IN2090-bibel

Yrjar Vederhus
yrjarv@ifi.uio.no

Contents

Introduksjon og motivasjon

Databaser

Vi bruker databaser fordi:

- Data kan lagres persistent på disk
- Disk er billigere enn RAM
- Data skilles fra kode og kan brukes av flere programmer
- Enklere å operere på enn filer

Det finnes flere typer databaser:

- Dokument-databaser
- Key-value stores
- Grafdatabaser
- Relasjonelle databaser

Vi bruker bare relasjonelle databaser i dette emnet.

En relasjonell database er en samling tabeller (relasjoner). Hver relasjon har et navn, en samling kolonner, og en samling rader. En kolonne har et navn og en type.

SQL er et deklarativt spørrespråk, så du forteller hva du vil ha og ikke hvordan man kommer fram til det.

Datamodellering

En database inneholder kun data, og ikke informasjon. Informasjonen "oppdager" vi når vi henter ut data fra databasen. Hvis man lager et godt databaseskjema kan man forenkle denne prosessen.

For å lage en modell av et domene bruker man et modelleringsspråk, vi bruker ER. Et ER-diagram kan da senere oversettes til et godt databaseskjema.

Relasjonsmodellen

Relasjonsmodellen ble laget i 1970 av Edgar Codd, og ble ikke implementert ordentlig før 1977. Det er fordi det er vanskelig å lage et databasesystem som oppfyller kravene til en relasjonell database. Heldigvis er det enklere å bruke et slikt databasesystem enn det er å utvikle det.

Kort sagt er relasjonsmodellen er formell/matematisk beskrivelse av hvordan man kan representere data i tabeller.

Relasjon

En relasjon er en signatur og en mengde tupler. En relasjonsdatabase er da en mengde med relasjoner med et unikt navn.

Dette er et eksempel på en relasjon:

```
(
    Customers(
        CustomerID: int,
        Name: text,
        Birthdate: date,
        NrProducts: int
    ),
    {
        (0, John Doe, 2000-01-01, 1),
        (1, Jane Doe, 1814-05-17, 99),
        (2, John Smith, 2025-12-12, 0)
    }
)
```

Relasjonssignatur

En relasjonssignatur består av et navn og en mengde attributter/kolonner. Merk at hvorvidt vi typeannoterer attributtene er valgfritt.

Relasjonsskjema

Et relasjonsskjema er en mengde med relasjonssignaturer.

Attributt

Et attributt er en kolonne, og består av et navn og en type (noen ganger kalt et domene). To attributter i samme relasjon kan ikke ha samme navn.

Tuppel

Et tuppel er en rad (teknisk sett en mengde med par av attributtnavn og verdier), dermed er det ikke nøyaktig det samme som tuplene vi vanligvis ser: `{CustomerID: '0', Name: 'John Doe', Birthdate: '2000-01-01', NrProducts: '1'}` er for eksempel et gyldig tuppel i denne sammenhengden. For å aksessere antall produkter i tuppelet over (la oss kalle det `jd_t`), kan vi "kalle på" `jd_t.NrProducts`.

Noen ganger kalles et tuppel for en "instans".

Verdi

Verdier er "celler" i tabellen: `2000-01-01` er en verdi. I relasjonsmodellen må verdier alltid være atomære - altså kan de ikke f.eks. være arrayer. I `Postgres` kan man la verdier være ikke-atomære, men det unngår vi i dette emnet. En verdi kan også være `NULL` hvis det mangler en verdi i den "cellen".

Nøkler

En tommelfingerregel når vi designer en database er at det er en relasjon per entitet, en relasjon per forhold, og aldri noe duplisert data. Dette kan vi ikke alltid følge, men det er en god intuitiv ide å bruke når man designer en database. Dette kommer vi tilbake til senere.

Supernøkkel

En supernøkkel er en mengde med attributter som alltid har unike verdier en relasjon. Altså kan man bruke en supernøkkel for å identifisere en rad. En relasjon kan ha veldig mange forskjellige supernøkler, og hvis `{a}` er en supernøkkel vil også `{a, b}` og `{a, c}` være supernøkler. Supernøkler sier altså hva som alltid er unikt, ikke bare hva som er unikt i relasjonen akkurat nå. Derfor er bestemming av supernøkler avhengig av domenet, siden det bare beskriver virkeligheten og ikke dataene.

For eksempel har vi følgende supernøkler for en relasjon `Student(brukernavn, fornavn, etternavn, adresse)`:

- `{brukernavn}`
- `{brukernavn, fornavn}`
- `{brukernavn, fornavn, etternavn}`
- `{brukernavn, fornavn, etternavn, adresse}`
- `{brukernavn, etternavn}`
- `{brukernavn, etternavn, adresse}`
- `{brukernavn, adresse}` Merk at selv hvis det per nå ikke finnes noen studenter med samme fornavn er ikke `{fornavn}` en supernøkkel.

Kandidatnøkkel

En kandidatnøkkel er en minimal supernøkkel. Det kan finnes flere kandidatnøkler i en relasjon. En supernøkkel er en kandidatnøkkel hvis det ikke er mulig å fjerne et attributt fra den og fortsatt ha en supernøkkel.

I eksempelet over er for eksempel kun `brukernavn` en kandidatnøkkel.

Relasjonen `Kurs(emnekodeprefix, emnenummer, tittel, studiepoeng)` har `{emnekodeprefix, emnenummer}` og `{tittel}` som kandidatnøkler - men kun hvis det ikke finnes noen emner med samme navn i hele domenet.

Primærnøkkel (primary key, `PK`)

En primærnøkkel er en av kandidatnøklene. Dette velger du ut selv når du lager databaseskjemaet. Så når man har kandidatnøklerne `{emnekodeprefix, emnenummer}` og `{tittel}`, kan man selv velge å ha f.eks. `{tittel}` som primærnøkkel. Merk at hvis en relasjon kun har en kandidatnøkkel må naturligvis denne bli primærnøkkelen.

Fremmednøkkel (foreign key, FK)

Hvis en relasjon `a` refererer til en annen relasjon `b` i et av attributtene, vil det attributtet i `a` være en fremmednøkkel til et attributt i `b`.

```
{
    Student(brukernavn, navn, adresse),
    Karakter(student, emnekode, resultat),
    Emne(emnekode, tittel studiepoeng)
}
```

Her er `Karakter(student)` fremmednøkkel til `Student(brukernavn)`, og `Karakter(emnekode)` er til `Emne(emnekode)`.

En fremmednøkkel betyr at alle verdier i attributtet/attributtene som er fremmenøkkel må finnes i attributtet/attributtene som den/de er fremmednøkkel til. Altså må alle brukernavn i `Karakter(student)` finnes i `Student(brukernavn)`.

