CompSci 516  
Database Systems  

Lecture 10  
Query Evaluation  
and  
Join Algorithms  

Instructor: Sudeepa Roy

Announcement

• Project proposal pdf due on sakai by 5 pm, tomorrow, Thursday 09/27  
  — One per group by any member

Reading Material

• [RG]  
  – Query evaluation and operator algorithms:  
    Chapter 12.2-12.5, 13, 14.1-14.3  
  – Join Algorithm: Chapter 14.4  
  – Set/Aggregate: Chapter 14.5, 14.6

Overview of Query Evaluation

• Two main issues in query optimization:
  1. For a given query, what plans are considered?  
     – Algorithm to search plan space for cheapest (estimated) plan  
  2. How is the cost of a plan estimated?

  • Ideally: Want to find best plan  
  • Practically: Avoid worst plans!
Some Common Techniques

• Algorithms for evaluating relational operators use some simple ideas extensively:
  • Indexing:
    – Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  • Iteration:
    – Examine all tuples in an input tuple
    – Sometimes, faster to scan all tuples even if there is an index
    – And sometimes, we can scan the data entries in an index instead of the table itself – Recall INDEX-ONLY plan – iterate over leaves in a tree
  • Partitioning:
    – By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs

Watch for these techniques as we discuss query evaluation!

System Catalog

• Stores information about the relations and indexes involved
• Also called Data Dictionary (basically a collection of tables itself)
• Catalogs typically contain at least:
  – Size of the buffer pool and page size
  – # tuples (#Tuples) and # pages (#Pages) for each relation
  – # distinct key values (#Keys) and #Pages for each index
  – Index height for each tree index
  – Lowest/highest key values (Low/High) for each index
• More detailed information (e.g., histograms of the values in some field) are sometimes stored
• Catalogs updated periodically.
  – Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok

Access Paths

• A way of retrieving tuples from a table
• Consists of
  – a file scan, or
  – an index + a matching condition
• The access method contributes significantly to the cost of the operator
  – Any relational operator accepts one or more table as input

Index “matching” a search condition

Recall
• A tree index matches (a conjunction of) terms that involve only attributes in a prefix of the search key.
  • E.g., Tree index on <a, b, c> matches the selection
    • a=5 AND b=3,
    • a=5 AND b>6,
    • but b=3
• A hash index matches (a conjunction of) terms that has a term attribute = value for every attribute in the search key of the index.
  • E.g., Hash index on <a, b, c> matches
    • a=5 AND b=3 AND c=5,
    • but it does not match b=3,
    • or a=5 AND b=3,
    • or a=5 AND b=3 AND c=5

Access Paths: Selectivity

• Selectivity:
  – the number of pages retrieved for an access path
  – includes data pages + index pages
• Options for access paths:
  – scan file
  – use matching index
  – scan index

Most Selective Access Paths

• An index or file scan that we estimate will require the fewest page I/Os
  – Terms that match this index reduce the number of tuples retrieved
  – other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
Selectivity: Example 1

- Hash index on sailors <rname, bid, sid>
- Selection condition (rname = ‘Joe’ ∧ bid = 5 ∧ sid = 3)
- # of sailors pages = N
- # distinct keys = K
- Fraction of pages satisfying this condition = (approximately) N/K
- Assumes uniform distribution

Selectivity: Example 2

- Hash index on sailors <bid, sid>
- Selection condition (bid = 5 ∧ sid = 3)
- Suppose $N_1$ distinct values of bid, $N_2$ for sid
- Reduction factors
  - for (bid = 5): $1/N_1$
  - for (bid = 5 ∧ sid = 3): $1/(N_1 × N_2)$
- Assumes independence
- Fraction of pages retrieved or I/O:
  - for clustered index = $1/(N_1 × N_2)$
  - for unclustered index = 1

Selectivity: Example 3

- Tree index on sailors <bid>
- Selection condition (bid > 5)
- Lowest value of bid = 1, highest = 100
- Reduction factor
  - $(100 - 5)/(100 - 1)$
  - assumes uniform distribution
- In general:
  - key > value : (High – value) / (High – Low)
  - key < value : (value – Low) / (High – Low)

Operator Algorithms

Relational Operations

- We will consider how to implement:
  - Join (⋈) Allows us to combine two relations (in detail)
- Also
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Deletes unwanted columns from relation.
  - Set-difference (−) Tuples in reln. 1, but not in reln. 2.
  - Union (∪) Tuples in reln. 1 and in reln. 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be composed
- After we cover each operation, we will discuss how to optimize queries formed by composing them (query optimization)

