Lecture 21:

Wednesday, November 20, 2002

Outline

• Query execution: 15.1 – 15.5

One-pass Algorithms

Hash join: R ⋈ S
• Scan S, build buckets in main memory
• Then scan R and join

• Cost: B(R) + B(S)
• Assumption: B(S) <= M

One-pass Algorithms

Duplicate elimination δ(R)
• Need to keep tuples in memory
• When new tuple arrives, need to compare it with previously seen tuples
• Balanced search tree, or hash table
• Cost: B(R)
• Assumption: B(δ(R)) <= M

Question in Class

Grouping:

Product(name, department, quantity)
γ_{department, sum(quantity)}(Product) → Answer(department, sum)

• Question: how do you compute it in main memory?
• Answer:
One-pass Algorithms

Binary operations: \( R \cap S, R \cup S, R - S \)
- Assumption: \( \min(B(R), B(S)) \leq M \)
- Scan one table first, then the next, eliminate duplicates
- Cost: \( B(R) + B(S) \)

Question in Class

Fill in missing lines to compute \( R \cup S \)

\[
H \leftarrow \text{emptyHashTable}
\]

* scan \( R \)*

For each \( x \) in \( R \)
insert(\( H, \))

* scan \( S \)*

For each \( y \) in \( S \)

* collect result *

for each \( z \) in \( H \)

Question in Class

Fill in missing lines to compute \( R - S \)

\[
H \leftarrow \text{emptyHashTable}
\]

* scan \( R \)*

For each \( x \) in \( R \)
insert(\( H, \))

* scan \( S \)*

For each \( y \) in \( S \)

* collect result *

for each \( z \) in \( H \)

Question in Class

Fill in missing lines to compute \( R \cap S \)

\[
H \leftarrow \text{emptyHashTable}
\]

* scan \( R \)*

For each \( x \) in \( R \)
insert(\( H, \))

* scan \( S \)*

For each \( y \) in \( S \)

* collect result *

for each \( z \) in \( H \)

Question in Class

Fill in missing lines to compute \( R \cap S \)

\[
H \leftarrow \text{emptyHashTable}
\]

* scan \( R \)*

For each \( x \) in \( R \)
insert(\( H, \))

* scan \( S \)*

For each \( y \) in \( S \)

* collect result *

for each \( z \) in \( H \)

Nested Loop Joins

- Tuple-based nested loop \( R \bowtie S \)

\[
\text{for each tuple } r \text{ in } R \text{ do}
\]
\[
\text{for each tuple } s \text{ in } S \text{ do}
\]
\[
\text{if } r \text{ and } s \text{ join then output } (r, s)
\]

- Cost: \( T(R) \ T(S) \), sometimes \( T(R) \ B(S) \)

Nested Loop Joins

- We can be much more clever

- Question: how would you compute the join in the following cases? What is the cost?
  - \( B(R) = 1000, B(S) = 2, M = 4 \)
  - \( B(R) = 1000, B(S) = 4, M = 4 \)
  - \( B(R) = 1000, B(S) = 6, M = 4 \)
Nested Loop Joins

- Block-based Nested Loop Join

\[
\text{for each } (M-1) \text{ blocks } bs \text{ of } S \text{ do } \\
\text{for each block } br \text{ of } R \text{ do } \\
\text{for each tuple } s \text{ in } bs \\
\text{for each tuple } r \text{ in } br \text{ do } \\
\text{if } r \text{ and } s \text{ join then output}(r,s)
\]

Two-Pass Algorithms Based on Sorting

- Recall: multi-way merge sort needs only two passes!
- Assumption: \( B(R) \leq M^2 \)
- Cost for sorting: \( 3B(R) \)

Two-Pass Algorithms Based on Sorting

Duplicate elimination \( \delta(R) \)

- Trivial idea: sort first, then eliminate duplicates
- Step 1: sort chunks of size \( M \), write
  - cost \( 2B(R) \)
- Step 2: merge \( M-1 \) runs, but include each tuple only once
  - cost \( B(R) \)
- Total cost: \( 3B(R) \), Assumption: \( B(R) \leq M^2 \)

Two-Pass Algorithms Based on Sorting

Grouping: \( \gamma_{a, \sum(b)}(R) \)

- Same as before: sort, then compute the \( \sum(b) \) for each group of \( a \)'s
- Total cost: \( 3B(R) \)
- Assumption: \( B(R) \leq M^2 \)
Two-Pass Algorithms Based on Sorting

