Introduction to Database Systems

CSE 444

Lecture #14
Feb 26 2001

B+ Tree and Indexes

Index on composite (concatenated) key:
(last name, first name)

What's the impact of ordering?

Index AND-ing or OR-ing

Age between \([40, 50]\) and Salary between \([100, 200]\)

Obtain the pointers (record identifiers) to data file for each qualifying leaf node

Sort and intersect (union)

Extensible Hash Table

For \(i=1, n=2, k=4\)

E.g. \(i=1, n=2, k=4\)

\(0(010)\)

\(1(011)\)

\(1(011)\)

Note: we only look at the first bit (0 or 1)

Insertion in Extensible Hash Table

Insert 1110

\(0(011)\)

\(1(110)\)

\(1(110)\)

Insert 1010

\(0(010)\)

\(1(013)\)

\(1(013)\)

Need to extend table, split blocks

\(i\) becomes 2
**Insertion in Extensible Hash Table**

Now insert 1110

\[ \begin{array}{c|c|c|c|c|c} 
00 & 01 & 10 & 11 \hline 
\hline 00(00) & 10(11) & 10(10) & 11(10) \end{array} \]

\( i=2 \)

**Insertion in Extensible Hash Table**

Now insert 0000, then 0101

\[ \begin{array}{c|c|c|c|c|c} 
00 & 01 & 10 & 11 \hline 
\hline 00(0000), 00(00) & 10(10) & 10(10) & 11(10) \end{array} \]

\( i=2 \)

Need to split block

**Insertion in Extensible Hash Table**

After splitting the block

\[ \begin{array}{c|c|c|c|c|c} 
00 & 01 & 10 & 11 \hline 
\hline 00(000), 00(00) & 10(10) & 10(10) & 11(10) \end{array} \]

\( i=2 \)

**Linear Hash Table Example**

N=3

\[ \begin{array}{c|c|c|c|c|c} 
00 & 01 & 10 \hline 
\hline 01(00) & 10(11) & 10(10) \end{array} \]

BIT FLIP

**Linear Hash Tables**

Key parameters

- \( I \) #of discriminating bits, \( N \) #of buckets, \( R \) # of records
- \( \text{Capacity Threshold} = \frac{R}{N} \)

Extension:

- when capacity threshold exceeds (say) 80%
- independent on overflow blocks
**Linear Hash Table Extension**

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

**BitMap Indexes (Reading: 5.4.1-5.4.3)**

- Bit Vector for every distinct value in the column
- As many bits as there are records in the data
- R1:25, R2:50 R3:25 R4:50 R5:50 R6:70 R7:70 R8:25
- 25: 10100001; 50: 01011000 70: 00000110
- Easy Index OR-ing (score = 25 or score = 50)
- Easy Index AND-ing (last score = new score)

**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

---

**Example**

- 13 0-s followed by 1. 4 bits to represent 13. Hence represent as (11101101)
- Decode: (11101101001011)
- Run-Length: (13,0,3)
- 0000000000000110001
- Note: Trailing 0-s not recovered

**Index AND-ing and OR-ing**

- Decode and then do Index AND and OR
- Can do stepwise
  - Decode one run at a time
  - Read Example 5.26
Query Execution

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

An Algebra for Queries

Logical Operators in the Algebra

- Union, intersection, difference
- Selection \( \sigma \)
- Projection \( \Pi \)
- Join \( \Join \)
- Duplicate elimination \( \delta \)
- Grouping \( \gamma \)
- Sorting \( \tau \)

Example

```
Select city, count(*)
From sales
Group by city
Having sum(price) > 100
```

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

Scanning Tables

- The table is clustered (i.e. blocks consists only of records from this table):
  - Table-scan: if we know where the blocks are
  - Index scan: if we have a sparse index to find the blocks
- The table is unclustered (e.g. its records are placed on blocks with other tables)
  - May need one read for each record
**Sorting While Scanning**

- Sometimes it is useful to have the output sorted.
- Three ways to scan it sorted:
  - If there is a primary or secondary index on it, use it during scan.
  - If it fits in memory, sort there.
  - If not, use multiway merging.

**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.

**Sorting**

- Illustrates the difference in algorithm design when your data is not in main memory:
  - Problem: sort 1Gb of data with 1Mb of RAM.
  - Arises in many places in database systems:
    - Data requested in sorted order (ORDER BY).
    - Needed for grouping operations.
    - First step in sort-merge join algorithm.
    - Duplicate removal.
    - Bulk loading of B+-tree indexes.

**2-Phase Merge-sort: Requires 3 Buffers**

- Phase 1: Read a page, sort it, write it.
  - Only one buffer page is used.
- Phase 2: Merge all sorted sublists.
  - Three buffer pages used.

