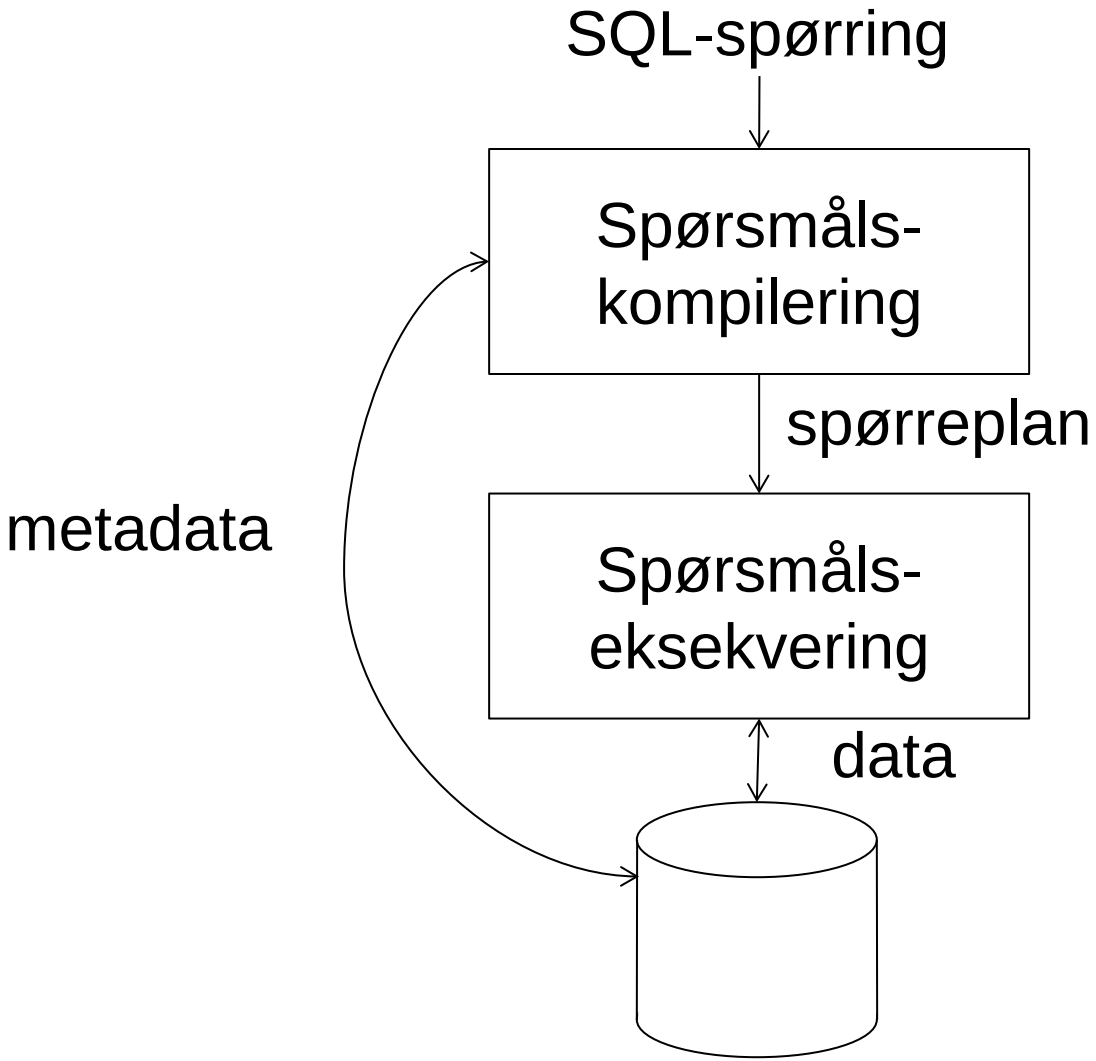
