CSE 305 / CSE532

Lecture 19 (Chapter 10)
Query Processing: The Basics

Lecturer: Sael Lee

Slide adapted from the author’s, Peter Bailis’s and Dr. Ilchul Yoon’s slides.
Query Processing Example

Select B,D
From R,S
Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C
<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example cont.**

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Answer**

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>x</td>
</tr>
</tbody>
</table>
How do we execute query?

One idea

- Do Cartesian product
- Select tuples
- Do projection
<table>
<thead>
<tr>
<th>RXS</th>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RXS</td>
<td>R.A</td>
<td>R.B</td>
<td>R.C</td>
<td>S.C</td>
<td>S.D</td>
<td>S.E</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bingo!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Got one...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Peter Bailis’s slides
Relational Algebra - can be used to describe plans...

Plan I

\[
\Pi_{B,D} \\
\sigma_{R.A = "C" \land S.E = 2 \land R.C = S.C} \\
\times \\
R \quad S
\]

OR: \[ \Pi_{B,D} [ \sigma_{R.A = "C" \land S.E = 2 \land R.C = S.C} (R \times S) ] \]
Another idea:

Plan II

\[ \Pi_{B,D}( \sigma_{R.A = "c"}(R) \bowtie \sigma_{S.E = 2}(S) ) \]

natural join
```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
</tr>
</tbody>
</table>

\[ \sigma(R) \]

\[ \sigma(S) \]

\[ C \quad D \quad E \]

\[ 10 \times 2 \]

\[ 20 \quad y \quad 2 \]

\[ 30 \quad z \quad 2 \]

\[ 40 \quad x \quad 1 \]

\[ 50 \quad y \quad 3 \]
```
Plan III: Utilizing Index

Use R.A and S.C Indexes

(1) Use R.A index to select R tuples with R.A = “c”
(2) For each R.C value found, use S.C index to find matching tuples

(3) Eliminate S tuples S.E ≠ 2
(4) Join matching R,S tuples, project B,D attributes and place in result
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>
(1) Use R.A index to select R tuples with R.A = “c”
### (2) For each R.C value found, use S.C index to find matching tuples

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>
(3) Eliminate S tuples $S . E \neq 2$
(4) Join matching R,S tuples, project B,D attributes and place in result
### Table R

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
</tr>
</tbody>
</table>

### Table S

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>

### Diagram

1. **A = “c”**
   - **I₁**  \(<c,2,10>\)
   - **I₂**  \(<10,x,2>\)

2. **check = 2?**
   - **output**: \(<2,x>\)

3. **next tuple**: \(<c,7,15>\)
External Sorting

• Sorting is used in implementing many relational operations

• Problem:
  • Relations are typically large, do not fit in main memory
  • So cannot use traditional in-memory sorting algorithms

• Approach used:
  • Combine in-memory sorting with clever techniques aimed at minimizing I/O
  • I/O costs dominate => cost of sorting algorithm is measured in the number of page transfers
External Sorting (cont’d)

- External sorting has two main components:
  - Computation involved in sorting records in buffers in main memory
  - I/O necessary to move records between mass store and main memory
Simple Sort Algorithm

- $M =$ number of main memory page buffers
- $F =$ number of pages in file to be sorted
- Typical algorithm has two phases:
  - 1 Partial sort phase: sort $M$ pages at a time; create $F/M$ sorted runs on mass store, cost = $2F$

Example: $M = 2$, $F = 7$

Original file

Partially sorted file

Example: $M = 2$, $F = 7$
Simple Sort Algorithm

- **2 Merge Phase**: merge all runs into a single run using M-1 buffers for input and 1 output buffer
  - Merge step: divide runs into groups of size $M-1$ and merge each group into a run; cost = $2F$
  - *Each step reduces number of runs by a factor of $M-1$*

![Diagram of k-way merge](image_url)

**FIGURE 10.2** k-way merge.
Merge: An Example

Input runs

2 3 5 6
1 7 10 15

Input buffers

2 3 5 6
1 7 10 15

Output run

1 2 3 5 6 7 10 15

Output buffer

8 6 101 25
10 75
Duplicate Elimination

- A major step in computing *projection*, *union*, and *difference* relational operators

**Algorithm:**
- Sort
- At the last stage of the merge step eliminate duplicates on the fly
- No additional cost (with respect to sorting) in terms of I/O
Duplicate elimination During Merge

Input runs

Input buffers

Output buffer

Last key used

Output run

Key 3 ignored: duplicate
Key 5 ignored: duplicate
Sort-Based Projection

- **Algorithm:**
  - Sort rows of relation at cost of $2F \log_{M-1} F$
  - Eliminate unwanted columns in partial sort phase (no additional cost)
  - Eliminate duplicates on completion of last merge step (no additional cost)

