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

• 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

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
  • 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
  - A tuples (N_Tuples) and B pages (N_Pages) for each relation
  - A distinct key values (NKey) and BPages 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,
  - 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' ∧ 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 \((\text{bid} = 5 \land \text{sid} = 3)\)
- 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 = \(1/N_1\)
  - 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:
  - Join \((\bowtie)\) Allows us to combine two relations (in detail)
- Also
  - Selection \((\sigma)\) Selects a subset of rows from relation.
  - Projection \((\Pi)\) Deletes unwanted columns from relation.
  - Set-difference \((\setminus)\) Tuples in reln. 1, but not in reln. 2.
  - Union \((\cup)\) Tuples in reln. 1 and in reln. 2.
  - Aggregation \((\text{SUM, MIN, etc.) and GROUP BY}\)

- 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

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

• 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

Common Join Algorithms

1. Nested Loops Joins (NLI) 
   — 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**

\( R \bowtie S \)

foreach tuple \( r \) in \( R \) do
  foreach tuple \( s \) in \( S \) where \( r_i == s_j \) do
    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

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

• If smaller relation \( S \) is outer 
  Cost: \( N + M \times N = 500 + 500 \times 1000 \)

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.
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: \( M \times (M + p_S) \times \text{cost of finding matching } S \text{ tuples} \)
- 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

- Hash-index (Alt. 2) on sid of Sailors (as inner), sid is a key
- Cost to scan Reserve:
  - 1000 page I/Os, 100*1000 tuples.
- Cost to find matching Sailors tuples?
  - For each Reserve 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 Block Nested Loops

- 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: \( M \times (M + p_S) \times \text{cost of finding matching } S \text{ tuples} \)
- 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 Block Nested Loops

- 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: \( M \times (M + p_S) \times \text{cost of finding matching } S \text{ tuples} \)
- 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

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 \( > \) 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 \( R \), (current \( R \) group) and all S tuples with same value in \( S \), (current \( S \) group)
  - match
  - find all the equal tuples
  - output \( \langle r, s \rangle \) 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

... and proceed till end
**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>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>puppy</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>

- 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

**Cost of Sort-Merge Join**

<table>
<thead>
<tr>
<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>
</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>puppy</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>

- Cost of sorting \(R\) + sorting \(S\) + merging \(R, S\)
  - Ignoring \(B\) (as the base of \(\log\))

**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 \(h(\ldots)\) 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)

**Algorithms for Joins**

3. **HASH JOINS**

**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 \(h_2(\# h)\)
- 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