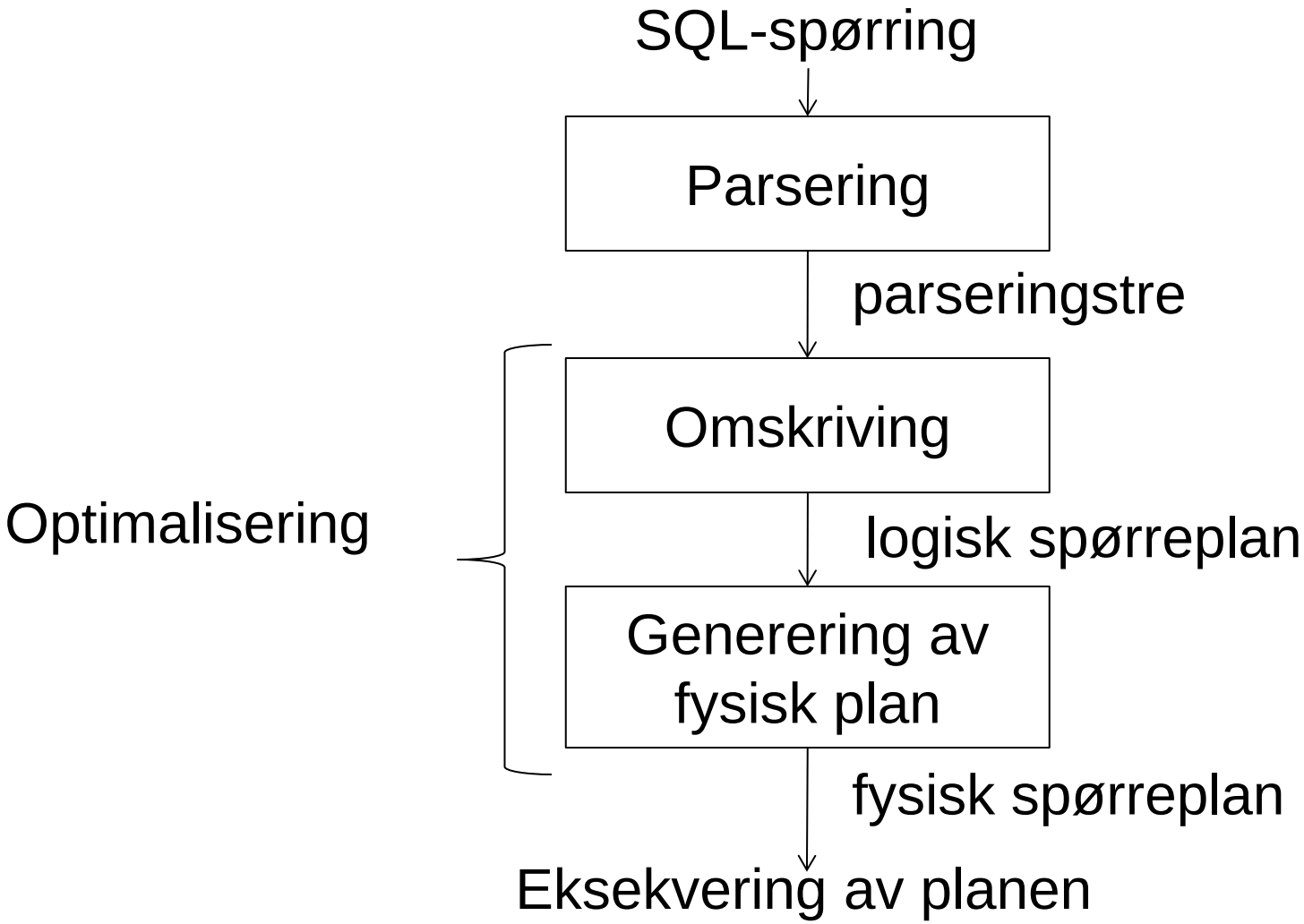
Effektiv eksekvering av spørsmål

