UNIVERSITY OF OSLO
Department of Informatics

Romlige/temporale modeller -Geografiske Informasjonssytemer sett i lys av Ogdens trekant

Fozia Jabeen Arif
Shomaila Kausar

30. januar 2005

FORORD

Våren 2003 fikk vi tildelt to forskjellige oppgaver av vår veileder Gerhard Skagestein. Oppgavene gikk ut på å finne ut om rom og tid kunne modelleres likt slik at de senere også kunne forenes og betraktes på samme måte. Oppgaven utviklet seg etter hvert til en diskusjon rundt Ogdens trekant og dette var starten på et fantastisk samarbeid. Det ble til tider veldig kaotisk og hektisk ettersom vi begge var i fulltidsjobb. Men dette har igjen ført til at vi lærte å koordinere og prioritere ting på en slik måte at vi fikk gjort ferdig denne oppgaven. Og slik ser den ut i dag.

Vi ber om unnskylding til våre familier og venner som vi ikke har hatt tid til, spesielt tre måneder før oppgaven ble innlevert. Vi vil spesielt benytte sjansen for å takke våre foreldre Liaqat Ali, Kausar Bibi, Mohammed Arshad og Shamim Akthar.

Vi vil samtidig benytte anledningen til å takke alle og enhver som har støttet oss gjennom våre "ups and downs". Vi vil takke Gerhard Skagestein for all kritikk og all ros. Vi vil også takke Ragnar Normann som har vært meget behjelpelig spesielt ved et par spørsmål knyttet til relasjonsdatabaser.

Til slutt må vi gratulere hverandre for å ha kommet oss gjennom ca 170 sider uten et eneste krangel. Dette tyder på at vi i tillegg til å utfylle hverandre, er meget flinke til å samarbeide. Det har i og for seg vært en opplevelse å komme seg gjennom dette arbeidet og vi har lært utrolig mye.

Vi vil igjen takke vår veileder Gerhard Skagestein for all tålmodighet, samtidig som vi vil takke Ifi Drift, Administrasjonen og BIBSYS som tok imot våre spørsmål med åpne armer.

Jeg Fozia Arif vil spesielt takke til mannen min (Arif Wasif) for at han oppfordret meg til å ta masterstudiet og støttet meg gjennom studiet. Spesielt vil vi begge takke for ideene han ga oss rundt algoritmen for å finne krysspunkt mellom to linjer.

Shomaila Kausar og Fozia Arif
Oslo, Norge: 30.januar 2005

INNHOLDSFORTEGNELSE

1 Problemstilling....8
1.1 Oppbygning av oppgaven.... 9
2 Interesseområdet....11
2.1 Rommet.... 11
2.2 Fenomener i rommet.... 11
2.3 Ikke romlige og romlige assosiasjoner.... 12
3 Den romlige modellen....13
3.1 Rommet.... 14
3.2 Objektmodellen.... 15
3.2.1 Geometriske modeller.... 16
3.2.2 Topologiske modeller.... 16
3.3 Feltmodellen.... 16
3.4 Romlige verdier.... 16
3.4.1 Operasjoner på romlige verdier.... 18
3.4.1.1 Topologiske operasjoner.... 18
3.5 Skranker.... 19
3.6 Passive og aktive modeller.... 20
3.7 Projeksjoner.... 21
3.8 Utvidelse av dimensjonalitet.... 21
3.8.1 Lagrede verdier.... 22
3.8.2 Bruk av to modeller.... 22
3.8.3 Dimensjonsutvidelse med en konstant.... 23
3.9 Monolittiske objekter vs. sammensetninger og aggregater.... 23
3.9.1 Aggregatmodellen.... 26
3.9.2 Delta-modellen.... 27
3.9.3 Forening av verdi og aggregat.... 27
4 Representasjon av romlige verdier....29
4.1 Rasterrepresentasjon.... 29
4.2 Vektorrepresentasjon.... 29
4.3 Halvplan representasjon.... 30
4.4 Vektor, raster og dimensjoner.... 31
4.5 Representasjon av fuzzy verdier.... 32
5 Koordinatsystemer....33
5.1.1 Origo.... 33
5.1.2 Akser og dimensjonalitet.... 33
5.1.3 Kartesiske koordinater vs. polarkoordinater.... 33
5.1.4 Skala og standarder.... 34
5.2 Oppløsningsevnen.... 34
5.3 Avrundingsfeil, diskretisering.... 35
5.3.1 Rose Algebra.... 36
5.3.2 Diskretisering i tre dimensjoner.... 36
5.3.3 Diskretisering ved hjelp av FD metoden.... 37
5.3.4 Diskretisering ved hjelp av FE metoden.... 38
5.3.5 Forskjeller ved bruk av FD og FE metoden:.... 38
5.3.6 Intervall og FD metoden.... 39
5.3.7 Forening av FD og FE metoden.... 39
6 Identifikatorer....41
6.1 0-dimensjonale verdier som del/hel av identifikatoren.... 41
6.2 1-dimensjonale verdier som hele/del identifikatoren.... 42
6.3 Konklusjon.... 44
7 Realisering av romlige/topologiske skranker....45

7.1 Skranker i spagettimodellen ... 46
7.2 Skranker i den topologisk modellen ... 47
7.2.1 Realisering av 1D Topologisk modell / Nettverk modell ... 47
7.2.2 Realisering av 2D Topologisk modell ... 49
7.2.3 Realisering av 3D topologisk modell ... 49
7.3 Oppsummering ... 50
8 Operasjoner og abstrakte datatyper ... 52
8.1 Operasjoner mot data i relasjonelle databasehåndteringssystemer ... 53
8.1.1 Operasjonen med SQL ... 55
8.1.2 Operasjonen med Java database connection ... 57
8.2 Operasjoner i objektorienterte systemer ... 59
8.3 Objektrelasjonelle databasesystemer ... 60
8.3.1 Utvide datamodellen med romlige ADT'er ... 61
8.3.1.1 Objekttyper ... 62
8.3.1.2 Metoder ... 63
8.3.1.3 Samlinger ... 64
8.3.1.4 Flernivå samlingtyper ... 65
8.3.1.5 Objekttabeller ... 66
8.3.2 Datamodellen for togobjekter i databasen ... 67
8.3.3 Algoritmen i ORDBMS ... 69
8.3.3.1 CURSOR ... 70
8.3.3.2 Resultat ... 74
8.4 Oppsummering ... 75
8.4.1 Relasjonelle databasesystemer ... 75
8.4.2 Objektorientering ... 75
8.4.2.1 Objektorienterte databasesystemer ... 76
8.4.2.2 Objektrelasjonelle databasesystemer ... 76
8.4.3 Konklusjon ... 76
9 Regning med romlige verdier ... 77
9.1 Objekt A ∩ Objekt B = Punkt / Punkter ... 77
9.2 Objekt A ∩ Objekt B = Linjer ... 77
9.3 Objekt A ∩ Objekt B = Flater ... 78
9.4 Generell framgangsmåte for å sjekke snitt ... 79
9.5 8.1.5 Beregning av snitt: ... 80
9.6 Intersection resultat: ... 80
9.7 Geometrialgoritmer ... 81
9.8 Algoritme 1a Areal av tredimensjonalt pyramide ... 82
9.9 Algoritme 1b Volum av tredimensjonalt plan polygon ... 83
9.10 Kombinasjon av x, y og t og Algoritme 1 ... 83
9.11 Interpolasjon ... 84
9.12 Interpolasjon og vektor ... 85
9.13 Kontinuerlige og diskontinuerlige målinger ... 85
9.14 Kontinuerlige interpolasjonsmetoder ... 86
9.15 Diskrete interpolasjonsmetoder ... 86
9.16 Ekstrapolasjon ... 87
10 Modellsyn i SQL92 med Oracle Spatial ... 89
10.1 Oracle Spatial ... 89
10.1.1 Oracle Spatial arkitektur ... 91
10.2 Representasjon av romlige objekter ... 93
10.2.1 Insert metadata informasjon ... 95
10.2.2 Romlig indeksering ... 96
10.3 Spørringer mot romlige objekter ... 98
10.4 Modellering av to ikke lineære linjer ... 99

10.5 Hvorfor en Romlig DBMS ? 100
11 Modellsyn i Simple Feature Spesifikasjon 102
11.1 Relasjonelle Operatorer 102
11.2 Arkitektur- SQL92 Implementasjon av Objekt tabeller 105
11.3 Arkitektur - SQL92 med Geometri Typer Implementasjon av Objekt tabeller 106
12 Modellsyn i Geography Mark-up Language (GML) 107
12.1 Geography Mark-up Language (GML) 107
12.2 GML Objektmodell 107
12.3 Modellering i GML 109
12.3.1 Geometrier i GML 109
12.3.1.1 Geometri schema 110
12.3.2 Egenskaper av geometriverdier 111
12.3.2.1 Feature skjema 111
12.4 GML topologi 112
12.5 GML temporale elementer 113
12.6 Konklusjon 113
13 Realiseringsplattformer og realiseringer 114
13.1 Typer av DBMS 114
13.2 Realisering av romlige objekter 114
13.2.1 Realisering på relasjonsdatabaseplattform 115
13.2.1.1 Implementasjonen 118
13.2.1.2 Prosedyrer 120
13.2.2 Realisering på objektrelasjonell databaseplattform 120
13.2.2.1 Romlige ADT operasjoner 123
13.2.2.2 Fordeler ved objektrelasjonelle databaser 124
13.2.3 Realisering på objekt orientert plattform 124
13.2.3.1 Fordeler ved objektmodeller 126
13.2.4 Oppsummering 126
14 Noen eksempler på modeller og bruken av dem 127
14.1 Korteste vei 127
14.2 Fra krysningspunkt til node: 128
14.3 Dijkstras algoritme 129
14.4 Restriksjoner og brukt tid 129
14.5 Seiling og korteste vei problemet 131
14.6 Effektiv algoritme 132
14.7 Generalisering av t som attributt til t som en egen akse 133
14.7.1 Tog i bevegelse 133
15 Påvirker representasjonen den konseptuelle modellen? 135
15.1 Enkel konseptuelll modell 135
15.2 Muligheter for å bytte representasjon 136
15.3 Skranker 136
15.4 Konklusjon 137
15.5 Forbedringsforslag og videre arbeid 138
16 VEDLEGG 139
16.1 Enkel kryss (Versjon 1) 139
16.1.1 RDBMS – Realsjonell databasehåndteringssytem 139
16.1.2 OO- Objekt orientert 144
16.1.3 ORDBMS – Objektrealsjonell databasehåndteringssytem 150
16.2 Avansert kryss (Versjon 2) 157
16.2.1 RDBMS – Realsjonell databasehåndteringssytem 157
16.2.2 ORDBMS – Objektrealsjonell databasehåndteringssytem 161
16.2.3 OO- Objekt orientert 165
16.2.4 165

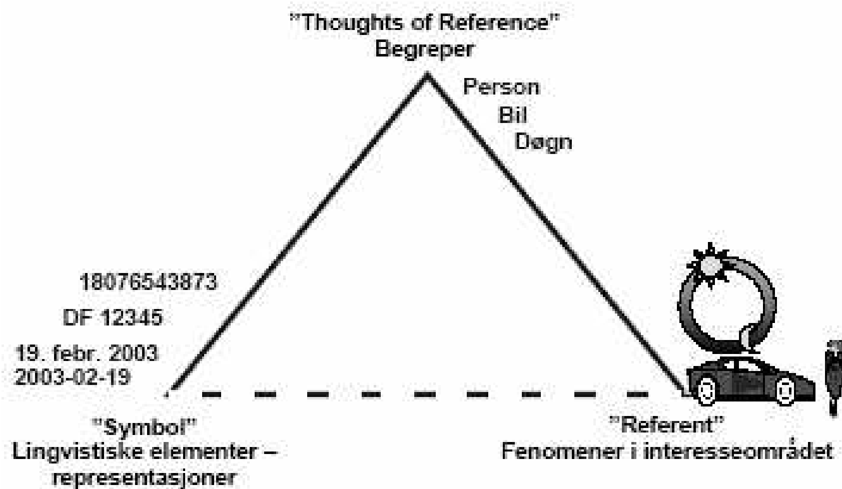
16.3 Oracle Spatial kryss (Versjon 3)....165
16.3.1 Datamodell....165
16.3.2 Enkel kryss....167
16.3.3 Avansert kryss....168
Referanse liste....170

FIGURLISTE

Figur 1 Ogdens trekant (hentet fra[Skagestein 2004]) ... 8
Figur 2 Hus, hentet fra [Skagestein 2004] ... 9
Figur 3 Fenomener i interesseområdet ... 11
Figur 4Hus, hentet fra [Skagestein 2004] ... 12
Figur 5 Modellen ... 13
Figur 6 Den romlige modellen beskriver interesseområdet og foreskriver realiseringen – hentet fra [Skagestein 12.mai 2004] ... 13
Figur 7 Lineær, forgrenet og syklisk tidsmodell ... 14
Figur 8 Ugruppert klassediagram i UML ... 15
Figur 9 Topologiske forhold mellom romlige egenskaper til to objekter – hentet fra [Egenhofer] ... 18
Figur 10 Anti projeksjon fra todimensjoner til tredimensjoner ... 22
Figur 11 Anti projeksjon mellom to eksisterende modeller ... 23
Figur 12 Pyntefigur ... 24
Figur 13 Jostedalsbreen, 2003-2005 ... 25
Figur 14 Sammensetning og aggregering ... 25
Figur 15 Lepidopten med aggregatforhold til de ulike fasene, inspirert av Cris Partidge [Partidge, 1996] ... 26
Figur 16 Fenomen A og fenomen B ... 26
Figur 17 Hendelsesorientert modell, figuren er hentet fra [Midtbo] ... 27
Figur 18 Dimensjonalbilder inndelt etter lag ... 28
Figur 19 Forening av dimensjon og aggregat metoden ... 28
Figur 20 ... 29
Figur 21 Et eksempel for hvordan man beskriver en polygon med halvplan representasjon. Dette er en ikke konveks polygon som må deles i konvekse deler og representeres med en polyhedron. Figuren er hentet fra [Skagestein 2004] ... 31
Figur 22 Polar koordinatsystem, hentet fra [www.wikipedia.com] ... 34
Figur 23 Diskretisering og forskjellige dimensjoner ... 35
Figur 24 Rose algebra og snitt ... 36
Figur 25 Diskretiseringstyper ... 37
Figur 26 Intervall med forsinkelser ... 37
Figur 27 Utregning ... 38
Figur 28 Bevegelser i et intervall ... 39
Figur 29: 0-dimensjonale begreper i 1-dimensjonalt rom ... 41
Figur 30: Høyden som en avhengig variabel av person (virkningsvariabel) ... 41
Figur 31: Mål under entydighetskranke ref:Gerhards foiler (http://www.ifi.uio.no/inf102/foiler/tidogrom.pdf)42
Figur 32: Bestilling av rom som viser overlapping av rom 3A fra kl 15.00 til kl 16.00 den 1.mars 2004 ... 43
Figur 33: At et gjerde (G) rundt en eiendom (E) sammenfaller med eiendomsgrensen kan sikres ved skranke eller avledning ... 45
Figur 34: To eiendommer med en felles grense 'k4' ... 46
Figur 35: Grensen mellom to tilstøtende eiendommer er representert to ganger i spagettimodellen ... 46
Figur 36 Representasjonsmodell for en nettverk-abstraksjon ... 48
Figur 37: Et enkelt veinettverk med den korresponderende topologiske modellen ... 49
Figur 38: 3D Topologisk model ref:A 3D topological model for augmented reality ... 50
Figur 39: Et eksempel på 3D-objekt ... 50
Figur 40: To tog på Bergensbanen (Bergen-Hønefoss, Hønefoss-Bergen) ... 53
Figur 41: Datamodellen over toglinjer og passeringer. ... 55
Figur 42: tegning(A) viser toglinjer representert ved hjelp av antall punkter (her 4) med vektorrepresentasjon, og tegning (B) viser toglinjer representert ved hjelp av halvplan representasjon (uendelig antall punkter) ... 56
Figur 43: Objektorientert modell for å finne krysspunkt mellom to toglinjer ... 59
Figur 44 Arkitektur for en ORDBMS, med geometriske datatyper ... 62
Figur 45: n_lpoints tabellen er nøstet i n_line tabellen og n_line tabellen er nøstet i line kolonnen ... 67
Figur 46: Modell over abstrakte datatyper i databasen for å finne krysspunktet ... 68
Figur 47 Snitt mellom to regelmessige kurver ... 77
Figur 48 Snitt av uregelmessige kurver ... 78
Figur 49 Sjekk av snitt ved hjelp av ytterpunkter ... 79
Figur 50 Beregning av snittmengde ... 79
Figur 51 Bounding box ... 80

Figur 52 Sjekk av punkter, linjer og flater ... 81
Figur 53 Pyramider med en polygon som grunnflate ... 82
Figur 54 Arealet av en triangle... 83
Figur 55 Eksempel beregning av areal... 84
Figur 56 Projeksjon av t... 84
Figur 57 konternierlige og diskonternierlige målinger ... 86
Figur 58 diskret interpolasjon, hentet fra [Etzelmüller, 1997] ... 87
Figur 59: Hierarkiske strukturen i Oracle Spatial på det spesielle eksemplet. ... 90
Figur 60: Mulige former som kan representeres i Oracle Spatial... 90
Figur 61: Modell arkitektur i Oracle Spatial... 91
Figur 62: Krysspunkt mellom tognr 609 og 62; tid:1021, km:287... 99
Figur 63 Dimensionally Extended Nine-Intersection Modell (DE-9IM), [OpenGis Simple Features Specification for SQL 1998]...102
Figur 64 Et eksempel på DE-9IM, [OpenGis Simple Features Specification for SQL 1998]...103
Figur 66 Berøring (Touches)...104
Figur 68 Inni (Within)...104
Figur 69: Skjema for objekt tabeller i SQL92, [OpenGis Simple Features Specification for SQL 1998] ...105
Figur 70: Virkeligetsbildet for forholdet "bro spenner elv". ...108
Figur 71: Representasjon av forholdet "bro spenner elv" i GML ...108
Figur 72: Modellering av forholdet "bro spenner elv" i GML. ...109
Figur 73 : Geometri schema (ref: http://www.ia.hiof.no/~gunnarmi/gml2.1.1.pdf) ...110
Figur 74: Basis geometriske egenskaper ...111
Figur 75 : Feature schema ...112
Figur 76: Krysspunkt mellom tognr 609 og 62; tid:1021, km:287...115
Figur 77 Representasjonsmodell for representasjon av kolonne med romlig verdi. ...116
Figur 78 Implementasjonsdetaljer for toglinjer schema ...119
Figur 79 Representasjonsmodell (1) for representasjon av romlige objekter med en kolonne av romlig verdi, av datatypen geometri, i ORDBMS (basert på Oracle Spatial modellen)...121
Figur 80 Objektrelasjonell modell(2) for representasjon av romlige objekter, med kolonne med romlig verdi av en geometrisk type. ...122
Figur 81 Objektorientert representasjonsmodell for romlige objekter. ...125
Figur 82 Problemer ved innleggelse av koordinater ...127
Figur 83 Vei 1 og Vei 2 fra figur 28 med "ekstra vei" ...128
Figur 84 Modellering av vei...128
Figur 85 Fra vei samling til graf med noder ...129
Figur 86 Klassediagram...129
Figur 87 Eksempel på veinett...130
Figur 88 Veinett med restriksjoner...131
Figur 89 Seilbåts bevegelser ...131
Figur 90 Beregning av korteste vei...132
Figur 91 Optimal rute og regnskyer ...133
Figur 92: Figuren ovenfor ilustrerer at modellen bør være totalt uavhengig av representasjonen slik at man kan velge mellom en mengde av reresentasjonstyper. ...135

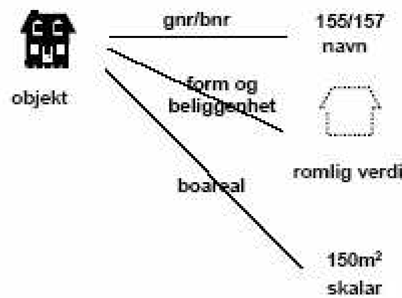
1 Problemstilling



Figur 1 Ogdens trekant (hentet fra[Skagestein 2004])

For å kunne skrive ned eller lagre noe om virkeligheten, må vi først danne oss et mentalt bilde eller en modell av virkeligheten. Hvis vi observerer et fenomen i virkeligheten (interesseområdet), for eksempel en person, må vi i vårt mentale bilde danne begrepet "person" før vi kan skrive ned opplysninger om denne personen. Dette kalles konseptualisering, og ble beskrevet av Ogden og Richards**.** Prinsippet er illustrert i den såkalte Ogdens trekant – se figur 1. Den viser at fenomener i interesseområdet (høyre hjørne) ikke kan beskrives ved lingvistiske elementer eller representasjoner (venstre hjørne) direkte, vi må gå veien om begreper (topphjørnet).
Videre er det slik at hvilke fenomener vi oppfatter i virkeligheten og hvilke av dem vi velger å ta med i vårt mentale bilde, er avhengig av hva vi vil bruke modellen til. Det er viktig at når flere mennesker skal samarbeide, og kanskje også bruke et datamaskinbasert informasjonssystem, at mennesker og datamaskiner har en felles oppfatning av virkeligheten, altså en felles modell. Slike modeller dokumenteres gjerne i form av datamodeller, for eksempel i form av Entity-Relationship-diagrammer eller dataorienterte UML klassediagrammer.

En spesiell variant av slike modeller følger idegrunnlaget for Ogdens trekant fullt ut ved at foreskriver at alt skal betraktes som begreper og assosiasjoner mellom slike begreper, selv fenomener som vanligvis oppfattes som egenskaper ved andre fenomener – for eksempel en persons høyde eller vekt. NIAM-datamodeller (se for eksempel [Skagestein 2002] kapittel 6) er det mest fremtredende eksempel på denne type modeller. Samme type modeller kan uttrykkes ved hjelp av ugrupperte UML klassediagrammer, se [Skagestein 2002] kapittel 5. Slike modeller skiller altså klart mellom begrepet (for eksempel Person) og representasjonen for dette begrepet (for eksempel fødselsnummer).



Figur 2 Hus, hentet fra [Skagestein 2004]

I dag har man fått et utvidet behov for å modellere fenomener fra virkeligheten med romlige egenskaper. Som eksempel kan vi se på et hus – se figur 2 – som kan være knyttet til ikke-romlige begreper som for eksempel gards og bruksnummer, og til et romlig begrep som er husets form og beliggenhet. utstrekning og/eller en beliggenhet i rommet. Også utviklingen over tid kan være av interesse, dette fører til såkalte romlig/temporale modeller. Et eksempel kan være et system som ved hjelp av historikk og prognoser skal gi muligheten til å simulere fram utviklingen av isbreer. Spørringene kan gå ut på å beregne overflate, volum eller andre romlige verdier og hvordan disse verdiene endrer seg med tiden.

I datamaskiner må representasjoner baseres på biter og bytes. Dette gir ekstra utfordringer i forbindelse med representasjon av verdier som har en romlig utstrekning, fordi slike verdier ikke lar seg representere så direkte som tekster og tall. Raster-, vektor- og halvplanrepresentasjoner er vanlige løsninger på disse utfordringene.

Vi skal i denne oppgaven se på konsekvensene av å bruke ideene bak Ogdens trekant i forbindelse med romlig/temporale modeller, altså modeller som omfatter fenomener som har en beliggenhet i rommet, som kan ha en utstrekning og som kan endre seg med tiden.

1.1 Oppbygning av oppgaven

For å besvare vår problemstilling har vi valgt å oppdele hovedfagsoppgaven slik at kapitelene blir inndelt i virkelighet, modell og representasjon som Ogdens trekant.

- I kapitel "2 Interesseområdet" beskriver vi rommet i virkeligheten
- Kapitel "3 Den romlige modellen" omhandler forskjellige typer modeller. Vi kommer blant annet inn på feltmodellen og typer operasjoner som kan gjøres på romlige verdier.
- I kapitel "4 Representasjon av romlige verdier" kommer vi inn på de forskjellige representasjonsmåtene, koordinatsystemer og ser på oppløsningsevne og diskretisering.

- Koordinatsystemer, oppløsningsevne og avrundingsfeil blir tatt opp i kapitel ”5 Koordinatsystemer”
- Valg av identifikatorer, både verdier av null og en dimensjon blir diskutert i kapitel ”6 Identifikator”
- I kapitel ”7 Realisering av romlige/topologiske skranker” ser vi på hvordan skranker blir brukt i spagetti- og objektmodellen samtidig som vi ser på hvordan dette kan realiseres i forskjellige topologiske modeller.
- Kapitel ”8 Operasjoner og abstrakte datatyper” tar opp hvordan operasjoner kan gjøres mot data i relasjonelle og objektorienterte databasehåndteringssystemer.
- Hvordan eksisterende algoritmer for romlige verdier kan benyttes til beregninger av romlige og temporale verdier blir vist i kapitel ”9 Regning med romlige verdier”.
- Kapitel ”10 Modellsyn i SQL92 med Oracle Spatial”, ”11 Modellsyn i Simple Feature Spesifikasjon” og ”12 Modellsyn i Geography Mark-up Language (GML)” viser hvordan modellene er.
- I kapitel ”13 Realiseringsplattformer og realiseringer” tar for seg hvordan realisering av romlige objekter blir utført på forskjellige databaseplattformer.
- Korteste vei algoritmen og generalisering av t som en egen dimensjon på lik linje med de romlige dimensjonene er noe av det som blir tatt opp i kapitel ”14 Noen eksempler på modeller og bruken av dem”.
- I kapitel ”15 Påvirker representasjonen den konseptuelle modellen?” kommer vi frem til et svar vi mener er riktig samtidig som vi kommer med forslag til videre arbeid.

Vi starter med å beskrive virkeligheten slik den er før vi går over på modell- og representasjonsnivået. Vi har prøvd å plassere tekstene slik at de kommer inn i kapiteler som omhandler disse emnene hver for seg, men noe stoff gjelder for både modeller og representasjon. Dette er blitt plassert der hvor vi mener at det er mest hensiktsmessig å plassere.

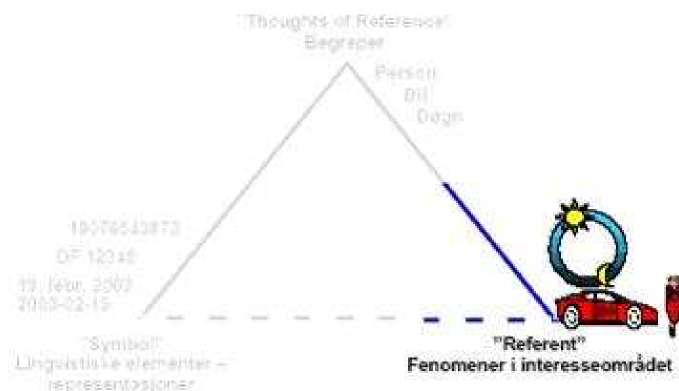
2 Interesseområdet

Den delen av virkeligheten som vi ønsker å beskrive, kalles interesseområdet (Universe of Discourse – UoD) – se for eksempel [Skagestein 2002] kapittel 5. Det spesielle med interesseområdene som diskuteres i denne oppgaven er at de omfatter også romlige aspekter. Praktisk talt alle utsnitt av ”virkeligheten” vil ha romlige aspekter, men det er ikke alltid vi er interessert i dem. Men det er vi her.

2.1 Rommet

Matematisk sett kan rommet ha et vilkårlig antall dimensjoner. Når vi betrakter den virkelige verden og representerer den eksempelvis i geografiske informasjonssystemer, begrenser vi oss vanligvis til tre dimensjoner. Av og til nøyer vi oss med bare to eller en eneste dimensjon. For å kunne fange opp endringer over tid kan vi på lik linje med de romlige dimensjonene innføre en fjerde dimensjon, nemlig tiden. Dermed åpner vi for mer komplette modeller av virkeligheten. I boken ”Business Objects, re-engineering for re-use” sier Cris Partidge [Partidge 1996] at man går fra et egosentrisk til et objektivt syn på verden ved å ta med t-aksen på lik linje med de romlige dimensjonene.

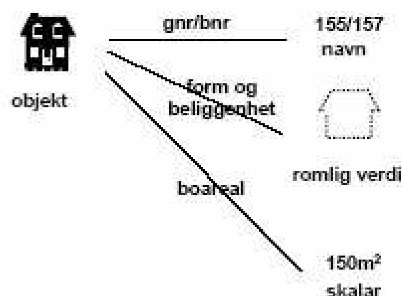
2.2 Fenomener i rommet



Figur 3 Fenomener i interesseområdet

I rommet eksisterer det fenomener som vi senere skal modeller som objekter. Her er vi fortrinnsvis interessert i fenomener som har et romlig aspekt. Det innebærer at fenomenet må ha minst en romlig egenskap, som for eksempel form og/eller beliggenhet.

For eksempel (se [Skagestein 2004]) kan fenomenet ”Hus” ha tre egenskaper: Eier (navn), navn (gnr/bnr), boareal (skalar) og form og beliggenhet (romlig verdi). Dette huset er et romlig fenomen fordi vi er interessert i en romlig egenskap, nemlig form og beliggenhet. Eierens navn og gårds- eller bruksnummer er noe som kan sies å være et navn (som regel representert som tekststrenger). Boarealet er en målbar verdi og er dermed en skalar (som regel representert som et heltall). Form og beliggenhet er representert med romlige verdier. Hvordan vi representerer romlige verdier avhenger av bruksformen. Som vi skal se, finnes det mange ulike prinsipper for å representere romlige verdier.
I tillegg kan huset bli påvirket over tid slik at disse verdiene blir endret. For eksempel kan huset skifte eier, eller det kan bli revet eller ombygget, noe som kan føre til nye romlige verdier



Figur 4Hus, hentet fra [Skagestein 2004]

Et annet eksempel: En isbre tre romlige dimensjoner – ved hjelp av disse kan vi finne formen og regne ut forskjellige skalare verdier, som for eksempel overflate og volum. La oss anta at breens romlige egenskaper og forhold endrer seg med tiden. Da kan vi ta inn i interesseområdet forandringer i breens overflate og volum som skyldes ytre påkjenninger som forandres gjennom tiden, for eksempel temperatur- og nedbørsforandringer, og enkelt kunne betrakte breen fra ulike perspektiver tilbake i tid, nå og fremover i tid (det siste forutsetter at vi har prognoser for utviklingen).

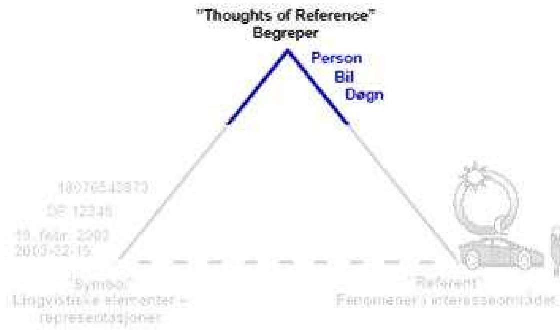
Noen andre eksempler på romlige fenomener av ulike dimensjoner:

- **0D**: Trigonometriske punkter og grensemerker.
- **1D**: Kommunegrenser og landegrenser
- **2D**: Land, innsjøens overflate og eiendommer
- **3D**: Jordkloden

2.3 Ikke romlige og romlige assosiasjoner

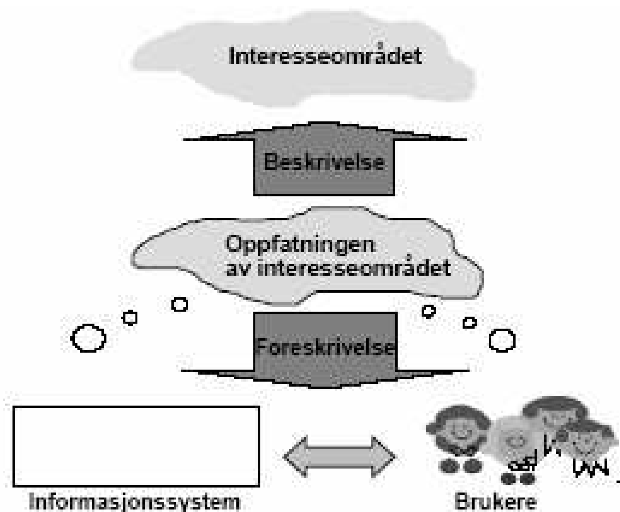
I tillegg til selve fenomenene er vi også interessert i assosiasjonene (forholdene) mellom dem. Et eksempel på en slik assosiasjon er at hus står på en eiendom. Slike assosiasjoner kan være av to typer, romlige og ikke romlige. Eksempler på ikke romlige assosiasjoner er ”eier”, ”er søsken av”, ”er en del av”. Eksempler på romlige assosiasjoner er ”inne i”, ”nordenfor”, og romlig/temporale assosiasjoner ”samtidig som”, ”før”, ”etter”. Mange av de romlige assosiasjonene er av typen topologiske assosiasjoner.

3 Den romlige modellen



Figur 5 Modellen

Den romlige modellen tjener til å beskrive vår oppfatning av interesseområdet, fortrinnsvis på en så presis måte at modellen samtidig kan brukes til å foreskrive større eller mindre deler av et informasjonssystem som en organisasjon kan bruke for å håndtere interesseområdet – se figur 6. Vi vil her arbeide med modeller der fenomener i virkeligheten modelleres som objekter. I tillegg må vi modellere de interessante assosiasjonene mellom fenomenene.



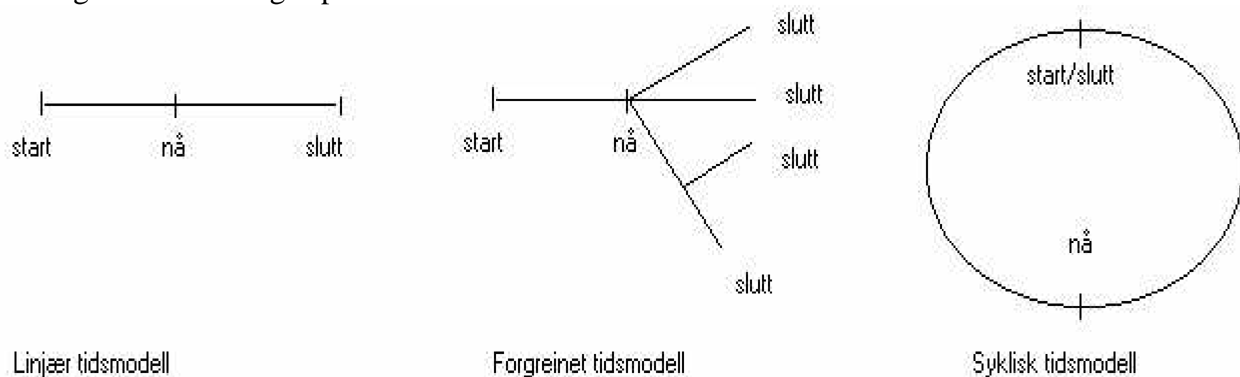
Figur 6 Den romlige modellen beskriver interesseområdet og foreskriver realiseringen – hentet fra [Skagestein 12.mai 2004]

Klassediagrammet i UML består hovedsaklig av klasser/objekter og deres forhold til hverandre, og kan benyttes som modell av virkeligheten. Klassediagrammet kan så detaljeres slik at de inneholder attributter og metoder for en hver klasse, men dette avhenger av hva vi har tenkt å bruke klassediagrammet til og hvilken grad av detaljering vi ønsker. Forholdene i klassediagrammet kan være av fire typer, avhengighet, assosiasjon, generalisering og realisering. Et forhold av typen avhengighet har man når en klasse bruker egenskapene til en annen klasse, generalisering bygger på arv mens assosiasjon har man når objektene i modellen har en eller annen form for forbindelse mellom seg.

3.1 Rommet

Matematikere og naturvitenskapsmenn gjennom historien mener at vi i geometrien har lengde, bredde og høyde – se for eksempel [Laudal, 2000]. Det kartesiske rombegrepet blir beskrevet ved hjelp av koordinatene x, y og z, hvor x og y er koordinater i gulvplanet og z er rett opp. Som vi har sett i forrige kapittel kan vi innføre tiden t som en fjerde dimensjon, dermed oppstår en koordinat med de fire aksene (x, y, z, t). Olav Arnfinn Laudal [Laudal 2000] velger å kalle dette firedimensjonale rommet for tidsrommet. I visse tilfeller er vi interessert i å betrakte rom med lavere dimensjonalitet, som for eksempel bare (x, y, z), (x, y, t), (x, y) eller (x, t).

En gjengs oppfatning av virkelighetens tid er at den løper lineært fra den uendelig fjerne fortid til den uendelig fjerne fremtid. Imidlertid viser A. Renolen [Renolen 1999] viser til ytterligere to modeller for tid i tillegg til den lineære tidsmodellen, nemlig forgrenet tidsmodell og syklisk tidsmodell. I den forgrenede tidsmodellen blir tiden sett på som lineær fra start til nå. Videre frem i tiden vil tiden bli delt inn i flere forgreninger hvor hver gren kan fange opp en sekvens av mulige handlinger. Syklisk tidsmodell er en modell hvor man lar fortidens start og fremtidens slutt møtes i ett og samme punkt. En slik modell kan for eksempel oppstå i forbindelse med rullerende ukeplanlegging, der vi modellerer en uke på en slik måte at når det eksempelvis kommer en ny mandag, glemmer vi alt om den forrige mandagen og lar den nye mandagen ta den forriges plass.



Figur 7 Lineær, forgrenet og syklisk tidsmodell

Kombinert med x, y og z gir de to alternative tidsmodellene noen interessante "rom" som avviker fra det tradisjonelle kartesiske rommet. Vi skal imidlertid ikke forfølge dette temaet videre i denne oppgaven.

Xianoyu Wang, Xiaofang Zhou og Sanglu Lu [Wang] viser til at blant annet disse kravene bør oppfylles for å kunne modellere rom og tid:

1) Objekter skal kunne bli representert med deres posisjoner i rommet samt deres eksistens i tid.
2) Det skal være mulig å kunne ta med endringer i et fenomens plassering i rommet over en tidsepoke. Forandringene kan være av typen kontinuerlige eller diskrete, og endringene kan enten forandre retningen eller formen til et fenomen.
3) Det må være mulig å kunne definere romlige attributter og organisere dem i lag eller felter, og bestemme hvilken type de har. Attributtene kan være av typen kontinuerlige, eller diskrete.
4) Romlige attributtene trenger å kobles til deres tilhørende objekter
5) Objektenes romlige forhold skal kunne bli beskrevet over tid.

3.2 *Objektmodellen*

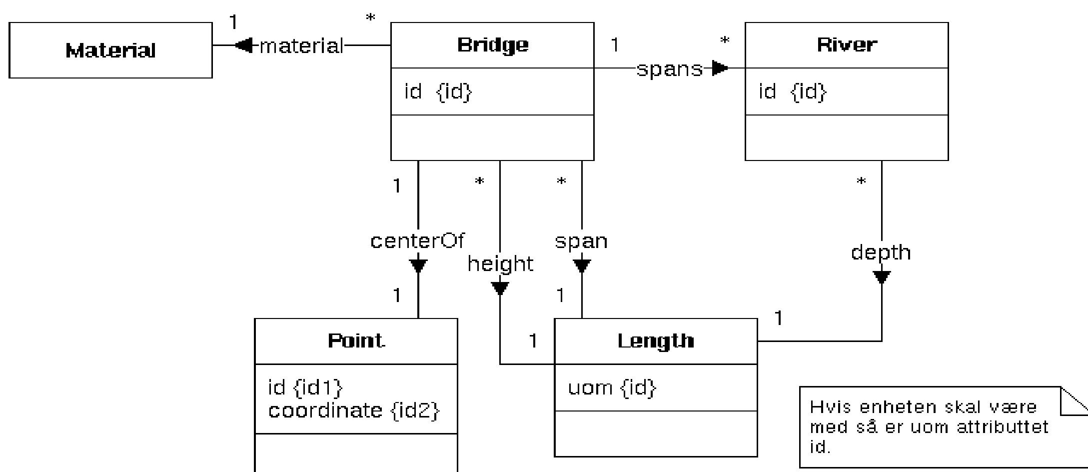
I objektmodellen betrakter vi interesseområdet som objekter og assosiasjon mellom objekter (se for eksempel Ron Lake [Lake 2004]). Objekter har egenskaper – også kalt attributter. Disse attributtene kan enten ha primitive verdier (typisk representert som boolean, integer, real, strings), geometriske verdier, og verdier som er assosiasjoner til andre objekter. Et romlig objekt må ha minst et attributt som kan ha en romlig verdi – det vil si en form og/eller en plassering, eller en romlig assosiasjon til et annet objekt.

En spesiell variant av objektmodellen er den ugrupperte modellen, der hvert objekt kan ha bare identifiserende attributter (som for eksempel fødselsnummeret for en person). Alle andre potensielle attributter modelleres istedenfor som assosiasjoner til objekter (som for eksempel fødestedet for en person). Denne modelleringstankegangen finner vi i NIAM (Nijssen Information Analysis Method) – senere kalt ORM (Object Role modell), i Skagesteins ugrupperte UML-klassediagrammer (se [Skagestein 2002] kapittel 5) og i programmeringsspråket Smalltalk. Fordelen med ugrupperte modeller er at vi ikke behøver å ta stilling til om ”noe” er en attributtverdi eller en assosiasjon til et objekt. Ulempen er at vi i modellen får objekter for alt, inkludert primitive verdier som for eksempel heltallet 7. Så snart verdiene blir romlige, og dermed mer kompliserte, blir det mer naturlig å se på verdiene som egne objekter. Ugrupperte modeller kan konverteres til mer implementasjons-orienterte modeller som objekter-med-attributter på et senere stadium i systemutviklingen.

I de ugrupperte modellene kan vi dermed skille mellom to typer av objekter:

1) **’Vanlige objekter’:** Objekter som gjenspeiler fenomener i interesseområdet
2) **’Verdiobjekter’:** Objekter som gjenspeiler verdier som blir assosiert med de vanlige objektene. Disse verdiene kan være primitive (for eksempel navn, tall og tidspunkter) eller mer komplekse (eksempelvis romlige verdier).

Figuren nedenfor viser et eksempel på et ugruppert modell i UML med vanlige objekter som *’Bridge’* og *’River’*, og verdi objekter som *’material’*, *’length’* og *’point’*. Nedenfor følger et eksempel på et ugruppert klassediagram i UML



Figur 8 Ugruppert klassediagram i UML

3.2.1 Geometriske modeller

I en geometrisk modell er objektene knyttet til romlige verdier som angir form og beliggenhet. Geometriske modeller egner seg godt som grunnlag for visualiseringer som for eksempel kart. Men for å finne ut om to objekter eksempelvis berører hverandre eller ikke må regne med de romlige verdiene – det vil si at vi må sammenlikne punktmengdene. Hvis punktmengdene ikke er helt nøyaktige, kan slike regnestykker gi feilaktige svar. Derfor benytter vi ofte topologiske modeller, eller en kombinasjon av geometriske og topologiske modeller.

3.2.2 Topologiske modeller

Topologi er en del av geometrien som beskriver hvordan fenomener henger sammen. Topologi kan også kalles gummi- og ballonggeometri, dette fordi topologi beskriver de egenskaper som ikke forandres selv om fenomenet eller rommet bøyes, strekkes eller deformeres.

I en topologisk modell er man – i motsetning til en geometrisk modell – bare interessert i hvordan objektene er plassert i forhold til hverandre i rommet. I det topologiske rommet kjenner vi ikke objektenes nøyaktige form og beliggenhet, bare deres beliggenhet i forhold til andre objekter. Topologiske modeller opprettholder de topologisk korrekte geometriske relasjoner som konnektivitet, naboskap og posisjon for objektet.

Ifølge Max J Egenhofer [Egenhofer] beskriver topologiske forhold assosiasjoner mellom objekter i rommet. Den topologiske strukturmodellen gir forbedret redigeringsfunksjonalitet og åpner muligheten for en rekke komplekse operasjoner som rute- og nettverksanalyser. Ved å flytte et romlig egenskap kan alle relaterte romlige egenskaper flyttes automatisk (aktiv) eller ved reprosessering av data (passiv).

3.3 Feltmodellen

Den såkalte feltmodellen blir ofte betraktet som et alternativ til objektmodellen. P.Rigaux, M. Scholl og A. Voisard [Rigaux 2002] skriver at man i en feltmodell har følgende; Hvert punkt i rommet er assosiert med en eller flere attributtverdier definert som kontinuerlige funksjoner av x, y og eventuelt z og/eller t. Målinger av flere fenomener kan bli samlet som attributtverdier som varierer med plasseringen i planet, for eksempel temperatur og forurensning. I motsetning til objektbasert modellering som ser på et fenomen som et objekt, ser feltbasert modellering på fenomenet som et kontinuerlig felt, det vil si det eksisterer verdier for feltet overalt i rommet. Feltmodellen kan benyttes hvor vi har å gjøre med et fenomen i virkeligheten uten fastsatte grenser**.** Også rasterbilder kan betraktes som en form for feltmodell, siden de sier noe om punkter i et todimensjonalt rom.

Feltmodellen kan imidlertid betraktes som en spesiell variant av objektmodellen, der modellen har objekter som ikke har en definert, fastlagt grense. Feltene kan beskrives ved hjelp av en eller flere romlige verdier. Eventuelt kan vi bruke aggregater (se avsnitt ”3.9.1 Aggregat modellen”) der hvert punkt med tilhørende funksjonsverdier utgjør elementene i aggregatet.

3.4 Romlige verdier

Romlige verdier kan betraktes som en mengde av punkter. Ved dimensjoner utover 0 (det vil si at verdiene har utstrekning) vil verdiene være tette punktmengder[1]. Verdiene kan i teorien ha vilkårlig mange dimensjoner, men i sammenheng med geografiske informasjonssystemer greier vi oss som oftest med maksimalt fire dimensjoner – x,y,z og t

[1] En tett punktmengde består av et uendelig antall punkter. Mellom to punkter kan det alltid legges inn et punkt til.

Romlige verdier angir form og/eller plassering i et rom av en viss dimensjonalitet. Dimensjonen av en verdi må være mindre eller lik dimensjonaliteten på det tilhørende rommet: En tredimensjonal verdi som en kuleform er en umulighet i en todimensjonal verden som et ark papir, mens en todimensjonal verdi som en linje kan eksistere både på et todimensjonalt papir og i det tredimensjonale (evt. firedimensjonale) rommet.

I de enkleste tilfellene kan den tette punktmengden defineres matematisk på denne måten:

{p | betingelse som p må oppfylle, for eksempel 0<p<7}

Tabell 1 gir en oversikt over ulike typer ikke-romlige og romlige verdier i rom av ulik dimensjonalitet. Verdier av typen navn har overhodet ingen tilknytning til rommet. Et punkt har ingen utstrekning, og kan eksistere i et rom med vilkårlig dimensjonalitet. En linje kan eksistere i rom med dimensjonalitet 1 eller høyere. Imidlertid tvinges linjen til å være rett i det endimensjonale rommet, mens den i høyere dimensjoner kan svinge og eventuelt gå tilbake til seg selv (eller krysse segs elv). På tilsvarende måte blir det for de høyere dimensjonene.

Tabell 1. Verdier og rom (etter [Skagestein 2002])

Virkelighetens Dimensjoner		Verdiens dimensjoner				
		0	1	2	3	4
	0	Ikke-romlige verdier (for eksempel navn)				
	1	Punkt	Rett linje			
	2	Punkt	Kurve	Flate		
	3	Punkt	Kurve	Flate	Volum	
	4	Punkt	Kurve	Flate	Volum	Hypervolum

I litteraturen treffer vi på begrepet "fuzzy objekter" – se for eksempel [Ameskamp 1997]. I den geografiske verden finner vi mange slike objekter, fordi det er umulig å fastlegge skarpe for hvor et objekt starter eller slutter. Hvor "begynner" egentlig et fjell? Og i hvilket område finnes jordsmonn av en bestemt type eller beskaffenhet?
Det som skiller "fuzzy objekter" fra andre objekter, er at de er assosiert med en uskarp romlig verdi. Vi vet altså ikke nøyaktig hvor punktene i punktmengden befinner seg. En vanlig løsning er for hvert punkt å angi en sannsynlighet for at punktet er med i punktmengden. Det er verd å legge merke til at vi opererer med uskarphet også for vanlige verdier som desimaltall, i og med at vi har en konvensjon som går ut på at vi ikke angir flere desimaler enn det vi mener det er grunnlag for.

Det er verd å legg merke til at dimensjonen for en romlig verdi svært ofte er noe som kan velges under modelleringen. Eksempelvis er formen og beliggenheten på en by strengt tatt en tredimensjonal verdi i et tredimensjonalt rom, men for praktiske formål modelleres den som regel som en todimensjonal verdi i et todimensjonalt rom (og kan vises frem som en flate på et kart) eller som en null-dimensjonal verdi i et todimensjonalt rom (og kan vises fram som et punkt på et kart). Vi ser at hvordan vi velger å modellere, er avhengig av hvilken skala (målestokk) vi arbeider i og dermed hvor detaljert vi ønsker at modellen skal være. I en stor målestokk er vi som oftest interessert i både formen og beliggenheten, mens vi i liten målestokk er interessert i bare beliggenheten.

3.4.1 Operasjoner på romlige verdier

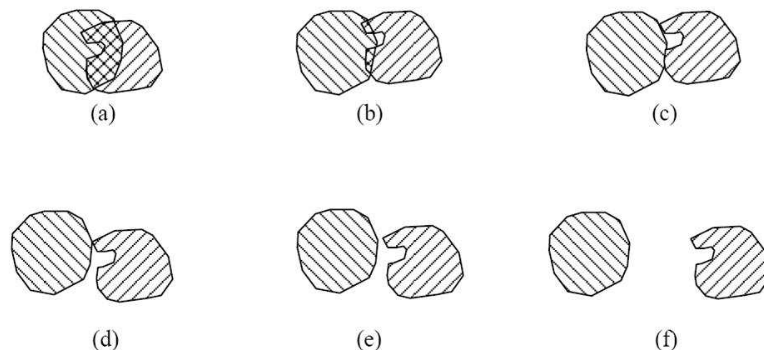
De vanligste operasjoner på romlige modeller er å gjøre beregninger av lengde og areal av romlige egenskapen til objekter, eller å finne avstand mellom romlige egenskaper til objekter. Denne type operasjoner er nyttige for plasseringsbaserte tjenester, slik som "returner maks 5 holdeplasser nærmest til universitetet, og avstanden til hver holdeplass". Andre vanlige operasjoner er:

- Beregning av areal av todimensjonale verdier.
- Beregne avstand mellom romlige verdier til to objekter.
- Beregne lengden av romlige verdien til et objekt.
- Beregne et nytt verdiobjekt som den topologiske differansen av to romlige verdier.
- Beregne et nytt verdiobjekt som den topologiske unionen av romlige verdier.
- Finne ut om to romlige verdier er innenfor en spesifisert avstand fra hverandre.
- Finne ut hvilken romlig verdi som er nærmeste nabo til en romlig verdi.

Av spesiell interesse er de såkalte topologiske operasjonene, som teoretisk sett kan betraktes som operasjoner på de tette punktmengdene som beskriver de romlige verdiene.

3.4.1.1 Topologiske operasjoner

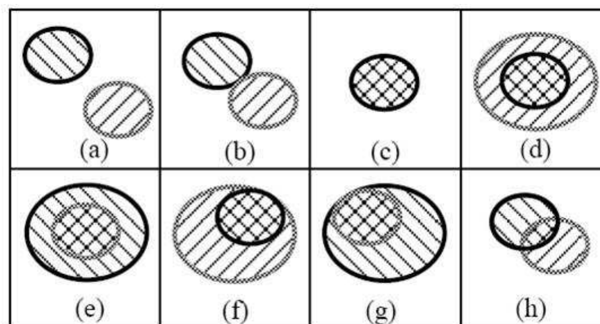
Man kan finne topologiske forhold mellom to objekter, og det binære topologiske forholdet mellom to objekter er basert på de fire snittene mellom grenselinjer og indre



Figur 9 Topologiske forhold mellom romlige egenskaper til to objekter – hentet fra [Egenhofer]

Ovenfor er det en figur –figur 9- som viser topologiske forhold mellom romlige egenskaper til to objekter.
Figur 9a viser *overlapp* eller *snitt* mellom to objekter. Karakteristikk for denne type topologiske forhold er relasjonen mellom objektenes romlige egenskaps grenselinjer og indre. Grenselinjene faller sammen på to punkter, grenselinjen for hvert romlig egenskap løper gjennom den andre romlige egenskapens indre, og begge indre sammenfaller delvis. Topologiske forholdet mellom to objekter forblir det samme så lenge karakteristikkene blir ivaretatt (figur 9b). Den vil forandre seg med en gang felles delene er de eneste sammenfallende grenselinjer delene, mens de indre har ingenting til felles med det andre objektets deler(figur 9c) denne type forhold kalles for *berøring* eller *møter*. Det gjør ingen forandring i det topologiske forholdet mellom to romlige egenskaper om de har en mindre kant til felles (figur 9d) de vil fortsatt ha det topologiske forholdet *berøring* eller *møter*. Figur 9e viser det topologiske forholdet mellom to romlige egenskaper hvor romlige egenskapene ikke lenger *berører* hverandre, men de er *disjointe* fra hverandre. Selv om B flyttes lenger fra A, vil

det topologiske forholdet mellom disse to romlige egenskapene fortsatt være det samme (figur 9f).
Nedenfor kommer en detaljert definisjon av de åtte topologiske forhold.



Figur 10 Eksempel på de åtte topologiske forhold –hentet fra [Egenhofer]

Figur 10 viser de åtte topologiske forhold mellom to punktmengder, begge punktmengder har sammenhengende grenselinjer, er 2 dimensjonale og er i R^2. Nedenfor er det en forklaring til det bildet viser:

a) **Disjoint:** Hvis alle fire snitt mellom alle fasettene er tomme da er to punktmengder disjointe.
b) **Møter:** Hvis snittet mellom grenselinjene er ikke tomme, men alle andre tre snitt er tomme, da møtes to punktmengder.
c) **Like:** To punktmengder er like hvis både snittet av grenselinjene og indre er ikke tomme, mens de to grense-indre snitt er tomme.
d) **Inni:** En punktmengde er A er inni et annet punktmengde B, hvis A og B deler et felles indre, men ikke grenselinjen, hvis A's grenselinje er en delmengde av B's indre, og ingenting av B's grenselinje snitter med noen del av A's indre.
e) **Inneholder:** En punktmengde B inneholder en punktmengde A (se definisjonen for **Inni**).
f) **DekketAv:** En punktmengde A er dekketAv et annet punktmengde B, hvis begge punktmengder deler en felles grenselinje og indre, hvis deler av A's indre snitter med deler av B's grenselinje, og ingenting av B's indre er del av A's grenselinje.
g) **Dekker:** En punktmengde B dekker et annet punktmengde A (se definisjon av **DekketAv**).
h) **Overlapp:** to punktmengder overlapper hvis de har felles grenselinjer og indre, og begge grenselinjer snitter med de motsatte indre.

3.5 Skranker

En skranke (constraint) er en regel som alltid skal være tilfredsstilt – som for eksempel at en by kan ligge bare i et eneste land, og må ligge i et land, eller at en person må ha en eneste biologisk mor. En skranke er altså invariant – den skal alltid gjelde, uansett hvordan verden og modellen utvikler seg. Skrankene skal gjenspeile de regler som gjelder i interesseområdet vi modellerer. Disse reglene kan være naturlover, administrative regler, juridiske regler og andre regler innenfor virksomheten vi modellerer (se [Skagestein 2001] kapittel 7). Ved å sette skranker inn i modellen får vi en bedre forståelse av virkeligheten. Men skrankene kan også tas inn i realiseringen av et informasjonssystem på ulike måter slik at systemet dermed kan bidra til en høyere datakvalitet.

Skrankene kan uttrykkes grafisk (se for eksempel [Skagestein 2002] kapittel 7) eller ved hjelp av et språk egnet for formålet, som OCL (Object Constraint Language) [Warmer 1999].

Modellering med skranker er velkjent fra datamodelleringsfaget. Det nye er at vi nå må uttrykke skranker som delvis involverer romlige verdier. Eksempler er at de enkelte deler i et vassdrag må henge sammen, og at vann alltid må renne nedover. Dersom det finnes en assosiasjon som uttrykker at en stasjon ligger etter en annen stasjon på en jernbanestrekning, må dette stemme med beliggenhetene for de to stasjonene.

Av spesiell interesse er de romlig/topologiske skrankene. Disse er nær beslektet med de topologiske operasjonene beskrevet i avsnitt 1.7.1.1, men poenget her er å få svar på om to romlige verdier eksempelvis overlapper, ikke overlapper, berører osv. Regelverket for noen vanlige sammenlikninger er vist i tabellen nedenfor.

Skranker for romlige verdier A, B er to romlige verdier, A° = A – grensen til verdi A	
En verdi befinner seg inni en annen verdi	(A er i B) = TRUE hviss[2] (A ∩ B = A) AND (A° ∩ B° = 0)
To verdier berører hverandre	(A berører B) = TRUE hviss (A ∩ B ≠ NULL)
Disjoint/A atskilt B	(A disjoint B) = TRUE hviss A ∩ B = Ø[3]
En verdi tilsvarer en annen verdi	(A samsvarer B) = TRUE hviss A ∩ B = A = B
En verdi inneholder en annen verdi	(A inneholder B) = TRUE hviss ((B er i A) = TRUE) AND ((A faller sammen med B) = FALSE

Tabell 2 Skranker for romlige verdier

3.6 Passive og aktive modeller

Det første steget i datamodellering er å designe en modell av den virkelige verden med fokus på de fenomener og objekter som vil være relevante for de aktuelle anvendelsene. Ifølge [Skagestein 2002] gir denne datamodellen (dvs. et dataorientert klassediagram eller et ORM-diagram) et statisk bilde av en del av virkeligheten. Dette gjelder selvsagt også for romlige modeller. Dersom vi skal utvikle forekomstene i modellen videre for å følge med virkeligheten og se hvordan det går, må dette skje ved hjelp av prosesser utenfor modellen [Skagestein, 2002]. Derfor kan laget over modellen (applikasjonen, forretningslogikken) bli relativt omfattende.

