CSE 562
Database Systems

Query Processing:
Physical Operators

Algorithms for Algebra Operators

- Three primary techniques
  - Sorting
  - Hashing
  - Indexing

- Three degrees of difficulty
  - data small enough to fit in memory
  - too large to fit in main memory but small enough to be handled by a “two-pass” algorithm
  - so large that “two-pass” methods have to be generalized to “multi-pass” methods (quite unlikely nowadays)

Outline – Query Optimization

- Overview
- Relational algebra level
  - Algebraic Transformations
- Detailed query plan level
  - Estimate Costs
    - Estimating size of results
    - Estimating # of IOs
  - Generate and compare plans

Estimating IOs

- Count # of disk blocks that must be read (or written) to execute query plan
**Additional Cost Estimation Parameters**

- \( B(R) = \) # of blocks containing \( R \) tuples
- \( f(R) = \) max # of tuples of \( R \) per block
- \( M = \) # memory blocks available
- \( HT(i) = \) # levels in index \( i \)
- \( LB(i) = \) # of leaf blocks in index \( i \)

**Clustering Index**

- Index that allows tuples to be read in an order that corresponds to physical order

**Clustering Can Radically Change Cost**

- Clustered file organization
  
  \[
  \begin{array}{cccc}
  R1 & R2 & S1 & S2 \\
  R3 & R4 & S3 & S4 \\
  \vdots & \vdots & \vdots & \vdots
  \end{array}
  \]

- Clustered relation
  
  \[
  \begin{array}{cccc}
  R1 & R2 & R3 & R4 \\
  R5 & R6 & R7 & R8 \\
  \vdots & \vdots & \vdots & \vdots
  \end{array}
  \]

- Clustering index

**Example**

- \( R1 \bowtie R2 \) over common attribute \( C \)

  \[
  \begin{array}{llllllllllll}
  T(R1) = 10,000 & T(R2) = 5,000 & S(R1) = S(R2) = 1/10 block & Memory available = 101 blocks
  \end{array}
  \]

  → Metric: # of IOs (ignoring writing of result)
Caution!

This may not be the best way to compare
• ignoring CPU costs
• ignoring timing
• ignoring double buffering requirements

Options

• Transformations: R1 $\bowtie$ R2, R2 $\bowtie$ R1
• Join algorithms:
  – Iteration (nested loops)
  – Merge join
  – Join with index
  – Hash join

Example

• **Iteration Join** (conceptually – without taking into account disk block issues)
  
  for each $r \in R_1$ do
  
  for each $s \in R_2$ do
  
  if $r.C = s.C$ then output $r,s$ pair

Example

• **Merge Join** (conceptually)
  
  (1) if $R_1$ and $R_2$ not sorted, sort them
  
  (2) $i \leftarrow 1; j \leftarrow 1$
  
  While $(i \leq T(R_1)) \land (j \leq T(R_2))$ do
  
  if $R_1{[i].C} = R_2{[j].C}$ then outputTuples
  
  else if $R_1{[i].C} > R_2{[j].C}$ then $j \leftarrow j+1$
  
  else if $R_1{[i].C} < R_2{[j].C}$ then $i \leftarrow i+1$
Example

Procedure \textit{outputTuples}

\[\text{While } (R1\{i\}.C = R2\{j\}.C) \land (i \leq T(R1)) \text{ do}\]
\[\text{ [ } jj \leftarrow j; \]
\[\text{ while } (R1\{i\}.C = R2\{jj\}.C) \land (jj \leq T(R2)) \text{ do}\]
\[\text{ [ output pair } R1\{i\}, R2\{jj\}; \]
\[jj \leftarrow jj+1 \]
\[i \leftarrow i+1 \]

Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{i}.C</th>
<th>R2{j}.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>52</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Example

• \textbf{Join with Index} (Conceptually)

For each \( r \in R1 \) do

[ \( X \leftarrow \text{index} (R2, C, r.C) \)

for each \( s \in X \) do

output \( r,s \) pair ]

\textbf{Note: } \( X \leftarrow \text{index} \) \text{rel, attr, value} \]
\then \( X = \text{set of rel tuples with attr = value} \)

Example

• \textbf{Hash Join} (Conceptual)
  - Hash function \( h \), range \( 0 \rightarrow k \)
  - Buckets for \( R1 \): \( G0, G1, ... Gk \)
  - Buckets for \( R2 \): \( H0, H1, ... Hk \)

\textbf{Algorithm}

1) Hash \( R1 \) tuples into \( G \) buckets
2) Hash \( R2 \) tuples into \( H \) buckets
3) For \( i = 0 \) to \( k \) do

  match tuples in \( Gi, Hi \) buckets
Simple Example

hash: even/odd

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th></th>
<th>R2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>13</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Buckets

Even: 2 4 8 4 12 8 14
Odd: 3 5 9 5 3 13 11

Factors that Affect Performance

(1) Tuples of relation stored physically together?
(2) Relations sorted by join attribute?
(3) Indexes exist?

