Datenbanksysteme II: Implementing Joins

Ulf Leser
Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join
Join Operator

- Join: Highly **time-critical operator**
  - Required in virtually all queries and in all applications
  - Often appears in groups (multi-way joins – much theory)
  - Problem: May create very large results,
  - Only relation op with worse than linear WC runtime: $O(n\times m)$
  - Many variations, suited for different situations

- Example: `SELECT * FROM R, S WHERE R.B = S.B`

```
A  B
A1 0
A2 1
A3 2
A4 1

R
B  C
1  C1
2  C2
1  C3
3  C4
1  C5

S
A  B  C
A2 1  C1
A2 1  C3
A2 1  C5
A3 2  C2
A4 1  C1
A4 1  C3
A4 1  C5
```

$R \bowtie S$
Implementation 1: Nested-loop Join

• Super-naïve

\[
\begin{align*}
&\text{FOR EACH } r \text{ IN } R \text{ DO} \\
&\quad \text{FOR EACH } s \text{ IN } S \text{ DO} \\
&\quad \quad \text{LOAD block}(r) \text{ into } M; \\
&\quad \quad \text{LOAD block}(s) \text{ into } M; \\
&\quad \quad \text{IF } (r.B=s.B) \text{ THEN OUTPUT } (r \bowtie s)
\end{align*}
\]

• Obvious improvement

\[
\begin{align*}
&\text{FOR EACH block } x \text{ IN } R \text{ DO} \\
&\quad \text{READ } x \text{ into } M; \\
&\quad \text{FOR EACH block } y \text{ IN } S \text{ DO} \\
&\quad \quad \text{READ } y \text{ into } M; \\
&\quad \quad \text{FOR EACH } r \text{ in } x \text{ DO} \\
&\quad \quad \quad \text{FOR EACH } s \text{ in } y \text{ DO} \\
&\quad \quad \quad \quad \text{IF } (r.B=s.B) \text{ THEN OUTPUT } (r \bowtie s)
\end{align*}
\]
Cost Estimation

- Let $b(R)$, $b(S)$ be 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 ops)
- Altogether IO: $b(R)+b(R)\times b(S)$
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
- But choice doesn’t really matter here ...
- Can’t we do better?
• There is no “m” in the formula
  – m: Size of main memory in blocks
• We are not using our available main memory
  – Only two blocks for reading and one for writing
• Rule of thumb: Use all memory you can get
  – Use all memory the buffer manager allocates to your process
Implementation 2: Blocked Nested-Loop Join

- **Blocked-nested-loop**

  FOR $i=1$ TO $\frac{b(R)}{(m-1)}$ DO
  
  READ NEXT $m-1$ blocks of $R$ into $M$
  
  FOR EACH block $y$ IN $S$ DO
  
  READ BLOCK $y$ into $M$
  
  FOR EACH $r$ in $R$-chunk DO
  
  FOR EACH $s$ in $y$ do
  
  IF ($r.B=s.B$) THEN OUTPUT ($r \bowtie s$)
Cost

- Outer relation is read once – in chunks
- Inner relation is read once for every chunk of R
- There are $\sim b(R)/m$ chunks
- Total IO: $b(R) + b(R) \times b(S)/m$
- Further advantage: Chunks of outer relation are read sequentially
Example

- Assume $b(R)=10.000$, $b(S)=2.000$, $m=500$
- $R$ as outer relation: $10.000 + 10.000 \times 2.000/500 = 50.000$
- $S$ as outer relation: $2.000 + 2.000 \times 10.000/500 = 42.000$
- Again: Use **smaller relation as outer relation**
- 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 the 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
Content of this Lecture

- Nested loop and blocked nested loop
- **Sort-merge join**
- Hash-based join strategies
- Index join
Sort-Merge Join

- Sort both relations on join attribute(s)
- Merge both sorted relations
- Caution if join values appear multiple times
  - The result size is $|R| \times |S|$ in worst case
  - If there are $r$ and $s$ tuples with value $x$ in the join attribute in $R$ and $S$, respectively, we need to output $r \times s$ tuples for $x$
### Example

#### Table 1: Initial A and B Values

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

#### Table 2: Updated A and B Values

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

#### Table 3: B and C Values

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

#### Table 4: Merged A, B, and C Values

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>1</td>
<td>C3</td>
</tr>
<tr>
<td>A2</td>
<td>1</td>
<td>C5</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>C5</td>
</tr>
<tr>
<td>A3</td>
<td>2</td>
<td>C2</td>
</tr>
</tbody>
</table>
Merge Phase