Objektorientert teknologi egner seg meget bra som plattform for systemer med romlige verdier fordi vi kan bruke objekter for å modellere både fenomener og deres romlige verdier. Det som skiller objektorientert teknologi fra andre teknologier er blant annet at metoder er definert sammen med data. Da er det nærliggende å benytte seg av mulighetene som foreligger ved at objektene selv kan utføre prosesser. Objektene kan da være både aktive og dynamiske og kan utføre en rekke funksjoner som kan gi ny verdi til systemet, istedenfor å være passive og vente på at en applikasjon kan ta dem i bruk. Vi får da en aktiv modell. Oppdateringer av et objekt kan selv løse ut oppdateringer av alle assosierte data, noe som sikrer nøyaktighet og samhandling for hele datasettet. Et objekt kan også inkludere kunnskap om tid: Det vil for eksempel si at et objekt kan endre geometri eller andre egenskaper avhengig av tid og årstid. For presentasjonsformål kan objekter vise seg fram på ulike måter avhengig av behovet: For eksempel kan det romlige attributtet til en by presenteres i full geometrisk detaljering med alle

[2] Hviss står for hvis og bare hvis.
[3] Ø står for tom mengde

veiene og lokalene i byen i stor skala, mens det i mindre skala kan presenteres som en enkel form (en enkel flate) og i enda mindre skala kan den presenteres som et punkt. Et annet eksempel er en elv hvor det romlige attributtet til elven vanligvis presenteres som en linje, men på tider av året når det regner mye kan presenteres som en flate.

Siden det er mindre behov for å operere på objektorienterte modeller utenfra, kan laget over modellen (applikasjonen) bli relativt tynt. Det er innlysende at modelltypen virker inn på hvordan vi ser på verden. Konklusjonen ifølge [Skagestein 2002] er at statiske modeller må manipuleres utenfra, mens aktive modeller kan ha eget liv.

3.7 Projeksjoner

Ulike projeksjonsteknikker blir anvendt for å kunne gi et forenklet bilde av virkeligheten. I følge Olve Øyehaug [Øyehaug] fins det to typer projeksjoner i plangeometri, parallell- og perspektivprojeksjon. Parallellprojeksjon inndeles inn i to typer, nemlig ortografisk og skjev projeksjon. Ortografisk projeksjon har man når man for hver projisering står vinkelrett med det fenomen man har tenkt å projisere: For eksempel kan man ta bilde vinkelrett i forhold til hver vegg av et hus. Ved skjev projeksjon kan vi ta med to vegger slik at vi ser huset slik at synsvinklene ikke står vinkelrett på veggene. Ved perspektivprojeksjon bestemmer vi oss for å se et fenomen fra et gitt perspektiv (for eksempel froskeperspektiv eller kjempeperspektiv) og deretter sette det inn i et koordinatsystem.

Når vi projiserer, avbilder vi et flerdimensjonalt fenomen inn i et rom som har færre dimensjoner enn fenomenets egentlige dimensjoner. Eksempelvis ønsker vi i forbindelse med kart å gjengi en beskrivelse av en tredimensjonal virkelighet, for eksempel jordoverflaten, på et todimensjonalt ark. En slik projeksjon kan være av typene plan-, kjegle- og sylinderprojeksjon. Velger man for eksempel å la en kules overflate blir beskrevet ved at man legger et ark på kulen og kopiere overflaten på dette arket rett opp, kalles dette planprojeksjon. Velger man derimot kjegle- eller sylinderprojeksjon kan dette gjøres ved at man legger et ark i en kjegleform eller sylinderform over den overflaten man velger å utføre projeksjon av. Sylinderprojeksjon blir blant annet utført når man skisserer hele jordoverflaten over på et ark.

Når vi velger projeksjon, må vi ta hensyn til at arealet, formen, avstander og retninger skal være mest mulig riktige. Alle typer projeksjoner vil føre til avvik enten i areal, retning eller form. Når vi for eksempel skal projisere jordkloden ved hjelp av sylinderprojeksjon fører dette til at områdene rundt ekvator blir nokså riktige, mens områdene i nord og sør blir strukket og ser litt annerledes ut enn virkeligheten, dette gjøres for at arealet skal forbli det samme.

3.8 Utvidelse av dimensjonalitet

La oss anta at vi har modellert to tredimensjonale objekter (dvs. objekter assosiert med tredimensjonale verdier), hvor det ene er plitt projisert og derfor modellert med to dimensjoner mens det andre er blitt modellert med alle tre dimensjoner. Hvis vi nå skal utføre operasjoner på disse to verdiene, foreligger det en inkompatibilitet i dimensjonaliteten av rommene. En mulig løsning er å utføre en projeksjon også på det andre objektet før operasjonene, men dette vil

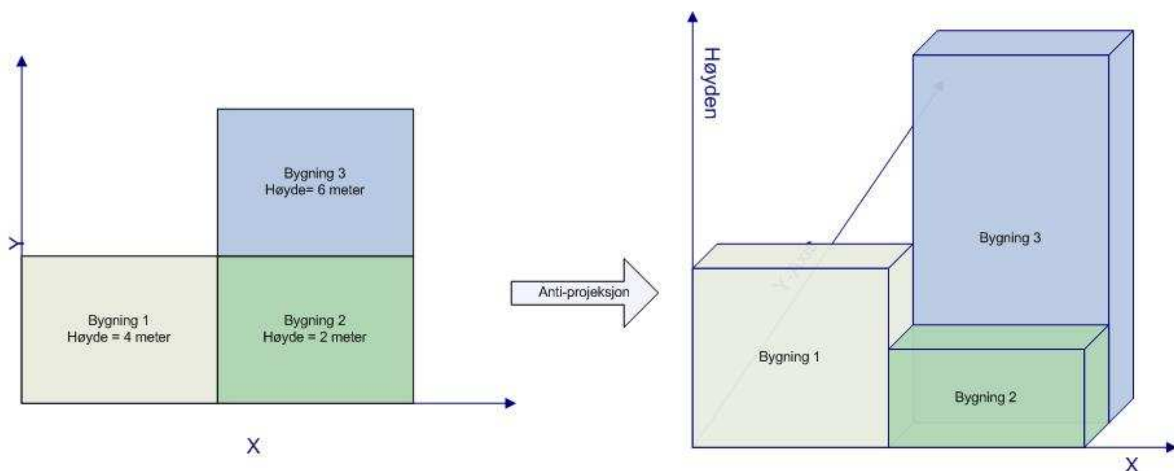
kanskje ikke gi oss de svarene vi ønsker. Derfor kan det være interessant å se på mulighetene for å kunne øke dimensjonen på det projiserte objektet, slik at operasjonene kan bli utført på en mer riktig måte.

Vi kan tenke oss tre ulike muligheter

1) At det i forbindelse med objektet finnes attributter med verdier for den ”manglende” dimensjonen
2) At vi trekker inn opplysninger fra en annen modell
3) At vi gjør enkle antagelser om verdien i den ”manglende” dimensjonen.

3.8.1 Lagrede verdier

Når det finnes attributter med verdier for den ”manglende” dimensjonen kan vi forsøke å utføre en ”anti projeksjon”.
Som nevnt tidligere velger man ved projeksjon et gitt antall av et fenomens dimensjoner, dette kan utføres på to måter, enten parallellprojeksjon eller ved perspektivprojeksjon. Som et eksempel kan vi ta et boligområde og velge å legge inn boligfeltet med perspektiv fra oven inn i et todimensjonalt koordinatsystem. La oss anta at vi nå har hatt denne modellen en god stund, og at boligfeltet ikke eksisterer i vårt UoD. Dermed vil det dermed ikke være mulig å modellere boligfeltet på nytt med det antall dimensjoner vi nå ønsker. Dette kan løses ved å se på hvordan projeksjonen ble utført. Her valgte vi perspektivprojeksjon. Perspektivet som ble valgt var fra oven og rett ned, i denne retningen vil vår ikke eksisterende dimensjon ligge. Boligene i dette feltet har videre verdier for høyden lagret hos hvert enkelt av boligobjektene, og dette kan brukes til å bli verdiene langs den tredje aksen.

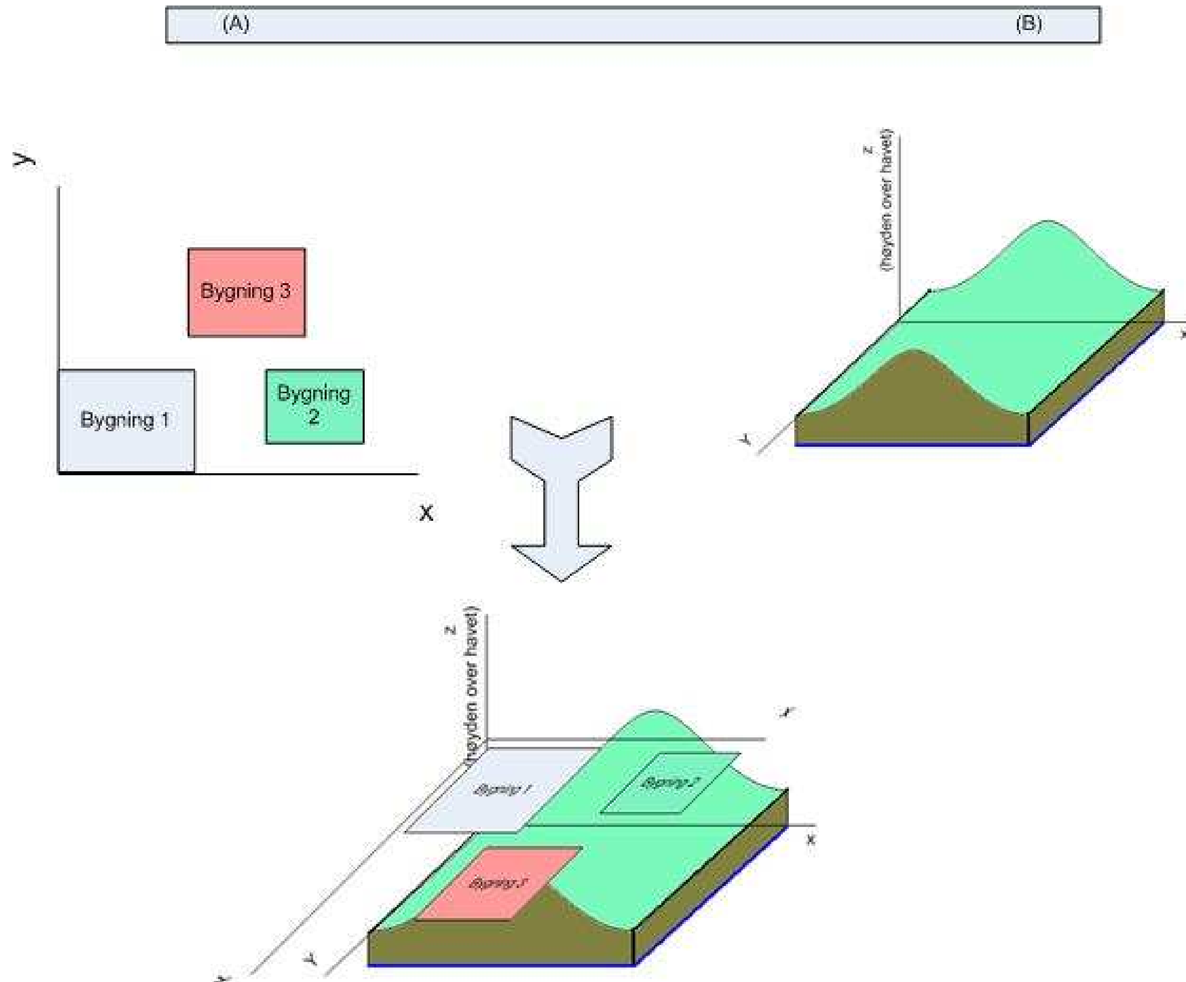


Figur 10 Anti projeksjon fra todimensjoner til tredimensjoner

3.8.2 Bruk av to modeller

La oss anta at romlige verdier for objektene i den ene modellen ikke har de verdiene vi trenger for å øke dimensjonaliteten. Det som kan gjøres da er å se om en annen modell har med den dimensjonen vi trenger. Det kan forklares ved å se på et enkelt eksempel hvor vi har en modell (A) over bygninger på et gitt sted, modellen med bygningen er i to dimensjoner og har ikke lagret høydeverdiene for enhver bygning som ved figur 10. En annen modell (B) består av tre dimensjoner hvor man beskriver høyden over havet, disse to modellene beskriver eksakt samme

sted. Siden modellen (B) gir høyden og det eksisterer en skranke som uttrykker at en bygning må stå på bakken, kan disse modellene kombineres for å et tredimensjonalt bilde av virkeligheten der vi fortsatt ikke har høyden på bygningene men nå vet hvor høyt de ligger over havet. For eksempel kan den nye modellen vise hvilke hus som vil rammes hvis det blir flom over visse nivåer.



Figur 11 Anti projeksjon mellom to eksisterende modeller

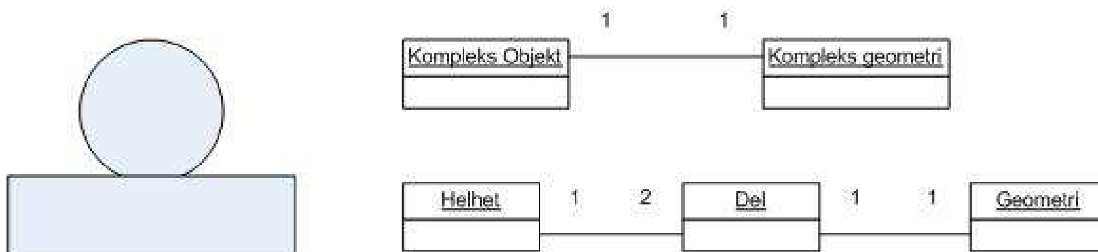
3.8.3 Dimensjonsutvidelse med en konstant

Denne typen utvidelse av dimensjonalitet kan betraktes å være den enkleste. Her antar vi at en verdi rett og slett kan "trekkes" utover i den manglende dimensjonen. Dette er spesielt aktuelt i forbindelse med tilføyelse av t-dimensjonen for objekter som ikke endrer seg over tid.

3.9 Monolittiske objekter vs. sammensetninger og aggregater

Anta at vi har et fenomen med en romlig utstrekning med en litt komplisert geometri i x, y, z og/eller t – som for eksempel pyntefiguren i figur 12. Denne pyntefiguren kan enten modelleres som et eneste monolittisk objekt assosiert med et litt komplisert geometriobjekt,

eller som en sammensetning eller et aggregat av to delobjekter, der hvert delobjekt er assosiert med hvert sitt mye enklere geometriobjekt.



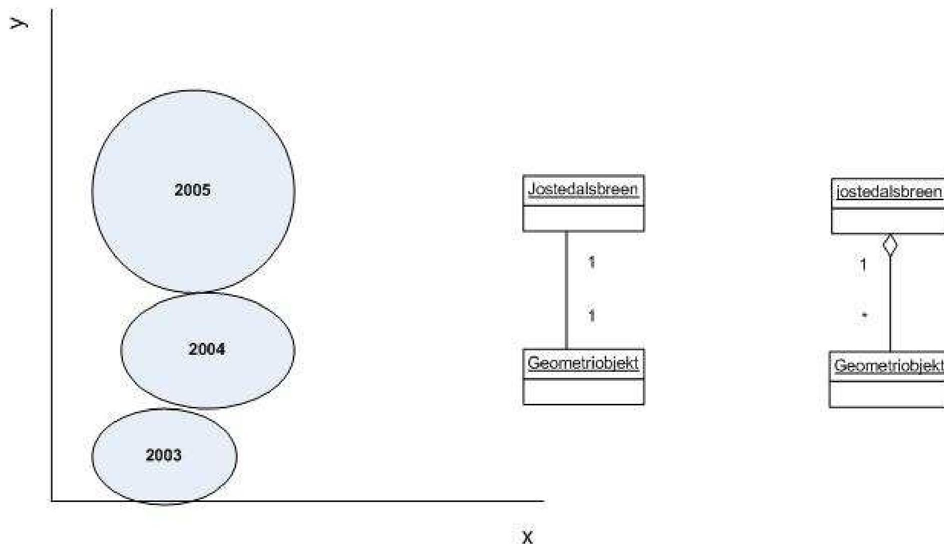
Figur 12 Pyntefigur

Hvilken av disse to modellene skal vi velge?

Det avgjørende kriterium for å velge en sammensetning eller et aggregat er om delobjektene er meningsfylte, det vil si om vi ser tilsvarende fenomener i virkeligheten. Hvis eksempelvis de to delene av pyntefiguren er laget av ulike materialer, og dette faktum er innenfor interesseområdet, skal modellen helt klart formes som en sammensetning eller et aggregat. Vi mener at en komplisert geometri er alene ikke god nok begrunnelse for å dele opp et fenomen.

Spesielle forhold gjør seg gjeldende når vi betrakter et fenomen over tid, og denne tiden strekker seg innover i en ukjent fremtid der vi ennå ikke kjenner hvordan fenomenets romlige form vil utvikle seg. Vi har da en situasjon der vi enten kan tilordne fenomenet stadig nye geometriobjekter som da inkluderer t-aksen og hvor man ”kaster” de gamle, eller vi kan betrakte fenomenet som et aggregat av stadig nye delobjekter som kommer til etter hvert som tiden går.

Imidlertid er det fremdeles slik at delobjektene som kommer til etter hvert bør ha sine paralleller som erkjennbare fenomener i interesseområdet. Det at fenomenet bare forandrer form, er muligens ikke nok. Jostedalsbreen i år 2005 er samme bre som Jostedalsbreen i 2004, selv om den er blitt mindre. Hvis det derimot er naturlig å knytte ytterligere opplysninger til delobjektene, må vi bruke et aggregat. Så hvis vi ikke bare skal ha den nye formen på isbreen ved utløpet av hver tidsperiode, men også eksempelvis gjennomsnittstemperaturen for hver av de samme tidsperiodene, må helheten Jostedalsbreen aggregeres fra delobjektene Jostedalsbreen-2004, Jostedalsbreen-2005 osv.

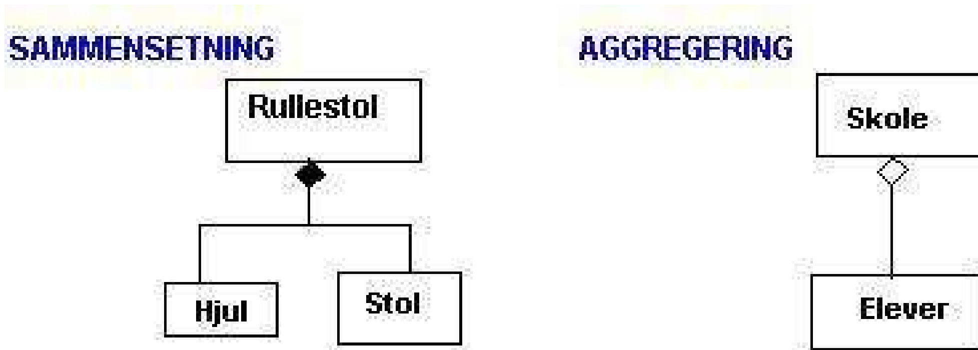


Figur 13 Jostedalsbreen, 2003-2005

I akkurat dette eksemplet kan vi også tenke oss en modell der fenomenet isbre er assosiert med et geometriobjekt med tidsakse, og at temperaturutviklingen er modellert som en endimensjonal kurve i det todimensjonale rommet T, t. Så kan vi bruke en romlig operator mellom de to geometriobjektene for å finne sammenhengen mellom formen på isbreen og gjennomsnittstemperaturen.

I UML skilles det mellom aggregater (aggregates) og sammensetninger (compositions).

I et aggregat kan delene i helheten gå inn og ut, mens i en sammensetning danner helheten og delene en uforanderlig enhet.

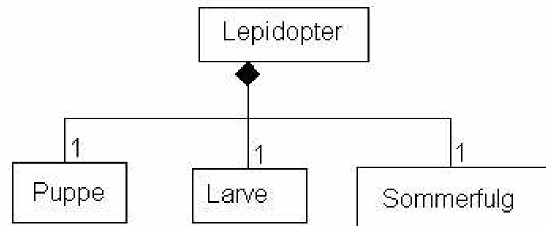


Figur 14 Sammensetning og aggregering

La oss betrakte et eksempel hvor man har en rullestol som består av hjul og stol. Dette kan betraktes å være et objekt med komplisert geometri eller to mindre objekter med hver sin enkle geometri. Før man bestemmer seg for sammensetning eller aggregering er det viktig å kunne bestemme om man skal dele inn objektet eller ikke. Hva applikasjon skal brukes til og hvordan den skal brukes bestemmer om objektet skal inndeles eller ikke. Et eksempel som kan forklare forskjeller på hvorfor man skal velge den ene eller den andre er å se på rullestolen og en skole. En skole består av elever og vil forbli skole selv om elver går inn og ut av skolen, rullestolen derimot er ikke lenge rullestol hvis hjul eller stol forsvinner.

3.9.1 Aggregatmodellen

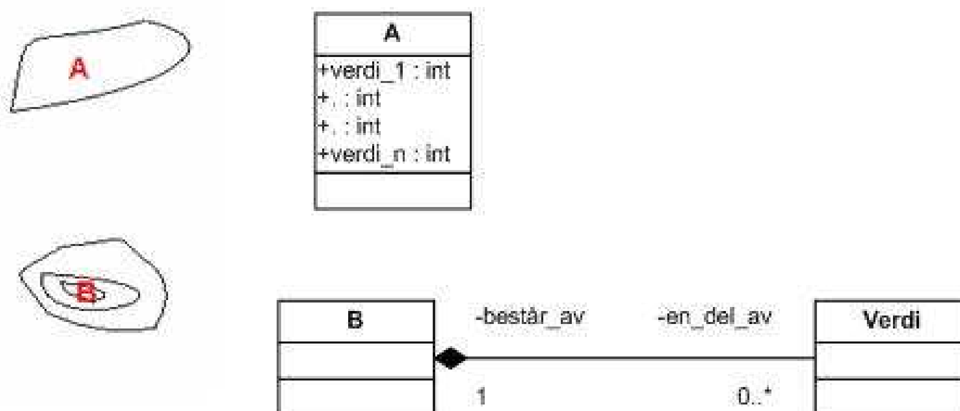
Puppe, larve og sommerfugl kommer inn under lepidopter, og selv om lepidoptet egentlig er en og samme fenomen er forandringene fra de forskjellige stadiene så drastiske at det er mer hensiktsmessig å betrakte dem hver for seg. Dermed kan puppe, larve og sommerfugl stå i aggregat forhold til lepidopten.



Figur 15 Lepidopten med aggregatforhold til de ulike fasene, inspirert av Cris Partidge [Partidge, 1996]

Videre kan man se på isbre eksemplet. Siden isbreen over en lang tidsperiode vil forbli samme fenomen kan det være hensiktmessig å betrakte fenomenets forandringer som forskjellige verdier ved forskjellig tid, i stedet for å tesselere isbreen slik at den blir å oppfatte som et aggregat. Dermed bør t aksen bli betraktet på lik linje med romlige verdier når forandringene på fenomenet er av en slik type at fenomenet ikke går over i en annen form. Med form menes ikke den romlige formen, men heller den forandringen som endre et fenomens egenskaper drastisk, som for eksempel at vann går over til gass eller larve går over til puppe.

Representasjon av et objekt med raster fører til at objektet ser ut til å bestå som oftest av mindre felter som til sammen utgjør et helt objekt. Skal objektet så står i aggregatforhold med de mindre feltene eller skal man ta hensyn til tidsdimensjonen? Svaret på dette spørsmålet kan gis ut ifra hvordan de forskjellige feltene avviker fra hverandre. La oss ta for oss et eksempel hvor vi har to fenomener, A og B. Fenomen A har like verdier over det hele mens fenomen B har forskjellige verdier og egenskaper ved visse steder, se figur 16. Uavhengig om de blir representert med raster eller vektorrepresentasjon vil de to fenomenene bli betraktet som forskjellige på det grunnlag av at den ene kan bli delt inn i deler med ulike verdier.



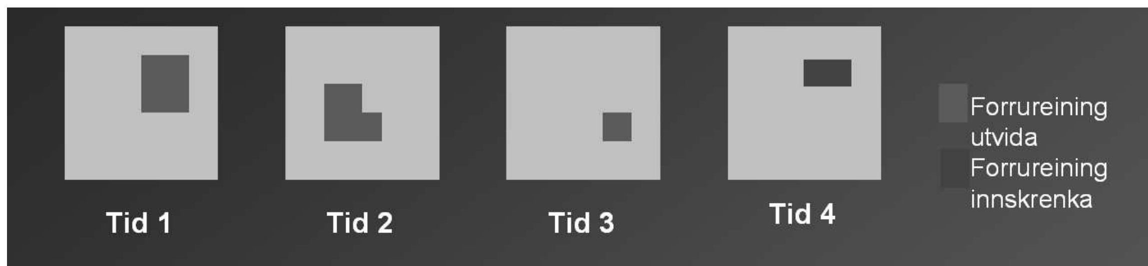
Figur 16 Fenomen A og fenomen B

Hvis vi per i dag har en bekk og modellerer dette fenomenet som objektet ”bekk”, hva skal man gjøre når det senere på året regner så mye at bekken nå kan klassifiseres som en liten elv? Med

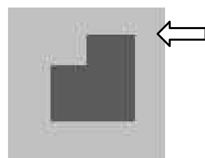
tidsperspektivet kan man kun være sikre på å kunne modellere objekter med verdier fra fortiden til nå, men det er ved fremtidlige verdier at man kan få problemer.

3.9.2 Delta-modellen

En hendelsesorientert modell følger aggregat-tankegangen, men i denne modellen ser vi bare på *endringene* fra et tidspunkt til et annet. Dette kan gi utfordringer når vi skal finne tilstanden til et objekt på et gitt tidspunkt: Vi må da gå gjennom alle tidligere endringer etter siste totalmodell og "legge" dem sammen med totalmodellen.



Figur 17 Hendelsesorientert modell, figuren er hentet fra [Midtbo]

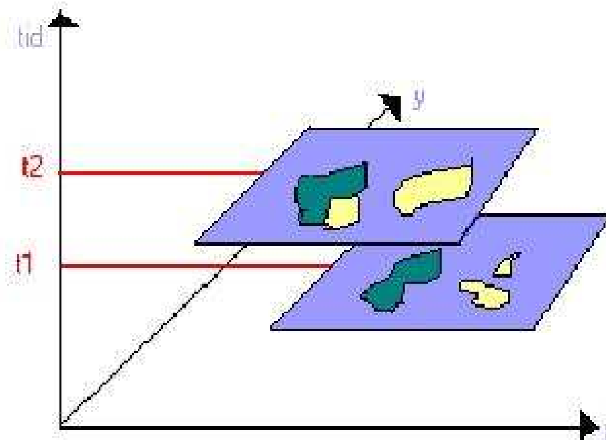


Hvis man ønsker å se tilstanden til objektet ved tid 3 får vi et resultat som ser slik ut. Denne modellen er lite hensiktsmessig hvis man skal bruke modellen til å undersøke hvordan fenomen har utviklet seg over forskjellige tider.

3.9.3 Forening av verdi og aggregat

I følge Martin Erwig og Markus Schneider [Erwig] kan man modellere objekter og deres utvikling fra en fortid til nå ved hjelp av dimensjonalbilder**.** Bruken av dimensjonalbilder er i og for seg brukbar men gir kun muligheten til å visualisere utviklingen fra en fortid og frem til en nåtid. Dimensjonalbilder gir snapshots som i riktig rekkefølge kan brukes som film for å kunne visualisere hvordan objekter utvikler seg. Denne måten å bruke dimensjonalbilder på er allerede mye brukt, for eksempel i værmeldinger hvor man ser hvordan temperaturen endrer seg på et område over en tidsperiode. Problemet med en slik måte er at man ikke har noen form for kontroll over hvordan filmen skal spilles. Dette kan for eksempel løses ved å legge inn en metode som øker brukerkontakt med filmen som blir laget.

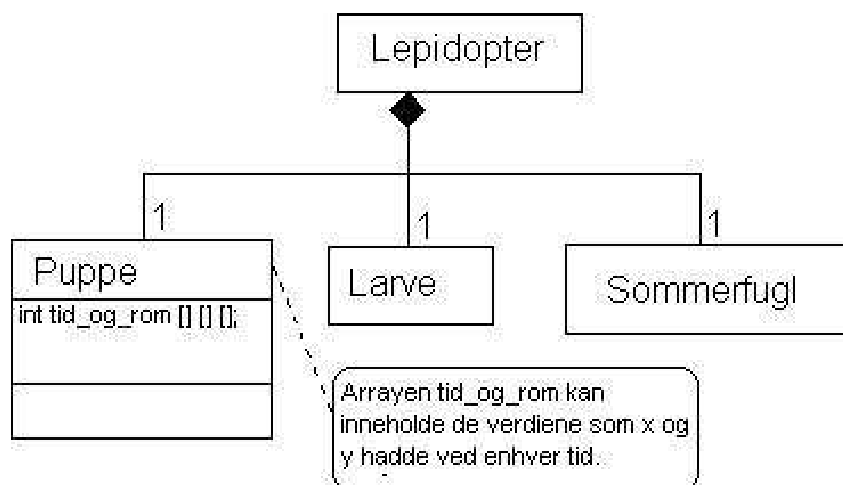
Dette er en veldig enkel måte å løse problemet på, men en mer sofistikert måte er å legge inn snapshots i et koordinatsystem. Dette vil gjøre det mulig å kunne sammenlikne hvilken som helst to snapshot og se hvordan de to utviklet seg over en viss tidsperiode.



Figur 18 Dimensjonalbilder inndelt etter lag

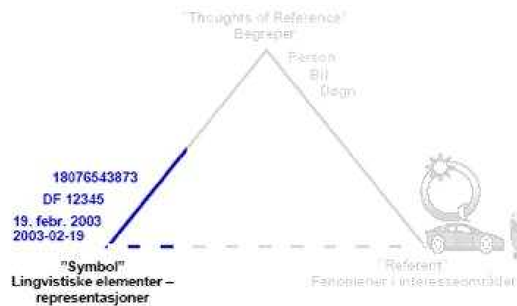
Som vi ser, kan hvert snapshot sees på som en mindre del av en helhet akkurat som ved snapshot, spatio composite data, hendelsesorienterte og objektorienterte modeller. Når disse snapshotene blir lagt inn i et koordinatsystem kan dette virke som om snapshotene følger en dimensjonal modell. Slik kan man få utført interpolasjon mellom snapshotene, og mellom "figurene" på de enkelte snapshotene. Denne foreningen av verdi og aggregat kan føre til at man får samlet fordeler som de enkelte modellene hadde hver for seg.

Vi kan også forene disse metodene slik at noe av et objekt blir del inn i mindre deler og noe annet blir stående som verdier i stedet for aggregater. La oss gå tilbake til lepidopter eksempelet. Hvis vi ser på den tidsperioden et lepidopter er i en puppe stadiet, endrer verdiene til puppen over en gitt tid, disse verdiene kan være av typen overflate og volum. Siden vi inndeler lepidopterlivet inn i tre faser, puppe, larver og sommerfugl, kan man tenke seg at de forandringene som skjer på puppen også kan behandles som aggregat. Dette er ikke hensiktsmessig, og det man heller kan gjøre er å la verdiene som endrer seg over tid stå som verdier og la aggregatforholdene fra puppen stå som de står. Slik forener vi aggregatmodellen og dimensjonsmodellen i et og samme objekt.



Figur 19 Forening av dimensjon og aggregat metoden

4 Representasjon av romlige verdier



Figur 20 Romlige verdier

Man kan tenke på fenomenets form og plassering representert som en mengde av et uendelig antall punkter i rommet, hvor hvert enkelt punkt har eksempelvis koordinater med to verdier (x, y) ved to dimensjoner. Stedfestingen kan også være i form av koordinater i tre dimensjoner x, y, z eller fire dimensjoner x, y, z og t.
Generelt har man at antall verdier i koordinater bestemmes av dimensjonaliteten i rommet vi velger å operere med, jamfør kapitel "5 Koordinatsystemer". Det er ikke mulig å få en slik punktmengde representert direkte i maskinen. Derfor bruker vi vektorrepresentasjon, rasterrepresentasjon eller halvplan for å representere romlige verdier.
Generelt kan modeller i OCL kan brukes til å beskrive romlige primitiver som for eksempel punkt, linje eller flate som er hovedelementene i en vektormodell og rastermodellen. De kan også brukes til å beskrive geometriske primitiver som for eksempel sirkel, trekant og pyramider.

4.1 Rasterrepresentasjon

Rasterrepresentasjon representerer i følge Rigaux, Scholl og Voisard [Rigaux, 2002] geometrien til fenomener i virkeligheten ved hjelp av et kontinuerlig sett med celler ordnet etter kolonner eller rader. Raster har som vektorer et koordinatfestet origo. Kjenner man antall rader og kolonner samt cellenes enhetsstørrelse (oppløsning) kan hvert punkt innenfor en rasterbasert representasjon entydig refereres. De romlige egenskapene er koblet til rastercellene. Detaljeringsgraden for både rasterepresentasjon og vektorrepresentasjon er avhengig av oppløsningen Med høy oppløsning øker datamengden
.
Rasterrepresentasjon har en eller flere verdier for et begrenset område. Verdiene som lagres for en celle kan være:

- enten gjennomsnittlig verdi for hele cellen
- eller verdien i cellens sentrum eller verdien av en kant node/ eller et hjørne

4.2 Vektorrepresentasjon

Ifølge Rigaux, Scholl og Voisard [Rigaux, 2002] består en vektorrepresentasjon av en ordnet liste av x og y koordinater (punkter), samt interpolasjonsregler som gir mulighet for å avlede andre punkter. Vi kan skille mellom følgene vanlige spesialtilfeller:

1) Punkter, representeres med en koordinat med like mange verdier som dimensjonaliteten som punktet ligger i, jamfør tabell 1.
2) Linjer, representeres med minst to punkter

3) Polyline er representert ved en liste av punkter <p_1,.......,p_n>, hvor hver p_i er et hjørne, og hvert par (p_i, p_{i+1}) representerer et av polyline's kanter.
4) Polygon, er også representert av en liste av punkter, men listen representerer en lukket polyline, og derfor et par (p_n, p_1) er også en kant av polygonet.
5) Region, et sett av polygoner.

En meget vanlig vektorrepresentasjon er TIN-modellen. TIN-modellen er en vektorrepresentasjon for flater, dvs. todimensjonale verdier i 3D-rommet.

En TIN-modell er en topologisk datastruktur som håndterer informasjon om noder som innbefatter hver trekant og naboene til hver trekant. Som med andre topologiske datastrukturer, kan informasjon om en TIN-modell lagres i databasetabeller.
TIN-modeller brukes mye for å modellere terrenger (digitale høydemodeller – DEM).. Områder med ujevnt terreng kan representeres med flere punkter enn områder med mer jevnt terreng, som kan representeres med færre punkter.
TIN-modeller gir fordeler for overflateanalyser. Variabel tetthet av trekanter betyr at en TIN er en effektiv måte for lagring av overflaterepresentasjoner.

En TIN-modell kan enkelt formaliseres ved hjelp av en skranke-datamodell. Representasjonen av et objekt som en punktmengde i et d-dimensjonalt rom kan ta fordel av den fundamentale egenskapen at et av attributtene kan defineres som en funksjon av en mengde av de andre attributtene. Denne funksjonen er en lineær interpolasjon basert på noen endelige mengder av prøveverdier.

En TIN-modell definerer en partisjon av 2D plan i trekanter T_i, ved å gi en høyde til hver av hjørnene i trekanten. Den interpolerte høyden *h* av et tilfeldig punkt p i planet er vanligvis beregnet ved å først finne T_i slik at T_i inneholder *p*. Verdien *h* er lineært interpolert av høydene av de tre hjørnene av T_i. Denne senere funksjonen avhenger bare av *i,* og kan defineres som en lineær funksjon $f_i(x, y)$, gyldig bare for punkter i T_i. Den naturlige symbolske representasjonen av en 3D-punktmengde TIN i lineær skrankemodellen er

$$TIN(x, y, h) = \vee_i ti(x, y) \wedge h= f_i(x, y)$$

der $t_i(x, y)$ er en symbolsk representasjon for trekanten T_i.

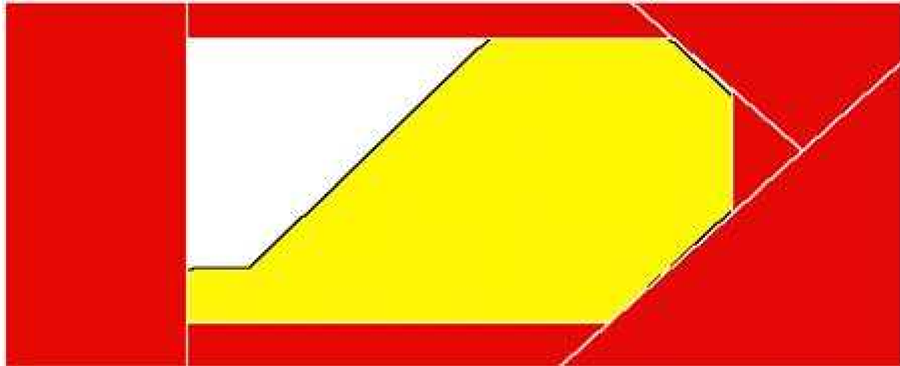
4.3 Halvplan representasjon

Et halvplan H i d-dimensjonal rom R^d kan defineres som en mengde av punkter $P(x_1, x_2,\ldots,x_d)$ som tilfredsstiller en ulikhet av formen

$$a_0+a_1x_1+a_2x_2+\ldots.. \leq 0$$

Derfor er en representasjon av H vektoren $[a_0, a_1, a_2,\ldots,a_{d+1}]$. En *konveks* d-dimensjonal *polytope* er definert som et snitt av en endelig antall av lukkede halvplan, som er avgrenset av linjestykker. Hvis H er en del av halvplanet som definerer P, kalles H ∩ P en fasett av P. Unionen av et endelig antall *polytoper* kalles en d-dimensjonal *polyhedron* Q i R^d. Q er ikke nødvendigvis konveks, og dens komponenter er ikke nødvendigvis sammenhengende, og de kan overlappe. Hver *polyhedron* deler rommet i dens indre, dens grense, og dens ytre. En konveks polygon med n kanter (n hjørner)er definert som snitt av n halv-plan avgrenset av linjer. En region er da en union av konvekse polygoner. Denne definisjonen gjør i stand til definisjonen av ikke enkle polygoner, regioner med hull, og enkle ikke konvekse polygoner.

En ikke konveks polygon kan ikke representeres med en 2D *polytope*, det vil si, den kan ikke lagres bare ved å bruke snitt av halvplaner. Isteden må den deles i konvekse deler og representeres med en *polyhedron* som er union av tilstøtende konvekse deler.



Figur 21 Et eksempel for hvordan man beskriver en polygon med halvplan representasjon. Dette er en ikke konveks polygon som må deles i konvekse deler og representeres med en polyhedron. Figuren er hentet fra [Skagestein 2004]

Halvplan i 3D:
Et halvplan i 3D er en skranke over disse tre variablene x, y og z, uttrykt som følgende:

$$ax+by+cz+d \leq 0$$

Muligheten for å representere og manipulere objekter innbefattet i 3D-rommet viser seg å være brukbar i modellering av felt basert data, som betrakter fenomenene som en kontinuerlig funksjon definert over rommet. Under denne synsvinkelen er det ingen identitet assosiert med en mengde av punkter. Isteden er en attributtverdi assosiert med hvert punkt: funksjonsverdien i dette punktet. Vi skal se på to applikasjoner for å illustrere hvordan slike data kan modelleres med skranker; *objekter i bevegelse og felt basert data.*
Vi beskriver hvordan slike objekter kan bli direkte modellert med en lineær skranke datamodell som uendelig, flate relasjoner.

4.4 Vektor, raster og dimensjoner

Med vektorrepresentasjon vil man for eksempel representere todimensjoner verdiene ved at punkter blir lagt inn der de passer i koordinatsystemet. Har et punkt verdiene lengde lik 2, bredde lik 4, tykkelse lik 5 og tid lik 1, vil man gå inn og sette inn dette punktet i et firedimensjonalt koordinatsystem på punktet (2,4,5,1).
Hos rasterrepresentasjon blir pixlenes dimensjoner bestemt av det antall dimensjoner rommet har. For eksempel vil en pixel i endimensjonalt rom bestå av endimensjonale pixler og ved tredimensjonalt rom vil pixlene ha tredimensjoner, også kalt voxel. En voxel er en tredimensjonal kube, og ved rasterrepresentasjon vil en mengde x, y og z eller t verdier ha et begrenset område med samme verdi. Man kan videre spørre seg det spørsmål om hvorfor det er nødvendig med voxel i stedet for piksel og todimensjonale celler, dette kan forklares ved hjelp av et relativt enkelt eksempel. La oss anta at vi har et objekt som har to romlige dimensjoner i tillegg til tidsdimensjonen og oppløsningen er satt til per meter/sekund. La oss videre anta at et fenomen beveger seg med en fart 0,5 meter/sekunder på en bestemt strekning. Videre kam man

stille seg det spørsmålet om hvordan dette kan representeres, benytter vi kun todimensjonale celler vil man for eksempel kun få verdier i en celle bestemt av en lengde x og et tidsintervall eller en x og en y lengde. Dette vil ikke gi hele virkelighetene og man er nødt til å legge inn et voxel som består av et intervall av x, y og t med kun en verdi i det avgrensede området.
Kort sagt unngår vektor denne problematikken ved flerdimensjoner, siden hvert vektorpunkt har en utstrekning på null dimensjoner mens raster har en utstrekning som er like stor som det antall dimensjoner rommet har. Vektor gir kun punkter, linjer eller avgrensninger mellom punktene blir bestemt av forskjellige interpolasjonsteknikker.

4.5 Representasjon av fuzzy verdier

Vi kan representere fuzzy verdier – for eksempel ved å legge en buffer rundt verdien, og la sannsynligheten for at et punkt i bufferen er med i verdien synke gradvis mot ytterkanten. Hvis verdien er et punkt, kan vi beskrive ”fuzzyness” ved hjelp av statistiske begreper som spredning og standardavvik .

5 Koordinatsystemer

Koordinatsystemer blir brukt til å bestemme et punkts plassering i det rommet man har for seg. Et enkelt matematisk koordinatsystem består av to akser, hvor den vannrette aksen blir kalt første akse eller x akse, mens andre aksen treffer første aksen normalt og blir oftest kalt y aksen. Generelt består et koordinatsystem av en origo og et gitt antall akser satt opp på en bestemt form med en gitt skala.

5.1.1 Origo

Origo, også kalt nullpunktet, blir valgt ut ifra det ståstedet man befinner seg på. Origo er det punktet som passer best til å være sentrum av vårt ståsted og som med hensyn på den delen av virkeligheten vi har tenkt å modellere, står stille fra start til slutt. Med andre ord velger vi origo til å være det fenomenet eller punktet som ikke endrer plassering underveis. Skifter vi så ståsted eller utvider den delen av virkeligheten vi har tenkt å modellere kan dette føre til at origo må byttes ut. Et eksempel som kan forklare dette er å la et hjørne i et vogntog være origo når vi kun velger å se på vognen. I forhold til for eksempel passasjerene og bagasjen i vognen vil hjørnet alltid befinne seg på den samme plassen, men hva skjer hvis vi skifter ståsted? La oss anta at vi nå velger vårt ståsted til å være stasjonen toget står på, vil det fortsatt være lurt å la et hjørne på vognen være origo? Kanskje i en liten stund, men hva skjer hvis toget begynner å bevege på seg? Da vil det for oss som står på stasjonen virke som om hjørnet beveger seg, og det gjør det egentlig ikke. Men siden toget beveger seg og vi har skiftet ståsted kan ikke hjørnet lenger være et hensiktsmessig valg av origo. Kort sagt bør origo velges å være det punktet eller det fenomenet som ved vårt valgte ståsted vil være i ro med hensyn på den delen av virkeligheten vi ønsker å modellere.

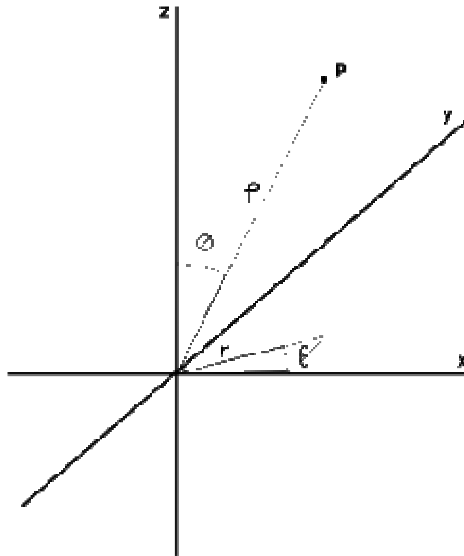
5.1.2 Akser og dimensjonalitet

Antall akser vi velger å ha i et koordinatsystem bestemmes av dimensjonaliteten for rommet i modellen. Det er ikke dermed sagt at denne dimensjonaliteten faller sammen med dimensjonen for de romlige verdiene til objektene i modellen. Den generelle regelen er:

ANTALL DIMENSJONER FOR ROMLIGE VERDIER $\leq$ ANTALL AKSER I KOORDINATSYSTEMET

5.1.3 Kartesiske koordinater vs. polarkoordinater

Koordinatene i et koordinatsystem kan plasseres på to forskjellige måter, enten på kartesisk eller polar form. Når et koordinatsystem er satt opp til å ha kartesiske koordinater er det slik at punktene i koordinatsystemer er helt uavhengig av hverandre og aksene kan inntil tre dimensjoner visualiseres til å stå 90° på hverandre. Ved polarkoordinater bruker vi istedenfor en eller flere vinkler og en radius r. Antallet vinkler økes avhengig av hvor mange dimensjoner man betrakter, ved for eksempel et koordinatsystem for et tredimensjonalt rom har vi to vinkler og en radius..



Figur 22 Polar koordinatsystem, hentet fra [www.wikipedia.com]

5.1.4 Skala og standarder

Enhetene på aksene blir også bestemt etter formål, det mest vanlige er kanskje bruken av lengdemål på x, y og z hvor lengdemål måles i meter. Tidsmål, det vil si på t aksen, kan bli målt i eksempelvis sekunder, minutter, timer eller døgn.
Siden både lengdemål og tidsmål kan bli målt opp i alt fra fot, sommer og vinter til millisekunder er det viktig å angi måleenheten. Vi foretrekker standardiserte enheter.

I GML indikerer for eksempel *uom*-attributtet at enheten av målingen for *height* egenskap er representert med en peker #m, som kan enten være en peker til et annet element i dokumentet, til en base URI spesifisert med en *xml: base* attributt, eller er en string som markerer en enhet av måling, for eksempel *m* indikerer lengde i meter.

5.2 Oppløsningsevnen

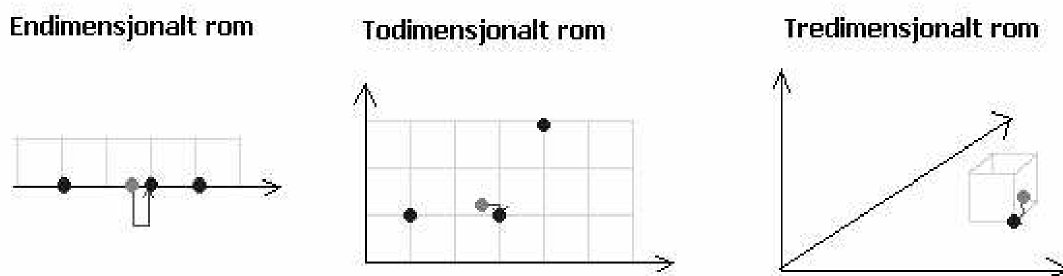
Siden datamaskiner har en endelig oppløsningsevne må man ty til avrunding av de dataverdiene man har å gjøre med. Disse avrundingene kan senere føre til feil ved matematiske beregninger, og ved avrundinger på verdier knyttet til geometrien kan det føre til at topologiske regler gir inkonsistens. Dette kan for eksempel gi overlapp mellom objekter i modellering mens de tilsvarende fenomener i virkeligheten slett ikke overlapper. Når man går fra virkelighet til modellering av virkeligheten trenger man å kunne oppdele virkeligheten.

La oss anta at vi skal modellere en veistrekning og velger å la oppdelingen eller målingene bli målt etter hver meter. Dermed har man oppdelt en virkelighet inn i veistykker på 1 meter og avrunder målinger. Oppdelingen av den fysiske modellen inn i mange beregningsområder kalles diskretisering. Denne oppdelingen blir bestemt av hvor nøyaktig brukeren skal modellere virkeligheten.

Når man begynner å representere rom- og tidverdier må man som ved alle andre ikke-skalare verdier ta hensyn til konsekvenser gjennomførte målinger kan få på grunn av diskretisering. Et eksempel på følger som kan forekomme på grunnlag av diskretiseringsvalg er hvis man skal

bestemme plasseringen til et fenomen i henhold til tiden. La oss se på et fenomen som beveger seg fra kl 13.45 til 13.50, resten av tiden står fenomenet på nøyaktig samme sted. Velger man å la tidsbegrepet gå over antall hele timer og interpolerer med en rettlinjet funksjon mellom timene, vil det si at man velger å la t-aksen (aksen til tiden) oppdeles i timer i stedet for minutter og det kan se ut som fenomenet beveger seg med en svært liten fart gjennom en hel time. Hvis man i stedet deler t-aksen i minutter gir det et mer riktig bilde av virkeligheten. Oppdelingen langs de resterende romlige aksene ville også blitt bestemt og vil på samme måte føre til "feil" resultater på grunn av diskretisering av disse aksene. På samme måte som t-aksen blir oppdelt i timer eller minutter kan man inndele de romlige aksene inn i blant annet centimeter, kilometer og meter. Forskjellen mellom kun en romlig virkelighet og en virkelighet som tar med temporale verdier er kun det at diskretiseringsnivået på en modell med kun romlige verdier kan sette opp aksene til å ha samme benevning mens ved et rom med tiden med i bildet må tiden bli oppdelt i en annen benevning enn de resterende romlige aksene.

Noe man kan lure på ved diskretisering er hva som vil være forskjellig fra en virkelighet med få dimensjoner til en med mange dimensjoner. Dette kan forklares enkelt ved hjelp av en figur som viser følgene av samme diskretiseringsnivå. Som figuren viser vil de målte punktene i et endimensjonalt rom forskyves til nærmeste diskretiseringspunkt, det samme skjer ved i et tredimensjonalt rom hvor man også går til nærmeste diskretiseringpunkt, men nå i tre dimensjoner.



Figur 23 Diskretisering og forskjellige dimensjoner

I visse tilfeller er man ute etter å ha en modell av virkeligheten som er så nøyaktig som mulig. For å få til dette må detaljeringsnivået økes og det samme skjer med diskretiseringsnivået. Er man heller ute etter å få raske svar kan man oppdele litt mer grovere med lavere diskretiseringsnivå. Dermed blir diskretiseringsnivået bestemt ut fra hva applikasjonene skal brukes til og hvordan de skal brukes. Det vil være unødvendig med en detaljert modell av virkeligheten når man kun ønsker å få et svar som for eksempel tilnærmet gjennomsnittsverdi eller andre spørringer hvor man er ute etter å få svar så raskt som mulig.

5.3 Avrundingsfeil, diskretisering

Diskretisering fører til at punkter i representasjonen kan bli liggende andre steder enn i virkeligheten. Dette er alvorlig hvis vi derved kommer i skade for å komme i konflikt med en eller flere topologiske skranker.

Realm ble foreslått som en metode for å håndterere avrunding av romlige systemer på en slik måte at det blir mest mulig riktig. Realm blir beskrevet til å være en brukerdefinert struktur som benyttes som basis for en eller flere datatyper. Realm som brukes for spatiale data er et sett av punkter eller et sett av ikke overlappende linjer. Realm er ment til å kunne beskrive hele geometrien til en

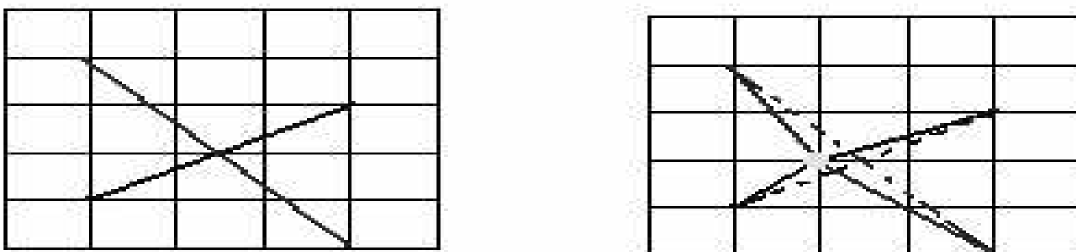
applikasjon, for eksempel linjer og punkter assosiert med et objekt. Ved modellering uten Realm kan objekter som snitter med hverandre i virkeligheten bli modellert slik at de ikke har noen snittmengde på grunn av begrenset oppløsning. Ved hjelp av Realm unngår man at dette skjer ved at en korrekt spesifikasjon retter opp og gir riktig topologisk svar.
Siden Realm gir korrekt topologisk svar vil fenomeners temporale data ikke ha noen form for innvirkning på selve spesifikasjonen, men ved forskjellig tid kan forhold som for eksempel snittet mellom to objekter endres i størrelse eller eventuelt bli lik tom mengde. Dermed må den brukerdefinerte strukturen kunne oppdateres og endres ved enhver tid. Man kan for eksempel løse dette ved å beskrive geometrien til en applikasjon gitt ved ethvert tidspunkt, for eksempel kan man si at to objekter vil snitte hverandre fra t_1 til t_2, og bli disjunkte i t_3.
David Skogan [Skogan] kommer inn på en rekke eksisterende metoder for å få bukt med problemer knyttet rundt diskretiserings- og avrundingsfeil. Han nevner blant annet at man kan øke oppløsningen og legge inn toleranseverdier. Spesielt ROSE algebra kan passe når man ønsker å svare på spørsmål som for eksempel om et beregnet krysningspunkt mellom to gitte linjer ligger på begge linjene.

5.3.1 Rose Algebra

Kort sagt er Rose Algebra et system bestående av spatiale datatyper og operasjoner mellom disse typene. Rose står for Robust Spatial Extension og er en realm basert algebra, fordi Rose Algebra benytter de datatyper som er blitt definert i Realm. I følge R. H. Guting og M. Schneider [Guting, 1995] inngår følgende operasjoner i Rose algebra:

- sammenligning av to spatiale verdier med hensyn på topologiske forhold (for eksempel intersect, inside og meets), hvor svaret er en boolsk verdi
- operasjoner som returnerer spatiale verdier som svar, for eksempel vil snittet mellom linje A og linje B returnere det punktet eller linjestykket hvor A snitter B
- operasjoner som returnerer nummer, for eksempel diameter, lengde og areal
- operasjoner hvor en mengde objekter blir operander og resultatet kan i noen tilfeller være et nytt objekt (for eksempel overlay og fusjon)

For å løse problemet om to linjers krysningspunkt ligger på begge linjene med hensyn på diskretiseringsvalget mener R. H. Guting og M. Schneider [Guting, 1995] at man kan endre linjene til å gå slik at de begge har krysningspunkt som et eget punkt. Dette kan vises ved hjelp av en figur - se figur 24, og som vi ser blir linjer forskyvet slik at krysningspunktet blir liggende på begge linjer.

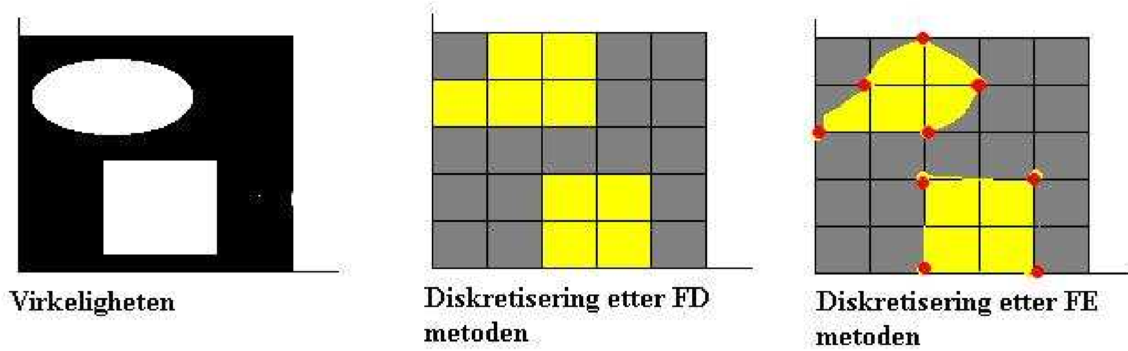


Figur 24 Rose algebra og snitt

5.3.2 Diskretisering i tre dimensjoner

I henhold til Adam Brun [Brun] kan diskretisering i tre diemnsjoner skje på to måter, enten ved å inndele den fysiske modellen inn i rektangulære områder eller ved å inndele inn i trekanter og polygoner. Den sistnevnte metoden kalles finite element (FE) metoden, den inndeler virkeligheten slik at man tar hensyn til et fenomens ytterpunkter og ved hjelp av interpolasjon begrenses de forskjellige elementene fra hverandre. Den førstnevnte metoden, finite difference (FD) metoden,

deler virkeligheten inn i "skoesker". FE-metoden følger en vektor-representasjons-ide, mens FD kan betraktes som et tredimensjonalt raster.

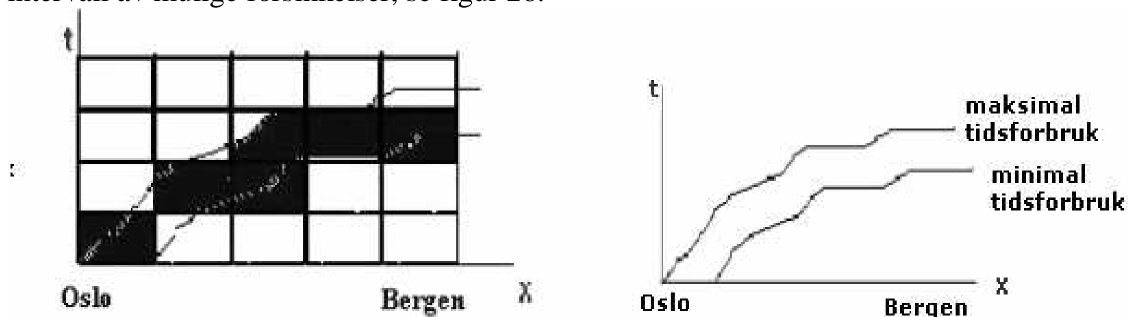


Figur 25 Diskretiseringstyper

5.3.3 Diskretisering ved hjelp av FD metoden

FD metoden på tredimensjonal romlig virkelighet for eksempel jordsmonn, kan i og for seg være hensiktsmessig. La oss anta at vi oppdeler virkeligheten inn i 1*1*1 meters blokker og for hver blokk finner man hvilken type jordsmonn man har å gjøre med. Har man flere typer jordsmonn i en blokk kan man inndele denne blokken inn i mindre blokker. Ved tiden med i bildet kan man inndele på samme måte og ved flere typer objekter i en blokk kan man inndele videre akkurat som ved romlig virkelighet. Problemet med tiden med i bildet er at man ikke på forhånd vet hvor mange blokker man i fremtiden må dele inn i mindre deler. Det samme kan i og for seg tenkes om en romlig virkelighet hvor man starter å modellere i et sted uten å tenke på hvordan blokkinndelingen vil bli senere.

