

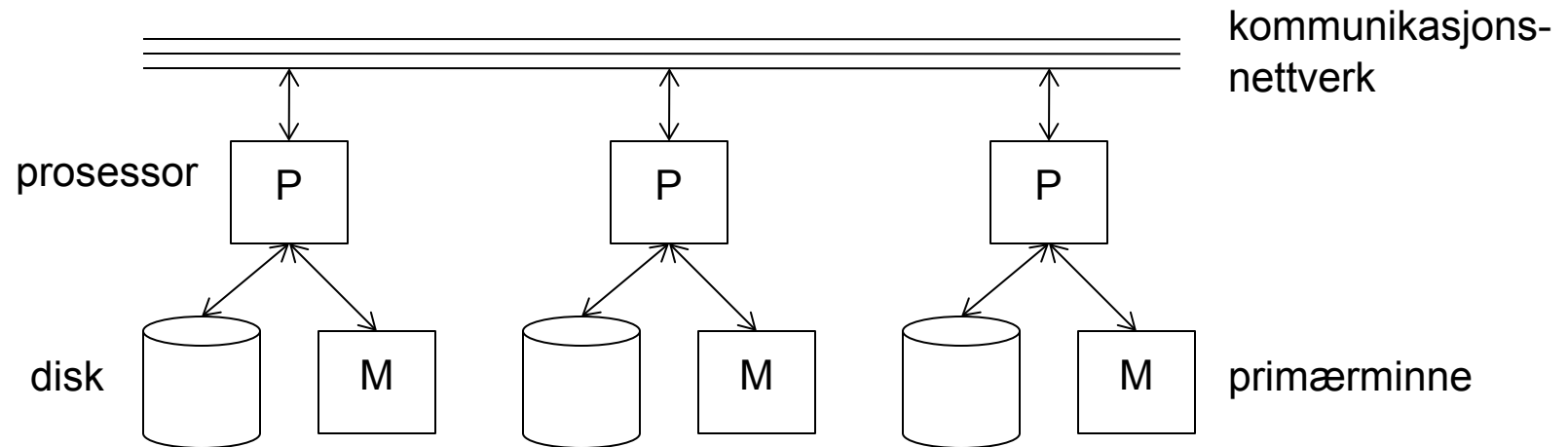


Parallele og distribuerte databaser Del I

Parallellberegninger

- **Database på én storskala parallellmaskin:**
Utnytter parallelliteten på dyre operasjoner, f.eks. join
- Modeller for parallellitet:
 - **Shared-memory-arkitektur**
 - Ett felles fysisk adresserom; hver prosessor har aksess til (deler av) primærminnet til de andre prosessorene
 - Hver prosessor har lokale disker
 - **Shared-disk-arkitektur**
 - Hver prosessor har lokalt primærminne
 - Hver prosessor har aksess til alle disker
 - **Shared-nothing-arkitektur**
 - Hver prosessor har lokalt primærminne og lokale disker
 - Er den arkitekturen som benyttes mest for databasesystemer

Shared-nothing-arkitekturen



- Hver prosessor har lokal cache, lokalt primærminne og lokale disk
- All kommunikasjon skjer ved meldingsutveksling over nettverket som forbinder prosessorene
 - Kostbart å sende data mellom prosessorer (betydelig overhead på hver enkelt melding)
 - Data bør om mulig bufres opp og sendes i større enheter

Parallele algoritmer for shared-nothing-arkitekturen

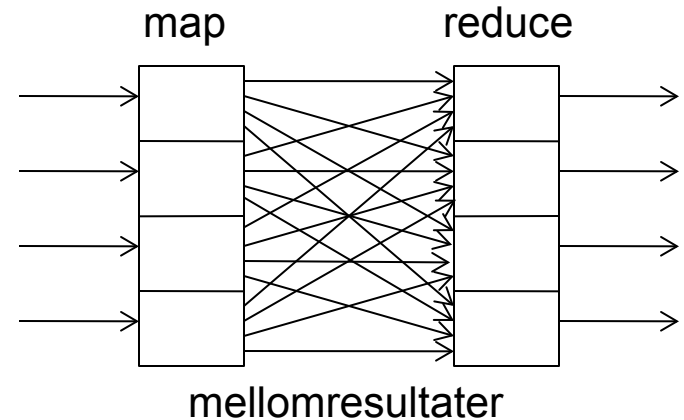
- Hovedprinsippet bak slike algoritmer er å fordele tuplene og arbeidsbelastningen mellom prosessorene. Tuplene fordeles slik at
 - kommunikasjonen minimeres
 - prosessorene kan utføre vanlige algoritmer lokalt på sine tupler
- Eksempler:
 - $\sigma_c(R)$: Fordel tuplene i R jevnt på alle diskene. Hver prosessor utfører σ_c lokalt på sine tupler
 - $R(X,Y) \bowtie S(Y,Z)$: Bruk en hashfunksjon h på Y til å fordele tuplene i R og S på p bønner (der p er antall prosessorer). Tuplene i bønne j sendes til prosessor j, som så kan utføre en lokal join på sine tupler