Merk for øvrig at `A(a, b) -> B(c, d)` ikke ekvivalent med `A(a) -> B(c)`, `A(b) -> B(d)`.

Notasjon

Kandidatnøkler understrekes gjerne, mens primærnøkler har dobbel understrek (hvis det finnes flere kandidatnøkler). Men man kan også bare skrive dem etter signaturen med vanlig tekst.

Relasjonsalgebra

Relasjonslagebra er et formelt (matematisk) språk for å jobbe med dataene som finnes i de ulike relasjonene.

Projeksjon (π)

π er unær, og har en mengde med attributter som subskrift. Det er tilsvarende `SELECT <subscript> FROM <argument>` i SQL. Den velger altså ut attributter fra relasjonen, og returnerer det.

$\pi_{\text{brukernavn,etternavn}}(\text{Student}) =$
`SELECT brukernavn, etternavn FROM Student`

Seleksjon (σ)

σ er også unær, og har et uttrykk med attributt-navnene som variabler som subscript. "Lovlige" symboler i uttrykket er $\wedge$, $\vee$, $\neg$, $\geq$, $\leq$, $>$, $<$, og $=$, samt konstanter. Tilsvarende `WHERE` i `SELECT`-spørringer i SQL. Dette brukes for å velge ut de radene vi er interessert i.

$\sigma_{\text{emnenummer} \geq 3000 \wedge \text{studiepoeng}=10}(\text{Emne}) =$
`SELECT * FROM Emne WHERE emnenummer >= 3000 AND studiepoeng = 10`

Omdøping (ρ)

Igjen, unær. Omdøper attributter til noe annet.

$\rho_{\text{emnekode}\to\text{ek},\text{studiepoeng}\to\text{poeng}}(Emne)$. Har ingen ekvivalent i SQL da...

Kartesisk produkt ($\times$)

Cross join, som `SELECT * FROM A, B`. Returnerer alle attributtene til begge relasjonene og alle kombinasjoner av tupler fra de to relasjonene.

Student $\times$ Emne =
`SELECT * FROM Student, Emne`

Join ($\bowtie$)

Kombinerer to tabeller på et attributt (eller flere) slik som `INNER JOIN` gjør.

Student $\bowtie_{\text{brukernavn}=\text{student}}$ Karakter =
`SELECT * FROM Student INNER JOIN Karakter ON brukernavn = student`

Naturlig join ($\star$)

Som vanlig join, men den gjør det automatisk på alle attributter med likt navn.

Student $\star$ $(\rho_{\text{student}\to\text{brukernavn}}(Karakter))$

Snitt, union, mengdedifferanse ($\cap$, $\cup$, $\setminus$)

Akkurat som forventet, gjør mengdeoperasjoner på mengder av tupler.

Henting av data med SQL

SQL brukes både til uthenting av data og til manipulering av data - og til og med til konfigurering av selve databasen..

Det finnes følgende nøkkelord (første ord i en spørring) i SQL:

- `SELECT` - henter informasjon
- `CREATE` - lager noe (f.eks. en tabell, et view, etc.)
- `INSERT` - setter inn rad(er) i en tabell
- `UPDATE` - oppdaterer data i en tabell
- `DELETE` - sletter rader fra en tabell
- `DROP` - sletter en hel ting (f.eks. en tabell, et view, etc.)

Rekkefølgen er som følger:

```
WITH <navngitte-spørringer>
SELECT <kolonner>
FROM <tabeller>
WHERE <uttrykk>
```

```
GROUP BY <kolonner>
HAVING <uttrykk>
ORDER BY <kolonner> [DESC]
LIMIT <N>
OFFSET <M>
```

`LIMIT` og `OFFSET` er de eneste som kan bytte plass. Man kan selvfølgelig også droppe enkelte klausuler, men man trenger `GROUP BY` for å ha `HAVING`

`WHERE`-klausulen (ved bruk av `SELECT`)

Finne alle rader i `table` hvor et gitt attribute heter er `'desired value'`

```
SELECT * FROM table WHERE attribute = 'desired value'
```

Finne alle rader hvor attributen slutter på `value`

```
SELECT * FROM table WHERE attribute LIKE '%value'
```

Eventuelt kan man bruke `SIMILAR TO` for en rar kombo av `LIKE`-syntaks og regex, eller `~` for regex. `LIKE` er alltid raskt, mens regex er mer risikabelt siden det kan ha ekstremt dårlig ytelse.

Bruk `NOT` for negasjon.

`NULL`

Alle uttrykk med `NULL` (bortsett fra `OR`) oppfører seg rart. Derfor bruker man `IS NULL` eller `IS NOT NULL` for å sjekke om et attributt har verdien `NULL`.

```
NULL AND TRUE -- NULL
NULL OR FALSE -- NULL
NULL AND FALSE -- FALSE
NULL OR TRUE -- TRUE
10 + NULL -- NULL
```

`SELECT`-klausulen

Man kan selecte kolonnenavn, eller gjøre utregninger etc.

```
SELECT x, y * 5 FROM z; -- Matte
SELECT fname || ' ' || lname FROM people; -- Strengkonkatenering
SELECT x AS y, y AS z, z * 5 AS x FROM table; -- Navngiving
SELECT DISTINCT x, y, z FROM table; -- Fjerning av duplikate rader
```

Aggregering

Vi har følgende aggregeringsfunksjoner som kan brukes i SELECT-klausulen:

- `avg` (gjennomsnitt)
- `max` (maksimum)

- `min` (minimum)
- `count` (antall rader)

De brukes slik:

```
SELECT agg_funk(x), y FROM table;
```

Da navngis kolonnen som aggregeringsfunksjonen har blitt brukt i likt som navnet på aggregeringsfunksjonen. For eksempel vil `SELECT count(*) FROM table` returnere en tabell med en rad og en kolonne, den kolonnen heter da `count` med mindre vi spesifiserer noe anned med `AS`.

`count` Merk at `count(*)` og `count(x)` - der `x` er en kolonne - gjør to forskjellige ting: `count(*)` teller antall rader, mens `count(x)` teller antall rader hvor `x` har en ikke-`NULL`-verdi.

`FROM`-klausulen

Vanligvis `SELECT`er vi `FROM` en tabell. Men vi kan også gi den flere tabller, i så fall får vi kryssproduktet av de to tabellene. Derfra kan vi legge på en `WHERE` for å begrense oss til f.eks. de radene hvor en attributt fra hver av kolonnene er like.

Joins

Det finnes mange type joins:

```
SELECT * FROM x, y; -- Cross join
SELECT * FROM x, y WHERE x.a = y.b; -- Equi-join
SELECT * FROM x, y WHERE <theta>(x.a, y.b) -- Theta-join
```

Merk at equi-join er en type theta-join, siden `=` er en funksjon.