Valg av FD metoden:
Når man har å gjøre med romlige fenomen uten tiden med i bildet kan FD metoden være mest hensiktsmessig når applikasjonen inneholder klart avgrensede fenomen. FD metoden kan også velges når fenomenenes vinkler er rettvinklede og formen er mer eller mindre rektangulært. Ved applikasjoner med tiden med i bildet, kan denne metoden være hensiktsmessig på er hvor man skal sette inn et intervall for verdier et fenomen kan ha, for eksempel togsporet fremkommelighet med et intervall av mulige forsinkelser, se figur 26.



Figur 26 Intervall med forsinkelser

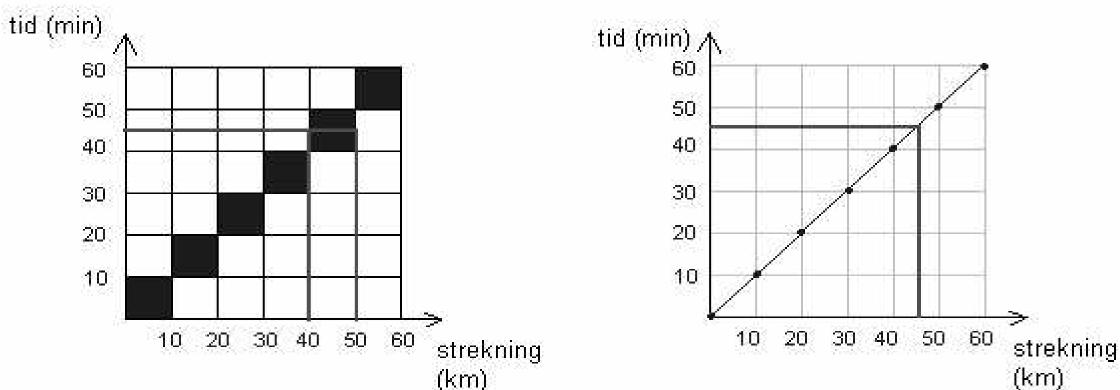
5.3.4 Diskretisering ved hjelp av FE metoden

Finite element metoden på en tredimensjonalt romlig virkelighet oppdelt i 1*1*1 meter vil avgrense slik at resultatet vil bli en samling trekanter eller polygoner. Finte element metoden kan også oppdele slik at man for hvert fenomen lagrer yterpunktene som kommer inn i diskretiseringsnivået, og ved hjelp av interpolasjon trekker linjer mellom disse ytterpunktene. Modellering av kun og rom og modelleringen med rom og tid vil ved FE metoden gjøres likt og det vil ikke være noen form for forkjeller mellom disse avhengig av om man tar med temporale verdier eller ikke.

Valg av FE metoden
Siden FE metoden støtter interpolasjon og inndeler inn i trekanter og polygoner vil det ved romlige fenomen av irregulær form være mest hensiktsmessig å bruke FE metoden i stedet for FD metoden. Med tiden med i bildet kan FE metoden være nyttig når man skal bruke applikasjonen til for eksempel å finne fenomenets størrelse ved et bestemt tidspunkt, siden FE metoden gir interpolasjon og vil til enhver tid ha "nøyaktige" målinger mellom to ytterpunkter.

5.3.5 Forskjeller ved bruk av FD og FE metoden:

Forskjellen mellom disse to måtene vil være at man ved modell inndelt etter FD metoden må utføre beregninger på en slik måte at man tar hensyn til blokkstørrelsen. Ved bruk av FE metoden vil man kunne beregne ved å regne med ytterpunktene og punktene på den kurven mellom dem som vil bli laget ved hjelp av interpolasjon. Veldig enkelt kan man vise dette ved å vise til et eksempel:
En bil kjører en strekning på 60 kilometer på en time med en konstant fart på 60 km/t. Hvor langt har bilen kjørt etter 45 min? Modellen for denne virkeligheten kan bli diskretisert etter FD og FE metoden med samme detaljeringsnivå, se figur 27.

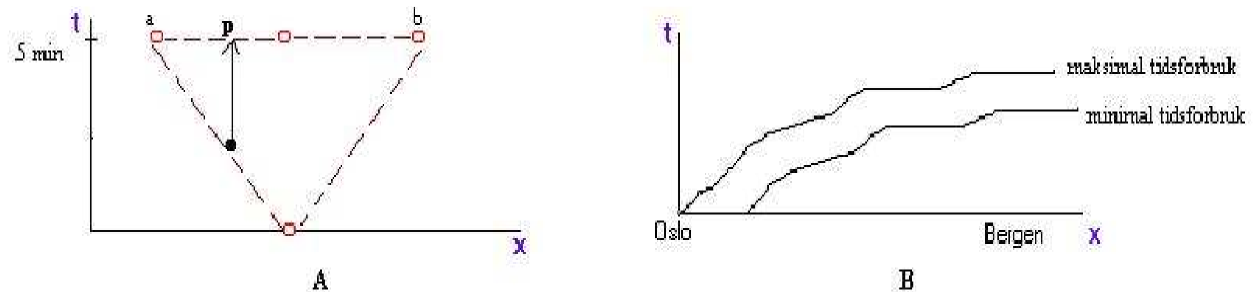


Figur 27 Utregning

Fra figur 23 ser man at ved diskretisering etter FD metoden hvor vi inndeler i blokker vil svaret på spørsmålet bli et intervall fra 40 km til 50 km, men FE metoden vil gi et eksakt svar på ca 45 km. Ett annet problem som FE metoden kan løse i motsetning til FD metoden er mangel på målinger. La oss anta at vi måler kontinuerlig i en periode men etter det blir det pause i målingene. Ved diskretisering med FD metoden vil den eller de blokken(e) dette gjelder for stå tomme, men ved hjelp av enkel metode kan man ved diskretisering etter FE metoden legge inn en regle som sier at man skal ta en interpolasjon mellom to eksisterende punkter, det vil si at estimere frem til ikke eksisterende punkter. Man kan legge inn en interpolasjonsmetode som automatisk, blant annet ved å strekke linjer mellom to målte tidspunkter. Eventuelt kan regresjon benyttes til å kunne estimere seg frem til et tidspunkt i fremtiden.

5.3.6 Intervall og FD metoden

Antar vi at for hver t verdi kun fins en x verdi, er dette et veldig enkelt syn på verden. En rutetabell laget over toget fra Oslo til Bergen kan bli lagt inn med et intervall over verdier ved forsinkelse. Som figur 28A viser har man nå to kurver med mulighet til å ha flere verdier for et gitt punkt. Dette kan også oppstå når for eksempel toget er så langt at dets lengde dekker flere x punkter. Da vil man for plassering i et gitt punkt har to tider, en tid for når lokomotivet nådde punkt x og en annen tid for når siste vogn passerer punkt x.
Et annet eksempel er et legeme (se figur 28A) som kan enten bevege seg i konstant fart forover eller bakover i det horisontale planet eller kan velge å stå stille. Legeme vil etter fem minutter befinne seg enten på samme plass eller på et intervall mellom a eller b. Hvis legeme for eksempel beveger seg bakover med den konstante farten i de første 2 minutter 30 sekundene og så stopper opp vil legeme blir tegnet inn på plass p i kurven på figur 28A.



Figur 28 Bevegelser i et intervall

Et tidspunkt har ikke lenger kun en bestemt verdi, men har muligheten til å få en verdi i et gitt intervall. Dette intervall trenger heller ikke å være likt over alle tidsperioder. Man kunne kanskje vurdere å bruke FE metoden, men da måtte man ha muligheten til å kunne velge hvilken av de forskjellige x verdiene et gitt tidspunkt kan ha. Vi kan si at togsporet består av en samling tidspunkter som hver igjen har en samling mulige x verdier. Togsporet står i aggregat forhold til tidspunktene som igjen har et mange til mange forhold med x verdiene.

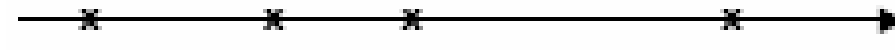
5.3.7 Forening av FD og FE metoden

La oss anta at vi har å gjøre med et fenomen som kan ha flere verdier ved enhver t verdi, det vil si være t verdi er slik at det har et intervall av x verdier. Jeg har tidligere sagt at en slik virkelighet bør inndeles et FD metoden, men hva skjer hvis det oppstår brudd i målinger? Generelt vil det oppstå hull i modellen vår, noe som ikke er tillatt etter virkelighetens regler. En løsning på dette vil være å kombinere fordelene ved FD og FE metoden slik at man kan løse dette problemet på en mest mulig hensiktsmessig løsning. I kapitel 6 "Tesselering og aggregering" kommer jeg inn på Martin Erwig og Markus Schneider [Erwig] mener at man kan modellere sik at man legger inn snapshots inn i et koordinatsystem men interpolasjon mellom objektene på de forskjellige snapshotene slik at man til enhver tid kan finne ut av hva de forskjellige verdiene er. Denne teorien kan tenkes å bruke om FD og FE metoden la oss anta at vi har blokker med verdier og det oppstår hull mellom to blokker, det vi kan gjøre da er å kjøre interpolasjon på disse to og komme fram til de ikke eksisterende målingene.
Hvis FD metoden kan få fordeler ved å forenes med Fe metoden må det være mulig å gjøre det samme med FE metoden. FE metoden kan gi feil svar hvis det eksisterer to eller flere ytterpunkter

innen for den inndelingen vi har valgt, med andre ord at ytterpunktene ligger i et intervall, her kan man velge hele blokken til å være en del av objektet og alle verdier innefor blokken vil bli en del av objektets verdier. Et eksempel på en virkelighet som kan følge denne modellen er et tog som kjører i rute men som har et forsinkelsesintervall mellom enkelte stasjoner.

6 Identifikatorer

6.1 0-dimensjonale verdier som del/hel av identifikatoren

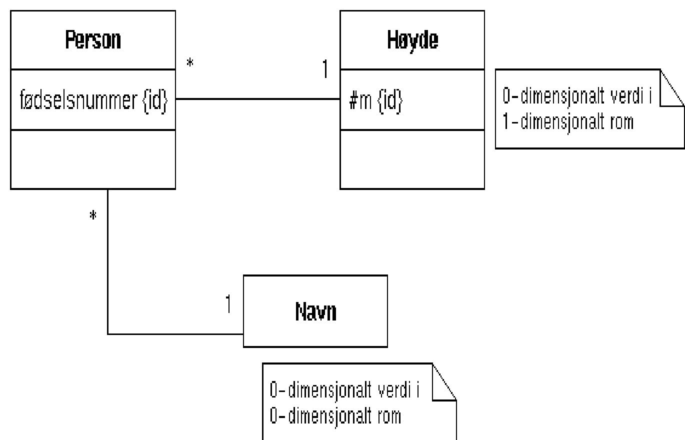


Figur 29: 0-dimensjonale begreper i 1-dimensjonalt rom

Begreper som representeres med et tall og eventuelt en enhet (mål, vekt, lengde, tid) en naturlig innebygd ordning –se figur 29. Vi kan avgjøre om et tall er større enn et annet, vi kan regne med tallene og vi kan sammenligne dem. Men tallene i seg selv har ingen utstrekning i tid eller rom. Slike begreper kan oppfattes som 0-dimensjonale begreper i et 1-dimensjonalt rom eller som punkter på en akse, og kalles gjerne for skalarer. Mål, vekt og lignende er eksempler på slike begreper. Et viktig 0-dimensjonalt begrep i et 1-dimensjonalt rom er *tidspunkt*. Slike begreper opptrer sjelden i roller som identifikatorer for en gjenstand.

Person	Høyde (#m)	Navn
01018056783	180	--
02027889076	180	-----
----		-----

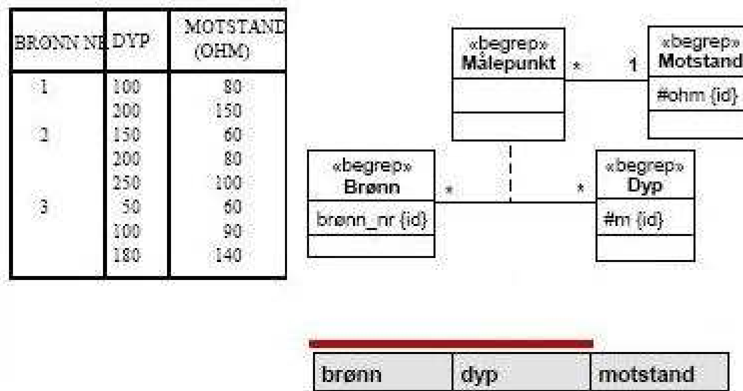
Tabell 3: Fødselsnummer som entydighetskranke.



Figur 30: Høyden som en avhengig variabel av person (virkningsvariabel)

La oss se på et eksempel. I tabellen ovenfor har vi opplysninger om alle personer ansatt i et konstruksjonsfirma. I denne tabellen er fødselsnummeret brukt for entydig identifikasjon av en person- å se tabell 1. Høyden kan ikke brukes som identifikator, fordi flere personer kan ha samme høyde og en tilfeldig høyde kan knyttes til flere personer, dette går fram av * på assosiasjonen mellom "Høyde" og "Person" i UML-klassesdigrammet - se figur 30. Fødselsnummeret er en *uavhengig variabel* (årsaksvariabel); en variable vi kan styre/kontrollere verdien av. Høyden, derimot, er en *avhengig variabel* (virkningsvariabel); en variabel hvis verdi er avhengig av de uavhengige variablene, høyden er avhengig av personen vi måler høyden på. Vi kan ifølge [Skagestein, 2002] konkludere med at avhengige variable skal aldri være gjenstand for entydighetsskranke, mens uavhengige variabler alltid skal være

gjenstand for entydighetsskranke. Dette er grunnen til at begreper med skalarer som representasjoner sjelden brukes til å identifisere noe.



BRØNN NR	DYP	MOTSTAND (OHM)
1	100	80
	200	150
2	150	60
	200	80
	250	100
3	50	60
	100	90
	180	140

Figur 31: Mål under entydighetskranke ref:Gerhards foiler (http://www.ifi.uio.no/inf102/foiler/tidogrom.pdf)

Imidlertid finnes det tilfeller hvor skalarer kan være med på å identifisere noe.
La oss se på et eksempel av 3 oljebrønner. På flere dyp i hver brønn måles elektrisk motstand, for eksempel i brønn 3 med en dybde på 100 er det 90 ohm motstand– se tabellen i figur 31. Motstanden er avhengig av brønn og dyp (koordinaten som velges i rommet for måling av motstand). Dyp er en uavhengig og virkningsvariabel for motstanden. Når skalaren ikke lenger er et tilfeldig måleresultat, men en forhåndbestemt verdi, som dybden i brønnen som velges for måling av motstand, så kan skalaren opptre under en entydighetsskranke. I dette tilfellet er dyp antall meter uavhengig variabel.
Et annet eksempel, la oss anta at hver ansatt på slutten av dagen må fylle et skjema hvor han beskriver hvilke oppgaver han har utført den dagen. I dette tilfellet må representasjon for en dag, det vi si. datoen, være med i entydighetsskranken, fordi det er forhåndsbestemt at alle ansatte må fylle skjema hver dag. Det er ikke tilfeldig hvilke dager man skal fylle ut skjemaet.
Historiedatabaser er et vanlig eksempel der representasjonen av tiden er med i entydighetskranken. Hvis for eksempel noen er interessert i å lese om norsk historie på 1800-tallet, så er tallet 1800 med på å bestemme hvilke artikler/bøker som skal listes/hentes opp av databasen.

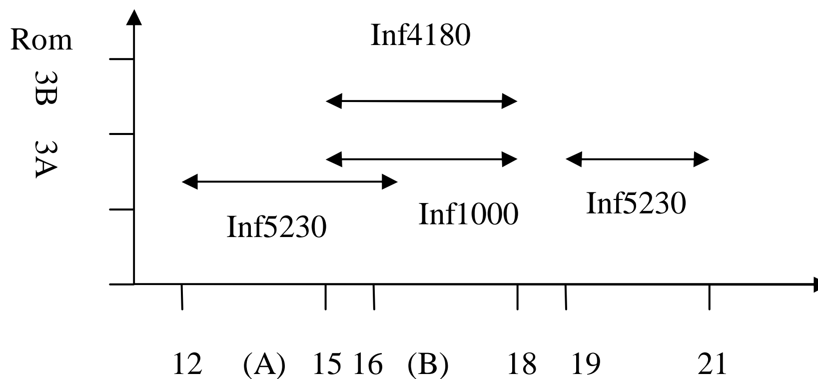
Det er altså vanlig at representasjonen av et tidspunktbegrep er gjenstand for entydighetsskranke, i hvert fall i fortid (se eksempelet med historie) og i fremtid hvis man planlegger hva man skal gjøre på et bestemt tidspunkt. Ifølge [Skagestein, 2002] vil tidspunkter med oppløsning på et sekund og bedre sjelden opptrer i roller som er gjenstand for entydighetsskranke. Slike oppløsninger brukes oftest i forbindelse med tidsmålinger, for eksempel i idretts konkurranser, og verdien vil være et tilfeldig tall. Hvis en person har løpt 60 meter på 9,8, så er det ikke noe i veien for at noen andre også kan gjøre det.

6.2 1-dimensjonale verdier som hele/del identifikatoren

Romnr	Periode	Bestilt_av
3A	2004/03/01/12.20-2004/03/01/16-00	Inf5230
3A	2004/03/01/15.00-2004/03/01/18-00	Inf1000

3B	2004/03/01/15.00-2004/03/01/18-00	Inf4180
3A	2004/03/01/19.00-2004/03/01/21-00	Inf212

Tabell 4: Romnr og periode som entydighetskranke. Periode er en 1-dimensjonal verdi i et 1-dimensjonalt rom.



Figur 32: Bestilling av rom som viser overlapping av rom 3A fra kl 15.00 til kl 16.00 den 1.mars 2004.

En periode er noe som har en utstrekning i tid, altså en 1-dimensjonal verdi i et 1-dimensjonalt rom. I tabell 3 har vi en 1-dimensjonal verdi ”Periode” under entydighetsskranken. Tabellen viser en oversikt over når og av hvem et rom er bestilt. Verdiene i ”Periode” kolonnen bryter ikke entydighetskranken, men denne entydighetskranken holder ikke for praktisk bruk av rommet. I figur 32 kan man se at rom 3A, er leid ut til to personer den 1.mars 2004 fra kl 15.00 til kl 16.00. Dette på grunn av at det er ingen regel her som sier at to tidsperioder for et rom ikke kan overlappes. Så lenge et av tidspunktene slutt eller start er forskjellige, så kan man sette inn så mange bestillinger som mulig, for et og samme rom, til og med en bestilling etter hvert minutt. Og hvis man vil ha en oversikt over alle tidsperioder et bestemt kurs har bestilt et rom, så kan man få usammenhengende 1-dimensjonale verdier, for eksempel inf5230 har bestilt et rom fra kl 12.00 til 16.00 og fra kl 19.00 til 21.00. Man må ha topologiske skranker for å kunne teste om to perioder overlapper eller ikke.
Hvis en kurs har flere bestillinger av samme rom, så kan man se på de bestillingene som flere bestillinger, istedenfor å se på det som en usammenhengende bestilling.
En 1-dimensjonal verdi i et 1-dimensjonalt rom er nødt til å være en linje, linjen kan være sammenhengende eller usammenhengende og hvert linjestykket i linjen kan representeres med to punkter, en start og en slutt, som tilsvarer grensen til linjestykket. Alt som er innenfor disse to punktene kan sees på som en tett punktmengde[4], og er linjestykkets indre. For å finne ut om to punktmengder A og B er like, kan vi undersøke om A er en delmengde av B samtidig som B også er en delmengde av A, hvis det er sant så er A og B like. Dette kan utrykkes matematisk på denne måten:

$$A \supseteq B \wedge B \subseteq A \Rightarrow A = B$$

[4] En tett punktmengde: et uendelig antall punkter, der det mellom punkter alltid kan legges inn et punkt til

Dette kan testes ved å sjekke at snittet mellom grensepunktene til A og B, og snittet mellom indre til A og B ikke gir en tom punktmengde. Mens snittet mellom grensepunkter til den ene linja og indre til den andre linja, og motsatt, gir en tom punktmengde (om to 2-dimensjonale punktmengder er like kan også sjekkes på samme måte)
To 1-dimmensjonale punktmengder er overlappende hvis:

- A sin indre snittet med B sin indre gir en ikke tom punktmengde
- A sin grense snittet med B sin indre gir en ikke tom punktmengde
- A sin ytre snittet med B sin indre gir en ikke tom punktmengde
- A sin indre snittet med B sin grense gir en ikke tom punktmengde
- A sin grense snittet med B sin grense gir en tom punktmengde
- A sin ytre snittet med B sin grense gir en ikke tom punktmengde
- A sin indre snittet med B sin ytre gir en ikke tom punktmengde
- A sin grense snittet med B sin ytre gir en ikke tom punktmengde
- A sin ytre snittet med B sin ytre gir en ikke tom punktmengde

(Teorien ovenfor gjelder også for å finne overlapp mellom to 2-dimensjonale punktmengder, men de må da i tillegg også ha en ikke tom punktmengde på snittet mellom grenser)

Denne metoden kan også brukes for usammenhengende linjer, den ensete forskjellen blir at i en sammenhengende linje består en linje av et linjestykket og vi kan finne snittet ved å teste det ene linjestykket mot det andre linjestykket i den andre linjen, men i en usammenhengende linje må man teste snittet mellom alle linjestykker til den ene linja mot alle linjestykker til den andre linja.

6.3 Konklusjon

To ting må sjekkes for at et begrep skal kunne spille en rolle som en del eller helhet av en identifikator til en gjenstand:

1. Entydighet
2. Test for overlapp

Hvis dimensjonen = 0, så degenererer 1 og 2 til samme test. Skalarer, eller et punkt som ikke har noen utstrekning i rommet og som er en årsaksvariabel kan opptre under entydighetsskranken, men hvis den har utstrekning eller er en avhengig variabel så kan den ikke være med på å identifisere noe. Da må man ha en annet identifikator som kan være med på å identifisere objektet. Man kan enkelt sjekke om to skalarer eller koordinater til to punkter overlapper hverandre, ved å sammenligne dem opp mot hverandre.
Hvis dimensjonen > 0, så er 1 og 2 to forskjellige tester. Entydigheten kan sjekkes ved å sammenligne verdiene. Imidlertiod er det ikke noen rett fram måte å sjekke om to linjer er overlappende, ikke overlappende, den ene fullstendig omsluttet av den andre, begynner de samtidig, slutter de samtidig, ligger et gitt punkt på linjen. Vi må bruke topologiske operasjoner for å finne ut av dette.

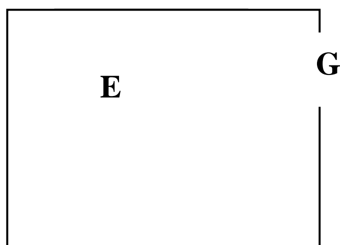
7 Realisering av romlige/topologiske skranker

For å kunne forbedre datakvaliteten må vi håndheve de skrankene vi har funnet frem til under modelleringen. Ved geografiske systemer retter disse skrankene seg blant annet mot topologiske[5] forhold som er med på å beskrive viktige romlige egenskaper.

Skranker kan realiseres og håndheves på fire måter:

1. **ved å realisere skranken i applikasjonsprogrammene:** I dette tilfellet legges ansvaret for å håndheve skranken i selve applikasjonsprogrammene – programmene må altså ved forsøk på oppdateringer selv kontrollere om skranken er tilfredsstilt etter at oppdateringen eventuelt er gjennomført. Dette er den eneste muligheten hvis applikasjonsprogrammet ikke benytter et databasehåndteringssystem som kan håndheve skranker. Løsningen er uheldig hvis mange applikasjonsprogrammer benytter samme database, fordi alle applikasjonsprogrammene da må håndheve skrankene. Et eneste program som ikke tar oppgaven alvorlig er nok til at databasen kan bli utsatt for ukorrekte oppdateringer.
2. **deklarere skranken og håndheve den med "vakthund":** I dette tilfellet deklareres skranken i databasehåndteringssystemets skjema (for eksempel med OCL) og håndheves av databasehåndteringssystemet under kjøring. Databasehåndteringssystemet har altså en innbygd "vakthund" for håndheving av skranker. Eksempler på skranker som vanligvis kan håndheves på denne måten er entydighet (primærnøkkel) og referanseintegritet (verdien av en fremmednøkkel må også eksistere som verdi i en primærnøkkel)
3. **programmere skranken i en trigger:** En trigger er en liten programbit som utføres automatisk når bestemte operasjoner forsøkes utført på databasen. Slike triggere kan blant annet brukes til å håndheve skranker. Triggerprogrammering brukes gjerne når skranken ikke kan håndheves med "vakthund" som i punkt 2.
4. **bygge skranken inn i datastrukturen:** En skranke kan ofte erstattes med en avledning. La oss for eksempel anta at vi har en skranke som uttrykker at et tall S alltid skal være summen av A og B. Istedenfor å lagre både A, B og S, kan vi lagre bare A og B og avlede S ved behov. Et tilsvarende eksempel for romlige verdier er en skranke som uttrykker at en bestemt del av grensene for to eiendommer skal være sammenfallende. Istedenfor å håndheve denne skranken kan vi representere fellesgrensen bare én gang og la denne fellesgrensen inngå i representasjonen for begge eiendomsgrensene.

La oss se på dette eksemplet med en eiendom ***E*** som har et gjerde ***G*** rundt seg



At gjerdet skal falle sammen med eiendomsgrensen kan utrykkes på tre måter:

a) skranke
   $G \equiv$ boundary (E)
b) avledning (avled gjerdet fra eiendommen)
   $G \leftarrow$ boundary (E)
c) avledning (avled eiendommen fra gjerdet)
   $E \leftarrow$ indre (G) $\cup$ boundary (G)

Figur 33: At et gjerde (G) rundt en eiendom (E) sammenfaller med eiendomsgrensen kan sikres ved skranke eller avledning

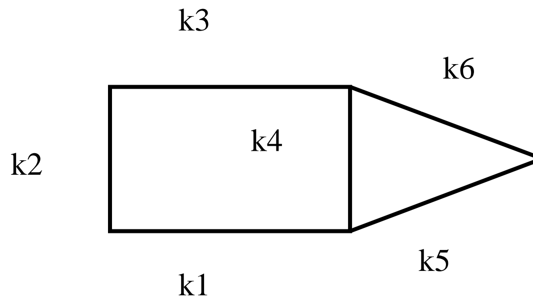
[5] Topologiske forhold blir beksrevet i kapitel "3 Fenomener i rommet"

Dette gir grunnlag for to ulike typer representasjon:

1. **Spagettimodell**: Alle objekter representeres uavhengig av hverandre. Skranker (både topologiske og andre) uttrykkes eksplisitt.
2. **Topologisk modell og nettverkmodell**: Skrankene (ikke nødvendigvis alle) bygges inn i representasjonen.

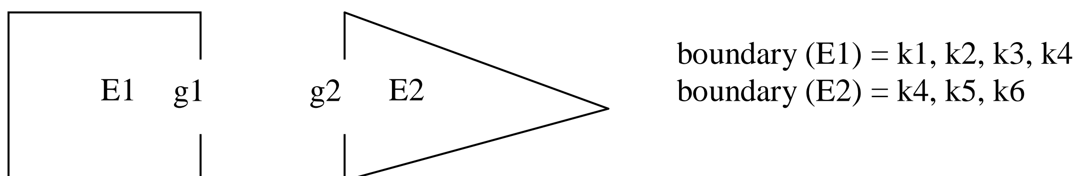
7.1 Skranker i spagettimodellen

I en spagettimodell er basis-abstraksjoner for å representere romlige verdier av et objekt i et 2-D rom punkt, linje og polygon. Et punkt representerer den romlig egenskapen til et objekt der hvor bare objektets plassering i rommet er relevant (ikke utstrekningen), for eksempel plassering av en by. En linje (kan bestå av mindre linjestykker) representerer den romlige egenskapen til et objekt hvor man er interessert i utstrekningen til objektet, for eksempel veier og elver. En polygon kan enten representere en 1 dimensjonal verdi i et 2D rom, for eksempel den romlige egenskapen til et gjerde (rundt en eiendom, som representeres med en lukket linje), eller en 2 dimensjonal verdi i et 2D rom, som for eksempel representerer romlige egenskapen til en eiendom (avgrenset av en gjerde).



Figur 34: To eiendommer med en felles grense 'k4'

I en spagettimodell er det en mengde av enkeltvise objekter og alle objekter representeres uavhengig av andre objekter, og skranken realiseres i applikasjonen eller databasehåndteringssystemet. Ingen topologi blir lagret i en slik modell, og alle topologiske forhold må uttrykkes i skranker og beregnes under spørringer. Denne strukturen impliserer også representasjonsredundans. For eksempel er grensen mellom to tilstøtende regioner, se figur 34, representert to ganger, se figur 35.



Figur 35: Grensen mellom to tilstøtende eiendommer er representert to ganger i spagettimodellen

Skranken for fellesgrensen kan uttrykkes slik

$$\mathbf{S1 = E1.k4 \equiv E2.k4}$$

og må håndheves av applikasjonsprogrammene eller databasehåndteringssystemet.
Denne enkle modellen muliggjør en heterogen representasjon som kan blande punkter, linjer og polygoner uten noen begrensninger. De romlige verdiene til objektene kan krysse hverandre i planet, men verdiene for eventuelle krysningspunkter finnes ikke i representasjonen.

Den største fordelen med spagettimodellen er dens enkelhet. Siden alle objekters romlige verdier er lagret uavhengig av andre objekters romlige verdier, tilbyr denne modellen sluttbrukeren en enkel input av romlige verdier for nye objekter. Ulempen med spagettimodellen er mangel på eksplisitt informasjon om topologiske forhold mellom de romlige verdiene til objektene, slik som tilstøtning og inkludering. Det er for eksempel ingen enkel måte å finne ut om grenselinjene til to polygoner deler et punkt.

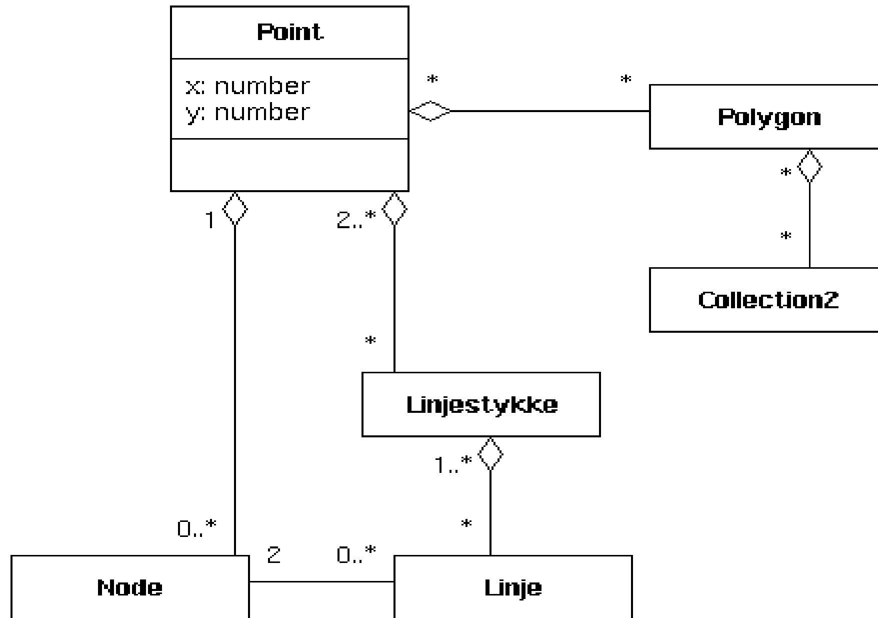
7.2 Skranker i den topologisk modellen

Noen mulige abstraksjoner for å representere de romlige egenskaper for romrelaterte samlinger av romlige objekter er partisjoner av rommet, nettverk og topologisk.
Ifølge [Emmanuell] kan en partisjon vises som en mengde av polygonobjekter som må være disjointe. Par av polygoner som har en felles grenselinje kan representeres med denne modellen, fordi denne representasjonsmodellen er spesielt egnet for tilstøtningsforhold.
I dette avsnittet skal vi se hvordan man kan realisere skranker i topologiske modeller ved å strukturere og lagre ting slik de er plassert i forhold til hverandre. Når for eksempel to eller flere linjer krysser på et punkt, blir dette punktet lagret som en node, og alle linjene som krysser på dette punktet lagres sammen med denne noden. Vi skal vise hvordan man realisere skranken i databasestrukturen i topologiske modeller både i 1D, 2D og 3D.

7.2.1 Realisering av 1D Topologisk modell / Nettverk modell

Ifølge P. Rigaux, M. Scholl og A. Voisard [Rigaux 2002] var den romlige *nettverk-modellen* opprinnelig designet for å representere nettverket i nettverk (graf)baserte applikasjoner, slik som transporttjenester eller håndtering av elektrisitet, telefon og lignende I denne modellen er topologiske forhold mellom punkter og linjer (kan bestå av flere rette linjestykker) lagret. To viktige begreper i denne modellen er *noder* og *kanter*. En node er et spesielt punkt som forbinder et antall kanter. En kant er en linje som starter i en node og slutter i en node. Siden noder forbinder linjer, er det mulig å navigere gjennom nettverket: Når man støter på en node kan man velge en ny kant å følge. Noder tillater effektive linjeforbindelse-tester og nettverk-beregninger (korteste vei).

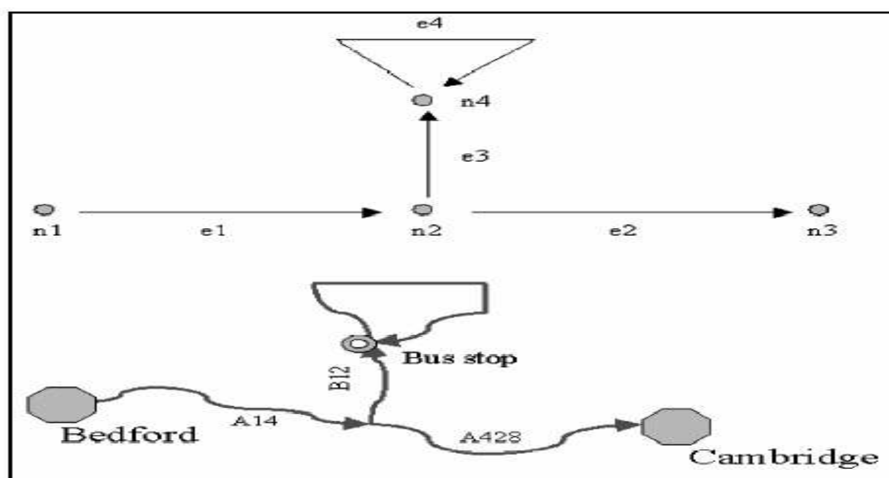
Her ser vi de forholdene mellom geometrielementer som trengs for å modellere/representere en nettverkmodell (begrenset til 2D):



Figur 36 Representasjonsmodell for en nettverk-abstraksjon

En node er enten et linjes sluttpunkt eller et isolert punkt. Linje- og polygonhjørner er regulære punkter. Avhengig av implementasjonen er nettverket enten planart eller ikke-planart. I et planart nettverk blir hvert kantkryss lagret som en node, selv om noden ikke korresponderer med et romlig objekt som tilsvarer en håndgripelig entitet fra den virkelige verden. I slike tilfeller er en planar nettverkmodell bare en representasjonsmodell og ikke modellering av virkeligheten. I et ikke-planart nettverk kan kanter krysse uten at det blir sett på som et kryss. Eksempler på ikke planare nettverk er undergrunnsbaner med tunneler. Dermed kan en ikke-planar modell være både en modell av virkeligheten og en representasjonsmodell. Et eksempel av [Watson 2002] viser et enkelt veinettverk sammen med den korresponderende topologiske modellen – se figur 37. Hvert objekt på kartet (by, vei) har en serie av attributter, inkludert geometri. Objektgeometrien kan altså bli sett på som sammensatt av primitive elementer (noder, kanter) i en topologisk struktur.

Med topologien tilgjengelig er det mulig å svare på spørsmål som ”Hvilken vei går gjennom Bedford og Cambridge?” uten å sammenligne byers og veiers geometrier på hver enkelt forbindelse. Ruten er enkelt sammensatt av veier ved bruk av kanter mellom nodene n1 og n3. Det er enkelt å trekke ut topologiske forhold når databasestrukturen er lagret på denne måten. Ellers ville det vært kostbart fordi man måtte gjøre multiple direkte sammenligninger mellom de romlige verdiene til objektene.



Figur 37: Et enkelt veinettverk med den korresponderende topologiske modellen

7.2.2 Realisering av 2D Topologisk modell

Topologiske modeller tilbyr mekanismer for å beskrive delte geometrier. Dette kan gjøres ved å tillate topologiske primitiver å bære geometrier. Der hvor to lineære romlige verdier av objekter passerer samme ruten mellom et par av noder, deler de det samme kantelementet. Ved å lagre det delte segmentet av geometrien i direkte tilknytning til kanten, kan for eksempel både en vei og en busslinje ha en referanse til den delte kanten.

Ifølge [Rigaux 2002] er den topologiske modellen identisk med nettverkmodellen, med det unntaket at nettverkmodellen er planar. Objektene i denne modellen er:

- Punkt: [x:real, y:real]
- Node : [punkt, (liste arc)]
- Arc : [node-start, node-end, venstre-poly, høyre-poly, (liste punkter)]
- Polygon : (liste arc)
- Region : {polygon}

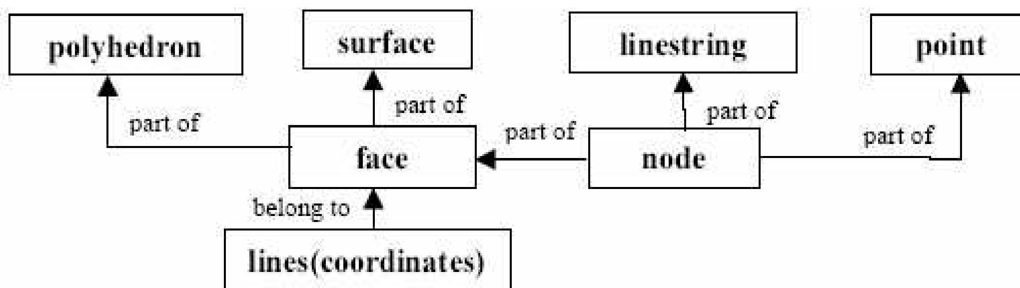
En node er representert med et punkt og en liste av kanter som enten starter eller slutter i denne noden. Hvis listen er tom, korresponderer noden til et punkt isolert fra nettverket. Isolerte punkter blir brukt for å identifisere plassering av punktobjekter eller områdeobjekter kollapset til et punkt slik, et tårn eller en skole.

I tillegg til endepunkter og en liste av hjørne punkter har en kant også dens høyre og venstre polygon representert, altså de polygonene som har denne kanten som en felles grense.
Det kan av effektivitetsgrunner eksistere noen redundante data for aksessering av romlige verdier. For eksempel kan polygoner aksessers enten gjennom polygoner eller kanter. Imidlertid er det ingen redundans i de lagrede geometriene, siden punkter og linjer er lagret kun en gang.

7.2.3 Realisering av 3D topologisk modell

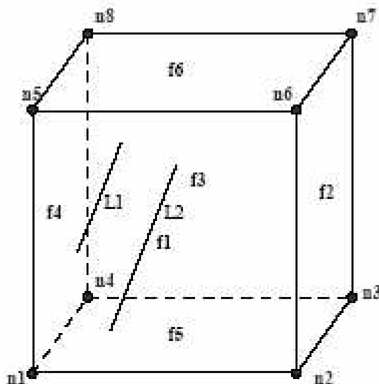
[Ziatanovas 2001] anbefaler en 3D modell som er en typisk implisitt grenselinjemodell. Hvert n-dimensjonalt objekt er assosiert med fire abstraksjoner *punkt, linje*, *overflate* og *polyhedron.* En polyhedron har en plassering og volum. Disse geometriske objektene **(GO)** er bygget opp av mer primitive konstruksjonselementer **(CnsO)**. Modellen består av to **(CnsO)** *node* og *fasett.* Node beskriver romlige objekter som kan representeres som linjer og punkter. Noder er også konstruksjonselementer for Fasetter. Rekkefølgen av nodene i en fasett er kjent. Fasetter blir brukt til å konstruere objekter som er assosiert med overflater og polyhedroner.
Figur 38 viser et skjema av modellen. Modellen viser mange til en forhold i pilens retning. For eksempel en fasett er en "del av" (part of) polyhedron og polyhedronen "består av"mange fasetter.

Alle linjeobjektene blir representert som en separat datamengde. Hver linje er betraktet som en rett linje representert med to mengder av koordinater.



Figur 38: 3D Topologisk modell [Ziatanovas 2001]

1D celle, ofte kalt for kant(arc) er utelatt i denne 3D modellen. En kant i 2D-rommet har en egen representasjon fordi en kant alltid har to noder og to polygoner (høyre side, venstre side). En kant har alltid to noder i 3D også, men her er det ingen regel for hvor mange polygoner en kant kan ha. For eksempel har kanten mellom n1 og n2 bare en polygon f5. Derfor vil en eksplisitt lagring av kanter ikke gjøre modellen noe enklere.



Her er et eksempel på en kube som består av seks fasetter (f1,…,f6), åtte noder (n1,…,n8) og to linjer (L1 og L2) på fasett f1.

Figur 39: Et eksempel på 3D-objekt

7.3 Oppsummering

Å håndheve skranken i applikasjonsprogrammene er ingen smart måte å realisere skranken på, hvis vi har et databasehåndteringssystem som kan håndheve skranker. Spesielt må vi sikre oss at alle applikasjonsprogrammer sjekker om skrankene er tilfredstilt. Dette fører til at man får

tyngre applikasjonsprogrammer, og siden flere applikasjonsprogrammer håndhever skranken på en database, så kan dette fort føre til betydelige feil, hvis et av applikasjonsprogrammene feiler på håndheving av skranken.

Den beste og mest fornuftige måten å håndheve skranken på er med ”vakthund”, fordi databasehåndteringssystemet håndhever skranker selv under kjøring, og applikasjonsprogrammene slipper å bekymre seg om håndheving av skranker. Vi får en global håndheving av skranker for alle applikasjoner mot databasen. Imidlertid kan vi av og til bli tvunget til å bruke triggere.
.
Ved å bygge skranken inn i datastrukturen får vi redusert redundans i data, fordi deler av verdiobjekter som er felles for flere objekter lagres kun en gang, og kan inngå i representasjon av alle objekter som bruker den. Vi får også en meget effektiv håndtering av skranken, fordi den er ivaretatt gjennom selve datastrukturen. En ulempe er at dersom skarkene endrer seg, kan dette bety en endring i selve datastrukturen.

8 Operasjoner og abstrakte datatyper

I dette kapitlet skal vi holde oss til 1-dimensjonale begreper, som omfatter kurver og linjer. Ifølge Gerhard Skagestein [Skagestein 2002] linjer og kurver er det enkle eksemplet på en datatype med både "utstrekning" og beliggenhet. En linje kan sees på som en mengde av et uendelig antall punkter, og hvert punkt blir representert med koordinater. Matematikere sier at denne mengden er tett fordi det mellom to punkter er alltid mulig å legge inn et punkt til. Vi må kunne regne med linjer, vi må kunne skjøte sammen linjer, trekke fra linjer, dele linjer, forkorte og forlenge linjer og flytte linjer.
Linjer må kunne inngå i tester, bl.a. med tanke på skrankehåndtering: Er to linjer overlappende, ikke overlappende, den ene fullstendig omsluttet av den andre, den ene før den andre, begynner de samtidig, slutter de samtidig, ligger et gitt punkt på linjen?
I det n-dimensjonale rom, n>1 kan vi ha både rette linjer og kurver i ulike fasonger. En lukket kurve har ingen start og endepunkter, og den krysser ikke seg selv.
Linjer og kurver representeres ved hjelp av utvalgte punkter pluss beregningsregler for de øvrige punktene på kurven

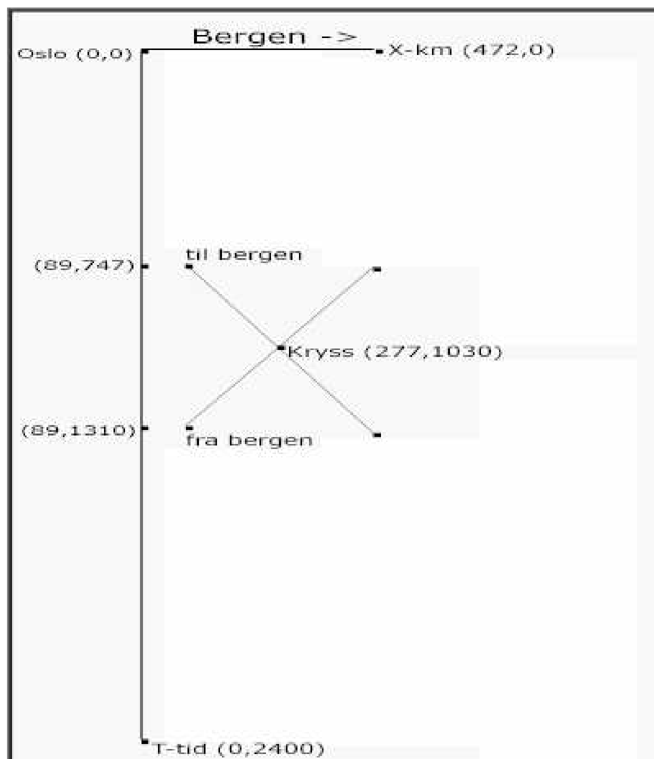
Databasesystemer er designet for å håndtere store mengder av data og tilbyr mange viktige tilleggs funksjoner som objektorienterte programmeringsspråk ikke gjør, eksempelvis;

- raske spørringer
- deling av objekter mellom programmer
- permanent lagring

I dette kapitlet har vi programmert et eksempel rundt togfremføringen på Bergensbanen. Et tog er et objekt, og fremføringen av dette toget kan betraktes som et 1-dimensjonalt begrep (en kurve) i et 2-dimensjonalt rom (x og t). Kurven viser sammenhengen mellom avstanden x fra et fast punkt på strekningen Oslo-Bergen og det tidspunktet t toget i henhold til ruteplanen skal befinne seg på dette stedet. Vi skal finne maks et krysspunkt mellom to tog. Vi har programmert løsningen i flere kommersielle databasehåndteringssystemer for å få en oversikt over de ulike mulighetene for representasjon og håndtering av romlige begreper.

Vi antar at toget går med konstant hastighet slik at vi får en lineær funksjon t=ax+b. Siden vi langs t-aksen regner med 100 dels timer istedenfor minutter, kan vi få svar for krysningspunktet som avviker fra den virkelige posisjonen for krysningspunktet. Avstanden blir regnet fra Oslo, det vil si avstanden er 0.0 km når et tog er på Oslo stasjon. Her ser vi to tog den ene som går fra Bergen (tog nr 62) kl 07:58 og da er den på avstand 471.25 km på vei til Hønefoss, og den andre som går fra Hønefoss (tognr 609) kl 07:47 på avstand 89.57 km på vei til Bergen.

(På grunn av utilgjengelige data fra Oslo til Hønefoss er denne strekningen blitt utelatt.)



Figur 40: To tog på Bergensbanen (Bergen-Hønefoss, Hønefoss-Bergen)

8.1 Operasjoner mot data i relasjonelle databasehåndteringssystemer

Relasjonell DBMS kjennetegnet ved:

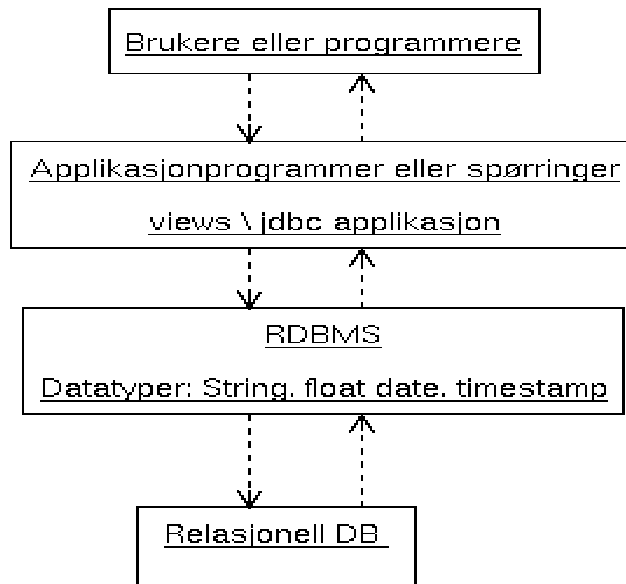
- flate tupler
- enkle data typer (char, integer, date, etc)
- bestemt mengde av data typer

En RDBMS lagrer data i en database som kan bestå av en flere tabeller av rader og kolonner. Radene korresponderer til en post (tuppel), kolonnene korresponderer til attributter (felter i en rad). Hver kolonne har en data type. Typer av data som kan bli lagret er begrenset til et bestemt antall av data typer, for eksempel character, stirng,, date også videre. En hvilken som helst attributt (felt) av en rad kan bare lagre en enkel verdi. Felter med variabel lengde er ikke støttet.

RDBMS bruker SQL for data definisjon , data håndtering, og data aksess og henting. Data hentes basert på verdien i en bestemt felt i en rad. Typer av spørringer som er støttet er spørringer fra enkle enkel-tabell spørringer til kompliserte fler-tabell spørringer, som involverer join, nøsting, mengde union/differanse, og andre.

Både romlige og ikke romlige verdier er representert på en tabellarisk form i en ren relasjonell modell. Operatorer som trengs for å manipulere romlige enheter er inneholdt i applikasjonslaget som er bygget på toppen av DBMS. Rollen til applikasjonslaget er å supplere mengden av funksjoner som er tilbudt av de underliggende systemarkitekturer, slik at funksjonell behov for romlig data håndtering blir tilfredstilt. Med andre ord, applikasjonslaget må tilby de GIS operasjoner som ikke er tilgjengelige i den underliggende DBMS'en.

Vi har laget operasjoner i applikasjonslaget (spørringer med views og jdbc applikasjon). Eksekveringen av operasjoner i applikasjonslaget i en RDBMS er ofte forbundet med opp/nedlasting av data fra/til databasen. Dette er kostbart og bør derfor unngås.

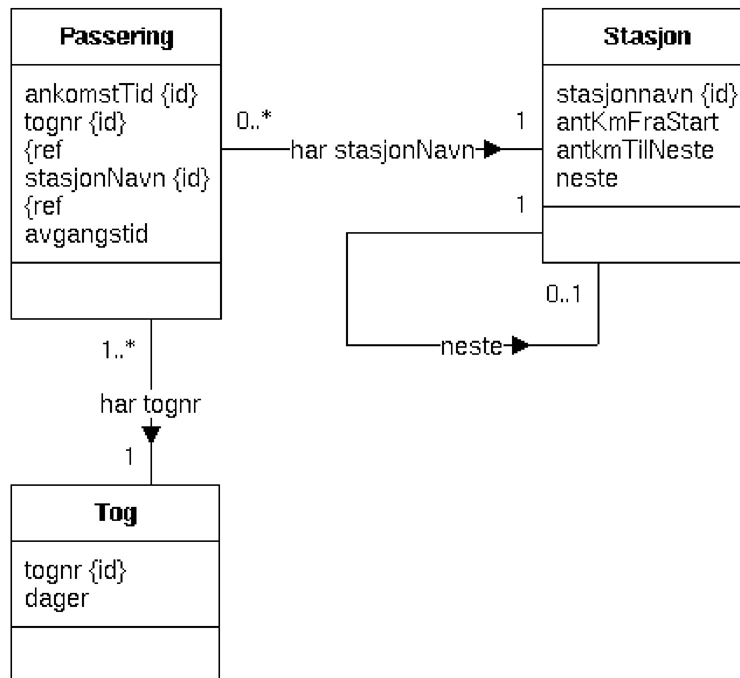


Figur 41 Klient tjener arkitektur for en relasjonell database for romlige objekter

Relasjonell database i et SQL92 miljø har et matematisk grunnlag, har enkel struktur av tabeller og er mest egnet for enkle strukturerte data. Tabeller blir joinet sammen ved å bruke felles rad/kolonne verdier, altså nøkler.

I den geometriske modellen er et romlig objekt en rad i en relasjon, og hver kolonne i relasjonen er attributtet til objektet. En kolonne med romlig verdi i objektet er implementert som en fremmed nøkkel til en geometritabell. Geometriverdien er lagret ved å bruke en eller flere rader i geometritabellen, som igjen kan ha fremmednøkler til andre tabeller som inneholder primitive geometrityper som punkt, linje eller polygon. Geometritabellen kan implementeres enten ved å bruke standard SQL numeriske typer eller binære typer.

Det første steget for å definere kravene til data i databasen er å designe en modell av den virkelige verden med fokus på de fenomener som vil være relevante for den aktuelle applikasjonen. Både de romlige og ikke romlige verdiene lagres felles i databasen Nedenfor er vist en datamodell som viser de viktigste fenomener i interesseområdet.



Figur 42: Datamodellen over toglinjer og passeringer.

Den første løsningen er basert på en enkel relasjonsdatabase i ORACLE9i med SQLPlus. Figuren viser forholdet mellom de tre tabellene i databasen, med tilhørende attributter til hver tabell:

- Stasjon
- Tog
- Passering

Et tog har mange passeringer, men en bestemt passering kan bare tilhøre et bestemt tog, ankomsttid, avgangstid på en bestemt stasjon. Og en passering kan ha bare en stasjon.

Det må finnes et startpunkt for hver toglinje, det kan være en av endestasjonene til toglinjen, startpunktet har da "antKmFraStart" lik 0.0 km. Attributtet "neste" er navn på neste stasjon og "antKmTilNeste" er antall km fra denne stasjonen til neste stasjon.

Databasen er initialisert med data om et par tog fra Bergensbanen, data om passeringene til disse togene på alle stasjonene samt data om alle stasjoner fra Oslo til Bergen.

De to togene som er blitt brukt gjennom hele eksemplet for å finne krysspunktet er tog nr 609 som går fra Oslo til Bergen og tog nr 62 som går fra Bergen til Oslo.
Geometrien ligger i denne modellen skjult dels i tabellen *Passering* (avgangstid), dels i tabellen *Stasjon* (antKmFraStart). For å kunne hente ut antKmFraStart er det en fremmednøkkel fra tabellen *Passering* til tabellen *Stasjon*.

8.1.1 Operasjonen med SQL

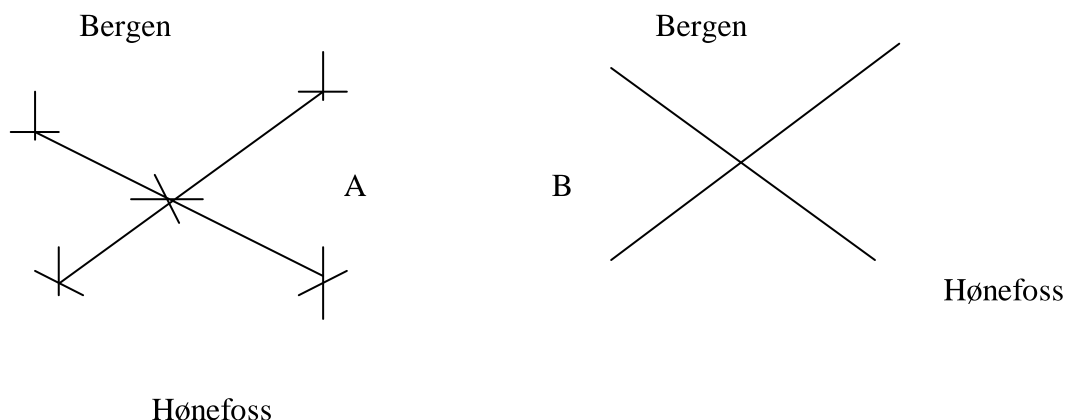
Hvis vi nå skal finne om det er krysspunkt mellom noen av togene som vi har lagt inn data om, kan vi forsøke å gjøre med en enkel SQL spørring som dette mot den relasjonelle databasen:

```
select p1.stasjonnavn,p1.ankomsttid,p1.tognr,p2.tognr
from passering p1, passering p2
where p1.ankomstTid=p2.ankomstTid and p1.stasjonnavn=p2.stasjonnavn
and not p1.tognr=p2.tognr;
```

Resultat:

```
no rows selected
```

Selv om resultatet viser ingen treff betyr nødvendigvis ikke det at det ikke finnes noe krysningspunkt mellom noen av togene. Grunnen er at denne spørringen bare vil gi treff hvis krysningspunktet er lagret som et punkt hos de togene som krysser på det punktet. For at en slik spørring som ovenfor skal kunne gi et korrekt svar må vi representere hver enkel toglinje som en tabell, men da må det lagres uendelig mange rader i hver tabell, og man kan ikke lagre uendelig mange rader i en endelig minne i maskinen.



Figur 43: tegning(A) viser toglinjer representert ved hjelp av antall punkter (her 4) med vektorrepresentasjon, og tegning (B) viser toglinjer representert ved hjelp av halvplan representasjon (uendelig antall punkter)

Hvis vi bruker en nettverkmodell representasjon istedenfor en ren vektorrepresentasjon kan vi få løst dette problemet. I en nettverkmodell er en node enten et linjes sluttpunkt eller et isolert punkt. I et planart nettverk blir hvert kantkryss lagret som en node, selv om noden ikke korresponderer med et romlig objekt som tilsvarer en håndgripelig entitet fra den virkelige verden. Dermed hvis vi bruker en planar nettverkmodell kan vi også representere krysspunktet mellom to tog eksplisitt i databasen. I slike tilfeller er en planar nettverkmodell bare en representasjonsmodell og ikke modellering av virkeligheten.
Man kan også bruke raster- eller halvplanrepresentasjon istedenfor vektorrepresentasjon for å løse dette problemet.

For å finne krysningspunktet mellom to tog i en relasjonell database med vektorrepresentasjon bruker vi formelen t=ax+b som beskriver funksjonene for begge linjene og setter funksjonene opp mot hverandre for å finne det eventuelle krysspunktet. Vi antar at linjene ikke kan være parallelle.
I den matematiske bruken av formelen t=ax+b kan vi egentlig bruke hvilke som helst to punkter på linjen. For å få en best mulig tilnærming velger vi imidlertid å bruke punktene som representerer togpasseringer i henholdsvis Bergen til Hønefoss.