Map-reduce-rammeverket for parallellisering

- Rammeverk som tillater å enkelt beskrive/programmere databaseprosesser og utføre et høyt antall av dem i parallell
 - Brukeren skriver kode for to funksjoner, **map** og **reduce**.
- Arkitektur:
 - Massiv parallellmaskin eller maskincluster
 - Vanligvis shared-nothing-arkitektur
- Datalagring:
 - Filer, typisk svært store
 - Filene er oppdelt i chunks, hver chunk er replikert på tvers av disker for å overleve diskkræsje

Map

- **Map**-funksjonen tar som input par av formen (k,v) der k er en nøkkel og v en verdi.
- Output (**mellomresultater**) er en liste av par (h,w) der h er en nøkkel og w en verdi
- **Rammeverket** samler prinsipielt alle mellomresultatene for all input i par på formen $(h, [w_1, w_2, \dots, w_{n_h}])$, ett for hver verdi av h

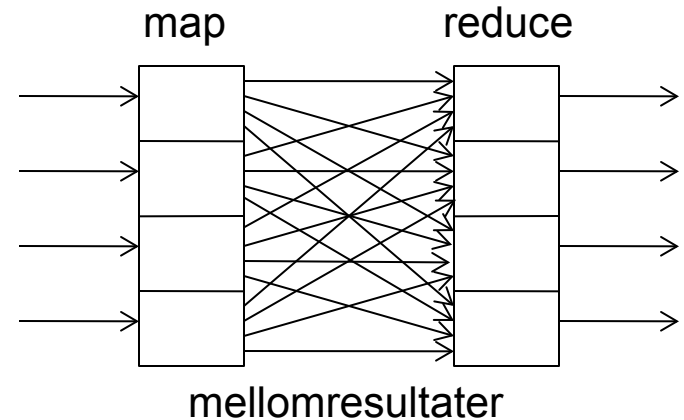


Reduce

- **Reduce**-funksjonen produserer for hvert par $(h, [w_1, w_2, \dots, w_{n_h}])$ ett par (h, x) der x er reduksjonen av $[w_1, w_2, \dots, w_{n_h}]$ (x er ofte av samme type som w_i -ene)

- **Mastercontrolleren**

- bestemmer hvor mange map- og reduce-prosesser som skal eksekveres, og på hvilke prosessorer
- fordeler chunks av inputdata blant map-prosessene
- bestemmer hvordan verdier i mellomresultatet skal pipelines mellom map- og reduce-prosessene



Eksempel:

Beregning av invertert indeks

- Map:
 - Input: Par på formen (i,d) hvor i er en dokumentID og d et dokument.
 - Virkemåte: Scanner d tegn for tegn; for hvert ord w produseres output (w,i) hvor w brukes som nøkkel
- Reduce:
 - Input: Par på formen $(w, [i_1, i_2, \dots, i_{n_w}])$
 - Virkemåte: Fjerner flerforekomster i $[i_1, i_2, \dots, i_{n_w}]$ og sorterer etter stigende i_k

Distribuerte databaser

- En database kalles **distribuert** hvis den er spredt over flere datamaskiner, kalt **noder** (sites), som er bundet sammen i et nettverk
- Hver node har sitt eget operativsystem og sitt eget DBMS
- Tre viktige formål med distribuerte databaser er:
 - større lagringskapasitet og raskere svartider
 - økt sikkerhet mot tap av data
 - økt tilgjengelighet av data for flere brukere
- Databasen kalles **distribusjonstransparent** hvis brukerne (applikasjonene) ikke merker noe til at databasen er distribuert (bortsett fra variasjon i svartidene)
- I dag er det en selvfølge at en distribuert database er distribusjonstransparent