Alle disse typene kalles inner joins.

Hvis det er like kolonnenavn refererer vi til dem slik:

```
SELECT * FROM x, y WHERE x.a = y.a;
-- ELLER
SELECT * FROM long_name_x AS x, long_name_y AS y WHERE x.a = y.a;
```

Inner joins Inner joins kan gjøres med en egen notasjon:

```
SELECT *
FROM x INNER JOIN y
    ON x.a = y.a
```

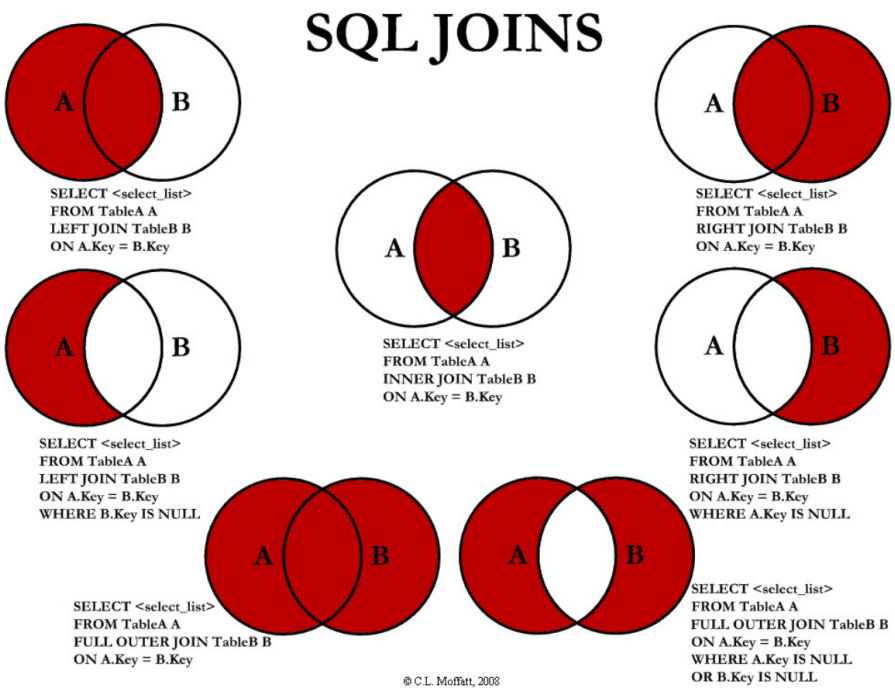
Self joins Dette er også mulig, men man må da huske å navngi de to "forskjellige" brukene av samme tabell forskjellig.

Naturlig join Gjøres med `NATURAL JOIN`. Joiner automatisk på alle kolonner med likt navn og projiserer vekk dupliserte kolonner. Hvis det er noen kolonner med forskjellige navn man vil at skal naturlig joines over må man navngi disse likt, og omvendt hvis det er to kolonner med samme navn som man ikke vil joine på.

Ytre joins Ytre joins bevarer alle rader fra en eller begge tabellene i joinen. Det gjøres med `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, og `FULL OUTER JOIN`. Der det ikke finnes noen match fylles nemlig med `NULL`.

Med `LEFT OUTER JOIN` vil alle rader i den venstre tabellen bli med i svaret, også hvor det ikke finnes en "tilhørende" rad i den høyre tabellen. Tilsvarende gjelder for `RIGHT OUTER JOIN`.

`FULL OUTER JOIN` er en kombinasjon av `LEFT OUTER JOIN` og `RIGHT OUTER JOIN`, som lar absolutt alle rader være med - selv de som ikke "hører sammen".



Merk også at `LEFT OUTER JOIN` = `LEFT JOIN`, `RIGHT OUTER JOIN` = `RIGHT JOIN`, og `FULL OUTER JOIN` = `FULL JOIN`.

Nøstede spørringer

Man kan putte en spørring inne i `FROM`:

```sql
SELECT x, y
FROM (SELECT x, y FROM table WHERE x = 1) AS query
WHERE query.y = 2
```

`WITH`

Man kan oppnå noe av det samme med `WITH`:

```sql
WITH
    query AS (SELECT x, y FROM table WHERE x = 1)
SELECT x, y FROM query WHERE query.y = 2
```

Sortering

`ORDER BY <kolonner>`

```sql
SELECT article_num, product, price
FROM catalogue
ORDER BY price
```

Standard er ascending (minst til størst), men man kan bruke `ORDER BY <kolonne> DESC, <kolonne2> DESC` for å få descending sortering (størst til minst).

Man sorterer først etter den første kolonnen, og deretter etter den andre. Så det er den siste kolonnen som har "mest" påvirkning på resultatet.

Begrense antall rader i resultatet og `OFFSET`

`LIMIT <number>`. Begynner seinere med `OFFSET <number>`

Grupper

`GROUP BY <kolonner>` grupperer radene i henhold til likhet på verdiene i disse kolonnene. Bruk av agreggeringsfunksjoner på gruppene i `SELECT`-klausulen skjer som vanlig, men da blir jo f.eks. `count(*)` brukt innad i hver gruppe.

Man kan bare bruke de grupperte kolonnene utenfor en aggregatfunksjon i `SELECT`-klausulen.

```sql
SELECT category, avg(price) AS average_price
FROM products
GROUP BY category
```

Hvis man grupperer på flere kolonner, blir det gruppert etter likhet på begge kolonnene. Så det blir ikke noen "inndeling" internt i gruppene, men heller en helt egen gruppering.

`HAVING`

I stedet for delspørringer kan vi f.eks. hente alle kategorier med flere enn 10 produkter ved å bruke `HAVING`:

```
SELECT category_name, nr_products
FROM (
    SELECT c.category_name, count(*) as nr_products
    FROM categories as c
        INNER JOIN products as p ON (c.category_id = p.category_id)
    GROUP BY c.category_name
)
WHERE nr_products > 10
```

```
SELECT c.category_name, count(*) AS nr_products
FROM categories as c
    INNER JOIN products as p ON (c.category_id = p.category_id)
GROUP BY c.category_name
HAVING count(*) > 10
```

Bruk av navn

Navnene fra `WITH` kan brukes i alle de etterfølgende spørringene. Navnene fra `SELECT` kan brukes i `ORDER BY` og alle ytre spørringer. Navnene fra `FROM` kan brukes i alle klausuler utenom samme `FROM`-klausul.