Den første løsningen er programmert med SQL og står skissert nedenfor.
De første fire views inneholder start- og sluttpunkter for begge togene. For eksempel viewet "B_Start" inneholder x og t verdier for toget som skal til Bergen og som starter fra 'Hønefoss', og et identisk view for hvor toget stopper, altså 'Bergen'. Og start fra "Bergen" og sluttpunkt i "Hønefoss" for det andre toget..

```
create or replace view B_Start
as
select antkmfrastart x1, ankomsttid t1
from passering p, stasjon s
where tognr=609 and upper(p.stasjonnavn)=upper('hønefoss')
and s.stasjonnavn=p.stasjonnavn;
```

Viewet "***a_b*** "tar vare på beregningene av konstantene a1 og b1 for toget fra Oslo til Bergen, og a2 og b2 for toget fra Bergen til Oslo.
Neste view ***"t_0"*** tar vare på tidspunktet hvor begge togene krysser i variabelen t0 ved å gjøre beregninger på konstantene i viewet "***a_b*** ".
Det siste viewet gjør beregninger på variablene i viewet "***a_b***"og variabelen t0 og tar vare på x0, som er antKmfrastart hvor begge tog krysser, i viewet ***"x_0".***

Spørringen:

```
select 'krysspunkt',x0,t0
from x0,t0
where x0>0 and x0<475 and t0<0 and t0<2400;
```

gir resultatet:

```
        X0              T0
----------      ----------
278.231844      1037.15013
```

Dette resultatet stemmer med den virkelige posisjonen for krysspunkt mellom disse togene (litt avvik på grunn av forutsetningene ovenfor).

Modellen ovenfor er en passiv geometrimodell. Den baserer seg på modellering av toglinjen med punkter (toglinjens start- og sluttpunkter), og operasjoner på geometrien eller topologiske forhold mellom to geometrier kodes som separate prosedyrer i applikasjonsprogrammet. I eksemplet her har vi funnet snittet mellom to toglinjer. Ansvaret for beregninger ligger ikke i modellen, men i applikasjonen.

8.1.2 Operasjonen med Java database connection

Jdbc (Java database connection) er designet for å tillate programmerere å bruke SQL for å gjøre spørringer mot et bredt spekter av databaser. Jdbc er ment å fri programmererne fra

bekymringer om spesielle databasehåndteringssystemer. Den tillater eksempelvis programmereren å gjøre spørringer mot en Access-database med nøyaktig den samme kildekoden som mot en Oracle database.

Denne applikasjonen er laget i en klasse kalt RelasjonellKall, og metodene for å finne krysspunktet er:

- **void finnKryss**(): Spør bruker om tognumrene og kaller på metoden finnTogene med tognumre som parametere til metoden.
- **void finnTogene(***tognr1,tognr2***):**
  - initialiserer to arrayer med x og t verdier fra databasen for toget fra Bergen til Oslo
  - initialiserer to arrayer med x og t verdier fra databasen for toget fra Oslo til Bergen
  - kaller på metoden finnkrysspunktet med første og siste punkt for det ene toget og første og siste punkt for det andre toget som parametere til metoden.
- **void finnKrysspunktet(***t1,x1,t2,x2,t3,x3,t4,x4***)**: Metoden bruker en algoritme for å finne funksjoner for begge rette linjer og finner deretter krysningspunktet ved å sette begge funksjonene lik hverandre. Til slutt blir det utført en test for å finne ut om den t og x verdien som er krysningspunktet for linjene også virkelig er et krysningspunkt for togene: Det blir testet om x verdien ligger innenfor strekningen Oslo og Hønefoss og om t verdien befinner seg mellom kl 0000 og 2400.

Resultat:

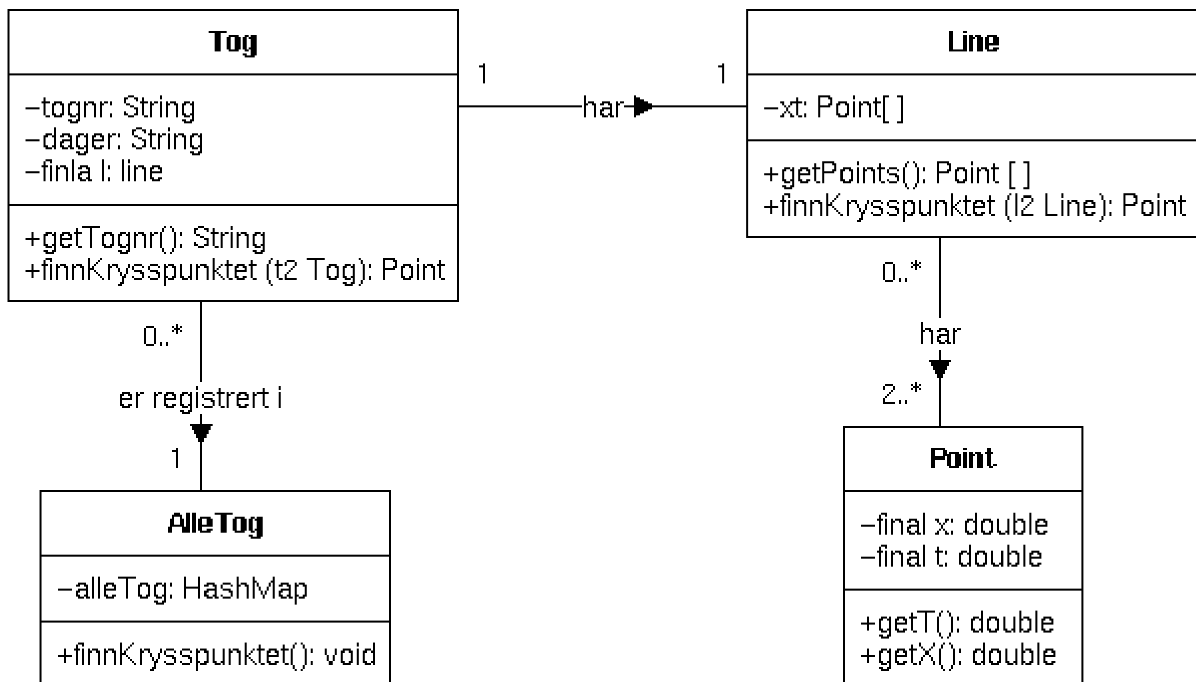
```
Oppgi tognr til toget fra Oslo s til Bergen:
609
Oppgi tognr til toget fra Bergen til Oslo s:
62

Krysspunktet for tognr 609 og tognr 62
tid:    1037.1501316944687
sted:    278.2318437225636 km fra Oslo s
```

Jdbc oppretter en brukergrensesnitt på programmerings nivå for kommunikasjon med databaser på samme måte som Microsoft's "Open Database Connectivity" (ODBC) komponent - se [Rawn Shah 1996].
Ifølge [Kurt Baumgarten 2001] Jdbc er designet for å bruke SQL til å gjøre spørringer mot en mengde av forskjellige database typer, og er ment til å gjøre programmereren fri fra lavere grads bekymringer om spesielle database typer. Den bør tillate programmereren, for eksempel, til å gjøre spørringer mot en SQLServer database med nøyaktig den samme kildekoden som med en Oracle database. Det finnes tilfeller hvor man ikke kan bruke samme kildekode mot to forskjellige database typer, for eksempel, både SQLServer og Oracle tillater å lage prosedyrer i databasen, og en prosedyre i Oracle kan returnere en cursor til en Jdbc applikasjon, mens en prosedyre i SQLServer kan kun returnere en cursor lokalt i databasen, men ikke til en ekstern applikasjon som Jdbc.

8.2 *Operasjoner i objektorienterte systemer*

Objektorientert modellering og -design er en ny måte å tenke på omkring problemer som benytter modeller organisert rundt fenomener og begreper i den virkelige verden. Objektorientert modellering og objektorientert teknologi har vist seg å være et meget fleksibelt verktøy.
Objektorienterte begreper og konsepter kan anvendes for datamodellering så vel som innen system- og programutvikling. En viktig egenskap ved objektorientert modellering er at den kan brukes uavhengig av hva som skal modelleres og utvikles. For geografisk informasjonsteknologi (GIT) og utvikling av geografiske informasjonssystemer (GIS) har objektorientert teknologi særlig betydning for datamodellering og databaseutvikling samt for system og programutvikling. Utvikling av en romlig database baserer seg på modellering av den virkelige verden ved å lagre egenskaps verdier til de fenomener som man er interessert i å beskrive.

Generelt betyr begrepet objektorientert at en organiserer programvare som en samling diskrete objekter som inkorporerer både datastrukturer og oppførsel. Dette i kontrast til konvensjonell programmering der datastruktur og operasjoner er bare løselig forbundet, slik som for eksempel i SQL spørringer mot relasjonelle databasesystemer.



Figur 44: Objektorientert modell for å finne krysspunkt mellom to toglinjer

Klassen AlleTog inneholder to lister: En liste over alle tog og en liste over alle stasjoner som finnes på den strekningen toget er ment å kjøre. Tog har et unikt nr, det er bestemte dager et tog kjører og den har en bestemt "line" som egentlig er togfremføringen. Line inneholder mange punkter hvor hvert punkt har en x- og en t-verdi. x forteller hvor langt toget er fra et bestemt startpunkt (altså "antKmfrastart") og t viser tidspunktet for når toget skal være på akkurat det stedet. Variabelen "antKmfrastart" i klassen stasjon brukes for å konvertere km til stasjonsnavn i metoden finnRuten for å gi brukeren en rute oversikt for toget.

Metode	Beskrivelse
AlleTog.finnKrysspunktet	Spør brukeren om tognr for de to togene brukeren vil se krysningspunkt for, og kaller på metoden ***tog1.finnKrysspunktet*** med det andre tog-objektet som parameter til metoden. Metoden returnerer et punkt.
Tog.finnKrysspunktet	Kaller på metoden ***line1.finnKrysspunktet***, med et annet linje-objekt som parameter til metoden. Metoden returnerer et punkt til ***AlleTog.finnKrysspunktet.***
Line.finnkrysspunktet	Inneholder algoritmen for å finne krysningspunkt mellom to rette linjer. Metoden returnerer et punkt.

Ved kompilering og kjøring av dette programmet får jeg dette resultatet.

```
*******************************************
Tog til Oslo s: 62 -602
-------------------------------------------
Tog til Bergen: 609 -61
*******************************************


***************Togoversikt*************
1. Finn Krysspunktet
2. Avslutt


Oppgi tognr til toget fra Oslo s til Bergen:
609
Oppgi tognr til toget fra Bergen til Oslo s:
62

Krysspunktet for tognr 609 og tognr 62
tid:    1037.1501316944687
sted:    278.2318437225636 km fra Oslo s
```

Dette resultatet viser at når brukeren vil finne krysningspunktet mellom to tog, oppgir han tognummeret til togene (i dette tilfellet tog nr 609 og 62) og får vite hvor og når togene krysser. Resultatet vi får her stemmer med forrige resultatet.

8.3 Objektrelasjonelle databasesystemer

I en objektrelasjonell databasesystem er all persistent informasjon fortsatt i tabeller, men noen av de tabellariske forekomstene kan ha en rikere datatyper, som blir kalt for abstrakte datatyper (ADTs).
Objektrelasjonelle databasesystemer konverterer innkommende data til databasen til databasetabeller med rader og kolonner og håndterer data på samme måte som i en relasjonell database, og applikasjoner håndterer informasjon i objektrelasjonelle databaser på en objektorientert måte. På samme måte må systemet når data blir hentet ut av databasen, samle data fra enkle tabeller til komplekse objekter før data kan returneres til applikasjonen.

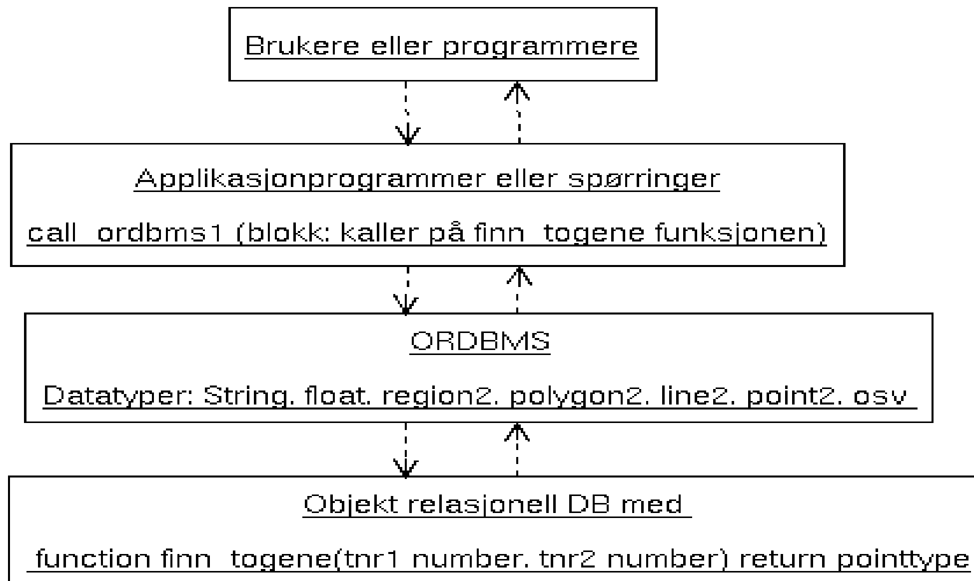
8.3.1 Utvide datamodellen med romlige ADT'er

En ADT representerer et abstrakt funksjonelt syn på objekter, og er konstruert ved kombinasjon av basis datatyper. En mengde av operasjoner er definert på objekter av en bestemt type. Ideen bak dette er å skjule den interne strukturen av datatypene for programmereren, som bare kan aksessere ADT'en gjennom operasjoner definert på det. Man lager brukerdefinerte funksjoner som blir utført via DBMS. Fra utsiden er brukergrensesnittet mot en ADT en liste av operasjoner. Et slikt skille mellom bruk og implementasjon kalles innkapsling (*encapsulation)*. Prinsippet setter oss i stand til å utvide DBMS med geometrisk funksjonalitet uavhengig av en spesifikk representasjon/implementasjon. Det blir mulig å definere en liste av romlige datatyper som tilbyr et API til programmereren.
Med en romlig ADT synsvinkel vil et romlig objekt, for eksempel en region, bli betraktet som en liste av operasjoner som man kan utføre på det, uavhengig av dets interne representasjon, for eksempel testing av snittet med andre objekter, beregning av snittet og beregning av områdestørrelse.
Det er en skille mellom geografiske datamodeller og romlige datamodeller:

1) Den romlige datamodellen utgjør det første laget på bunnen. Den tilbyr datastrukturer for å kunne representere datatypen av kolonnen med romlige verdien i ADT'en samtidig som den tilbyr operasjoner på disse datastrukturene.
2) Mekanismene er skjult i det øvre laget som håndterer geografiske objekter, for eksempel kan et romlig objekt som en instans av By ha et navn av typen String, antall_innbyggere av typen number og en geometrikolonne av typen region. Region og andre romlige datatyper i DBMS forekommer nå på samme abstraksjonsnivå som String og float.

Abstraksjon av region i instansen By i den geografiske datamodellen (dens uavhengighet av romlig datamodell) tillater et hvilket som helst valg av romlig datamodell (spagetti, nettverk, eller topologisk). Så lenge de operasjonene som er synlige i brukergrensesnittet er de samme, gjør det ingen forskjell fra programmererens perspektiv. En topologisk representasjon tilbyr naturlig nok bedre støtte for topologiske operasjoner. Eksempelvis kan spørsmål som om en polygon har linjesnitt med en annen polygon i en region besvares lettere ut fra en topologisk modell enn en spagetti-modell som krever en kompleks, kostbar, og mindre robust beregning.

Figur 45 viser lagring av data og operasjoner i en objektrelasjonell database.



Figur 45 Arkitektur for en ORDBMS, med geometriske datatyper

Spørrespråk som blir brukt i ORDBMS er ObjectSql og pl/sql, begge disse språkene er utvidelser av SQL. Poenget med utvidelsen er å gi støtte for objektmodellen. Typiske utvidelser inkluderer spørringer som involverer nøstede objekter, inkludering av metoder/funksjoner i søkepredikater og spørringer som involverer abstrakte datatyper.
Nedenfor er en oversikt over måter og muligheter for å definere ADTer med PL/SQL.

8.3.1.1 Objekttyper

En objekttype er en datatype som kan brukes på lik linje med andre vanlige datatyper som *NUMBER* og *VARCHAR2.* For eksempel man kan spesifisere en objekttype som datatype for en kolonne i en relasjonell tabell, og man kan deklarere en variabel av denne typen.
Objekttyper har to deler; den ene delen er alle attributtene og den andre er alle metoder assosiert til objekttypen. Hvert attributt har en deklarert datatype, som kan være en vanlig datatype eller også en annen objekttype. Attributtene til en objektinstans inneholder objektets data.

Metoder er prosedyrer eller funksjoner som tilbys for å gjøre applikasjoner i stand til å utføre nyttige operasjoner på attributtene til objekttypen. Metoder er et valgfritt element i en objekttype.
Når man oppretter en variabel av en objekttype, oppretter man en instans av den typen: Resultatet er et objekt. Et objekt har alle metoder og attributter som er definert for den aktuelle typen. Siden en objektinstans er en konkret ting, kan man gi verdier til objektets attributter og kalle dets metoder.

En prinsipiell bruk av metoder er å tilby tilgang til objektets data. Man kan definere metoder for operasjoner som en applikasjon ønsker å utføre på data slik at applikasjonen selv ikke trenger å inneholde kode for disse operasjonene. For å utføre operasjonen, kaller en applikasjon den aktuelle metoden i det aktuelle objektet. Man bruker `CREATE TYPE`-uttrykk for å definere objekttyper og samlingstyper.

Syntaksen for å definere en objekttype:

```
CREATE TYPE t AS OBJECT (
    liste av attributter og metoder
);
/
```

- ➢ Legg merke til slash på slutten, denne trengs for å få SQL-plus til å prosessere typedefinisjonen.

For eksempel er her en definisjon av en type punkt, som består av to tall:

```
create type pointtype AS OBJECT
       (x NUMBER, t NUMBER);
/
```

For å lese mer om typearv og hvordan man definerer referanser i objekt attributer se: ref : Oracle9i Application Developer's Guide-Object-Relational Features Release 1 (9.0.1)

8.3.1.2 Metoder

I en typedefinisjon kan det være tre typer metoder:

- Member
- Static
- Constructor method

Constructor method: Constructor method er en metode som systemet definerer for hver objekttype. Man kan kalle en types constructor metode for å konstruere eller opprette en objektinstans av typen. I pl/sql-koden i det følgende eksempel opprettes det et nytt objekt (en instans) av *pointtype* med verdien til variablene x0 og t0 satt inn i objektets attributter, og objektet tilordnes en variabel (p0).

```
t0:=1030;
x0:=277;
p0:= NEW pointtype(x0,t0)
```

Member method: Member methods er metoder som gir applikasjonen aksess til dataene i et objekt. Man definerer en member method i objekttypen for hver operasjon som man vil at et objekt av denne typen skal være i stand til å utføre.
Member methods har en innebygget parameter *SELF* som betegner det objektet som den aktuelle metoden er blitt påkalt i. Member methods kan referere attributter og metoder i *SELF.*

Følgende kodeeksempel viser en metodedeklarasjon for en member method:

```
CREATE type linetype as object(
    lpoints ptable,
    member function finnKrysspunktet (l linetype)
    return pointtype));
```

(Attributtet lpoints av typen ptable vil bli forklart i neste delkapittel.)

Denne metodedeklarasjon sier at finnkrysspunktet er en funksjon. En funksjon må alltid ha en returverdi. I dette tilfellet får funksjonen et linetype-objekt som en innparameter til funksjonen, og funksjonen returnerer et punkt-objekt.

Dette eksemplet er forenklet. Det viser ikke hvordan man spesifiserer kroppen til metoden `finnKrysspunktet`. Dette må gjøres med `CREATE OR REPLACE TYPE BODY` uttrykket som i dette eksemplet:

```
create or replace type body linetype as
/* her kommer koden av alle funksjoner eller prosedyrer som er spesifisert
med objekt deklarasjonen*/

member function finnkrysspunktet (l linetype) return pointtype
as

/* all innmaten til funksjonen finnkrysspunktet må skrives her
og funksjonen må returnere et punkt objekt før den avslutter.*/

end finnkrysspunktet;

/* Her kommer neste funksjon o.s.v. */
End;
/
```

Static method: Static methods hører hjemme i objekttypen, ikke i de enkelte objekter (instanser). Man bruker statiske metoder for operasjoner som er globale til typen og som ikke trenger å referere data i et bestemt objekt. Man kaller en statisk metode ved å bruke en ”dot” notasjon for å kvalifisere metodekallet med navnet på objekttypen: *type_navn.metode().*

8.3.1.3 Samlinger

[Oracle9i 2002] støtter to *mengde*-datatyper: vararrays og nøstede tabeller. Mengdetyper kan bli brukt på alle de stedene hvor andre datatyper kan bli brukt: man kan ha objektattributter av mengdetyper, kolonner av mengdeattributter, osv.

- En vararray er en ordnet samling av elementer: Posisjonen for hvert element har et indeksnummer, og man bruker dette nummeret til å aksessere bestemte elementer. Når man definerer en varray, må man spesifisere det maksimale antall elementer den kan inneholde. Man kan endre dette antallet på et seinere tidspunkt.

- En nøstet tabell kan ha et hvilket som helst antall elementer: Intet maksimum blir spesifisert i definisjonen av tabellen. Rekkefølgen av elementene blir ikke bevart. Man kan gjøre select, insert, delete osv. i en nøstet tabell på samme måte som man gjør på en ordinær tabell. Elementer av en nøstet tabell er faktisk lagret i en separat tabell som inneholder en kolonne som identifiserer foreldretabellraden eller objektet som hvert element tilhører.

Hvis man bare trenger å lagre et bestemt antall elementer, eller gå gjennom elementene i en ordnet rekkefølge, eller ofte hente eller manipulere hele samlingen som en verdi, da bør man bruke en varray.

Hvis man trenger å kjøre effektive spørringer på en samling, håndtere et tilfeldig antall av elementer, eller utføre en mengde insert/update/delete operasjoner, bør man bruke en nøstet tabell.

Spesielt hvis man vet at man kommer til å gjøre oppdateringer for en rad eller et objekt bør man absolutt bruke en nøstet tabell, fordi man ikke kan gjøre oppdateringer på en varray: Verdiene i varrays kan ikke endres, de er "immutable". For å illustrere dette kan vi se på et eksempel hvor vi har en objekt relasjonell eller en relasjonell database over forfattere. I denne databasen vil vi lagre generell informasjon om alle forfattere, og en liste over alle bøkene hver forfatter har skrevet. For hver gang en forfatter skriver en ny bok må det gjøres en oppdatering i listen til forfatteren ved å legge den nye boka i listen. Hvis listen er lagret som en nøstet tabell, kan man legge inn den nye boka ved å gjøre en oppdatering i listen som inneholder de allerede utgitte bøkene. Hvis derimot listen er lagret som en varray og man forsøker å legge inn den nye boka ved å gjøre en oppdatering på listen, vil den nye boka erstatte hele listen, det vil si. at det eneste som vil være i listen etter oppdateringen, er den nye boka. Hvis man likevel vil bruke varray for å lagre boklisten må man i enhver oppdatering for å legge inn en ny bok legge inn alle bøkene på nytt i listen.

Hvis kolonnen i en nøstet tabell er en objekttype, kan tabellen bli vist som en flerkolonne tabell med en kolonne for hvert attributt av objekttypen. For eksempel sier

```
CREATE  type ptable as table of pointtype;
/
```

at type *ptable* er en tabell som har rader (tupler) av typen *pointtype* som igjen har to komponenter, x og t , som er reelle tall.

Konstruktørmetoden for den nøstede tabell typen **ptable** for oppretting av `pointtype` objekter ser slik ut:

```
ptable(  pointtype(85,0755),
         pointtype(277,1030)
      );
/
```

En tabelltypedefinisjon allokerer ikke plass. Den definerer en type som kan bli brukt som:

- Datatype for en kolonne for en relasjonell tabell
- Et objekttypeattributt:

  ```
  CREATE type linetype as object(
   lpoints ptable
   );
  ```

  Her har vi deklarasjonen for en line-objekttype, *linetype*, som har et attributt *lpoints* som er av datatypen ptable. Det vil si at hvert enkelt linetype-objekt har en tabell knyttet til seg som er en samling av alle punktene som dette bestemte line-objektet består av.

- En pl/sql variabel, parameter, eller funksjon return type.
  (kan se på eksempler senere i kapitlet)

8.3.1.4 Flernivå samlingtyper

Flernivå samlingtyper er samling typer som har elementer som direkte eller indirekte er en annen samlingtype. Mulige flernivå samlingtyper er:

- Nøstet tabell av nøstet tabelltype
- Nøstet tabell av varraytype
- Varray av nøstet tabelltype
- Varray av varraytype
- Nøstet tabell eller varray av brukerdefinerte type som har et attributt som er en nøstet tabell eller varray type. Her er et eksempel:

```
CREATE type linetable as table of linetype
```

Her oppretter vi en tabell type av *linetype* som er en brukerdefinert type, som har et attributt *lpoint,* som er av typen nøstet tabell *ptable*.

8.3.1.5 Objekttabeller

En objekttabell er en spesiell type tabell hvor hver rad representerer et objekt.
Det er to måter å se på tabellen AlleTog som er beskrevet nedenfor:

- Som en enkel-kolonne tabell hvor hver rad er et tog-objekt som tillater å utføre objekt orienterte operasjoner.

```
Create table AlleTog of togtype
   nested table line store as n_line (
     nested table lpoints store as n_lpoints));
```

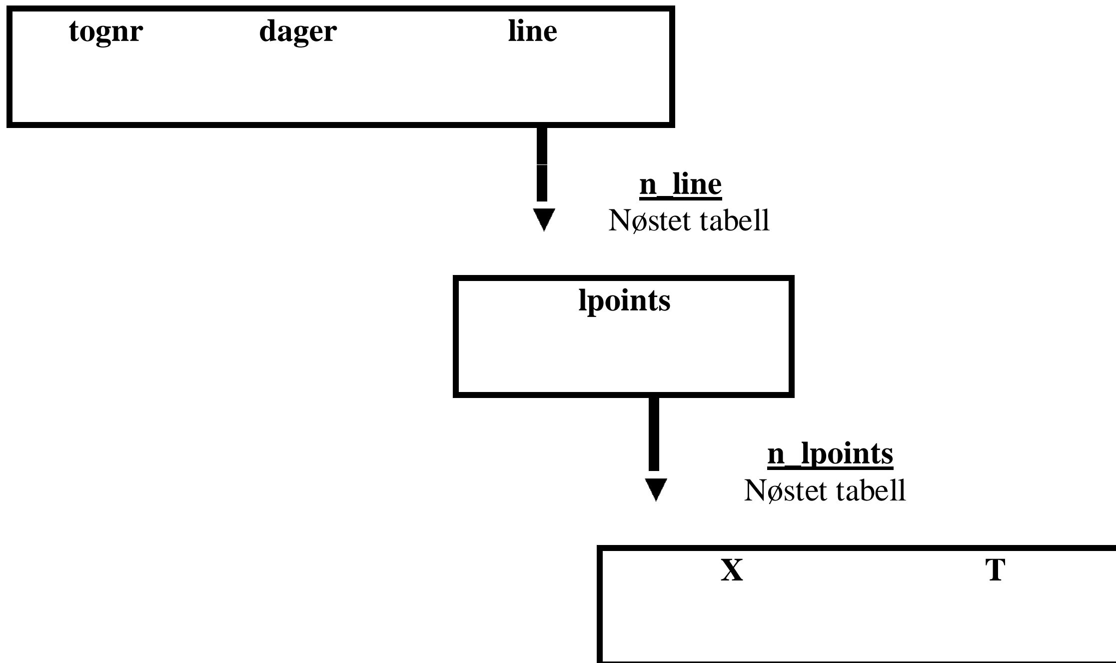
- Som en fler-kolonne tabell hvor hvert attributt av objekttypen tog okkuperer en kolonne som tillater å utføre relasjonelle operasjoner.

```
Create type togtype as object (
     tognr NUMBER, dager VARCHAR(100),line linetable,
     member function krysspunkt (t togtype)
     return pointtype));
```

For eksempel vil uttrykket ovenfor opprette et tog-objekt og definere en objekttabell for *tog*-objekter.
Tabellen "*AlleTog"* er et eksempel på en flernivå samlingstype, som er en nøstet tabell av nøstede tabeller. Eksemplet viser at hvert tog har et attributt *line* som har en type av flernivå samling (nøstet tabell av en objekttype som har et nøstet tabellattributt *lpoints*).
Oracle lagrer nøstede tabelldata for alle objekter i en enkel lagringstabell som er assosiert med objekttabellen. I eksemplet ovenfor blir tabellen **n_line** brukt som lagringstabell for line-attributtet og tabellen **n_lpoints** som lagringstabell for lpoints.
Oracle tilbyr innebyggede konstruktører for verdier av en deklarert type, og disse konstruktørene har samme navn som objekttypen

Nedenfor er det en figur som illustrerer nøstingen av tabeller.

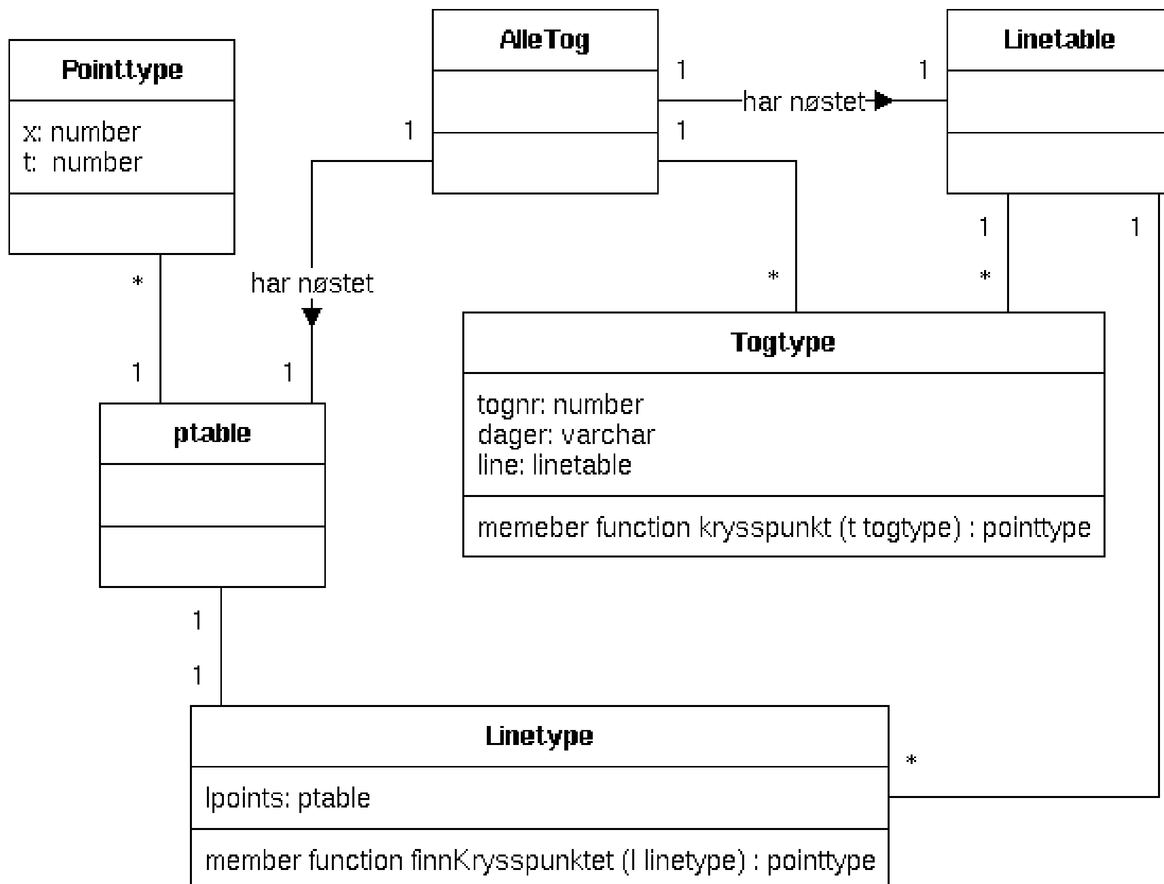
AlleTog
Tabell



Figur 46: n_lpoints tabellen er nøstet i n_line tabellen og n_line tabellen er nøstet i line kolonnen

8.3.2 Datamodellen for togobjekter i databasen

Hittil har vi sett på begreper og hvordan ting fungerer i den objektrelasjonelle verden.
Her kommer en oversikt over hvordan ting henger sammen i vår objektrelasjonelle database.



Figur 47: Modell over abstrakte datatyper i databasen for å finne krysspunktet

Beskrivelse av arkitekturen for tabellen *AlleTog*:

- Lager en ”pointtype” av typen object som representerer et punkt som har koordinatene x og t.
- Lager en ”ptable” av typen table, som er en tabell som inneholder alle punkt-objektene.
- Lager en ”linetype” som også er av typen objekt og som representerer en linje. Det eneste attributtet den inneholder er ”lpoints” som er av datatypen ptable, altså en tabell med alle punkter som tilhører denne bestemte linjen. Hvert enkelt linetype-objekt har altså sin egen bestemte lpoints tabell.
- Lager en tabell ”linetable”, som er av typen table og som inneholder alle linje-objektene.
- Lager en ”togtype” av typen objekt som representere tog-objektene. Attributtene i tog-objektet er et unikt tognr, dager og line, det siste er av typen linetable. Hvert objekt har en eget tabell over alle linjene som er knyttet til toget og attributtet som inneholder denne tabellen heter line.
- Lager hovedtabellen som inneholder alle tog-objektene. Tabellen heter togene og er av typen togtype. Det vil si at tabellen AlleTog har attributter tognr, dager og line som har en type av flernivå samling.

Ut fra tabellstrukturen kan et tog ha flere toglinjer og hver toglinje har flere punkter, men i den avgrensede problemstillingen som vi bruker som eksempel og skal se på videre skal vi holde oss til en toglinje per tog. Her er et lite eksempel på hvordan man initialiserer slike objekter, her

skal vi bare legge inn to av punktene til toget som har tognr 609. Flere punkter kan legges inn på samme måte:

```
insert into AlleTog values
              (609,'alle dager',
               linetable
                     (linetype
                       (ptable
                          (pointtype
                             (89.57,747))
                          (pointype
                             (202.38,906)))));
```

.
Nå har vi lagret relasjonelle objekter i databasen. Nå gjenstår det å lage funksjoner og prosedyrer i databasen for å finne krysningspunktet mellom to togfremføringer. Hvis vi har algoritmen liggende i databasen slipper vi å legge den inn i applikasjonsprogrammet. For å programmere algoritmen bruker vi pl/sql.

8.3.3 Algoritmen i ORDBMS

SQL er et utmerket spørrespråk, men det er begrenset. Oracle Corporation har utvidet språket med prosedyreorienterte elementer og dermed kommet fram til språket som er kjent som "Programming langage/Structured Query Language" (PL/SQL), altså prosesserings språk. Hvis man er kjent med et annet prosedyreorientert programmeringsspråk, vil man finne at PL/SQL bygger på de samme prinsippene som Pascal, C, eller Visual Basic 6. Det inneholder trekk fra moderne språk slik som:

- Feilhåndtering
- Informasjonskjuling
- Objektorientert programkonstruksjoner (OOP)

PL/SQL tillater også innpakning av SQL-uttrykk og datamanipulering inne i blokker. SQL-uttrykk blir brukt for å hente data og PL/SQL-kontrolluttrykk blir brukt for manipulere eller prosessere data i et PL/SQL-program. Data kan bli lagt inn, slettet eller oppdatert gjennom en pl/sql blokk, noe som gjør det til et effektivt transaksjonsprosesseringsspråk.

PL/SQL tillater å bruke alle SQL sammenlignings-, mengde- og rad-operatorer i SQL-uttrykk:

Operator	Beskrivelse
ALL	Sammenligner en verdi med alle verdier i en liste eller returnert av en subquery og returnerer TRUE hvis alle de individuelle sammenligningene gir TRUE.
ANY, SOME	Sammenligner en verdi med alle verdier i en liste eller returnert av en subquery og returnerer TRUE hvis noen av de individuelle sammenligningene gir TRUE.
BETWEEN	Tester om en verdi ligger i en spesifisert område.
EXISTS	Returnerer TRUE hvis en subquery returnerer minst en rad.
IN	Tester for mengde-medlemskap.
IS NULL	Tester for nuller.
LIKE	Tester om en character string inneholder et spesifisert mønster

8.3.3.1 CURSOR

Ifølge [Oracle/PLSQL Topics 2004] en cursor er en mekanisme som man kan bruke for å navngi et "select statement". PL/SQL bruker to typer av cursorer: *implicit* og *explicit*. PL/SQL deklarer en cursor implisitt for alle SQL datamanipuleringsuttrykk, inkludert spørringer som returnerer bare en rad. For spørringer som returnerer mer enn en rad, avhengig av hvor mange rader som tilfredsstiller søkekriteriet, må man enten deklarere en eksplisitt cursor, bruke en cursor *FOR* løkke, eller bruke en *BULK COLLECT* setning.

Man trenger tre kommandoer for å kontrollere en cursor: *OPEN* ,*FETCH* og *CLOSE* . Først initialiserer man cursoren med et *OPEN* uttrykk som identifiserer resultatmengden. Så kan man eksekvere *FETCH* fortløpende inntil alle rader er blitt hentet ut, eller så kan man bruke *BULK COLLECT* setningen til å hente alle radene på en gang. Når den siste raden er blitt prosessert, frigjør man cursoren med *CLOSE* uttrykket. Man kan prosessere flere spørringer parallelt ved å deklarere og åpne flere cursorer samtidig.

En cursor kan ha parametere som kan forekomme alle steder i spørringen hvor en konstant kan forekomme. Scopet av cursor-parametere er lokale til cursoren, som betyr at de kan bare bli referert i spørringen spesifisert i cursor-deklarasjonen. Cursoren kan også ha en return_type, men den må representere en tuppel eller en rad i en databasetabell.
For å lese mer om cursor - se [PL/SQL User's Guide and references Release2(9.2)]

Her er et eksempel på en enkel blokk i PL/SQL med cursor:

```
DECLARE
    Deklarasjon av konstanter, variabler, cursorer og exceptions
    --Cursor c1 is select ......
BEGIN
    PL/SQL and SQL uttrykk
    --Open c1 fetch c1 into ---  close cursor
EXCEPTIONS
    Behandling av feiltilstander
END;
```

DECLARE og EXCEPTION-nøkkelordene er valgfrie, men BEGIN og END nøkkelordene er nødvendige.

Det er to typer av metoder man kan lage i PL/SQL :

- **Prosedyrer**: En prosedyre er en navngitt PL/SQL programblokk som kan utføre en eller flere oppgaver. Den kan ha tre typer av parametere :

    1) IN : Brukes for å sende en verdi inntil programmet. Read-only type av verdi.
    2) OUT : Brukes for å sende en verdi tilbake fra programmet. Write-only type av verdi.

3) IN OUT : Sender en verdi inn og returnerer eventuelt den endrede verdien tilbake.

Syntaksen for en prosedyre :

```
 create or replace procedure prosedyreNavn
      [(parameter1[,parameter2..])]
IS
    [Konstant/variabel deklarasjoner]
Begin
    Eksekverbar uttrykk
[Exception
    exception håndterings uttrykk]
END [ prosedyreNavn];
```

- **Funksjoner :** En funksjon lik en prosedyre. Som en prosedyre er den også en lagret kodebit. Hovedforskjellen mellom en funksjon og en prosedyre er at en funksjon alltid gir en verdi tilbake til den kallende blokken. En prosedyre kan gjøre oppdateringer i databasen mens en funksjon har restriksjoner; den kan ikke gjøre en INSERT, DELETE, eller UPDATE i databasen. Les mer om restriksjoner for funksjoner her: http://www.hk8.org/old_web/oracle/prog2/ch17_04.htm
  Ifølge [Nilesh Shah] kan en funksjon karakteriseres med følgende:

    - En funksjon kan ha en, flere eller ingen parametere
    - En funksjon må ha et eksplisitt return-uttrykk i den eksekverbare seksjonen for å returnere en verdi.
    - Datatypen for returverdien må bli deklarert i funksjonens øverste del (header), på denne måten :

```
create or replace function
   finn_togene(tnr1 number,tnr2 number)
     return pointtype is
```

    - En funksjon kan ikke bli eksekvert som et frittstående program.

En funksjon kan ha parametere av typen IN, OUT, og IN OUT typer, men den primære bruken av en funksjon er å returnere en verdi med et eksplisitt RETURN uttrykk.

Her er et eksempel på hvordan man kan bruke en pl/sql blokk. Denne blokken kaller på en funksjon for å finne kryssningspunkt mellom to tog.

```
DECLARE
     p    pointtype;
     tog_nr_1 number:=609;
     tog_nr_2 number:=62;
BEGIN
     p:=finn_togene(tog_nr_1,tog_nr_2);
     dbms_output.put_line('X-verdi'|| ' ' || p.x);
     dbms_output.put_line('T-verdi'|| ' ' || p.t);
END;
```

```
/
```

I *declare*-delen har vi en variabel som har en brukerdefinert datatype *pointtype*, og to variabler som er av datatypen number og som har fått konstante verdier 609 og 62. I *begin*-delen kaller vi på en funksjon som heter *finn_togene* og som har to inn-parametere, altså tognumre. Denne funksjonen returnerer en *pointtype* som blir lagret i variabelen p. For å se hvilken t og x verdi *pointtype* p har fått skriver vi ut til SQL-bufferen med kommandoen `dbms_output.put_line` (Husk å skrive "set serveroutput on" i SQL prompten for å kunne se utskriften, før du skriver denne blokken i prompten).

```
create or replace function
    finn_togene(tnr1 number,tnr2 number)
      return pointtype is

    point pointtype;
    tog1 togtype;
    tog2 togtype;
```

Funksjonen finn_togene ovenfor henter ut tog_objektene som har tognumre som er blitt sendt inn som parametere og har et punkt-objekt som returtype.
Datatypen til en cursor må være i samsvar med datatypen til den variabelen som cursoren skal fetche into, som i dette tilfellet er tog1 og tog2. På denne måten kan man også deklarere en variabel som har samme datatype som en cursor, og være sikker på at cursor og variabelen har samme datatype:

```
tog1 t1%rowtype;
tog2 t2%rowtype;
```

For å hente ut et objekt istedenfor en rad fra en objekttabell sier man **select value(t)** istedenfor **select * .** I denne funksjonen har vi deklarert to cursorer **t1** og **t2** hvor begge henter ut hvert sin tog.

```
cursor t1 is
   select value(t)
   from AlleTog t where tognr=tnr1;

cursor t2 is
   select value(t)
   from AlleTog t where tognr=tnr2;
```

Som sagt tidligere må vi bruke fetch for å legge verdien fra en cursor inn i en variabel, og det må gjøres innenfor *Begin* blokken**.**

Det gjøres på denne måten:

```
Begin
   OPEN t1;
     FETCH t1 INTO tog1;
     close t1;

  OPEN t2;
    FETCH t2 INTO tog2;
    close t2;
```

```
        point:=tog1.krysspunkt(tog2);

    return point;
```

Her kan vi også se at det er et kall på funksjonen **tog1.krysspunkt** som får inn et tog-objekt *tog2* som parameter. Den funksjonen som blir kalt på, er den som tilhører *tog1*-objektet. Altså har alle tog -objekter en funksjon *krysspunkt* som får inn et tog-objekt som parameter og som returner en verdi som er et punkt-objekt. Funksjonen returnerer dette punkt-objektet tilbake til den blokken som kaller på denne funksjonen.

Istedenfor to cursorer kan vi greie oss med bare en cursor, der denne ene cursoren har tognummer som inn-parameter. Da åpner vi cursoren på denne måten isteden:

```
 OPEN t1(tnr1);
    FETCH t1 INTO tog1;
    close t1;

OPEN t1(tnr2);
    FETCH t1 INTO tog1;
    close t1;
```

Her er det helt nødvendig at vi lukker den første åpningen av cursoren før vi åpner cursoren på nytt med en annen parameter. I tilfellet hvor vi har to cursorer er det nærliggende å lukke begge cursorer samtidig på slutten av programmet, men av sikkerhetsmessige grunner er det best å lukke cursoren med en gang man er ferdig med den.

Funksjonen *krysspunkt* henter ut indeksen for den første linjen i tabellen, og lagrer den indeksen i variabelen *j*. Funksjonen henter også ut indeksen for linjen til det andre tog objektet, som er en parameter inn til funksjonene (egentlig så hentes ut begge linjene fra første plassene i tabellene, fordi, det er blitt bare lagret en linje per tog). Til slutt utføres et kall på metoden *finnkrysspunktet* som ligger i linje-objektet for tog nr 609 og sender med det andre linje-objektet som parameter inn til funksjonen. Funksjonen returnerer et punkt som blir lagret og sendt tilbake til funksjonen finn_togene som kalte på denne funksjonen. Her er syntaksen :

```
      point pointtype;

      i  BINARY_INTEGER;
      j  BINARY_INTEGER;

   Begin
      i := t.line.FIRST;
      j := self.line.FIRST;
      point:=self.line(j).finnkrysspunktet(t.line(i));

   return point;
```

Funksjonen finn_kryssspunktet henter ut x- og t-verdiene til alle punkter for begge linjer og legger dem inn i fire forskjellige tabeller. For eksempel vil x1-tabellen inneholder alle x-verdiene til alle punktene for dette linje-objektet og t2 inneholder alle t-verdiene til punkter som tilhører linjen som kom inn som parameter. Den første verdien i x1-tabellen og den første verdien i t2-tabellen tilhører begge det samme punkt-objektet. I resten av funksjonen beregner vi krysningspunktet mellom de to linjene med en algoritme. Til slutt oppretter vi et nytt punkt

med den x- og t-verdi som vi har funnet som krysningspunkt og returnerer dette punktet tilbake til funksjonen krysspunkt som kalte på denne funksjonen.

```
type point_tab is  table of number index by binary_integer ;
x1 point_tab;
t1 point_tab;
x2 point_tab;
t2 point_tab;
p0 pointtype;

Begin

FOR element IN 1..lpoints.COUNT
LOOP
   x1(element):= ( lpoints(element).x );
   t1(element):= ( lpoints(element).t );
END LOOP;

FOR element IN 1..l.lpoints.COUNT
LOOP
   x2(element):= ( l.lpoints(element).x );
   t2(element):= ( l.lpoints(element).t );
END LOOP;

/*Algoritmen som finner kryssningspunktet og lagrer verdiene i variablene x0
og
  t0 */

p0:= NEW pointtype(x0,t0);

return p0;
```

8.3.3.2 Resultat

Hvis vi nå bruker blokken til å kalle på funksjoner i databasen for å finne krysningspunktet vil vi få dette resultatet:

```
X-verdi  278.23184372256
T-verdi  1037.1501316944
```

Vi kan også kalle på prosedyrer og funksjoner fra en applikasjon på denne måten:
For å lese mer om dette se ” Oracle9i JDBC Developer’s Guide and Reference (Basic Features 3)”

```
String query= "begin ? := finn_togene(?,?); end;";
CallableStatement cstmt = conn.prepareCall(query);
cstmt.registerOutParameter (1, Types.STRUCT,"POINTTYPE" );

cstmt.setInt(2,609);
cstmt.setInt(3, 62);
cstmt.execute();
Struct punkt = (Struct)(cstmt.getObject(1));
Object[] attrs = punkt.getAttributes();
System.out.println("Sted(x) : " + punkt.getAttributes()[0]);
System.out.println("Tid(t)  : " + punkt.getAttributes()[1]);
```

Dette gir dette resultatet :

```
*******************************************
Tog til Oslo s: 62 -602
-------------------------------------------
Tog til Bergen: 609 -61
*******************************************

Oppgi tognr til toget fra Oslo s til Bergen:
609
Oppgi tognr til toget fra Bergen til Oslo s:
62

Sted(X) : 278,23184372256
Tid (t) : 1037.1501316944
```

8.4 Oppsummering

Relasjonelle databasesystemer lagrer data i tabeller ved å lenke tabellene med fremmed nøkler, de felter som har en felles verdi område.
Objektrelasjonelle databasesystemer lagrer også data i tabeller, men hver av tabellene representerer et objekt som romlige brukere er mer komfortabelt med. Tabellene er arrangert i et hierarki med objekter lavere i hierarkiet som arver attributter fra de objekter på høyere nivå. Det er også mulig å ha objekter som er sammensatt av andre objekter.

8.4.1 Relasjonelle databasesystemer

Ifølge [Silvia Nittel]ulemper ved bruk av RDBMS for romlige data er:

- Geometrien må representeres som en mengde av Integer attributter.
- DBMS har ingen kunnskap om geometri.
- Det er ingen støtte for romlige operasjoner i en RDBMS.
- Forhold mellom tabellene må rekonstrueres hver gang under spørreprosesseringen, dette kan være veldig dyrt for komplekse romlige objekter.

Relasjonell DBMS blir fortsatt brukt for å lagre romlige data. Grunnen til dette er at RDBMS'r er kommersielt tilgjengelige, er skalerbare til veldig store data størrelser, er robuste systemer, og at det er mulig å bruke BLOBS (binary large objects) for å lagre geometriattributter.

8.4.2 Objektorientering

Den dominerende utviklings metodologien for objekt orienterte applikasjoner er basert på UML (Unified Modelling Language).
Ulikt en relasjonell database, assosiasjoner i en objekt database er enten av formen **'is a'** eller formen **'has a'**. Det første tilfellet tillater man å bygge hierarkier og tillater arv. Det andre tillater strukturer som viser aggregering, eller hvordan et objekt er sammensatt av andre objekter.
Største fordelen av å bruke objekter som basis for database design er at objekt termer brukes i hverdagen og er det er lett å identifisere objekter for programmerere. For det andre objekt modellen kan direkte bli implementert i en relasjonell form, spesielt aggregeringen eller **'has a'** forhold.

Objekt modeller er mer gode på å beskrive mer komplekse strukturer enn relasjonelle modeller. Uansett mesteparten av databasehåndteringssystemer er ikke i stand til å implementere alle klasse strukturer som kan bli modellert.

8.4.2.1 Objektorienterte databasesystemer

Ifølge [Steve McClure 1997] har den semantiske mistilpassning mellom objektorienterte språk og relasjonelle databaser ledet til utviklingen av objektorienterte databasehåndteringssystemer som direkte støtter objektmodellen.

8.4.2.2 Objektrelasjonelle databasesystemer

Objektrelasjonelle databasesystemer er fortsatt relasjonell, fordi data er lagret i tabeller og kolonner, og SQL med utvidelser for å aksessere ADT'r er fortsatt det primære grensesnittet mot database, og brukes for datadefinisjon, datamanipulering og spørring. ORDBMS tilbyr mer fleksible strukturer for å representere mer komplekse data og frittstående funksjoner, men tilfredsstiller ikke det fundamentale kravet om innkapsling av operasjoner med data. Det er ikke noen direkte støtte for klienter, objektorienterte språk og deres objekter, programmere tvinges til å oversette mellom objekter og tabeller.
ORDBMS konverterer data automatisk mellom et objektorientert format og et RDBMS-format. Dette er den største fordelen ved ORDBMS databaser, siden det da ikke er nødvendig for programmereren å skrive kode for å konvertere mellom disse formatene og det er lett å aksessere databasen fra et objektorientert programmeringsspråk. Imidlertid synker ytelsen for databasen, fordi databasehåndteringssystemet må bruke tid på konverteringen.

8.4.3 Konklusjon

Det viser seg at programmere er vant til å tenke med objekt termer, derfor er det enkelt for programmere å identifisere alle objekter som trengs til en applikasjon, alle assosiasjoner mellom objektene, og definisjonen av objekter. De ser altså på metodologien for å finne objekter som enkel. Metodologien for å finne tabeller i en objektrelasjonell database kalles for normalisering. Normalisering er en prosess som produserer en database med minimal redundans. Normalisering er en vel utviklet metodologi, som hvis den blir fulgt, produserer et skjema som kan konverteres direkte til tabeller i RDBMS, som for eksempel SQLServer og Oracle. Det er lett å gjøre en feil i prosessen og utvikle for mange tabeller, eller definere attributter til feil tabeller. Gruppering av ugrupperte modeller er et alternativ.
Alle tabellkolonner er avhengig av en primærnøkkel til å identifisere kolonner. Med en gang den spesifikke kolonnen er identifisert, kan data fra en eller flere rader assosiert med denne kolonnen hentes ut eller endres. For å lagre objekter i en relasjonell database, må objektene beskrives ved hjelp av enkle termer som string, integer, date også videre. Det å fordele kompleks informasjon på enkle data tar tid, og disse data må kanskje fordeles på ulike tabeller. Og for å kunne hente ut all informasjon om et objekt må flere tabeller kanskje joines sammen. Objektdatabaser bør brukes når det er komplekse data og/eller komplekse data-assosiasjoner... Ulemper ved bruk av objektdatabaser sammenlignet med relasjonelle databaser er lavere effektivitet når data er enkle og assosiasjonene er enkle og relasjonelle tabeller er enklere. Objektdatabaser bør derfor ikke brukes når det vil være få joininger tabeller og det er store mengder av enkle data.
Relasjonelle databasesystemer tilbyr liten støtte for data manipulering, og objektorientert programmering er ypperlig for data manipulering, derfor legges alle funksjoner for manipulering av data hentet fra databasen i det objektorienterte Java programmet.

9 Regning med romlige verdier

En romlig verdi sees på om et tett punktmengde vi kan utføre diverse operasjoner på. Åpenbare operasjoner er alle mengdeoperasjoner som for eksempel snitt, union og differanse. Operasjoner som går ut på å finne grensen og den inverse operasjonen som er å finne alle punkter som ligger på og innenfor en lukket grense. Videre kan operasjoner være av typen finne ut volum, omkrets og areal.
En punktmengde er definert til å være matematiske kurver som oppfyller en rekke krav, kravene kan for eksempel være definisjon av hvilke type kurve vi har å gjøre med. For eksempel kan kurven til sirkel definere hvis krav til antall grad og fast radius.
I dette kapitelet skal vi forsøke å beskrive hvordan antall dimensjon spiller inn i på operasjoner mot romlige verdier. Vi starter med diskutere mengdeoperasjoner som snitt før vi så går over til å beregne volum og arealer i forskjellige dimensjoner.

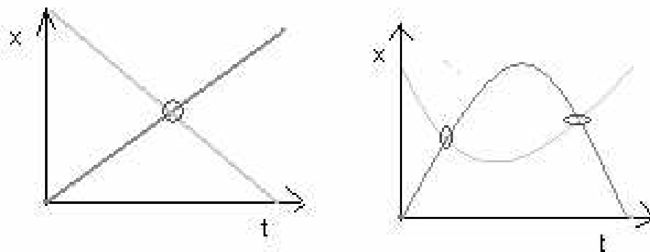
9.1 Objekt A ∩ Objekt B = Punkt / Punkter

For å finne snittet mellom to rettet linjer eller grafer kan man sette grafene lik hver andre. Verdiene man får er det hvor linjene skjærer hverandre, det vil si der hvor linje A og linje B deler et punkt. Hvis et objekt beveger seg med konstant fart kan vi finne dets endringsfunksjon ved å utføre regresjon eller finne linking for en rett linje ved hjelp av formelen:

$y = ax + b$, hvor $a = (y2-y1)/(x2-x1)$ og b er der hvor linjen skjærer 2-aksen.

Vi kan omforme denne likningen til å gjelde våre x,t grafer slikt:
$x = at + b$ og $a = (x2-x1)/(t2-t1)$

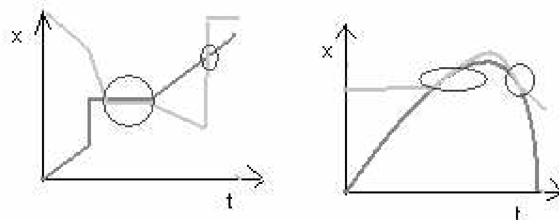
Etter at funksjonene for linjene er funnet kan man sette funksjonsuttrykk mot hverandre og se hvilke verdier man vil få. Figur 48 (figuren til høyre) vil som vi ser gir oss to verdier. Funksjonene til disse objektenes bevegelse kan vi bestemme ved hjelp av regresjon eller ved å dele opp bevegelsen til hver av dem i mindre linjer og se på hver linjestykke som en rettlinjet graf. Til slutt kan vi kombinere linjestykkene mot linjestykkene hos det andre objektet.



Figur 48 Snitt mellom to regelmessige kurver

9.2 Objekt A ∩ Objekt B = Linjer

Hittil har vi bare sett på objekt som kun deler et eller flere punkter. Ved noen tilfeller kan et objekt bevege seg i en og samme strekning over en bestemt tid og dermed har vi ikke lenger å gjøre med punkter men med punktsamlinger som igjen utgjør linjer. Figur 49 (figuren til venstre) har to kurver hvor regresjon vil gi absolutt feil funksjon. Siden endringen i objektets bevegelse ikke følger noen kjent matematisk formel (er verken rett, potensial- eller logaritmeformel) er vi nødt til å dele bevegelsene inn i mindre deler. Når vi så har å gjøre med biter som ser ut til å følge en eller annen matematisk formel kan vi regne oss fram til funksjonsutrykket og sette funksjonene mot hverandre. Der hvor snitt resultatet gir en linje i resultat vil vi få ERROR ved utregning. Dette vil være et klart tegn på at vi har å gjøre med linjer. Hvis vi derimot betrakter to funksjonsutrykk og får krysningspunkt som ligger utenfor det intervallet vi ønsker å se på, det vil si den delen av hele banen som funksjonene er en mindre del av, vet vi at vi har å gjøre med to linjer som ikke krysser hverandre i det hele tatt.



