Why Learn About Op Algs?

• Implemented in commercial DBMSs
  – DBMSs implement different subsets of known algorithms

• Good algorithms can greatly improve performance

• Need to know about physical operators to understand query optimization
Cost Parameters

• In database systems the data is on disk
• **Cost = total number of I/Os**
• Parameters:
  – \( B(R) \) = # of blocks (i.e., pages) for relation \( R \)
  – \( T(R) \) = # of tuples in relation \( R \)
  – \( V(R, a) \) = # of distinct values of attribute \( a \)
    • When \( a \) is a key, \( V(R,a) = T(R) \)
    • When \( a \) is not a key, \( V(R,a) \) can be anything \( \leq T(R) \)
• Main constraint: **\( M = \# \text{ of memory (buffer) pages} \)**
Cost

• Cost of an operation = number of disk I/Os to:
  – Read the operands
  – Compute the result

• Cost of writing the result to disk is *not included*
  – Need to count it separately when applicable
Outline for Today

• **Join operator algorithms**
  – One-pass algorithms (Sec. 15.2 and 15.3)
  – Index-based algorithms (Sec 15.6)
  – Two-pass algorithms (Sec 15.4 and 15.5)

  – **Note about readings:**
    • In class, we will discuss only algorithms for join operator (because other operators are easier)
    • Read the book to get more details about these algs
    • Read the book to learn about algs for other operators
Basic Join Algorithms

• Logical operator:
  – Product(pname, cname) ∞ Company(cname, city)

• Propose three physical operators for the join, assuming the tables are in main memory:
  ________________
  __________________
  __________________
Hash Join

Hash join: $R \bowtie S$

• Scan $R$, build buckets in main memory
• Then scan $S$ and join
• Cost: $B(R) + B(S)$

• One-pass algorithm when $B(R) \leq M$
  – By “one pass”, we mean that the operator reads its operands only once. It does not write intermediate results back to disk.
Hash Join Example

Patient(pid, name, address)
Insurance(pid, provider, policy_nb)

Patient \Join Insurance

Patient

<table>
<thead>
<tr>
<th>1</th>
<th>'Bob'</th>
<th>'Seattle'</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>'Ela'</td>
<td>'Everett'</td>
</tr>
<tr>
<td>3</td>
<td>'Jill'</td>
<td>'Kent'</td>
</tr>
<tr>
<td>4</td>
<td>'Joe'</td>
<td>'Seattle'</td>
</tr>
</tbody>
</table>

Insurance

<table>
<thead>
<tr>
<th>2</th>
<th>'Blue'</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>'Prem'</td>
<td>432</td>
</tr>
<tr>
<td>4</td>
<td>'Prem'</td>
<td>343</td>
</tr>
<tr>
<td>3</td>
<td>'GrpH'</td>
<td>554</td>
</tr>
</tbody>
</table>
Hash Join Example

Patient \( \bowtie \) Insurance

Showing pid only

Memory \( M = 21 \) pages
Hash Join Example

Step 1: Scan Patient and create hash table in memory

Memory $M = 21$ pages

Hash $h$: pid $\% 5$

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4 6 6</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3 1 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8 8 9</td>
</tr>
<tr>
<td>8 5</td>
<td></td>
</tr>
</tbody>
</table>

Input buffer

$\leftarrow$

[10]
Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

| 5 | 1 | 6 | 2 | 3 | 8 | 4 | 9 |

Disk

Patient | Insurance
--- | ---
1 | 2
3 | 4
9 | 6
8 | 5

Input buffer: 2 4

Output buffer: 2 2

Write to disk
Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

Disk

Patient | Insurance
---|---
1 2 | 2 4 6 6
3 4 | 4 3 1 3
9 6 | 2 8
8 5 | 8 9

Input buffer: 2 4
Output buffer: 4 4
Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