Mengdeoperasjoner

Man kan putte mengdeoperasjoner mellom to `SELECT`-spørringer:

```
SELECT 1, 2
UNION
SELECT 3, 4
```

```
| ? | ? |
| 1 | 2 |
| 3 | 4 |
```

Vi har følgende mengdeoperatorer:

- `UNION` (union)
- `INTERSECT` (snitt)
- `EXCEPT` (mengdedifferanse)

I tillegg kan vi legge til `ALL` på slutten av alle disse hvis man ønsker å tillate duplikater. Antall ganger en gitt rad er med i resultatet er som følger:

- `UNION ALL` - Summen av antall ganger raden var med i begge spørringene
- `INTERSECT ALL` - Det minste antall ganger raden er med i begge

- `EXCEPT ALL` - Antall ganger raden er med i den ene, minus antall ganger den er med i den andre (eller 0)

Union-kompatabilitet

Kolonnene i tabellene som man gjør mengdeoperasjoner på må ha kompatible typer, og det må være like mange kolonner.

`EXISTS`

Hvis vi vil sjekke om en spørring har et svar kan vi bruke `EXISTS <spørring>`:

```
SELECT p.name
FROM persons AS p
WHERE NOT EXISTS (
    SELECT *
    FROM companies AS c
    WHERE c.country = p.country
) -- Henter alle navn på personer i land hvor det ikke er noen bedrifter
```

Datamanipulering med SQL

SQL har mange ulike underspråk. DQL (Data Query Language) bruke vi for å hente data, og nå skal vi bruke DML (Data Manipulation Language) for å manipulere data. Med "manipulere data" mener vi innsetting, endring, og sletting.

Lage ting

```
CREATE <type_ting> <navn> <evt_mer>;
CREATE SCHEMA northwind
CREATE TABLE table (<kolonner>);
```

For å lage tabeller spesifiserer vi hver kolonne slik: `<type> <navn> <eventuelle skranker>,`.

`SERIAL` er en "type" som automatisk genererer en ny int for hver rad som settes inn, og det hindrer da duplikater på egenhånd. Man skal ikke oppgi en verdi for `SERIAL`-kolonner når man setter inn i en tabell.

Skranker

- `NOT NULL` - krever at en kolonne ikke inneholder `NULL`-verdier
- `UNIQUE` - krever at det ikke er noen duplikater i en kolonne
- `PRIMARY KEY` - kombinerer både `UNIQUE` og `NOT NULL`. Hver tabell kan kun ha en primærnøkkel, så hvis det skal være flere kandidatnøkler må man bruke `UNIQUE NOT NULL`.

- `DEFAULT` er ikke egentlig en skranke, men lar deg spesifisere en default-verdi med `name type DEFAULT 'value'`
- `REFERENCES` - brukes for FK, og gjør så alle verdier i kolonnen må allerede finnes i kolonnen den refererer til:

```
CREATE table takesCourse (
    studentId INT REFERENCES students(studentId),
    courseId INT REFERENCES courses(courseId),
    semester TEXT
)
```

Man kan også definere skranker nederst i `CREATE`, for eksempel hvis man vil gjøre en kombinasjon av attributter til PK:

```
CREATE TABLE students (
    studentId int,
    name text NOT NULL,
    birthdate date,
    CONSTRAINT students_pk PRIMARY KEY (studentId, birthdate)
)
```

I eksempelet over er `{studentId, birthdate}` PK for `students`-relasjonen.

En spesiell type skranke er `CHECK`-skranker. De lar oss bruke et generelt uttrykk for å avgjøre om verdier kan settes inn i kolonnen eller ikke. De brukes akkurat likt som alle andre skranker, men har syntaks `CHECK (<UTTRYKK>)`.

Innsetting av data

```
INSERT INTO people
VALUES (0, 'John', 'Doe'),
       (1, 'Jane', 'Doe')
```

Vi kan også bruke resultatet fra en `SELECT`-spørring:

```
INSERT INTO early_people
SELECT * FROM all_people WHERE id < 100
```

I tillegg er det mulig å lage en table slik:

```
CREATE TABLE early_people AS
SELECT * FROM all_people WHERE id < 100
```

Da må vi legge til skranker seinere.

Innsetting fra CSV-fil

Ikke sikkert at dette er relevant, men dette er hvordan man kopierer fra en CSV-fil:

```
COPY tabell
FROM '/path/to/file.csv' DELIMITER ',' NULL AS '';
```

Eventuelt kan man bruke `psql` til å lese fra `stdin` hvis man bytter ut fil-path-stringen med `stdin`.

Slette ting

```
DROP <TYPE> <NAVN>;
DROP SCHEMA northwind;
```

Hvis man vil slette en tabell og alle tabellene som inneholder referanser til den, bruker man `DROP TABLE table CASCADE`.

Slette data

```
DELETE FROM table
WHERE condition
```

Oppdatere ting

```
ALTER <TYPE> <NAVN>
<ACTION>
```

`<ACTION>` kan da f.eks. være `RENAME TO new_name`, `ADD COLUMN new_col <TYPE>`, eller `ADD CONSTRAINT constr_name PRIMARY KEY (column_name)`

Oppdatere data

```
UPDATE table
SET column = 'value'
WHERE <CONDITION>
```

Typer

Det er mange innebygde typer, og man kan definere egne typer (ikke pensum).

Casting

Ved å prøve å sette inn en string med et tall i en kolonne som skal være int castes det automatisk: `INSERT INTO numberlist VALUES ('1')` blir ekvivalent med `INSERT INTO numberlist VALUES (1)`.

Vi kan også caste eksplisitt: `value::type` (`'1'::int`), `cast(value AS type)` (`cast('1' AS int)`), eller `type value` (`int '1'`).

Numerisk

Vi har følgende "tall" (numeriske typer):

Navn	Størrelse	Mulige verdier
`smallint`	2 bytes	Heltall -32768 til 32767
`integer` / `int`	4 bytes	Heltall -2147483648 til 2147483647
`bigint`	8 bytes	Heltall -9223372036854775808 til 9223372036854775807
`decimal`	Varierer	Opp til 131072 tegn før komma, opp til 16383 etter
`numeric`	Varierer	Opp til 131072 tegn før komma, opp til 16383 etter
`real`	4 bytes	6 desimaltegn presisjon
`double precision`	8 bytes	15 desimaltegn presisjon
`smallserial`	2 bytes	1 til 32767
`serial`	4 bytes	1 til 2147483647
`bigserial`	8 bytes	1 til 9223372036854775807

Alle numeriske typer har følgende funksjoner:

- `ceil(number)` - runder opp
- `floor(number)` - runder ned
- `sqrt(number)` - kvadratrot
- `power(base, exponent)` - potens
- `sin(number)`, `cos(number)`, `tan(number)` - trigonometriske funksjoner
- `random()` - genererer et tilfeldig tall ≥ 0 og < 1