**2-Way Merge Sort**

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

**Can We Do Better?**

- We have more main memory.
- Should use it to improve performance.
Cost Model for Our Analysis

- \(B\): Block size
- \(M\): Size of main memory
- \(N\): Number of records in the file
- \(R\): Size of one record

External 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: \(1 + \left\lceil \log_{M/B-1} \left[ N/R/M \right] \right\rceil\)
- Think differently:
  - Given \(B = 4KB, M = 64MB, R = 0.1KB\)
  - Pass 1: runs of length \(M/R = 640000\)
  - Pass 2: runs increase by a factor of \(M/B - 1 = 16000\)
    - Have now sorted runs of \(640000 \times 16000 = 10^{10}\) records
  - Pass 3: runs increase by a factor of \(M/B - 1 = 16000\)
    - Have now sorted runs of \(10^{10}\) records
  - Nobody has so much data!
- Can sort everything in 2 or 3 passes!

Cost of the Scan Operator

- Clustered relation:
  - Table scan: \(B(R)\); to sort: \(3B(R)\)
  - Index scan: \(B(R)\); to sort: \(B(R)\) or \(3B(R)\)
- Unclustered relation
  - \(T(R)\); to sort: \(T(R) + 2B(R)\)
### One-Pass Algorithms

Selection $\sigma(R)$, projection $\Pi(R)$

%Both are *tuple-at-a-Time* algorithms
%Cost: $B(R)$

<table>
<thead>
<tr>
<th>Input buffer</th>
<th>Unary operator</th>
<th>Output buffer</th>
</tr>
</thead>
</table>

### One-pass Algorithms

Duplicate elimination $\delta(R)$

%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: $B(\delta(R)) \leq M$

### One-pass Algorithms

Grouping: $\gamma_{\text{city}, \text{sum(price)}}(R)$

%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

### 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: $B(R) + B(S)$

### Nested Loop Joins

%Tuple-based nested loop $R \bowtie S$

For each tuple $r$ in $R$
  For each tuple $s$ in $S$
    if $r$ and $s$ join then output $(r,s)$
%Cost: $T(R) T(S)$, sometimes $T(R) B(S)$

### Nested Loop Joins

%Block-based Nested Loop Join

For each (M-1) blocks $b$ of $S$ do
  for each block $b_r$ of $R$ do
    for each tuple $s$ in $b$
      for each tuple $r$ in $b_r$
        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
- $R \bowtie S$: $R$=outer relation, $S$=inner relation

Two-Pass Algorithms Based on Sorting

Recall: multi-way merge sort needs only two passes!
Assumption: $B(R) \leq M^2$
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: $\gamma_{city, \text{sum(price)}}(R)$
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: $B(R) \leq M^2$

Two-Pass Algorithms Based on Sorting

Binary operations: $R \cap S, R U S, R - S$
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: $B(R)+B(S)\leq M^2$
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: $B(R) \leq M^2$, $B(S) \leq M^2$

Two-Pass Algorithms Based on Sorting

Join R $\bowtie$ 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: $B(R) + B(S) \leq M^2$

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. $B(R) \leq M^2$

Hash Based Algorithms for $\delta$

Recall: $\delta(R) =$ duplicate elimination
- Step 1. Partition R into buckets
- Step 2. Apply $\delta$ to each bucket (may read in main memory)
- Cost: $3B(R)$
- Assumption: $B(R) \leq M^2$

Hash Based Algorithms for $\gamma$

Recall: $\gamma(R) =$ grouping and aggregation
- Step 1. Partition R into buckets
- Step 2. Apply $\gamma$ to each bucket (may read in main memory)
- Cost: $3B(R)$
- Assumption: $B(R) \leq M^2$

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
Partitioned Hash Join

\[ R \bowtie S \]

1. **Step 1:**
   - Hash \( S \) into \( M \) buckets
   - Send all buckets to disk

2. **Step 2**
   - Hash \( R \) into \( M \) buckets
   - Send all buckets to disk

3. **Step 3**
   - Join every pair of buckets

**Cost:** \( 3B(R) + 3B(S) \)

**Assumption:** \( \min(B(R), B(S)) \leq M^2 \)

Hash Join

1. **Step 1:** Partition both relations using hash function \( h \): \( R \) tuples in partition \( i \) will only match \( S \) tuples in partition \( i \).
2. **Step 2:**
   - Read in a partition of \( R \), hash it using \( h_2 (\leftrightarrow h_1) \). Scan matching partition of \( S \), search for matches.

Hybrid Hash Join Algorithm

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

**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 \( B(S)/M \)

**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:
  - \( (3-2/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S)) \)