Assumption: ignore final write

- i.e. assume that your final results can be left in memory
  - and does not be written back to disk
  - unless mentioned otherwise
- Why such an assumption?
Equality Joins With One Join Column

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

- In algebra: R x S
  - Common! Must be carefully optimized
  - R x S is large; so, R x S followed by a selection is inefficient

- Cost metric: # of I/Os
  - Remember, we will ignore output costs (always)
  - the cost to write the final result tuples back to the disk

Common Join Algorithms

1. Nested Loops Joins (NLJ)
   - Simple nested loop join
   - Block nested loop join
   - Index nested loop join

2. Sort Merge Join
   - Very similar to external sort

3. Hash Join

Simple Nested Loops Join

\[
\begin{align*}
& \text{R x S} \\
& \text{foreach tuple r in R do} \\
& \quad \text{foreach tuple s in S where } r_i = s_j \text{ do} \\
& \quad \quad \text{add } <r, s> \text{ to result}
\end{align*}
\]

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: \( M + (p_R \cdot M) \cdot N = 1000 + 100 \cdot 1000 \cdot 500 \text{ I/Os.} \)

- Page-oriented Nested Loops Join:
  - For each page of R, get each page of S
  - and write out matching pairs of tuples \(<r, s>\)
  - where \( r \) is in R-page and \( S \) is in S-page.
  - Cost: \( M \cdot N = 1000 \cdot 1000 \cdot 500 \)

- If smaller relation (S) is outer
  - Cost: \( N \cdot M \cdot N = 500 \cdot 500 \cdot 1000 \)

Block Nested Loops Join

- Simple-Nested does not properly utilize buffer pages (uses 3 pages)
- Suppose have enough memory to hold the smaller relation \( R \) + at least two other pages
  - e.g., in the example on previous slide (S is smaller), and we need \( 500 + 3 = 502 \) pages in the buffer
- Then use one page as an input buffer for scanning the inner
  - one page as the output buffer
  - For each matching tuple \( r \) in R-block, s in S-page, add \( <r, s> \) to result
- Total I/O = \( M + N \)

- What if the entire smaller relation does not fit?
Block Nested Loops Join

- If R does not fit in memory,
  - Use one page as an input buffer for scanning the inner S
  - one page as the output buffer
  - and use all remaining pages to hold "block" of outer R.
- For each matching tuple r in R-block, s in S-page, add <r, s> to result
- Then read next R-block, scan S, etc.

Cost of Block Nested Loops

- R is outer
- B-2 = 100-page blocks
- How many blocks of R?
- Cost to scan R?
- Cost to scan S?
- Total Cost?

Cost to find matching S-tuples?
Cost to scan Reserves?
Cost of retrieving matching S tuples
Assuming uniform distribution, 2.5 reservations per sailor (100,000/40,000)
Cost of retrieving them is 1.2 I/Os depending on whether the index is clustered

Index Nested Loops Join

- Suppose there is an index on the join column of one relation
  - say S
  - can make it the inner relation and exploit the index
  - Cost: M * ( (M*p) * cost of finding matching S tuples)
  - For each R tuple, cost of probing S index (get k*) is about
    - 1-2 for hash index
    - 2-4 for B+ tree.
  - Cost of then finding S tuples (assuming Alt. 2 or 3) depends on clustering
    - See lecture 7-8

Cost of Index Nested Loops

#outer blocks = #pages of outer relation/
blocksize

Cost of Index Nested Loops

- Hash-index (Alt. 2) on sid of Sailing (as inner), sid is a key
- Cost to scan Reserves:
  - 1000 page I/Os, 100*1000 tuples.
- Cost to find matching Sailors tuples?
  - For each Reserves tuples:
    - (Suppose on avg) 1.2 I/Os to get data entry in index
    - + 1 I/O to get (the exactly one) matching Sailors tuple
- Total cost:
  - \(1000 + 100 \times 1000 \times 2.2 = 221,000\) I/Os
Algorithms for Joins

2. SORT-MERGE JOINS

Sort-Merge Join: 1/3

- Advance scan of R until current R-tuple $\geq$ current S-tuple
  - then advance scan of S until current S-tuple $\geq$ current R-tuple
  - do this as long as current R tuple = current S tuple

<table>
<thead>
<tr>
<th>S</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Yuppy</td>
<td>9</td>
<td>35.5</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>5</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>Yuppy</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/1/96</td>
<td>Dustin</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>Lubber</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>101</td>
<td>10/11/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>101</td>
<td>11/12/96</td>
<td>Dustin</td>
<td></td>
</tr>
</tbody>
</table>