Tekst

Navn	Beskrivelse
`varchar(n)`	Tekst med maks lengde `n`
`char(n)`	Tekst med lengde `n`, paddes med "blanks" (`0x00`)
`text`	Tekst med automatisk tilpasset lengde

Man kan bruke følgende funksjoner på strenger:

- `position('substring' in 'string')` - Finner posisjonen til `substring` i `string`
- `substring('string' from <START_INDEX> for <LENGTH>)` - Henter ut en substring av `string` fra index `<START_INDEX>` med lengde `<LENGTH>`

Tid

Navn	Størrelse	Beskrivelse
`timestamp`	8 bytes	Dato og tid, uten tidssone
`timestamp with time zone` / `timestamptz`	8 bytes	Dato og tid, med tidssone
`date`	4 bytes	Dato (ikke tid)
`time`	8 bytes	Tid uten tidssone, uten dato
`time with time zone`	12 bytes	Tid med tidssone, uten dato
`interval`	16 bytes	Tidsintervall

Datoer kan oppgis på følgende formater:

- `YYYY-MM-DD`
- `Month DD, YYYY`
- `YYYY-Mon-DD`
- `DD-Mon-YYYY`
- `YYYYMMDD`
- `YYMMDD`
- `YYYYXXX` der `XXX` er dag-nummer i året

I tillegg støttes noen andre formater i ulike moduser, men disse er trygge.

Tid kan oppgis på følgende formater:

- `HH:MM:SS.MSS`
- `HH:MM:SS`
- `HH:MM`
- `HHMMSS`
- `HHMM [A/P]M`
- Alle formatene over etterfulgt av tidssone-forkortelse, offset mot Zulu i `HH` eller `HH:MM`, eller fullt navn på tidssone

Intervaller Man kan bruke vanlig pluss og minus med `interval '<NUMBER> <UNIT>'`.

Hente ut deler fra en tidstype `extract(<UNIT> from <VALUE>)`

Andre funksjoner

- `now()` brukes for å gi et `timestamp` med nåværende tidspunkt
- `current_date` er en konstant som inneholder dagens dato.
- `OVERLAPS` mellom to par av tidstyper sjekker om "intervallene" parene representerer overlapper:

```
SELECT (DATE '2025-01-01', DATE '2025-08-01') OVERLAPS
       (DATE '2025-07-01', DATE '2025-12-31')
```

(dette returnerer `true`)

Diverse andre typer

Postgres støtter bl.a. også arrayer, XML, JSON, og bitstrenger.

Views

Views er en måte å "gjenbruke" queries:

```
CREATE VIEW viewname (colname1 <TYPE>, colname2 <TYPE>)
AS <SELECT_QUERY>
```

Deretter kan dette viewet brukes som om det er en vanlig tabell, og innholdet beregnes på nytt hver gang det brukes. Så det tar ikke opp noen plass, men gir heller ingen ytelses-fordeler.

Views kan for eksempel brukes for å vise utledbare verdier.

Materialiserte views

Materialiserte views lages med `CREATE MATERIALIZED VIEW` i stedet for `CREATE VIEW`, og må oppdateres manuelt med `REFRESH MATERIALIZED VIEW viewname`. Materialiserte views lagres som om de var tabeller, men har ingen "egen" data. Derfor bruker de litt mer plass, men hvis man må kjøre samme spørring ofte kan de gi en brukbar ytelsesboost.

SQL-scripts og transaksjoner

Man lager ofte et script som oppretter hele databasen. Dette kan kjøres med `psql <vanlige innloggings-flagg> -f </path/to/script.sql>` eller fra `psql`-shellet med `\i /path/to/script.sql`. For å opprette et script som gjenskaper databasen kan man bruke `pg_dump <vanlige_innloggings_flagg> db > /path/to/backup/script.sql`.

Trygge kommandoer

For å hindre at `CREATE TABLE` eller `DROP TABLE` feiler hvis hhv. en tabell allrede finnes og ikke finnes, bruker vi `CREATE TABLE IF NOT EXISTS` og `DROP TABLE IF EXISTS` i mange scripts.

Meta-kommandoer

`\echo` kan gi output til terminalen, `\set varname 'value'` setter en variabel som kan brukes med `:varname` senere i scriptet.

Transaksjoner

En transaksjon gjør at hvis en spørring feiler så feiler hele greia, og ingen endringer har da blitt gjort permanent. Start en transaksjon med å skrive `BEGIN;` og "commit" endringene når du er ferdig: `COMMIT;`.

En god transaksjon burde være atomær, konsistent, isolert, og holdbar. Det viktigste der er isolert: Hver transaksjon skal kunne kjøres parallelt med de andre transaksjonene uten at det blir noen problemer.

ER-diagrammer (datamodellering)

ER er et visuelt modelleringsspråk som vi bruker for modellering av domener. Det består av 3 typer bokser og streker mellom disse: Entiteter, attributter, og relasjoner.

Entiteter og attributter

Entiteter tegnes som rektangler. Det representerer en konkret ting. De skal alltid ha et unikt navn, og har verdier (attributter) knyttet til seg. Ofte brukes begrepene entitet og entitetstype om hverandre.

Attributter tegnes som ovaler og knyttes til nøyaktig 1 entitetstype med en strek. De må ha et unikt navn innenfor samme entitet. Alle entiteter kan ha en verdi knyttet til attributten, og alle entiteter kan kun ha verdier knyttet til en av entitetstypens attributter.

Nøkkelattributter har en **understrek** under navnet. Dette er attributter som må være unike, og som alle entiteter av den angitte entitetstypen må ha en verdi for.

Flerverdiattributter er attributter hvor hver entitet kan ha flere verdier. Disse markeres med en **dobbel oval**

Relasjoner

Relasjoner tegnes med diamanter. De representerer forhold mellom entiteter. De har et unikt navn, og relaterer 2 eller flere entiteter. Strekene mellom relasjonen og entitetstypene kalles *deltakelser*

Attributter i relasjoner

Relasjoner kan også ha attributter, det gir verdier som knyttes til forholdet mellom entitetene. Her kan man ikke ha nøkler.

Øvre skranker

Øvre skranker kan enten være 1 eller mange (typisk `N` eller `M`). Tallet/bokstaven **lengst vekk fra en entitet** sier hvor mange den entiteten høyst kan være relatert til.

Nedre skranker

Nedre skranker skiller vi på om er "minst 0" eller "minst 1". Dette representeres med hhv. **enkel og dobbel linje**. Streken **nærmest en entitet** sier antall den entiteten minst må være relatert til.

