CompSci 516
Database Systems

Lecture 10
Query Evaluation and Join Algorithms

Instructor: Sudeepa Roy
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)

• Relational Algebra Tree/Plan = 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 -- 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 (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
Announcements

• Midterm on 10/11 (next week Thursday)
  – everything until 10/4 included

• No class on 10/9
  – fall break

• Change in Sudeepa’s office hour time 10/4 (Thursday)
  – at 1 pm
  – or send me an email for an appointment
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,
    • and a=5 AND b>6,
    • but not 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 \((rname = 'Joe' \land bid = 5 \land sid = 3)\)
- \#of sailors pages = \(N\)
- \#distinct keys = \(K\)
- Fraction of pages satisfying this condition = (approximately) \(N/K\)
- Assumes \textit{uniform distribution}
Selectivity: Example 2

- Hash index on sailors $\langle bid, sid \rangle$
- Selection condition $(bid = 5 \land sid = 3)$
- Suppose $N_1$ distinct values of $bid$, $N_2$ for $sid$
- Reduction factors
  - for $(bid = 5)$: $1/N_1$
  - for $(bid = 5 \land sid = 3)$: $1/(N_1 \times N_2)$
- 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 (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?
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
  - Remember, 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
Algorithms for Joins

1. NESTED LOOP JOINS
Simple Nested Loops Join

\[ \text{R } \bowtie \text{ S} \]

- 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: \( N + M \times N = 500 + 500 \times 1000 \)

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

How many buffer pages do you need?
**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?
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

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
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
  }
```

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = \([\#\text{pages of outer relation}/\text{blocksize}]\)
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 7-8

$M = 1000$ 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

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
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)**

---

**SQL Code:**

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

**Nested Loops Example:**

```
foreach tuple r in R do
  foreach tuple s in S where ri == sj do
    add <r, s> to result
```

---

- 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 >= current S tuple
  - then advance scan of S until current S-tuple >= current R tuple
  - do this as long as current R tuple = current S tuple
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
Sort-Merge Join: 3/3

• Then resume scanning R and S
Sort-Merge Join: 3/3

• ... and proceed till end

<table>
<thead>
<tr>
<th>sid</th>
<th>stname</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>stname</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

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 $M*N$!
  - assume the same single value of join attribute in both $R$ and $S$
  - but it is extremely unlikely

<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>
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 ~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 16500, 6000, 2500

### Table

<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>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>

M = 1000 pages in R
\(p_R = 100\) tuples per page

N = 500 pages in S
\(p_S = 80\) tuples per page
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 $h(\cdot \cdot)$ 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
  – \(3 \times (1000 + 500)\)
  – 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
Other operator algorithms
Algorithms for Selection

• No index, unsorted data
  – Scan entire relation
  – May be expensive if not many `Joe’s

• No index, sorted data (on ‘rname’)
  – locate the first tuple, scan all matching tuples
  – first binary search, then scan depends on matches

• B+-tree index, Hash index
  – Discussed earlier
  – Cost of accessing data entries + matching data records
  – Depends on clustered/unclustered

• More complex condition like `day<8/9/94 AND bid=5 AND sid=3`
  – Either use one index, then filter
  – Or use two indexes, then take intersection, then apply third condition
  – etc.

SELECT *
FROM    Reserves R
WHERE   R.rname = 'Joe'
Algorithms for Projection

• Two parts
  – Remove fields: easy
  – Remove duplicates (if distinct is specified): expensive

• Sorting-based
  – Sort, then scan adjacent tuples to remove duplicates
  – Can eliminate unwanted attributes in the first pass of merge sort

• Hash-based
  – Exactly like hash join
  – Partition only one relation in the first pass
  – Remove duplicates in the second pass

• Sort vs Hash
  – Sorting handles skew better, returns results sorted
  – Hash table may not fit in memory – sorting is more standard

• Index-only scan may work too
  – If all required attributes are part of index
Algorithms for Set Operations

- Intersection, cross product are special cases of joins
- Union, Except
  - Sort-based
  - Hash-based
  - Very similar to joins and projection
Algorithms for Aggregate Operations

• SUM, AVG, MIN etc.
  – again similar to previous approaches

• 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