Disk

Patient | Insurance
---|---
12 | 2466
34 | 4313
96 | 28
85 | 89

Input buffer

Keep going until read all of Insurance

Output buffer

Cost: B(R) + B(S)
Hash Join Details

```java
Open() {
    H = newHashTable();
    S.Open();
    x = S.GetNext();
    while (x != null) {
        H.insert(x); x = S.GetNext();
    }
    S.Close();
    R.Open();
    buffer = [];
}
```
Hash Join Details

```c
GetNext( ) {
    while (buffer == [ ]) {
        x = R.GetNext( );
        if (x==Null) return NULL;
        buffer = H.find(x);
    }
    z = buffer.first( );
    buffer = buffer.rest( );
    return z;
}
```
Hash Join Details

```c
Close() {
    release memory (H, buffer, etc.);
    R.Close();
}
```
Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- $R$ is the outer relation, $S$ is the inner relation

```
for each tuple $r$ in $R$ do
    for each tuple $s$ in $S$ do
        if $r$ and $s$ join then output ($r$, $s$)
```

- Cost: $B(R) + T(R) \cdot B(S)$
- Not quite one-pass since $S$ is read many times
for each page of tuples r in R do
  for each page of tuples s in S do
    for all pairs of tuples
      if r and s join then output (r,s)

• Cost: $B(R) + B(R)B(S)$
Nested Loop Example

Disk

Patient | Insurance
---|---
1 2 | 2 4 | 6 6
3 4 | 4 3 | 1 3
9 6 | 2 8 |
8 5 | 8 9 |

1 2 | Input buffer for Patient
2 4 | Input buffer for Insurance
2 2 | Output buffer
Nested Loop Example

Disk

Patient | Insurance
---|---
1 2 | 2 4 | 6 6
3 4 | 4 3 | 1 3
9 6 | 2 8
8 5 | 8 9

Input buffer for Patient

Input buffer for Insurance

Output buffer
Nested Loop Example

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4 6 6</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3 1 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>

Input buffer for Patient

Input buffer for Insurance

Keep going until read all of Insurance

Then repeat for next page of Patient… until end of Patient

Cost: B(R) + B(R)B(S)
Sort-Merge Join

Sort-merge join:  \( R \bowtie S \)
- Scan \( R \) and sort in main memory
- Scan \( S \) and sort in main memory
- Merge \( R \) and \( S \)

- Cost: \( B(R) + B(S) \)
- One pass algorithm when \( B(S) + B(R) \leq M \)
- Typically, this is NOT a one pass algorithm
Sort-Merge Join Example

Step 1: Scan Patient and sort in memory

Memory $M = 21$ pages

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>
Sort-Merge Join Example

Step 2: Scan Insurance and sort in memory

Memory $M = 21$ pages

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4 6 6</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3 1 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8 8 9</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>
Sort-Merge Join Example

Step 3: Merge Patient and Insurance

Memory M = 21 pages

Output buffer
Sort-Merge Join Example

Step 3: Merge Patient and Insurance

Memory M = 21 pages

Disk

Patient Insurance

1 2 2 4 6 6
3 4 4 3 1 3
9 6 2 8
8 5 8 9

Output buffer

Keep going until end of first relation

1 2 3 4 5 6 8 9
1 2 2 3 3 4 4 6
6 8 8 9
2 2

Output buffer
Outline for Today

• Join operator algorithms
  – One-pass algorithms (Sec. 15.2 and 15.3)
  – Index-based algorithms (Sec 15.6)
  – Two-pass algorithms (Sec 15.4 and 15.5)
Review: Access Methods

• **Heap file**
  – Scan tuples one at the time

• **Hash-based index**
  – Efficient selection on equality predicates
  – Can also scan data entries in index

• **Tree-based index**
  – Efficient selection on equality or range predicates
  – Can also scan data entries in index
Index Based Selection

• Selection on equality: $\sigma_{a=v}(R)$

• $V(R, a) = \# \text{ of distinct values of attribute } a$

• Clustered index on $a$: cost $B(R)/V(R,a)$
• Unclustered index on $a$: cost $T(R)/V(R,a)$

• Note: we ignored I/O cost for index pages
Index Based Selection

Example:

\[
\begin{align*}
B(R) &= 2000 \\
T(R) &= 100,000 \\
V(R, a) &= 20
\end{align*}
\]

cost of \( \sigma_{a=v}(R) = ? \)

Table scan: \( B(R) = 2,000 \) I/Os

Index based selection

- If index is clustered: \( B(R)/V(R,a) = 100 \) I/Os
- If index is unclustered: \( T(R)/V(R,a) = 5,000 \) I/Os

Lesson

- Don’t build unclustered indexes when \( V(R,a) \) is small!
Index Nested Loop Join