- Spørsmålshåndtering
- Modell for kostnadsberegning
- Kostnad for basisoperasjoner
- Implementasjonsalgoritmer

Spørsmålshåndtering



Spørsmålskompilering



Fysiske operatorer

- En spørring består av et relasjonsalgebrauttrykk.
  - En fysisk spørreplan implementeres av et sett operatorer (algoritmer) tilsvarende operatorene fra relasjonsalgebraen.
  - I tillegg trenger vi basisoperatorer for å lese (skanne) en relasjon, sortere en relasjon osv.
- For å kunne velge en god plan, må vi kunne estimere kostnaden til hver operator.
  - Vi vil bruke **antall disk-IO** og antar generelt at
    - Operandene må initielt hentes fra disk.
    - Resultatet brukes direkte fra minnet.
    - Andre kostnader kan ignoreres.

Kostnadsparametre

- Hvilken mekanisme som har lavest kostnad, avhenger av flere faktorer, inkludert
  - Antall tilgjengelige blokker i minnet, M
  - Om det finnes indekser, og hva slags.
  - Layout på disken og spesielle diskegenskaper.
  - ...
- For en relasjon R trenger vi i tillegg å vite
  - Antall blokker som kreves for å lagre alle tuplene, B_R
  - Antall tupler i R, T_R
  - Antall distinkte verdier for et attributt a, $V_R(a)$
    Gjennomsnittlig antall tupler som er like på a, er da $T_R/V_R(a)$

Faktorer som øker kostnaden

- Bruk av indeks som ikke ligger i minnet.
- Ideelt sett skal tuplene til en relasjon ligge **tettpakket** (**clustered**) og derfor oppta B blokker. Men:
  - Et sett med tupler som passer inn i B blokker, kan likevel bruke B+1 blokker ved at de ikke starter på begynnelsen av den første blokken.
  - Blokkene kan inneholde ledig plass for innsetting av nye tupler.
  - Data kan være sortert og gruppert, med hver gruppe lagret i separate blokker.
  - Relasjonen kan være lagret sammenfiltret med andre.

spredt (scattered)

- **Vi vil generelt *ikke* inkludere disse faktorene i våre estimater.**

Kostnad for basisoperasjoner

- Kostnaden for å lese en diskblokk er 1 disk-IO
- Kostnaden for å skrive en diskblokk er 1 disk-IO
- Oppdatering koster 2 disk-IO

- Tre fundamentale operasjoner:
  - **Usortert innlesning** av en relasjon R:
    Kostnaden avhenger av lagringen
    - Tettpakket (clustered) relasjon – B_R disk-IO
    - Spredt (scattered) relasjon – worst-case T_R disk-IO
  - **Sortering**
  - **Hashpartisjonering**

- **Vi vil generelt anta *tettpakkede* relasjoner.**

Sortering

- Sort-scan leser en relasjon R og returnerer R i sortert rekkefølge mhp. en mengde attributter X.
  - Hvis R har en B-tre-indeks på X, kan denne brukes til å traversere tuplene i R i sortert rekkefølge
  - Hvis hele R får plass i minnet, bruk en effektiv sorteringsalgoritme – kostnad B_R disk-IO
  - Hvis R er for stor til å få plass i minnet, må vi bruke en sorteringsalgoritme som håndterer data i flere pass
    - Ofte brukt: two-phase multiway merge sort (TPMMS)

Two-Phase Multiway Merge Sort (TPMMS)

- Fase 1: Sorter deler av relasjonen (så mye som det er plass til i minnet) om gangen.
  - Fyll alle tilgjengelige minneblokker med blokker som inneholder relasjonen.
  - Sorter dette utsnittet i minnet.
  - Skriv det sorterte resultatet (en subliste) tilbake til disk.
  - Gjenta til alle blokkene er lest og alle postene er sortert i sublister.
  - Kostnad $2B_R$, dvs alle blokkene er både lest og skrevet.