Disk-Oriented Computation Model

- There are $M$ main memory buffers
  - Each buffer has the size of a disk block
- The input relation is read one block at a time
- The cost is the number of blocks read
- The output buffers are not part of the $M$ buffers mentioned above
- Pipelining allows the output buffers of an operator to be the input of the next one
- We do not count the cost of writing the output

Notation

- $B(R) =$ number of blocks that $R$ occupies
- $T(R) =$ number of tuples of $R$
- $V(R, [a_1, a_2, ..., a_n]) =$ number of distinct tuples in the projection of $R$ on $a_1, a_2, ..., a_n$
One-Pass Main Memory Algorithms for Unary Operators

- Assumption: Enough memory to keep the relation
- Projection and selection:
  - Scan the input relation $R$ and apply operator one tuple at a time
  - Cost depends on
    - clustering of $R$
    - whether the blocks are consecutive
- Duplicate elimination and aggregation
  - create one entry for each group and compute the aggregated value of the group
  - it becomes hard to assume that CPU cost is negligible
    - main memory data structures are needed

One-Pass Nested Loop Join

- Assume $B(R)$ is less than $M$
- Tuples of $R$ should be stored in an efficient lookup structure
- Exercise: Find the cost of the algorithm below

```plaintext
for each block $Br$ of $R$
do
store tuples of $Br$ in main memory
for each block $Bs$ of $S$
do
for each tuple $s$ of $Bs$
do
join tuples of $s$ with matching tuples of $R$
```

Generalization of Nested-Loops

```plaintext
for each chunk of $M-1$ blocks $Br$ of $R$
do
store tuples of $Br$ in main memory
for each block $Bs$ of $S$
do
for each tuple $s$ of $Bs$
do
join tuples of $s$ with matching tuples of $R$
```

Exercise: Compute the cost of the above algorithm

Simple Sort-Merge Join

- Assume natural join on $C$
- Sort $R$ on $C$ using the two-phase multiway merge sort
  - if not already sorted
- Sort $S$ on $C$
- Merge (opposite side)
  - assume two pointers $Pr$, $Ps$ to tuples on disk, initially pointing at the start
  - sets $R'$, $S'$ in memory
- Remarks:
  - Very low average memory requirement during merging (but no guarantee on how much is needed)
  - Cost:

```plaintext
while $Pr'$=EOF and $Ps'$=EOF
if $*Pr'[C] == *Ps'[C]$
do_cart_prod($Pr$, $Ps$)
else if $*Pr'[C] > *Ps'[C]$
$Ps++$
else if $*Ps'[C] > *Pr'[C]$
$Pr++$
function do_cart_prod($Pr$, $Ps$)
val = $*Pr'[C]$
while $*Pr'[C] == val$
do
store tuple $*Pr$ in set $R'$
while $*Ps'[C] == val$
do
store tuple $*Ps$ in set $S'$
output $R' \times S'$ // product
```
Efficient Sort-Merge Join

- **Idea:** Save two disk I/O’s per block by combining the second pass of sorting with the “merge”

  - **Step 1:** Create sorted sublists of size \( M \) for \( R \) and \( S \)
  - **Step 2:** Bring the first block of each sublist to a buffer
    - assume no more than \( M \) sublists in all
  - **Step 3:** Repeatedly find the least \( C \) value \( c \) among the first tuples of each sublist. Identify all tuples with join value \( c \) and join them.
    - When a buffer has no more tuple that has not already been considered load another block into this buffer

Efficient Sort-Merge Join Example

- Assume that after first phase of multiway sort we get 4 sublists, 2 for \( R \) and 2 for \( S \)
- Also assume that each block contains two tuples

Two-Pass Hash-Based Algorithms

- **General Idea:** Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value
  - If there are \( M \) buffers pick \( M-1 \) as the number of hash buckets
- **Example:** Duplicate Elimination
  - Phase 1: Hash each tuple of each input block into one of the \( M-1 \) bucket/buffers. When a buffer fills, save to disk
  - Phase 2: For each bucket:
    - load the bucket in main memory
    - treat the bucket as a small relation and eliminate duplicates
    - save the bucket back to disk
- **Catch:** Each bucket has to be less than \( M \)
- **Cost:**

