Introduction to Database Systems

CSE 444

Lecture #15
Feb 28 2001

Announcement

- Project Report due today
- HW#4 available on the web
  - Optional, but you can only benefit from it!
- Lecture on March 5
  - Given by Vivek Narasayya (my colleague)
  - Material included in Finals
  - Discussion on Finals postponed to beginning of lecture on March 7
- Watch posting on mailing list
  - Limited exclusion of material

Review of Selected Material

Insertion in Extensible Hash Table

- Insert 1110

Insertion in Extensible Hash Table

- Now insert 1010
  - Need to extend table, split blocks
  - $i$ becomes 2

Insertion in Extensible Hash Table

- Now insert 1110
**Insertion in Extensible Hash Table**

- Now insert 0000, then 0101
  - Need to split block

**Insertion in Extensible Hash Table**

- After splitting the block

**Linear Hash Table Example**

- N=3
  - Insert 1000: overflow blocks...

**Linear Hash Table Example**

- Insert 1000: overflow blocks...

**Linear Hash Table Extension**

- From n=3 to n=4
  - Only need to touch one block (which one?)

**Compressed BitMaps: Run Length Encoding**

- Represent sequence of I 0-s followed by 1 as a binary encoding of I
- Concatenate codes for each run together
- But, must be able to recover runs
- Scheme
  - B_I = # of bits in binary encoding of I
  - Represent as B_I – 1 1-s followed by 0 and then binary encoding of I
Indexes: Compressed BitMap

% Decode: (11101101001011)

% Run-Length: (13,0,3): Why?
% 00000000000110001
% Note: Trailing 0-s not recovered

Indexes: Multi-column or Multiple Indexes

% Multi-column index
  - On concatenation of field1 and field2
  - Asymmetric for B+ Trees
% Index AND-ing and OR-ing
  - For Selection
  - For Join

Indexing: When are indexes useful?

% Select Name, Age
% From Person
% Where Person.salary > 100 K and
  Person.state IN [NY, CA, WA]
% Group By City

Query Execution (Contd.)

Required Reading: 2.3.3-2.3.5, 6.1-6.7
Suggested Reading: 6.8, 6.9

Review of Last Lecture

2-Way Merge Sort

% Each pass we read + write each page in file.
% N pages in the file => the number of passes
% \( \lceil \log_2 N \rceil + 1 \)
% So total cost is:
% \( 2N(\lceil \log_2 N \rceil + 1) \)
% Improvement: start with larger runs
% Sort 1GB with 1MB memory in 10 passes
Multiway Merge-Sort

Phase one: load M bytes in memory, sort
Result: runs of length M/R records

Phase Two
Merge M/B – 1 runs into a new run
Result: runs have now M/R (M/B – 1) records

Phase Three
Merge M/B – 1 runs into a new run
Result: runs have now M/R (M/B – 1)^2 records

Cost of External Merge Sort
Number of passes:
1 + \left\lceil \frac{\log M}{\log B} \right\rceil \frac{N R}{M}

Logical and Physical Operators

SELECT S.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
Q.city='seattle' AND
Q.phone > '5430000'

Query Plan:
• logical tree
• implementation choice at every node
• scheduling of operations

Some operators are from relational algebra, and others (e.g., scan, group) are not.

Estimating the Cost of Operators

Very important for the optimizer (next week)
Parameters for a relation R
- B(R) = number of blocks holding R
  Meaningful if R is clustered
- T(R) = number of tuples in R
  E.g. may need when R is unclustered
- V(R,a) = number of distinct values of the attribute a
Scanning Tables

- The table is **clustered**
  - Table-scan: if we know where the blocks are
- The table is unclustered (e.g. its records are placed on blocks with other tables)
  - May need one read for each record
- Also, index scan (discussed later)

Cost of the Scan Operator

- Clustered relation:
  - B(R); to sort: 3B(R)
- Unclustered relation
  - T(R); to sort: T(R) + 2B(R)

Sorting While Scanning

- Sometimes it is useful to have the output sorted
- Three ways to scan it sorted:
  - If it fits in memory, sort there
  - If not, use multiway merging

One-pass Algorithms

- Grouping: \( \gamma_{city, sum(price)}(R) \)
- Need to store all cities in memory
- Also store the sum(price) for each city
- Balanced search tree or hash table
- Cost: B(R)
- Assumption: number of cities fits in memory