Svake entiteter

En svak entitet er en entitet som har en nøkkel som avhenger av en en annen entitets nøkkel. Altså har en svak entitet en nøkkel som kun er unik i en kontekst (gitt ved en relasjon til en annen entitet). Dette gjøres ved å **merke nøkkelen med stiplet underlinje og gi entiteten en dobbel boks**.

hvilken relasjon det er som angir den nødvendige konteksten markerer vi med **dobbel diamant**. Den relasjon kalles indentifiserende/relasjon.

VIKTIG Svake entiteter må alltid være relatert til nøyaktig 1 via den identifiserende relasjonen (dobbel linje nærmest den svake entiteten, og 1 lengst unna den)

Det er også mulig for en svak entitet å avhenge av flere entiteter/relasjoner.

Ternære og N-ære relasjoner

Det kan finnes N-ære relasjoner, men som regel forholder vi oss til ternære relasjoner (relasjoner som relaterer 3 entiteter av gangen). De samme prinsippene gjelder for ternære og N-ære relasjoner.

Ternære relasjoner er like som binære med at kombinasjonen av de relaterte elementene er unike, og at relasjonen kan ha attributter. Den største forskjellen er altså at de alltid relaterer 3 entiteter.

Øvre skranker

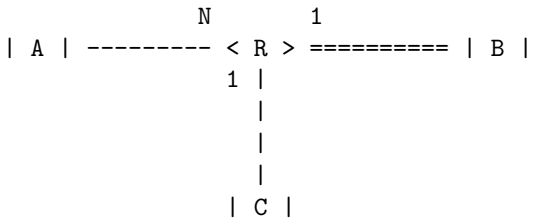
Øvre skranker sier hvor mange entiteter vi kan ha av 1 entitetstype, gitt 1 kombinasjon av entiteter fra de andre entitetstypene. Tallet nærmest en entitet sier hvor mange av den som maks kan bli relatert til av hvert par av to andre entiteter. Sagt annerledes er det det tallet lengst unna et par av entiteter som sier hvor maks hvor mange ulike av den tredje som dette paret kan være relatert til.

Så kort sagt: Tallet lengst unna et par av entitetstyper angir maksimalt hvor mange forskjellige av den entitetstypen et gitt par kan relateres til.

Nedre skranker

Nedre skranker sier hvor mange entiteter vi må ha av en entitetstype i relasjonen. Det tegnes som alltid på den linja som er nærmest den entitetstypen vi jobber med.

Så kort sagt: Linja nærmest en entitet sier hvor mange av den entiteten som minst må være med i den relasjonen.



Relasjonen over har altså at:

- Hvert mulige par (A, C) skal være relatert til nøyaktig 1 entitet B
- Hvert mnulige par (B, C) kan være relatert til 0 eller flere entiteter A
- Hvert mulige par (A, B) kan være relatert til 0 eller 1 entitet C.

Binær vs ternær

Merk for øvrig at en ternær relasjon ikke er ekvivalent med 3 binære. Binære relasjoner kan imidlertid representeres med en svak entitet, som relateres til hver av de andre entitetene. Dette gjelder kun hvis de øvre skrankene er "ubegrenset".

Realisering

Realisering er prosessen hvor man oversetter en ER-modell til et relasjonsskjema.

Realiseringsalgoritmen er komplisert og krever menneskelige valg, og derfor gjør vi den manuelt.

Vi gjør realisering i følgende rekkefølge:

1. Entiteter
2. Svake entiteter
3. 1-1-relasjoner
4. 1-N-relasjoner
5. M-N-relasjoner
6. Flerverdi-attributter
7. Ternære/N-ære relasjoner

Attributter på relasjoner blir bare attributter i relasjonens realisering, men blir aldri en del av kandidatnøkkel/PK.

Entiteter

Entiteter blir relasjoner. Vi ignorerer utledbare attributter og venter med flerverdi-attributter. Attributter blir da relasjons-attributter. Nøkler blir kandidatnøkler, og vi må velge en PK.

Svake entiteter

Svake entiteter blir også relasjoner. Attributter blir i tillegg PK til den/de identifiserende entitet/entitetene. PK blir de identifiserendes PK pluss den svake nøkkelen. Deretter setter vi FK til å referere til de identifiserende relasjonenes PK.

1-1-relasjoner

Her har vi følgende alternativer:

Sammensmeltning

Vi lager en ny relasjon som er sammensmeltningen av de to relasjonene. Den får da alle deres attributter (man må manuelt velge PK). Man fjerner de gamle attributtene og oppdaterer FKer. Dette alternativet kan **kun brukes hvis alle deltakelsene mellom de to entitetene er totale**.

FK fra den ene relasjonen

Vi legger til attributter i den ene relasjonen tilsvarende den andre relasjonens PK, og markerer disse som FK. Hvis den første relasjonen har total deltakelse blir FK-ene også kandidatnøkler.

Ny relasjon

Man kan også lage en ny relasjon med referanser til hver av de andre relasjonenes PK. Da får den nye relasjonen kandidatnøkkel lik hver av disse PK-ene, og en PK må da velges. Til slutt legger vi til FK til hver av disse PK-ene.

1-N-relasjoner

Her har vi igjen to valg:

Legge FK inn i N-siden

Tilsvarende som for 1-1: Vi lager en FK i N-siden som refererer til 1-siden.

Ny relasjon

Igjen, tilsvarende som for 1-1, men N-sidens PK blir alltid den nye relasjonens PK.

N-M-relasjoner

Her må vi alltid lage en ny relasjon. Dette gjøres igjen som i 1-1 og 1-N, men PK blir da lik begge PK-ene fra de relaterte relasjonene.

Flerverdi-attributter

De blir en egen relasjon som relaterer entitetens PK til attributten. Kandidatnøkkel/PK er da lik alle attributtene i hele relasjonen. I tillegg har vi da en FK til entitetens PK.

Ternære relasjoner

Likt som M-N-relasjoner: Vi lager alltid en egen relasjon. I relasjonens PK har vi kun de attributtene som kommer fra N-"sider" - det betyr at i en ternær N-M-O-relasjon er alle attributtene i PK, mens i en N-M-1-relasjon er det kun attributtene som kom fra N- og M-sidene som blir med i PK.

Databasedesign

Man burde ikke putte all data i samme tabell, siden vi da ikke kan ha separate oversikter over ulike "ting": I en tabell med personinfo, emneinfo, og karakterinfo, risikerer man å slette studenter hvis man sletter et emne, eller omvendt. I tillegg er det vanskelig å sette inn en student uten å sette inn et emne og en karakter. Alt dette kalles anomalier.

Et bra databaseskjema har derfor ulike "ting" i ulike tabeller, og relaterer dem til hverandre med nøkler.

