CompSci 516
Data Intensive Computing Systems

Lecture 8
Query Evaluation and Join Algorithms

Instructor: Sudeepa Roy
Announcements

• **Project Proposal due next Wednesday 09/28**
  – About 1 to 3 pages
  – Template on sakai
  – Use and learn latex!
  – Intro, related work, problem definition and plan

• **Use piazza for project-related communication**
  – private email thread for each group on piazza
  – size 1 : 3 groups
  – size 2 : 4 groups
  – size 3 : 4 groups

• **Happy to discuss in person any time**
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

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Overview of Query Evaluation
Overview of Query Evaluation

• How queries are evaluated in a DBMS
  – How DBMS describes data (tables and indexes)

• Recall Relational Algebra = Logical Query Plan

• Now Algorithms will be attached to each operator = Physical Query Plan

• Plan = Tree of RA ops, with choice of algorithm for each op.
  – Each operator typically implemented using a “pull” interface
  – when an operator is “pulled” for the next output tuples, it “pulls” on its inputs and computes them
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
    • Does not use the index structure (hash or tree structure – can 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 (NTuples) and # pages (NPages) for each relation
- # distinct key values (NKeys) and NPages 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

• A tree index \textit{matches} (a conjunction of) terms that involve only attributes in a \textit{prefix} of the search key.
  • E.g., Tree index on \(<a, b, c>\) matches the selection
    • \(a = 5 \text{ AND } b = 3\),
    • and \(a = 5 \text{ AND } b > 6\),
    • but not \(b = 3\)

• A hash index \textit{matches} (a conjunction of) terms that has a term \textit{attribute} \(= \text{value}\) for every \textit{attribute} in the search key of the index.
  • E.g., Hash index on \(<a, b, c>\) matches
    • \(a = 5 \text{ AND } b = 3 \text{ AND } c = 5\);
    • but it does not match \(b = 3\),
    • or \(a = 5 \text{ AND } b = 3\),
    • or \(a > 5 \text{ AND } b = 3 \text{ AND } c = 5\)
A Note on Complex Selections

- If index (hash or tree) on
  - search key \(<bid, sid>\)
- Selection condition
  - \(rname = 'Joe' \text{ AND } bid = 5 \text{ AND } sid = 3\)
- What would you do?

- \(<bid, sid>\) can be used to retrieve all tuples with \(bid = 5\) and \(sid = 3\)
  - then apply \(rname = 'Joe'\) to each such tuple to eliminate more
A Note on Complex Selections

• Suppose two indexes
  – B+ tree index on day
  – index on search key <bid, sid>

• Selection condition
  – day<8/9/94 AND bid = 5 AND sid = 3

• What would you do?

• Two choices
• Part of the index not matched – check for each retrieved tuple
  – We only discuss case with no ORs
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 \((\text{rname} = \text{‘Joe’} \land \text{bid} = 5 \land \text{sid} = 3)\)
- \#of sailors pages = \(N\)
- \#distinct keys = \(K\)
- Fraction of pages satisfying this condition = (approximately) \(N/K\)
Selectivity : Example 2

- Hash index on sailors <bid, sid>
- Selection condition \((\text{bid} = 5 \land \text{sid} = 3)\)
- Suppose \(N_1\) distinct values of bid, \(N_2\) for sid
- Reduction factors
  - for \((\text{bid} = 5)\) : \(1 / N_1\)
  - for \((\text{bid} = 5 \land \text{sid} = 3)\) : \(1 / (N_1 \times N_2)\)
  - assumes independence
- Fraction of pages retrieved or I/O:
  - for clustered index = \(1 / (N_1 \times N_2)\)
  - for unclustered index = \(1\)
Selectivity : Example 3

- Tree index on sailors <bid>
- Selection condition \((\text{bid} > 5)\)
- Lowest value of bid = 1, highest = 100
- Reduction factor
  - \((100 - 5)/(100 - 1)\)
  - assumes uniform distribution
- In general:
  - key > value : \((\text{High} - \text{value}) / (\text{High} - \text{Low})\)
  - key < value : \((\text{value} - \text{Low}) / (\text{High} - \text{Low})\)
Operator Algorithms
Relational Operations

