Exercise 9

1. **Review the following terms.**

External sort-merge – Eksternal sorterings fletning er en sorteringsmetode der anvendes når data er for store til at passe ind I hovedhukommelsen.
Den er baseret på merge sort og er opdelt I 2 faser.

- Fase 1 læser en side ad gangen fra disken, sortere den I hukommelsen og skriver den tilbage til disken som en midlertidig fil, kaldet run.
- Fase 2 består af flere gennemløb, hvor runs flettes indtil der kun er en sorteret run af længde N (Antal side).

Runs – Runs er de sorterede midlertidige filer, der producers I fase 1 af External Merge Sort.

- Givet et input på N sider, producerer fase 1 typisk N runs (I en 2-vejs sortering).
- I Multi-Way External Merge Sort er længden af hver run B sider, hvor B er antallet af tilgængelige buffer-sider I hukommelsen.

Join Strategies

De følgende termer beskriver forskellige strategier til at udføre join-operationer.

Simple Nested-loop join – Hvad er et Nested-loop join?

Forestil dig, at du har to tabeller (relationer):

- R (den ydre tabel)
- S (den indre tabel)

Et Nested-loop join betyder, at du for hvert række (tuple) i R, går igennem alle rækker i S for at finde dem, der passer sammen (f.eks. hvor et id er det samme).

Hvordan foregår det?

1. Tag første række i R.
2. Gå igennem alle rækker i S og tjek, om der er et match.
3. Gem de rækker, der passer sammen.
4. Gå videre til næste række i R og gentag.

Så du "nester" (lægger) én løkke (for R) inde i en anden løkke (for S) → deraf navnet Nested-loop join.

At læse data fra disken koster "I/O" (input/output).

Jo flere sider (blokke af data) der skal læses, jo dyrere er operationen.

Formlen:

$$\text{Omkostning} = |R| + (p_r \cdot |R|) \cdot |S|$$

Betyder:

- $|R|$: Vi skal læse hele R én gang.
- p_r: Antal rækker (tupler) på hver side i R.
- $(p_r \cdot |R|)$: Det samlede antal rækker i R.
- $(p_r \cdot |R|) \cdot |S|$: For **hver række i R**, læser vi hele S.

🔷 **Eksempel:**

Lad os sige:

- R har 100 sider
- S har 50 sider
- Der er 10 rækker per side i R → $p_r = 10$

Så:

$$\text{Omkostning} = 100 + (10 \times 100) \times 50 = 100 + 1000 \times 50 = 50.100 \text{ I/O'er}$$

Det er **meget dyrt**, fordi du tjekker hele S for hver række i R.

Page-Oriented Nested Loops Join - Denne metode forbedrer *Simple Nested-loop join* ved at reducere antallet af komplette læsninger af den indre tabel.

- ◦ I stedet for at iterere over tupler, itererer den over sider i den ydre relation og sider i den indre relation.
- ◦ Omkostningen (i I/O'er), hvis er den ydre relation, estimeres til

Omkostningsformel:

$$\text{Omkostning} = |R| + |R| \cdot |S|$$

Forklaring:

- $|R|$: Vi læser hele R én gang.
- For **hver side** i R ($|R|$), skal vi læse **hele S** ($|S|$).
- Derfor: $|R| + |R| \times |S|$.

Eksempel:

Lad os sige:

- R = 100 sider
- S = 50 sider

Så bliver:

$$\text{Omkostning} = 100 + 100 \times 50 = 5100 \text{ I/O'er}$$

Block nested loop join - Denne strategi udnytter yderligere buffere i hovedhukommelsen mere effektivt end *Page-Oriented Nested Loops Join*.

◦ Den indlæser en blok af tupler fra den ydre relation (svarende til sider, hvor er det samlede antal buffer-sider) og sammenligner denne blok med hele den indre relation

Omkostningsformel:

$$\text{Omkostning} = |R| + \left(\frac{|R|}{B-2}\right) \cdot |S|$$

Forklaring:

- $|R|$: Hele R skal stadig læses én gang.
- $\frac{|R|}{B-2}$: Antallet af blokke af R, du kan læse ad gangen.
- For hver blok skal du læse hele S → derfor $\left(\frac{|R|}{B-2}\right) \cdot |S|$.

Eksempel:

Antag:

- R = 100 sider
- S = 50 sider
- B = 12 buffer-sider

Så:

$$\text{Omkostning} = 100 + \left(\frac{100}{12-2}\right) \times 50 = 100 + (10) \times 50 = 600 \text{ I/O'er}$$

Sort-merge join – En Sort-Merge Join går ud på at:

- Sortere begge tabeller (R og S) efter det felt (attribut), de skal joines på.
- Gennemløbe (scanne) de to sorterede tabeller samtidig (som når man fletter to sorterede lister).
- Matche de rækker, der har samme join-værdi.