Funksjonelle avhengigheter

Funksjonelle avhengigheter (FD-er) en rent semantisk greie, og kan ikke uttrykkes med verken ER eller databaseskjemaer. De skrives $X \to Y$, som betyr at Y (eller alle attributter i mengden Y) kan utledes fra X (eller alle attributtene i mengden X).

FD-er er transitive, så hvis $A \to B$ og $B \to C$ blir det at $A \to C$.

Vi kan bruke FD-er til å bestemme hvilke supernøkler og kandidatnøkler vi kan ha.

Tillukninger

Tillukningen X^+ av X, der X er en mengde attributter, er mengden attributter som er funksjonelle avhengige av X.

Finne kandidatnøkler

- Hvis A ikke forekommer i noen høyreside, er A med i **alle** kandidatnøkler.
- Hvis A forekommer i minst en høyreside, men ingen venstresider, er A **ikke del** av en kandidatnøkkel.

Så vi begynner med å gradvis bygge opp delmengder av attributter, men vi starter med attributter som ikke forekommer på høyre side. Da beregner vi tillukningen - hvis tillukningen er alle attributter burde vi sjekke minimalitet. Hvis ikke utvider vi gradvis med ett og ett nytt attributt.

Normalformer

BCNF $\subseteq$ 3NF $\subseteq$ 2NF $\subseteq$ 1NF

Høyere normalform gir færre anomalier, men også flere tabeller. Gitt et skjema finnes det alltid en algoritme som lager et ekvivalent skjema på den NF man ønsker.

Hvis alle tabeller i et skjema oppfyller kravene til en gitt normalform, oppfyller hele skjemaet den normalformen.

1NF

1NF er den enkleste, og krever kun at alle attributter er **atomære**. Det betyr egentlig bare at de ikke kan inneholde "komplekse" datatyper, sånn at man ikke må gjøre noe manipulering av datastrukturer eller strenger for enkle operasjoner.

I dette emnet antar vi alltid at 1NF er oppfylt.

2NF

Ingen attributter som ikke er nøkkelattributter kan være funksjonelt avhengige av kun en delmengde av en kandidatnøkkel.

Så kort sagt: Alt som ikke er i en kandidatnøkkel skal kun være avhengig av en hel kandidatnøkkel - aldri av kun deler av kandidatnøkkelen.

I tillegg må tabellen være på 1NF.

Så en tabell bryter 2NF hvis det finnes et ikke-nøkkelattributt som er avhengig av en delmengde av en kandidatnøkkel.

3NF

Alle ikke-nøkkelattributter skal kun være avhengige av kandidatnøkler.

Dette utelukker altså tilfellet hvor A er en kandidatnøkkel, og vi har FD-ene $A \to B$, $B \to C$. Siden B kun er avhengig av hele kandidatnøkkelen og C kun er avhengig av B (som ikke er en delmengde av kandidatnøkkelen), er dette på

2NF. Men siden C er avhengig av B som ikke er en kandidatnøkkel, er ikke dette på 3NF.

BCNF (Boyce-Codd normalform)

Alle attributter skal kun være avhengig av en kandidatnøkkel.

Altså fjerner vi unntaket for nøkkelattributter som er i 3NF. Det betyr at som regel er 3NF-tabeller også på BCNF - siden det unntaket er lite brukt.

Så kort sagt skal absolutt alle attributter være avhengige av en kandidatnøkkel, hele kandidatnøkkelen, og bare kandidatnøkkelen. Huskeregelen for dette er "The key, the whole key, and nothing but the key".

Bestemme normalform

Vi starter med å splitte opp høyresidene slik at det kun er 1 attributt på hver høyreside.

Deretter fjerner vi alle redundate FD-er: Hvis $A \to B$ så vil $AX \to B$ for alle X. Og dette trenger vi da ikke å ha med.

Så, for hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel: Hvis ja er FD-en på BCNF og man kan gå videre til neste FD. Hvis ikke er den et brudd på BCNF, og vi går videre til 2.
2. Er A et nøkkelattributt: Hvis ja er FD-en på 3NF og man kan gå videre til neste FD. Hvis ikke er den et brudd på 3NF, og vi kan gå videre til 3.
3. Er X ikke en del av en kandidatnøkkel: Hvis ja, er FD-en på 2NF og man kan gå videre til neste FD. Hvis den er det, er den et brudd på 2NF og skjemaet er på 1NF.

Tapsfri dekomposisjon

Tapsfri dekomponering er en måte å lage et nytt skjema på BCNF fra et skjema som ikke er på BCNF.

En dekomponering av $R(X, Y, Z)$ til $S_1(X, Y), S_2(X, Z)$ er tapsfri hvis og bare hvis $X \to Y$. Altså kan man skille ut noen attributter og det de alle er avhengige av.

Dekomponeringsalgoritmen

Tapsfri dekomponering av $R(X)$ med FDer F:

1. Beregn alle nøklene til R (ved hjelp av F)
2. Splitt alle FDer i F slik at det kun er ett attributt på høyresiden av hver FD (f.eks. $A, B \to C, D$ blir $A, B \to C$ og $A, B \to D$).
3. Sjekk om R bryter med BCNF. Hvis R er på BCNF, stopp og returner R.
4. Finn 1 FD $Y \to A$ i F som bryter med BCNF

5. Beregn tillukningen av Y med hensyn på FDene i F
6. Dekomponer R til $S_1(Y^+)$ og $S_2(Y, X \setminus Y^+)$
7. Fortsett rekursivt over S_1 og S_2

Design i praksis

Man kan designe seg frem til cirka BCNF uten å dekomponere: Hvis man har en tabell per entitetstype slik at ett tuppel er en entitet, og at relasjoner ellom entiteter enten representeres via FK fra en entitetstabell til en annen eller via egne tabeller.

Realiseringsalgoritmen kan aldri garantere noe mer enn 1NF siden ER ikke kan fange opp alle FDer, men ofte blir det BCNF.

Samtidig er ikke BCNF alltid idealet, siden dekomponeringsalgoritmen kan gi et BCNF-skjema som lager to tabeller som burde vært en.

Programmering med databaser

For å interagere med Postgres fra Python bruker man biblioteket `psycopg2`. Man lager et `Connection`-objekt, og fra det lager man `Cursor`-objekter. Disse kan kjøre spørringer via `exequte(query)`. Da kan man hente ut spørringene som lister av tupler med `fetchall()`.