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

\[
\begin{align*}
\quad \text{WHILE NOT EOR(R) and } r[B] = b \text{ DO} \\
\quad \quad \text{FOR EACH } e \text{ in } B \text{ DO} \\
\quad \quad \quad \text{OUTPUT } (r,e); \\
\quad \quad \quad r := \text{next (R)}; \\
\quad \quad \text{END DO;}
\end{align*}
\]

\[
\text{END DO;}
\]

Code ignores other than join attributes
Cost estimation

- Sorting R costs $\sim 2 \cdot b(R) \cdot \text{ceil}(\log_m(b(R)))$
- Sorting S costs $\sim 2 \cdot b(S) \cdot \text{ceil}(\log_m(b(S)))$
- Merge phase reads each relation once
- Total: $b(R) + b(S) + 2 \cdot b(R) \cdot \text{ceil}(\log_m(b(R))) + 2 \cdot b(S) \cdot \text{ceil}(\log_m(b(S)))$
- 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
- If sort was performed already somewhere down in the tree, sort phase can be skipped
Better than Blocked-Nested-Loop?

- Assume $b(R) = 10,000$, $b(S) = 2,000$, $m = 500$
  - BNL costs 42,000 (with S as outer relation)
  - SM: $10,000 + 2,000 + 4 \times 10,000 + 4 \times 2,000 = 60,000$
  - Improved SM: 36,000

- Assume $b(R) = 1,000,000$, $b(S) = 1,000$, $m = 500$
  - BNL costs $1000 + 1,000,000 \times 1000/500 = 2,001,000$
  - SM: $1,000,000 + 1,000 + 6 \times 1,000,000 + 4 \times 1,000 = 7,005,000$

- When is SM better than BNL?
  - Consider improved version with
    - $2 \times b(R) \times \lceil\log_m(b(R))\rceil + 2 \times b(S) \times \lceil\log_m(b(S))\rceil - b(R) - b(S) \sim$
    - $2 \times b(R) \times (\log_m(b(R)) + 1) + 2 \times b(S) \times (\log_m(S) + 1) - b(R) - b(S) =$
    - $2 \times b(R) \times \log_m(b(R)) + 2 \times b(S) \times \log_m(S) + b(R) + b(S) \sim$
    - $b(R) \times (2 \times \log_m(b(R)) + 1) + b(S) \times (2 \times \log_m(S) + 1)$
  - Compare to BNL: $b(R) + b(R) \times b(S)/m$
Comparison

- Assume two relations of equal size $b$
- SM: $2b(2\log_m(b)+1)$
- BNL: $b + \frac{b^2}{m}$
- BNL > SM iff
  - $b + \frac{b^2}{m} > 2b(2\log_m(b)+1)$
  - $1 + \frac{b}{m} > 4\log_m(b) + 2$
  - $b > 4m\log_m(b) + m$
- Example
  - $b=10.000$, $m=100$ ($10.000 > 500$)
    - BNL: $10.000 + 1.000.000$, SM: $6\times10.000 = 60.000$
  - $b=10.000$, $m=5000$ ($10.000 < 25.000$)
    - BNL: $10.000 + 20.000$, SM: $6\times10.000 = 60.000$
Comparison 2

- $b(R) = 1.000.000$, $b(S) = 2.000$, $m$ between 100 and 90.000

- BNL very good if one relation is much smaller than other and sufficient memory available (~1 pass suffices)
- SM can better cope with limited memory (and can be pipelined)
Comparison 3

- $b(R)=1.000.000$, $b(S)=50.000$, m between 500 and 90.000

- BNL very sensible to small memory sizes
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 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
Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join
Hash Join

- As usual, we can avoid sorting if a good hash function is available
- Assume a very good hash function
  - Distributes hash values uniformly over hash table
  - If we have good histograms (later), a simple interval-based hash function might help a lot
- How can we apply hashing to joins?
Idea

- Use join attribute(s) as hash keys in both R and S
  - Assume hash table of size m (use all memory)
  - Each bucket will have size approx. $b(R)/m$ or $b(S)/m$