Binary operations: R ∪ S, R ∩ S, R − S
- Idea: sort R, sort S, then do the right thing
- A closer look:
  - Step 1: split R into runs of size M, then split S into runs of size M. Cost: 2B(R) + 2B(S)
  - Step 2: merge M/2 runs from R; merge M/2 runs from S; output a tuple on a case by cases basis
- Total cost: 3B(R)+3B(S)
- Assumption: B(R)+B(S)<= M²

Two-Pass Algorithms Based on Sorting

Join R ∣ S
- 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)
- Assumption: B(R) + B(S) <= M²

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 <= M, i.e. B(R) <= M²

Hash Based Algorithms for δ

- Recall: δ(R) = duplicate elimination
- Step 1. Partition R into buckets
- Step 2. Apply δ to each bucket (may read in main memory)
- Cost: 3B(R)
- Assumption: B(R) <= M²

Hash Based Algorithms for γ

- Recall: γ(R) = grouping and aggregation
- Step 1. Partition R into buckets
- Step 2. Apply γ to each bucket (may read in main memory)
- Cost: 3B(R)
- Assumption: B(R) <= M²
Partitioned Hash Join

\[ R \bowtie S \]

- Step 1:
  - Hash \( S \) into \( M \) buckets
  - Send all buckets to disk
- Step 2
  - Hash \( R \) into \( M \) buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

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 \( h_2 \) (\( \Rightarrow h \)). Scan matching partition of \( S \), search for matches.

Partitioned Hash Join

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

Hybrid Hash Join Algorithm

- Partition \( S \) into \( k \) buckets
- But keep first bucket \( S_1 \) in memory, \( k-1 \) buckets to disk
- Partition \( R \) into \( k \) buckets
  - First bucket \( R_1 \) is joined immediately with \( S_1 \)
  - Other \( k-1 \) buckets go to disk
- Finally, join \( k-1 \) pairs of buckets:
  - \((R_2,S_2), (R_3,S_3), \ldots, (R_k,S_k)\)

Hybrid Join Algorithm

- How big should we choose \( k \) ?
- Average bucket size for \( S \) is \( B(S)/k \)
- Need to fit \( B(S)/k + (k-1) \) blocks in memory
  - \( B(S)/k + (k-1) \leq M \)
  - \( k \) slightly smaller than \( B(S)/M \)

Hybrid Join Algorithm

- How many I/Os ?
- Recall: cost of partitioned hash join:
  - \( 3B(R) + 3B(S) \)
- Now we save 2 disk operations for one bucket
- Recall there are \( k \) buckets
- Hence we save \( 2/k(B(R) + B(S)) \)
- Cost: \( (3-2/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S)) \)
Indexed Based Algorithms

- Recall that in a clustered index all tuples with the same value of the key are clustered on as few blocks as possible

- Note: book uses another term: “clustering index”. Difference is minor…

Index Based Selection

- Selection on equality: \( \sigma_{\text{eq}}(R) \)
- Clustered index on \( a \): cost \( B(R)/V(R,a) \)
- Unclustered index on \( a \): cost \( T(R)/V(R,a) \)

Index Based Selection

- Example: \( B(R) = 2000 \), \( T(R) = 100,000 \), \( V(R, a) = 20 \), compute the cost of \( \sigma_{\text{eq}}(R) \)
- Cost of table scan:
  - If \( R \) is clustered: \( B(R) = 2000 \) I/Os
  - If \( R \) is unclustered: \( T(R) = 100,000 \) I/Os
- Cost of index based selection:
  - If index is clustered: \( B(R)/V(R,a) = 100 \)
  - If index is unclustered: \( T(R)/V(R,a) = 5000 \)
- Notice: when \( V(R,a) \) is small, then unclustered index is useless

Index Based Join

- \( R \bowtie S \)
- Assume \( S \) has an index on the join attribute
- Iterate over \( R \), for each tuple fetch corresponding tuple(s) from \( S \)
- Assume \( R \) is clustered. Cost:
  - If index is clustered: \( B(R) + T(R)B(S)/V(S,a) \)
  - If index is unclustered: \( B(R) + T(R)T(S)/V(S,a) \)

Index Based Join

- Assume both \( R \) and \( S \) have a sorted index (B+ tree) on the join attribute
- Then perform a merge join (called zig-zag join)
- Cost: \( B(R) + B(S) \)