• We will consider how to implement:
  – Selection ($\sigma$) Selects a subset of rows from relation.
  – Projection ($\pi$) Deletes unwanted columns from relation.
  – Join ($\bowtie$) Allows us to combine two relations *(in detail)*
  – Set-difference (-) Tuples in reln. 1, but not in reln. 2.
  – Union ($\cup$) 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
Algorithms for Joins
Equality Joins With One Join Column

- **In algebra:** $R \bowtie S$
  - Common! Must be carefully optimized
  - $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient

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

```
SELECT *  
FROM Reserves R, Sailors S  
WHERE R.sid=S.sid
```
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**  
   Very similar to duplicate elimination in projection
Algorithms for Joins

1. NESTED LOOP JOINS
Simple Nested Loops Join

\[ R \bowtie S \]

- For each tuple in the outer relation \( R \), we scan the entire inner relation \( S \).
  - Cost: \( M + (p_R \times M) \times N = 1000 + 100 \times 1000 \times 500 \) 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: \( M + M \times N = 1000 + 1000 \times 500 \)

- If smaller relation (\( S \)) is outer
  - Cost: \( N + M \times N = 500 + 500 \times 1000 \)
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 $500 + 2 = 502$ 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?

![Diagram of Block Nested Loops Join](image)
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 $\langle r, s \rangle$ to result
- Then read next $R$-block, scan $S$, etc.

Join Result

Input buffer for $S$  Output buffer
Cost of Block Nested Loops

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

foreach block of B-2 pages of R do
  foreach page of S do {
    for all matching in-memory tuples r in R-block and s in S-page
      add <r, s> to result
  }

M = 1000 pages in R
p_R = 100 tuples per page
N = 500 pages in S
p_S = 80 tuples per page

Input buffer for S
Output buffer

Join Result
Cost of Block Nested Loops

- R is outer
- B-2 = 100-page blocks
- How many blocks of R? 10
- Cost to scan R? 1000
- Cost to scan S? 10 * 500
- Total Cost? 1000 + 5000 = 6000
- (check yourself)
  - If space for just 90 pages of R, we would scan S 12 times, cost = 7000

```
foreach block of B-2 pages of R do
  foreach page of S do {
    for all matching in-memory tuples r in R-block and s in S-page
      add <r, s> to result
  }
```

\[ M = 1000 \text{ pages in } R \]
\[ p_R = 100 \text{ tuples per page} \]
\[ N = 500 \text{ pages in } S \]
\[ p_S = 80 \text{ tuples per page} \]

for blocked access, it might be good to equally divide buffer pages among R and S ("seek time" less)
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: $M + (M*p_R) * \text{cost of finding matching S tuples}$
- 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 5-6

\[
\begin{align*}
M &= 1000 \text{ pages in } R \\
p_R &= 100 \text{ tuples per page} \\
N &= 500 \text{ pages in } S \\
p_S &= 80 \text{ tuples per page}
\end{align*}
\]
Cost of Index Nested Loops

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

- Hash-index (Alt. 2) on sid of Sailors (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 tuple:
    - (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 * 1000 * 2.2 = 221,000 I/Os

M = 1000 pages in R  
\( p_R = 100 \) tuples per page  
N = 500 pages in S  
\( p_S = 80 \) tuples per page
Cost of Index Nested Loops

- Hash-index (Alt. 2) on sid of Reserves (as inner), sid is NOT a key

- Cost to Scan Sailors:
  - 500 page I/Os, 80*500 tuples.

- For each Sailors tuple:
  - 1.2 I/Os to find index page with data entries
  - + cost of retrieving matching Reserves tuples
    - Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000).
    - Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered

- Total cost = 500 + 80 * 500 * 2.2 = 88,500 if clustered
  up to ~ 500 + 80 * 500 * 3.7 = 148,500 if unclustered (approx)

- even with unclustered index, index NLJ may be cheaper than simple NLJ
Algorithms for Joins

2. SORT-MERGE JOINS
Sort-Merge Join

• Sort R and S on the join column
• Then scan them to do a "merge" (on join col.)
• Output result tuples.
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>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</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>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>puppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>puppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>
Sort-Merge Join: 3/3

