Computing Joins

- The cost of joining two relations makes the choice of a join algorithm crucial
- Simple block-nested loops join algorithm for computing \( r \bowtie_{A=B} s \)

\[
\text{foreach page } p_r \text{ in } r \text{ do}
\text{foreach page } p_s \text{ in } s \text{ do}
\text{output } p_r \bowtie_{A=B} p_s
\]

- If we do this in tuple level, \( \text{Page}(R) + \text{Tuple}(R) \ast \text{Page}(S) \)
- Consider that \( \text{Page}(R) = 1000 \), \( \text{Page}(S) = 100 \), \( \text{tuple}(R) = 10,000 \),
  - If outer loop is for \( R \), \( 1000 + 10000 \ast 100 = 1,001,000 \) page transfer. --- too many…
  - If outer loop is for \( S \),
  - \( 100 + 1000 \ast 1000 = 1,000,100 \) page transfer. --- fewer, too many…
Block-Nested Loops Join

- If $\beta_r$ and $\beta_s$ are the number of pages in $r$ and $s$, the cost of algorithm is

$$\beta_r + \beta_r \cdot \beta_s + \text{cost of outputting final result}$$

- If $r$ and $s$ have $10^3$ pages each,
  cost is $10^3 + 10^3 \cdot 10^3$

- Choose smaller relation for the outer loop:
  - If $\beta_r < \beta_s$ then $\beta_r + \beta_r \cdot \beta_s < \beta_s + \beta_s \cdot \beta_s$

- Cost can be reduced to

$$\beta_r + \left(\frac{\beta_r}{(M-2)}\right) \cdot \beta_s + \text{cost of outputting final result}$$

by using $M$ buffer pages instead of 1.

**FIGURE 10.6** Block-nested loops join.
Index-Nested Loop Join $r \Join_{A=B}s$

- Use an index on $s$ with search key $B$ (instead of scanning $s$) to find rows of $s$ that match $t_r$
  - Cost = $\beta_r + \tau_r \omega + \text{cost of outputting final result}$

- Effective if number of rows of $s$ that match tuples in $r$ is small (i.e., $\omega$ is small) and index is clustered

```java
foreach tuple $t_r$ in $r$ do {
    use index to find all tuples $t_s$ in $s$ satisfying $t_r.A = t_s.B$;
    output ($t_r, t_s$)
}
```
Sort-Merge Join \( r \bowtie_{A=B} s \)

- sort \( r \) on \( A \);
- sort \( s \) on \( B \);
- while \(!\text{eof}(r)\) and \(!\text{eof}(s)\) do {
  - Scan \( r \) and \( s \) concurrently until \( t_r.A = t_s.B = c \);
  - Output \( \sigma_{A=c}(r) \times \sigma_{B=c}(s) \)
}

Join During Merge Illustrated

\[ r \bowtie_{A=B} s \]

\[ r \]

\[ D \]

\[ \begin{array}{cccccccc}
   A & 1 & 3 & 0 & 9 & 8 & 7 & 3 & 5 & 7 & 1 & 1 \\
   p & p & p & q & q & s & s & s & u & u & u & v \\
\end{array} \]

\[ A \]

\[ \begin{array}{cccccccc}
   B & p & p & r & s & t & t & u & u & x \\
   4 & 0 & 9 & 7 & 2 & 5 & 2 & 5 & 0 & 0 \\
\end{array} \]

\[ r \]

\[ s \]

\[ \begin{array}{cccccccc}
   1 & 3 & 1 & 3 & 5 & 7 & 5 & 7 & 5 & 7 & 5 & 7 \\
   p & p & p & s & s & s & u & u & u & u & u & u \\
\end{array} \]

\[ \begin{array}{cccccccc}
   4 & 0 & 0 & 4 & 7 & 7 & 7 & 2 & 2 & 5 & 0 & 0 \\
\end{array} \]

\[ 1 \]

\[ 0 \]

\[ \]
Cost of Sort-Merge Join

- Cost of sorting assuming $M$ buffers:
  - $2 \beta_r \log_{M-1} \beta_r + 2 \beta_s \log_{M-1} \beta_s$

