Query Processing with Indexes

CPS 216
Advanced Database Systems

Announcements (February 24)

❖ More reading assignment for next week
  ▪ Buffer management (due next Wednesday)
❖ Homework #2 due next Thursday
❖ Course project proposal due in 1½ weeks
❖ Midterm in two weeks

❖ Christos Faloutsos (CMU) talk
  ▪ "Data Mining Using Fractals and Power Laws"
  ▪ 4-5pm, Monday, February 28
  ▪ 130A North Building (telecast from UNC)

Review

❖ Many different ways of processing the same query
  ▪ Scan (e.g., nested-loop join)
  ▪ Sort (e.g., sort-merge join)
  ▪ Hash (e.g., hash join)
  ☞ Index

Selection using index

❖ Equality predicate: $\sigma_{A = v}(R)$
  ▪ Use an ISAM, B$^+$-tree, or hash index on $R(A)$
❖ Range predicate: $\sigma_{A > v}(R)$
  ▪ Use an ordered index (e.g., ISAM or B$^+$-tree) on $R(A)$
  ▪ Hash index is not applicable

❖ Indexes other than those on $R(A)$ may be useful
  ▪ Example: B$^+$-tree index on $R(A, B)$
  ▪ How about B$^+$-tree index on $R(B, A)$?

Index versus table scan

Situations where index clearly wins:
❖ Index-only queries which do not require retrieving actual tuples
  ▪ Example: $\pi_A(\sigma_{A > v}(R))$
❖ Primary index clustered according to search key
  ▪ One lookup leads to all result tuples in their entirety

Index versus table scan (cont'd)

BUT(!):
❖ Consider $\sigma_{A > v}(R)$ and a secondary, non-clustered index on $R(A)$
  ▪ Need to follow pointers to get the actual result tuples
  ▪ Say that 20% of R satisfies $A > v$
    • Could happen even for equality predicates
  ▪ I/O's for index-based selection: lookup $+$ 20% $|R|$
  ▪ I/O's for scan-based selection: $B(R)$
  ▪ Table scan wins if a block contains more than 5 tuples

Index nested-loop join

❖ $R \bowtie_{R.A = S.B} S$
❖ Idea: use the value of $R.A$ to probe the index on $S(B)$
❖ For each block of R, and for each r in the block:
    Use the index on $S(B)$ to retrieve s with $s.B = r.A$
    Output rs
❖ I/O's: $B(R) + |R| \cdot$ (index lookup)
  ▪ Typically, the cost of an index lookup is 2-4 I/O's
  ▪ Beats other join methods if $|R|$ is not too big
  ▪ Better pick R to be the smaller relation
❖ Memory requirement: 2

Tricks for index nested-loop join

Goal: reduce $|R| \cdot$ (index lookup)
❖ For tree-based indexes, keep the upper part of the tree in memory
❖ For extensible hash index, keep the directory in memory
❖ Sort or partition R according to the join attribute
  ▪ Improves locality: subsequent lookup may follow the same path or go to the same bucket

Zig-zag join using ordered indexes

❖ $R \bowtie_{R.A = S.B} S$
❖ Idea: use the ordering provided by the indexes on $R(A)$ and $S(B)$ to eliminate the sorting step of sort-merge join
❖ Trick: use the larger key to probe the other index
  ▪ Possibly skipping many keys that do not match

B$^+$-tree on $R(A)$
⇨1 ⇨2   3   4 ⇨7 ⇨9 ⇨18

⇨1 ⇨7 ⇨9 ⇨11   12   17 ⇨19
B$^+$-tree on $S(B)$
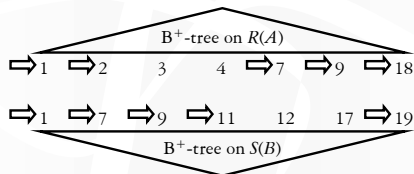
More indexes ahead!