Nested Loop Joins

- Block-based Nested Loop Join
  - For each \((M-1)\) blocks \(bs\) of \(S\) do
    - for each block \(br\) of \(R\) do
      - for each tuple \(s\) in \(bs\)
        - for each tuple \(r\) in \(br\) do
          - if \(r\) and \(s\) join then output\((r,s)\)
**Nested Loop Joins**

- **Block-based Nested Loop Join**
- **Cost:**
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-1) times, and each time need to read R: costs B(S)B(R)/(M-1)
  - Total cost: B(S) + B(S)B(R)/(M-1)
- Notice: it is better to iterate over the smaller relation first
- R ⨯ S: R = outer relation, S = inner relation

**Two-Pass Algorithms Based on Sorting**

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

**Grouping:** γ_city, sum(price) (R)

- Same as before: sort, then compute the sum(price) for each group
- As before: compute sum(price) during the merge phase.
- Total cost: 3B(R)
- Assumption: B(R) <= M^2

**Two-Pass Join Algorithms Based on Sorting**

- Start by sorting both R and S on the join attribute:
  - Cost: 4B(R) + 4B(S) (because need to write to disk)
  - Read both relations in sorted order, match tuples
  - Cost: B(R) + B(S)
  - Difficulty: many tuples in R may match many in S
    - If at least one set of tuples fits in M, we are OK
    - Otherwise need nested loop
  - Total cost: 5B(R) + 5B(S)
  - Assumption: B(R) <= M^2, B(S) <= M^2

**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) <= M^2

**Query Execution (contd.)**

[New Material]
**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) <= M^2

**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) <= M^2

**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) <= M^2

**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-based Join**

R \(\bowtie\) S
- **Recall the main memory hash-based join:**
  - Scan S, build buckets in main memory
  - Then scan R and join

**Hash-Join**

- **Partition both relations using hash fn h:** R tuples in partition i will only match S tuples in partition i.
- **Read in a partition of R, hash it using h2 (\(\Leftrightarrow\) h1). 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 $B(S)/M$

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: $\sigma_{a=v}(R)$
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 \( \sigma_{a=v(R)} \).

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

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)

Optimization

Algebraic laws provide alternative execution plans.
Estimate costs of alternative modes of execution.
Efficiently search the space of alternatives.
- Simplify search by applying heuristics (without costing)
- Apply laws that seem to result in cheaper plans

Converting from SQL to Logical Plans

Select a1, ..., an
From R1, ..., Rk
Where C
\[ \Pi_{a_1, \ldots, a_n}( \sigma_C(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k) ) \]

Converting from SQL to Logical Plans

Select a1, ..., an
From R1, ..., Rk
Where C
Group by b1, ..., bl
\[ \Pi_{a_1, \ldots, a_n}( \gamma_{b_1, \ldots, b_l, \text{aggs}}( \sigma_C(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k) ) ) \]
Algebraic Laws

Commutative and Associative Laws
R ∪ S = S ∪ R, R ∪ (S ∪ T) = (R ∪ S) ∪ T
R ∩ S = S ∩ R, R ∩ (S ∩ T) = (R ∩ S) ∩ T
R ▷◁ S = S ▷◁ R, R ▷◁ (S ▷◁ T) = (R ▷◁ S)

Distributive Laws
R ▷◁ (S ∪ T) = (R ▷◁ S) ∪ (R ▷◁ T)

Example:
R(A, B, C, D), S(E, F, G)
σ F=3 (R ∪ S) = ?
σ A=5 AND G=9 (R ∪ S) = ?

Laws involving selection:
σ C AND C'(R) = σ C(R) ∩ σ C(R)
σ C OR C'(R) = σ C(R) ∪ σ C(R)
σ C(R ▷◁ S) = σ C(R) ▷◁ S

Laws involving projections
Π M(R ∪ S) = Π M(Π N(R) Π P(S))
Π M(Π N(R)) = Π M,N(R)

Example:
R(A,B,C,D), S(E, F, G)
Π A,B,G(R × S) = Π A,B,G(R) × Π A,B,G(S)

Heuristic: Predicate Pushdown
σ price>100 AND city="Seattle" (Product, Company)
σ price>100 AND city="Seattle" (Product, Company)
σ price>100 AND city="Seattle" (Product, Company)
The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to loose an important ordering of the tuples).

Determining Join Order
Select-project-join
Push selections down, pull projections up
Hence: we need to choose the join order
This is the main focus of an optimizer