- Hash phase
  - Scan R, add to bucket, writing full blocks to disk immediately
  - Scan S, add to bucket, writing full blocks to disk immediately
  - [Better to use some $n < b(R)/m$ to allow for sequential writes]

- Merge phase
  - Iteratively, load same buckets of R and of S (assume we can)
  - Compute join in memory
Comparing Join Methods

Nested-Loops-Join

Merge-Join

Hash-Join
• Assume we can always load both buckets into main memory
• Hash phase: 2\*b(R) + 2\*b(S)
• Merge phase: b(R) + b(S)
• Total: 3\*(b(R)+b(S))

• What happens if hash function creates skew?
Hash Join with Large Tables

• Merge phase assumes two buckets can be held in memory
  – For uniform hashing: \( b(R)/m < m \), \( b(S)/m < m \)
  – Note: Merge phase of sorting requires \(|\text{runs}| \) blocks (where runs have equal and fixed size), hashing requires 2 buckets to be loaded (where buckets need not have equal and restricted size)

• What if not?
  – Two phase hash join: First partition \( R \) and \( S \) such that each partition hopefully has buckets smaller than \( m^2/2 \)
  – Compute buckets for all partitions in both relations
  – Merge in cross-product manner
    • \( P_{ABC} \): Relation \( A \), partition \( B \), hashkey \( C \)
    • \( P_{R,1,1} \) with \( P_{S,1,1,}, P_{S,2,1}, \ldots, P_{S,n,1} \)
    • \( P_{R,2,1} \) with \( P_{S,1,1,}, P_{S,2,1}, \ldots, P_{S,n,1} \)
    • \( \ldots \)
    • \( P_{R,m,k} \) with \( P_{S,1,k}, P_{S,2,k}, \ldots, P_{S,n,k} \)
Improvement

- Actually, it suffices if either $b(R)$ or $b(S)$ is small enough
- Load smaller bucket into main memory
  - And sort for faster look-up
- Load same bucket in other relation block by block and filter tuples
Cost (with Partitioning)

Assume \( b(R) = b(S) = b \)

How many partitions \((p)\) do we need (if buckets are of equal size)?
- Goal: For each partition \( P \), \( b(P) < m^2/2 \)
- Hence: \( b/p \sim m^2/2 \), or \( p \sim 2b/m^2 \)

In each partition, there are (still) \( m \) buckets of size \( \sim m/2 \)

Hash/partition phase: \( 2b + 2b \) (partitions are not materialized)

Merge phase: \( b + p \times m \times p \times m/2 = b + p^2m^2/2 = b + 2b^2/m^2 \)
- There are \( p \times m \) buckets in outer relation
- For each bucket of outer relation, we have to read \( p \) buckets of inner relation, each of size \( m/2 \)
Alternative

- Accept overly large buckets
- Perform **blocked-nested loop for each pair** of buckets
- There are m buckets, each of size n=b/m (>m/2)
- Hash phase: 2b+2b
- BNL phase: m * (n + n*n/m) = m*(b/m+b²/m³) = b+b²/m²
  - There are m bucket pairs
  - For each, we perform blocked nested loop over two buckets of size n
- Note: Since in fact only one relation must be small enough, the cross-product large hash join has app. the same cost
Hybrid Hash Join

• Assume that \( \min(b(R), b(S)) < \frac{m^2}{2} \)
• Note: During merge phase, we used only \( \frac{(b(R) + b(S))}{m} \) memory blocks (size of two buckets)
• This does usually not fill the entire memory
• Improvement
  – Chose smaller relation (assume S)
  – Chose a number k of buckets (with \( k < m \))
    • Again, assuming perfect hash functions, each bucket has size \( \frac{b(S)}{k} \)
  – When hashing S, keep first \( x \) buckets completely in memory, but only one block for each of the \( (k-x) \) other buckets
    • These first \( x \) buckets are never written to disk
– ... 

– When hashing R
  • If hash value maps into buckets 1..x, perform join immediately
  • Otherwise, map to the k-x other buckets and write to disk
– After first round, we have already computed the join on x buckets and have k-x buckets of both relations on disk
– Perform “normal” merge phase on k-x buckets
Cost

- Total saving (compared to normal hash join)
  - We save 2 IO for every block in either relation that is never written
  - We keep x buckets in memory, having \( \approx \frac{b(S)}{k} \) and \( \approx \frac{b(R)}{k} \) blocks
  - Together, we save \( 2\times x \times \frac{(b(S)+b(R))}{k} \) IO operations

- How should we choose k and x?

