Principles of Database Management Systems

5: Query Processing

Pekka Kilpeläinen
(partially based on Stanford CS245 slide originals by Hector Garcia-Molina, Jeff Ullman and Jennifer Widom)

Query Processing

• How does the query processor execute queries?
  Query \rightarrow \text{Query Plan}
  - SQL - expression of methods to realize the query

Focus: Relational System

Example

SELECT B, D
FROM R, S
WHERE R.C = S.C AND R.A = "c" AND S.E = 2

Answer

R A B C S C D E
a 1 10 10 x 2
b 1 20 20 y 2
c 2 10 30 z 2
d 2 35 40 x 1
e 3 45 50 y 3

RxS

• How to execute query?
  - Do Cartesian product RxS
  - Select tuples
  - Do projection

Basic idea

Bingo!
Got one...
Problem

• A Cartesian product RxS may be LARGE
  - need to create and examine n x m tuples, where n = |R| and m = |S|
  - For example, n = m = 1000 => 10^6 records
-- need more efficient evaluation methods

Overview of Query Execution

- SQL query
- parse tree
- logical query plan
- apply laws
- "improved" l.q.p
- estimate result sizes
- l.q.p. + sizes
- statistics
- estimate costs
- pick best
- execute

{P1,P2,...}

Relational Algebra - used to describe logical plans

Ex: Original logical query plan

\[
\text{SELECT B,D} \rightarrow \Pi_{B,D} \\
\text{WHERE} \rightarrow \sigma_{R.A = "c" \land S.E=2 \land R.C=S.C} \\
\text{FROM} \rightarrow R \bowtie S
\]

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

Improved logical query plan:

\[
\begin{align*}
\text{Plan II} & \rightarrow \Pi_{B,D} \\
\sigma_{R.A = "c"} & \leftarrow R \\
\sigma_{S.E=2} & \leftarrow S \\
R \bowtie S & \rightarrow \text{natural join}
\end{align*}
\]

Physical Query Plan:

Detailed description to execute the query:

- algorithms to implement operations; order of execution steps; how relations are accessed; For example:
  1. Use R.A index to select tuples of R with R.A = "c"
  2. For each R.C value found, use the index on S.C to find matching tuples
  3. Eliminate S tuples with S.E ≠ 2
  4. Join matching R.S tuples, project on attributes B and D, and place in result
### Outline (Chapter 6)
- (Relational algebra for queries
  - representation for logical query plans
  - operations that we need to support)
- Algorithms to implement relational operations
  - efficiency estimates
    (for selecting the most appropriate)
- We concentrate on algorithms for **selections** and **joins**

### Physical operators
- Principal methods for executing
  operations of relational algebra
- Building blocks of physical query plans
- Major strategies
  - scanning tables
  - sorting, indexing, hashing

### Cost Estimates
- Estimate only # of disk I/O operations
  - dominating efficiency factor
    • exception: communication of data over network
- Simplifying assumptions
  - **ignore** the cost of **writing the result**
    • result blocks often passed in memory to further
      operations ("pipelining")
  - I/O happens **one block at a time**
    (e.g., ignore usage of cylinder sized blocks)

### Parameters for Estimation
- **M**: # of available main memory buffers
  (estimate)
- Kept as statistics for each relation R:
  - **T(R)**: # of tuples in R
  - **B(R)**: # of blocks to hold all tuples of R
  - **V(R, A)**: # of distinct values for attribute R.A
  = **SELECT COUNT (DISTINCT A) FROM R**

### Cost of Scanning a Relation
- Normally assume relation R to be **clustered**, that is, stored in blocks exclusively (or predominantly) used for representing R
- For example, consider a clustered-file organization of relations
  `DEPT(Name, ...)` and
  `EMP(Name, Dname, ...)`

---

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</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>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d</td>
<td>2</td>
</tr>
<tr>
<td>2c</td>
<td>y 2</td>
</tr>
<tr>
<td>3z</td>
<td>2</td>
</tr>
<tr>
<td>4x</td>
<td>1</td>
</tr>
<tr>
<td>5y</td>
<td>3</td>
</tr>
</tbody>
</table>
• Relation EMP might be considered clustered, relation DEPT probably not
• For a clustered relation R, sufficient to read (approx.) B(R) blocks for a full scan
• If relation R not clustered, most tuples probably in different blocks => input cost approx. T(R)

Classification of Physical Operators (1)
• By method:
  - sort-based
    • process relation(s) in the sorted order
  - hash-based
    • process relation(s) partitioned in hash buckets
  - index-based
    • apply existing indexes
    • especially useful for selections

Classification of Physical Operators (2)
• By applicability and cost:
  - one-pass methods
    • if at least one argument relation fits in main memory
  - two-pass methods
    • if memory not sufficient for one-pass
    • process relations twice, storing intermediate results on disk
  - multi-pass
    • generalization of two-pass for HUGE relations