- Cost of merging:
  - Scanning $\sigma_{A \leftarrow C}(r)$ and $\sigma_{B \leftarrow C}(s)$ can be combined with the last step of sorting of $r$ and $s$ --- costs nothing
  - Cost of $\sigma_{A \leftarrow C}(r) \times \sigma_{B \leftarrow C}(s)$ depends on whether $\sigma_{A \leftarrow C}(r)$ can fit in the buffer
    - If yes, this step costs 0
    - In not, each $\sigma_{A \leftarrow C}(r) \times \sigma_{B \leftarrow C}(s)$ is computed using block-nested join, so the cost is the cost of the join. (Think why indexed methods or sort-merge are inapplicable to Cartesian product.)

- Cost of outputting the final result depends on the size of the result

Hash-Join $r \bowtie_{A=B} s$

- **Step 1**: Hash $r$ on $A$ and $s$ on $B$ into the same set of buckets
- **Step 2**: Since matching tuples must be in same bucket, read each bucket in turn and output the result of the join
- **Cost**: $3 (\beta_r + \beta_s) + \text{cost of output of final result}$
  - assuming each bucket fits in memory
**Hash Join**

- **r** \( \bowtie \) \( \text{cond}_1 \) \( r_1 \) \( \bowtie \) \( \text{cond}_2 \) ... \( \bowtie \) \( \text{cond}_n \) \( r_n \)
- Each \( \text{cond}_i \) involves only the attributes of \( r_j \) and \( r \)

**Star Joins**

- \( r \bowtie \bowtie \text{cond}_1 \ r_1 \bowtie \bowtie \text{cond}_2 \ r_2 \bowtie \bowtie \text{cond}_n \ r_n \)
- Each \( \text{cond}_i \) involves only the attributes of \( r_j \) and \( r \)
Star Join

- Use join index
  - Scan \( r \) and the join index \( \{<r, r_1, ..., r_n>\} \) (which is a set of tuples of rids) in one scan
  - Retrieve matching tuples in \( r_1, ..., r_n \)
  - Output result

Computing Star Joins
Computing Star Joins

- Use *bitmap indices*
  - Use one bitmapped join index, $J_i$, per each partial join $r_j \bowtie_{\text{cond}_i} r_i$
  - Recall: $J_i$ is a set of <$v, \text{bitmap}$>, where $v$ is an rid of a tuple in $r_i$ and \text{bitmap} has 1 in $k$-th position iff $k$-th tuple of $r$ joins with the tuple pointed to by $v$

1. Scan $J_i$ and logically OR all bitmaps. We get all rids in $r$ that join with $r_i$
2. Now logically AND the resulting bitmaps for $J_1, ..., J_n$.
3. Result: a subset of $r$, which contains all tuples that can possibly be in the star join
   - Rationale: only a few such tuples survive, so can use indexed loops

Computing Aggregated Functions

- Require full scan
- In case that tuples are *grouped by attributes*,
  - Need to partition relation with the attribute values
    - Sorting
    - Hashing
    - Indexing
Choosing Indices

- DBMSs may allow user to specify
  - Type (hash, B+ tree) and search key of index
  - Whether or not it should be clustered

- Using information about the frequency and type of queries and size of tables, designer can use cost estimates to choose appropriate indices

- Several commercial systems have tools that suggest indices
  - Simplifies job, but index suggestions must be verified

Choosing Indices – Example

- If a frequently executed query that involves selection or a join and has a large result set,
  - Use a clustered B+ tree index
  - e.g., Retrieve all rows of Transcript for StudId

- If a frequently executed query is an equality search and has a small result set,
  - An unclustered hash index is best, since only one clustered index on a table is possible, choosing unclustered allows a different index to be clustered
  - e.g., Retrieve all rows of Transcript for (StudId, CrsCode)