Figur 49 Snitt av uregelmessige kurver

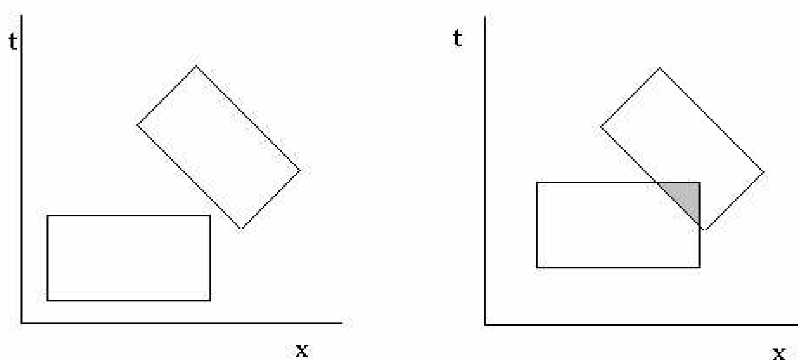
Et eksempel:

La oss anta at vi i et intervall finner ut at to linjer har funksjonene g(x) = 2x+4 og f(x) = x+4+x, når disse sjekkes opp mot hverandre får vi følgende resultat:
2x+4 = x+4+x
2x-x-x = 4 -4
0x = 0 Noe som resulterer i at begge funksjonene er like og dermed har man intersection i intervallet.

Hvis vi hadde hatt g(x) = 3x og f(x) = x+6 og intervallet lå mellom [-1,1], vil vi når vi setter g(x)= f(x) få x = 3. Med andre ord krysser funksjonene i et punkt, noe som ville resultere i intersection men siden punktet ligger utenfor vårt intervall, er det ingen intersection mellom g(x) og f(x).

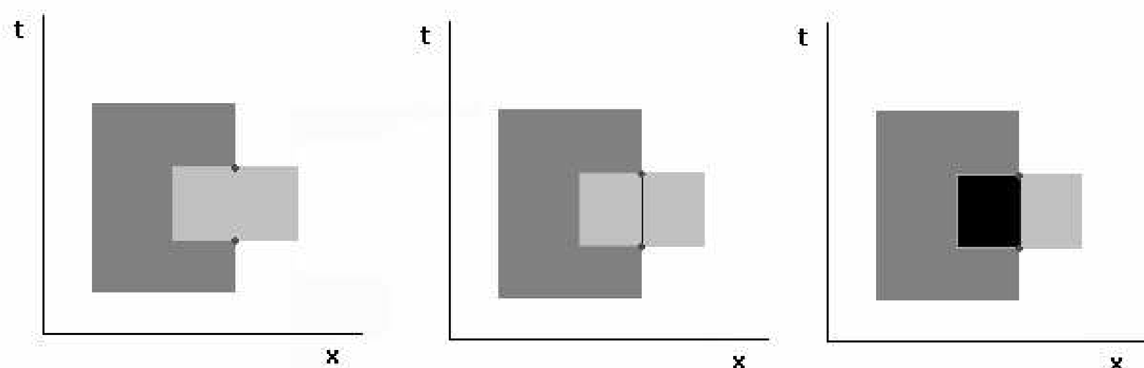
9.3 Objekt A ∩ Objekt B = Flater

Første sjekk på ikke krumme flater er sjekk av parallellitet. Hvis de er parallelle er snittet mellom de to flatene lik den tomme mengden. Hvis to flater ikke er parallelle kan man sjekke om flatenes ytterpunkter overlapper hverandre eller er slik at den enes ytterpunkter er innenfor den andre flatens ytterpunkter. Hvis det er slik at det er overlapp har man intersection. Ved krumme flater kan man også bruke sjekk av ytterpunkter til å finne om det eksiterer noen form for snittmengde.



Figur 50 Sjekk av snitt ved hjelp av ytterpunkter

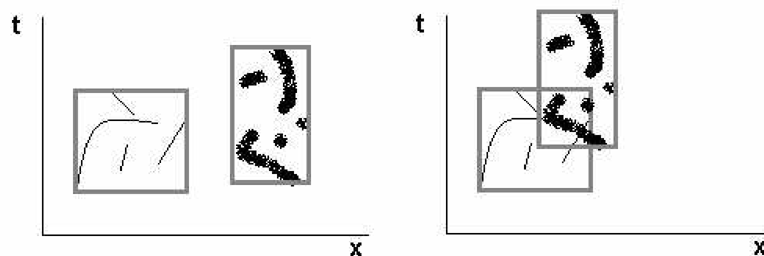
Beregne intersection resultat mellom ikke krumme flater:
Snittet mellom flater kan så beregnes ved først å finne ut hvilke to ytterpunkter som er like for de to flatene. Deretter velger man ytterpunktene til den flaten som har ytterpunkter som blir overlappet av den andre flaten. Så velger man de ytterpunktene til de flatene som eksisterer inne i den andre flaten. Slikt har man fått beregnet ytterpunktene til snittmengden, dermed vil alle punkter innen for disse punktene være en del av snittmengden.



Figur 51 Beregning av snittmengde

9.4 *Generell framgangsmåte for å sjekke snitt*

Før man startet med å beregne hvor to objekter snitter hverandre bør man sjekke om de i det hele tatt har muligheten til å snitte hverandre. En måte å sjekke dette på er å lage en boks, en bounding box, rundt ytterkantene av et hvert objekt og sjekke om verdiene innen for denne boksen er lik verdiene innenfor en annen boks. The bounding box skal være slik at den innehar alle verdiene til et objekt samtidig som den er så liten som mulig. Figur 52 viser objekter hvis bounding boxs inneholder noen like verdier og objekter med ingen like verdier.



Figur 52 Bounding box

Hvis det er slik at to bounding box snittet med hverandre gir en mengde større enn den tomme mengden kan det være mulighet for at de objekt vi har å gjøre med har en snittmengde større enn null.

Sjekke intersection i Java med bounding box rundt et objekt:

I Java biblioteket har man en klasse Bounding Box, den extends Bounds som hører under java.lang.Object. Vi kan lage en bounding box rundt et objekt ved å kalle på konstruktøren BoundingBox(Point3d lower, Point3d upper) ved hjelp av max og min verdier for x, y og z. Videre kan intersection bli sjekket ved å kjøre metoden intersect(Bounds boundsObject, BoundingBox newBoundBox) som sjekker med et annet bounding box om det er mulighet for intersection mellom de to.

9.5 Beregning av snitt:

Etter at vi har funnet to bounding box som snitter med hverandre, kan man finne hvilke verdier som er felles for de to objektene vi ser på. Algoritmen for å gjøre dette kan beskrives til å følge følgende punkter:

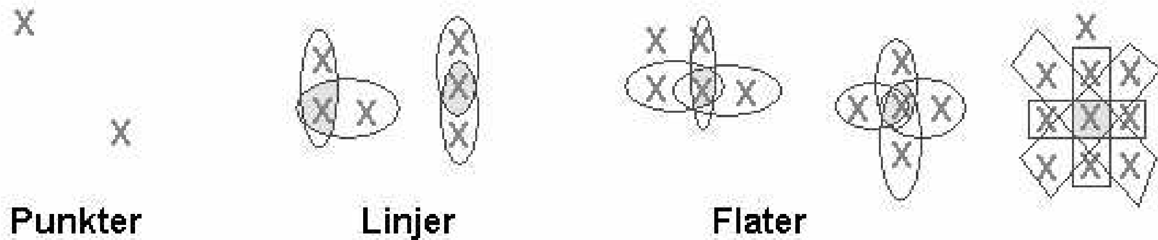
- Finn ut det intervallet vi skal sjekke. Dette gjøres ved å avgrense verdiene etter hvor de to bounding boxene skjærer hverandre.
- Så kan man dele området inn i grids, eller felter. Og for hvert felt kan vi sjekke om objektene har verdier i det bestemte feltet. Hvis begge objektene har verdier i et felt kan man ha mulighet for snitt. Det som gjøres etter at man få en slik mulighet er å sjekke verdiene mot hverandre, er de helt like har man snitt ellers ikke.

Enkelte ganger kan området avgrenset av snittet av to bounding boxer være tilnærmet lik bounding boxene, det vil si at ligger mer eller mindre oppå hverandre. I et slikt tilfelle vil det være dumt å dele hele området inn i grid og sjekke hvert enkelt felt. En løsning på et slikt problem er å dele opp objekter bounding box inn i mindre deler og sjekke snitt på disse mindre bounding boxene med det andre objektets samling bounding boxer.

9.6 Intersection resultat:

Ved hjelp av bounding box kan vi skille ut de objektene som ikke snitter med noen andre og de bounding boxene som snitter med en eller flere andre bounding boxer. Problemet videre er å kunne sjekke ut hva vi får som resultat av intersection, nemlig sjekke om det er et enkelt punkt, en samling punkter, linjer eller flater. Dette kan gjøres ved å legge inn en bufferteknikk, som i og for seg er matematisk uholdbart og kan i verste fall føre til gale svar. Man legger inn en viss minste avstand det bør være mellom to punkter for å kunne definere dem som linjer eller eventuelt flater hvis man har samling av punkter.

Sjekk av resultatet
Hvis snittet består kun av en verdi har man å gjøre med et punkt. Hvis det er flere verdier må man begynne å sjekke om det er en samling punkter eller om verdiene egentlig er en del av en større mengde med andre ord om det er linjer eller flater. Hvis punktene ligger tett inntil hverandre har man å gjøre med linjer eller flater. For å skille disse fra hverandre kan man sjekke hvordan verdier som ligger tett inntil den verdien vi ser på. Hvis det eksisterer tre eller flere punkter tett inntil det punktet man sjekker, har man å gjøre med flater. Hvis det derimot kun fins to eller en verdi tett inntil verdien, har man å gjøre med et snitt som er en linje.



Figur 53 Sjekk av punkter, linjer og flater

Ulemper med denne metoden:
Noen ganger kan det være slik at selv om to punkter ligger tett inntil hverandre, kan det hende at de ikke utgjør en linje. Dette fordi det noen ganger er slik at det fins et tomt rom mellom de to. Før vi avgjør om en samling punkter utgjør en linje eller en flate må man få sjekket om definisjonsmenge er slik at det ikke fins et tomt rom mellom to eventuelt flere punkter. Denne metoden er enkel men tidskrevende og man må gjære mye før man kommer fram til et svar. Det tar dermed mer tid og flere ressurser.

9.7 Geometrialgoritmer

Geometri algoritmer er de algoritmene som blir utviklet for å løse geometriske problemer. Tidligere ble disse løst for hånd og man benyttet matematiske formler for å finne svar på for eksempel krysningspunkter mellom to linjer eller et areal for et gitt område. Men utviklingen av datamaskiner og datamaskinenes hastighet og lagringsplass har gjort det mulig å kunne finne svar på problemer som før ikke var mulig å løse for hånd, for eksempel kan man nå finne areal av et irregulær polygon med et stort antall kanter.

Hva er en god algoritme?

Det er visse krav til hva som kreves for at en algoritme skal være god nok for det problemet vi har tenkt å løse. Det første kravet er rett og slett at algoritmen løser et gitt problem, videre følger krav til effektivitet og minst mulig lagringsplass. De sist nevnte kan få ulik vekt avhengig av hva vi er ute etter, ønsker vi en rask algoritme blir kravet om minst mulig lagringsplass prioritert ned.

Hva kan geometriske algoritmer gi svar på?

Geometriske algoritmer kan deles inn i to deler, enkle og komplekse problemer. Under følger en liste over enkle problemer med et par eksempler innefor hvert problemområde:

- representasjon av geometriske objekter - linjer, polygoner, samling av objekter og deres forhold til hverandre
- enkle beregninger – areal, volum og avstand mellom to objekter
- euklidske beregninger – tangenter, paralleller og innskrevne sirkler

- mengde teori – union, snitt og differanse

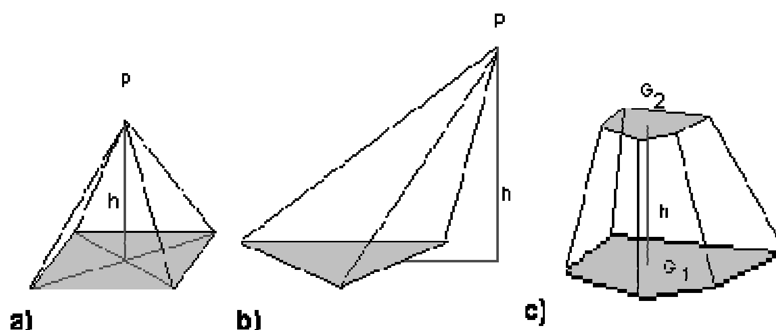
Komplekse problemer kan nå løses siden moderne datamaskiner kan utføre flere titalls operasjoner per sekund, disse kan være av disse typene:

- avansert mengde teori – for eksempel snitt mellom er stort antall objekter
- rute planlegging – for eksempel kortest vei avhengig av minst distanse, tid eller minst risiko
- konstruksjon – fra flere todimensjonale bilder til et tredimensjonalt objekt
- graf teori – minimal spanning tree

Flere og flere bruker tredimensjoner i stedet for kun x og y, når de skal modellere et fenomen fra virkeligheten. Den romlige dimensjonen brukes mest for å få bedre visualisering av virkeligheten. Men i tillegg til bedre visualisering kan disse tredimensjonale objektene brukes til en god del annet, for eksempel beregning av volum og snitt. Siden tiden sammen med x og y også betraktes å være tredimensjonalt, vil det være nyttig å sjekke om de algoritmene som brukes på et tredimensjonalt rom med x, y og z kan brukes i et rom med x, y og t som koordinater.

9.8 Algoritme 1a Areal av tredimensjonalt pyramide

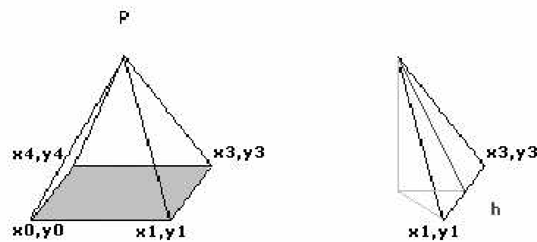
Et plan polygon er en "pyramide" som har en polygon som grunnflate.



Figur 54 Pyramider med en polygon som grunnflate[6]

Grunnflaten ligger på et plan, det vil si at alle x og y par har samme z verdi. For å regne ut at arealet av grunnflaten ser man på alle kantene til grunnflatene, det vil si x og y. Den matematiske formelen for areal av et plan polygon er $=\frac{1}{2}\sum (x_i y_{i+1} - x_{i+1} y_i)$, når i går fra 0 til N(antall kanter) - 1.
Arealet av pyramidens vegger finner man ved å finne arealet for hvert triangel og summere sammen alle triangel arealene.

[6] Bildet er hentet fra http://www.matematikk.net/per/per_oppslag.php?aid=304



$$\text{Areal av triangel} = \frac{(\sqrt{(x3-x2)^2 + (y3-y2)^2}) * h}{2}$$

Figur 55 Arealet av en triangle

9.9 Algoritme 1b Volum av tredimensjonalt plan polygon

For å finne volumet finner man først arealet av grunnflaten, deretter finner man høyden som skal stå normalt på grunnflaten. Deretter blir volum regnet ut ved hjelp av formelen

V = ⅓* grunnflatens areal * høyde

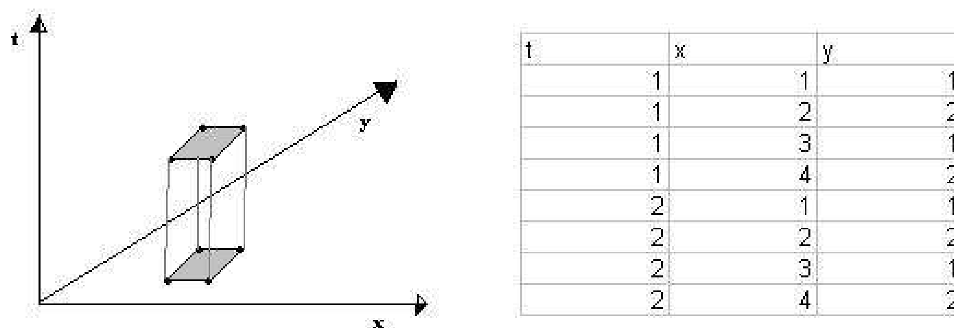
Ikke alle pyramider er spisse på toppen. Hvis vi har en pyramide med stump, det vil si en polygon på toppen blir utregningen av volumet litt annerledes. Nå må både topp- og bunn polygon tas med i utregningen og formelen blir slik:

V = (h/3) (grunnflate(bunnpolygon) + √ grunnflate(bunnpolygon) * grunnplate(toppolygon) + grunnplate(toppolygon))

9.10 Kombinasjon av x, y og t og Algoritme 1

Når et todimensjonalt objekt med x og y verdier, utvikler seg over tid vil det med hensyn på tidsdimensjonen sees på som tredimensjonalt objekt. Siden det er mulig å finne volumet for et tredimensjonalt objekt med x, y og z verdien er det kanskje mulig å finne arealet av et tredimensjonalt objekt med x, y og t verdier. Problemet vi vil støtte på er at z verdien måles med samme benevning som x og y mens t verdien for en benevning av typen tid som kan være alt fra millisekund til time eller døgn. Det samme vil vi støtte på når vi regner ut volumet av et objekt, men man kan jo gi at svaret med en benevning med m^2sek eller m^2time.
I følge Dan Sunday [Sunday] kan man ved hjelp av projeksjon[7], regne ut arealet av en pyramide i todimensjoner i stedet for å ta hensyn til den tredje dimensjonen når man regner ut høyden i hver triangel. Hvis jeg velger å benytte meg av denne Dan Sundays [Sunday]påstand om projeksjon på et tredimensjonalt objekt med x, y og t verdier, vil vi også fint klare å regne ut arealet av hvilken som helst tredimensjonalt objekt ved å forskyve objektet langs to dimensjoner for eksempel x og y som begge er i meter, slik at benevningen blir geometrisk korrekt. Dette kan vises ved hjelp av et enkelt eksempel, la oss anta at vi har et objekt som i vokser i løpet av en tid, se figur 56.

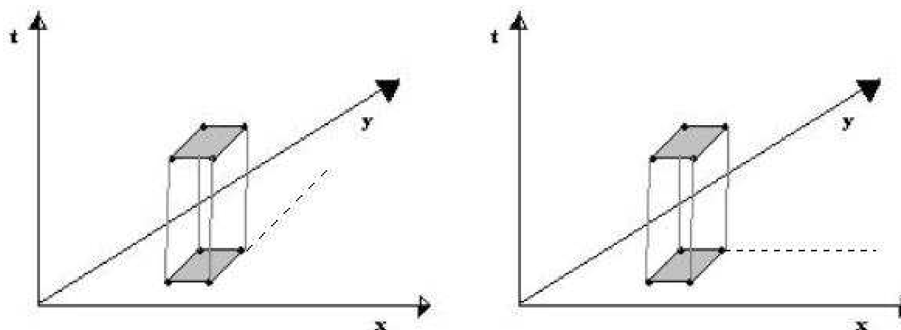
[7] avbildning av en flate på et plan. Flaten strekkes ut slik at man utfører målinger ved hjelp av benevningene langs den eller de dimensjonen/dimensjoner man ønsker.



t	x	y
1	1	1
1	2	2
1	3	1
1	4	2
2	1	1
2	2	2
2	3	1
2	4	2

Figur 56 Eksempel beregning av areal

Arealet av de skraverte områdene(se figur 56) er enkelt å beregne og vi får at arealet blir 4m². Det vi har igjen er fire flater hvor to er slik at arealet blir (2m * 1 sek) og de to resterende har 1m*1 sek. Her støtter vi på problemet med benevning av areal, og kan prøve oss med projeksjon av de fire flatene langs x og y planet.



Figur 57 Projeksjon av t

Ut ifra figuren får vi at 1 sek blir tilnærmet lik 8 meter, noe som gir et areal lik 2*(2*8) + 2(1*8) + 4 = 46m² noe som er ganske stort i forhold til det vi hadde tidligere det vil si 4m²2 + 2(2m * 1sek) + 2(1 m*1 sek) = 10 m²sek.
Problemet med denne metoden er at svaret vi får, kan ikke sies å være rett. Uten projeksjon får vi utregnet og svaret er riktig men benevningen er ikke slik man er vant til. Med projeksjon kan man prøve å få til en benevning som er riktig i henhold til geometriske utrengninger, men svaret kan være feil siden man ikke kan si at 1 sekund tilsvarer 8 meter i virkeligheten. Dermed er det bedre å kunne ta med tidsaksessen være beste løsning siden dette ikke gir noen feil i svaret men gir kun ”feil” med hensyn på benevningen, siden m²sek ikke er noen form for geometrisk måleenhet.

9.11 Interpolasjon

Ved vektormodellen blir objekter representert med en sekvens av enkeltstående x og y koordinater, mens raster modellen representerer objekter slik at de blir representert med et sett av celler ordnet i kolonner og rader. Interpolering av et objekt representert ved hjelp av vektormodellen kan betraktes å bli gjort på måtene beskrevet i avsnittene 5.1.1 Diskrete interpolasjonsmetoder og 5.1.1

Kontinuerlige interpolasjonsmetoder. Interpolering ved en rastermodell blir noe annerledes og kan gjøres på tre måter:

- interpolering skjer ved at en celle for en verdi ut ifra verdien til nærmeste eksisterende celle
- interpolere en celles verdi ved å se på de fire nærmeste cellenes verdier, denne metoden er bedre enn metode 1
- en ukjent celle for en verdi ved at man finne gjennomsnittet for 16 nærmeste celler

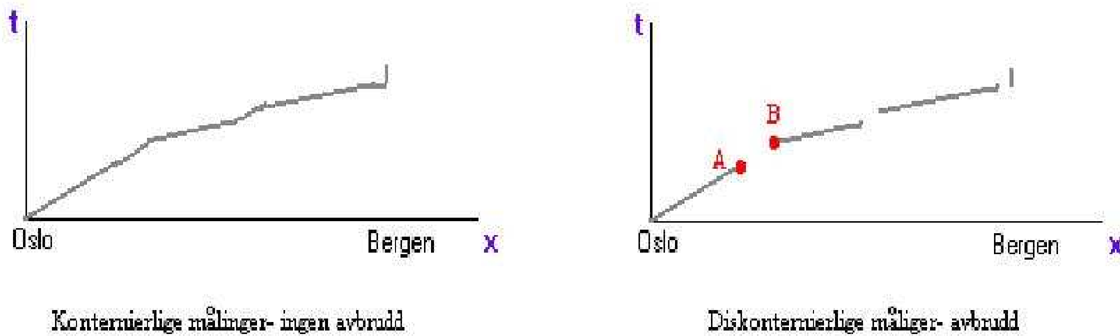
Videre kan man vurdere om hvilke modell som bør velges avhengig av behovet for interpolasjon, men før man har sjekket de forskjellige resultatene kan man ikke på forhånd påstå at den ene representasjonsmåten er bedre enn den andre. Avhengig av hvilke typer fenomen man har å gjøre med velger man vektor eller rasterrepresentasjon, har man en klart avgrenset fenomen i virkeligheten vil det være bedre å representere fenomenet ved hjelp av vektorrepresentasjon og deretter bruker de ulike teknikkene for å beregne ikke eksisterende punkter.

9.12 Interpolasjon og vektor

Siden vi må diskretisere og avrunde når vi modellere en del av virkeligheten blir vi ved enkelte situasjoner nødt til å finne de punkter som ikke er blitt satt inn i datamaskinen. Man kan estimere fram til ikke eksisterende punkter mellom to eller flere eksisterende punkter ved hjelp av interpolasjon. Skal vi derimot estimere utenfor de oppmålte punktenes rekkevidde kalles dette ekstrapolasjon og dette blir blant annet brukt ved målinger frem i tiden. Da ser man på hvordan utviklingen har vært til nå og beregner videre verdier. Dette kan også gjøres ved en romlig virkelighet ved at man for hver akse finner en mulig endringsfunksjon.
Veldig enkelt kan man interpolere slik at man mellom to målte punkter legger inn en rettlinjet funksjon som finner de ikke eksisterende punktene og setter denne kurven sammen med resten av punktene og deres lineære kurver. Dette vil som ved bil eksempelet nevnt i begynnelsen av dette kapitelet føre til feil men en mulig løsning på dette kan være at man går bort fra funksjonen om rett linje og heller velger andre måter å interpolere på. Videre kan man skille mellom metoder som tar seg av kontinuerlig og diskrete målinger.

9.13 Kontinuerlige og diskontinuerlige målinger

Ved beregning av data kan det oppstå en del problemer, kanskje har man klart å måle i en kontinuerlig periode men så dukker det opp en tid hvor man kanskje ikke måler like mye som før. På grunnlag av dette vil vi ved modellering av manglende verdier over en tid få avbrudd eller hull. Man har data ganske tett mot hverandre og plutselig kan man få en tid med lite eller ingen data som forårsaker et plutselig hopp i verdiene ved de fleste tilfeller. Ser vi på toglinjen fra Oslo til Bergen og måler diskonternierlig over den tiden et tog bruker fra Oslo til Bergen, vil vi ved modellering få plutselige hopp fra et tidspunkt med en x verdi til et annet tidspunkt med en annen x verdi. Tegner man dette inn i en graf (se figur 58) vil det ut som toget har kommet fram til et sted (B) uten å måtte kjøre strekningen fra stedet hvor avbruddet startet (A). Noe som er imot virkelighetens regler.



Figur 58 konternierlige og diskonternierlige målinger

På grunn av oppløsningsevne og diskretiseringsvalg vil det forekomme muligheter for avbrudd ved modellering av fenomen og dets verdier. Siden vi vet at et tog som må kjøre strekningen fra A til B og videre til C, ikke kan komme til C direkte fra A, må dette kunne håndteres ved modellering. Ved å legge inn regler for hva som er tillat og hva som ikke er tillatt er det mulig å kunne håndtere og løse dette problemet. Skranker blir i modelleringen brukt til å legge restriksjoner for hvilke data som skal aksepteres av informasjonssystemet. Skranker står egentlig for de regler som gjelder i virkelighet, dette kan være regler fra naturlover til foretningsregler for en bedrift.
En topologisk skranke skal for eksempel kunne utrykke at kurvene i grafen i figur 58 må til sammen dekke hele toglinjen fra Oslo til Bergen.

9.14 Kontinuerlige interpolasjonsmetoder

I følge Bernd Etzelmüller [Etzelmüller, 1997] kan man skille mellom to typer interpolasjon ved kontinuerlige interpolasjonsmetoder, global og lokalt interpolasjon. Global interpolasjon blir utført på objekter som endrer seg kontinuerlig over det hele mens lokale interpolasjonsmetoder blir brukt når man har å gjøre med objekter som endrer seg kun i bestemte intervaller.

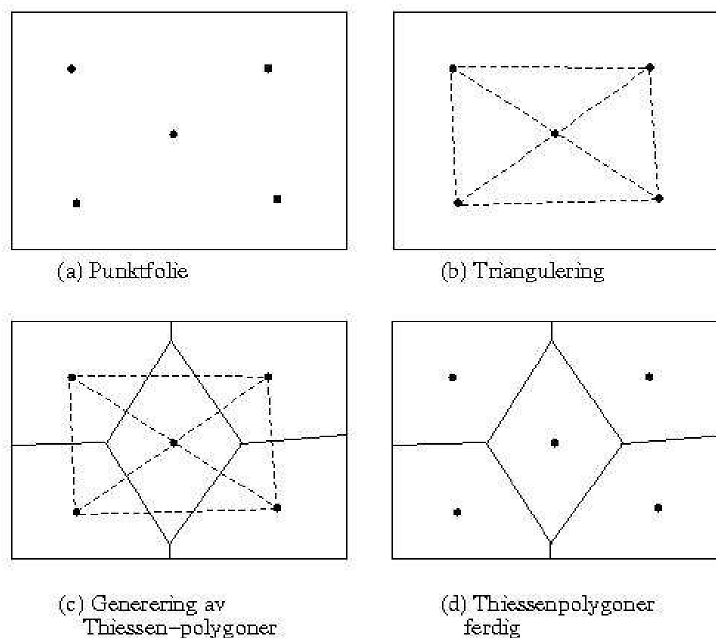
Globale interpolasjonsmetoder
Regresjon er en enkel måte som kan benyttes for å finne ikke eksisterende verdier, dette kan gjøres ved å beregne første ordens polynomer og så velge høyre grads polynomer hvis det er behov. Men her må vi få sjekke om man har å gjøre med et endimensjonalt objekt i tredimensjoner eller om det er snakk om en flate i et tredimensjonaltrom.

Lokale interpolasjonsmetoder
Akkurat som man kunne interpolere globalt ved hjelp av regresjon kan man benytte samme teknikk når det gjelder kun noen få punkter. Splines, glidende gjennomsnitt og kriging er kjente teknikker som blir brukt til å finne en interpolerende kurve for den datamengden man har å gjøre med. Splines er en matematisk metode som går ut på få en krum linje gjennom en mengde med punkter, disse metodene kan løses slik at de gir eksakt kurve eller en kurve som passer nesten.

9.15 Diskrete interpolasjonsmetoder

Interpolasjon av 1D i 2D(og 3D) rom:
Den enkleste metoden for å finne ikke eksisterende punkter er å følge de matematiske formeler for kurver, man kan enten bruke formelen y - y1 = a (x - x1) eller velge å bruke andre- eller tredjegradsfunksjoner. Men dette vil gi en samling punkter mellom to eksisterende punkter men vi er kun interessert i et punkt på denne kurven.

Interpolasjon av 2D i 3D rom:
Bernd Etzelmüller [Etzelmüller, 1997] viser til at man kan finne ikke eksisterende punkter ved å bruke teorien om Thiessen[8] polygoner. Først trekker man rette linjer mellom de målepunktene man har, så finner man normalen på hver linje og kobler normalene sammen. Slik har man mulighetene til å finne ut hvor omtrent punktet vil være, og man vil kun få et punkt i stedet for en kontinuerlig kurve som ved bruk av matematiske kurvefunksjoner.



Figur 59 diskret interpolasjon, hentet fra [Etzelmüller, 1997]

Videre kan man vurdere hvordan denne metoden vil bli seende ut ved forksjellige dimensjoner, i figur 59 betrakter vi et system som kun består av todimensjoner men hva med når man har å gjøre med tredimensjoner. Selv om dimensjonene øker vil man i hovedsak benytte samme teorem og finne punkter ved hjelp av Thiessen polygoner.

9.16 Ekstrapolasjon

La oss anta at vi kun ha et objekts verdier til et tidspunkt t, etter t har vi ingen målbare verdier men vi ønsker å vite hva objektets verdier vil være i fremtiden. Dette kan gjøres ved å estimere fremover ved hjelp av tidligere endringer og forsette å bruke den interpolasjonsmetodikken man har brukt på objektets endringer før t. Men dette vil kun gi et estimat av verdier basert på trendsanalyser, og vil bli betegnet som ekstrapolasjon og ikke interpolasjon. Hvis vi for eksempel vet at isen på en isbre smelter hver sommer frem til i dag kan man på grunnlag av dette sette opp en hypotese om at isbreen vil smelte neste sommer også.

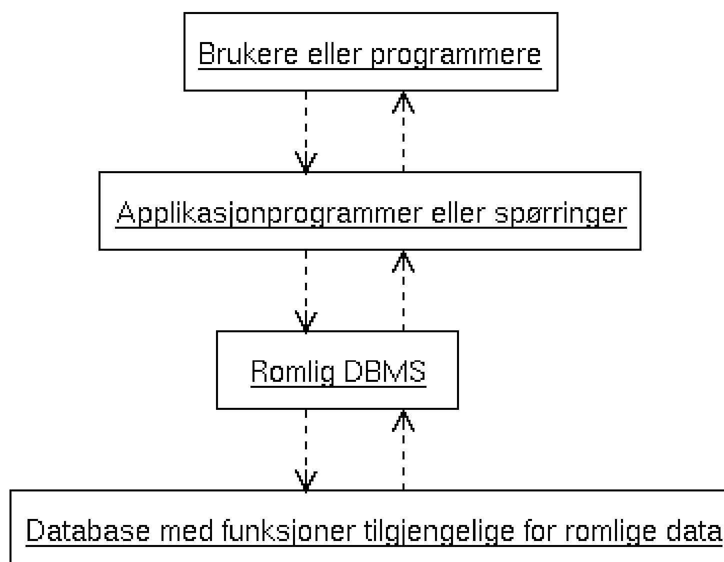
---
[8] Thiessen polygoner kan også kalles voronoi diagrammer,

Et problem vi kan støtte på ved ekstrapolasjon av fremtidlige verdier er når et objekt ikke følger en trend, et eksempel på slikt er prisen for en vare. La oss anta at en kjøpmann selv kan bestemme prisen for de varene han selger, dermed vil ikke valget av pris kun bli bestemt ut fra utviklingen av prisen de siste par årene men av kjøpmannens egen vilje. Det er slike eksempler som kan være problematiske å løse. Plutselige endringer i fremtiden kan føre til at ekstrapolasjons kurve blir helt feil.
Ekstrapolasjon med objektet i en temporal og romlig virkelighet kan skje fremover i tid, med en romlig virkelighet vil endringer av objekter ikke forekomme på grunnlag av tid men vi kan bruke samme metodikk og fine resterende punkter ved for eksempel objekter hvor vi kun har noe få verdier. La oss anta at vi kun har målte verdier for et objekts grunnflate, videre vet vi at objektets tilhørende fenomen er en kjegle og har en viss areal, på grunnlag av følgende verdier kan vi ekstrapolere og finne kurver slik at punkter vi finner er slik at objektets areal blir lik fenomenets målte areal.
Men som ved en temporal virkelighet vet vi ikke med sikkerhet om det vi har estimert er riktig, dette fordi de verdier som blir funnet blir ekstrapolert ikke målt. Både romlig og temporalt rom kan betraktes å ha like fordeler og ulemper og ekstrapolasjonsmetodene kan brukes på samme måte uavhengig av hvilken type rom man har å gjøre med.

10 Modellsyn i SQL92 med Oracle Spatial

Tradisjonelt, database håndetringssystemer er hovedsakelig delt inn i to kategorier; relasonelle, objekt orienterte. Objekt relasjonelle systemer kombinerer det beste fra relasjonelle og objekt orienterte systemer. Objekt relasjonelle systemer gir mulighet til definisjon, lagring, henting og manipulering av bruker definerte data typer i databasen gjennom bruk av bruker definerte funksjoner og indeks metoder. På grunn av disse mulighetene en ORDBMS kan nå håndtere romlig informasjon representert ved å bruke en romlig objekt datatype, og aksesseres eller manipuleres ved å bruke romlige indeksmetoder og funksjoner. I et romlig DBMS er rommet bare et attributt på lik linje med andre attributter representert i databasen, og brukere kan bruke det på linje med alle andre kriterier når de leter eller browser i databasen.

Slik ser arkitekturen ut for en romlig DBMS:

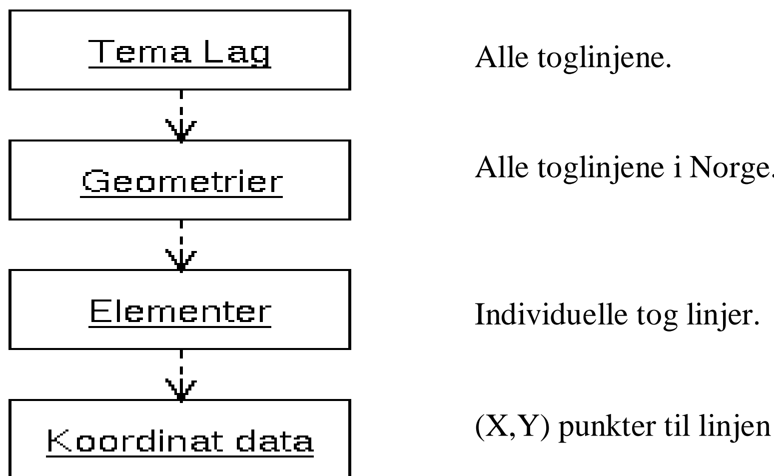


Database-tjeneren inneholder eksempelvis funksjonalitet for å finne krysspunkt mellom to rette linjer. Vi trenger bare å definere datamodellen. Oracle Spatial bruker en objektrelasjonell modell, legger til en innebygget datatype SDO_GEOM med funksjoner .

10.1 Oracle Spatial

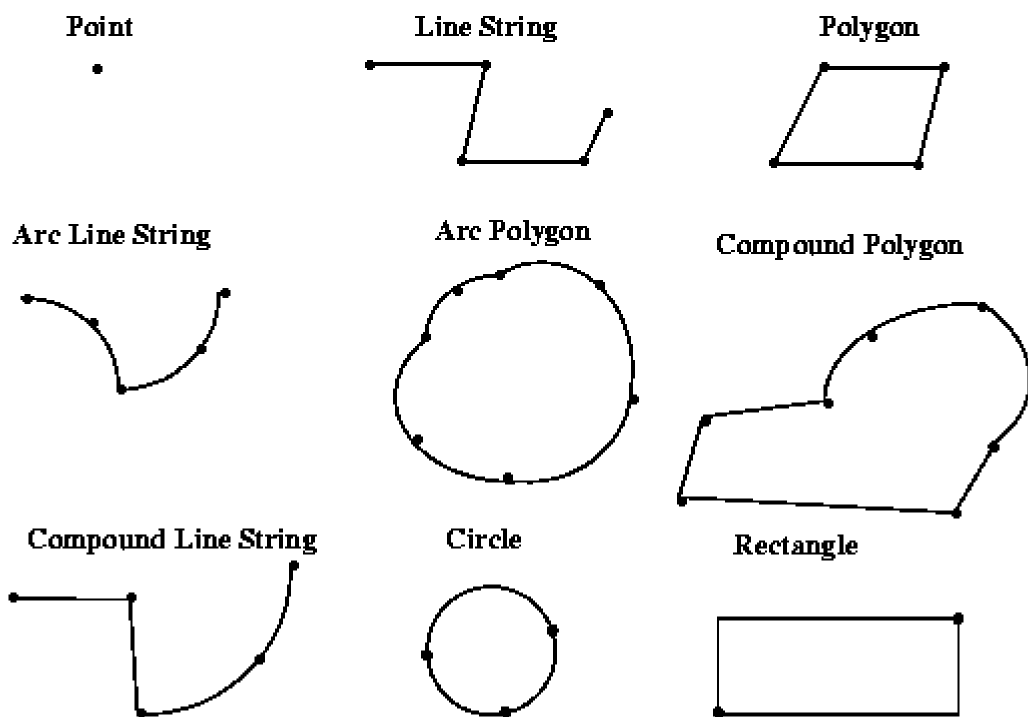
Oracle Spatial tilbyr en fullstendig åpen arkitektur for håndtering av romlige data i et databasehåndteringssystem. Funksjonaliteten i Oracle Spatial er fullstendig integrert i database-tjeneren. Brukere definerer og manipulerer romlige data ved hjelp av SQL.

Her er den romlige hierarkiske strukturen i Oracle Spatial (på tog eksemplet):



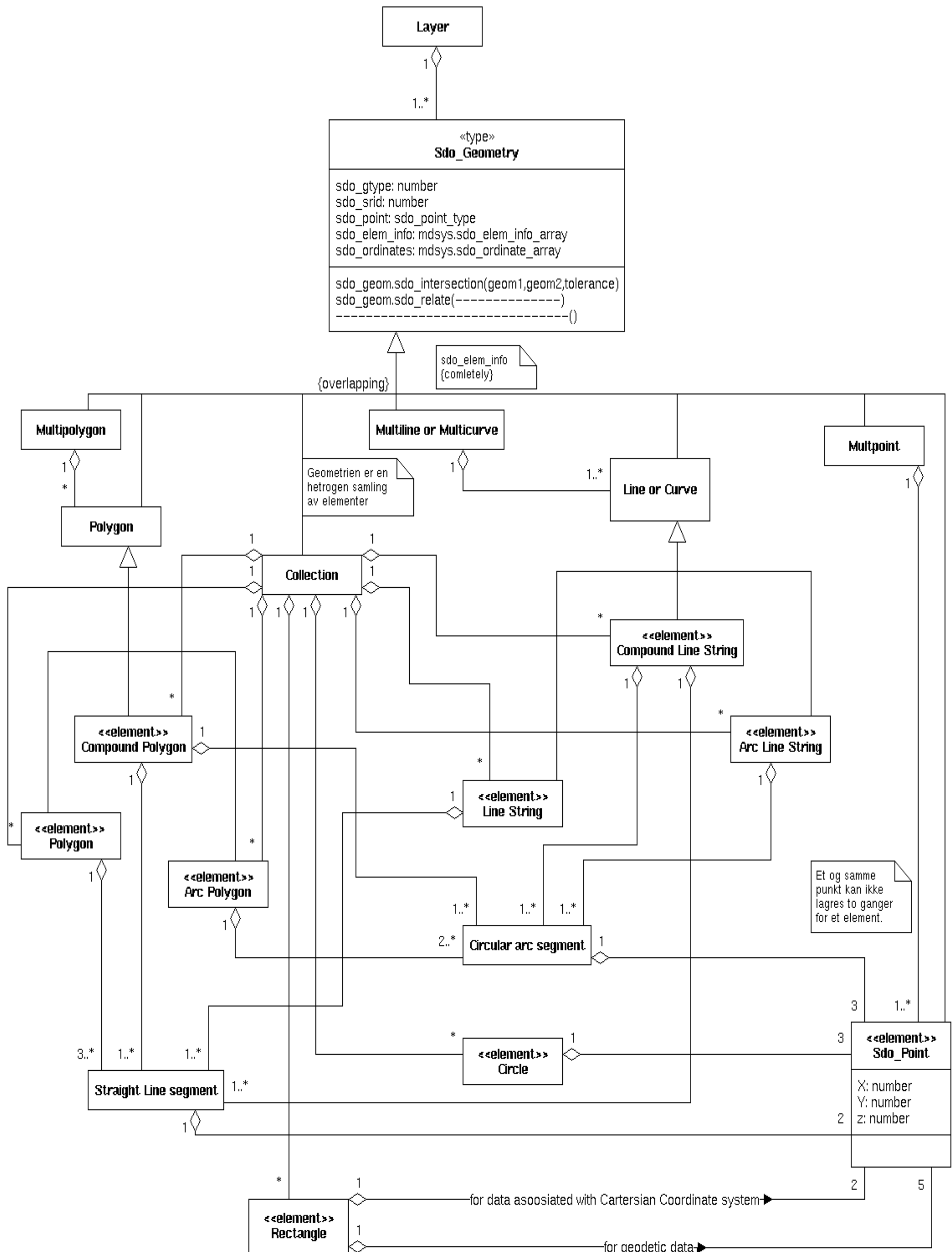
Figur 60: Hierarkiske strukturen i Oracle Spatial på det spesielle eksemplet.

Nedenfor kan vi se eksempler på mulige former som kan representeres ved bruk av SDO_GEOMETRY.
Arclinestring og Arcpolygon er elementer som har hjørner som er knyttet til hverandre med sirkulære kanter. Et Compound element har hjørner som er knyttet til hverandre med en blanding av rette og sirkulære linjestykker.



Figur 61: Mulige former som kan representeres i Oracle Spatial

10.1.1 Oracle Spatial arkitektur



Figur 62: Modell arkitektur i Oracle Spatial

Forklaring til figuren ovenfor:

- Et ***temalag*** er definert som en kolonne og tabell, par i databasen, og er en heterogen samling av geometrier som deler det samme attributtsettet. Temalag korresponderer til en tabell eller en mengde av tabeller, mens en geometri er en instans av typen MDSYS.SDO_GEOMETRY og er lagret i en bestemt rad eller kolonne i en tabell.

- En ***geometri*** er en ordnet sekvens av hjørner som er knyttet sammen enten med rette linjestykker eller sirkulære kanter. Den kan bestå av et enkelt element eller en homogen eller heterogen samling av primitive typer. En geometri kan kun være en av disse typene:
  a) Point
  b) Multipoint
  c) Line eller Curve
  d) Multiline eller Multicurve
  e) Polygon
  f) Multipolygon
  g) Collection

- Et ***element*** er den grunnleggende byggeklossen for en geometri. For eksempel kan elementer representere buss-stopper (punkter), veier (linjer), eller grenselinjer for byer (polygoner). I tilfelle av polygoner med huller (slik som en sjø med en øy i) **er** den ytre og den indre ringen betraktet som to forskjellige elementer som til sammen utgjør en multipolygon. Oracle Spatial støtter disse elementtypene:
  a) Point
  b) Line String
  c) Arc Line String
  d) Compound Line String
  e) Polygon
  f) Arc Polygon
  g) Compound Polygon
  h) Circle
  i) Rectangle

I modellen av arkitekturen i Oracle Spatial ovenfor er det definert to elementer: linjestykke og sirkulærkant, som egentlig ikke er definert som særkilte elemeter i Oracle Spatial, som andre elementer.. Disse to (falske) elementene har ingen egen representasjon i Oracle Spatial, de er med i modellen for å lettere kunne illustrere at for eksempel en sammensatt polygon kan bestå av to enkle linjestykker (som ikke nødvendigvis henger sammen) og en sirkulær kant. Tankemessig består alle elementer(utenom punkt) av enten sirkulærekanter/linjestykker eller en blanding av begge typene.
Alt er representert med punkter i databasen. Hvis en linje består av et rett linjestykke, så lagres det to punkter i databasen, start- og sluttpunktet. Hvis derimot en linje består av to rette linjestykker som henger sammen, så lagres det tre punkter i databasen og ikke fire, som man kunne tro, fordi et punkt (i Oracle Spatial) ikke kan lagres to ganger for å representere slutt på ett linjestykke/sirkulærkant og start på neste linjestykke/sirkulærkant.

For å representere en lukket polygon må man derimot lagre både slutt- og startpunktet for å lukke polygonen, det vil si at sluttpunktet på siste linjestykke/sirkulærkant er det samme som startpunktet for første linjestykke/sirkulærkant.

10.2 *Representasjon av romlige objekter*

Typen MDSYS.SDO_GEOMETRY er en beholder for å lagre punkter, linjer, polygoner eller homogene eller heterogene samlinger av disse elementene.

Nå skal vi se hvordan vi kan finne krysspunkt mellom to tog ved hjelp av Oracle spatial. Definisjon av tog linjene er fortsatt det samme som vi hadde tidligere – se Operasjoner og abstrakte datatyper. Altså ser vi på toglinjen som en rett linje.

Data modellen for tog objektene ser nå slik ut i Oracle spatial:
SDO_TOG

TOGNR	REKKE	RUTE
tognr	linje_nr	:mdsys.sdo_geometry sdo_gtype sdo_srid sdo_point sdo_elem_info sdo_ordinates sdo_geom.sdo_intersection(geom1,geom2,tolerance) sdo_geom.sdo_relate(-------------------) --------------------------------

:sdo_tog

TOGNR	REKKE	RUTE
1_609	1	:mdsys.sdo_geometry 2002, null, null, mdsys.sdo_elem_info_array(1,2,1), mdsys.sdo_ordinates_array(0747, 89.57,1334,471.25) sdo_geom.sdo_intersection(geom1,geom2,tolerance)

Ovenfor har vi en objektrelasjonell tabell med navn SDO_TOG som har tre attributter: TOGNR, REKKE og RUTE. REKKE definerer rekkefølgen av eventuelt linjestykker. Hvis vi har brudd på en linje (som ikke er tilfelle her) så kan vi ved hjelp av REKKE-nummeret finne ut rekkefølgen på linjestykkere. Attributtet RUTE er av den abstrakte data typen MDSYS.SDO_GEOMETRY.

Her er en kort definisjon av attributtene i MDSYS.SDO_GEOMETRY:

- SDO_GTYPE: Identifiserer geometritypen. Dette er en firesifret verdi i formatet *dltt (dltt =2002 for tognr 1_609):*
  - *d* identifiserer antall dimensjoner 2,3 eller 4. (1_609 har: d =2)
  - *l* identifiserer lineær henvisende målings dimensjon, eller for å akseptere Spatial sin default av den siste dimensjonen som måling for en LRS geometri, spesifiser 0. (1_609 har: l=0)
  - *tt* identifiserer geometritypen 00 til 07. (1_609 har: tt=02)
    Hvis tt=02: Geometrien er en linje som kan inneholde rette eller sirkulære linjestykker, eller begge.

    (For å se en oversikt verdien av gyldige geometrityper, se: Oracle spatial User Guide and Reference kapittel 2 s.6)

- SDO_SRID: Kan brukes til å identifisere et koordinatsystem som skal assosieres med geometrien. SDO_SRID settes til null hvis intet koordinatsystem skal assosieres med geometrien.
- SDO_POINT: Er definert ved å bruke SDO_POINT_TYPE objekttypen som har attributtene X, Y og Z, alle av typen *number*. Hvis både SDO_ELEM_INFO og SDO_ORDINATES array er null, og SDO_POINT attributtet ikke er null, da blir X- og Y-verdiene betraktet som koordinatene for et punkt element. I andre tilfeller blir SDO_POINT attributtet ignorert av Spatial.

SDO_ELEM_INFO:
Er definert ved bruk av et variabel-lengde array av tall. Dette attributtet forteller hvordan man skal tolke koordinatene lagret i SDO_ORDINATES-attributtet.

- SDO_STARTING_OFFSET: Indikerer plassen i SDO_ORDINATES arrayet der den første koordinaten ligger for dette elementet.(plassnr begynner fra 1 ikke 0) Da vil den første koordinaten for det første elementet lagres i SDO_GEOMETRY.SDO_ORDINATES(1). Hvis det hadde vært et element til, ville dette elementets først koordinat vært på den første ledige plassen i SDO_ORDINATES-arrayet, etter å ha lagt inn alle koordinatene til det første elementet.

- SDO_ETYPE: Indikerer typen av elementet.
  - Verdiene 1, 2, 1003, og 2003 blir betraktet som enkle elementer.
    * 1 : punkt element
    * 2 : linje element
    * 1003: ytre polygon ring (må spesifiseres i rekkefølge mot klokken)
    * 2003: indre polygon ring (må spesifiseres i rekkefølge med klokken).

  - Verdiene 4, 1005 og 2005 blir betraktet som sammensatte elementer.

* 4 : sammensatt linje
* 1005: ytre polygon ring (må spesifiseres i rekkefølge mot klokken)
* 2005: indre polygon ring (må spesifiseres i rekkefølge med klokken).

- SDO_INTERPRETATION: Har to betydninger, avhengig av om SDO_ETYPE er et sammensatt element eller ikke. Hvis SDO_ETYPE er et sammensatt element, spesifiserer dette feltet hvor mange av de etterfølgende trippel-verdiene som er en del av dette elementet. Hvis elementet ikke er sammensatt, bestemmer *interpretation*-attributtet hvordan sekvensen av koordinater for dette elementet skal tolkes (linje, polygon o.l). For sammensatte elementer et sett av n-tripler brukt for å beskrive elementet. Siste punktet for ett delelement er første punkt for det neste delelementet.

Her følger et eksempel for tognr 1_609:

```
mdsys.sdo_elem_info_array(1,2,1)
```

Det første tallet 1 betyr at koordinatene for dette elementet starter fra plass 1 i SDO_ORDINATES. Den etterfulgte kombinasjonen 2,1 betyr at dette elementet er en enkel rett linje (hjørnene av linjen kan også være knyttet til andre rette linjestykker).

SDO_ORDINATES:

Er lagret som variabel-lengde array (1048576) av typen *number* som lagrer koordinatverdiene som utgjør grensen av et romlig objekt.
I eksemplet nedenfor kan vi se at tognr 1_609 består av to punkter: Et startpunkt og et sluttpunkt for linjen; disse punktene lagres på denne måten (t1,x1,t2,x2) :

```
mdsys.sdo_ordinate_array(0747,89.57,1334,471.25)
```

10.2.1 Insert metadata informasjon

Geometrien metadata som beskriver dimensjoner, nedre og øvre grenser og toleransen i hver dimensjon, er lagret i en global tabell som er eiet av MDSYS. Denne tabellen skal programmerere aldri oppdatere direkte. Toleranse blir brukt for å assosiere et presisjonsnivå for romlige data, for ikke_geodetiske data kan toleranseverdien være opptil 1. Hver bruker av Oracle Spatial har et view USER_SDO_GEOM_METADATA tilgjengelig i skjemaet sitt. For hver tabellkolonne som er av typen MDSYS.SDO_GEOMETRY må det være en forekomst i viewet USER_SDO_GEOM_METADATA. Viewet er definert på denne måten:

```
View USER_SDO_GEOM_METADATA(
  TABLE_NAME   VARCHAR2(32),
  COLUMN_NAME  VARCHAR2(32),
  DIMINFO      MDSYS.SDO_DIM_ARRAY,
  SRID         NUMBER
  );
```

hvor SDO_DIM_ARRAY er et VARRAY(4) av MDSYS.SDO_DIM_ELEMENT, og SDO_DIM_ELEMENT er definert som

```
Create Type SDO_DIM_ELEMENT as OBJECT (
SDO_DIMNAME VARCHAR2(64),
SDO_LB NUMBER,
SDO_UB NUMBER,
SDO_TOLERANCE NUMBER);
```

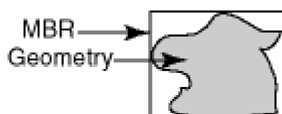
For eksempel må følgende `insert`-operasjon i SDO_TOG. RUTE-kolonnen nedenfor gjøres før oppretting av noen som helst type av romlig indeks eller bruk av romlige funksjoner eller operatorer.

```
insert into user_sdo_geom_metadata
     values('SDO_TOG',
            'rute',
            mdsys.sdo_dim_array(
            mdsys.sdo_dim_element('t',0,2400,0.005),
            mdsys.sdo_dim_element('x',0,471.25,0.005)),
            null
            );
```

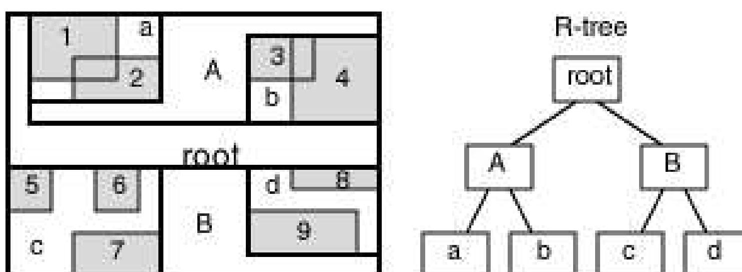
10.2.2 Romlig indeksering

En romlig indeks, som lik en hvilken som helst annet indeks, tilbyr en mekanisme for å begrense antall søk, men i dette tilfellet basert på romlige kriterier som snitt og beholder.
En romlig indeks er påkrevd for å effektivt kunne prosessere spørringer, slik som å finne objekter inne i et indeksert data område, som overlapper et gitt punkt eller interesse område.
Oracle tilbyr en lineær quad-tre-basert indekseringsmåte og en R-tre-basert indekserings måte for å indeksere romlig data.

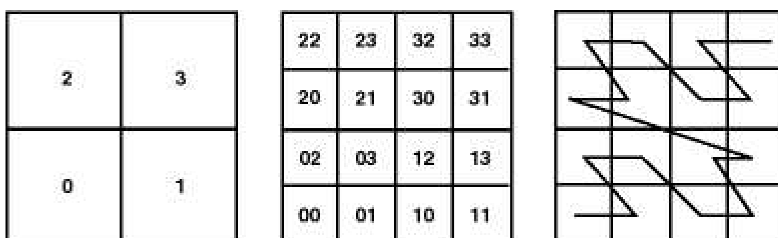
- **R-Tree Indeksering** : En Spatial R-tre-indeksering kan indeksere romlige data med opptil 4 dimensjoner. En R-tre-indeks approksimerer hver geometri med et enkelt rektangel som er det minimale rektangelet for å innelukke geometrien (kalles "minimum bounding rectangle" eller MBR)



For et lag av geometrier består en R-tre-indeks av en hierarkisk indeks på MBRer av geometriene i et lag, på følgende måte:



- **Quad-Tree Indeksering :** Et lineært quad-tre avbilder geometriske objekter på en mengde av nummererte fliser. En flis i 2D-rommet er en boks som har kanter ortogonalt til de to aksene. Koordinatrommet der alle geometriene ligger, nedbrytes på en regulær hierarkisk måte. Området av koordinatene (koordinatrommet) er vist som et rektangel. På første nivå av nedbrytingen er dette rektanglet delt i to i hver av koordinatretning, de fire delflisene blir kalt quads. Hver flis som krysser geometrien nedbrytes igjen til fire fliser. Denne prosessen fortsetter til et eller annet termmineringskriterium, slik som størrelsen av fliser eller maksimum antall fliser per geometri, er møtt.
  Spatial kan bruke fast størrelse på fliser (SDO_LEVEL), variabel størrelse på fliser (SDO_NUMTILES) eller en blanding av begge til å dekke et geometri. Hybrid indeksering inneholder en blanding av begge type fliser. Fixed indeksering bruker bare fast størrelse på alle fliser.



SDO_LEVEL -verdien bestemmer størrelsen av faste fliser brukt for å dekke den indekserte geometrien fullstendig. Man kan bruke funksjonen SDO_TUNE.ESTIMATE_TILING_LEVEL for å bestemme en tilfeldig start SDO_LEVEL verdi, også sammenligne utførelsen ved å bruke høyere eller lavere SDO_LEVEL verdier, og tilsutt bruke den verdien som gir best utføring.
Her er for eksempel denne funksjonen brukt for å estimere hvilken begynner verdi SDO_LEVEL skal ha:

```
select sdo_tune.Estimate_Tiling_level
      ('SDO_TOG', 'rute', 10000,'avg_gid_extent')
from dual;
```

```
SDO_TUNE.ESTIMATE_TILING_LEVEL('SDO_TOG','RUTE',10000,'AVG_GID_EXTENT')
-----------------------------------------------------------------------
                                                                      0
```

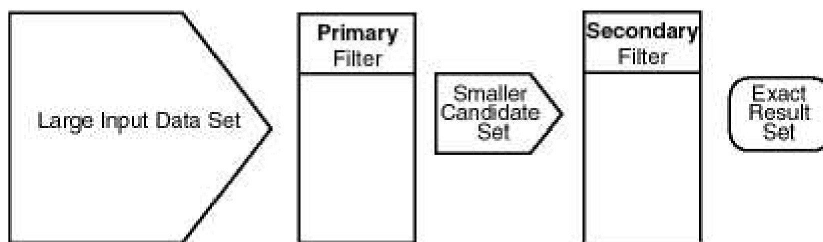
Her ser vi at den anbefalte verdien for SDO_LEVEL er 0, etter å ha gjort et par tester med forskjellige SDO_LEVEL verdier fant vi ut at svarpresisjonen i dette tilfellet var uavhengig av SDO_LEVEL verdien. Derfor brukte vi verdien 0. Her er et eksempel på å lage en Fix indeks:

```
create index tog_fix on SDO_TOG(rute) indextype is
mdsys.spatial_index
parameters ('sdo_level=0');
```

Denne indeksen representerer objektene i en rasterrepresentasjon (med fast størrelse på cellene i koordinatsystemet).

10.3 Spørringer mot romlige objekter

Spatial bruker en to-lags spørremodell til å løse romlige spørringer og å gjøre spatial join. Termen to-lags indikerer at to forskjellige operasjoner blir utført i rekkefølge for å løse spørringer.
De to operasjonene refereres til som primærfilter- og sekundærfilter-operasjoner.