- **Cost:** the cost of sorting
Hash-Based Projection

- **Phase 1:**
  - Input rows
  - Project out columns
  - Hash remaining columns using a hash function with range $1...M-1$ creating $M-1$ buckets on disk
  - **Cost** = $2F$

- **Phase 2:**
  - Sort each bucket to eliminate duplicates
  - **Cost** (assuming a bucket fits in $M-1$ buffer pages) = $2F$

- **Total cost** = $4F$
Comparison

- Assume
  - M=10000-page buffer (40MB) use as hash table
  - We have F=10^8-page file to process (400GB = 40M*10000)

- Hash-based projection
  - 4*10^8

- Sort-based projection
  - \[ 2F \log_{(M-1)} F = 2 \times 10^8 \times \log_{10^4-1} 10^8 \geq 4 \times 10^8 \]

- However, it requires
  - Even distribution from hash function
  - In-memory sort of each bucket
Computing Selection $\sigma_{(attr \ op \ value)}$

- **No index on attr:**
  - **If rows are not sorted on attr:**
    - Scan all data pages to find rows satisfying selection condition
    - Cost = $F$
  - **If rows are sorted on attr and op is =, >, < then:**
    - Use *binary search* (at $\log_2 F$) to locate first data page containing row in which $(attr = value)$
    - Scan further to get all rows satisfying $(attr \ op \ value)$
    - Cost = $\log_2 F + (\text{cost of scan})$
Computing Selection $\sigma_{(\text{attr } \text{op } \text{value})}$

- **Clustered** $B^+$ tree index on $\text{attr}$ (for “=” or range search):
  - Locate first index entry corresponding to a row in which $(\text{attr} = \text{value})$.
    - **Cost**: depth of tree
  - **Rows** satisfying condition packed in sequence in successive data pages; *scan those pages*.
    - **Cost**: number of pages occupied by qualifying rows
Computing Selection $\sigma(\text{attr } \text{op} \text{ value})$

- **Unclustered $B^+$ tree index on attr (for “=” or range search):**
  - Locate first index entry corresponding to a row in which $(\text{attr} = \text{value})$.
    - **Cost** = depth of tree
  - **Index entries** with pointers to rows satisfying condition are packed in sequence in successive index pages
    - Scan entries and sort record IDs to identify table data pages with qualifying rows; Any page that has at least one such row must be fetched once.
    - **Cost** = number of rows that satisfy selection condition
Unclustered B⁺ Tree Index

index entries (containing row Ids) that satisfy condition

B⁺ Tree

data page

Data file
Computing Selection $\sigma_{\text{attr} = \text{value}}$

- **Hash index** on attr (for “=” search only):
  - Hash on value. Cost (of finding the right bucket) $\approx 1.2$
    - 1.2 – typical average cost of hashing (> 1 due to possible overflow chains)
  - Finds first the (unique) bucket containing all index entries satisfying selection condition. Then,
  - **Clustered index** – all qualifying rows packed in the bucket (a few pages)
    Cost: number of pages occupies by the bucket
  - **Unclustered index** – sort row Ids in the index entries to identify data pages with qualifying rows
    Each page containing at least one such row must be fetched once
    Cost: min(number of qualifying rows in bucket, number of pages in file)
Computing Selection $\sigma_{(attr = value)}$

- Unclustered hash index on $attr$ (for equality search)
Access Path

- **Access path** is the notion that denotes *algorithm + data structure* used to locate rows satisfying some condition.

**Examples:**

- **File scan**: can be used for any condition.
- **Hash**: equality search; *all* search key attributes of hash index are specified in condition.
- **B+ tree**: equality *or* range search; a *prefix* of the search key attributes are specified in condition.
  - B+ tree supports a variety of access paths.
- **Binary search**: relation sorted on a sequence of attributes and some *prefix* of that sequence is specified in condition.
Access Paths Supported by B\(^+\) tree

- **Example**: Given a B\(^+\) tree whose search key is the sequence of attributes \(a_2, a_1, a_3, a_4\)
  - Access path for search \(\sigma_{a_1 > 5 \text{ AND } a_2 = 3 \text{ AND } a_3 = 'x'}\) (\(R\)):
    - find first entry having \(a_2 = 3\) AND \(a_1 > 5\) AND \(a_3 = 'x'\) and scan leaves from there until entry having \(a_2 > 3\) or \(a_3 \neq 'x'\). Select satisfying entries
  - Access path for search \(\sigma_{a_2 = 3 \text{ AND } a_3 > 'x'}\) (\(R\)):
    - locate first entry having \(a_2 = 3\) and scan leaves until entry having \(a_2 > 3\). Select satisfying entries
  - Access path for search \(\sigma_{a_1 > 5 \text{ AND } a_3 = 'x'}\) (\(R\)):
    - Scan of \(R\)
Choosing an Access Path