- **Best 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
  - Optimum reached at \( k \approx \frac{b(S)}{m} \)
    - Note: \( k \) must be a little smaller: One block for each other bucket

- Together, we save \( 2 \times \frac{(b(S)+b(R))}{m} \)

- Total cost: \( (3-\frac{2m}{b(S)}) \times (b(S)+b(R)) = 6b-4m \)
  - With \( b=b(R)=b(S) \)
Quantitative Comparison

- BNLJ sensitive to memory and size differences
- HJ with robust performance, sometimes better, sometimes worse than SMJ
Comparing Hash Join and Sort-Merge Join

- With **enough memory**, both require approximately the same number of IO
  - Hybrid-hash join improves slightly
- SM generates **sorted results** – sort phase of other joins in query plan can be dropped, **entire queries** get faster
- HJ does not need to perform sorting in main memory
- HJ only requires that **one relation** is “small enough”
- HJ only performs well if we have **equally sized buckets**
  - Otherwise, performance might degrade due to unexpected paging
  - To prevent, estimate $k$ conservative and do not fill $m$ completely
- Both can be tuned to generate **more sequential IO**
Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join
Index Join

- Assume we have an index “B_Index” on join attribute B in one relation

- Choose indexed relation as inner relation

  ```plaintext
  FOR EACH r IN R DO
      X = { SEARCH (S.B_Index, <r.B>) }
      FOR EACH TID i in X DO
          s = READ (S, i) ; output (r \Join s).
  ```

- Nested loop with index access
Cost

- Typical situation: R.B is **primary key**, S.B is **foreign key**
  - Every tuple from R has zero, one or more join tuples in S
- Let $v(X,B)$ be # of unique values of B in relation X
  - Each value in S.B appears $v \sim |S|/v(S,B)$ times
- For each $r \in R$, we read all tuples with given value in S
- Assume every $r$ has at least one join partner:
  \[
  b(R) + |R|*(\log_k(|S|)) + v/k + v
  \]
  - Outer relation read once
  - Find value in B*-tree index, read all matching TIDs (with block size $k$), access S for each TID (assume they are all in different blocks)
- Assume only $r$ tuples of R have partner:
  \[
  b(R) + |R|*\log_k(|S|) + r(v/k + v)
  \]
Comparison

• Compare to sort-merge join
  – Neglect $\log_k(|S|) + v/k$
    • First term is mostly $\sim 2$, second mostly $\sim 1$
  – $SM > IJ$ roughly requires
    • Assume that 2 passes suffice for sorting
    • $3*(b(R)+b(S)) > b(R)+|R|*b(S)/v(S,B)$

• Example
  – $b(R)=10.000$, $b(S)=2.000$, $m=500$, $v(S,B)=10$, $k=50$
  – $SM: 36.000$
  – $IJ: 10.000 + 10.000*50*2.000/10 \sim 1.000.000.000$

• When is an index join a good idea?
Index Join: Advantageous Situations

- When r is really small
  - The join is highly selective – few tuples find a partner
  - For instance, if join is combined with selection on R
  - Most tuples are filtered, only very few require access to S

- When r is very small, R.B is foreign key, S.B is primary key
  - Similar to previous case
  - If S is primary key, then \( v(S,B) = |S| \), and hence \( v = 1 \)
  - R can be read fast and “probes” into S
Index Join with Sorting

- **Note**: Blocks of S are read many times
  - Caching will reduce the overhead – difficult to predict

- **Alternative**
  - First compute all necessary TID’s from S
  - Sort and read tuples from S in **sorted order**
    - Sort by TID and hope that tuples didn’t move too often and TIDs are created in sequential order
  - Advantage: Blocks of S sometimes will be in cache when accessed
  - Requires enough memory for keeping TID list and join tuples of R
  - Pipeline breaker
Index Join with 2 Indexes

- Assume we have an index on both join attributes
- What are we doing?
Index Join with 2 Indexes

- **TID-list join**
- Read both indexes sequentially
- Join (value, TID) lists on value
- Probe into R and S only if necessary
- Large advantage if intersection is small
  - Because indexes are much more compact than data blocks and data blocks are almost never accessed
- Otherwise, we need sorted tables (index-organized)
  - But then sort-merge is probably faster