- **Primærfilteret** tillater en rask seleksjon av kandidattupler som skal sendes videre til sekundærfilteret (betraktet som et lavkostnads filter).
- **Sekundærfilteret** bruker eksakte beregningsgeometrier på resultatsettet fra primærfilteret. Disse eksakte beregningene gir et eksakt svar på spørringen (mer kostbart filter).

I den abstrakte datatypen MDSYS.SDO_GEOMETRY er det en god del forhåndsdefinerte geometrifunksjoner for den objektrelasjonelle modellen. La oss se på *intersection*-funksjonen for å se hvordan disse funksjonene kan brukes. Funksjonen har dette formatet:

```
SDO_GEOM.SDO_INTERSECTION(
   geom1 IN MDSYS.SDO_GEOMETRY,
   geom2 IN MDSYS.SDO_GEOMETRY,
   tolerance IN NUMBER
) RETURN MDSYS.SDO_GEOMETRY;
```

Vi har brukt denne funksjonen for å finne krysningspunktet mellom tognr 1_62 og tognr 1_609. Slik ser spørringen ut:

```
select sdo_geom.sdo_intersection(t1.rute,t2.rute,0.005)
from SDO_TOG t1, SDO_TOG t2
where t1.tognr='1_609' t1.rekke=1 and
      t2.tognr='1_62'and t2.rekke=1;
```

Spørringen ovenfor ga dette resultatet:

```
Resultat:
**********
```

```
SDO_GEOM.SDO_INTERSECTION(T1.RUTE,T2.RUTE,0.005)(SDO_GTYPE, SDO_SRID,
SDO_POINT(
--------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1037.15013, 278.231844))
```

Resultatet er et punktobjekt som har **t:** `1037.15013` og **x:** `278.231844`.

Selv om vi brukte en egendefinert algoritme på de tidligere plattformene, og her kalte på den ferdig implementert geometrifunksjonen for å finne krysspunktet, har vi fått nøyaktig det samme svaret. Grunnen er at både Oracle Spatial og vi går ut fra at det er en rett linje mellom to oppgitte punkter. Vi antar at toget holder en konstant fart, som gir en lineær fremføring, altså en rett linje, og Oracle Spatial antar at det må alltid være en lineær linje mellom to punkter (må gi minst tre punkter for sirkulær). Derimot hvis linjen består av flere rette linjestykker, eller en blanding av rette og sirkulære linjestykker, så blir linjen representert ved representasjon av hvert linjestykket for seg selv.

10.4 Modellering av to ikke lineære linjer

Hva om vi nå ser på en toglinje som en linje som består av flere rette linjestykker.
Her ser vi to togfremføringer i x og t som ikke er lineære og som krysser hverandre i et punkt.

Figur 63: Krysspunkt mellom tognr 609 og 62; tid:1021, km:287

Datamodellen for denne løsningen er den samme som for rette linjer, dvs. at vi legger inn to rader til i SDO_TOG tabellen med tognr 2_609 og tognr 2_62. I dette tilfellet ser vi bort fra brudd, og ikke lineære linjestykker (f.eks sirkulære kanter) i linjen.
Her er et eksempel på hvordan man legger inn en linje som består av flere linjestykker:

```
insert into sdo_tog values (
 '2_62',
  1,
```

```
mdsys.sdo_geometry(
2002,
null,
null,
mdsys.sdo_elem_info_array(1,2,1),
mdsys.sdo_ordinate_array(0758,471.25,0948,335.8,0951,335.8,1310,89.57))
);
```

Første punkt på linjen '2_62' er (0758, 471.25). Her begynner første linjestykket. Neste punkt er (0948, 335.8). I dette punktet slutter det første linjestykket og begynner det andre linjestykket. Det andre linjestykket går til punktet (0951, 335.8) og det fortsetter på denne måten til siste punktet på linjen er nådd.

Hvis det hadde vært brudd i denne linjen, skulle vi ha lagt inn denne linjen som to linjer i tabellen SDO_TOG, det vil si vi skulle da ha fått to forekomster (to linjer) av tognr '2_62' i tabellen SDO_TOG, og hvor 'rekke' kolonnen skulle avgjøre rekkefølgen på linjene. Og hvis linjen besto av en blanding av sirkulære og rette linjestykker, kunne vi ha brukt SDO_ETYPE 4 som forteller at elementet består av en blanding av sirkulære og rette kanter.

3.4.1 Krysningspunktet mellom linjene

Den samme funksjonen blir brukt for å finne krysningspunktet mellom to ikke lineære linjer som blir brukt for to lineære linjer. Ved bruk av SDO_GEOM.SDO_INTERSECTION funksjonen på tognr '2_609' og tognr '2_62', får vi dette resultatet:

```
Resultat:
**********

SDO_GEOM.SDO_INTERSECTION(T1.RUTE,T2.RUTE,0.005)(SDO_GTYPE, SDO_SRID,
SDO_POINT(
--------------------------------------------------------------------------
----
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1021.8178, 287.227671))
```

Krysspunktet **t:** **`1021.8178`** og **x:** **`287.227671`** stemmer nøyaktig med det virkelige krysspunktet for tognr 609 og 62, fordi de dataene som er satt inn i tabellen stemmer med de virkelige data.

10.5 Hvorfor en Romlig DBMS ?

SDBMS er mye mer effektivt enn en standard DBMS, fordi i en SDBMS er den romlige algebraen implementert på integrert måte med DBMS spørreprosesseringen. En SDBMS har altså kunnskap om geometri.
En annen fordel med å bruke SDBMS er at romlige objekter kan representeres med geometri datatyper istedenfor en mengde av integer-attributter som i en relasjonell DBMS, og at man derfor slipper å gjøre mange join-operasjoner for å rekonstruere et romlig objekt som i en relasjonell DBMS. I en ORDBMS må man først definere datamodellen for romlige objekter og definere operasjoner for dem før man kan begynne å representere geometriobjektene. Man kan

ikke bruke egendefinerte romlige ADT’er på samme måte som ferdig implementerte ADT’er i en SDBMS.

11 Modellsyn i Simple Feature Spesifikasjon

Målet for [OpenGis Simple Features Specification for SQL 1998] er å definere et standard SQL-schema som støtter lagring, henting, spørring og oppdatering av enkle romlige objektsamlinger via et ODBC API. Et enkelt objekt (simple feature) er definert av OpenGis Abstrakt spesifikasjon til å ha både romlige og ikke-romlige attributter. Romlige attributter har en romlig verdi, og enkle objekter er basert på 2D geometri med lineær interpolasjon mellom hjørnene. Denne spesifikasjonen beskriver en standard mengde av SQL Geometri typer basert på OpenGis Geometri modellen, sammen med SQL funksjoner på disse typene.
Enkle romlige objektmengder vil konseptuelt bli lagret som tabeller av kolonner med romlige verdier i en Relasjonell DBMS (RDBMS), hvert objekt vil bli lagret som en rad i tabellen. En slik tabell med enkle objekter blir kalt objekttabell (feature table). Objekt tabell implementasjonen er beskrevet for to SQL miljøer:

1. **SQL92:** en kolonne med romlig verdi er implementert som en fremmednøkkel til en geometritabell.
2. **SQL92 med Geometri Typer:** er blitt utvidet med en mengde av geometri typer. I dette miljøet er en kolonne med romlig verdi implementert som en kolonne som har SQL typen trukket fra en mengde av geometrityper

For å være i samsvar med denne OpenGIS ODBC/SQL spesifikasjonen for romlige objektsamlinger, må ett av følgende tre alternativer (1a,1b eller 2) bli brukt for implementasjonen:

1. SQL92 implementasjon av objekttabeller
   a) Ved bruk av numeriske SQL typer for geometri lagring og ODBC aksess.
   b) Ved bruk av binære SQL typer for geometri lagring og ODBC aksess.
2. SQL92 med Geometri Typer implementasjon av objekttabeller som støtter både tekstlig og binær ODBC aksess til geometrien

11.1 Relasjonelle Operatorer

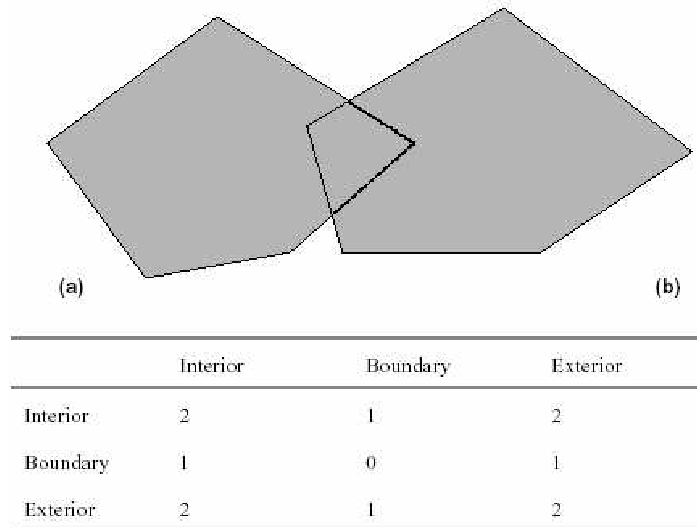
Relasjonelle operatorer er Boolean metoder som blir brukt for å teste eksistensen av et spesifisert topologisk romlig forhold mellom to geometrier. Den grunnleggende måten å sammenligne to geometrier på er å lage parvis tester av snittet mellom indre, grense og ytre av to geometrier og klassifisere forholdet mellom to geometrier basert på forekomster i den resulterende snittmatrisen. Grenselinjen til en geometri er en mengde av geometrier av neste lavere dimensjonen.

	Interior	Boundary	Exterior
Interior	$dim(I(a)\cap I(b))$	$dim(I(a)\cap B(b))$	$dim(I(a)\cap E(b))$
Boundary	$dim(B(a)\cap I(b))$	$dim(B(a)\cap B(b))$	$dim(B(a)\cap E(b))$
Exterior	$dim(E(a)\cap I(b))$	$dim(E(a)\cap B(b))$	$dim(E(a)\cap E(b))$

Figur 64 Dimensionally Extended Nine-Intersection Modell (DE-9IM), [OpenGis Simple Features Specification for SQL 1998]

- I(a): Interior
- B(a): Boundary
- E(a): Exterior
- dim(x): returnerer maksimum dimensjon(-1, 0,1, eller 2), -1 korresponderer til dim(Ø)

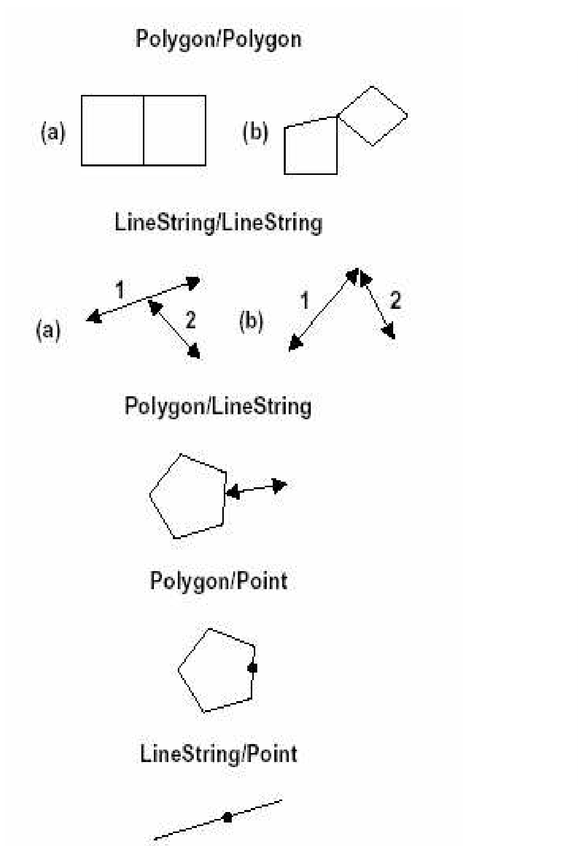
Et eksempel med DE-9IM:

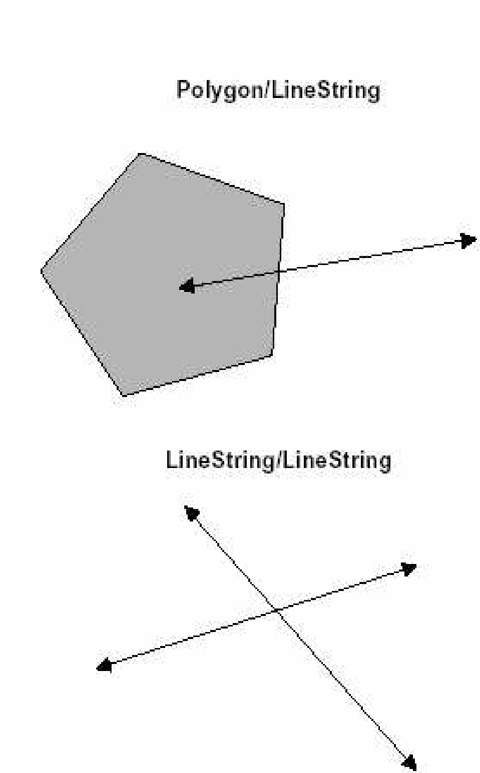


	Interior	Boundary	Exterior
Interior	2	1	2
Boundary	1	0	1
Exterior	2	1	2

Figur 65 Et eksempel på DE-9IM, [OpenGis Simple Features Specification for SQL 1998]

Navnet romlige predikater basert på DE-9IM :

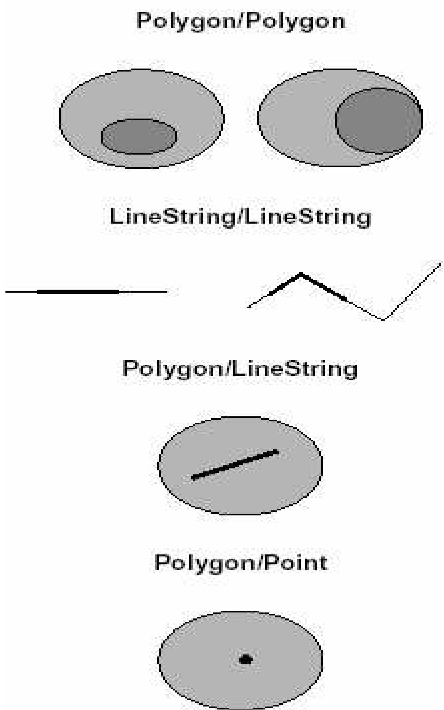




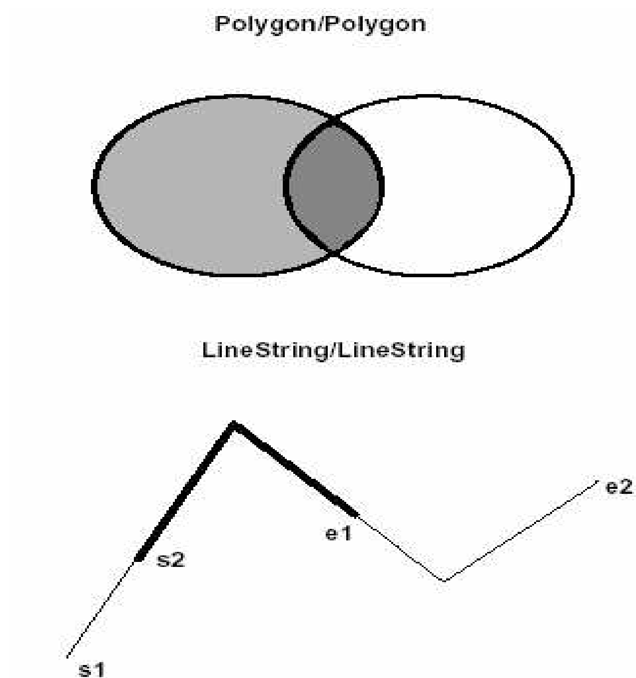
Figur 66 Krysser (Crosses)

Figur 67 Berøring (Touches)

Figur 63 og figur 64 er hentet fra [OpenGis Simple Features Specification for SQL 1998]



Figur 69 Inni (Within)

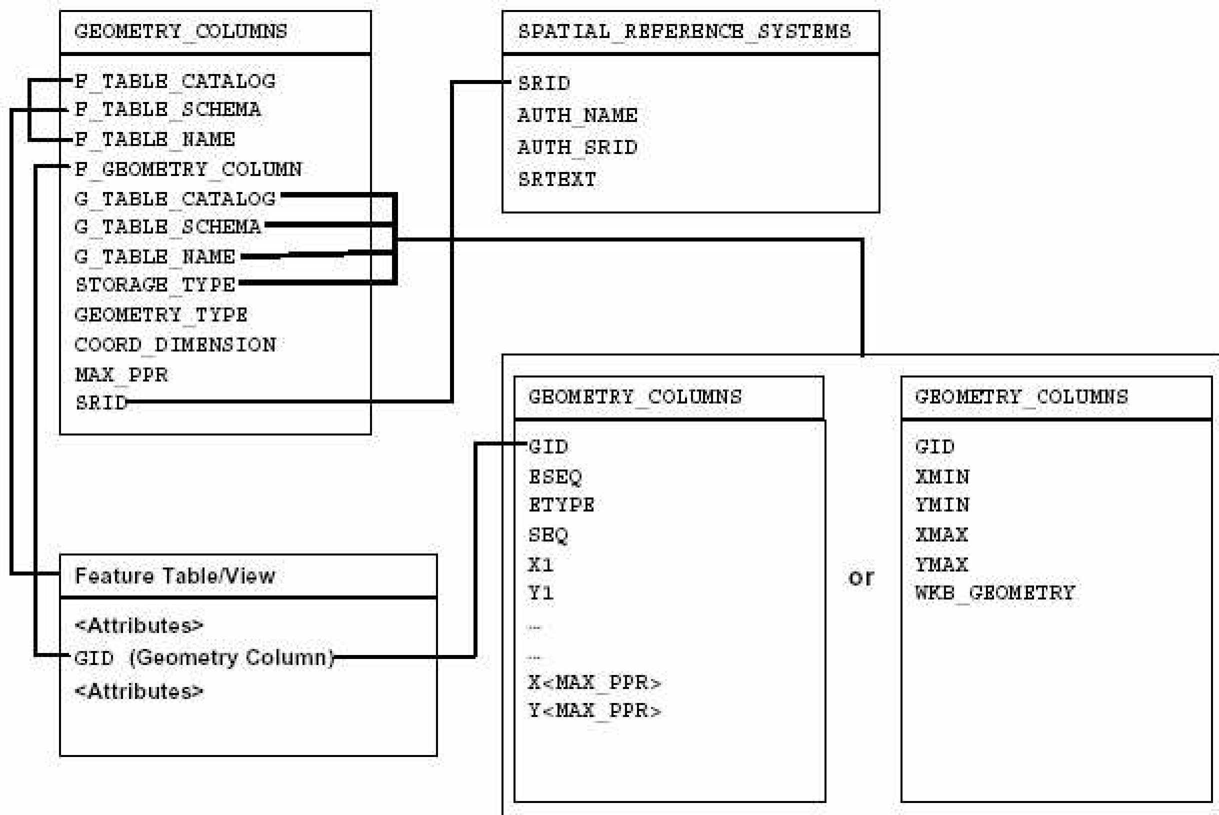


Figur 68 Overlapping (Overlaps)

Figur 65 og figur 66 er hentet fra [OpenGis Simple Features Specification for SQL 1998]

11.2 Arkitektur- SQL92 Implementasjon av Objekt tabeller

En **SQL92** implementasjon av OpenGis enkle romlige objekt samlinger definerer en skjema –se figur 70, for lagring av objekt tabeller, geometri og romlig referanse system informasjon. En objekttabell korresponderer til en OpenGis objektklasse. Hvert objekt view inneholder et antall objekter representert som rader i viewet . Hvert objekt inneholder et antall av geometriske attributtverdier representert som kolonner i objekt-viewet. Hver geometrisk kolonne i et objekt view er assosiert med et bestemt geometrisk view eller en tabell som inneholder geometri-instanser i et enkelt romlig referansesystem. Korrespondansen mellom objekt-instansen og geometri-instansen skal være dannet gjennom en fremmednøkkel som er lagret i geometrikolonnen i objekttabellen. Denne fremmednøkkelen refererer GID primærnøkkelen av geometritabellen.



Figur 70: Skjema for objekt tabeller i SQL92, [OpenGis Simple Features Specification for SQL 1998]

11.3 Arkitektur - SQL92 med Geometri Typer Implementasjon av Objekt tabeller

- **Objekt tabell:** En tabell som har en eller flere kolonner som har SQL typer trukket fra mengden av Geometri SQL typer.
- **Abstrakt Data Type (ADT):** Utvider SQL type systemet
- **Implementasjon:** Støtter konseptet av referanser til ADT instanser
- **Standardiserer:**
    - Navn og geometriske definisjoner av OpenGis SQL typene for geometri.
    - Navner, undertegner og geometrisk definisjon av OpenGis SQL funksjoner for Geometri

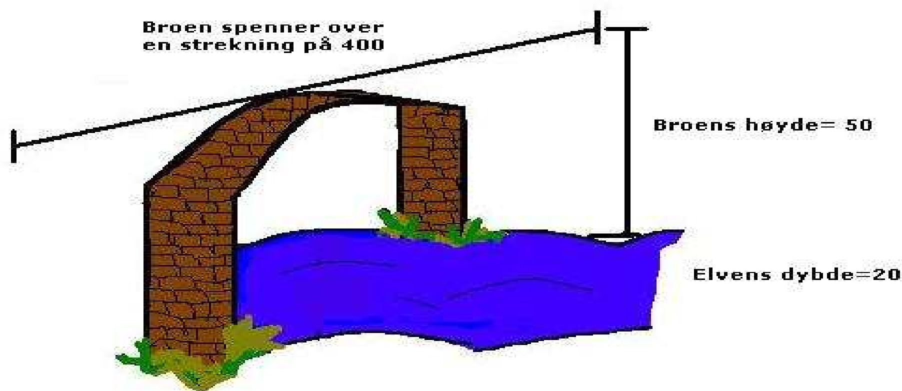
12 Modellsyn i Geography Mark-up Language (GML)

12.1 Geography Mark-up Language (GML)

GML er et markeringsspråk (mark-up language) spesifisert av OpenGis Consortium, og blir brukt for å beskrive geografiske fenomener i verden rundt oss. GML utrykker geografisk informasjon på en slik måte at den kan leses på Internett. GML bygger på eXtensible Mark-up Language (XML), i og med at GML er en spesialisering av XML –se [Lake, 2004]
I GML fenomener fra virkelig verden blir kalt objekter, som kan kategoriseres i spesielle typer, hver type er ekvivalent med en *klasse* i objektmodelleringsterminologien. GML-typer kan være konkrete som elver, bygninger eller veier, eller abstrakte og konseptuelle som politiske grenser eller helseområder. Et objekt er beskrevet ved hjelp av dets egenskaper som kan være geometriske egenskaper som plassering, form og utstrekning, eller ikke geometriske egenskaper som farge, høyde og fart. Spesifikke objekttyper som elver og veier er ikke definert inne i GML. Disse objekttypene er definert i applikasjonsskjema som vanligvis opprettes av databaseadministratorer.
GML-skjemaer er basert på XML-schema, og beskriver strukturen av GML-data og definerer elementer og attributter som er brukt i datainstanser. GML-instanser er filer eller deler av filer som inneholder faktiske geografiske data, for eksempel spesielle veier eller elver kodet i GML. Hvis man eksempelvis er interessert i å kode data om broer, kan man opprette *Bro*-instanser. For å forstå innholdet av en *Bro*-instans må det eksistere et skjema som spesifiserer hvordan denne bro-instansen skal være strukturert. Det vil si det må være et GML applikasjonsskjema som definerer et *Bro*-element, og som har en innholdsmodell som beskriver hvordan en *Bro*-instans bør struktureres.

12.2 GML Objektmodell

Objektet er kodet som et XML-element hvor navnet til objektet er en objekttype. Objektinstansen inneholder objektegenskaper, hver av dem som et XML-element med navn som egenskapsnavnet. Hver av disse inneholder et annet element som har det samme navnet som typen av egenskapsverdien eller instansen.
Egenskapene har vanligvis bare én verdi. Denne verdien kan også være referanse til et annet objekt. Siden verdien av et GML-objekt kan også være et annet objekt, er det mulig å tolke en slik egenskap som en kobling mellom dette og det andre objektet eller som en assosiasjon mellom de to objektene. I GML kan vi utrykke *rollen* til hvert av de to objektene i en assosiasjon: For eksempel man kan si *"Bro spenner Elv"* eller *" Elv er spent av Bro"*. Her har vi et virkelighetsbilde som illustrerer forholdet *"Bro spenner elv"*.



Figur 71: Virkeligetsbildet for forholdet ”bro spenner elv”.

Her er en beskrivelse for representasjon av geografiske fenomener.

```
<app:Bridge gml:id="B1">
   <app:span>400</app:span>
   <app:height>50</app:height>
   <app:material>wood</app:material>
   <gml:centerOf>
      <gml:Point gml:id="p1" srsName"#myRefSystem">
        <gml:coordinates>100.1, 23.2</gml:coordinates>
      </gml:Point>
   </gml:centerOf>
   <app:spans>
       <app:River gml:id="R1">
       <app:depth>20</app:depth>
       </app:River
   </app:spans>
</app:Bridge
```

Figur 72: Representasjon av forholdet ”bro spenner elv” i GML

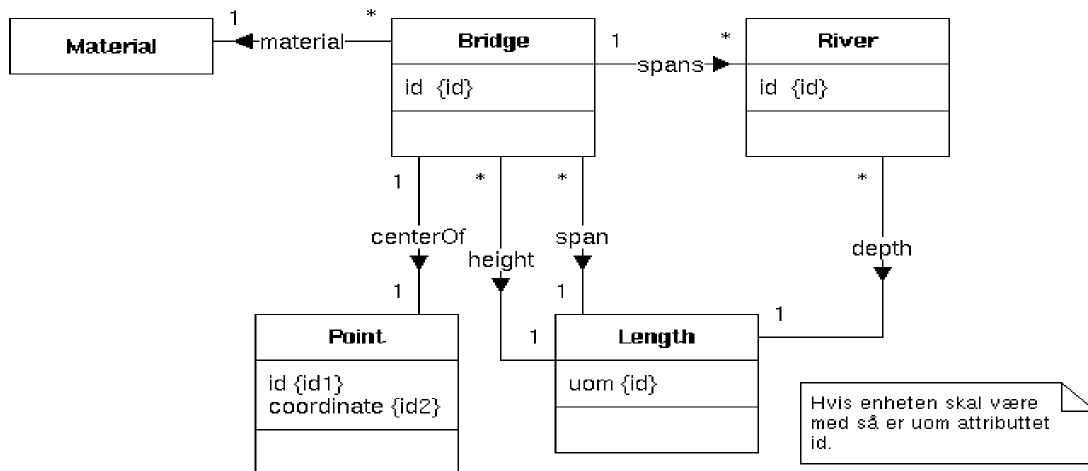
Denne Bro-instansen har egenskaper: *id* er *B1*, *span* er 400, *høyde* er 50, materialet er tre, egenskapen med geometrisk verdi har verdien *centerOf,* som inneholder et geometriobjekt Point, punktet har koordinatstringer 100.1 og 23.2, disse koordinatene er linket til CRS (Coordinate Reference System) som tilbyr den virkelige verdens kontekst for koordinater, *River*-instansen er verdien av egenskapen *spans* og har en attributt *depth* som beskriver dybden av elven.

For å betegne en enhet av måling for en hvilken som helst egenskap verdi, tilbyr GML et *uom* attributt, som spesifiserer (ved referanse) enheten av målingen for en egenskap. Hvis man skal bruke mål enheten for *height* egenskap verdien i eksemplet ovenfor, så må *uom* attributtet inkluderes i egenskapsdefinisjonen i et GML applikasjonsskjema på følgende måte:

```
<app:height uom="#m">50</app:height>
```

I eksemplet ovenfor indikerer *uom*-attributtet at enheten av målingen for egenskapen *height* er representert med en peker #m, som kan enten være en peker til et annet element i dokumentet, til en base URI spesifisert med et *xml: base* attributt, eller er en string som markerer en måleenhet, for eksempel indikerer *m* lengde i meter.

Figur 73 er et ugruppert klassediagram som viser representasjonsmodellen i GML.



Figur 73: Modellering av forholdet "bro spenner elv" i GML.

12.3 Modellering i GML

12.3.1 Geometrier i GML

GML tilbyr en mengde geometrielementer som kan bli brukt til å beskrive geometriske aspekter ved et romlig objekt, slik som posisjonen av en Bro, midtlinja på en vei eller utstrekningen av en elv. Noen features har bare en eneste egenskap med en geometrisk verdi, mens andre har flere egenskaper med geometriske verdier som beskriver forskjellige aspekter av objektet. For eksempel kan en bro ha en egenskap posisjon som beskriver plasseringen på jordas overflate og en egenskap form (shape) som beskriver den faktiske strukturen av broen.
Geometrier kan være selvstendige objekter som kan refereres gjennom egenskaper av features og andre GML objekter. Slik kan GML tillate geometriobjekter til å bli delt mellom flere GML romlige objekter..

I **GML 2** det er bare noen få geometrier som alle er lineære geometrier, det vil si at de er sammensatt av rette linjestykker. Programmereren kan imidlertid legge til sine egne geometrier til GML. Hovedgeometrityper i GML 2:

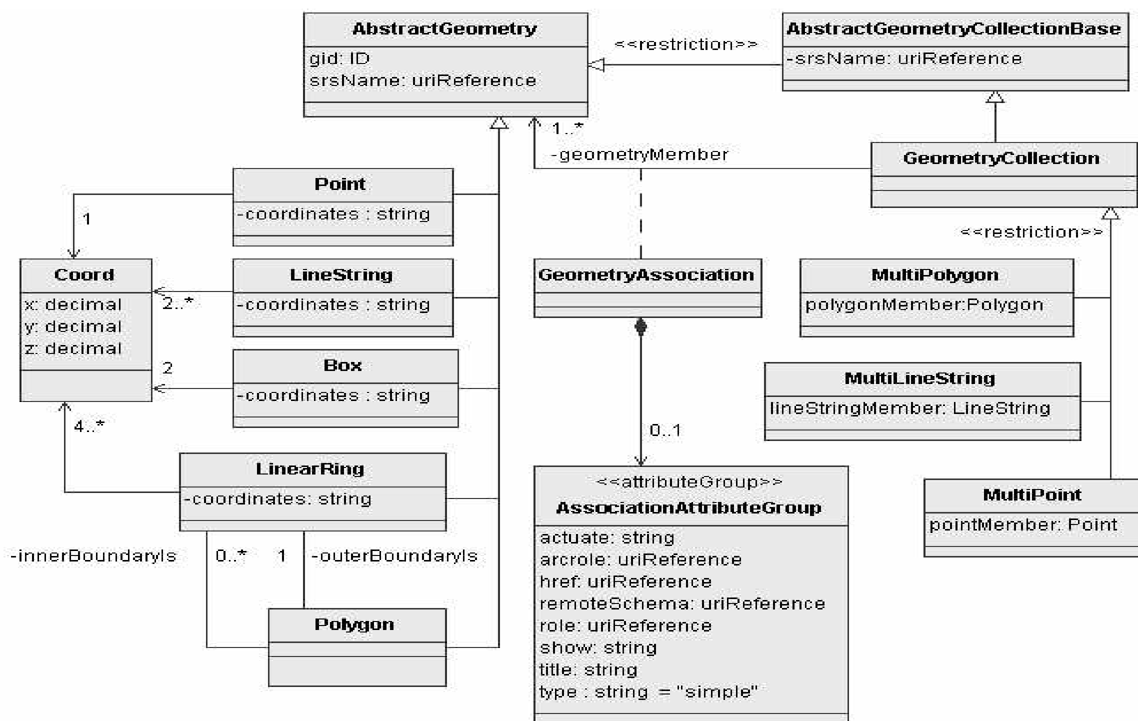
- Point
- LineString
- Polygon
- MultiPolygon

Geometriskjemaene i **GML 3.0**[9] er blitt utvidet med mange nye geometrier**,** inkludert følgende:

- Curve
- Surface
- Solid

12.3.1.1 Geometri schema

GML geometriskjema inkluderer typedefinisjoner for både abstrakte geometrielementer, (multi) point, line og polygon geometrielementer, og komplekse typedefinisjoner for underliggende geometrityper. Figuren nedenfor er en UML-representasjon av geometriskjemaet. (Dette skjemaet er en modell av assosiasjon mellom geometrier i GML).



Figur 74 : Geometri schema (ref: http://www.ia.hiof.no/~gunnarmi/gml2.1.1.pdf)

Stereotype «restriction» indikerer at for eksempel *MultiLineString* klassen er en geometrisamling hvor et medlem må være en *LineString*. Egenskapsverdier kan utrykkes på to måter: Som en in-line-verdi eller ved fjernreferanser basert på xlinks.xsd schema. AssosiationAttributeGroup-elementet tillater fjernreferanser basert på xlinks.xsd schema og tilbys gjennom gmlBase.xsd schema

[9] Geometrityper i GML 3.0 er diskutert i detalj i Volume 2: GML. A Technical Reference Guide, og i GML Versjonen 3.00 OpenGis Implementation Specification (http://www.opengis.org/docs/02-023r4.pdf).

12.3.2 Egenskaper av geometriverdier

Siden OGC abstrakt definisjon definerer en liten mengde av basis geometrier, GML definerer en mengde av geometriske egenskap elementer for å assosiere disse geometriene med romlige objekter. GML egenskaper med geometriske verdier er brukt for å beskrive rollen av geometrien i forhold til en bestemt romlig objekt. Det er tre nivåer for å navngi geometri egenskaper i GML.

1. **Formelle navn** som betegner geometriegenskaper på en måte basert på type av geometri tillatt som en egenskap verdi.
2. **Deskriptive navn** som tilbyr en mengde av standardiserte synonymer for de formelle navnene; disse tillater en mer brukervennlig mengde av termer.
3. **Applikasjonsspesifikke navn** valgt av brukere og definert i applikasjonsskjemaer basert på GML.

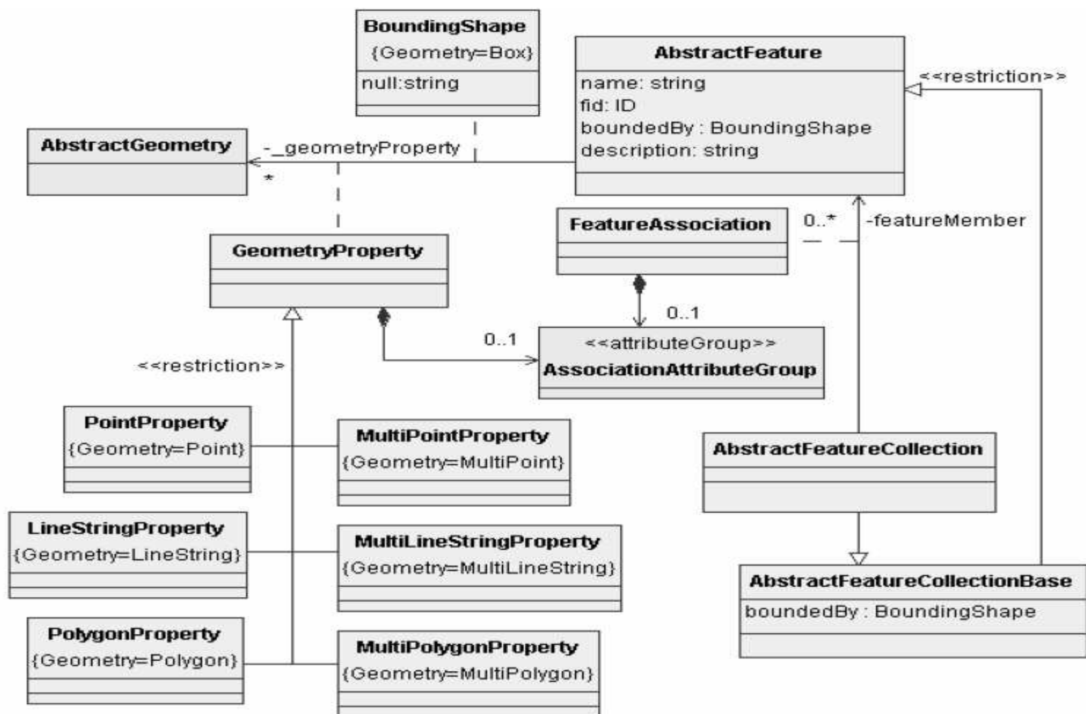
Formal name	Descriptive name	Geometry type
boundedBy	-	Box
pointProperty	location, position, centerOf	Point
lineStringProperty	centerLineOf, edgeOf	LineString
polygonProperty	extentOf, coverage	Polygon
geometryProperty	-	*any*
multiPointProperty	multiLocation, multiPosition, multiCenterOf	MultiPoint
multiLineStringProperty	multiCenterLineOf, multiEdgeOf	MultiLineString
multiPolygonProperty	multiExtentOf, multiCoverage	MultiPolygon
multiGeometryProperty	-	MultiGeometry

Figur 75: Basis geometriske egenskaper

Det er ikke nødvendig å bruke disse egenskapene (deskriptive navn) for å utrykke geometrien av en feature. Databaseadministratorer kan opprette egne egenskaper med geometriverdi når de definerer et eget GML applikasjonsskjema.

12.3.2.1 Feature skjema

Figuren nedenfor er en UML-representasjon av et Feature skjema. En geometrisk egenskap er modellert som en assosiasjonsklasse som knytter et romlig objekt til en geometri.



Figur 76 : Feature schema

12.4 GML topologi

GML topologi tilbyr en modell for koding av strukturelle assosiasjoner mellom abstrakte objekter.
GML3 topologi modell er basert på en delmengde av topologiske typer definert i ISO TC 211/DIS 19107(ISO, 2000), og innbefatter fire topologiske primitiver og en mengde av egenskaper.
Tabellen nedenfor viser forskjellige topologiske primitiver i GML3 og de korresponderende geometriobjekter som realiserer dem.

Topologiske Primitiver	Korresponderer med...
Node	Point
Edge	Curve
Face	Surface
TopoSolid	Solid

Nodene og kantene kan bli brukt til å kode sammenhenger i forskjellige fysiske nettverk, som et nettverk av veier der nodene representerer krysspunkter, og kantene representerer veiene (se nettverkmodell kapittel 4.1.2.1). Hver node og kant har to mulige orienteringer, negativ og positiv. For eksempel har en kant med to noder to *directedNode* egenskaper, og hver av disse egenskapene kan inneholde eller referere til en node med et *orientation*-attributt. Noden med negativ (-) orientering representerer startpunktet til kanten mens noden med positiv (+) orientering representerer sluttpunktet.
Hver *fasett* har en grense som består av en liste av rettede kanter. For eksempel for å realisere geometrien av en fasett *surfaceProperty* egenskap kan bli brukt til å inneholde eller referere en *Surface* objekt, som en *Polygon.*

I GML 3 kan *TopoSolid* primitiven bli brukt for å kode tredimensjonale legemer. Grenselinjen til *TopoSolid* inneholder en mengde av *directedFace* egenskaper.

12.5 GML temporale elementer

GML tilbyr støtte for modellering av dynamiske romlige objekter, som er objekter med tidsvarierende egenskaper. Den inneholder også konstruksjoner for oppretting og referering av temporale referansesystemer. Den romlige-temporale modellen i GML 3.0 er basert på den konseptuelle modellen beskrevet i ISO 19108. Tidsprimitiver i GML er basert på ISO 8601, og tidsintervaller er delvis basert på strukturer definert i ISO 11404. To geometriske tidsprimitiver er definert i GML: *TimeInstant* og *TimePeriod* .
Temporal.xsd schema tilbyr støtte for tre typer av temporale referansesystemer: temporal, ordinal temporal og temporal koordinat. Frame attributtet, som kvalifiserer *timePosition* egenskapen er brukt i GML-instanser for å referere alle disse typer av referanse systemer.
GML tilbyr TimeOrdinalReferenceSystem, component, og TimeOrdinalEra for oppretting av temporal ordinal referansesystem.
Romlige objekter kan bare ha temporale egenskaper hvis de er definert som dynamiske romlige objekter i applikasjonsskjemaet. Disse dynamiske romlige objektene har dynamiske egenskaper som inneholder tidsprimitiver som TimeInstant og TimePeriod.
En annen dynamisk egenskap er timeStamp. Når et dynamisk romlig objekt i en GML-instans har en timeStamp egenskap, tjener denne egenskapen til å definere en ’snapshot’ av det romlige objektet som representerer objektets tilstand på en instant eller i løpet av en intervall. Man kan opprette flere tidsstemplede versjoner av et og samme romlig objekt. For å inkludere alle disse forskjellige versjonene i samme instans, de trengs å bli inkludert gjennom en history egenskap.
En history er en annen dynamisk egenskap som knytter dynamiske objekter med en serie av tids skiver, hvor hver av dem inneholder en delmengde av romlige objektegenskaper som forandrer seg over tid, statiske egenskaper er ikke inkludert.

12.6 Konklusjon

Ifølge [Ron Lake] kan objektmodellen sees på som objekter og assosiasjon mellom objekter. Objekter kan ha en blanding av enkle egenskaper (boolean, integer, real strings), geometriske egenskaper og egenskaper som kan være assosiert til andre objekter. For eksempel hvis vi ser på figur 72 –”bro spenner elv”, så kan vi se disse egenskapene for objektet *Bridge:*

1. Har noen enkle egenskaper: *span, height, material*
2. Har en geometrisk egenskap: *centerOf*
3. Har en egenskap som er et annet objekt: *spans*

Modellsynet i GML er ganske likt andre modeller vi har brukt i denne oppgaven. Man må på forhånd vite hva slags rolle den romlige verdien spiller i assosiasjon til objektet. Rollen må tas med i representasjonen som geometri egenskap for å kunne representere den riktige geometritypen. For eksempel, for representasjon av et punkt geometri, må roller som; *location, postion, og centerOf* brukes i representasjonen. Og roller som *centerLineOf og edgeOf* brukes for representasjon av en *lineString.*

13 Realiseringsplattformer og realiseringer

13.1 Typer av DBMS

Ifølge Paul A. Longley, Michael F. Goodchild, David J. Maguire, David W. Rhind [Longley, 2002] er et databasehåndteringssystem (Data Base Management System – DBMS) en applikasjon som gir effektiv lagring og aksessering av data. I dag benytter alle store geografiske informasjonssystemer (GIS) DBMS teknologi for å lagre og manipulere romlige data, data som representerer modeller av virkeligheten.
DBMS kan klassifiseres etter måten de lagrer og manipulerer data på. Tre viktige typer av DBMS er brukt i dagens GIS: relasjonell (RDBMS), objekt (ODBMS), og objektrelasjonell ORDBMS). En relasjonell database innbefatter en mengde av tabeller, hver tabell har en mengde av rader som inneholder attributter til objektene vi finner i modellen. Denne enkle strukturen har vist seg å være svært fleksibel og brukbar for et bredt område av applikasjonsområder. Ifølge Paul A. Longley, Michael F. Goodchild, David J. Maguire, David W. Rhind [Longley, 2002] er i dag over 95 % av data i DBMS lagret i RDBMS

Relasjonelle databaser dominerer GIS i dag,
som de gjør innenfor mange andre forretningsområder.

Objektdatabaser var opprinnelig designet for å adressere svakhetene av RDBMS. De inkluderer muligheten for å lagre fullstendige objekter direkte i databasen (objekttilstand og oppførsel). Siden RDBMS var primært fokusert på håndtering av forretningsapplikasjoner som banksystemer, biblioteksystemer og skolesystemer, var en RDBMS egentlig aldri designet for håndtering av avanserte datatyper, som romlige verdier, lyd og video. Enda et problem med RDBMS er den svake ytelsen for mange typer av romlige spørringer. Disse problemene består dels av vanskelighetene med å utvide RDBMS med støtte for romlige data typer, og dels mangelfull støtte for prosessering av funksjoner
Selv om ODBMS er mer elegante enn RDBMS, har de ikke vist seg å bli så kommersielt vellykkede som først antatt. En grunn er den store installerte basen av RDBMS. En annen grunn er at produsentene av RDBMS nå har lagt mange viktige ODBMS-egenskaper inn i sine produkter, slik at det oppstår hybride objektrelasjonelle DBMS (ORDBMS). En ORDBMS kan tenkes som en RDBMS-motor som er tilpasset for å håndtere objekter. Både data som beskriver hva et objektet vet (attributter) og oppførselen som bestemmer hva et objekt gjør (metoder, funksjoner, algoritmer) er lagret sammen som en integrert enhet. En ideell romlig ORDBMS kan støtte romlige objekttyper og funksjoner på forskjellige måter (for eksempel Oracle Spatial)
Vi har brukt RDBMS, ORDBMS og ORDBMS med romlig håndtering for å lagre og opererer på romlige dataene i dette kapittelet, for å kunne se hvilken muligheter som foreligger på de ulike plattformene for håndtering av romlige data.

13.2 Realisering av romlige objekter

Det laveste nivået på brukerinteraksjonen med en romlig database ligger vanligvis på objektlaget, som er en organisert samling av data på en utvalgt temalag – som for eksempel alle veiene i et nettverk (linjer), alle grenselinjer til byene i et land (polygoner), osv. Alle objektlag er lagret i databasetabeller. Hvert objektlag er lagret som en enkel databasetabell i et databasehåndteringssystem DBMS. Hver tabellrad tilsvarer et objekt (instanser av objektlaget). Hver kolonne tilsvarer et objektattributt. Romlige databasetabeller atskiller seg fra ikke romlige tabeller gjennom tilstedeværelsen av en eller flere geometri kolonner. Hvordan man implementerer denne geometrikolonnen er avhengig av hvilken type DBMS man bruker –

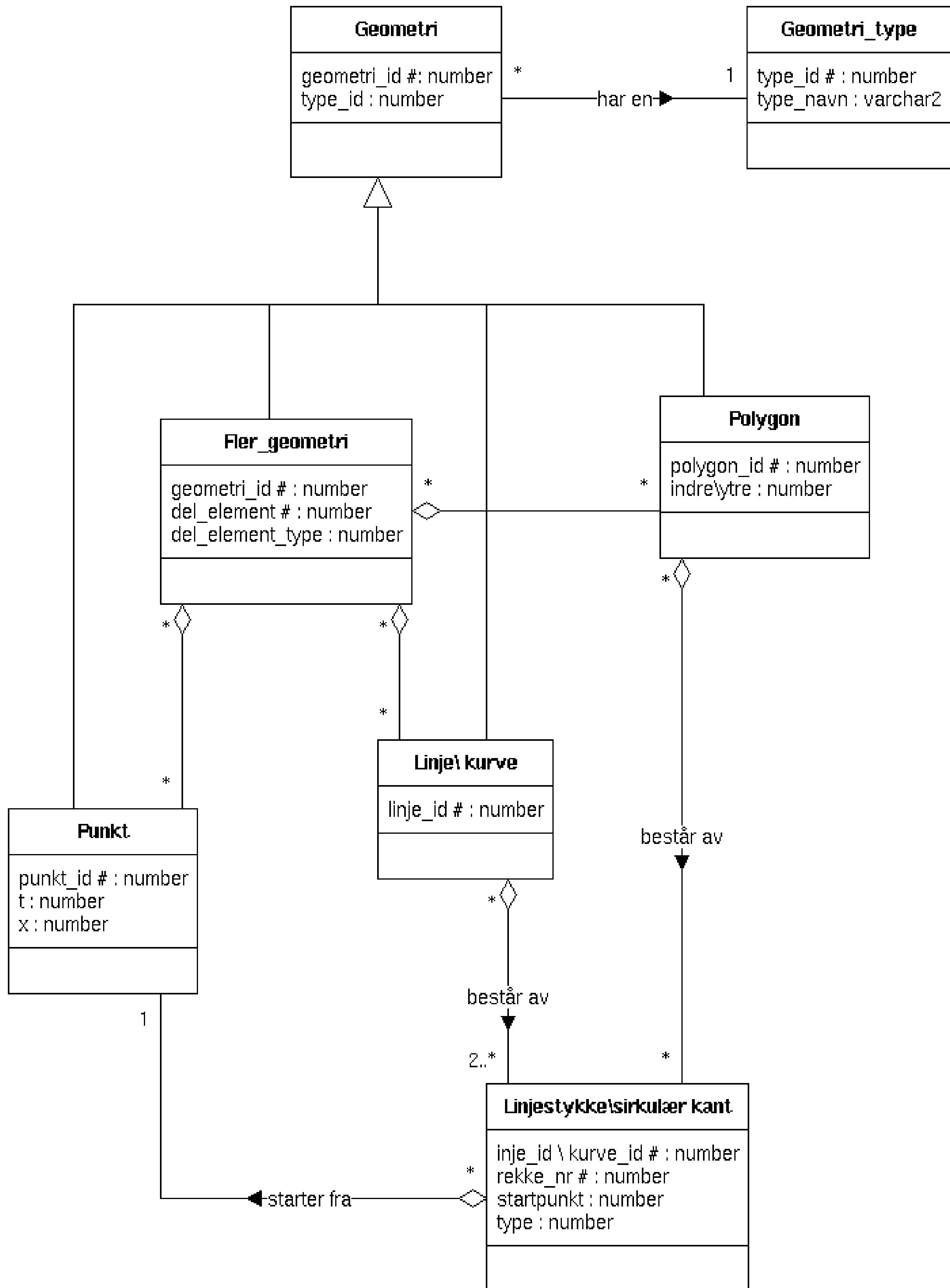
RDBMS eller ORDBMS. I dette avsnittet viser vi et eksempel på implementasjon av romlige objekter som har et attributt med en romlig verdi, på et RDBMS, på et ORDBMS, og på en objektorientert plattform[10].
Eksemplet som er implementert på alle tre plattformene går ut på å finne krysspunkt mellom to linjer når linjene består av rette linjestykker. Snitt av to linjer kan i virkeligheten være enten et linjestykke, en samling av linjestykker, en mengde av punkter, eller en blanding av punkter og linjestykker, men vi antar i eksemplet at linjene kan ha maksimum et krysspunkt, og at linjene kan ikke være parallelle. Vi skal fortsatt bruke den samme algoritmen som er beskrevet i kapittel2 (x = at + b). Her er figuren som illustrerer tog eksempelet:

Figur 77: Krysspunkt mellom tognr 609 og 62; tid:1021, km:287

13.2.1 Realisering på relasjonsdatabaseplattform

Siden RDBMS har et fastlagt utvalg av datatyper (bare enkle standard datatyper), så gir dette en begrensning for hva prosedyrer og funksjoner i en relasjonell database kan returnere. En funksjon kan kun returnere én verdi. Denne verdien kan riktignok være et array eller et objekt som består av flere numeriske verdier, men siden vi ikke kan ha andre utvidede datatyper enn standard numerisk datatyper i en RDBMS, så kan vi bare returnere en numerisk verdi (som enten er en varchar, eller en number, eller en date osv) om gangen fra en funksjon. En prosedyre kan returnere flere verdier. Derfor er det implementert en prosedyre, ikke en funksjon i databasen.
Nedenfor viser vi et konseptuelt diagram som viser alle tabeller og forhold mellom tabellene, men som skjuler implementasjonsdetaljer (ikke utvidet med geometrityper, bare enkle datatyper) for en implementasjonsorientert modell for en relasjonsdatabase som kan representere romlige objekter.

[10] Objekter i denne løsningen er imidlertid ikke persistente, noe som enkelt kunne vært ivaretatt ved bruk av et objektorientert DBMS. Imidlertid er dette av minimal betydning for utformingen av oppgaven.



Figur 78 Representasjonsmodell for representasjon av kolonne med romlig verdi.

Forklaring til figur 78

Hash-tegnet '#' bak kolonne navnet betyr at kolonnen er en primærnøkkel. Hvis det er mer enn en kolonne i tabellen som har '#' tegnet, inngår alle disse kolonnene i primærnøkkelen i tabellen.

- Tabellen ***Geometri_type*** inneholder alle geometrityper som kan implementeres med utgangspunkt i denne databasemodellen (denne tabellen kan oppdateres etter hvert som man lager nye geometrityper). Geometrityper som er støttet foreløpig:
    a) **01:** Punkt
    b) **02:** Linje\ Kurve
    c) **03:** Polygon
    d) **04:** Flere_Geometri

- Tabellen ***Geometri*** har en type_id som er fremmednøkkel til tabellen geometri_typer, og som forteller typen på geometrien. En geometri kan enten være en geometri som består en homogen eller en heterogen blanding av de tre andre typene (type samling), eller være et enkelt element som er et av primærelementene (punkt eller linje/kurve, eller polygon).

- Tabellen ***Fler_geometri*** inneholder geometrier som kan være en homogen eller en heterogen blanding av primær elementene punkt, linje og polygon. Antall forekomster av en geometri i tabellen fler_geometri er lik antall del_elementer geometrien består av. Alle del_elementer av en Fler_geometri må lagres i en tabell. Hvilken tabell del_elementet skal lagres i avhenger av om elementet har *del_element_type* lik punkt, polygon eller linje/kurve.
  Del_element_type er fremmednøkkel til tabellen geometri_typer.

- Primærnøkkel i tabellen ***Linje*** er kolonnen linje_id (denne linje_id må enten være en *geometri_id* i tabellen geometri eller et *del_element_id* i tabellen fler_geometri). En linje består av ett eller flere linjestykker/ sirkulære_kanter.

- En polygon kan være sirkel, firkant eller en vanlig polygon. Polygon_id er primærnøkkel i tabellen ***Polygon*** (polygon_id må enten være en *geometri_id* i tabellen geometri eller et *del_element_id* i tabellen fler_geometri). En polygon består av flere linjestykker/ sirkulære_kanter.
  Kolonnen indre/ytre må ha en av disse to verdiene:
    a) **10:** Polygonen er en ytre polygon
    b) **20:** Polygonen er en indre polygon.

- Punkt_id er primærnøkkel i tabellen ***Punkt*** (punkt_id må enten være en *geometri_id* i tabellen geometri eller en *del_element_id* i tabellen fler_geometri

- Alle linjestykker, sirkulære eller rette lagres i tabellen **Linjestykke/sirkulær kant**. Hver linjestykke har:
    - **startpunkt:** fremmednøkkel til tabellen punkt
    - **linje_id:** fremmednøkkel til tabellen Linje /kurve
    - **rekke_nr:** som forteller sekvensnummeret til linjestykket i linjen
    - **type:** som kan være en av disse tre:
        a) **1:** det starter et rett linjestykke fra dette startpunktet

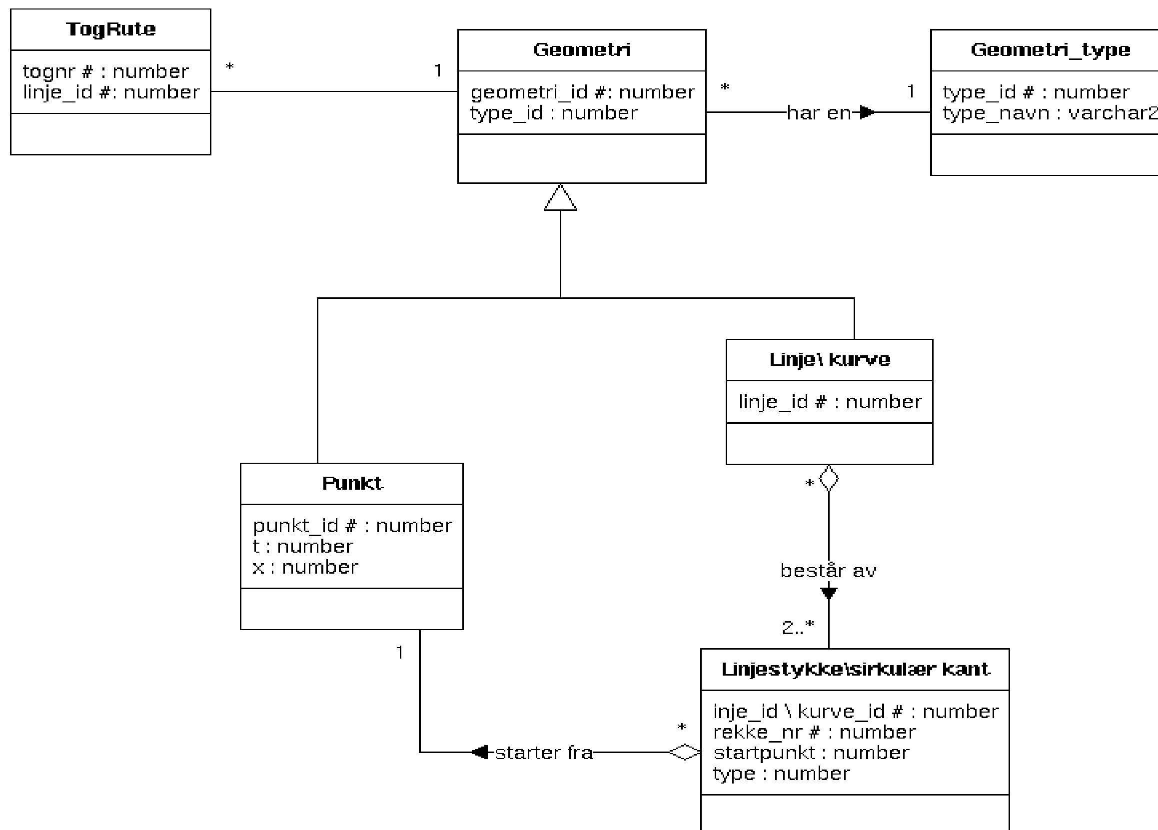
b) **2:** det starter et sirkulært linjestykke fra dette startpunktet
c) **”null”:** som betyr at dette er sluttpunktet for denne linjen

Når man skal lagre en polygon, så må man lagre første punktet to ganger i tabellen linjestykke/sirkulær kant for å lukke polygonen. Når man lagrer det første punktet første gang i tabellen, må man oppgi om det går et rett eller sirkulært linjestykke fra dette punktet, og når man lagrer punktet for andre gang, så oppgir man verdien ’null’ for å markere at polygonen lukker seg her, og at ikke noe linjestykke starter fra dette punktet.

Et punkt lagres kun en gang i tabellen for punkter, og blir referert av alle geometrier som består av dette punktet. Vi lagrer hver linje uavhengig av andre linjer, hvis flere linjer deler et linjestykke, vil dette linjestykket bli lagret en gang for hver av linjene som består av dette linjestykket, og vi vil få redundans av data i tabellen som inneholder linjestykker. For å sjekke om to linjer deler et linjestykke, kan man sjekke alle linjestykker til den ene linjen mot alle linjestykker til den andre linjen, og for hvert par av linjestykker sjekke om startpunktet og sluttpunktet er likt for begge linjestykkene, og om linjestykket som er mellom punktene har samme linjetype for begge linjene. Hvis alt dette stemmer, har linjene et felles linjestykke.
På samme måte kan man også sjekke om to polygoner tilstøter hverandre.