Omkostning (I/O'er)

Den totale pris består af:

$$\text{Omkostning} = \text{SortR} + \text{SortS} + (|R| + |S|)$$

Forklaring:

- **SortR** = omkostningen for at sortere R.
- **SortS** = omkostningen for at sortere S.
- **(|R| + |S|)** = selve join-trinnet (scanningen igennem begge relationer).

Eksempel:

Forestil dig:

- R = 100 sider
- S = 50 sider
- SortR = 200 I/O'er
- SortS = 120 I/O'er

Så:

$$\text{Omkostning} = 200 + 120 + (100 + 50) = 470 \text{ I/O'er}$$

Hash join - En Hash Join bruger en hash-funktion til at dele dataene op i mindre dele (kaldet *partitioner*).
Så i stedet for at sammenligne alle rækker mod alle, sammenlignes kun de rækker, der kan høre sammen — altså dem, som havner i samme partition.

Hvornår er Hash Join bedst?

Når joinet er på lighed (=) mellem attributter (fx R.kunde_id = S.kunde_id)
Når tabellerne ikke er sorterede
Når der er nok RAM til at bygge hash-tabellerne i hukommelsen

Omkostning (I/O'er)

En **2-pass hash join** har typisk:

$$3(|R| + |S|)$$

Eksempel:

Antag:

- R = 100 sider
- S = 50 sider

Så:

$$\text{Omkostning} = 3(100 + 50) = 450 \text{ I/O'er}$$

Det er ofte hurtigere end både **Sort-Merge Join** og **Nested-Loop Join**.

Equivalence rules - Ækvivalensregler (equivalence rules) bruges i logisk forespørgselsoptimering til at omskrive relationsalgebra-udtryk til andre udtryk, der giver samme resultat, men som kan udføres mere effektivt.

To forskellige algebraiske udtryk kan altså være ækvivalente, fordi de returnerer det samme resultat, selvom de udføres på forskellige måder. Formålet er at finde den mest effektive rækkefølge af operationer som selektion, projektion og join.

Statistics estimation - For at kunne beregne omkostningerne ved en forespørgsel indsamler databasen statistiske oplysninger om tabeller og kolonner.
De vigtigste nøgletal er:

- $\mathbf{n_r}$: antallet af tupler (rækker) i relationen r.
- **V(A, r)**: antallet af forskellige (distinkte) værdier for attributten A i relationen r.

Disse værdier bruges til at forudsige, hvor mange rækker der sandsynligvis vil matche en given søgebetingelse.

- **Size estimation -** Størrelsesestimering handler om at bruge statistikkerne til at forudsige størrelsen af resultatet af en operation, fx en selektion eller et join. Denne estimering hjælper query-optimereren med at vælge den mest effektive strategi.

Eksempler:

1. **Selektion med lighedsbetingelse**
   Hvis man vil finde alle rækker, hvor attributten A har en bestemt værdi (A = v):

   $$c = n_r / V(A, r)$$

   Det betyder, at vi antager, at værdierne for A er jævnt fordelt i tabellen.

   Eksempel:
   Hvis en tabel r har 10.000 rækker og 100 forskellige værdier i kolonnen A, forventes resultatet af σA=v(r) at være

   $$10.000/100 = 100 \text{ rækker.}$$

Histograms - *Histograms* er datastrukturer, der bruges til at gemme statistisk information om attributter (f.eks. fordelingen af værdier for en attribut som 'age')

2. Let relations *r1* and *r2* have the following properties: *r1* has 20,000 tuples, *r2* has 45,000
tuples, 25 tuples of *r1* fit on one page, and 30 tuples of *r2* fit on one page.

1) Estimate the I/O cost required using each of the following join strategies for *r1* ⋈ *r2*:

- Simple nested-loop join.
- Page-oriented nested-loop join.
- Sort-merge join. (Assume that one buffer page is needed to hold the evolving output page.)
- Hash join. (Assume that there is no need for recursive partitioning.)

Givet

- r1 har **20.000 tuples**
- r2 har **45.000 tuples**
- r1: **25 tuples pr. side**
- r2: **30 tuples pr. side**

Først: beregn antal sider (det er altid trin 1)

r1:
20.000 / 25 = **800 sider**

r2:
45.000 / 30 = **1.500 sider**

1) Simple Nested-Loop Join

Idé:
For hver tuple i r1 læses **hele r2**.

Formel:
I/O = (tuples i r1) × (sider i r2) + sider i r1

Beregning:
20.000 × 1.500 + 800
= **30.000.800 I/O**

Konklusion:
Meget langsom – læser r2 igen og igen.

2) Page-Oriented Nested-Loop Join

Idé:
For hver **side** i r1 læses **alle sider** i r2.