Implementing Selection
• How to evaluate \( \sigma_C(R) \)?
  - Sufficient to examine one tuple at a time
  - Easy to evaluate in one pass:
    • Read each block of R using one input buffer
    • Output records that satisfy condition C
  - If R clustered, cost = B(R); else T(R)
• Projection \( \pi_A(R) \) in a similar manner

Index-Based Selection
• Consider selection \( \sigma_{A='c'}(R) \)
• If there is an index on R.A, we can locate tuples t with t.A='c' directly
• What is the cost?
  - How many tuples are selected?
    • estimate: \( T(R)/V(R,A) \) on the average
    • if A is a primary key, \( V(R,A) = T(R) \) => 1 disk I/O

Index-Based Selection (cont.)
• Index is clustering if tuples with A='c' are stored in consecutive blocks (for any 'c')
Selection using a clustering index

- We estimate a fraction $T(R)/V(R,A)$ of all $R$ tuples to satisfy $A=c$. Apply same estimate to data blocks accessible through a clustering index $\Rightarrow \frac{B(R)}{V(R,A)}$ is an estimate for the number of block accesses

- Further simplifications: Ignore, e.g.,
  - cost of reading the (few) index blocks
  - unfilled room left intentionally in blocks
  - ...

Selection Example

Consider $\sigma_{A=0}(R)$ when $T(R)=20,000$, $B(R)=1000$, and there's an index on $R.A$

- simple scan of $R$
  - if $R$ not clustered: cost = $T(R) = 20,000$
  - if $R$ clustered: cost = $B(R) = 1000$
- if $V(R,A)=100$ and index is ...
  - not clustering $\Rightarrow$ cost = $T(R)/V(R,A) = 200$
  - clustering $\Rightarrow$ cost = $B(R)/V(R,A)= 10$
- if $V(R,A)=20,000$ (i.e., A is key) $\Rightarrow$ cost = 1

Time
- disk I/O
  - 5 ms
  - 5 min
  - 15 sec
  - 3 sec
  - 0.15 sec
  - 15 ms

Processing of Joins

- Consider natural join $R(X,Y) \bowtie S(Y,Z)$
  - general joins rather similarly, possibly with additional selections (for complex join conditions)

- Assumptions:
  - $Y =$ join attributes common to $R$ and $S$
  - $S$ is the smaller of relations: $B(S) = B(R)$

One-Pass Join

- Requirement: $B(S) < M$, i.e., $S$ fits in memory
- Read entire $S$ in memory (using one buffer);
  Build a dictionary (balanced tree, hash table) using join attributes of tuples as search key
- Read each block of $R$ (using one buffer);
  For each tuple $t$, find matching tuples from the dictionary, and output their join
- I/O cost: $B(S) + B(R)$

What If Memory Insufficient?

- Basic join strategy:
  - "nested-loop" join
  - "1+n pass" operation:
    - one relation read once, the other repeatedly
  - no memory limitations
  - can be used for relations of any size

- Nested-loop (conceptually)

  for each tuple $s \in S$ do
  for each tuple $r \in R$ do
    if $r.Y = s.Y$ then
      output join of $r$ and $s$;
  
- Cost (like for Cartesian product):
  $T(S) \ast (1 + T(R)) = T(S) + T(S)T(R)$
• If R and S clustered, can apply block-based nested-loop join:

```plaintext
for each chunk of M-1 blocks of S do
  Read blocks in memory;
  Insert tuples in a dictionary using the join attributes;
  for each block b of R do
    Read b in memory;
    for each tuple r in b do
      Find matching tuples from the dictionary;
      output their join with r;
```

Cost of Block-Based Nested-Loop Join

• Consider R(X,Y) S(Y,Z) when B(R)=1000, B(S)=500, and M = 101
  - Use 100 buffers for loading S
  -> 500/100 = 5 chunks
  - Total I/O cost = 5 x (100 + 1000) = 5500 blocks
• R as the outer-loop relation -> I/O cost 6000
  - in general, using the smaller relation in the outer loop gives an advantage of B(R) - B(S) operations

Analysis of Nested-Loop join

• B(S)/(M-1) outer-loop iterations;
  Each reads M-1 + B(R) blocks
  -> total cost = B(S) + B(S)B(R)/(M-1), or approx. B(S)B(R)/M blocks
• Not the best method, but sometimes the only choice
• Next: More efficient join algorithms

Sort-Based Two-Pass Join

• Idea: Joining relations R and S on attribute Y is rather easy, if the relations are sorted using Y
  - IF not too many tuples join for any value of the join attributes. (E.g. if π_Y(R) = π_Y(S) = {y}, all tuples match, and we may need to resort to nested-loop join)
  - If relations not sorted already, they have to be sorted (with two-phase multi-way merge sort, since they do not fit in memory)

Sort-Based Two-Pass Join

1. Sort R with join attributes Y as the sort key;
2. Do the same for relation S;
3. Merge the sorted relations, using 1 buffer for current input block of each relation:
   - skip tuples whose Y-value y not in both R and S
   - read blocks of both R and S for all tuples whose Y value is y
   - output all possible joins of the matching tuples r ∈ R and s ∈ S