Distribusjonsmodeller

- **Sharding:**
Hver node har ansvaret for et utsnitt (sin del) av dataelementene
- **Replikering:**
Hvert dataelement er lagret på flere noder
- De to modellene er uavhengige:
Et system kan bruke en eller begge teknikker

Distribusjon av relasjonelle data

- Et eksempel på distribuerte data kan vi finne i en butikkjede hvor alle salg registreres av kassaapparatene og lagres lokalt på en datamaskin (node) i hver enkelt butikk
- Logisk sett har butikkjedens relasjonsdatabase én relasjon som inneholder alle salgsdata fra alle butikkene
- Salgsdataene er **horisontalt fragmentert** med ett fragment på hver node
- Hvis de horisontale fragmentene er disjunkte, er fragmenteringen **total**
- En relasjon er **vertikalt fragmentert** hvis ulike attributter er lagret på ulike noder
- Vertikale fragmenter må inneholde primærnøkkelen
- En vertikal fragmentering er **total** hvis ingen andre attributter enn primærnøkkelen ligger på flere noder

Replikerte data

- I en konsistent tilstand er replikatene like (de er kopier av hverandre)
- Hvis alle data er replikert til alle noder, har vi en **fullreplikert** database (**speildatabase**)
- Internt bruker DBMS et replikeringsskjema som forteller hvilke data som ligger på hvilke noder
- Distribusjonstransparens medfører at applikasjonene ikke må ha kjennskap til replikeringsskjemaet og at de ikke har ansvar for å oppdatere replikatene
- Replikering er dyrt, men det gir økt lesehastighet og økt tilgjengelighet til data

Distribuerte transaksjoner og queries

- Når optimalisereren og planleggeren skal lage fysiske eksekveringsplaner, må de ta hensyn til hvilke noder de ulike dataene ligger på (denne informasjonen finnes i replikeringsskjemaet som er kopiert til alle noder)
- Det er to hovedstrategier å velge mellom:
 - Kopier (på billigste måte) de data som trengs, til samme node og utfør eksekveringen der
 - Splitt eksekveringen opp i subtransaksjoner på de aktuelle nodene og gjør mest mulig eksekvering der dataene er (viktig for projeksjon og spesielt seleksjon)
- En god optimaliserer kombinerer de to strategiene for å minimalisere datatransmisjonen mellom nodene

Distribuert commit

- En transaksjon i en distribuert database kan oppdatere data på flere noder (spesielt må dataelementer som er endret, oppdateres på alle noder med replikater)
- Den noden der en transaksjon T initieres, kalles **startnoden** (eller **utgangsnoden**) til T
- Planleggeren finner ut hvilke noder T trenger å aksessere og starter en subtransaksjon på hver av disse (inklusive startnoden) for å gjøre Ts jobb lokalt
- For å oppnå global atomisitet må enten alle subtransaksjonene gjøre commit, eller alle må abortere
- Følgelig kan ikke T gjøre commit før alle subtransaksjonene har gjort det

Tofasecommit

- **Tofasecommit (2PC)** er en protokoll for å sikre atomisitet av distribuerte transaksjoner
- 2PC bygger på følgende forutsetninger:
 - Det finnes ingen global logg
 - Hver node logger sine operasjoner (inklusive 2PC-meldinger som den har sendt til andre noder)
 - Hver node sikrer atomisitet for sine lokale transaksjoner
 - Ingen node går ned permanent
 - To vilkårlige noder kan kommunisere med hverandre
- 2PC forutsetter at en av nodene utpekes til **koordinator**. Vanligvis, men ikke alltid, er det startnoden som velges