Formel:
I/O = (sider i r1 × sider i r2) + sider i r1

Beregning:
800 × 1.500 + 800
= **1.200.800 I/O**

Konklusion:
Meget bedre end simple nested-loop, men stadig dyr.

---

3) Sort-Merge Join

Idé:
Begge relationer sorteres først, derefter flettes de.

Antagelse: **1 buffer-side til output**

Sorteringsomkostning (2-pass sort):

- r1: 3 × 800 = **2.400**
- r2: 3 × 1.500 = **4.500**

Merge-fasen:

- læs r1 + r2 én gang:
  800 + 1.500 = **2.300**

Total I/O:
2.400 + 4.500 + 2.300 = **9.200 I/O**

Konklusion:
Meget effektiv sammenlignet med nested-loop.

---

4) Hash Join

Idé:
Del begge relationer op i buckets med hash, og join kun matchende buckets.

Antagelse: **ingen rekursiv partitionering**

Formel:
I/O = 3 × (sider i r1 + sider i r2)

Beregning:
3 × (800 + 1.500)

= 3 × 2.300
= **6.900 I/O**

Konklusion:
Hurtigste metode her.

Sammenfatning (meget vigtig)

- Simple nested-loop: **30.000.800**
- Page nested-loop: **1.200.800**
- Sort-merge join: **9.200**
- Hash join: **6.900** ← bedst

2) Say we have M = 100 + 2 memory buffers, estimate the I/O cost required using the following join strategy for *r1* ⋈ *r2*:

- e. Block nested-loop join.

Givet

- r1 = **800 sider**
- r2 = **1.500 sider**
- Hukommelse: **M = 100 + 2 = 102 buffers**

Ved **block nested-loop join**:

- **M − 2 buffers** bruges til den ydre relation
  → **100 sider ad gangen**

Trin 1: Vælg ydre relation

Man vælger altid den **mindste relation** som ydre relation.

→ **r1 (800 sider)** er ydre
→ **r2 (1.500 sider)** er indre

Trin 2: Hvor mange blokke af r1?

- 100 sider kan være i hukommelsen ad gangen
- r1 har 800 sider

Antal blokke:
800 / 100 = **8 blokke**

Trin 3: Beregn I/O-omkostning

Formel:
I/O = sider(r1) + (antal blokke × sider(r2))

Beregning:

- Læs r1 én gang: **800**
- For hver blok læses hele r2:
  8 × 1.500 = **12.000**

Total:
800 + 12.000 = **12.800 I/O**

Endeligt svar

Block nested-loop join I/O = 12.800

Kort forklaring (note-version)

- Brug den mindste relation som ydre
- Læs 100 sider ad gangen
- For hver blok læses hele den anden relation
- Læg det hele sammen

3. Given a relation $r(A, B, C)$ with $n_r = 10000$ and $V(A, r) = 500$.
($V(A, r)$ means the number of distinct values that appear in the relation r for attribute A.)

   a. Estimate the size of the selection operation $\sigma_{A=10}(r)$.

   b. Assume the range of values for an attribute C is [7, 59] and the values are uniformly distributed. Estimate the size of the selection operation $\sigma_{c<10}(r)$.

Givet

- Relation: r(A, B, C)
- Antal tupler:
  nr = 10.000
- Antal forskellige værdier for A:
  V(A, r) = 500

---

a) σ A = 10 (r)

Hvad betyder spørgsmålet?

Vi vil finde **hvor mange tupler** der forventes at opfylde betingelsen
A = 10.

Antagelse

Værdierne i A er **jævnt fordelt** (uniform distribution).

Det betyder:

- De 10.000 tupler er fordelt ligeligt på 500 forskellige A-værdier
- Hver A-værdi optræder lige ofte

Beregning

Antal tupler per A-værdi:
10.000 / 500 = 20

Svar

Størrelsen af σ A = 10 (r) er **ca. 20 tupler**

---

b) σ C < 10 (r)

Hvad betyder spørgsmålet?

Vi vil finde hvor mange tupler der opfylder:
$C < 10$

Givet om C

- C's værdier ligger i intervallet:
  [7, 59]
- Værdierne er **jævnt fordelt**

Trin 1: Hvor stort er intervallet?

$59 - 7 + 1 = 53$ mulige værdier

Trin 2: Hvilke værdier opfylder C < 10?

Det er værdierne:
7, 8, 9
→ 3 værdier

Trin 3: Andel af tupler

3 / 53 af alle tupler opfylder betingelsen

Trin 4: Beregn antal tupler

$(3 / 53) \times 10.000 \approx 566$

Svar

Størrelsen af σ C < 10 (r) er **ca. 566 tupler**

Kort opsummering (eksamens-venlig)

- **Lig med (=)** → divider med antal forskellige værdier
- **Ulighed (<, >)** → find andelen af intervallet
- Gange andelen med nr