mean = mean + newLength;
  end loop;
  set mean = mean /movieCount;
  close movieCursor;
end;
```


CLI – Call-Level Interface

- Programmerer skriver all kode i et vertsspråk 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)

Datatyper i CLI-biblioteket

- **Environments** – SQLHENV:
  En record av denne typen må opprettes av applikasjonen (klienten)
- **Connections** – SQLHDBC:
  Sørger for å knytte opp applikasjonsprogrammet mot databasen (tjeneren) og opprettholde forbindelsen til denne
  Hver connection eksisterer bare innen en environment
- **Statements** – SQLHSTMT:
  Holder informasjon om én SQL-setning, med implisitt tilhørende cursor hvis setningen er en query
  Hvert statement eksisterer bare innen en connection
- **Descriptions** – SQLHDESC:
  Holder informasjon om tupler og parametre
- For hvert statement dannes implisitt en mengde usynlige descriptions tilpasset statementet
  Eksplisitte descriptions knyttet til gitte statements deklareres ved behov
- Hver av de ovenstående er representert i applikasjonsprogrammet ved en ***handle*** = peker til tilhørende record
  En handle opprettes ved å kalle SQLAllocHandle()

C + SQL/CLI, eksempel

```
#include sqlcli.h
void worthRanges() {
  int i, digits, counts[15];
  SQLHENV myEnv;
  SQLHDBC myCon;
  SQLHSTMT execStat;
  SQLINTEGER worth, worthInfo;
  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &myEnv);
  SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
  SQLAllocHandle(SQL_HANDLE_STMT, myCon, &execStat);
  SQLPrepare(execStat, "SELECT netWorth FROM MovieExec", SQL_NTS);
  SQLBindCol(execStat, 1, SQL_INTEGER, &worth, size(worth), &worthInfo);
  while (SQLFetch(execStat) != SQL_NO_DATA) {
    digits = 1;
    while ((worth /= 10) > 0) digits++;
    if (digits <= 14) counts [digits]++;
  }
  for (i=0; i<15; i++) printf("digits = %d: number of execs = %d\n", i, counts[i]);
}
```


Parameteroverføring til queries i CLI

```
SQLPrepare(myStat,
  "INSERT INTO Studio(name, address) VALUES (?,?)",
  SQL_NTS);
SQLBindParameter(myStat, 1, ..., studioName, ...);
SQLBindParameter(myStat, 2, ..., studioAddr, ...);
SQLExecute(myStat);
```

Parameteren SQL_NTS betyr at SQLPrepare selv må beregne lengden på den midterste parameteren

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

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();
```


DBMS – Database Management System (repetisjon)

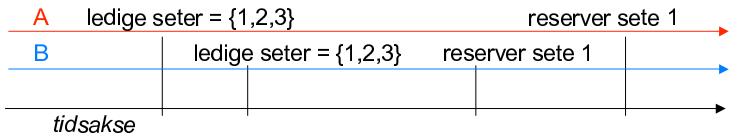
- Spesialisert SW
- Karakteristika:
  - Persistens
  - Transaksjonshåndtering
    - **A** tomicity
    - **C** onsistency
    - **I** solation
    - **D** urability
  - Programmeringsgrensesnitt

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

Transaksjoner i embedded SQL

- Hver SQL-setning er automatisk en transaksjon (jf. transaksjoner i det generiske SQL-interfacet)
- 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;

COMMIT og ROLLBACK

- **COMMIT**: Markerer at transaksjonen var vellykket
  - Fra BEGIN TRANSACTION og 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 BEGIN 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 en transaksjon bare foretar leseoperasjoner, kan vi tillate flere parallelle utførelser av den
  - Vi hjelper systemet med å optimalisere ved å markere transaksjonen som en ren lesetransaksjon
    EXEC SQL SET 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
    EXEC SQL SET TRANSACTION READ WRITE;
    (Se isolasjonsnivåer for når read-only er default og når read-write er det)

Dirty read (skitten les(?))

- **Dirty data** = data skrevet av en transaksjon som ennå ikke har gjort commit
- **Dirty read** = lesing av dirty data
  Man risikerer at transaksjonen som foretok skrivingen, senere gjør en rollback slik at dataene er ugyldige
- Noen ganger er det *ikke* kritisk om man leser dirty data og fra tid til annen risikerer at dataene blir forkastet
  Å tillate dirty reads vil øke gjennomstrømmingen av transaksjoner i systemet

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**
  - **Read-uncommitted**
  - **Read-committed**
  - **Repeatable-read**

Serializable

- **Serializable** = Det fins en eksekvering som gir samme resultat og som er seriell
  Dette er default modus for transaksjoner
  Kan eventuelt si dette eksplisitt:

```
EXEC SQL SET TRANSACTION
ISOLATION LEVEL SERIALIZABLE;
```

- Default for serializable er at transaksjonen er read-write

Read-uncommitted

- Read-uncommitted = dirty read

```
EXEC SQL SET TRANSACTION
ISOLATION LEVEL READ UNCOMMITTED;
```

- Default for read-uncommitted er at transaksjonen er read-only
  For å få read-write, skriv:

```
EXEC SQL SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;
```


Read-committed

- Ved **read-committed** er det forbud mot dirty-read
  Hvis en query foretas flere ganger innen samme transaksjon, kan imidlertid resultatet endres fra gang til gang

```
EXEC SQL SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;
```

- Default for read-committed er at transaksjonen er read-write

Repeatable-read

- Heller ikke ved **repeatable-read** er dirty-read tillatt
  Hvis man foretar en query flere ganger i samme transaksjon, kan man imidlertid oppleve at nye tupler kommer til
  (men de tuplene man har fått tilslag på, vil ikke forsvinne og ikke endres)

```
EXEC SQL SET TRANSACTION
ISOLATION LEVEL REPEATABLE READ;
```

- Default for repeatable-read er at transaksjonen er read-write

Sikkerhet og brukerautentisering i SQL

- Hver bruker har en **autorisasjonsID**
  Hver ID tilordnes **privilegier**:
  - **select**: Tillat select på noen attributter i en relasjon
  - **insert**: Tillat insert på noen attributter i en relasjon
  - **update**: Tillat update på noen attributter i en relasjon
  - **delete**: Tillat delete på en relasjon
  - **references**: Tillat bruk av en relasjon i en integritetsregel (assertion/attributtbasert/tuppelbasert)
  - **usage**: Tillat bruk av et databaseelement
  - **trigger**: Tillat definisjon av triggere mot en relasjon
  - **execute**: Tillat eksekvering av PSM-kodebiter
  - **under**: Tillat opprettelse av subtyper

Privilegier

- Privilegier tildeles ved
  grant <privilegier> **on** <databaseelement>
  to <brukere> [**with grant option**];
  og fratas ved
  revoke <privilegier> **on** <databaseelement>
  from <brukere> [**cascade** | **restrict**]