13.2.1.1 Implementasjonen

I dette avsnittet skal vi vise den faktiske implementasjonen av databasen, basert på implementasjonsmodellen i figur 78. Det er gjort endringer i modellen hvor det var relevant for å kunne gjøre spørringer og applikasjoner effektivt mot databasen.
I figur 78 er vist databasemodellen som ble implementert for å representere toglinjene i RDBMS :



Figur 79 Implementasjonsdetaljer for toglinjer schema

Endringer fra den konseptuelle modellen – se figur 79

Vi har ikke lenger to lineære linjer i problemstillingen, men linjer som består av linjestykker som er rette (men som i andre applikasjoner også kan være sirkulære). For at vi skal kunne bruke algoritmen som er beskrevet i kapittel 2 for å finne krysspunktet mellom to linjer, må vi først kontrollere at linjene som skal brukes i algoritmen består av bare rette linjestykker. I den konseptuelle modellen ovenfor har vi en *type* kolonne i tabellen *linjestykke\ sirkulærkant.* For å sjekke om en linje består bare av rette linjestykker, må vi i tabellen *linjestykke\ sirkulærkant* sjekke om hvert linjestykke som utgjør linjen er rett eller sirkulær. Dette er en kostbar join, derfor lager vi isteden en kolonne *linje_type* i tabellen *linje\kurve* hvor brukeren først må oppgi type på linjen som lagres, før brukeren lagrer linjestykkene til linjen.

- *Linje_type* i tabellen **Linje\kurve** kan ha en av disse tre verdiene:
  a) **1:** Linjen består bare av rette linjestykker
  b) **2:** Kurven består av bare sirkulære kanter
  c) **3:** Linjen består av en blanding av sirkulære og rette linjestykker.

- Tabellen TogRute har et tognr og en linje_id. Linje_id er fremmednøkkel til tabellen Linje.

13.2.1.2 Prosedyrer

Vi har laget i databasen lagt inn en prosedyre *rdbms_krysspunkt* som finner krysspunkt mellom to linjer. For å kunne bruke denne prosedyren må linjene bestå av rette linjestykker. Vi kan legge inn en sjekk i prosedyren på at linje_type=1 for begge linjene, hvis det er sant kan prosedyren prøve å finne om det er noe krysspunkt mellom linjene, i motsatt fall kan den gi en feilmelding. Vi har utelatt denne sjekken fordi vi går ut fra at linjene består av rette linjestykker i vårt eksempel.
Hvis linjene krysser i mer enn et punkt (noe som ikke er tilfellet i vårt eksempel), så kan man returnere et array med alle krysspunkter istedenfor en x-verdi og en t-verdi.

```
rdbms_krysspunkt(t1 in number,t2 in number,t out number,x out number )
```

Kort beskrivelse av prosedyren>

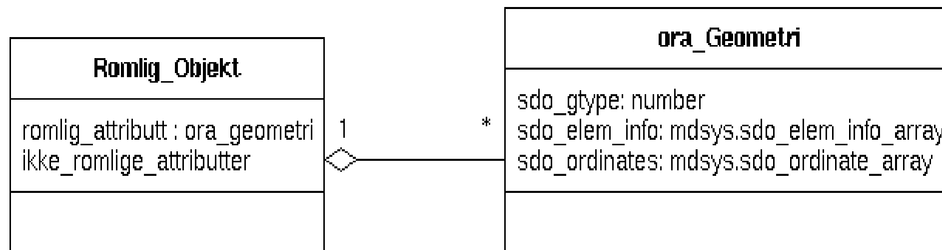
- ✓ Tar imot to linje_id og returnerer en x-verdi og en t-verdi
- ✓ Deklarer en ytre løkke som går gjennom alle linjestykkene til den første linja. Den henter ut et og et linjestykke fra linjestykke-tabellen hvor linjestykket har samme linje_id som denne linja. For hvert linjestykke hentes ut startpunktet og sluttpunktet (sluttpunktet er startpunktet til det neste linjestykket)
- ✓ Deklarer en indre løkke som gjør det samme som den ytre løkken, men for den andre linja.
- ✓ Bruker de fire punktene til å finne krysspunktet (to linjer som ikke er parallelle vil alltid få et krysspunkt dvs. x og t vil alltid få en verdi).
- ✓ Sjekker om x0 og t0 ligger mellom alle de fire punktene for linjestykkene, hvis det er sant har vi et krysspunkt, ellers ikke.
- ✓ x og t settes lik x0 og t0 hvis det er et krysspunkt, ellers settes x og t lik -1.

Denne prosedyren finner ut om det er et krysspunkt mellom to linjer uavhengig av applikasjonen. Vi lager en applikasjon i en PL/SQL-blokk som passer til applikasjonsområdet vårt og som kaller på denne prosedyren. Denne applikasjonen henter ut toget som har disse linjene og skriver ut om det er krysspunkt mellom togene eller ikke, avhengig av hvilken verdi x og t har.

13.2.2 Realisering på objektrelasjonell databaseplattform

Vi har laget to modeller for representasjon av geometriske datatyper for romlige objekter i den objektrelasjonelle databasen.

I. Denne første modellen representerer geometriske datatyper på samme måte som Oracle Spatial modellen – se modellsyn i SQL92:



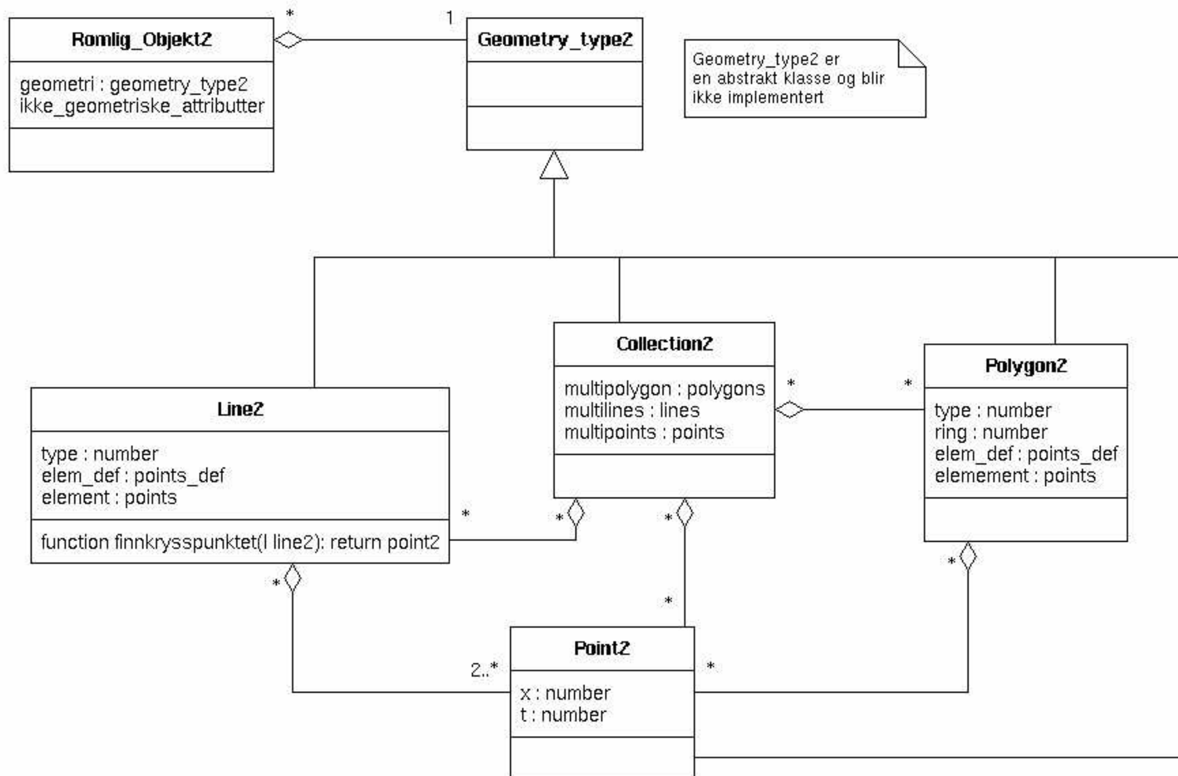
Figur 80 Representasjonsmodell (1) for representasjon av romlige objekter med en kolonne av romlig verdi, av datatypen geometri, i ORDBMS (basert på Oracle Spatial modellen)

I tillegg til annen informasjon (ikke romlig informasjon) har ethvert romlig objekt et romlig_attributt av datatypen geometri som inneholder den romlige verdien. Typen Geometri:

- ADT' en 'geometri' har en g_type som forteller hvilken geometritype geometrien har (alle geometrier er 2-D).
- Elem_info_array er et tall-array som inneholder informasjonen om
  - alle elementer geometrien inneholder (et enkelt objekt eller en spagetti modell)
  - typen på elementene (eller objektet (point, line, eller polygon))
  - hvor hvert element (eller objekt) begynner i ordinate _array
- Ordinate_array som inneholder alle koordinater til geometrien

Hvilke geometrityper og elementer som er tillatt i modellen ovenfor fremgår av kapittel "Modelsyn i SQL92".

II. Den andre modellen (som ble implementert med tog eksemplet) for å representere kolonnen med romlig verdi av et romlig objekt, ser slik ut:



Figur 81 Objektrelasjonell modell(2) for representasjon av romlige objekter, med kolonne med romlig verdi av en geometrisk type.

Her er et eksempel på hvordan man lagrer et romlig objekt basert på den objektrelasjonelle modellen i figur 81:

```
CREATE  type Geomtog as object(
      tognr NUMBER,
      geometry line2,
      member function krysspunkt(t Geomtog) return point2);


CREATE table geomtogene of Geomtog;
```

Her lager vi først en type Geomtog som er et togobjekt og har et *tognr,* et *geometri*-attributt som er av datatypen line2 og en funksjon *krysspunkt* som tar imot et *geomtog*-objekt og returnerer et *point2*-objekt.
Deretter lager vi en tabell *geomtogene* som lagrer alle *geomtog* objektene.

Et eksempel på hvordan vi legger inn et geomtog objekt i denne databasen:

```
insert into geomtogene values(62,line2(1,points_def(1,1),
points(point2(0758,471.25),point2(0948,335.8),point2(0951,335.8),
point2(1310,89.57))));
```

Beskrivelse av modell2 – se figur 81

- Romlig objektet i modell2 har noen ikke-geometriske attributter og et geometrisk attributt som kan enten være av typen Line2, Point2, Polygon2 eller Collection2. Polygon2 kan være en sirkel, en firkant eller en vanlig polygon. I eksemplet ovenfor har vi et ikke-geometrisk attributt *tognr* som vi setter til å være 609. *Geometri*-attributtet i eksemplet er av datatypen Line2.
- Region kan inneholde en homogen eller en heterogen blanding av Polygon2, Line2 og Point2. Alle disse multiattributter er av datatypen varrays med maks 10000 elementer av de korresponderende datatypene i hver varray:

    - Points:
      ```
      CREATE type points as varray(10000) of point2;
      ```
    - Lines:
      ```
      CREATE type lines as varray(10000) of line2;
      ```
    - Polygons:
      ```
      CREATE type polygons as varray(10000) of polygon2;
      ```

- Point2: Har to attributtverdier t og x, begge av typen number.
- Både Line2 og Polygon2 har :
    - En unik id (i eksemplet linje_id er '2101')
    - Type som forteller om elementet består av bare rette linjestykker (1), eller bare sirkulære kanter (2), eller en blanding av sirkulære og rette linjestykker(3). (I eksemplet: linjen har type '1', som betyr at linjen består av kun rette linjestykker)
- Elem_def er et varray av typen number som inneholder par av tallkombinasjoner. Antall kombinasjoner er avhengig av hvor mange linjestykker linjen består av. Første tallet i hver kombinasjon forteller indeksen for det første punktet til del-linjen i arrayet, hvor første punktet for del-linjen er det samme som siste punktet for del-linjen før, og andre tallet i kombinasjonen forteller typen til del-linjen. I eksemplet består linjen bare av rette linjestykker (altså ingen kombinasjoner), derfor har vi bare en kombinasjon i elem_def arrayet '1,1' som forteller at linjen begynner fra indeks 1 i varrayet og er av typen 1).
- Element er et varray av typen points som inneholder alle punktobjekter til elementet. For linjer kan det samme punktet ikke gjentas i varrayet, men for polygoner må det første punktet lagres to ganger: Som start- og sluttpunkt for polygonen.
- Polygon2 har en ring som betyr at polygonen enten er en ytre polygon (1003) eller en indre polygon (2003).
- Line2 har en funksjon *finnkrysspunktet* som tar inn en linjeobjekt som parameter og returnerer et punktobjekt.

13.2.2.1 Romlige ADT operasjoner

Resultatet av en operasjon må enten være en atomær type (number eller varchar o.s.v.) eller en av de definerte abstrakte typene. Det vil si at resultatet av en operasjon basert på vår objektrelasjonelle modell2 schema kan være en av disse typene:

- Point2 eller Points
- Line2 eller Lines
- Polygon2 eller Polygons
- Region

Operasjoner har ikke lov til å returnere et resultat som ikke er av enten standard DBMS type eller en av de sju ovennevnte abstrakte typer.

I funksjonen *finnkrysspunktet* ovenfor begrenser vi oss til at når vi finner krysspunkt mellom to linjer, så vil vi alltid få enten ett eller intet punktobjekt som krysspunktet for linjene.

Metoder for å finne krysspunktet mellom toglinjene:

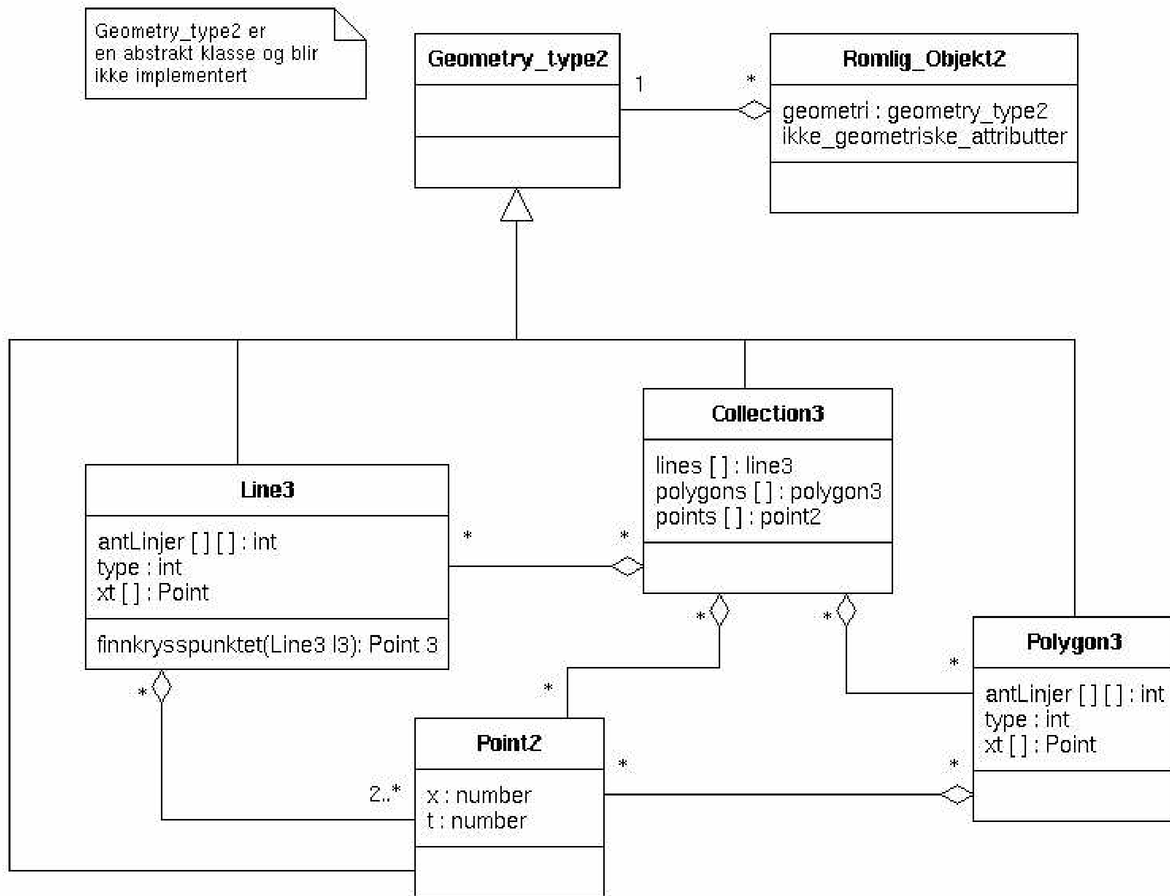
1. En blokk i PL/SQL kaller på funksjonen *finn_togene* med to tognr som input-parametre (tognummer for togene som det skal sjekkes krysspunkt mellom) og funksjonen returnerer et punktobjekt. I blokken sjekkes om punktet er et krysspunkt. Hvis x- og t-verdier til punktet er -1 er det ingen krysspunkter, ellers er det et krysspunkt mellom linjene.
2. Funksjonen finn_togene kaller på *krysspunkt*-funksjonen til et av togobjektene med det andre togobjektet som inn-parameter til funksjonen. Funksjonen *krysspunkt* returnerer et punktobjekt.
3. Funksjonen *krysspunkt* kaller på funksjon *finnkrysspunktet* som er i line objektet og sender med det andre togets geometri objekt som inn-parameter til funksjonen og funksjonen *finnkrysspunktet* returnerer et punktobjekt.
4. Funksjonen *finnkrysspunktet* tar imot et line2-objekt og finner krysspunktet mellom seg selv (som er et line objekt) og det innkommende line2 objektet
   - Funksjonen bruker en algoritme til å finne krysspunktet,
   - Sjekker om punktet virkelig er et krysspunkt
   - Hvis virkelig krysspunkt opprettes det et nytt punktobjekt med krysspunkt verdiene x0 og t0, ellers opprettes det et nytt punktobjekt med -1 verdi for x0 og t0.

13.2.2.2 Fordeler ved objektrelasjonelle databaser

Objektrelasjonelle databaser blir stadig mer valgt for implementasjon av romlige databaser, siden de tilbyr utvidelsesmuligheter i forhold til tradisjonelle relasjonelle databaser. Og dette fører til at geometriske data typer kan bli implementert.

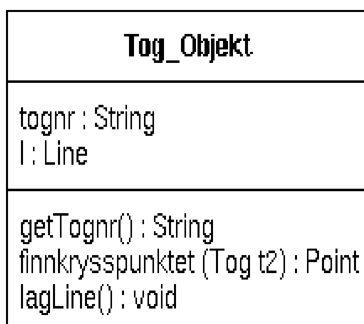
13.2.3 Realisering på objekt orientert plattform

I objektorienterte programmer er det en direkte kobling mellom attributter (tilstand) og metoder (oppførsel) som operer på disse attributtene, denne koblingen av attributter og metoder er inneholdt i en enkel pakke, og denne samlede enheten blir kalt objekt.
Nedenfor er det en modell (figur 82) som viser hvordan man kan implementere romlige objekter i et objektorientert program.



Figur 82 Objektorientert representasjonsmodell for romlige objekter.

Klassifisering er et viktig konsept i objektorienterte programmer. For eksempel et romlig objekt har en romlig verdi, og den romlige verdien er et objekt av en geometritype (Point2, Line2, Polygon2 eller Collection2). En geometritype har en mengde av dataelementer (attributter) og en mengde av basis romlige prosedyrer. Hvert objekt kjenner igjen og kan svare på spørsmål som ”*sjekk om du har et krysspunkt med dette gitte geometriobjektet*”.



For eksempel har et togobjekt en mengde attributter (tognr og rute(linje)), og et antall prosedyrer. Et tog objekt er koblet med prosedyren
for å finne krysspunktet: togobjektet sjekker linjen sin mot det andre objektet2 sin linje og svarer om det er krysspunkt mellom linjene eller ikke. Hver gang dette objektet (attributt/metode bunten) blir kalt ,*” objektets metoder er eksekvert på dens attributter ”.*

Når ”finnkrysspunktet (Objekt o) ” er koblet med andre type objekter, kan den også returnere andre type objekter enn punktobjekter. Hvis man for eksempel kobler denne metoden til geometritype polygon og punkt også, kan man finne krysspunkt mellom to polygoner, en

polygon og en linje, eller en polygon og et punkt. Metoden vil da enten returnere en samling eller et enkelt objekt av de tre definerte geometri typene.
Resultatet avhenger av de metodene som eksekveres, for eksempel vil metoden *finnkrysspunktet* være basert på høyst ulike algoritmer avhengig av geometritypen den er definert i.

13.2.3.1 Fordeler ved objektmodeller

Den største fordelen med å bruke objekter som basis database-design er objektenes enkelhet av å bruke dem både for design og for bruker verifisering. En annen ting er at objekt modeller kan direkte bli implementert i en relasjonell form.

13.2.4 Oppsummering

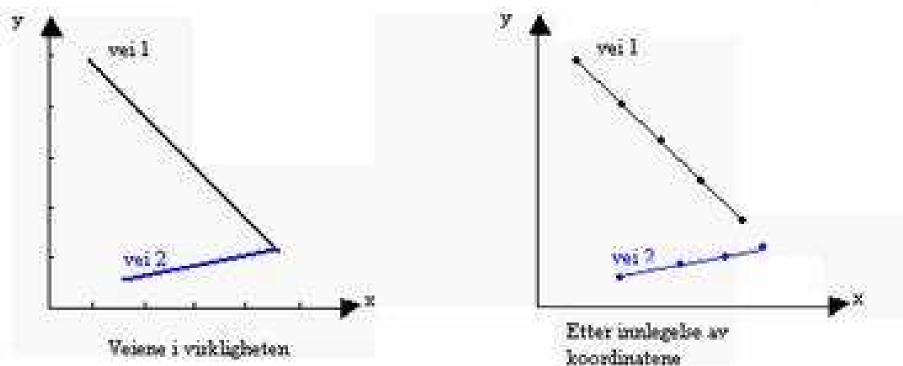
I en RDBMS er mengden av datatyper bestemt, i objektrelasjonelle og objektorienterte databaser er det en innebygget mulighet for å utvide mengden av data typer. Og denne muligheten er en klar fordel, spesielt hvis man arbeider med utradisjonelle applikasjoner som GIS, men byrden med å konstruere syntaktiske og semantiske riktige datatyper ligger nå på databaseapplikasjonsutvikleren. For å kunne fjerne noe av byrden, har kommersielle DBMS-produsenter introdusert applikasjon_spesifikke ”packages” som tilbyr et brukergrensesnitt til databaseutvikleren. For eksempel har Oracle en *Spatial Data Cartridge* pakke for GIS_relaterte applikasjoner.
Applikasjoner som aksesserer objektrelasjonelle data trenger mye mindre detaljerte kunnskaper om struktur av data fordi Java klassen som representerer data tilbyr passende, enkle å bruke metoder for å manipulere det konsistent.

14 Noen eksempler på modeller og bruken av dem

I lang tid har mennesker ferdet fra et sted til et annet, hele tiden har det vært meningen vært å bruke minst mulig energi og tid ved slik ferdsel. Man velger da de veier hvor man kommer lengst på kortest tid med minst bruk av ressurser. Et eksempel som bruker dette som grunnlag men legger til litt ekstra er transportarbeidernes vei valg. De skal transportere vare fra et sted til et annet sted så raskt som mulig. Dagens veier er slik at de har begrensninger for når på dagen er slikt kjøretøy kan kjøre på veien, avhengig av last. Videre kan veier bli stengt på grunn av rasfare eller ved lavere temperatur enn en gitt nedre grense. Et annet problem kan være at det kan være deler av en vei er stengt mens andre deler igjen er åpen for ferdsel. Slike begrensinger gjør det vanskelig å regne seg frem til hvilken veistrekning som vil være mest hensiktsmessig med en bestemt last og ved en bestemt tid.
Et annet problem med modelleringen av generelle veier er faren for feil på grunn av avrunding. Datamaskiners evne til å ta med uendelig mengde data er begrenset og ved enkelte situasjoner kan man for eksempel få feil som er slik at to veier som i virkeligheten er koblet sammen ser ut i vårt system som om de står fra hverandre. Dette kan rettes på ved å legge inn visse regler, for eksempel kan man si at vei A sitt ende punkt er lik vei B sitt startpunkt, eller eventuelt legge inn endepunkter og startpunkter som egne objekter som har koblet veier til seg.

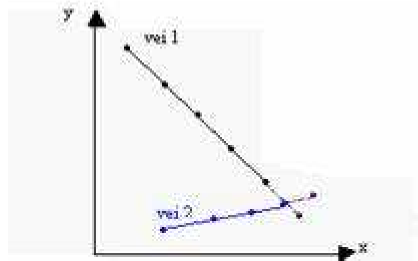
14.1 Korteste vei

Det første som trengs å gjøres i et slikt system er å kunne sette inn skranker som viser at to veier er koblet sammen, om en vei er blindvei eller ikke ender i en annen vei. Dette kan for eksempel gjøres ved at man legger inn topologiske skranker. Man legger inn det punktet der to eller flere veier møtes som en node. La oss anta at vi har to veier som i virkeligheten er knyttet sammen, når informasjonen om dem legges inn i datamaskinene kan det på grunn av avrunding se ut som noe helt annet. Veiene kan stå fra hverandre, siden vi ikke tar med alle verdiene. La oss anta at vi ta med koordinatene til veiene for hver meter, og vei 1 er på 4,5 meter mens vei 2 er på 3 meter.



Figur 83 Problemer ved innleggelse av koordinater

Som vi ser av figur 83 vil veiene ikke lenger være knyttet sammen i et punkt siden dette punktet kun blir lagret for vei 2 og ikke for vei 1. Dette kan løses ved å hente ut koordinater for hvert cm av veien. Med dette vil igjen slå ut feil på med et mer detaljert nivå. En annen løsning kan være å forlenge veien ved å legge litt ekstra lengde på veien slik at man får et krysningspunkt for to veier, men dette vil gi et annet problem.

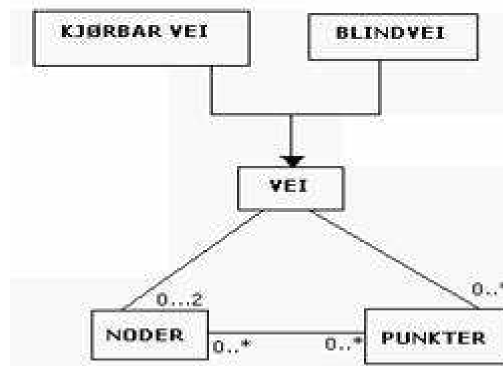


Figur 84 Vei 1 og Vei 2 fra figur 28 med “ekstra vei”

Det vil dermed bli vanskelig å kunne bestemme om et kjøretøy skal forsette å kjøre på vei 1 eller kjøre av på vei 2 ved krysningspunkter. Dette må det legges inn beskrakninger på, for eksempel kan man definere det ekstra veistykket, det vil si den delen av veien som kommer etter krysningspunktet som en blindvei, privat vei eller eventuelt stengt for ferdsel. Ved hjelp av ekstra lengde, som en buffer, vil alle veier som er knyttet med andre veier ha en mulighet til å bli knyttet uten at man tenke på diskretiseringnivå og hvilke koordinater man tar med eller ikke. Et annet problem som kan oppstå på grunn av tillegg av ekstra lengde på veiene er at to veier som ikke er knyttet sammen men som befinner seg ganske nær hverandre kan få et krysningspunkt som ikke eksiterer i virkeligheten. Dette kan løses ved å at man for alle veier tar med en verdi som gir den virkelige lengden til en bestemt vei og sjekke om krysningspunkter til to veier ligger innenfor de lengdene som er lagret for begge veiene.

14.2 Fra krysningspunkt til node:

Siden vi lagrer en mengde koordinater for hver vei objekt, er det naturlig å modellere dette ved at vei objektet består av en mengde med punktobjekter. Av disse punktobjektene er noen punkter slik at de snitter med et veiobjekts samling punkter og disse krysningspunktene kan kalles noder. Vei objektet igjen består av to deler, nemlig den ”kjørbare” veien og en blindvei, det vil si den delen av veien som er lenger enn veiens virkelige lengde.



Figur 85 Modellering av vei

Etter at nodene blir funnet kan man bruke denne informasjonen til å finne korteste vei. Hver vei har en fart og en virkelig lengde samt en del restriksjoner. Disse kan sees på som kantene til nodene vi har funnet. Vekten til kantene kan bli den tiden et kjøretøy bruker fra veiens start til veiens slutt. Dette kan sees på som en vektet graf med veiene som kanter og de punkter hvor det er snitt som noder.

Figur 86 Fra vei samling til graf med noder

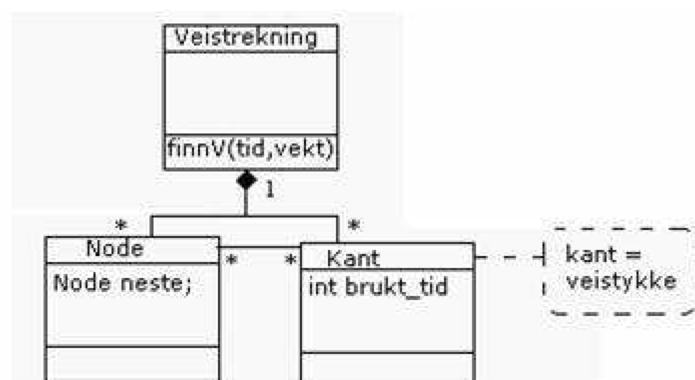
14.3 Dijkstras algoritme

For å finne kortest vei i et generelt eksempel kan man benytte seg av diverse teknikker og algoritmer. En kjent algoritme for å finne kortest vei er Dijkstras algoritme som følger seks punkter:

1. Kall startnoden for start. Sett distansen lik 0 og marker start som "kjent".
2. Sett distansen til alle nabonoder w lik kosten fra start til w, dvs. dw := cs;w.
3. Sett bakoverpekerne for nabonodene lik ps.
4. Velg ukjent node v med minst distanse, og marker v som "kjent".
5. Se på alle ukjente nabonoder w:
(a) Reduserer distansen for w dersom lengden vi får ved å følge stien gjennom v er kortere enn den gamle lengden: dw := min(dw; dv +cv;w).
(b) Hvis lengden ble redusert, så sett bakoverpekeren lik pv.
6. Så lenge det finnes ukjente noder, gå til punkt 3.

Hentet fra http://www.uio.no/studier/emner/matnat/ifi/INF1020/h04/undervisningsmateriale/grafintroforelesning.pdf

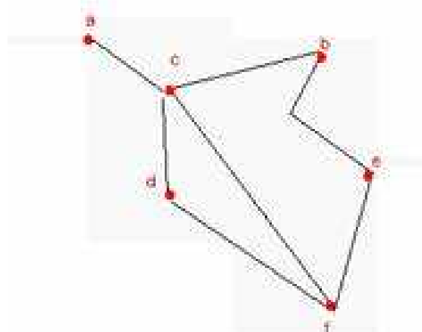
Vi er interessert i å finne den veistrekningen som bruker minst mulig tid, og kan kjøre algoritmen mot brukt tid i stedet for distansen mellom to noder.
For hvert veistykke regner man ut den totale tiden et kjøretøy vil bruke med en vekt og ved bestemt tid. Veistykket kan sees på som kanten til noden som er vektet med tiden kjøretøyet vil bruke.
Dermed kan Dijkstras algoritme bidra med å finne korteste vei i en samling av noder med vektede kanter. Tidsforbruket vil bli O(|antall kanter| +|antall noder|²)



Figur 87 Klassediagram

14.4 Restriksjoner og brukt tid

Ved første utkast vil det se ut som om hovedproblemet ligger i å finne kosten for hvert veistykke eller kant. Men et av problemene som dukker opp litt senere er hvordan man kan bestemme kosten til et veistykke når et veistykkets egenskaper forandrer seg med tiden.



Figur 88 Eksempel på veinett

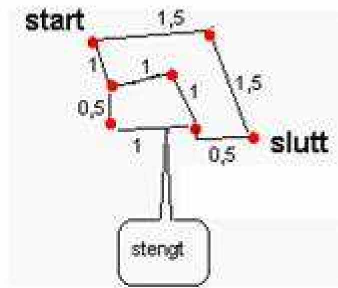
La oss ta for oss et enkelt eksempel, se figur 88. Veistykket fra c til f er slik at det mellom klokken 14.00 til 16.30 er stengt for kjøretøy med en vekt over 6 tonn. La oss videre anta at et kjøretøy starter fra a og skal kjøre til f, og ønsker å bruke minst mulig tid. Tiden kjøretøyet starter i a er satt til 12.00. Veistykkene cd, cb, be og ef er satt uten restriksjoner og er åpne for ferdsel hele døgnet, mens veistykket df er stengt fra 01. januar til 15. mars for kjøretøy over 5 tonn døgnet rundt. Vekten for de forskjellige veistykkene vil bli regnet ut fra fart og vil bli som tabellen viser nedenfor:

Veistykket	Fart(km/t)	Strekning(km)	Restriksjoner	Brukt tid /vekt
a-c	50	100	-	2
c-b	50	100	-	2
c-d	60	100	-	1 t 40 min≈1,668
c-f	90	350	Stengt mellom 14 til 16.30 hvis vekt > 6 tonn	3 t 53 min≈3,889*
b-e	60	60	-	1
e-f	60	120	-	2
d-f	100	300	Stengt hvis vekt > 5 tonn	3 t*

Tabell 5 – Beregn av brukt tid med hensyn på restriksjoner

La oss anta at et kjøretøy på 6,5 tonn skal kjøre fra a til f. Hvis vi ser bort fra restriksjonene vil den korteste veien bli fra a til c og fra c til f. Men siden veistykket c til f har restriksjonen om at kjøretøy over 6 tonn ikke kan kjøre mellom kl 14 til 16.30 må vi ta hensyn til dette. Hvis klokken er 14.00 ved punkt c vil de være mer hensiktsmessig å prøve de andre veiene. Følger vi veien fra c til d og fra d til f vil kjøretøyet bruke 4 timer og 40 min mens det vil bruke 5 timer fra c til b, b til e og fra e til f. Kjører vi mellom 16.mars til 31. desember vil veistykket cdf være å foretrekke, men ved en annen tid på året blir vi nødt til å snu ved d og kjøre tilbake og velge en annen rute. Vekten for veistykker uten restriksjoner blir beregnet enkelt, men ved restriksjoner må vi ta hensyn til flere ting, blant annet hva klokken og datoen er ved hver veistykkets start. Starter kjøretøyet for eksempel 02. februar klokken 14.15 fra a vil det være ved c kl 16.15. Dermed kan vi velge å vente et kvarter og kjøre strekningen fra c til f og dette vil være den raskete veien. Starter vi derimot klokken 12.00 vil vi ved c måtte vente 2 timer og 30 minutter og dermed brukt 3 timer og 53 minutter på å kjøre fra c til f noe som gir en total tid på 8 timer og 23 minutter. Hvis kjøretøyet kjører veistykket acbef vil den totale tiden bli 6 timer noe som er betydelig mindre enn veistykket acf.
Algoritmen vil dermed bli slik at man bestemmer vekten til veistykkene før et veistykke med restriksjoner og kjøre dijkstras algoritme på disse veistykkene. Når man kommer til veistykket med restriksjoner kan man sjekke restriksjonene opp mot faktiske forhold og bestemme vekten for ett

gitt veistykke ut i fra dette. Et problem som kan oppstå ved å bruke denne algoritmen er at man ikke kan ta hensyn til helheten. Hvis vi følger djikastras algoritme vil vi komme til et punkt som er slik at videre veier er stengt i lang tid fremover. Figuren nedenfor viser et enkelt eksempel som kan brukes til å vise at dijkstras algoritme ikke helt vil bli utført som planlagt. Vi følger dijkstras algoritme og setter start til s, kjent og distanse lik 0, deretter velger dijkstra veien som er vektet 1 og så veien som er vektet lik 0,5. Følger vi dijkstras algoritme vil vi dermed komme til veien som er vektet lik 1 men som er stengt.

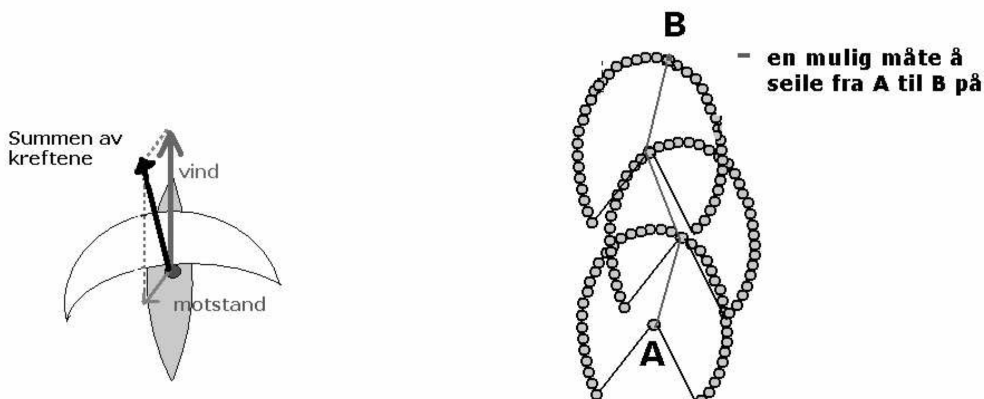


Figur 89 Veinett med restriksjoner

Dermed kan man beregne korteste vei ved å regne ut tiden man vil bruke ved å kjøre alle mulige veier fra et sted til et annet med de restriksjonene som er satt opp og trekke ut den veien som bruker minst mulig tid ved en bestemt start tid og en bestemt dato. Tidsforbruket vil bli O(E*V), noe som er betydelig større enn dijkstras algoritme.

14.5 Seiling og korteste vei problemet

Transport eksempelet har hatt visse begrensinger; Kjøretøyet kan kun kjøre på det som er blitt beregnet som en vei. En seilbåt på et åpent hav kan kjøre overalt uten å være knyttet til bestemte "veier". En seilbåt skal kjøre fra et sted A til et annet sted B, for å komme fortest frem må man ta hensyn til vindstyrke og størrelse på seilet. Vindretningen og motstanden vinden møter i seilet bestemmer hvilken vei seilbåt skal seile, hvis vi ser bort fra roret. Vindkraften møter motstand fra seilet og summen av disse kreftene er lik kraften fremover. Fra newtons lover vet vi at $\sum F = f_1 + f_2$. Velger man positiv retning fremover vil vind fremover ha en positiv retning, mens motstanden fra seilet har en negativ retningen slik at $\sum F = F_{vind} - F_{motstand}$.



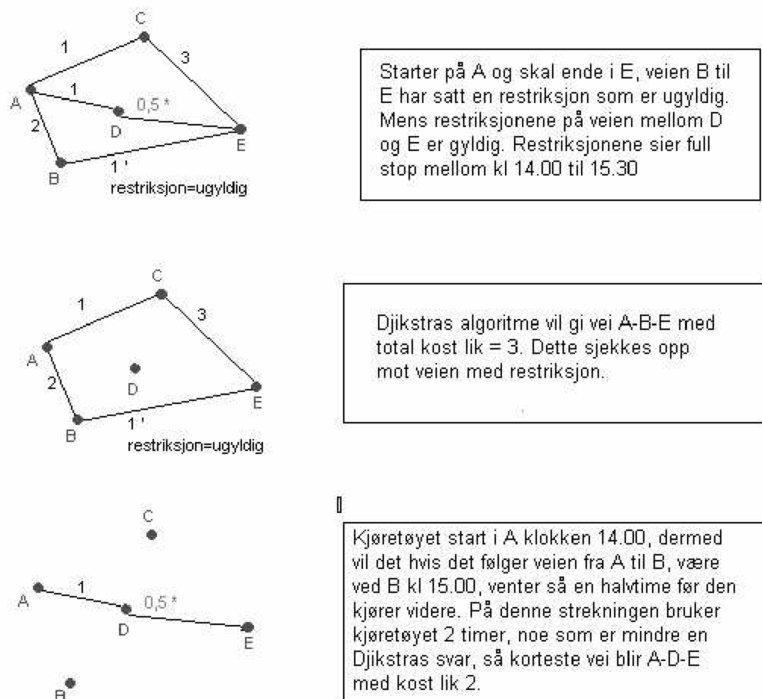
Figur 90 Seilbåts bevegelser

Benytter vi oss av samme algoritme som beskrevet i avsnitt 4.3 vil datamaskinene bruke veldig lang tid på å gi et svar, antall mulige måter en seilbåt kan bevege seg fra A til B på vil være ekstremt

mange og vil gi veldig stor O(I). Det samme ville ha skjedd hvis et veinett bestod av mange veier. En ny og effektiv algoritme må velges.

14.6 Effektiv algoritme

I) Veinett
En kombinasjon av Dijkstras algoritme og en algoritme som sjekker alle mulige veier kan være mer hensiktsmessig enn kun enkel algoritme. I transport eksempelet ser vi først bort fra de veiene som har restriksjoner knyttet til seg og der hvor restriksjonene er ugyldige på grunn av tiden. Deretter finner vi den korteste veien ved hjelp av Djikstras algoritme på disse, før man sjekker om svar er mindre enn alle resterende svar som vi vil få på veier med restriksjoner. Dette kan vises ved hjelp av et lite eksempel hvor et kjøretøy starter fra et sted A klokken 14.00 02.juni og skal kjøre til et sted F. (figur 91)



Figur 91 Beregning av korteste vei

Hvis kjøretøyet derimot hadde startet klokken 13.00 ville det gitt at kosten for strekning A-D-E ville ha blitt 3, og da ville begge svarene vært like. Hvis kjøretøyet derimot hadde kjørt 12.30 ville kjøretøyet komme fra til D klokken 13.30 og ville rukket å kjøre strekning D til E før restriksjonen inntreffer. Konklusjonen er den at hvis et kjøretøy starter i A før 12.30 og etter 13.30 vil restriksjonene på vei D til E bli satt som ugyldig, og man kan kjøre Dijkstras algoritme på hele veinettet, helt frem til restriksjon på vei D til E blir gyldig igjen.

2) Seilas

Ta ut tidene og de sterkeste vindene, regn ut hvor langt båten kan komme slik at det er minst mulig borte fra den rette linjen mellom A og B.

Se om det er mulig å koble disse med hverandre slik at det blir en gjennomgående seiling uten hull. Hvis hull, kan man med hensyn til t legge inn de resterende seilingene med litt mindre vind

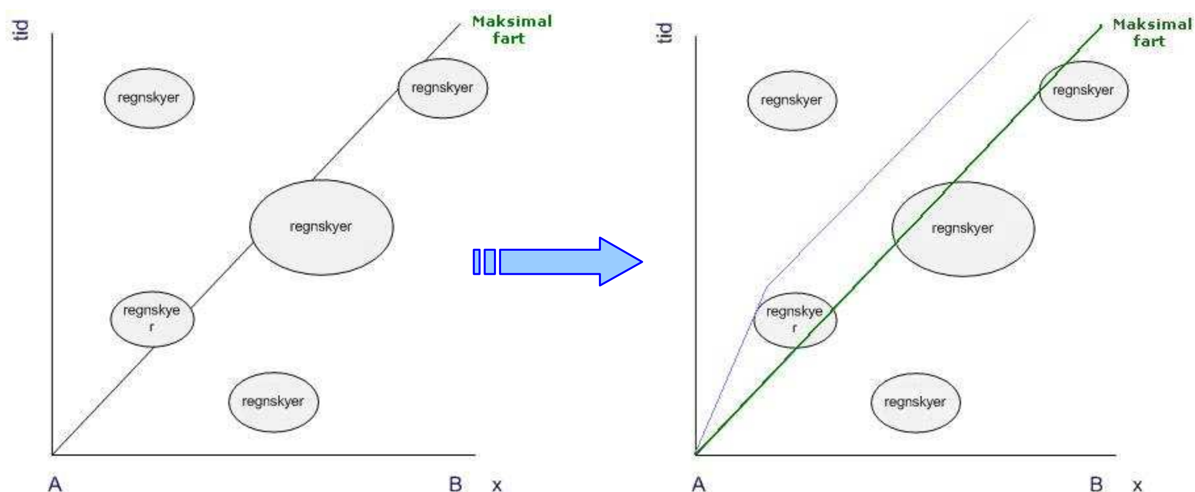
Regn ut rett fra A til B hvor lang tid det med gjennomsnittlig vindkraft vil ta. Dette er kortest ikke-mulige rute som sammenliknes med en intervall for å sjekke hvilke vinder vi kan benytte oss av i det tidsrommet vi seiler i

Kun dijkstras algoritme?

Hvis kun dijkstras algoritme benyttes vil det ved et tidspunkt hvor alle restriksjoner er ugyldig være den mest effektive algoritmen, men ved restriksjoner fører disse til at algoritmen til Dijkstra slår feil. På en annen side kan kanskje utvidelser i Dijkstras algoritme bidra med å løse dette problemet på en mer effektiv måte enn tidligere nevnte løsninger.

14.7 Generalisering av t som attributt til t som en egen akse

La oss se på et enkelt eksempel hvor en person skal sykle fra et sted A til et sted B uten å bli våt. (se figur 92). Regnbyggene blir lagret med sine x og t verdier og det er disse vedkommende skal unngå. Helt enkelt kan man først beregne maksimal farten, her en rett linje fra A til B. Deretter kan man sjekke om det eksisterer noen snittmengde, hvis snittet er lik den tomme mengde er jo den rette linjen den korteste veien uten at personen blir våt. Men hvis det oppstår snitt mellom regnområdene og linjen kan man beregne hvilke vei som blir kortest ved å kjøre på grensen av regnområdene. Som figuren til høyre viser kan syklisten sykle med litt mindre fart til å begynne med slik at den unngår den første regnskyer og deretter sykle med maksimal fart resten av strekningen uten å måtte ta hensyn til andre regnskyer, vi ser også bort fra det faktum at veien kan nå være glatt etter regn.



Figur 92 Optimal rute og regnskyer

Dette kan betraktes å være en generell løsning, men eksemplet vi vurderer i dette avsnittet er enkelt. Algoritmen for å finne korteste vei for veinettet og seilbåten hvor tiden blir betraktet som en egen akse vil bli mer kompleks.

14.7.1 Tog i bevegelse

Objekter som beveger seg i 2D-rommet har en plassering som er avhengig av tid. Vi betrakter punktene som beveger seg i 2D-rommet. Hvis deres plassering bare er kjent i noen instanser av tida, veiene fullt opp av punkter (trajectory) kan bli approksimert med en sekvens av

sammenhengende linjestykker i 3D-rommet hvor aksene er x, y og tiden t. Objekter i bevegelse har noen matematiske egenskaper felles med TIN modeller, hvor en eller flere av variablene kan utrykkes som en lineær funksjon av andre.
En vei fylt opp av et punkt i bevegelse i 2D rommet kan representeres med en mengde av punkter i 3D rommet (x, y, t). Den komplette veien fylt opp av punktet kan beregnes ut av disse punktene ved lineær interpolasjon mellom punktene. Hvis vi nå antar at disse punktene i 3D rommet tilsvarer togstasjoner med plassering x og y, og tidspunktet t for når toget (punktet) er på den stasjonen, og at toget går med konstant hastighet mellom stasjonene, så kan vi se på denne modellen som en modell av en virkelighet. La oss anta vi har fått gitt fire punkter (0,0,0), (10,5,1), (10,10,2) og (20,15,5). Veien fylt opp av toget i bevegelsen med de gitte punktene kan representeres på denne måten:

$$x = 10t \wedge y = 5t \wedge 0 \leq t \leq 1$$

$$\vee$$

$$y = 5t \wedge x = 10 \wedge 5 \leq y \leq 10$$

$$\vee$$

$$3x = 10t + 10 \wedge 3y = 5t + 20 \wedge 2 \leq t \leq 5$$

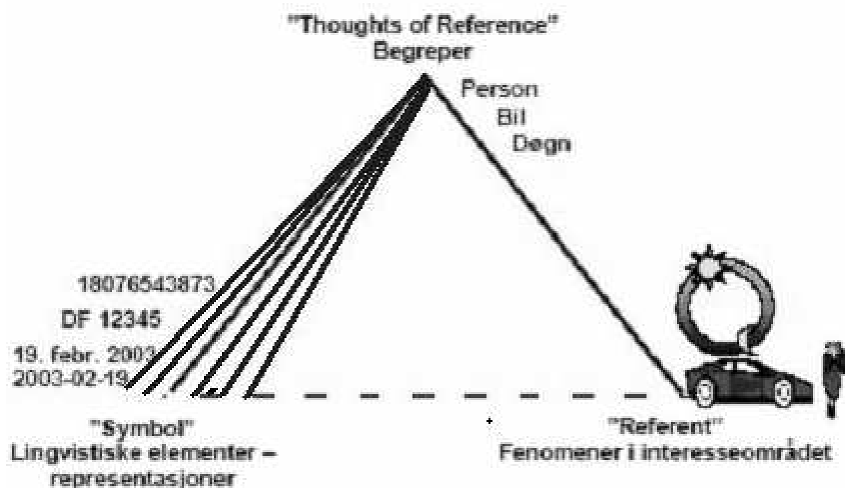
Mer generelt, la oss anta at posisjonen til toget er kjent på mange tidspunkter. Dette definerer et endelig antall tidsintervaller T_k. Hvis farten til toget er konstant gjennom hvert tidsintervall, er togets fart bestemt av $x = v_i\, t + x_i$ og $y = v_j\, t + y_j$, hvor $_i$ og $_j$ er indeksen til intervallet t_k som inneholder t og v_i og v_j er hastigheten til toget på det intervallet, og hvor (x_i, y_j) er valgt passende slik at posisjonen til toget er riktig både i begynnelsen og slutten av intervallet. Da kan den veien som toget følger bli representert i en lineær skrankemodell på følgende måte:

$$Traj(x, y, t) = \vee_{i,j}\, t_k(t) \wedge x = f_k(t) \wedge y = g_k\,(t)$$

der $t_k(t)$ er skranker som definerer tidintervallet T_k og F_k, G_k er lineære funksjoner definert tidligere.

15 Påvirker representasjonen den konseptuelle modellen?

Hvorvidt den konseptuelle modellen skal bli påvirket av representasjonen er et spørsmål som er blitt tatt opp i flere artikler og lenge har vært til diskusjon.(Legg inn en link eller referanse til noen som har diskutert dette emnet) I denne oppgaven har vi kommet frem til at modellen av virkeligheten og hvordan den så blir representert bør være uavhengige av hverandre, spesielt modellen bør være uavhengig av representasjonen, fordi at ved å skjule representasjonsdetaljer fra modellen, kan man få mer robuste modeller. Hvis vi antar at det vi modellerer vil kunne bli representert uten å påvirke modellen vil dette dermed øke sannsynligheten for å få et system som beskriver virkeligheten med stor nøyaktighet. Hvis vi derimot lar representasjon få påvirke vårt syn på virkeligheten, vil dette i verste fall føre til feil eller endringer i konseptualiseringen. Dermed vil et ønske om å kunne modellere virkeligheten så riktig som mulig falle sammen. En ideell modell er den modellen som er totalt uavhengig av representasjonen og som gir programmereren et valg mellom en mengde av representasjonstyper – se figur 93



Figur 93: Figuren ovenfor ilustrerer at modellen bør være totalt uavhengig av representasjonen slik at man kan velge mellom en mengde av reresentasjonstyper.

15.1 Enkel konseptuell modell

Modeller må lages så enkle som mulig for at det skal kunne dannes en felles eller en generell oppfatning som kan nedfelles i en konseptuell modell, som også er et av hovedpoengene med en konseptuell modell.

For å få en vellykket utvikling av et hvilket som helst geografisk informasjonssystemprosjekt trengs det en omhyggelig design og implementasjon av romlige databaser ved å modellere og representere romlige data. Det involverer å forstå den underliggende romlige datamodellen, romlige datatyper og operatorer, romlige spørrespråk og romlige indekseringsteknikker, og bruke og UML og/eller NIAM for modelleringen. All tilgjengelig informasjon relatert til applikasjonen blir organisert ved hjelp av UML eller NIAM-diagrammer. Fokuset er på begreper, deres assosiasjoner og skranker i applikasjonene. Faktiske implementasjonsdetaljer blir holdt utenfor denne modelleringsfasen.
Dermed får man enklere konseptuelle modeller. Ved å ikke bringe inn kompliserende elementer som bare har med representasjonen å gjøre, og som brukere nødvendigvis ikke skjønner, får

man en bedre overblikk og antakelig en riktigere modell. Denne enkle modellen kan brukes av ulike brukere til ulike formål, og ikke bare av programmerere som er kjent med den spesifikke plattformen som modellen skal realiseres på.

Objektorientert programmering er en type programmering der programmerere ikke bare definerer datatypen for en datastruktur, men også typer av operasjoner (funksjoner) som kan anvendes på disse datastrukturene, mens implementasjonsdetaljer av operasjoner er skjult fra brukeren/modellen. På denne måten slipper brukeren å bli overveldet av unødvendige implementasjonsdetaljer og får bare vite det han trenger å vite av detaljer; altså signaturen til funksjonen; for eksempel hva funksjonen heter (det den gjør), inn-parametere til funksjonen, og returverdier fra funksjonen.

Klassediagrammer egner seg godt som modelleringsspråk, fordi det viser bare signaturen til en funksjon. I kapitlet om ”Operasjoner og abstrakte datatyper” har vi vist flere modeller på forskjellige plattformer og alle modellene løser den ene problemstillingen om å finne krysspunktet mellom to togfremføringer. Felles for alle modellene er at ingen av dem viser hvordan funksjonene er implementert, men bare hva de tar imot som parametere og hvilke returverdier de har. Modellene er ulik hverandre på grunn av at de er implementert på ulike plattformer, og alle plattformer har noen begrensninger på hva som er mulig å implementere. Det er for eksempel ikke mulig å implementere noen ”objekter” på den relasjonelle plattformen, og dermed heller ingen funksjoner som opererer på sine egne data som funksjoner i et objekt. Det blir isteden implementert frie prosedyrer som henter ut data fra tabellene og opererer på dem.

De fleste modeller har per dags dato romlige egenskaper hvor tiden ser ut til å kun å bli betraktet som en separat verdi. Vi ser ikke på hvordan fenomener forandrer eller utvikler seg over tid. I denne oppgaven har vi forsøkt å ta med tid som en egen dimensjon og kommet frem til at det kan betraktes på lik linje som ved de romlige dimensjoner. Hvis det er mulig å lage en modell over en virkelighet med romlige verdier, skal det være fullt mulig å gjøre det samme når tiden er med i bildet.

15.2 Muligheter for å bytte representasjon

En modell bør utformes på en slik måte at det er mulig å bytte ut en representasjon mot en annen representasjonstype uten at modellen blir påvirket av dette. Og det er kun mulig hvis den gamle eller eventuelt den første representasjonen ikke er modellert i modellen, men fullstendig utelatt fra modellen. Det vil si modellen at er uavhengig av alle representasjonstyper, og man kan i etterkant velge eller bytte over til en representasjonstype som passer, og eventuelt finne den beste representasjonen ved prøving og feiling.

I nesten alle modellene våre i denne oppgaven har vi implementert objekter som har et attributt eller en kolonne av datatypen *Geometri*. Ingen av objektene har kunnskap om hvilken geometritype de har (om typen blir realisert som en linje, polygon eller punkt), det eneste objektet har kunnskap om, er hvilken datatype hver enkelt kolonne har, og at hvis de er romlige objekter så har de minst en kolonne av datatypen *Geometri.* Dermed står det programmereren fritt å velge en representasjonstype som passer til hans applikasjon.

15.3 Skranker

I kapitlet om ”Realisering av romlige/topologiske skranker” fant vi ut at det var fire måter å realisere en skranke på:

- **ved å realisere skranken i applikasjonsprogrammene:** skranken kan realiseres etter at modellen er blitt realisert, skranken brukes først etter at modellen er blitt implementert og trengs ikke å ta med i datamodellen, fordi det har ingenting direkte med implementasjon av datamodellen å gjøre.
- **deklarere den og håndheve den med "vakthund":** skranken må realiseres før datamodellen kan implementeres, fordi skranken må overholdes ved implementasjon av modellen, skranken kan defineres med OCL i datamodellen.
- **programmere skranken i en trigger:** skranken kan realiseres etter at modellen er blitt realisert, skranken brukes først etter at modellen er blitt implementert og trengs ikke å ta med i datamodellen, den trer i kraft først når man skal oppdatere databasen
- **bygge skranken inn i databasestrukturen:** når skranken er innebygd i databasestrukturen så kommer skranken automatisk inn i datamodellen og trenger ingen eksplisitt modellering av skranken.

Hvilken måte man velger å realisere skranken på, kommer an på hva datamodellen skal brukes til. Hvis datamodellen skal brukes til applikasjoner hvor forhold mellom objektene er viktig, bør skranken bygges inn i databasestrukturen og tas med i datamodellen. Når datamodellen skal brukes til flere formål, er det best å lage en generell datamodell ved å trekke skranken ut av modellen, og bygge skranken inn i enhver applikasjon etter behov.

I kapitel "9 Regning med romlige verdier" ser vi på hvordan diverse algoritmer kan bli benyttet ved romlige og temporale dimensjoner. Vi ser bort fra realiseringen og lar virkeligheten stå til grunn for vårt modellsyn. Eksempelet med veistykker og korteste vei tar opp spørsmål rundt avrunding og de konsekvenser man må ta hensyn til på grunn av diskretisering og unøyaktighet på grunn av begrensninger i den mengden data datamaskinen kan håndtere. Dette løste vi ved å legge inn skranker og ekstra lengder på veiene slik at kryss i virkeligheten kunne også eksistere i datamaskinen. For å få riktig representasjon av modellen kan vi si at man har tatt hensyn og latt modellen bli påvirket av representasjon. Men dette kan ikke sies å være helt riktig siden avrundinger vi gjør i målingene kan tenkes å være bestemt allerede på modellaget. Vi har ikke tenkt på hvordan dette skal bli representert og om modellen skal lages annerledes på grunn av de valg vi gjør når det gjelder valg av representasjonstyper.

15.4 Konklusjon

I kapitel "Eksempler på modeller" drøfter vi hvordan et veinett i virkeligheten skal bli modellert. Vi har ved dette eksempelet tatt i betraktning hvordan evne til å ta med uendelig mengde data er begrenset i datamaskinene. Det var med på å påvirke hvordan modellen vår ble til slutt. Kan dette sies å være representasjonene som har hatt påvirkning på modellen? Nei, dette er en begrensning i datamaskinen som må bli tatt hensyn til uansett hvilke type representasjon vi velger å arbeide videre med.