Eksempel fra innlevering 5:

```
import psycopg2

user = 'yrjarv'
pwd = 'hunter2' # https://knowyourmeme.com/memes/hunter2

connection = \
    "dbname='" + user + "' " +  \
    "user='" + user + "' " + \
    "port='5432' " +  \
    "host='postgres.yrjar.tech' " + \
    "password='" + pwd + "'"

def main():
    conn = psycopg2.connect(connection)
    planet_sok(conn)
    legg_inn_resultat(conn)

def planet_sok(conn):
    print("--[ PLANET-SØK ]--")
    molecule1, molecule2 = (input("Molekyl: ") for _ in range(2))
    if not molecule1:
```

```
        print("Ugyldig input, minst ett molekyl må oppgis")
        return

    cursor = conn.cursor()
    params = [molecule1]
    query = """
            SELECT DISTINCT
                planet.navn,
                planet.masse,
                stjerne.masse,
                stjerne.avstand,
                liv
            FROM planet
                INNER JOIN stjerne ON planet.stjerne = stjerne.navn
                NATURAL JOIN materie AS m1
                INNER JOIN materie AS m2
                    ON m1.planet = m2.planet
            WHERE m1.molekyl = %s
            """
    if molecule2:
        query += "AND m2.molekyl = %s"
        params.append(molecule2)
    query += " ORDER BY stjerne.avstand;"

    cursor.execute(query, params)
    for line in cursor.fetchall():
        print("--Planet--")
        print(f"Navn: {line[0]}")
        print(f"Planet-masse: {line[1]}")
        print(f"Stjerne-masse: {line[2]}")
        print(f"Stjerne-distanse: {line[3]}")
        print(f"Bekreftet liv: {"Ja" if line[4] else "Nei"}\n")

def legg_inn_resultat(conn):

    print("--[ LEGG INN RESULTAT ]--")
    planet = input("Planet: ")
    skummel = True if input("Skummel: ") == "j" else False
    intelligent = True if input("Intelligent: ") == "j" else False
    beskrivelse = input("Beskrivelse: ")

    cursor = conn.cursor()
    query = """
        UPDATE planet
        SET skummel = %s, intelligent = %s, beskrivelse = %s
        WHERE navn = %s
```

```
    """
    cursor.execute(query, (skummel, intelligent, beskrivelse, planet))
    if cursor.rowcount == 0:
        print("Something went wrong")
        return
    conn.commit()
    print("Resultat lagt inn.")

if __name__ == "__main__":
    main()
```

Sikkerhet

I databasen

Man kan kontrollere tilgang via brukere, roller, og rettigheter.

Brukere og roller

```
CREATE ROLE ifi_student;

CREATE USER yrjarv WITH PASSWORD 'hunter2'
    ROLE ifi_student, annen_rolle;
```

Roller og brukere slettes med `DROP` (som alt annet). Teknisk sett er `CREATE USER` i Postgres bare et alias for `CREATE ROLE` med tilgangen `LOGIN`.

Man kan begrense varigheten av en bruker/rolle med `VALID UNTIL 'YYYY-MM-DD'`, og man kan begrense maks antall samtidig åpne tilkoblinger med `CONNECTION LIMIT <n>`.

Rettigheter

```
GRANT <privilegier> ON <type> <objekt> TO <rolle>
```

I `<privilegier>` kan man ha f.eks. `SELECT, UPDATE, INSERT, DELETE, CREATE, CONNECT, USAGE, ALL`. `<objekt>` er en database, en tabell, et skjema, etc.

Fjerning av rettigheter kan gjøres tilsvarende med `REVOKE`

I programmer som jobber opp mot databasen

Det er viktig å alltid stenge connectionene man har mot databasen når man er ferdig med å bruke dem.

SQL injection

Dette er den største trusselen. Det løses med parameterization (se eksempel under programmerings-delen). Man kan ved å gi f.eks. input `' SELECT 1;--` i et sårbart tekstfelt kan vi avslutte querien tidlig og gi et proof of concept på at det er en sårbarhet.

Hvordan en DBMS funker

Indeksstrukturer

En indeksstruktur er en datastruktur på disk som lar databasen raskere finne bestemte rader i en tabell. Det finnes to hovedtyper indekser: hash-baserte og tre-baserte.

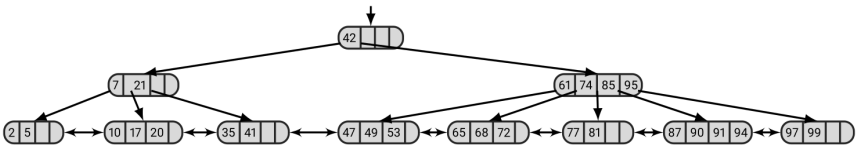
Når man markerer en kolonne med `PRIMARY KEY` blir det automatisk opprettet en B-tre-indeks på denne kolonnen. Derfor er alltid joins over primærnøkler relativt effektive. Men hvis vi vil gjøre søk, oppslag, eller joins over kolonner som ikke er primærnøkler må vi lage indeksene selv:

```
CREATE INDEX <navn> ON <tabell>(<kolonner>)
```

Merk at hvis man lister opp flere kolonner blir indeksen over alle kolonnene samtidig.

B-tre-indekser

Dette er en trestruktur hvor hver node kan ha mange barn. Nodene har samme størrelse som en disk-blokk. De minimerer antall oppslag på disk, siden hver verdi i løvnodene har pekere til dens tilhørende rad i den tilhørende tabellen.

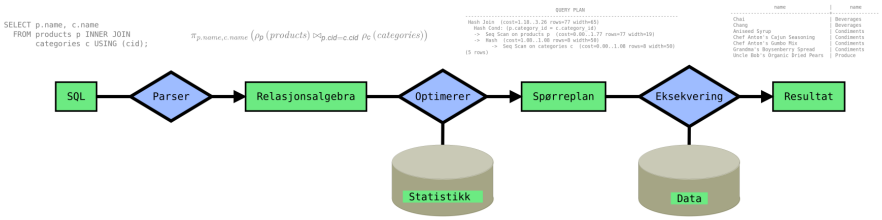


Hash-indekser

En hash-indeks bruker en hash-funksjon for å oversette en verdi til en minneadresse. På minneadressen ligger en liste med pekere til rader som har denne verdien. De er mer effektive på oppslag på konkrete verdier, men kan ikke brukes for intervaller.

Spørreprosessering

En spørring går gjennom følgende steg før den til slutt blir evaluert:



Parsere sjekker spørringen syntaktisk, og oversetter spørringen til et spørretre over relasjonsalgebraen. Relasjonell algebra kan man manipulere algebraisk, og dermed kan man lege forskjellige (men ekvivalente) spørringer. Dermed tilordner man en omtrentlig kostnad for hver av de forskjellige mulige spørringene, og velger den som er billigst å utføre.

Til slutt lages det en spørreplan for den valgte spørringen, som er en detaljert plan for hvordan den skal eksekveres. Denne kjøres (evalueres over databasen).