2PC-protokollen – fase I

- Koordinatoren for en distribuert transaksjon T bestemmer seg for å gjøre commit
 - Koordinatoren skriver $\langle \text{Prepare } T \rangle$ i loggen på sin node
 - Koordinatoren sender meldingen ***prepare T*** til alle noder som har subtransaksjoner av T
- Hver mottaker fortsetter eksekveringen til den vet om dens subtransaksjon T_i kan gjøre commit
 - Skriv nok i loggen til at det ved behov kan gjøres gjenoppretting på T_i
 - Hvis kan committe,
 - skriv $\langle \text{Ready } T \rangle$ i loggen og skriv loggen til disk
 - send meldingen ***ready T*** til koordinatorenDeretter: Vent på fase II
 - Hvis må abortere,
 - skriv $\langle \text{Don't commit } T \rangle$ i loggen
 - send meldingen ***don't commit T*** til koordinatorenDeretter: Kan rulle transaksjonen tilbake mens den venter på fase II

2PC-protokollen – fase II

- Hvis koordinatoren har mottatt **ready T** fra alle nodene (subtransaksjonene), skriver koordinatoren <AgreeCommit T> i sin logg og sender **commit T** til alle andre involverte noder
 - En node som mottar **commit T**, gjør commit på sin subtransaksjon, skriver <Commit T> i loggen sin og sender **ack T** til koordinatoren
 - Når koordinator har mottatt **ack T** fra alle nodene, skriver koordinator <Commit T> i sin logg og foretar avsluttende commit på transaksjonen
- Hvis koordinatoren har mottatt **don't commit T** fra minst én node eller ikke alle har svart ved «timeout», skriver koordinatoren <Abort T> i sin logg og sender **abort T** til alle andre involverte noder.
 - En node som mottar **abort T**, skriver <Abort T> i loggen sin og ruller tilbake sin subtransaksjon hvis ikke dette alt er gjort

Feilhåndtering ved 2PC – I

- Hvis en ordinær node går ned, er det opplagt hva den skal gjøre med mindre dens siste loggpost er <Ready T>. I så fall må den spørre en annen node om den skal gjøre commit T eller abort T.
- Hvis koordinatoren går ned, velges en ny koordinator
- Den nye koordinatoren starter med å innhente status fra alle oppegående noder

Feilhåndtering ved 2PC – II

- Med ett unntak kan den nye koordinatoren fullføre 2PC
- Unntaket er hvis alle nodene har <Ready T> som siste loggpost. Da kan man ikke avgjøre om den opprinnelige koordinatoren har forberedt commit eller abort
 - Hvis den opprinnelige koordinatoren var den eneste noden som gikk ned, kan den nye koordinatoren anta at ingen commit har funnet sted og fortsette med abort T
 - Hvis det i tillegg til den opprinnelige koordinatoren var minst én ordinær node som gikk ned, kan den ordinære noden alt ha mottatt commit T fra den opprinnelige koordinatoren og rukket å committe. Det er to mulige fortsettelser:
 - Vente til koordinatoren kommer opp igjen
 - DBA griper inn og fatter en manuell avgjørelse

Låsing i distribuerte systemer

- Låsing av en replikat krever varsomhet:
 - Anta at T har leselås på en replikat A_1 av et dataelement A
 - Anta at U har skrivelås på en annen replikat A_2 av A
 - Da kan U oppdatere A_2 , men ikke A_1
 - Resultatet blir en inkonsistent database
- Med replikerte data må vi skille mellom to typer låsing:
 - låsing av et logisk dataelement A (global lås)
 - fysisk låsing av en av replikatene av A (lokal lås)
- Reglene for logiske lese- og skrivelåser er de samme som de som gjelder for vanlige låser i en ikke-distribuert database
- Logiske låser er fiktive – de må implementeres ved hjelp av de fysiske

Sentralisert låsing

- Den enkleste måten å implementere logiske låser på er å utnevne en av nodene til **låsesjef**
- Låsesjefen håndterer alle ønsker om logiske låser og bruker sin egen låstabell som logisk låstabell
- Det er to viktige svakheter ved et slikt sentralisert låsesystem:
 - låsesjefen blir fort en flaskehals ved stor trafikk
 - systemet er svært sårbart; hvis låsesjefen går ned, får ingen satt eller hevet noen lås
- Kostnaden er minst tre meldinger for hver lås som settes:
 - en melding til låsesjefen for å be om en lås
 - en svarmelding som innvilger låsen
 - en melding til låsesjefen for å frigi låsen