Sort-Merge Join: 2/3

- At this point, all R tuples with same value in $R_i$ (current R group) and all S tuples with same value in $S_j$ (current S group)
  - match
  - find all the equal tuples
  - output $<$r, s$>$ for all pairs of such tuples

<table>
<thead>
<tr>
<th>S</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Yuppy</td>
<td>9</td>
<td>35.5</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>5</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>Yuppy</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/1/96</td>
<td>Dustin</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>Lubber</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>101</td>
<td>10/11/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>101</td>
<td>11/12/96</td>
<td>Dustin</td>
<td></td>
</tr>
</tbody>
</table>

Sort-Merge Join: 3/3

- Then resume scanning R and S

<table>
<thead>
<tr>
<th>S</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Yuppy</td>
<td>9</td>
<td>35.5</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>5</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>Yuppy</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/1/96</td>
<td>Dustin</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>Lubber</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>101</td>
<td>10/11/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>101</td>
<td>11/12/96</td>
<td>Dustin</td>
<td></td>
</tr>
</tbody>
</table>

Sort-Merge Join: 3/3

- ... and proceed till end

<table>
<thead>
<tr>
<th>S</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Yuppy</td>
<td>9</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>Yuppy</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/1/96</td>
<td>Dustin</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>Lubber</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>101</td>
<td>10/11/96</td>
<td>Guppy</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>101</td>
<td>11/12/96</td>
<td>Dustin</td>
<td></td>
</tr>
</tbody>
</table>
Sort-Merge Join: 3/3

- ... and proceed till end

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>15.0</td>
</tr>
</tbody>
</table>

WRITE ONE OUTPUT TUPLE

Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>dusty</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>15.0</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Total 7500 I/O

Second merge phase of sort

Example: Sort

- 4N = 500 pages in S
- 8N = 1000 pages in R

Typical Cost:
- \( O(M \log M) + O(N \log N) + (M+N) \)
- ignoring \( B \) as the base of log
- cost of sorting \( R \) + sorting \( S \) + merging \( R, S \)
- The cost of scanning in merge-sort, \( M+N \), could be \( M\times N! \)
- assume the same single value of join attribute in both \( R \) and \( S \)
- but it is extremely unlikely

Cost of Block Nested 15000, 6000, 2500

Cost of Sort

- 100 buffer pages
- Sort \( R \):
  - Pages \( 35 \times 100/100 = 35 \) sorted runs
  - Pages \( 35 \) merge 35 runs
  - Sort cost = \( 35 \times 2 \times 35 \times 2 \times 35 \)
  - \( B = 3500 \) pages
- Similarly, Sort \( S \):
  - \( 80 \times 500 = 40000 \) I/O
  - \( 80 \times 500 \times 2 \) pages
- Total 7500 I/O

Two Phases

1. Partition Phase
- partition \( R \) and \( S \) using the same hash function \( h \)

2. Probing Phase
- join tuples from the same partition (same \( h(\cdot) \) value) of \( R \) and \( S \)
- tuples in different partition of \( h \) will never join
- use a “different” hash function \( h2 \) for joining these tuples
  - (why different – see next slide first)

Example of Cost

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>dusty</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>15.0</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Typical Cost: \( O(M \log M) + O(N \log N) + (M+N) \)

Cost of sort-merge
- cost of \( R \) + cost of \( S \) + merging \( R, S \)

Check yourself:
- Consider the buffer pages 35, 100, 300
- Cost of sort-merge \( = 7500 \) in all three
- Cost of block nested 15000, 6000, 2500

Two Phases

1. Partition Phase
- partition \( R \) and \( S \) using the same hash function \( h \)

2. Probing Phase
- join tuples from the same partition (same \( h(\cdot) \) value) of \( R \) and \( S \)
- tuples in different partition of \( h \) will never join
- use a “different” hash function \( h2 \) for joining these tuples
  - (why different – see next slide first)

Example of Cost

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>dusty</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>15.0</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Typical Cost: \( O(M \log M) + O(N \log N) + (M+N) \)

Cost of sort-merge
- cost of \( R \) + cost of \( S \) + merging \( R, S \)

Check yourself:
- Consider the buffer pages 35, 100, 300
- Cost of sort-merge \( = 7500 \) in all three
- Cost of block nested 15000, 6000, 2500

Two Phases

1. Partition Phase
- partition \( R \) and \( S \) using the same hash function \( h \)

2. Probing Phase
- join tuples from the same partition (same \( h(\cdot) \) value) of \( R \) and \( S \)
- tuples in different partition of \( h \) will never join
- use a “different” hash function \( h2 \) for joining these tuples
  - (why different – see next slide first)

