Aufgabe 2

Join-Methoden
Differential Snapshots
Join Operator

• **JOIN**: Most important relational operator
  - Potentially very expensive
  - Required in all practical queries and applications
  - Often appears in groups of joins
  - Many variations with different characteristics, suited for different situations

• Example: Relations R (A, B) and S (B, C)
  
  ```
  SELECT * FROM R, S WHERE R.B = S.B
  ```

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>C4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C5</td>
</tr>
</tbody>
</table>

R \(\bowtie\) S
Nested-loop Join

- **Super-naïve**
  
  ```
  FOR EACH r IN R DO
    FOR EACH s IN S DO
      IF ( r.B=s.B) THEN OUTPUT (r \bowtie s)
  ```

- **Slight improvement**

  ```
  FOR EACH block x IN R DO
    FOR EACH block y IN S DO
      FOR EACH r in x DO
        FOR EACH s in y DO
          IF ( r.B=s.B) THEN OUTPUT (r \bowtie s)
  ```

- **Cost estimations**
  
  - \(b(R), b(S)\) number of blocks in \(R\) and in \(S\)
  - Each block of outer relation is read once
  - Inner relation is read once for each block of outer relation
  - Inner two loops are free (only main memory operations)
  - Altogether: \(b(R)+b(R)*b(S)\) IO
Example

- Assume $b(R)=10.000$, $b(S)=2.000$
- $R$ as outer relation
  - $IO = 10.000 + 10.000 \times 2.000 = 20.010.000$
- $S$ as outer relation
  - $IO = 2.000 + 2.000 \times 10.000 = 20.002.000$
- Use smaller relation as outer relation
  - For large relation, choice doesn’t really matter

- Can’t we do better??
• There is no “m” in the formula
  - m: Size of main memory in blocks
• This should make you suspicious
• We are not using our available main memory
Blocked nested-loop join

- **Rule of thumb:** *Use all memory you can get*
  - Use all memory the buffer manager allocates to your process

- **Blocked-nested-loop**
  
  ```
  FOR i=1 TO b(R)/(m-1) DO
    READ NEXT m-1 blocks of R into M
    FOR EACH block y IN S DO
      FOR EACH r in R-chunk DO
        FOR EACH s in y do
          IF (r.B=s.B) THEN OUTPUT (r ⋈ s)
  ```

- **Cost estimation**
  - Outer relation is read once
  - Inner relation is read once for every chunk of R
  - There are \( \sim b(R)/m \) chunks
  - IO = b(R) + b(R)*b(S)/m
  - Further advantage: Outer relation is read in chunks – sequential IO
Example

- Example
  - Assume \( b(R) = 10.000, b(S) = 2.000, m = 500 \)
  - \( R \) as outer relation
    - \( IO = 10.000 + 10.000 \times 2.000 / 500 = 50.000 \)
  - \( S \) as outer relation
    - \( IO = 2.000 + 2.000 \times 10.000 / 500 = 42.000 \)
  - Compare to one-block NL: 20.002.000 IO

- Use smaller relation as outer relation
  - Again, difference irrelevant as tables get larger

- But sizes of relations do matter
  - If one relation fits into memory \((b < m)\)
  - Total cost: \( b(R) + b(S) \)
  - One pass blocked-nested-loop

- We can do a little better with blocked-nested loop??
Zig-Zag Join

- When finishing a chunk of outer relation, hold last block of inner relation in memory
- Load next chunk of outer relation and compare with the still available last block of inner relation
- For each chunk, we need to read one block less
- Thus: Saves $b(R)/m$ IO
  - If $R$ is outer relation
Sort-Merge Join

- How does it work?
  - Sort both relations on join attribute(s)
  - Merge both sorted relations
- Caution if duplicates exist
  - The result size still is $|R|*|S|$ in worst case
  - If there are $r/s$ tuples with value $x$ in the join attribute in $R / S$, we need to output $r*s$ tuples for $x$
  - So what is the worst case??
- Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>C5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>C4</td>
</tr>
</tbody>
</table>
Example Continued

\[
\begin{array}{c|c|c}
\text{R} & A & B \\
\hline
A1 & 0 \\
A2 & 1 \\
A4 & 1 \\
A3 & 2 \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{S} & B & C \\
\hline
1 & C1 \\
1 & C3 \\
2 & C2 \\
3 & C4 \\
\end{array}
\]