Primærkopilåsing

- Primærkopilåsing er en annen type sentralisert låssystem
- I stedet for en felles låsesjef velger vi for hvert logisk dataelement ut en av replikatene som **primærkopi**
- Den fysiske låsen på primærkopien brukes som logisk lås på dataelementet
- Metoden reduserer faren for flaskehalsen ved låsing
- Ved å velge replikater som ofte blir brukt, til primærkopier, reduseres antall meldinger ved håndtering av låser

Avledede logiske låser

- Metoden går ut på at en transaksjon får en logisk lås ved å låse et tilstrekkelig antall av replikatene
- Mer presist:
Anta at databasen har n replikater av et dataelement A
Velg to tall s og x slik at $2x > n$ og $s+x > n$
 - En transaksjon får logisk leselås på A ved å ta leselås på minst s replikater av A
 - En transaksjon får logisk skrivelås på A ved å ta skrivelås på minst x replikater av A

Det er maksimalt n låser til utdeling:

- At $2x > n$ medfører at to transaksjoner ikke begge kan ha logisk skrivelås på A
- At $s+x > n$ betyr at to transaksjoner ikke samtidig kan ha henholdsvis logisk leselås og logisk skrivelås på A

Leselås-én, skrivelås-alle

- Dette oppnår vi ved å velge $s = 1$ og $x = n$
- Logisk skrivelås krever i verste fall minst $3(n-1)$ meldinger og blir svært dyr
- Logisk leselås krever høyst 3 meldinger, og hvis det finnes en replikat på transaksjonens startnode, krever den ingen meldinger
- Metoden egner seg der skrivetransaksjoner er sjeldne
- Eksempel:
Elektronisk bibliotek der nodene har kopi av ofte leste dokumenter

Majoritetslåser

- Dette oppnår vi ved å velge $s = x = \lceil (n+1)/2 \rceil$
- Logisk skrivelås kan ikke bli billigere enn dette
- Men det at logisk leselås også krever omtrent $3n/2$ meldinger, virker svært dyrt
- I systemer som tilbyr kringkasting av meldinger, blir kostnaden lavere
- Fordelen er at metoden er robust mot nettverksfeil
- Eksempel:
Dersom en nettverksfeil deler databasen i to, kan den delen som inneholder flertallet av nodene, fortsette som om intet var hendt.
I minoritetsdelen kan ingen få så mye som en leselås

Distribuert vranglås

- Faren for vranglås i et distribuert låsesystem er stor
- Det finnes mange varianter av Vent-på-grafer som kan forhindre distribuert vranglås
- Erfaring sier at det enkleste og beste i de fleste tilfeller er å bruke «timeout»:
Transaksjoner som bruker for lang tid, rulles tilbake

Peer-to-peer-systemer

- Et P2P-nettverk er et nettverk av maskiner som
 - er **autonome**: noder kan bli med i eller forlate nettverket etter eget forgodtbefinnende
 - er **løst koblet**: kommunikasjon via et universalnett (f.eks. Internett)
 - har **lik funksjonalitet**: ingen leder- eller kontrollnoder
 - er villige til å **dele ressurser** med hverandre

Databaser på peer-to-peer-systemer

- En database i et P2P-nettverk må ivaretas kollektivt av nodene i nettverket. Krav til databasesystemet:
 - **Distribuert**: data er fordelt på nodene i nettverket
 - **Desentralisert**: alle nodene har samme administrative ansvar for databasen
 - **Feiltolerant**: systemet må være pålitelig selv om noder feiler og selv om den mengden av noder som utgjør nettverket, kontinuerlig endres
 - **Skalerbart**: systemet må være effektivt også for store nettverk

Distribuert hashing

- **Distributed hash tables** (DHT) er en klasse av teknikker for å realisere **oppslagstjenester i P2P-nettverk**
- Anta at databasen består av skjemaet $R(\underline{K}, V)$ der K er primærnøkkel og V tilhørende verdi. En DHT består av
 - en **nøkkelpartisjonering**, dvs. en hashfunksjon h
 - For hvert tuppel (k, v) i R brukes $h(k)$ til å beregne identiteten til den noden som har ansvaret for å lagre tuppelet (k, v)
 - et **overleggsnettverk (overlay network)**, dvs. en logisk struktur som knytter sammen nodene i nettverket
 - Hver node trenger bare å ha direkte kjennskap til en liten del av nettverket
 - Likevel kan noden som lagrer (k, v) , nås fra en hvilken som helst annen node med et lite antall meldingsutvekslinger