R \bowtie S

• Assume S has an index on the join attribute
• Iterate over R, for each tuple fetch corresponding tuple(s) from S

• Cost:
  – If index on S is clustered: \( B(R) + T(R)B(S) / V(S,a) \)
  – If index on S is unclustered: \( B(R) + T(R)T(S) / V(S,a) \)
Outline for Today

• Join operator algorithms
  – One-pass algorithms (Sec. 15.2 and 15.3)
  – Index-based algorithms (Sec 15.6)
  – Two-pass algorithms (Sec 15.4 and 15.5)
Two-Pass Algorithms

- What if data does not fit in memory?
- Need to process it in multiple passes

- Two key techniques
  - Hashing
  - Sorting
Two Pass Algorithms Based on Hashing

- Idea: partition a relation $R$ into buckets, on disk
- Each bucket has size approx. $B(R)/M$

Does each bucket fit in main memory? 
- Yes if $B(R)/M \leq M$, i.e. $B(R) \leq M^2$
Partitioned (Grace) Hash Join

$R \bowtie S$

• Step 1:
  – Hash $S$ into $M-1$ buckets
  – Send all buckets to disk

• Step 2
  – Hash $R$ into $M-1$ buckets
  – Send all buckets to disk

• Step 3
  – Join every pair of buckets
Partitioned Hash Join

- Partition both relations using hash fn \( h \)
- \( R \) tuples in partition \( i \) will only match \( S \) tuples in partition \( i \).
Partitioned Hash Join

• Read in partition of R, hash it using $h2 \neq h$
  – Build phase
• Scan matching partition of S, search for matches
  – Probe phase
Partitioned Hash Join

- Cost: $3B(R) + 3B(S)$
- Assumption: $\min(B(R), B(S)) \leq M^2$
External Sorting

• Problem: Sort a file of size $B$ with memory $M$

• Where we need this:
  – ORDER BY in SQL queries
  – Several physical operators
  – Bulk loading of B+-tree indexes.

• Sorting is two-pass when $B < M^2$
External Merge-Sort: Step 1

- Phase one: load M pages in memory, sort.
External Merge-Sort: Step 2

• Merge $M - 1$ runs into a new run
• Result: runs of length $M \times (M - 1) \approx M^2$

If $B \leq M^2$ then we are done
External Merge-Sort

- **Cost:**
  - Read+write+read = 3B(R)
  - Assumption: B(R) <= M^2

- **Other considerations**
  - In general, a lot of optimizations are possible
Two-Pass Join Algorithm Based on Sorting

Join \( R \bowtie S \)

- **Step 1**: sort both \( R \) and \( S \) on the join attribute:
  - Cost: \( 4B(R)+4B(S) \) (because need to write to disk)
- **Step 2**: Read both relations in sorted order, match tuples
  - Cost: \( B(R)+B(S) \)
- **Total cost**: \( 5B(R)+5B(S) \)
- **Assumption**: \( B(R) \leq M^2, B(S) \leq M^2 \)
Two-Pass Join Algorithm Based on Sorting

Join R \bowtie S

- If \( B(R) + B(S) \leq M^2 \)
  - Or if use a priority queue to create runs of length \( 2|M| \)
- If the number of tuples in R matching those in S is small (or vice versa)
- We can compute the join during the merge phase

- Total cost: \( 3B(R)+3B(S) \)
Summary of Join Algorithms

• **Nested Loop Join**: $B(R) + B(R)B(S)$
  – Assuming page-at-a-time refinement

• **Hash Join**: $3B(R) + 3B(S)$
  – Assuming: $\min(B(R), B(S)) \leq M^2$

• **Sort-Merge Join**: $3B(R) + 3B(S)$
  – Assuming $B(R) + B(S) \leq M^2$

• **Index Nested Loop Join**: $B(R) + \frac{T(R)B(S)}{V(S,a)}$
  – Assuming $S$ has clustered index on $a$
Summary of Query Execution

• For each logical query plan
  – There exist many physical query plans
  – Each plan has a different cost
  – Cost depends on the data

• Additionally, for each query
  – There exist several logical plans

• Next lecture: query optimization
  – How to compute the cost of a complete plan?
  – How to pick a good query plan for a query?