Partial join \{<A1,0>,<A2,1>,<A4,1>\} \bowtie S

\[
\begin{array}{c|c|c}
\text{R} & A & B & C \\
\hline
A2 & 1 & C1 \\
A2 & 1 & C3 \\
A2 & 1 & C5 \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{S} & B & C \\
\hline
2 & C2 \\
3 & C4 \\
\end{array}
\]

new
Merge Phase

\[
\begin{align*}
r &:= \text{first (R)}; \quad s := \text{first (S)}; \\
\text{WHILE NOT EOR (R) and NOT EOR (S) DO} \\
&\quad \text{IF } r[B] < s[B] \text{ THEN } r := \text{next (R)} \\
&\quad \text{ELSEIF } r[B] > s[B] \text{ THEN } s := \text{next (S)} \\
&\quad \text{ELSE} \quad /* r[B] = s[B] */ \\
&\qquad b := r[B]; \quad B := \emptyset; \\
&\qquad \text{WHILE NOT EOR(S) and } s[B] = b \text{ DO} \\
&\qquad \quad B := B \cup \{s\}; \\
&\qquad \quad s = \text{next (S)}; \\
&\qquad \text{END DO;} \\
&\quad \text{WHILE NOT EOR(R) and } r[B] = b \text{ DO} \\
&\qquad \quad \text{FOR EACH } e \text{ in } B \text{ DO} \\
&\qquad \quad \quad \text{OUTPUT } (r,e); \\
&\qquad \quad \quad r := \text{next (R)}; \\
&\qquad \text{END DO;} \\
&\text{END DO;} \\
\end{align*}
\]
Cost estimation

- Sorting R costs \(2 \cdot b(R) \cdot \text{ceil}(\log_m(b(R)))\)
- Sorting S costs \(2 \cdot b(S) \cdot \text{ceil}(\log_m(b(S)))\)
- Merge phase reads each relation once
- Total IO
  - \(b(R) + b(S) + 2 \cdot b(R) \cdot \text{ceil}(\log_m(b(R))) + 2 \cdot b(S) \cdot \text{ceil}(\log_m(b(S)))\)
- Improvement
  - While sorting, do not perform last read/write phase
  - Open all sorted runs in parallel for merging
  - Saves \(2 \cdot b(R) + 2 \cdot b(S)\) IO
- Sometimes, we are able to save the sorting phase
  - If sort is performed somewhere down in the tree
  - Needs to be a sort on the same attribute(s)
- Merge-sort join: No inner/outer relation
Merge-Join and Main Memory

- We have no „m“ in the formula of the merge phase
  - Implicitly, it is in the number of runs required
- More memory doesn’t decrease number of blocks to read, but can be used for sequential reads
  - Always fill memory with \( m/2 \) blocks from R and \( m/2 \) blocks from S
  - Use asynchronous IO
    1. Schedule request for \( m/4 \) blocks from R and \( m/4 \) blocks from S
    2. Wait until loaded
    3. Schedule request for next \( m/4 \) blocks from R and next \( m/4 \) blocks from S
    4. Do not wait – perform merge on first 2 chunks of \( m/4 \) blocks
    5. Wait until previous request finished
       1. We used this waiting time very well
    6. Jump to 3, using \( m/4 \) chunks of M in turn
Hash Join

• As always, we may save sorting if good hash function available

• Assume a very good hash function
  – Distributes hash values almost uniformly over hash table
  – If we have good histograms (later), a simple interval-based hash function will usually work

• How can we apply hashing to joins??
Hash Join Idea

- Use join attributes as hash keys in both R and S
- Choose hash function for hash table of size m
  - Each bucket has size b(R)/m, b(S)/m
- Hash phase
  - Scan R, compute hash table, writing full blocks to disk immediately
  - Scan S, compute hash table, writing full blocks to disk immediately
  - Notice: Probably better to use some n<b(R)/m to allow for sequential writes
- Merge phase
  - Iteratively, load same bucket of R and of S in memory
  - Compute join
Hash Join Cost