Example: Join of R and S sorted on Y
Analysis of Sort-Based Two-Phase Join

- Consider \( R(X,Y) \bowtie_S (Y,Z) \) when \( B(R)=1000, \ B(S)=500, \) and \( M = 101 \)
  - Remember two-phase multiway merge sort:
    - each block read + written + read + written once
    - \( 4 \times (B(R) + B(S)) = 6000 \) disk I/Os
  - Merge of sorted relations for the join:
    - \( B(R) + B(S) = 1500 \) disk I/Os
  - Total I/O cost = \( 4 \times (B(R) + B(S)) = 7500 \)
- Seems big, but for large \( R \) and \( S \) much better than \( B(R)B(S)/M \) of block-based nested loop join

Two-Phase Join with Hashing

- Idea: If relations do not fit in memory, first hash the tuples of each relation in buckets. Then join tuples in each pair of buckets.
- For a join on attributes \( Y \), use \( Y \) as the hash key
  - **Hash Phase**: For each relation \( R \) and \( S \):
    - Use 1 input buffer, and \( M-1 \) output buffers as hash buckets
    - Read each block and hash its tuples; When output buffer gets full, write it on disk as the next block of that bucket

Hash-Join: The Join Phase

- For each \( i = 1, ..., M-1 \), perform one-pass join between buckets \( R_i \) and \( S_i \)
  - the smaller one has to fit in \( M-1 \) main memory buffers
- Average size for bucket \( R_i \) is approx. \( B(R)/M \), and \( B(S)/M \) for bucket \( S_i \)
- \( \text{Approximated memory requirement} \quad \min(B(R), B(S)) < M^2 \)

Cost of Hash-Join

- Consider \( R(X,Y) \bowtie_S (Y,Z) \) when \( B(R)=1000, \ B(S)=500, \) and \( M = 101 \)
- Hashing \( \rightarrow \) 100 buckets for both \( R \) and \( S \), with avg sizes 1000/100=10 and 500/100=5
- I/O cost 4500 blocks:
  - hashing phase 2x1000 + 2x500 = 3000 blocks
  - join phase: 1000 + 500 (in total for the 100 one-pass joins)
- In general: \( \text{cost} = 3(B(R) + B(S)) \)
Index-Based Join

- Still consider \( R(X,Y) \bowtie S(Y,Z) \)
- Assume there’s an index on \( S.Y \)
- Can compute the join by
  - reading each tuple of \( R \)
  - locating matching tuples of \( S \) by index-lookup for \( t.Y \), and
  - outputting their join with tuple \( t \)
- Efficiency depends on many factors

Cost of Index-Based Join

- Cost of scanning \( R \):
  - \( B(R) \), if clustered; \( T(R) \), if not
- On the average, \( T(S)/V(S,Y) \) matching tuples found by index lookup; Cost of loading them (total for all tuples of \( R \)):
  - \( T(R)T(S)/V(S,Y) \), if index not clustered
  - \( T(R)B(S)/V(S,Y) \), if index clustered
- Cost of loading tuples of \( S \) dominates

Example: Cost of Index-Join

- Again \( R(X,Y) \bowtie S(Y,Z) \) with \( B(R)=1000, B(S)=500; T(R) = 10,000, T(S) = 5000, and V(S,Y) = 100 \)
- Assume \( R \) clustered, and the index on \( S.Y \) is clustering
  \( \Rightarrow \) I/O cost \( 1000 + 10,000 \times 500/100 = 51,000 \) blocks
- Often not this bad…

Index-Join useful …

… when \( |R| \ll |S| \), and \( V(S,Y) \) large (i.e., the index on \( S.Y \) is selective)

- For example, if \( Y \) primary key of \( S \):
  - each of the \( T(R) \) index lookups locates at most one record of relation \( S \)
  \( \Rightarrow \) at most \( T(R) \) input operations to load blocks of \( S \)
  \( \Rightarrow \) Total cost only
    - \( B(R) + T(R) \), if \( R \) clustered, and
    - \( T(R) + T(R) = 2T(R) \), if \( R \) not clustered

Joins Using a Sorted Index  [Book: 6.7.4]

- Still consider \( R(X,Y) \bowtie S(Y,Z) \)
- Assume there’s a sorted index on both \( R.Y \) and \( S.Y \)
  - B-tree or a sorted sequential index
- Scan both indexes in the increasing order of \( Y \)
  - like merge-join, without need to sort first
  - if index dense, can skip nonmatching tuples without loading them
  - very efficient
- Details to exercises?

Summary - Query Processing

- Overall picture:
  - Query -> logical plan -> physical plan -> execution
- Physical operators
  - to implement physical query plans
    - selections, joins
    - one-pass, two-pass, (multi-pass)
    - based on scanning, sorting, hashing, existing indexes
- Next: More about query compilation (Chapter 7)