Two-Phase Multiway Merge Sort (TPMMS)

- Fase 2: Flett (merge) alle de sorterte sublistene til én sortert liste ved gjentatt å gjøre følgende:
  - Les den første blokken fra hver subliste inn i minnet og sammenlign det første elementet i hver blokk.
  - Plasser det minste elementet i den endelige listen og fjern elementet fra sublisten.
  - Hent inn nye blokker fra hver subliste ved behov.
  - Kostnad B_R
- Total kostnad $3B_R$
- Forutsetning: Antall sublister må være mindre eller lik antall tilgjengelige minneblokker: $B_R/M \leq M$, dvs. $M \geq \sqrt{B_R}$

Hashpartisjonering

- Hashfunksjonen samler tupler som skal ses på i sammenheng.
- Med M tilgjengelige buffere:
  Bruk M-1 buffere for buckets, 1 for å lese diskblokker
- Algoritme:

```
FOR hver blokk b i relasjon R {
        les b inn i buffer M
        FOR hvert tuppel t i b {
                IF NOT plass i bucket h(t) {
                        kopier bucket h(t) til disk
                        initialiser ny blokk for bucket h(t) }
                kopier t til bucket h(t) }
        }
FOR hver ikketom bucket { skriv bucket til disk }
```

- Kostnad $2B_R$ – les alle data og skriv dem partisjonert tilbake. (NB: Dette inkluderer altså skriving tilbake!)

Eksekvering av spørsmål

- Tre hovedklasser med algoritmer:
  - sorteringsbaserte
  - hashbaserte
  - indeksbaserte
- I tillegg kan kostnad og kompleksitet deles inn i ulike nivåer
  - en-passalgoritmer – data passer i minnet, leses bare en gang fra disk.
  - to-passalgoritmer – data for stort for minnet, les data, prosesser, skriv tilbake, les på nytt
  - n-passalgoritmer – rekursive generaliseringer av to-pass-algoritmer for metoder som trenger flere pass over hele datasettet

Ulike grupper operatorer

- tuppel-om-gangen, unære operasjoner:
  - seleksjon (σ)
  - projeksjon (π)
- full-relasjon, unære operasjoner:
  - gruppering (γ)
  - duplikat-eliminasjon (δ)
- full-relasjon, binære operasjoner:
  - sett og bag union ($\cup$)
  - sett og bag snitt ($\cap$)
  - sett og bag differanse ($-$)
  - join ($\bowtie$)
  - produkt ($\times$)

Vi skal se på flere måter å implementere disse operatorene ved hjelp av ulike algoritmer og ulike antall pass.

Seleksjon og projeksjon: Tuppel-om-gangen-operatorer

- Både seleksjon (σ) og projeksjon (π) har opplagte algoritmer – uavhengig av om relasjonen passer i minnet eller ikke: Hent en og en blokk inn i minnet og prosesser hvert tuppel i blokken.
- Eneste krav til minnet er $M \geq 1$ for inputbufferet.
- Kostnaden avhenger av hvor R ligger:
  - i minnet – 0
  - på disk, typisk
    - B_R disk-IO hvis R ligger tettpakket
    - T_R disk-IO (worst-case) hvis R ligger spredt

Merk at vi bare foreleser utvalgte deler og oppsummeringer!
For flere algoritmer, se læreboken

Seleksjon: Worst-case

- **Worst-case**-eksempel: $T_R = 20.000$, $B_R = 1000$, $\sigma_{a = v}(R)$
  - ingen indeks
    - R tettpakket – hent alle blokkene → 1000 disk-IO
    - R spredt – alle tuplene er på forskjellige blokker → 20.000 disk-IO
  - indeks, R spredt – hent T_R / $V_R(a)$ blokker
    - $V_R(a) = 100$ → 20.000 / 100 = 200 disk-IO
    - $V_R(a) = 10$ → 20.000 / 10 = 2000 disk-IO
  - clusterindeks (R tettpakket) – hent B_R / $V_R(a)$ blokker
    - $V_R(a) = 100$ → 1000 / 100 = 10 disk-IO
    - $V_R(a) = 10$ → 1000 / 10 = 100 disk-IO
  - V(R, a) = 20.000, dvs. a er en kandidatnøkkel → 1 disk-IO