- **Selectivity** of an access path = number of pages retrieved using that path
  - If several access paths support a query, DBMS chooses the one with lowest selectivity
  - Size of domain of attribute is an indicator of the selectivity of search conditions that involve that attribute

Example: \( \sigma_{\text{CrsCode}=\text{CS305} \text{ AND Grade}=\text{B}} \) (Transcript)

- Assume that we have *two* \( B^+ \) trees; one with search key \( \text{CrsCode} \), and the other with \( \text{Grade} \)
- A \( B^+ \) tree with search key \( \text{CrsCode} \) has lower selectivity than a \( B^+ \) tree with search key \( \text{Grade} \)
Selections with Complex Conditions

- **Selection with conjunctive conditions**
  - Use the most selective access path to retrieve the corresponding tuples
    - e.g., one condition is for an indexed attribute
  - Use several access paths that cover the expression
    - e.g., use the most selective first, and use the other ones.

- **Selection with disjunctive conditions**
  - If the condition contain disjunctions, convert to disjunctive normal form. (disjunction of conjunctive conditions)
  - Check available access paths for the individual disjuncts and choose the appropriate strategy
    - e.g., what if a disjunct need file scan?
    - e.g., what if each disjunct has better access path than file scan?
Computing Joins

- The cost of joining two relations makes the choice of a join algorithm crucial

**Simple block-nested loops** join algorithm for computing

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

```plaintext
foreach page pr in r do
  foreach page ps in s do
    output pr \bowtie_{A=B} ps
```

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

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

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

- **Choose smaller relation for the outer loop:**
  - If $\beta_r < \beta_s$ then $\beta_r + \beta_r \times \beta_s < \beta_s + \beta_r \times \beta_s$
Block-Nested Loops Join

- Cost can be reduced to
  \[
  \beta_r + \left(\frac{\beta_r}{(M-2)}\right) \times \beta_s + \text{cost of outputting final result}
  \]
  by using M buffer pages instead of 1.

**FIGURE 10.6** Block-nested loops join.
Block-Nested Loop Illustrated

Input buffer for r

Input buffer for s

Output buffer

...and so on
Index-Nested Loop Join  \( r \bowtie_{A=B} s \)

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

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

```
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 $r \bowtie_{A=B} s$

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

$\sigma_{A=c}(r)$

$\sigma_{B=c}(s)$
Join During Merge Illustrated

\[ r \]

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

\[ \begin{array}{cccccccc}
B & E & 4 & 0 & 9 & 7 & 2 & 5 & 2 & 5 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccccccc}
1 & 3 & 1 & 3 & 8 & 7 & 3 & 5 & 7 & 5 & 7 & 5 & 7 \\
p & p & p & p & s & s & s & u & u & u & u & u & u \\
p & p & p & p & s & s & s & u & u & u & u & u & u \\
4 & 0 & 0 & 4 & 7 & 7 & 7 & 2 & 2 & 5 & 5 & 0 & 0 \\
\end{array} \]

\[ r \bigotimes_{A=B} s \]
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 $\sigma_{A=r}(r)$ and $\sigma_{B=s}(s)$ can be combined with the last step of sorting of $r$ and $s$ --- costs nothing
  - Cost of $\sigma_{A=r}(r) \times \sigma_{B=s}(s)$ depends on whether $\sigma_{A=r}(r)$ can fit in the buffer
    - If yes, this step costs 0
    - In not, each $\sigma_{A=r}(r) \times \sigma_{B=s}(s)$ 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

Stage 1

Stage 2
Star Joins

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

COURSE

<table>
<thead>
<tr>
<th>CrsCode</th>
<th>DeptId</th>
<th>CrsName</th>
<th>Description</th>
</tr>
</thead>
</table>

TEACHING

<table>
<thead>
<tr>
<th>ProflId</th>
<th>CrsCode</th>
<th>Semester</th>
</tr>
</thead>
</table>

TRANSCRIPT

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
</table>

STUDENT

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Status</th>
<th>Address</th>
</tr>
</thead>
</table>
Computing Star Joins

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

- **Use bitmap indices**
  - Use one bitmapped join index, $J_i$, per each partial join
    $r \bowtie_{\text{cond}_i} r_i$
  - **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 $J_1, \ldots, J_n$.
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)