Example of Cost

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>dusty</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>15.0</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>15.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Typical Cost: \( O(M \log M) + O(N \log N) + (M+N) \)

Cost of sort-merge
- cost of \( R \) + cost of \( S \) + merging \( R, S \)

Check yourself:
- Consider the buffer pages 35, 100, 300
- Cost of sort-merge \( = 7500 \) in all three
- Cost of block nested 15000, 6000, 2500

Two Phases

1. Partition Phase
- partition \( R \) and \( S \) using the same hash function \( h \)

2. Probing Phase
- join tuples from the same partition (same \( h(\cdot) \) value) of \( R \) and \( S \)
- tuples in different partition of \( h \) will never join
- use a “different” hash function \( h2 \) for joining these tuples
  - (why different – see next slide first)
Cost of Hash-Join

- In partitioning phase
  - read+write both relns; \(2(M+N)\)
  - In matching phase, read both relns; \(M+N\) I/Os
  - remember – we are not counting final write

- In our running example, this is a total of 4500 I/Os
  - 3 * (1000 + 500)
  - Compare with the previous joins

Sort-Merge Join vs. Hash Join

- Both can have a cost of \(3(M+N)\) I/Os
  - if sort-merge gets enough buffer (see 14.4.2)
- Hash join holds smaller relation (see 14.4.2)
- Hash Join shown to be highly parallelizable
- Sort-Merge less sensitive to data skew
  - also result is sorted

General Join Conditions

- Equalities over several attributes
  - e.g., \(R.\text{sid}=S.\text{sid} \text{ and } R.\text{name}=S.\text{name}\)
  - For Index Nested Loop, build index on \(<\text{sid}, \text{name}>\) (if \(S\) is inner); or use existing indexes on \(\text{sid}\) or \(\text{name}\)
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions
  - e.g., \(R.\text{name} < S.\text{name}\)
  - For Index NL, need (clustered) B+ tree index.
  - Hash Join, Sort Merge Join not applicable

Review: Join Algorithms

- Nested loop join:
  - for all tuples in \(R\), for all tuples in \(S\)
  - variations: block-nested, index-nested
- Sort-merge join
  - like external merge sort
- Hash join

  • Make sure you understand how the I/O varies
  • No one join algorithm is uniformly superior to others
    - depends on relation size, buffer pool size, access methods, skew

Schema for Examples

<table>
<thead>
<tr>
<th>Sailing</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sailors</td>
<td>(id: integer, name: string, rating: integer, age: real)</td>
<td></td>
</tr>
<tr>
<td>Reserves</td>
<td>(id: integer, bid: integer, day: dates, rname: string)</td>
<td></td>
</tr>
</tbody>
</table>

- **Sailors**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
- **Reserves**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages
Selection: 1
No Index, Unsorted Data

Naive approach
- Scan the entire relation
- Check the condition and build answer set

- Cost = 1000 I/O
- If only a few tuples with ‘Joe’
  - expensive, does not use selection

Selection: 2
No Index, Sorted Data

- Here, sorted file on “rname”
- Locate the first tuple that satisfies the condition
- scan the relation until the condition is no longer satisfied

- Cost of binary search = \( \log_2 1000 = 10 \) (approx)
- Cost of scan will depend on #satisfying tuples
  - can range from 0 to 1000 (#pages)

Selection: 3
B+ tree Index

- Search the tree to find the first index entry pointing to a qualifying tuple
- Scan the leaves to find all data entries
- Then retrieve the tuples

- Cost of identifying the starting leaf page:
  - typically 2 or 3 I/O
- Cost of scanning leaves will depend on #such data entries
- Cost of retrieving tuples will depend on (if not alternative 1)
  - = #qualifying tuples
  - whether the index is clustered (probably just one I/O if all tuples fit in a page)
  - or unclustered (could be one I/O per qualifying tuple)

Selection: 4
Hash Index, Equality

- Retrieve the bucket page
- Then retrieve the qualifying tuples

- Cost of retrieving the bucket
  - typically 1 or 2 I/O
- Cost of scanning leaves will depend on #such data entries
- (same as tree) Cost of retrieving tuples will depend on (if not alternative 1)
  - #qualifying tuples
  - whether the index is clustered (probably just one I/O if all tuples fit in a page)
  - if index on a key, just one tuple and one page
  - or unclustered (could be one I/O per qualifying tuple)