Duplikateliminasjon (δ): En-pass

- Duplikateliminasjon (δ) kan utføres ved å lese en blokk om gangen og for hvert tuppel
  - kopiere til output-buffer hvis det er første forekomst
  - ignorere hvis vi har sett et duplikat
- Krever en kopi av hvert tuppel i minnet for sammenligning.
- Total kostnad: B_R
- Minnekrav: 1 blokk til R, $B_{\delta(R)}$ blokker til å holde en kopi av hvert tuppel der $\delta(R)$ er antall forskjellige tupler i R, så $M \geq 1 + B_{\delta(R)}$

Duplikateliminasjon (δ): To-pass

- Sorteringsbasert algoritme:
  Tilsvarende Two-Phase Multiway Merge Sort (TPMMS)
  - Les M blokker inn i minnet av gangen.
  - Sortert disse M blokkene og skriv sublisten til disk.
  - I stedet for å flette sublistene, kopier første tuppel og eliminer duplikater i front av sublistene.
- Hashbasert algoritme:
  - Partisjoner relasjonen.
  - Duplikate tupler vil ha samme hashverdi.
  - Les hver bucket inn i minnet og utfør en-pass-algoritmen som fjerner duplikater.

Oppsummering: Kostnad og minnekrav for duplikateliminasjon $\delta(R)$

Algoritme	Minnekrav	Disk-IO
En-Pass	$M \geq 1 + B_{\delta(R)}$	B_R
To-Pass Sortering	$M \geq \sqrt{B_R}$	$3B_R$
To-Pass Hashing	$M \geq \sqrt{B_R}$	$3B_R$

Oppsummering: Kostnad og minnekrav for gruppering $\gamma(R)$

Algoritme	Minnekrav	Disk-IO
En-Pass	$M \geq 1 + B_{\gamma(R)}$	B_R
To-Pass Sortering	$M \geq \sqrt{B_R}$	$3B_R$
To-Pass Hashing	$M \geq \sqrt{B_R}$	$3B_R$

Binære full-relasjonsoperasjoner

- En binær operasjon tar to relasjoner som argumenter:
  - union: $R \cup S$
  - snitt: $R \cap S$
  - differanse: $R - S$
  - join $R \bowtie S$
  - produkt: $R \times S$

  Merk at det er forskjell på set- og bagversjonene av disse operatorene.
- Vi skal se på algoritmer for set- og bagunion samt naturlig join; de resterende operatorene kan implementeres på tilsvarende måte
- Alle operasjoner som krever sammenligning, må bruke en søkestruktur (f.eks. binærtrær eller hashing) som også krever ressurser. Vi tar imidlertid ikke med disse i våre estimater.

Union ($\cup$) : En-pass

- **Bag-**union kan beregnes med en veldig enkel en-pass algoritme:
  - Les og kopier hvert tuppel i R til outputbufferet.
  - Les og kopier hvert tuppel i S til outputbufferet.
- Total kostnad $B_R + B_S$ disk-IO
- Minnekrav 1 (les hver blokk direkte til outputbuffer)
- **Set**-union må fjerne duplikater
  - Les den *minste* relasjonen (anta S) inn i M-1 buffere og kopier hvert tuppel til output, men behold S i minnet.
  - Les blokkene i R inn i det siste bufferet, og sjekk for hvert tuppel om det eksisterer i S; hvis ikke, kopier til output
- Minnekrav er $B_S < M$

Union (∪) : To-pass, sortering

- **Set**-union to-pass sortering:
  - Gjør fase 1 av TPMMS for både R og S (lag sorterte sublister).
  - Bruk ett buffer for hver av sublistene til R og S.
  - Gjenta: Finn minste gjenværende tuppel i hver subliste
    - Output tuppelet.
    - Fjern duplikater fra fronten av alle listene
- Total kostnad er $3B_R + 3B_S$ disk-IO
- Minnekrav
  - M buffere gir rom for å lage sublister av lengde M blokker
  - $(B_R + B_S)/M \leq M$ gir $\sqrt{(B_R + B_S)} \leq M$
  - Hvis $B_R + B_S > M^2$ får vi mer enn M sublister, og algoritmen vil ikke fungere.