Eksempel på en DHT: Chord

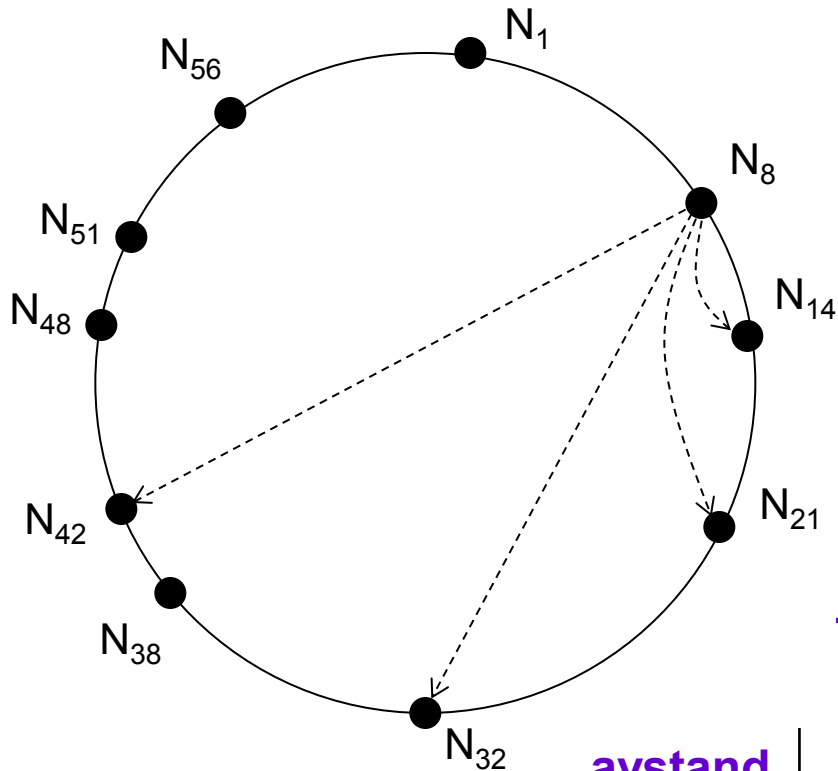
- Overleggsnettverket består av
 - en (logisk) **sirkel** som omfatter alle nodene i nettverket
 - strenger (**chords**) ”på tvers” i sirkelen
- Hashfunksjon:
 - $h(x) = x \bmod 2^m$ for et passende heltall m
 - maksimalt antall noder er 2^m

Overleggsnettverket i Chord

- En node med ID w plasseres i sirkelen i posisjon $h(w)$
 - Vi antar at alle noder har en ID og at disse er entydige
- Hver node har en **fingertabell** med peker til de nodene som har posisjon $j+1, j+2, j+4, \dots, j+2^{m-1}$ i sirkelen, der j er nodens egen posisjon i sirkelen
 - Hvis det ikke er noen node i posisjon $j+2^i$ for en i , inneholder fingertabellen den noden med urviseren som er nærmest denne posisjonen. (Posisjonene beregnes modulo 2^m .)
 - Fingertabellen har lengde m og tar derfor ikke stor plass selv for store nettverk (dvs. stor 2^m)
- I tillegg har hver node en peker til sin forgjenger

Eksempel: $m=6$

Maksimalt $2^6 = 64$ noder



De stiplede pilene viser nodene som er med i fingertabellen til N_8 . I tillegg vet N_8 hvilken node som kommer foran den i sirkelen.

Under er alle disse opplysningene samlet i én tabell.

Tabell for N_8 :