❖ Bitmap index
  ▪ Generalized value-list index
❖ Projection index

❖ Bit-sliced index

Search key values × tuples

Tuples

Search key values	0	1	2	...	$n - 1$
8	1	1	0	...	0
9	0	0	0	...	0
10	0	0	1	...	1
26	0	0	0	...	0
108	0	0	0	...	0
...	...	...	...	...	...

1 means tuple has the particular search key value
0 means otherwise

❖ Looks familiar?
  ▪ Keywords × documents

Bitmap index

❖ Value-list index—stores the matrix by rows
  ▪ Traditionally list contains pointers to tuples
  ▪ B$^+$-tree: tuples with same search key values
  ▪ Inverted list: documents with same keywords
❖ If there are not many search key values, and there are lots of 1's in each row, pointer list is not space-efficient
  ▪ How about a bitmap?
  ▪ Still a B$^+$-tree, except leaves have a different format

Technicalities

- ❖ How do we go from a bitmap index (0 to $n-1$) to the actual tuple?
- ☞ One more level of indirection solves everything
- ☞ Or, given a bitmap index, directly calculate the physical block number and the slot number within the block for the tuple
- ❖ In either case, certain block/slot may be invalid
  - ▪ Because of deletion, or variable-length tuples
  - ▪ Keep an existence bitmap: bit set to 1 if tuple exists

Bitmap versus traditional value-list

- ❖ Operations on bitmaps are faster than pointer lists
  - ▪ Bitmap AND: bit-wise AND
  - ▪ Value-list AND: sort-merge join
- ❖ Bitmap is more efficient when the matrix is sufficiently dense; otherwise, pointer list is more efficient
  - ▪ Smaller means more in memory and fewer I/O's
- ❖ Generalized value-list index: with both bitmap and pointer list as alternatives

Projection index

- ❖ Just store $\pi_A(R)$ and use it as an index!

Could be implicit and not explicitly stored

TID	A	B	...
0	8	...	...
1	8	...	...
2	26	...	...
3	108	...	...
...	...	...	...
$n-1$	10	...	...

Projection index

Why projection index?

- ❖ Idea: still a table scan, but we are scanning a much smaller table (project index)
  - ▪ Savings could be substantial for long tuples with lots of attributes
- ❖ Looks familiar?
  - ▪ DSM!
  - ▪ Except that we keep the original table

Bit-sliced index

- ❖ If a column stores binary numbers, then slice their bits vertically
  - ▪ Basically a projection index by slices

Projection index

TID	A							
0	0	0	0	0	1	0	0	0
1	0	0	0	0	1	0	0	0
2	0	0	0	1	1	0	1	0
3	0	1	1	0	1	1	0	0
...								
$n-1$	0	0	0	0	1	0	1	0

Slice 7 ... Slice 0

Bit-sliced index

Aggregate query processing example

```
SELECT SUM(dollar_sales)
FROM Sales
WHERE condition;
```

- ❖ Already found B_f (a bitmap or a sorted list of TID's that point to *Sales* tuples that satisfy *condition*)
  - ▪ Probably used a secondary index
- ❖ Need to compute SUM(*dollar_sales*) for tuples in B_f

SUM without any index

❖ For each tuple in B_f, go fetch the actual tuple, and add *dollar_sales* to a running sum

❖ I/O's: number of *Sales* blocks with B_f tuples
  ▪ Assuming we fetch them in sorted order

SUM with a value-list index

❖ Assume a value-list index on *Sales*(*dollar_sales*)
❖ Idea: the index stores *dollar_sales* values and their counts (in a pretty compact form)

❖ sum = 0;
  Scan *Sales*(*dollar_sales*) index; for each indexed value v with value-list B_v:
      sum += $v \times$ count-1-bits(B_v AND B_f);

❖ I/Os: number of blocks taken by the value-list index
❖ Bitmaps can possibly speed up AND and reduce the size of the index