Hash-Join Algorithms

- Assuming natural join, use a hash function that
  - is the same for both input arguments \( R \) and \( S \)
  - uses only the join attributes
- **Phase 1:** Hash each tuple of \( R \) into one of the \( M-1 \) buckets \( R_i \) and similar each tuple of \( S \) into one of \( S_i \)
- **Phase 2:** For \( i = 1..M-1 \)
  - load \( R_i \) and \( S_i \) in memory
  - join them and save result to disk
- **Question:** What is the maximum size of buckets?
- **Question:** Does hashing maintain sorting?
Index-Based Join: Simplest Version

- Assume that we do natural join of R(A,B) and S(B,C) and there is an index on S
  
  for each Br in R do
  
  for each tuple r of Br with B value b
  
  use index of S to find tuples \{s_1, s_2, \ldots, s_n\} of S with B=b
  
  output \{rs_1, rs_2, \ldots, rs_n\}

- Cost: Assuming R is clustered and non-sorted and the index on S is clustered on B then
  
  \[B(R) + \frac{T(R)B(S)}{V(S,B)} + \text{some for reading index}\]

Example 1(a)

**Iteration Join R1 \Join R2**

- Relations not contiguous
- Recall \[
  \begin{align*}
  T(R1) &= 10,000 \\
  T(R2) &= 5,000 \\
  S(R1) &= S(R2) = 1/10 \text{ block} \\
  \text{MEM} &= 101 \text{ blocks}
  \end{align*}
\]

Cost: for each R1 tuple:

\[\text{[Read tuple + Read R2]}\]

Total =10,000 \[1+5000\]=50,010,000 IOs

Opportunities in Joins Using Sorted Indexes

- Do a conventional Sort-Join avoiding the sorting of one or both of the input operands

Can we do better?

- Use our memory
- (1) Read 100 blocks of R1
- (2) Read all of R2 (using 1 block) + join
- (3) Repeat until done
Cost

- for each R1 chunk:
  - Read chunk: 1000 IOs
  - Read R2: 5000 IOs
  - 6000

  Total = 10,000 x 6000 = 60,000 IOs

  1,000

Can we do better?

- Reverse Join Order: R2 \(\bowtie\) R1

  Total = 5000 x (1000 + 10,000) =

  \[
  \frac{5 \times 11,000}{1000} = 55,000 \text{ IOs}
  \]

Example 1(b)

Iteration Join \(R2 \bowtie R1\)

- Relations contiguous

Cost
  For each R2 chunk:
  - Read chunk: 100 IOs
  - Read R1: 1000 IOs

  Total = 5 chunks x 1,100 = 5,500 IOs

Example 1(c)

Merge Join

- Both R1, R2 ordered by C; relations contiguous

Memory

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Total cost: Read R1 cost + read R2 cost

= 1000 + 500 = 1,500 IOs
Example 1(d)

Merge Join
- R1, R2 not ordered, but contiguous

→ Need to sort R1, R2 first... HOW?

One Way to Sort

Merge Sort
(i) For each 100 block chunk of R:
- Read chunk
- Sort in memory
- Write to disk

(ii) Read all chunks + merge + write out

Sorted file
Memory

Sorted Chunks

Cost: Sort
Each tuple is read, written, read, written

so...
Sort cost R1: 4 x 1,000 = 4,000
Sort cost R2: 4 x 500  = 2,000
Example 1(d) (Cont.)

**Merge Join**
- R1, R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500 = 7,500 IOs

**But:** Iteration cost = 5,500
so merge joint does not pay off!

Example 1(d) (Cont.)

But say

- R1 = 10,000 blocks contiguous
- R2 = 5,000 blocks not ordered

Iterate:

\[
\frac{5000 \times (100+10,000)}{100} = 50 \times 10,100 = 505,000 \text{ IOs}
\]

Merge join:

\[
5(10,000+5,000) = 75,000 \text{ IOs}
\]

Merge Join (with sort) WINS!

Merge Sort

**How much memory do we need for merge sort?**

E.g: Say I have 10 memory blocks

[R1]

\[
\text{100 chunks} \Rightarrow \text{to merge, need 100 blocks!}
\]

In General

Say k blocks in memory
x blocks for relation sort

# chunks = \((x/k)\)
size of chunk = k

# chunks \(\leq\) buffers available for merge

so...

\[
\frac{x}{k} \leq k
\]

or \(k^2 \geq x\) or \(k \geq \sqrt{x}\)
In Our Example

R1 is 1000 blocks, \( k = 31.62 \)
R2 is 500 blocks, \( k = 22.36 \)

Need at least 32 buffers

Can we improve on merge join?