Refinement for Unclustered Index for Selections

1. Find qualifying data entries.
2. Sort the rid’s of the data records to be retrieved.
3. Fetch rid’s in order.
   - This ensures that each data page is looked at just once
   - however, no. of such pages likely to be higher than with clustering

General Selection

- What if we have more complex selection conditions?
  - instead of attr <op> value
  - we could have logical AND (\( \land \)) and OR (\( \lor \))

- Two main approaches
Approach 1: Filtering

- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don’t match the index:
  - Consider \( \text{day<8/9/94 AND bid=5 AND sid=3} \)
  - A B+ tree index on \( \text{day} \) can be used
    - then, bid=5 and sid=3 must be checked for each retrieved tuple
  - A hash index on \( \langle \text{bid}, \text{sid} \rangle \) could be used;
    - \( \text{day<8/9/94} \) must then be checked.

Approach 2: Intersection

- If we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries:
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids
  - Retrieve the records and apply any remaining terms.
  - Consider \( \text{day<8/9/94 \land bid=5 \land sid=3} \)
    - if we have a B+ tree index on \( \text{day} \) and an index on \( \text{sid} \), both using Alternative (2)
      - we can retrieve rids of records satisfying \( \text{day<8/9/94} \) using the first
      - rids of records satisfying \( \text{sid=3} \) using the second
      - intersect
      - retrieve records and check bid=5

Handling Disjunctions in Practice

1. convert the query into a union of queries without OR
2. if same attribute, \( A < 5 \lor A > 10 \), use a nested query with an IN and an index
3. simply apply the disjunction condition on the retrieved tuples
4. use bitmap
   - see [RG] 14.2.3.

- Most DBMSs do not handle disjunctions too efficiently, we won’t discuss them in detail

Algorithms for Projection

Projection

- Two parts
  - Remove some fields (easy)
  - Remove duplicates (hard)

- The expensive part is removing duplicates
  - SQL systems don’t remove duplicates unless the keyword DISTINCT is specified in a query
  - Then just scan the table or use index (if the key contains all the necessary fields)
  - Otherwise, need to delete duplicates

Scan R and eliminate unwanted attributes
Sort this set with all attributes as the key for sorting
Scan the sorted result, compare adjacent tuples, discard duplicates

Improvement:
- project out unwanted attribute in the first pass of external sorting
- Eliminate duplicates during merging
Projection: 2A
Hashing-based

Assume 8 buffers are available

Like hash-join

Step A: Partitioning phase
• Read R using one input buffer
  • For each tuple, discard unwanted attributes, apply hash function h1 to choose one of B-1 output buffers.
    – Result is B-1 partitions (of tuples with no unwanted fields)
    – Two tuples from different partitions guaranteed to be distinct
• Write each partition back to the disk
• Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

Discussion of Projection

• Sort-based approach is the standard
  – better handling of skew (many duplicates)
  – hash table may not fit in memory
  – result is sorted
  – external sorting is provided in most DBMS as a utility
• If an index on the relation contains all wanted attributes in its search key, can do index-only scan
  – Apply projection techniques to data entries (much smaller!)

Set Operations

• Intersection and cross-product special cases of join.
• Union (Distinct) and Except similar; we’ll do union
  – very similar to external sort and join algorithms
• Sorting based approach to union:
  – Sort both relations (on combination of all attributes)
  – Scan sorted relations and merge them
• Hash based approach to union:
  – Partition R and S using hash function h.
  – For each S-partition, build in-memory hash table (using h2), scan corresponding R-partition and add tuples to table while discarding duplicates

Algorithms for Set Operations

Step B: Duplicate elimination phase
For each partition
• Build an in-memory hash table
• Read it one page at a time into memory
• Hash using function h2 (different from h1) on all fields
  – For two tuples in the same bucket, check for duplicates, then discard duplicates.
• Why does h2 have to be different from h1?
  – since h1 hashes the same partition to the same value
• If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

Algorithms for Aggregate Operations
Aggregate Operations (AVG, MIN, etc.)

- **Without grouping:**
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do **index-only scan**

- **With grouping:**
  - Sort on group-by attributes
  - or, hash on group-by attributes
  - can combine sort/hash and aggregate
  - can do index-only scan here as well

Summary

- A virtue of relational DBMSs: queries are composed of a few basic operators
  - the implementation of these operators can be carefully tuned (and it is important to do this).

- Many alternative implementation techniques for each operator
  - no universally superior technique for most operators

- Must consider available alternatives for each operation in a query and choose best one based on system statistics and the overall query
  - This is part of the broader task of optimizing a query composed of several ops