I avsnitt "3.1 Rommet" kom vi inn på tre forskjellige modeller for tiden, her kom vi litt inn på om valget av tidsmodellen vil påvirke virkeligheten. Vi kom også fram til at virkeligheten følger den lineære tidsmodellen, men vi kan i noen få tilfeller betrakte tiden til å følge en annen tidsmodell en den lineære men må da forholdet oss til en liten del av virkeligheten. Men vi kan heller ikke med sikkerhet si at virkeligheten følger en lineær tidsmodell, flere vitenskapsmenn har påstått at tiden følger en syklisk modell. Men dette er ikke blitt bevist, og kan diskuteres i det uendelige. Hvis en tidsmodell passer for en gitt del av virkeligheten, så vil det ikke bety at resten av virkeligheten også følger den bestemte tidsmodellen.

I kapitel "4.10 Forslag til datamodeller" diskuteres det rundt hvor vidt man skal benytte seg av aggregat modellen eller sammensetningsmodellen, modellen blir her ikke påvirket av representasjonen. Men hva hadde blitt forskjellig hvis vi hadde tatt hensyn til representasjonsvalg? Ville modellene bli noe annerledes hvis vi hadde hatt representasjon i bakhodet? Vi har kommet frem til at det ikke vil ha noe å si, siden vi lar modellen bli basert på virkeligheten og antar at valget av representasjonstyper ikke vil ha mye å si for hvordan modellen vår blir.

Et annet perspektiv som kan tas med i betraktning, er det faktum at vi behandler rom og tid i en og sammen modell. Vil dermed representasjon av disse verdiene påvirket modellen vår? Nei, akkurat som tidligere nevnt, antar vi at representasjon av tiden vil bli håndtert på samme måte som med romlige verdier.

Vi har kommet frem til konklusjonen at Ogdens trekant bør benyttes på en slik måte at virkeligheten påvirker modellen og modellen påvirker representasjon og ikke omvendt. En modell kan ikke endre virkeligheten og dermed bør ikke representasjon føre til endringer på det konseptuelle planet.

15.5 Forbedringsforslag og videre arbeid

Denne oppgaven kunne ha blitt bedre hvis vi tidligere hadde kommet frem til den konklusjon at vi skulle lever en samlet oppgave. Nedenfor har vi listet opp forslag til forbedringsforslag.
I kapitel ”Eksempel på modeller” har vi tatt to problemer og prøvd å komme med en mulig løsning til disse. Vi har en mulig løsning for hvordan man kan generalisere t som attributt til t som egen akse på lik linje som de romlige aksene. Dette har vi ikke programmert, og kan dermed ikke bevise at dette vil kunne løse problemet på maskinnivå. Vi har konsentrert oss rundt modellaget og kommet frem til mulige løsninger rundt dette laget men har ikke hatt anledning til å vise hvordan disse modellene kan bli implementert. Et forslag til videreutvikling vil kunne være å bevise at man kan gå over fra modell til å kunne representere verdier og kjøre programmer som er slike at deres valg av representasjonstyper ikke vil få konsekvenser for modellaget. I tillegg kan man forsøke å løse problemer som er tatt opp i denne oppgaven i høyere dimensjoner slik at man kan få algoritmer som støtter alle mulige antall dimensjoner.

16 VEDLEGG

16.1 Enkel kryss (Versjon 1)

16.1.1 RDBMS – Realsjonell databasehåndteringssytem

```
/*
Versjon1
Oppretting av tabeller
i RDBMS*/

create table Stasjon(
    stasjonNavn varchar(100),
    antKmFraStart number,
    antKmTilNeste number,
    neste varchar(100),
    primary key (stasjonNavn));



create table KryssSpor(
    spornr int,
    kryssSporLengde number,
    kryssSted varchar (100),
    primary key (spornr,kryssSted);


create table Tog(
    togNr int,
    dager varchar(100),
    primary key (togNr);



create table Passering(
    ankomstTid number,
    avgangsTid number,
    togNr int,
    stasjonNavn varchar (100),
    primary key (ankomstTid,togNr,stasjonNavn),
    foreign key (stasjonNavn) references Stasjon(stasjonNavn),
    foreign key (togNr) references Tog(togNr));

commit;

/*
Versjon 1

Finner krysspunkt mellom to tog (som holder konstant fart)
i den relasjonelle databasen, vsd å sette inn algoritmen
i SQL
*/


/* Toget som skal oslo_bergen*/

/* Bruk tegnet '&1' istedenfor nummeret 609 og 62*/


create or replace view B_Start
```

```
as
select antkmfrastart x1, ankomsttid t1
from passering p, stasjon s
where tognr=609 and upper(p.stasjonnavn)=upper('hønefoss')
and s.stasjonnavn=p.stasjonnavn;
/

create or replace view B_End
as
select antkmfrastart x2, ankomsttid t2
from passering p, stasjon s
where tognr=609 and upper(p.stasjonnavn)=upper('Bergen')
and p.stasjonnavn=s.stasjonnavn;
/

/*Toget som skal bergen-oslo */

create or replace view O_Start
as
select antkmfrastart x3,ankomsttid t3
from passering p, stasjon s
where tognr=62 and upper(p.stasjonnavn)=upper('Bergen')
and  p.stasjonnavn=s.stasjonnavn;


create or replace view O_end
as
select antkmfrastart x4, ankomsttid t4
from passering p, stasjon s
where tognr=62 and upper(p.stasjonnavn)=upper('Hønefoss')
and p.stasjonnavn=s.stasjonnavn;


create or replace view a_b
as
select (x1-x2)/(t1-t2) a1,((x2*t1)-(x1*t2))/(t1-t2) b1,
       (x3-x4)/(t3-t4) a2,((x4*t3)-(x3*t4))/(t3-t4) b2
from B_Start, B_End, O_end, O_Start;


create or replace view t_0
as
select (b2-b1)/(a1-a2) t0
from a_b;


create or replace view x_0
as
select a1*t0+b1 x0
from t_0,a_b;


select 'krysspunkt',x0,t0
from x0,t0
where x0>0 and x0<475 and t0<0 and t0<2400;

commit;

/*Versjon 1
 *
 * Dette Jdbc programmet finner krysspunktet
```

```
 * mellom tog data lagret i den relasjonelle databasen,
 * altså finner krysspunktet mellom to tog (som holder konstant fart
 * */

import java.sql.*;
import oracle.jdbc.driver.*;
import easyIO.*;

class RelasjonellKall{


    static int bergen,oslo;
    static Passord p=new Passord();
    static In tast=new In();


    private static Statement getStatement() throws Exception {

      DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
      Connection conn=DriverManager.getConnection
          ("jdbc:oracle:thin:@delphinium.ifi.uio.no:1521:IFIORA",p.user,
           p.pwd);
      return conn.createStatement();
    }



    public static void main (String args[]) throws Exception
    {    int valg;

      System.out.println(" ");
      System.out.println("*******************************************");
      System.out.println("Tog til Oslo s: 62 -602 ");
      System.out.println("-------------------------------------------");
      System.out.println("Tog til Bergen: 609 -61 ");
      System.out.println("*******************************************");
      System.out.println(" ");

      do {

          System.out.println("**************Togoversikt************");
          System.out.println("1. Finn togruten");
          System.out.println("2. Finn Krysspunktet");
          System.out.println("3. Avslutt");
          valg = tast.inInt();

          switch(valg){

          case 1:
            finnTogRute();
            break;

          case 2:

            finnKryss();
            break;

          case 3:
            System.out.println("Du har avsluttet programmet");
            break;
```

```
      default:
        System.out.println("Feil kommando! Prøv igjen");
      }
  }while (valg != 3);


}//main


static void finnKryss(){

  System.out.println("Oppgi tognr til toget fra Oslo s til Bergen: ");
  bergen=tast.inInt();
  System.out.println ("Oppgi tognr til toget fra Bergen til Oslo s: ");
  oslo=tast.inInt();
  System.out.println("");
  finnTogene(bergen,oslo);
}//end finnKryss

static void finnTogRute()throws Exception{

  System.out.println("Tast inn et av tognummerene over " +
                "for å få en oversikt over ruten til toget.");
  System.out.println("Tast 0 for å avslutte!" );
  System.out.println("*************************************");
  System.out.print("Tognr: ");
  int nr=tast.inInt();
  while(nr!=0){
      finnRuten(nr);
      System.out.print("Tognr: ");
      nr=tast.inInt();

  }
}//end togrute





static void finnTogene(int bergen,int oslo){
  try{
      int lengde1=0,lengde2=0,i=0,j=0;
      Statement stmtantb =getStatement();
      Statement stmtanto =getStatement();

      ResultSet rsetantb=stmtantb.executeQuery
        ("select count(*)as ant from passering where tognr="+bergen);
      while (rsetantb.next()){
        lengde1=rsetantb.getInt("ant");
      }
      double []tidb=new double [lengde1];
      double []stedb=new double [lengde1];

      ResultSet rsetanto=stmtanto.executeQuery
        ("select count(*)as ant from passering where tognr="+oslo);
      while (rsetanto.next()){
        lengde2=rsetanto.getInt("ant");
      }
      double []tido=new double [lengde2];
      double []stedo=new double [lengde2];
```

```
  Statement stmtb =getStatement();
  Statement stmto =getStatement();

  /*"b" betyr toget som går fra oslo til bergen det vil si dette
   *resultsettet inneholder
   *informasjon om toget som går fra oslo til bergen, og
   * "o" betyr det motsattet
   */

  ResultSet rsetb=stmtb.executeQuery
      ("select ankomstTid,antKmFraStart "+
       "from Passering p,Stasjon s "+
       "where s.stasjonNavn=p.stasjonNavn and p.tognr= "+bergen);

  ResultSet rseto=stmto.executeQuery
      ("select ankomstTid,antKmFraStart "+
       "from Passering p,Stasjon s "+
       "where s.stasjonNavn=p.stasjonNavn and p.tognr= "+oslo);


  while (rsetb.next()){
      tidb[i]= rsetb.getDouble("ankomstTid");
      stedb[i]=rsetb.getDouble("antKmFraStart");
      i++;
  }


  while (rseto.next()){
      tido[j]=rseto.getDouble("ankomstTid");
      stedo[j]= rseto.getDouble("antKmFraStart");
      j++;
  }


  finnKrysspunkt
      (tido[0],stedo[0],tido[tido.length-1],stedo[stedo.length-1],
       tidb[0],stedb[0],tidb[tidb.length-1],stedb[stedb.length-1]);

  System.out.println(" ");

  }catch (SQLException e){
      e.printStackTrace();
  }catch (Exception t){
      System.out.println("Feil : " +t);
  }

}//finnTogne




static void finnKrysspunkt(double t1, double x1,double t2, double x2,
                     double t3,double x3,double t4, double x4)
{

  double a1,b1,a2,b2,t0,x0;
```

```
a1=(x1-x2)/(t1-t2);
b1=((x2*t1)-(x1*t2))/(t1-t2);

a2=(x3-x4)/(t3-t4);
b2=((x4*t3)-(x3*t4))/(t3-t4);

t0=(b2-b1)/(a1-a2);
x0=a1*t0+b1;

if(x0>0 && t0>0){
    if(471.25>x0 && 2400>t0){

      System.out.println("Krysspunktet for tognr "+bergen+
                   "og tognr " +oslo+" : ");
      System.out.println("tid:    " +t0 );
      System.out.println("sted;    " +x0 +" km fra Oslo s");
    }else
      System.out.println("Det finnes ingen krysspunkt mellom" +
                    " disse to togene");
}else

    System.out.println("Det finnes ingen krysspunkt mellom" +
                  " disse to togene");

}//finnkrysspunkt



static void finnRuten(int n)throws SQLException {
  try{
      Statement stmt =getStatement();
      ResultSet rs= stmt.executeQuery( "select stasjonnavn,ankomsttid "+
                            "from passering where tognr="+n);
      while(rs.next()){
        System.out.print(rs.getString("stasjonnavn"));
        System.out.println("  kl: "+rs.getDouble("ankomstTid"));
      }
      System.out.println("***************************************");
  }catch(Exception e){System.out.println("feil  " +e); }
}//finnruten


}//RelasjonellKall
```

16.1.2 OO- Objekt orientert

```
/*Versjon 1
 *
 * Tog objekter i det objekt orinterte programmet,
 * med algoritmen sydd inn i objektene for å finne
 * krysspunktet mellom to tog (som holder konstant fart)
 * */

import java.sql.*;
import oracle.sql.*;
import javax.swing.*;
import oracle.jdbc.driver.*;
import easyIO.*;
import java.util.*;
```

```
class AlleTog {


    public  static HashMap alleStasjoner = new HashMap();
    public static HashMap alleTog = new HashMap();
    static In tast=new In();
    static Passord p=new Passord();

    public static Statement getStatement() throws Exception {

      DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
      Connection conn=DriverManager.getConnection
          ("jdbc:oracle:thin:@delphinium.ifi.uio.no:1521:IFIORA",
           p.user,p.pwd);
      return conn.createStatement();
    }//statement


    public static void main (String args[]) throws Exception {

      int valg;
      initialiserTogene();
      lagStasjoner();

      System.out.println(" ");
      System.out.println("*******************************************");
      System.out.println("Tog til Oslo s: 62 -602 ");
      System.out.println("-------------------------------------------");
      System.out.println("Tog til Bergen: 609 -61 ");
      System.out.println("*******************************************");
      System.out.println(" ");
      System.out.println(" ");


      do {

          System.out.println("**************Togoversikt************");
          System.out.println("1. Finn Togruten");
          System.out.println("2. Finn Krysspunktet");
          System.out.println("3. Avslutt");
          valg = tast.inInt();

          switch(valg){

          case 1:
            finnRuten();
            break;

          case 2:

            finnKrysspunktet();
            break;

          case 3:
            System.out.println("Du har avsluttet programmet");
            break;

          default:
            System.out.println("Feil kommando! Prøv igjen");
          }
```

```
  }while (valg != 3);


}//main



static void initialiserTogene(){
  try{
      Statement stmt =getStatement();
      ResultSet rs= stmt.executeQuery( "select * from Tog ");
      Tog tog=null;
      while(rs.next()){
        tog = new Tog(rs);
        alleTog.put(tog.tognr,tog);
      }//slutt while

      Iterator it = alleTog.values().iterator();
      while(it.hasNext()){
        tog= (Tog) it.next();
        tog.lagLine();
      }

  }catch(Exception e){System.out.println("feil  " +e); }

}// end init




static void lagStasjoner(){
  try{
      Statement stmt =getStatement();
      ResultSet rs= stmt.executeQuery
        ("select stasjonnavn, antkmfrastart "+
         "from  Stasjon s ");
      Stasjon s= null;
      while(rs.next()){
        s=new Stasjon(rs);
        alleStasjoner.put(s.stasjonnavn,s);
      }
  }catch(Exception e){System.out.println("feil  " +e); }
}//lagline




static void finnKrysspunktet (){
  String bergen,oslo;
  Tog t1=null,t2=null;
  System.out.println("Oppgi tognr til toget fra Oslo s til Bergen: ");
        bergen=tast.inWord();
  if(alleTog.containsKey(bergen))
      t1=(Tog)alleTog.get(bergen);
  System.out.println ("Oppgi tognr til toget fra Bergen til Oslo s: ");
        oslo=tast.inWord();
  if(alleTog.containsKey(oslo))
      t2=(Tog)alleTog.get(oslo);
  System.out.println("");
  Point p= t1.finnKrysspunktet(t2);
  double x0=p.getX();
```

```
      double t0=p.getT();

      if(x0>0 && t0>0){
          if(471.25>x0 && 2400>t0){

            System.out.println("Krysspunktet for tognr "+ bergen +
                               " og tognr "+ oslo );
            System.out.println("tid:    " +t0 );
            System.out.println("sted:    " +x0 +" km fra Oslo s");
          }else
            System.out.println("Det finnes ingen krysspunkt mellom" +
                               " disse to togene");
      }else

          System.out.println("Det finnes ingen krysspunkt mellom" +
                             " disse to togene");
      System.out.println("");
    }//Krysspunktet;



    static void finnRuten() {

      System.out.println("--------------------------------------------");
      System.out.println("Tast inn et av tognummerene over " +
                         "for å få en oversikt over ruten til toget.");
      System.out.println("Tast 0 for å avslutte avslutte!" );
      System.out.println("--------------------------------------------");
      System.out.print("Tognr: ");
      Tog t;
      String nr=tast.inWord();
      while(nr==null || !nr.equals("0")){
          t=(Tog)alleTog.get(nr);
          if( t!= null){
            t.finnRuten(alleStasjoner);
          }else
            System.out.println("Feil tognr, prøv igjen !");

          System.out.print("Tognr: ");
          nr=tast.inWord();
          System.out.println("--------------------------------------------
");

      }//while

      }//finnRuten

}//AlleTog




class Tog {

    String tognr;
    String dager;
    final Line l= new Line();

    Tog (ResultSet row) throws Exception {
      tognr=row.getString( "tognr");
      dager=row.getString("dager");
```

```
     }

     String getTognr(){
       return tognr;
     }



     void finnRuten(HashMap liste){
       Point [] xt=l.getPoints();
       for(int j=0; j<xt.length;j++){
           if(xt[j]!=null){
             Iterator r = liste.values().iterator();
             while(r.hasNext()){
                 Stasjon s= (Stasjon) r.next();
                 if(s.antkmfrastart==xt[j].x)
                   System.out.println(s.stasjonnavn+"   tid: " +xt[j].t);
             }//while
           }//if
       }//for
     }//finnruten


    Point finnKrysspunktet(Tog t2){
       Point p= l.finnKrysspunktet(t2.l);
       return p;
    }



     void  lagLine(){
       try{
           int lengde=0;
           Statement stmtant=AlleTog.getStatement();
           ResultSet rsetant=stmtant.executeQuery
             ("select count(*)as ant from passering where tognr="+tognr);

           while (rsetant.next()){
             lengde=rsetant.getInt("ant");
           }
           l.deklArray(lengde);

           Statement stmt = AlleTog.getStatement();
           ResultSet rs= stmt.executeQuery
             ("select ankomsttid, antkmfrastart "+
              "from Passering p, Stasjon s " +
              "where p.stasjonnavn=s.stasjonnavn and tognr=" +tognr);

           while(rs.next()){
             l.leggTil(rs);
           }
       }catch(Exception e){System.out.println("feil  " +e); }
     }//lagline

}//Tog



class Line {
      int i;
      Point [] xt;
```

```
    void deklArray(int lengde){
      xt=new Point[lengde];

    }

    void leggTil(ResultSet row) throws Exception {
      xt[i]=new Point(row);
      i++;
    }

    Point [] getPoints(){
      return xt;
    }//getLine



    Point finnKrysspunktet(Line l2) {

      double a1,b1,a2,b2,t0,x0;

      Point [] xt2=l2.getPoints();

      a1=(xt[0].x-xt[xt.length-1].x)/(xt[0].t-xt[xt.length-1].t);
      b1=((xt[xt.length-1].x*xt[0].t)-(xt[0].x*xt[xt.length-1].t))/
          (xt[0].t-xt[xt.length-1].t);

      a2=(xt2[0].x-xt2[xt2.length-1].x)/(xt2[0].t-xt2[xt2.length-1].t);
      b2=((xt2[xt2.length-1].x*xt2[0].t)-(xt2[0].x*xt2[xt2.length-1].t))/
          (xt2[0].t-xt2[xt2.length-1].t);

      t0=(b2-b1)/(a1-a2);
      x0=(a1*t0)+b1;

      Point p0=new Point(t0,x0);
      return p0;

    }//krysspunkt


}//Line


class Point {

    final double t,x;

    public Point (ResultSet row) throws SQLException{
      t=row.getDouble( "ankomsttid");
      x=row.getDouble("antkmfrastart");

    }

    public Point(double t, double x){
      this.t=t;
      this.x=x;
    }

    double getT(){
```

```
      return t;
    }

    double getX (){
      return x;
    }

}//point


class Stasjon {

    String stasjonnavn;
    double antkmfrastart;

    public Stasjon (ResultSet row) throws SQLException{
      stasjonnavn=row.getString( "stasjonnavn");
      antkmfrastart=row.getDouble("antkmfrastart");

    }

}//Stasjon
```

16.1.3 ORDBMS – Objektrealsjonell databasehåndteringssytem

```
/*Versjon 1
 *
 * Dette programmet implementerer ADT'r i et objektrelasjonell database
 * for å finne krysspunkt mellom to tog (som har konstant fart)
 * */

import java.sql.*;
import oracle.sql.*;
import javax.swing.*;
import oracle.jdbc.driver.*;
import easyIO.*;
import java.util.*;



public class OrdbmsDataModell {

    static Passord p=new Passord();
    public static  Statement stmt;
    public static Statement getStatement() throws Exception {

      DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
      Connection conn=DriverManager.getConnection
          ("jdbc:oracle:thin:@delphinium.ifi.uio.no:1521:IFIORA",
           p.user,p.pwd);
      return conn.createStatement();
    }//statement


    public static void main (String args[]) throws Exception {

      try {
          stmt =getStatement();
```

```
        /*sletter alle tabeller og typer i databasen hvis de eksisterer i
          databasen*/
        stmt.execute("drop table AlleTog");
        stmt.execute("drop type togtype");
        stmt.execute("drop type linetable");
        stmt.execute("drop type linetype");
        stmt.execute("drop type ptable");
        stmt.execute("drop type pointtype");


       /*Oppretter alle tabeller og type objekter med funksjoner*/

        stmt.execute
          ("CREATE type pointtype AS OBJECT" +
           "(x NUMBER, t NUMBER)");

        stmt.execute("CREATE  type ptable as table of pointtype");

        stmt.execute("CREATE type linetype as object(lpoints ptable, " +
                 "member function finnKrysspunktet (l linetype) " +
                 "return pointtype)");


        stmt.execute("CREATE type linetable as table of linetype");

        stmt.execute("CREATE type togtype as object ( " +
                 "tognr NUMBER, dager VARCHAR(100),line linetable, " +
                 "member function krysspunkt (t togtype) " +
                 "return pointtype)");



        stmt.execute("CREATE table alleTog of togtype "  +
                 "nested table line store as n_line ( " +
                 "nested table lpoints store as n_lpoints)");


        /*kaller på metoden som skal initialisere tog objektene*/

        lagTogObjekter();

    } catch (SQLException e) {

        e.printStackTrace();
    }


  }//main



  /* henter ut verdiene fra relasjonsdatabasen og oppretter
objektdatabasen*/

  static void lagTogObjekter() throws Exception{
    try{
        Statement st =getStatement();
        /*henter ut alle verdiene/attributtene fra relasjonsdatabasen
          min som jeg trenger for å initialisere objektene jeg har laget:
          tognr, dager , antkmfrastart(x),ankomsttid(t)*/
```

```
ResultSet r= st.executeQuery
  ("select p.tognr,dager,s.antKmFraStart,ankomstTid "+
   "from Passering p,Stasjon s,tog t "+
   "where s.stasjonNavn=p.stasjonNavn and p.tognr=t.tognr") ;

String sql,dag; int nr=0,i,j=0; double km=0, t=0;

/* Dette arrayet bruker jeg som et sjekk for å finne ut om et
   togobjekt eksisterer i databasen, hvis den finnes så
   skal det ikke lages noe nytt togobjekt, men bare
   legge inn et punkt for toglinjen*/

int[] allenr= new int [10];

while(r.next()){
  i=0;
  sql=null;
  nr=r.getInt("tognr");
  dag=r.getString("dager");
  km=r.getDouble("antKmfrastart");
  t=r.getDouble("ankomsttid");
  /*henter ut tognumrene fra togobjektene hver gang det skal
    legges inn et nytt rad fra relasjonsdatabasen, og*/

  ResultSet r2=stmt.executeQuery("select tognr from AlleTog");

  /* initialiserer test arrayet*/
  while(r2.next()){
      allenr[i]=r2.getInt("tognr");
      i++;
  }
  /*sjekker om jeg har laget togobjektet med det
  tog nummeret som kommer fra relasjonsdatabasen nå*/
  boolean gnr=false;
  for (int k=0; k<i;k++){
      if(allenr[k]==nr)
        gnr=true;
  }
  /*Hvis det er absolutt den første raden som kommer fra
    relasjonsdatabasen da skal denne if testen slå ut*/
  if(j<1) {
      sql = "insert into AlleTog values ";
      sql+= "("+nr+",'"+dag+"', ";
      sql+= "linetable(linetype(ptable(pointtype ";
      sql+= "("+km+","+t+")))))";
  }
  /*hvis ikke togobjektet eksiterer fra før av da skal denne
    if testen slå ut og lage et nytt tog objekt*/
  else if(!gnr){
      sql ="insert into AlleTog values ";
      sql+= "("+nr+",'"+dag+"', ";
      sql+= "linetable(linetype(ptable(pointtype ";
      sql+= "("+km+","+t+")))))";
  }
  /* Hvis tog objektet finnes i databasen og det er ikke
     den første raden fra relasjonsdatabasen da må det legges
     inn et nytt punkt for toglinjen med det tognummeret
     som kommer inn nå (må gjøre en oppdatering)*/
  else{
      sql =" insert into table(select k.lpoints " ;
```

```
                sql+= "from table(select t.line from AlleTog t ";
                    sql+=" where t.tognr =" +nr+")"+"k )";
                sql+=" values ("+km+","+t+")";

            }
            stmt.executeUpdate(sql);
            j++;

          }//while

      } catch (SQLException e) {

          e.printStackTrace();
      }

    }//lagtogobjekter

}//OrdbmsDataModell




/*
Versjon 1

Funksjoner som ble implementert for
objekter i den objektrelasjonelle databasen
for å finne krysspunkt mellom to tog
(som holder konstant fart)

*/

create or replace type body linetype as
member function finnkrysspunktet (l linetype) return pointtype
is a1 number; b1 number; a2 number; b2 number; t0 number; x0 number;

type point_tab is  table of number index by binary_integer ;
x1 point_tab;
t1 point_tab;
x2 point_tab;
t2 point_tab;
p0 pointtype;

Begin

FOR element IN 1..lpoints.COUNT
LOOP
   x1(element):= ( lpoints(element).x );
   t1(element):= ( lpoints(element).t );
END LOOP;

FOR element IN 1..l.lpoints.COUNT
LOOP
   x2(element):= ( l.lpoints(element).x );
   t2(element):= ( l.lpoints(element).t );
END LOOP;



a1:=(x1(x1.first)-x1(x1.last))/(t1(x1.first)-t1(x1.last));
b1:=((x1(x1.last)*t1(x1.first))-(x1(x1.first)*t1(x1.last)))/
```

```
    (t1(x1.first)-t1(x1.last));

a2:=(x2(x2.first)-x2(x2.last))/(t2(x2.first)-t2(x2.last));
b2:=((x2(x2.last)*t2(x2.first))-(x2(x2.first)*t2(x2.last)))/
    (t2(x2.first)-t2(x2.last));

t0:=(b2-b1)/(a1-a2);
x0:=(a1*t0)+b1;

if x0>0 and t0>0 and 471.25>x0 and 2400>t0
then
p0:= NEW pointtype(x0,t0);
else
p0:= NEW pointtype(-1,-1);
end if;

return p0;

end finnkrysspunktet;

End;
/


create or replace type body togtype as
member function krysspunkt (t togtype)
return pointtype is
point pointtype;
i  BINARY_INTEGER;
j  BINARY_INTEGER;

Begin

i := t.line.FIRST;
j := self.line.FIRST;
point:=self.line(j).finnkrysspunktet(t.line(i));

return point;

end krysspunkt;

End;
/

create or replace function
finn_togene(tnr1 number,tnr2 number)
return pointtype is

cursor t1 is
select value(t)
from AlleTog t where tognr=tnr1;

cursor t2 is
select value(t)
from AlleTog t where tognr=tnr2;

point pointtype;
tog1 togtype;
tog2 togtype;

begin
```

```
OPEN t1;
   FETCH t1 INTO tog1;
close t1;

OPEN t2;
   FETCH t2 INTO tog2;
close t2;

point:=tog1.krysspunkt(tog2);

return point;
end;
/
```

```
/*
Versjon 1

Denne blokken kaller på funksjonen
finn_togene for å finne krysspunktet
mellom to tog (som har konstant fart). Funksjonen kaller
på member funksjoner til objekter i den objekt relasjonelle
databasen */
DECLARE
   p    pointtype;
   tog_nr_1 number:=609;
   tog_nr_2 number:=62;
BEGIN
   p:=finn_togene(tog_nr_1,tog_nr_2);

if  p.x=-1 and p.t= -1
 then
  dbms_output.put_line('Ingen krysspunkt');
else
   dbms_output.put_line('X-verdi'|| ' ' ||p.x);
   dbms_output.put_line('T-verdi'|| ' ' ||p.t);
end if;

END;
/
```

```
/*
Versjon 1

Dette Jdbc programmet kaller også på funksjonen
finn_togene i objektrelasjonelle databasen
for å finne krysspunktte mellom
to tog (som holder konstant fart
*/

import oracle.sql.STRUCT;
```

```
import java.sql.*;
import oracle.sql.*;
import javax.swing.*;
import oracle.jdbc.driver.*;
import easyIO.*;
import java.util.*;



public class Prosedyrekall {


    public static Connection conn;
    public static In tast=new In();
    public static  Statement stmt;
    static Passord p=new Passord();

    public static void main (String args[]) throws SQLException {

      DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
      conn=DriverManager.getConnection
          ("jdbc:oracle:thin:@delphinium.ifi.uio.no:1521:IFIORA",
           p.user,p.pwd);

      try {
          System.out.println(" ");
          System.out.println("*******************************************");
          System.out.println("Tog til Oslo s: 62 -602 ");
          System.out.println("-------------------------------------------");
          System.out.println("Tog til Bergen: 609 -61 ");
          System.out.println("*******************************************");
          System.out.println(" ");
          System.out.println(" ");

          System.out.println("Oppgi tognr til toget fra Oslo til Bergen: ");
          int Otog=tast.inInt();
          System.out.println("Oppgi tognr til toget fra Bergen til Oslo: ");
          int Btog=tast.inInt();
          System.out.println(" ");

          // Declare that the first ? is a return value of type struct
          String query= "begin ? := finn_togene(?,?); end;";
          CallableStatement cstmt = conn.prepareCall(query);
          cstmt.registerOutParameter (1, Types.STRUCT,"POINTTYPE" );

          cstmt.setInt(2, Otog);
          cstmt.setInt(3, Btog);
          cstmt.execute();
          Struct punkt = (Struct)(cstmt.getObject(1));
          Object[] attrs = punkt.getAttributes();
          System.out.println("Sted(x) : " + punkt.getAttributes()[0]);
          System.out.println("Tid(t)  : " + punkt.getAttributes()[1]);
          cstmt.close();
          conn.close();

      } catch (SQLException e) {

          e.printStackTrace();
      }catch (Exception k){}
```

```
    }//main

}// class prosedyrekall
```

16.2 Avansert kryss (Versjon 2)

16.2.1 RDBMS – Realsjonell databasehåndteringssytem

```
/* Versjon 2

Datamodellen i rdbms for å finne
krysspunkt mellom to tog (som ikke holder
konstant fart)
*/

drop table Rute;
drop table Geometry_typer;
drop table Linjestykker;
drop table Linje;
drop table Punkt;

create table  Punkt(
   punkt_id  number,
   x         varchar2(100),
   t         number,
   unique (x,t),
   primary key (punkt_id),
   foreign key(x) references Stasjon(stasjonnavn));

/*for tognr 609*/
insert into  punkt values(01,'Hønefoss',747);
insert into  punkt values(02,'Haugastøl',1005);
insert into  punkt values(03,'Haugastøl',1013);
insert into  punkt values(04,'Finse',1033);
insert into  punkt values(05,'Finse',1039);
insert into  punkt values(06,'Myrdal',1107);
insert into  punkt values(07,'Myrdal',1114);
insert into  punkt values(08,'Dale',1244);
insert into  punkt values(09,'Arna',1317);
insert into  punkt values(10,'Arna',1324);
insert into  punkt values(11,'Bergen',1334);


/*for tognr 62*/
insert into  punkt values(12,'Bergen',0758);
insert into  punkt values(13,'Myrdal',0948);
insert into  punkt values(14,'Myrdal',0951);
insert into  punkt values(15,'Hønefoss',1310);


create table  Linje(
   linje_id    number
   type  number,
```

```
   check (line_type) in(1,2,3),
   primary key (linje_id));


insert into  linje values(2101,1);
insert into  linje values(2102,1);


create table  Linjestykker(
    linje_id    number,
    rekke_nr    number,
    startpunkt  number,
    type        number,
    check (type in (1,2)),
    primary key (linje_id,rekke_nr),
    foreign key (linje_id) references  Linje(linje_id),
    foreign key (startpunkt) references  Punkt(punkt_id));

/*type-kolonnen kan være rett eller sirkulær*/

/*2101 betyr to dimensjonalt-rettlinje- og tallet 1 er id */
/*22** betyr to dimensjonalt-sirkulærlinje*/
/*1, betyr rett, 2 betyr sirkulær, null betyr ingen linje strter fra dette
punktet og et dette er siste punktet for denne linjen og det er ikke noen
linje som starter fra dette punktet*/

/*for tognr 609*/
insert into  linjestykker  values(2101,01,01,1);
insert into  linjestykker  values(2101,02,02,1);
insert into  linjestykker  values(2101,03,03,1);
insert into  linjestykker  values(2101,04,04,1);
insert into  linjestykker  values(2101,05,05,1);
insert into  linjestykker  values(2101,06,06,1);
insert into  linjestykker  values(2101,07,07,1);
insert into  linjestykker  values(2101,08,08,1);
insert into  linjestykker  values(2101,09,09,1);
insert into  linjestykker  values(2101,10,10,1);
insert into  linjestykker  values(2101,11,11,null);


/*for tognr 62*/
insert into  linjestykker  values(2102,01,12,1);
insert into  linjestykker  values(2102,02,13,1);
insert into  linjestykker  values(2102,03,14,1);
insert into  linjestykker  values(2102,04,15,null);

create table Geometri_typer(
   type_id  number primary key,
   type_navn varchar2(100));

insert into  Geometri_typer values(01,'PUNKT');
insert into  Geometri_typer values(02,'LINJE\KURVE');

create table Geometri(
   geometri_id  number primary key,
   type_id      number,
   foreign key (type_id) references  Geometri_typer(type_id));

insert into  Geometri values(2101,02);
insert into  Geometri  values(2102,02);
```

```
create table  Rute(
    togNr          number,
    linje_id       number,
    primary key (linje_id,tognr),
    foreign key (linje_id) references  Geometri(geometri_id),
    foreign key (togNr) references Tog(togNr));


/*tognr 609 og linjestykkens id */
insert into  rute values(609,2101);

/*tognr 62 og linjestykkens id*/
insert into rute values(62,2102);



commit;

/*Versjon2

Prosedyren i rdbms som finner
krysspunkt mellom to tog
(som ikke holder konstant fart)
*/

create or replace procedure rdbms_krysspunkt(nr1 in number,nr2 in number,t
out number, x out number)
as

cursor tog1_cur is
   select antkmfrastart x1,t t1,p.x2, p.t2,l1.linje_id l,l1.rekke_nr nr
   from stasjon s1,linjestykker l1,punkt p1,linje lj1,
        ( select antkmfrastart x2,t t2,l2.linje_id l,l2.rekke_nr lnr
          from stasjon s2,linjestykker l2,punkt p2
          where l2.linje_id=nr1 and
                p2.punkt_id=l2.startpunkt and
                s2.stasjonnavn=p2.x ) p
   where l1.linje_id=p.l and
          s1.stasjonnavn=p1.x and l1.startpunkt=p1.punkt_id
          and l1.rekke_nr=(p.lnr-1) and
          lj1.type=1 and lj1.linje_id=l1.linje_id;



cursor tog2_cur is
   select antkmfrastart x3,t t3,p.x4, p.t4,l1.linje_id l,l1.rekke_nr nr
   from stasjon s1,  linjestykker l1,punkt p1,linje lj1,
        ( select antkmfrastart x4,t t4,l2.linje_id l,l2.rekke_nr lnr
          from stasjon s2,linjestykker l2,punkt p2
          where l2.linje_id=nr2 and
                p2.punkt_id=l2.startpunkt and s2.stasjonnavn=p2.x) p
   where  l1.linje_id=p.l and
          s1.stasjonnavn=p1.x and l1.startpunkt=p1.punkt_id
          and l1.rekke_nr=(p.lnr-1)and
          lj1.type=1 and lj1.linje_id=l1.linje_id;





tog1_rec tog1_cur%rowtype;
```

```
tog2_rec tog2_cur%rowtype;
a1 number;  b1 number;
a2 number;  b2 number;
ant_kryss number:=0;
t0 number; x0 number;
begin

open tog1_cur;
   loop
   fetch tog1_cur into tog1_rec;
   exit when tog1_cur%notfound;
   open tog2_cur;
   loop
   fetch tog2_cur into tog2_rec;
   exit when tog2_cur%notfound;



   a1:=(tog1_rec.x1-tog1_rec.x2)/(tog1_rec.t1-tog1_rec.t2);
   b1:=((tog1_rec.x2*tog1_rec.t1)-(tog1_rec.x1*tog1_rec.t2))/
                (tog1_rec.t1-tog1_rec.t2);

   a2:=(tog2_rec.x3-tog2_rec.x4)/(tog2_rec.t3-tog2_rec.t4);
   b2:=((tog2_rec.x4*tog2_rec.t3)-(tog2_rec.x3*tog2_rec.t4))/
       (tog2_rec.t3-tog2_rec.t4);

  if(a1-a2)>0
   then
     t0:=(b2-b1)/(a1-a2);
     x0:=(a1*t0)+b1;

  end if;

 if(a1-a2)>0
   then
     t0:=(b2-b1)/(a1-a2);
     x0:=(a1*t0)+b1;
  end if;


  if tog1_rec.t1<=t0 and t0<=tog1_rec.t2
     and tog2_rec.t3<=t0 and t0<=tog2_rec.t4
    then
     if tog1_rec.x1<=x0 and x0<=tog1_rec.x2
        and tog2_rec.x3>=x0 and x0>=tog2_rec.x4
       then
         ant_kryss:=ant_kryss+1;
         t:=t0;
         x:=x0;
     end if;
   end if;

end loop;

close tog2_cur;
end loop;
close tog1_cur;

if ant_kryss=0
  then
```

```
  t:= -1;
  x:= -1;

end if;

end rdbms_krysspunkt;
/

/*Versjon 2

Blokken som kaller prosedyren i rdbms
for å finne krysspunkt mellom to tog
(som ikke holder kopnstant fart)
*/

DECLARE
   linje_nr_1 number:=2101;
   linje_nr_2 number:=2102;
   t1 number;
   t2 number;
   t number;
   x number;
BEGIN
  rdbms_krysspunkt(linje_nr_1,linje_nr_2,t,x);
  select tognr into t1 from Rute where linje_id=linje_nr_1;
  select tognr into t2 from Rute where linje_id=linje_nr_2;

if t=-1 and x=-1
  then
  dbms_output.put_line('Ingen kryss punkt mellom togene '||t1||' og '||t2);
  else
 x:=round(x,5);
 t:=round(t,5);
   dbms_output.put_line('Togene har krysspunkt:');
   dbms_output.put_line('X :'||x);
   dbms_output.put_line('T :'||t);
end if;

END;
/
```

16.2.2 ORDBMS – Objektrealsjonell databasehåndteringssytem

```
/*Versjon 2

Data modellen i ordbms
for å finne krysspunkt mellom to tog
(som ikke holder konstant fart)
*/

drop table geomtogene;
drop type geomtog;
drop type region;
drop type polygons;
drop type polygon2;
drop type lines;
drop type line2;
drop type points;
```

```
drop type point2;
drop type points_def;


CREATE type point2 as object(
      t number,
      x number);
/
CREATE type points as varray(10000) of point2;
/
CREATE type points_def as varray(10000) of number;
/
CREATE type line2 as object(
        type number,
      elem_def points_def,
        element  points,
        member function finnKrysspunktet(l line2) return point2);
/
CREATE type lines as varray(1048576) of line2;
/
CREATE type polygon2 as object(
        type number,
        ring number,
        elem_def points_def,
        element  points
        );
/

CREATE type polygons as varray(50000) of polygon2;
/
CREATE type region as object(
 multipolygon polygons,
 multilines   lines,
 multipoints  points
 );
/

CREATE  type Geomtog as object(
      tognr NUMBER,
      geometry line2,
      member function krysspunkt(t Geomtog) return point2);
/

create table geomtogene of Geomtog;

/



insert into geomtogene values(609,line2(1,points_def(1,1),
points(point2(0747,89.57),point2(1005,275.5),point2(1013,275.5),
point2(1033,302.1),point2(1039,302.1),point2(1107,335.8),point2(1114,335.8),
point2(1244,425.29),point2(1317,461.93),point2(1324,461.93),
point2(1334,471.25))));


insert into geomtogene values(62,line2(1,points_def(1,1),
points(point2(0758,471.25),point2(0948,335.8),point2(0951,335.8),
point2(1310,89.57))));
```

```
 select tognr,t.geometry
 from geomtogene t;


/*Versjon 2

Funksjoner for å finne krysspunkt i ordbms
mellom to tog (som ikke holder konstant fart)

*/

create or replace type body line2 as
member function finnkrysspunktet (l line2) return point2
is a1 number; b1 number; a2 number; b2 number; t0 number; x0 number;

type coord_tab is  table of number index by binary_integer ;

x1 coord_tab;
t1 coord_tab;
x2 coord_tab;
t2 coord_tab;
p0 point2;
antall_kryss number;
Begin
FOR counter IN 1..element.LAST
LOOP
   x1(counter):= ( element(counter).x );
   t1(counter):= ( element(counter).t );
END LOOP;

FOR counter IN 1..l.element.COUNT
LOOP
   x2(counter):= ( l.element(counter).x );
   t2(counter):= ( l.element(counter).t );
END LOOP;


FOR i IN x1.FIRST..x1.LAST-1
loop
 FOR j IN x2.FIRST..x2.LAST-1
  loop
     a1:=(x1(i)-x1(i+1))/(t1(i)-t1(i+1));
     b1:=((x1(i+1)*t1(i))-(x1(i)*t1(i+1)))/
         (t1(i)-t1(i+1));

     a2:=(x2(j)-x2(j+1))/(t2(j)-t2(j+1));
     b2:=((x2(j+1)*t2(j))-(x2(j)*t2(j+1)))/
         (t2(j)-t2(j+1));


if(a1-a2)>0
    then
      t0:=(b2-b1)/(a1-a2);
      x0:=(a1*t0)+b1;
end if;

if  t1(i)<=t0 and t0<=t1(i+1)and t2(j)<=t0 and t0<=t2(j+1)
 then
   if  x1(i)<=x0 and x0<=x1(i+1)and x2(j)>=x0 and x0>=x2(j+1)
```

```
  then

  p0:= NEW point2(t0,x0);
 antall_kryss:=antall_kryss+1;

  end if;
end if;


  end loop;
end loop;

 if antall_kryss=0
then
 p0:= NEW point2(-1,-1);
end if;

return p0;

end finnkrysspunktet;
end;
/


create or replace type body Geomtog as
member function krysspunkt (t geomtog)
return point2 is

point  point2;

Begin
point :=self.geometry.finnkrysspunktet(t.geometry);
return point;

end krysspunkt;
end;
/




create or replace function
finn_togene(tnr1 number,tnr2 number)
return point2 is

cursor tog(nr number) is
select value(t)
from geomtogene t where tognr=nr;

p point2;
tog1 geomtog;
tog2 geomtog;

begin
OPEN tog(tnr1);
   FETCH tog INTO tog1;
close tog;

OPEN tog(tnr2);
   FETCH tog INTO tog2;
close tog;
```

```
p:=tog1.krysspunkt(tog2);

return p;
end;
/



/*Versjon 2

Blokken som kaller prosedyren i ordbms
for å finne krysspunkt mellom to tog
(som ikke holder kopnstant fart)
*/

DECLARE
   p    point2;
   tog_nr_1 number:=609;
   tog_nr_2 number:=62;
   x number;
   t number;
BEGIN
   p:=finn_togene(tog_nr_1,tog_nr_2);

if  p.x=-1 and p.t= -1
 then
  dbms_output.put_line('Ingen krysspunkt');
else
   x:= round(p.x,5);
   t:= round(p.t,5);
   dbms_output.put_line('X-verdi'|| ' ' ||x);
   dbms_output.put_line('T-verdi'|| ' ' ||t);
end if;

END;
/
```

16.2.3 OO- Objekt orientert

16.2.4

16.3 Oracle Spatial kryss (Versjon 3)

16.3.1 Datamodell

```
/* Versjon 3

Data modellen som inneholder
to linjer for for begge tognummerene:
1_609: linjen som representerer lineære togfremføringen for tognr 609
1_62: linjen som representerer lineære togfremføringen for tognr 62
2_609: linjen som representerer ikke lineære togfremføringen for tognr 609,
       men den faktiske fremføringen av toget.
2_62: linjen som representerer ikke lineære togfremføringen for tognr 62,
       men den faktiske fremføringen av toget.
*/

drop table sdo_tog;
```

```
create table sdo_tog(
tognr varchar(10),
rekke number,
rute mdsys.sdo_geometry,
primary key(tognr,rekke)
);

/*endringer her*/


insert into sdo_tog values (
'1_609',
 1,
mdsys.sdo_geometry(
 2002,
 null,
 null,
 mdsys.sdo_elem_info_array(1,2,1),
mdsys.sdo_ordinate_array(0747,89.57,1334,471.25)));



insert into sdo_tog values (
'1_62',
 1,
mdsys.sdo_geometry(
 2002,
 null,
 null,
 mdsys.sdo_elem_info_array(1,2,1),
mdsys.sdo_ordinate_array(0758,471.25,1310,89.57)));





insert into sdo_tog values (
'2_609',
 1,
mdsys.sdo_geometry(
 2002,
 null,
 null,
 mdsys.sdo_elem_info_array(1,2,1),
mdsys.sdo_ordinate_array(0747,89.57,1005,275.5,1013,275.5,1033,302.1,1039,30
2.1,1107,335.8,1114,335.8,1244,425.29,1317,461.93,1324,461.93,1334,471.25)));


insert into sdo_tog values (
'2_62',
 1,
mdsys.sdo_geometry(
 2002,
 null,
 null,
 mdsys.sdo_elem_info_array(1,2,1),
mdsys.sdo_ordinate_array(0758,471.25,0948,335.8,0951,335.8,1310,89.57)));
```

```
delete from  user_sdo_geom_metadata
where upper(table_name)=upper('sdo_tog');



insert into user_sdo_geom_metadata
values('sdo_tog',
       'rute',
        mdsys.sdo_dim_array(
            mdsys.sdo_dim_element('t',0,2400,0.005),
            mdsys.sdo_dim_element('x',0,471.25,0.005)
            ),
        null
);


drop index tog_fix;


SDO_TUNE.ESTIMATE_TILING_LEVEL('SDO_TOG','RUTE',10000,'AVG_GID_EXTENT')
-----------------------------------------------------------------------
                                                                      0

create index tog_fix on sdo_tog(rute) indextype is mdsys.spatial_index
parameters ('sdo_level=0');


commit;
```

16.3.2 Enkel kryss

```
/* Versjon 3

Disse spørringene finner kryss
mellom to tog (som har konstant fart, altså lineær fremføring)
når data er lagret i Oracle Spatial
*/
select sdo_geom.sdo_intersection(t1.rute,t2.rute,0.005)
from sdo_tog t1, sdo_tog t2
where t1.tognr='2_609' and t1.rekke=1 and t2.tognr='2_62'and t2.rekke=1;


Resultat:
**********

SDO_GEOM.SDO_INTERSECTION(T1.RUTE,T2.RUTE,0.005)(SDO_GTYPE, SDO_SRID,
SDO_POINT(
----------------------------------------------------------------------------
----
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1021.8178, 287.227671))



SELECT SDO_GEOM.SDO_INTERSECTION(t1.rute, m.diminfo,t2.rute , m.diminfo)
```

```
  FROM sdo_tog t1, sdo_tog t2, user_sdo_geom_metadata m
  WHERE m.table_name = 'SDO_TOG' AND m.column_name = 'RUTE'
  AND t1.tognr='2_609' and t1.rekke=1 and t2.tognr='2_62'and t2.rekke=1;

Resultat:
**********

SDO_GEOM.SDO_INTERSECTION(T1.RUTE,M.DIMINFO,T2.RUTE,M.DIMINFO)(SDO_GTYPE,
SDO_SR
----------------------------------------------------------------------------
----
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1021.8178, 287.227671))

select sdo_tune.Estimate_Tiling_level('sdo_tog', 'rute', 10000,
'avg_gid_extent')
from dual;
```

16.3.3 Avansert kryss

```
/*Versjon 3

Disse spørringene finner kryss
mellom to tog (som ikke har en lineær fremføring)
*/
select sdo_geom.sdo_intersection(t1.rute,t2.rute,0.005)
from sdo_tog t1, sdo_tog t2
where t1.tognr='1_609' and t1.rekke=1 and t2.tognr='1_62' and t2.rekke=1;


Resultat:
**********

SDO_GEOM.SDO_INTERSECTION(T1.RUTE,T2.RUTE,0.005)(SDO_GTYPE, SDO_SRID,
SDO_POINT(
----------------------------------------------------------------------------
----
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1037.15013, 278.231844))


SELECT SDO_GEOM.SDO_INTERSECTION(t1.rute, m.diminfo,t2.rute , m.diminfo)
  FROM sdo_tog t1, sdo_tog t2, user_sdo_geom_metadata m
  WHERE m.table_name = 'SDO_TOG' AND m.column_name = 'RUTE'
  AND t1.tognr='1_609'and t1.rekke=1 and t2.tognr='1_62'and t2.rekke=1;



Resultat:
**********
SDO_GEOM.SDO_INTERSECTION(T1.RUTE,M.DIMINFO,T2.RUTE,M.DIMINFO)(SDO_GTYPE,
SDO_SR
----------------------------------------------------------------------------
----
SDO_GEOMETRY(2001, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1, 1),
SDO_ORDINATE_ARRAY(
1037.15013, 278.231844))
```

Referanse liste

[Ameskamp 97]Martin Ameskamp, Kiel University, 1997 : "Combining Field and object view: a three dimensional continues soil model" URL:http://esrnt1.tuwien.ac.at/ak_quant/eucol/c-3-1.htm

[Baumgarten 2001] Kurt Baumgarten 2001: "How to Use Java and JDBC To Access a Database" URL: http://cse-ferg41.unl.edu/BORE/BoreDocs/JDBC.html

[Bennett] Brandon Bennett: "Regions Representable in the Closure Algebra of Half-Planes"

[Bjørn 1997] Skjellaug Bjørn 1997: "Temporal Data: Time and Object Database." Research Report 245. Oslo, Norway.

[Brun] Adam Brun " DISKRETISERING AF MODELOMRÅDET I TID OG STED" URL: http://www.vandmodel.dk/staabi-kap7.pdf

[D. Shah] Nilesh D. Shah: "Database Systems Using Oracle: A Simplified Guide to SQL and PL/SQL"

[Egenhofer] Max J.Egenhofer: "A model for detailed binary topological relationships*" URL:http://www.spatial.maine.edu/~max/dimCountTopRel.pdf

[Etzelmuller, 1997] Bernd Etzelmüller: " Interpolasjonsmetoder" URL:http://www.geografi.uio.no/prosjekt/gisinterpol.html Universitet i Oslo 1997

[Erwig] Markus Erwig og Markus Schneider: " The Honeycomb Model of Spatio-Temporal Partitions. Hagen" Germany

[Goodchild unit003] Michael F. Goodchild. "Unit 055 – Rasters" URL:http://www.ncgia.ucsb.edu/giscc/units/u055/u055.html.University of California.

[Goodchild unit002] Michael F. Goodchild. "Unit 002 - What is geographic Information Science?" URL:http://www.ncgia.ucsb.edu/giscc/units/u002/u002.html.University of California.

[Guting] Ralf Hartmut Guting og Markus Schneider: "Realm-Based Spatial Data Types: The ROSE Algebra" FernUniversitåt Hagen, Germany.

[Guting, 1995] Ralf Hartmut Güting, Markus Schneider. 1995 VLDB Journal URL: http://citeseer.ist.psu.edu/guting95realmbased.html

[Joseph 2002] Joseph F. Jr. 2002 Massachusetts Institute of Technology: "Spatial Data Models" URL:http//:gis.mit.edu/classes/11.520/lectures/

[Jensen 2000] Christian S Jensen 2000: " Temporal Database Management" Aalborg University

[Kristoffersen] Marius Vartdal Kristoffersen: "Topologi" URL:http://www.geocities.com/CapeCanaveral/Hangar/3736/mobius.htm

[Kujipers] Bart Kujipers: "Spatial and Spatio-Temporal Data Models for GIS" URL:http://www.lore.ua.ac.be/Teaching/Sem2LIC/GISDatamodel.ppt Limburgs Universitair Centrum

[Laudal , 03.04.2000] Olav Arnfinn Laudal , 03.04.2000:"Rom og dimensjon – hva er det?" UiO Apollon

[Lake] Ron Lake: "Geography Mark-Up Language, Foundation for the web"

[Logley 2002] Mike F. Goodchild, Paul A Logley, David W. Rhind 2002: "Geogrphic Information systems and science"

[Midtbo] Terje Midtbo: " Handtering av tidsdimensjonen for stadfesta informasjon" URL:http://www.geomatikk.ntnu.no/tempgis/presentasjonar/TempGIS_01.ppt NTNU

[McClure 1997] Steve McClure 1997: "Object Database vs. Object-Relational Databases" URL:http://www.ca.com/products/jasmine/analyst/idc/14821E.htm

[OpenGis 1998] OpenGis 1998: " OpenGis Simple Features Specification for SQL" URL:http://www.cast.net.cn/gis/OpenGIS%20Simple%20Features%20Specification%20For%20SQL%20Revision%201.0%20PDF.pdf

[Oracle9i 2002] Oracle9i 2002 Application Developer's Guide - Object-Relational Features Release 2 (9.2) – "2 Basic components of oracle objects" URL:http://www.stanford.edu/dept/itss/docs/oracle/9i/appdev.920/a96594/adobjbas.htm

[Oracle]Oracle "Oracle9i JDBC Developer's Guide and Reference (Basic Features 3)" URL: http://www.informit.com/articles/article.asp?p=26251&seqNum=6

[Oracle Spatial]Oracle Spatial: 'User's Guide and reference' Release 9.2 URL: http://www.stanford.edu/dept/itss/docs/oracle/9i/appdev.920/a96630/toc.htm

[Oracle/PLSQL Topics 2004] Oracle/PLSQL Topics 2004: "Cursors" URL: http://www.techonthenet.com/oracle/cursors/

[OGC 2002]OGC 2002: " OpenGis Geography Markup language (GML) Implementation Specification, version 2.1.1" URL: http://www.opengeospatial.org/docs/02-009.pdf

[Parent] Christine Parent, Stefano Spaccapietra og Esteban Zimànyi. "Spatio-Temporal Conceptual Models: Data Structures + Space + Time". URL: http://yeroos.isys.ucl.ac.be/file.pdf/P-99-01.pdf Sveits og Belgia.

[PL/SQL User's Guide and references Release2(9.2] PL/SQL User's Guide and references Release2(9.2): "6 Interaction Between PL/SQL and Oracle" URL: http://www.stanford.edu/dept/itss/docs/oracle/9i/appdev.920/a96624/06_ora.htm#870

[Patridge, 1996] Cris Partidge. Business Object, re-engineering for re-use. Great Britain 1996

[Renolen, 1999] Renolen A 1999: "Concepts and Methos for Modelling Temporal and Spatiotemporal Information" Institutt for Kart og Oppmåling. NTNU

[Rigaux 2002] P. Rigaux, M. School og A. Voisard. 2002 :"Spatial Databases - with applications to GIS". Morgan Kaufmann Publishers. S Francisco, USA

[Ryu] Keun Ho Ryu og Yun Ae Ahn, 2001:"Application of Moving Objects and Spatiotemporal Reasoning".KOSEF (Korea Science and Engineering Foundation)

[Skagestein 2002] Gerhard Skagestein 2002: "Systemutvikling fra kjernen og ut" kapittel 12.

URL:http://www.tietowayla.fi/borland/cplus/revguide/001_rel.html)

[Skagestein, 12.mai 2004] G. Skagestein 12.mai 2004 - Institutt for informatikk.

URL:http://www.ifi.uio.no/~inf1050/foiler/oppsum.pdf

[Skagestein2] Gerhard Skagestein 2 : "Representations, identifiers and visualization data"

[Shekar] Shashi Shekar, Ranga Raju Vatsavi, Sanjay Chawla: "Spatial Pictogram Enhanced Conceptual Data Models and their translation to logical data models"

URL: http://www-users.cs.umn.edu/~vatsavai/papers/isd99-raju.pdf

[Silvia Nittel] Silvia Nittel, SIE 555: "Implementation Approaches to Spatial Database Systems"

[Shah 1996] Rawn Shah 1996: "Integrating Databases with Java via JDBC"

URL: http://www.javaworld.com/javaworld/jw-05-1996/jw-05-shah_p.html

[Snodgrass] Snodgrass R. T., Modern Database System, chapter 19 , 1995: "Temporal Object Oriented Database: A Critical Comparison" ACM Press. New York, USA

[Sunday] Dan Sunday: "Algorithm 4" URL:http://softsurfer.com/Archive/algorithm_0104/

[Tryfona] Nectaria Tryfona og Thansis Hadzilacos: "Logical Data Modelling of SpatioTemporal Applications: Definitions and a Model"

URL: www.cti.gr/RD3/DKE_old_9May2002/pubs/confis/ideas98.pdf Danmark og Helles

[Uio, terrengmodeller] University of Oslo:"Terrengmodeller"

URL:http://www.goegrafi.uio.no/prosjekt/

[Zlatanova 2001] Siyka Zlatanova 2001: "3D modelling for augmented reality"

URL:http://www.gdmc.nl/zlatanova/thesis/html/refer/ps/SZ-DMGIS.pdf

[Watson 2002] Paul Watson (january 2002): "Topology and ORDBMS Technology"

URL:http://www.laser-scan.com/pdf/topo_ordbms.pdf

[Wang] Xianoyu Wang, Xiaofang Zhou og Sanglu Lu. "Spatiotemporal Data Modelling and Management." URL: www.itee.uq.edu.au/~zxf/_papers/STDBSurvey.pdf Brisbane, Australia.

[36] Skogans sin doktoravhandling

[Øyehaug] Olve Øyehaug. "Projeksjoner"

URL: http://www.ii.uib.no/undervisning/kurs/v01/i291/i291/F03_Projeksjoner.pdf