Union (∪) : To-pass, hashing

- **Set**-union to-pass hashalgoritme:
  - Partisjoner både R og S i M-1 buckets med samme hashfunksjon.
  - Gjør union på hvert bucketpar $R_i \cup S_i$ separat (en-pass setunion)

- Total kostnad: $3B_R + 3B_S$ disk-IO
  - 2 for partisjoneringen
  - 1 for union av bucketpar

- Minnekrav:
  - M buffere gir rom for å lage M-1 buckets for hver relasjon
  - For hvert bucketpar R_i og S_i: enten $B_{Ri} \leq M-1$ eller $B_{Si} \leq M-1$
  - Tilnærmet $\min(B_R, B_S) \leq M^2$ dvs $\sqrt{\min(B_R, B_S)} \leq M$
  - Hvis den minste av R_i og S_i ikke får plass i M-1 buffere for en i, vil algoritmen ikke fungere.

Oppsummering: Kostnad og minnekrav for R∪S

Hvis $B_S \leq B_R$:

BAG-versjon :

Algoritme	Minnekrav	Disk-IO
En-Pass	$M \geq 1$	$B_R + B_S$

SET-versjon :

Algoritme	Minnekrav	Disk-IO
En-Pass	$M \geq 1 + B_S$	$B_R + B_S$
To-Pass Sortering	$M \geq \sqrt{B_R + B_S}$	$3B_R + 3B_S$
To-Pass Hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

Oppsummering: Kostnad og minnekrav for R∩S

Hvis $B_S \leq B_R$:

Algoritme	Minnekrav[1]	Disk-IO
En-Pass	$M \geq 1 + B_S$	$B_R + B_S$
To-Pass Sortering	$M \geq \sqrt{B_R} + B_S$	$3B_R + 3B_S$
To-Pass Hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1]BAG-versjonen trenger i tillegg plass til tuppeltellere.

Oppsummering: Kostnad og minnekrav for R–S

Hvis $B_S \leq B_R$:

Algoritme	Minnekrav[1]	Disk-IO
En-Pass	$M \geq 1 + B_S$	$B_R + B_S$
To-Pass Sortering	$M \geq \sqrt{B_R} + B_S$	$3B_R + 3B_S$
To-Pass Hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1]BAG-versjonen trenger i tillegg plass til tuppeltellere.

Naturlig join (⋈) : en-pass

- Naturlig join (⋈) konkatenerer tupler fra R(X,Y) og S(Y,Z) som er slik at R.Y = S.Y
- En-pass-algoritme:
  - Les den *minste* relasjonen (anta S) inn i M-1 buffere.
  - Les relasjon R blokk for blokk. For hvert tuppel t
    - Join t med matchende tupler i S
    - Flytt resultatet til output

- Total kostnad $B_R + B_S$ disk-IO
- Minnekrav $B_S < M$

Naturlig join (⋈) : Nestet loop join – tuppel-basert algoritme

- Nestet loop join kan brukes for relasjoner av vilkårlig størrelse
- *Tuppel-basert* algoritme:

```
FOR hvert tuppel s i relasjon S
    FOR hvert tuppel r i R
        IF  r og s matcher, join til output
```

- Worst-case kostnad $T_R T_S$ disk-IO
- Minnekrav 2 (en R-blokk og en S-blokk)