- Hash phase costs $2b(R) + 2b(S)$
- Merge phase costs $b(R) + b(S)$
- Total: $3(b(R) + b(S))$

- Under what assumption??
Hybrid Hash Join

- Assume that \( \min(b(R), b(S)) < m^2/2 \)
- Notice: During merge phase, we used only \((b(R) + b(S))/m\) memory blocks (size of two buckets), although there might be much more
- Improvement
  - Chose smaller relation (assume S)
  - Chose a \textbf{number k of buckets} (with \( k < m \))
    - Again, assuming perfect hash functions, each bucket has size \( b(S)/k \)
  - When hashing S, \textit{keep first x buckets completely in memory}, but only one block for each of the \((k-x)\) other buckets
    - These x buckets are \textit{never written to disk}
  - When hashing R
    - If hash value maps into buckets 1..x, \textbf{perform join immediately}
    - Otherwise, map to the k-x other buckets and write to disk
  - After first round, we have performed the join on x buckets and have k-x buckets of both relations on disk
  - Perform “normal” merge phase on k-x buckets
Hybrid Hash Join - Cost

- Total saving (compared to normal hash join)
  - We save 2 IO for every block in either relation that is never written to disk
  - We keep x buckets in memory, having ~ b(S)/k and ~b(R)/k blocks
  - Together, we save 2*x*(b(S)+b(R))/k IO operations
- How should we choose k and x?
- Optimal solution
  - x=1 and k as small as possible
  - Build buckets as large as possible, such that still one entire bucket and one block for all other buckets fits into memory
  - Thus, we use as much memory as possible for savings
  - Optimum reached at ~k=b(S)/m
    - Actually, k must be smaller, so that M can accommodate 1 block for each other bucket
- Together, we save 2*(b(S)+b(R))*m/b(S)
- Total cost: (3-2m/b(S))*(b(S)+b(R))
  - Compared to 3*(b(R)+b(S)) for normal hash join
Comparing Hash Join and Sort-Merge Join

- If enough memory provided, both require approximately the same number of IO
  - $3 \times (b(R) + b(S))$
  - Hybrid-hash join improves slightly
- SM generates sorted results – sort phase of other joins in query plan can be dropped
  - Advantage propagates up the tree
- HJ does not need to perform $O(n \times \log(n))$ sorting in main memory
- HJ requires that only one relation is “small enough”, SM needs two small relations
  - Because both are sorted independently
- HJ depends on roughly equally sized buckets
  - Otherwise, performance might degrade due to unexpected paging
  - To prevent, estimate $k$ more conservative and do not fill $m$ completely
  - Some memory remains unused
- Both can be tuned to generate mostly sequential IO
Comparing Join Methods

Nested-Loops-Join

Merge-Join

Hash-Join
HashJoin: Schritt 1

Value1
Value2
Value3
Value4
Value5
Value6
Value7
Value8

Bucket 1
Bucket 2
Bucket 3
Bucket 4
Bucket 5
Bucket 6

Bucket 1'
Bucket 2'
Bucket 3'
Bucket 4'
Bucket 5'
Bucket 6'

Value1
Value4
Value5
Value6
Value9
Value3
Value7

HashJoin: Schritt 2
Zu beachten

• Die erste Hashfunktion sollte die Tupel gleichmäßig auf die Buckets verteilen
• Es sollten beim ersten Schritt nicht zu viele Buckets entstehen
  – Benötigen jeweils ein Filehandle
  – Datei öffnen & schließen etc ist langsam
• Bei zu wenigen Buckets enthält aber jedes zu viele Daten für den späteren Vergleich im Hauptspeicher
• Also: Maximale Größe der Buckets irgendwie abschätzen
• Bezüglich der benötigten Zeit für die Berechnung lohnt es sich, manches selber zu programmieren
  – Java bietet die Klasse util.Hashtable, die aber langsamer ist als eine eigene Implementierung
  – Vor allem auf die IO Routinen muss man achten, da hier die Hauptlast entsteht