- Then resume scanning R and S

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rasty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

WRITE THREE OUTPUT TUPLES
Sort-Merge Join: 3/3

- ... and proceed till end

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

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

NO MATCH, CONTINUE SCANNING S
Sort-Merge Join: 3/3

- ... and proceed till end
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>uppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- **Cost:** $O(M \log M) + O(N \log N) + (M+N)$
  - cost of sorting $R$ + sorting $S$ + merging $R$, $S$
  - The cost of scanning in merge-sort, $M+N$, could be $M*N$!
    - assume the same single value of join attribute in both $R$ and $S$
    - but it is extremely unlikely
Cost of Sort-Merge Join

- **100 buffer pages**
- **Sort R:**
  - (pass 0) $1000/100 = 10$ sorted runs
  - (pass 1) merge 10 runs
  - read + write, 2 passes
  - $4 \times 1000 = 4000$ I/O
- **Similarly, Sort S:** $4 \times 500 = 2000$ I/O
- **Second merge phase of sort-merge join**
  - another $1000 + 500 = 1500$ I/O
  - assume uniform $\sim 2.5$ matches per sid, so $M+N$ is sufficient
- **Total 7500 I/O**

**Check yourself:**

- Consider #buffer pages 35, 100, 300
- Cost of sort-merge = 7500 in all three
- Cost of block nested 15000, 6000, 2500

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>
Algorithms for Joins

3. HASH JOINS
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 $h()$ value) of R and S
   - tuples in different partition of $h$ will never join
   - use a “different” hash function $h_2$ for joining these tuples
     - (why different – see next slide first)
Hash-Join

- Partition both relations using hash function $h$
- $R$ tuples in partition $i$ will only match $S$ tuples in partition $i$

- Read in a partition of $R$, hash it using $h_2$ ($\neq h$).
- Scan matching partition of $S$, search for matches.
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 in buffer—better if limited buffer
• 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., $R.sid = S.sid$ AND $R.rname = S.sname$
  – For Index Nested Loop, build index on $<sid, sname>$ (if S is inner); or use existing indexes on sid or sname
  – For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

• Inequality conditions
  – e.g., $R.rname < S.sname$
  – 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
Algorithms for Selection
Schema for Examples

Sailors (sid: integer, sname: string, rating: integer, age: real)
Reserves (sid: integer, bid: integer, day: dates, rname: string)

• Reserves:
  – Each tuple is 40 bytes long, 100 tuples per page, 1000 pages

• Sailors:
  – Each tuple is 50 bytes long, 80 tuples per page, 500 pages
Selection: 1
No Index, Unsorted Data

Naïve 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)

SELECT *
FROM    Reserves R
WHERE   R.rname = ‘Joe’

Reserves has
  • 1000 pages
  • each page holds 100 tuples
  • each tuple is 40 bytes
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)

SELECT *
FROM Reserves R
WHERE R.rname = 'Joe'

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes
1. Find qualifying data entries.
2. Sort the rid’s of the data records to be retrieved.
3. Fetch rids 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 day<8/9/94 AND bid=5 AND sid=3

- A B+ tree index on day can be used
  - then, bid=5 and sid=3 must be checked for each retrieved tuple

- A hash index on <bid, sid> could be used;
  - 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 \( \text{day}<8/9/94 \land \text{bid}=5 \land \text{sid}=3 \)
  – If we have a B+ tree index on day and an index on 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 \( \text{bid}=5 \)
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

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection: 1

Sorting-based

- 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

SELECT DISTINCT R.sid, R.bid
FROM Reserves R

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes
Projection: 2A
Hashing-based

Assume B buffers are available

Step A: Partitioning phase

• Read R using one input buffer
• For each tuple, discard unwanted attributes, apply hash function \( h_1 \) 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.

Reserves has
• 1000 pages
• each page holds 100 tuples
• each tuple is 40 bytes

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection: 2B Hashing-based

Assume B buffers are available

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.

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes

SELECT DISTINCT R.sid, R.bid
FROM Reserves R

Reserves has
- 1000 pages
- each page holds 100 tuples
- each tuple is 40 bytes
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!)

• If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better:
  – Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
Algorithms for Set Operations
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 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