Naturlig join (⋈) : Nestet loop join – blokk-basert algoritme

- Hold så mye som mulig av den minste relasjonen i M-1 blokker
- Algoritme: (anta S minst)

```
FOR hver partisjon p av S med størrelse M-1 {
        les p inn i minnet
        FOR hver blokk b for R {
                les b inn i minnet
                FOR hvert tuppel t i b {
                        finn tupler i p som matcher t
                        join hver av disse med t til output }}}
```

- Total kostnad $B_S + B_R B_S/(M-1)$ disk-IO
  (Les S en gang, les R en gang for hver S-partisjon)
- Minnekrav 2 (en R-blokk og en S-blokk; størrelsen på S-partisjonene tilpasses M, så i verste fall er hver S-partisjon én blokk)

Naturlig join (⋈) : To-pass sortering – enkel algoritme

- Sorter R og S hver for seg ved hjelp av TPMMS på joinattributtene.
- Join de sorterte R og S ved å
  - Hvis et buffer for R eller S er tomt, hente ny blokk fra disk
  - Finne tupler som har den minste v-verdien for join-attributtene
    (NB: Kan også finnes i påfølgende blokker; da må i såfall alle holdes i minnet!)
  - Hvis det finnes v-verditupler i både R og S, joine disse og skrive til output
  - Ellers, dropp alle v-verditupler.
- Total kostnad: $5B_R + 5B_S$ disk-IO
  - 4 for TPMMS
  - 1 for å joine de sorterte R og S
- Minnekrav:
  - TPMMS krever $B \leq M^2$ for hver av relasjonene, dvs. $B_R \leq M^2$ og $B_S \leq M^2$
  - Hvis det finnes en v-verdi der tilhørende tupler ikke får plass i M blokker, virker ikke algoritmen.

Naturlig join (⋈) : To-pass hashing

- Algoritme:
  - Partisjoner både R og S i M-1 buckets med samme hash-funksjon på joinattributtene Y. (Trenger 1 blokk til å holde fortløpende blokker av R og S.)
  - Gjør naturlig join på hvert bucketpar separat – $R_i \bowtie S_i$ – en-pass join.
- Total kostnad: $3B_R + 3B_S$ disk-IO
  - 2 for å partisjonere relasjonene
  - 1 for å gjøre join
- Minnekrav:
  - M buffere kan lage M-1 buckets for hver relasjon
  - For hvert par R_i og S_i må enten $B_{Ri} \leq M-1$ eller $B_{Si} \leq M-1$
  - Omtrent $\min(B_R, B_S) \leq M^2$ dvs $\sqrt{\min(B_R, B_S)} \leq M$
  - Hvis den minste av R_i og S_i ikke får plass i M-1 buffere, virker ikke algoritmen

Naturlig Join (⋈): Indeksbasert

- Algoritme $R(X,Y) \bowtie S(Y,Z)$:
  - Anta at det finnes en indeks for S på attributtene Y.
  - Les hver blokk for R. For hvert tuppel
    - finn tupler i S med samme joinattributter ved hjelp av indeksen
    - les de tilsvarende blokkene, gjør join på de aktuelle tuplene og output resultatet
- Total kostnad:
  - Hvis R er tettpakket, trenger vi B_R disk-IO, ellers opptil T_R for å lese alle R-tupler.
  - I tillegg må vi for *hvert tuppel i R* lese de tilsvarende S-tuplene:
    - Med clusterindeks på Y (så S er sortert på Y):
      $\lceil B_S / V_S(Y) \rceil$ S-tupler for hvert R-tuppel, totalt $T_R \lceil B_S / V_S(Y) \rceil$
    - Hvis S ikke er sortert på Y: $T_S / V_S(Y)$, totalt $T_R\, T_S / V_S(Y)$
  - Kostnaden domineres fullstendig av S, vi ignorerer derfor konstnaden ved å lese R
- Minnekrav 2 (en R-blokk og en S-blokk)