Hint: do we really need the fully sorted files?

Cost of Improved Merge Join

\[
C = \text{Read R1} + \text{write R1 into runs} + \text{read R2} + \text{write R2 into runs} + \text{join} \\
= 2000 + 1000 + 1500 = 4500
\]

→ Memory requirement?

Example 1(e)

Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered

- Assume R1.C index fits in memory
Example 1(e) (Cont.)

Cost: Reads: 500 IOs
for each R2 tuple:
- probe index - free
- if match, read R1 tuple: 1 IO

What is expected # of matching tuples?
(a) say R1.C is key, R2.C is foreign key
    then expect = 1
(b) say V(R1,C) = 5000, T(R1) = 10,000
    with uniform assumption
    expect = 10,000/5,000 = 2

Total Cost with Index Join

(a) Total cost = 500+5000(1)1 = 5,500
(b) Total cost = 500+5000(2)1 = 10,500

What if index does not fit in memory?

Example: say R1.C index is 201 blocks
- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is
  \[ E = \frac{(0)99 + (1)101}{200} = 0.5 \]
Total Cost (including probes)

\[= 500 + 5000 \text{ [Probe + get records]}\]
\[= 500 + 5000 \{0.5 + 2\} \text{ uniform assumption}\]
\[= 500 + 12,500 = 13,000 \text{ (case b)}\]

So Far

<table>
<thead>
<tr>
<th>contiguous</th>
<th>not contiguous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate R2 (\bowtie) R1</td>
<td>55,000 (best)</td>
</tr>
<tr>
<td>Merge Join</td>
<td>_______</td>
</tr>
<tr>
<td>Sort+ Merge Join</td>
<td>_______</td>
</tr>
<tr>
<td>R1.C Index</td>
<td>_______</td>
</tr>
<tr>
<td>R2.C Index</td>
<td>_______</td>
</tr>
</tbody>
</table>

Example 1(f)

Hash Join
- R1, R2 contiguous (un-ordered)
- Use 100 buckets
- Read R1, hash, + write buckets

Example 1(f) (Cond.)

- Same for R2
- Read one R1 bucket; build memory hash table
- Read corresponding R2 bucket + hash probe

\[\rightarrow \text{ Then repeat for all buckets}\]
Cost

- “Bucketize:” Read R1 + write
- Read R2 + write
- Join: Read R1, R2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum Memory Requirements

- Size of R1 bucket = (x/k)
  k = number of memory buffers
  x = number of R1 blocks

- So... (x/k) < k

- k > \(\sqrt{x}\)  
  need: k+1 total memory buffers

Trick

Keep Some Buckets in Memory
E.g., k’=33  
R1 buckets = 31 blocks  
keep 2 in memory

called hybrid hash-join

Next

Bucketize R2
- R2 buckets = 500/33 = 16 blocks
- Two of the R2 buckets joined immediately with G0,G1
Finally

Join remaining buckets
• for each bucket pair:
  – read one of the buckets into memory
  – join with second bucket

Cost
• Bucketize R1 = 1000 + 31 \times 31 = 1961
• To bucketize R2, only write 31 buckets:
  so, cost = 500 + 31 \times 16 = 996
• To compare join (2 buckets already done)
  read 31 \times 31 + 31 \times 16 = 1457

Total cost = 1961 + 996 + 1457 = 4414

How Many Buckets in Memory?

Another Hash Join Trick
• Only write into buckets
  <val,ptr> pairs
• When we get a match in join phase,
  must fetch tuples

→ See textbook for answer...
Another Hash Join Trick (Cont.)

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R2 in memory
  5000 tuples → 5000/100 = 50 blocks
- Read R1 and match
- Read ~ 100 R2 tuples

  Total cost =  
  \[
  \begin{array}{c|c}
  \text{Read R2:} & 500 \\
  \text{Read R1:} & 1000 \\
  \text{Get tuples:} & 100 \\
  \end{array}
  \]
  \[= 1600 \]

So Far

- Iterate 5500
- Merge join 1500
- Sort+merge joint 7500
- R1.C index 5500 → 550
- R2.C index _____
- Build R.C index _____
- Build S.C index _____
- Hash join 4500+
  with trick, R1 first 4414
  with trick, R2 first _____
- Hash join, pointers 1600

Summary

- Iteration ok for “small” relations (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, hash join usually best
- Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use merge join
- If index exists, it could be useful (depends on expected result size)