avstand	-1	1	2	4	8	16	32
node	N_1	N_{14}	N_{14}	N_{14}	N_{21}	N_{32}	N_{42}

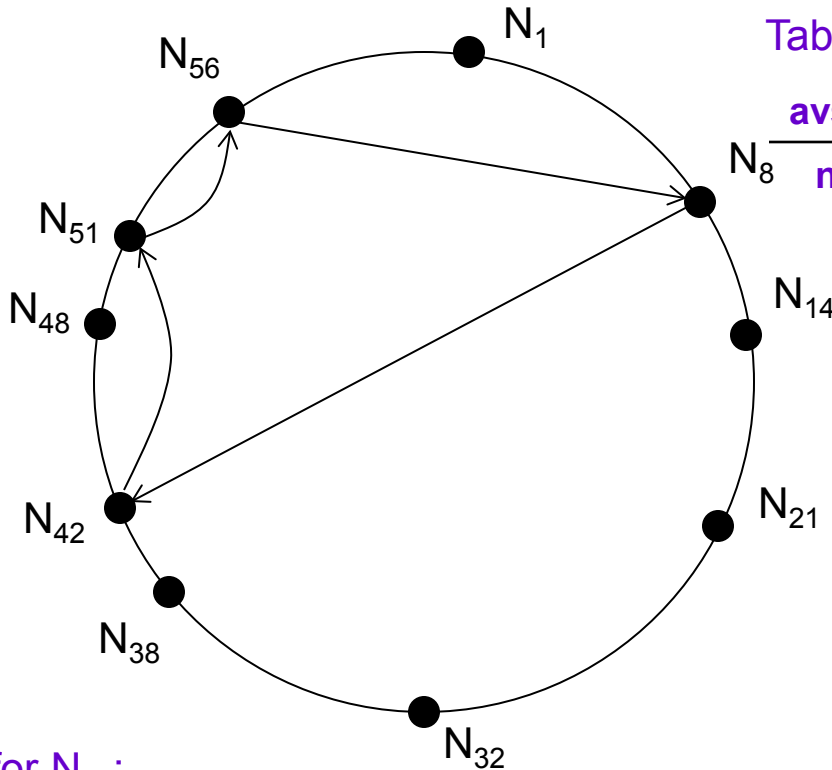
Datahåndtering i Chord

- Et tuppel (k,v) i tabellen $R(K,V)$ lagres på noden som har posisjon $h(k)$.
 - Hvis det ikke er noen node i posisjon $h(k)$, lagres tuppelet på den noden som er nærmest denne posisjonen (med urviseren).
- Hvordan finne et tuppel i nettverket: Anta at node N_i ønsker å finne v for en gitt k , men ikke har tuppelet lagret lokalt.
 1. Beregn $j = h(k)$. Sett $N_c = N_i$.
 2. N_c slår opp i tabellen sin og ser om den inneholder en node N_h der h er mindre eller lik j .
 - Hvis det ikke finnes noen slik h , er tuppelet (k,v) hos etterfølgernoden (N_e). N_c sender en melding til N_e og ber den om å returnere tuppelet direkte til N_i .
 - Hvis det finnes en slik h , så velg h størst mulig (men mindre eller lik j).
 - Hvis $h = j$, er tuppelet (k,v) hos N_h . N_c sender en melding til N_h og ber den om å returnere tuppelet direkte til N_i .
 - Hvis $h < j$, må tuppelet (k,v) befinne seg i en posisjon lenger ut i sirkelen enn h . N_c sender en melding til N_h og ber den om å overta rollen som N_c . Gjenta punkt 2.

Det trengs maksimalt $m+1$ meldinger: m forespørselsmeldinger + 1 melding med resultatet.

Eksempel:

N_8 ønsker å finne (k,v) der $h(k)=52$



Tabell for N_8 :

avstand	-1	1	2	4	8	16	32
node	N_1	N_{14}	N_{14}	N_{14}	N_{21}	N_{32}	N_{42}

Tupplet (k,v) ligger på den noden som ligger nærmest posisjon 52 (regnet med urviseren), dvs. node N_{56} . N_8 kan regne ut $h(k)=52$, men vet ikke at det ikke fins noen node N_{52} , og heller ikke at tupplet befinner seg i node N_{56} .

1. Nærmeste foran posisjon 52 i tabellen til N_8 er N_{42} . N_8 sender en forespørsel til N_{42} .
2. Nærmeste foran posisjon 52 i tabellen til N_{42} er N_{51} . N_{42} sender en forespørsel til N_{51} .
3. Siden etterfølgeren til N_{51} er N_{56} , må tupplet befinne seg der. N_{51} ber N_{56} sende tupplet til N_8 .

Totalt: 4 meldinger.

Tabell for N_{42} :

avstand	-1	1	2	4	8	16	32
node	N_{38}	N_{48}	N_{48}	N_{48}	N_{51}	N_1	N_{14}