Oppsummering: Kostnad og minnekrav for $R \bowtie S$

Hvis $B_S \leq B_R$:

Algoritme	Minnekrav	Disk-IO
En-pass	$B_S \leq M - 1$	$B_R + B_S$
Tuppelbasert nestet loop	$2 \leq M$	worst case $T_R T_S$
Blokkbasert nestet loop	$2 \leq M$	$B_S + B_R B_S / (M-1)$
Enkel to-pass sortering	$\sqrt{B_R} \leq M$	$5 B_R + 5 B_S$
Sort-join	$\sqrt{B_R + B_S} \leq M$	$3 B_R + 3 B_S$
Hash-join	$\sqrt{B_S} \leq M$	$3 B_R + 3 B_S$
Hybrid hash-join	$\sqrt{B_S} \leq M$	$(3 - 2M/B_S)(B_R + B_S)$
Indeks-join, clusterindeks	$2 \leq M$	$T_R \lceil B_S / V_S(Y) \rceil$
Indeks-join, S ikke sortert på Y	$2 \leq M$	$T_R T_S / V_S(Y)$
Zig-zag indeks-join	$B(T_R/V_{R,a}) + B(T_S/V_{S,a}) \leq M$	$B_R + B_S$

Naturlig join: eksempel

- Eksempel:
  - T_R = 10.000, T_S = 5.000
  - $V_R(a)$ = 100, $V_S(a)$ = 10
  - Både R og S er tettpakkede
  - 4 KB blokker (ingen blokkhoder)
  - Både R og S poster er på 512 B (inkludert blokkhoder)
  - Har clusterindeks på attributt a for både R og S

⇨ B_S = 5.000 / 8 = 625
B_R = 10.000 / 8 = 1250

Naturlig join: eksempel (forts.)

T_R = 10.000, T_S = 5.000, B_R = 1250, B_S = 625, M = 101

Algoritme	Minimum minne	Disk-IO
En-pass	626	1875
Tuppelbasert nestet loop	2	781875
Blokkbasert nestet loop	2	9375
Enkel to-pass sortering	36	9375
Sort-join	44	5625
Hash-join	25	5625
Hybrid hash-join	25	5019
Indeks-join	2	626250 (S-indeks) 63125 (R-indeks)
Zig-zag indeks-join	76	1875

Valg av algoritme

- En-passalgoritmer er bra hvis ett av argumentene (relasjonene) får plass i minnet.
- To-passalgoritmer må brukes hvis vi har store relasjoner.
  - Hashbaserte algoritmer
    - Krever mindre minne sammenlignet med sorteringstilnærminger – er bare avhengig av den minste relasjonen – brukes ofte.
    - Antar omtrent lik bucketstørrelse (god hashfunksjon) – i virkeligheten vil det være noe variasjon, må anta mindre bucket-størrelser.
  - Sorteringsbaserte algoritmer
    - Gir sortert resultat, som kan utnyttes av påfølgende operatorer.
  - Indeksbaserte algoritmer
    - Utmerket for seleksjon
    - Utmerket for for join hvis begge argumentene har clusterindekser.

N–passalgoritmer

- For virkelig store relasjoner er ikke to pass tilstrekkelig.
- Eksempel: B_R = 1.000.000
  - TPMMS krever at $B_R < M^2$ → $M > 1000$
  - Hvis ikke 1000 blokker er tilgjengelige, virker ikke TPMMS.
  - ⇨ Trenger flere pass gjennom datasettet.
- Sorteringsbaserte algoritmer:
  - Hvis R får plass i minnet, sorter.
  - Hvis ikke, partisjoner R i M grupper og sorter hver R_i rekursivt.
  - Flett sublistene.
  - Total kostnad: $(2k– 1)B_R$, k er antall pass som er nødvendig
  - Vi trenger $\sqrt{B_R}$ buffere, dvs $B_R \leq M^k$
- Tilsvarende kan gjøres for hashbaserte algoritmer.