SUM with a projection index

❖ Assume a project index on *Sales*(*dollar_sales*)
❖ Idea: merge join B_f and the projection index, add joining tuples' *dollar_sales* to a running sum
  ▪ Assuming both B_f and the index are sorted on TID

❖ I/O's: number of blocks taken by the projection index
  ▪ Compared with a value-list index, the projection index may be more compact (no empty space or pointers), but it does store duplicate *dollar_sales* values
❖ Also: simpler algorithm, fewer CPU operations

SUM with a bit-sliced index

❖ Assume a bit-sliced index on *Sales*(*dollar_sales*), with slices $B_{k-1}, \ldots, B_1, B_0$

❖ sum = 0;
  for i = 0 to $k-1$:
      sum += $2^i \times$ count-1-bits(B_i AND B_f);

❖ I/O's: number of blocks taken by the bit-sliced index
❖ Conceptually a bit-sliced index contains the same information as a projection index
  ▪ But the bit-sliced index does not keep TID
  ▪ Bitmap AND is faster

Summary of SUM

❖ Best: bit-sliced index
  ▪ Index is small
  ▪ B_f can be applied fast!
❖ Good: projection index
❖ Not bad: value-list index
  ▪ Full-fledged index carries a bigger overhead
    • The fact that we have counts of values helped
    • But we did not really need values to be ordered

MEDIAN

    SELECT MEDIAN(*dollar_sales*)
    FROM *Sales*
    WHERE *condition*;

❖ Same deal: already found B_f (a bitmap or a sorted list of TID's that point to *Sales* tuples that satisfy *condition*)
❖ Need to find the *dollar_sales* value that is greater than or equal to ½ $\times$ count-1-bits(B_f) *dollar_sales* values among B_f tuples

MEDIAN with an ordered value-list index

❖ Idea: take advantage of the fact that the index is ordered by *dollar_sales*

❖ Scan the index in order, count the number of tuples that appeared in B_f until the count reaches ½ × count-1-bits(B_f)

❖ I/O's: roughly half of the index

MEDIAN with a projection index

❖ In general, need to sort the index by *dollar_sales*
  ▪ Well, when you sort, you more or less get back an ordered value-list index!

❖ Not useful unless B_f is small

MEDIAN with a bit-sliced index

❖ Tough at the first glance—index is not sorted
❖ Think of it as sorted
  ▪ We won't actually make use of the this fact

Look at B_{k-1} first
More than half are 0's?

$$
\begin{array}{c|c}
0 & 0 \;\; 0\dots \\
0 & 0 \;\; 1\dots \\
\hline
1 & 0 \;\; 0\dots \\
1 & 1 \;\; 0\dots \\
1 & 1 \;\; 1\dots \\
\end{array}
$$

Yes; continue searching for median here

No; continue searching for median here

By looking at B_{k-1} we know the $(k-1)$-th bit of the median

MEDIAN with a bit-sliced index

❖ median = 0;
  $B_{current} = B_f$; // which tuples we are considering
  sofar = 0; // number of tuples whose values are less
  // than what we are considering
  for $i = k - 1$ to 0:
    if (sofar + count-1-bits($B_{current}$ AND NOT(B_i))
          ≤ ½ × count-1-bits(B_f)):
      $B_{current} = B_{current}$ AND B_i;
      sofar += count-1-bits($B_{current}$ AND NOT(B_i);
      median += 2^i;
    else:
      $B_{current} = B_{current}$ AND NOT(B_i);

❖ I/O's: still need to scan the entire index

Summary of MEDIAN

❖ Best: ordered value-list index
  ▪ It helps to be ordered!

❖ Pretty good: bit-sliced index
  ▪ Could beat ordered value-list index if B_f is "clustered"
    • Only need to retrieve the corresponding segment

More variant indexes

"Improved Query Performance with Variant Indexes," by O'Neil and Quass. *SIGMOD*, 1997

❖ MIN/MAX, and range query using bit-sliced index
❖ Join indexes for star schema
  ▪ Traditional: one for each combination of foreign columns
  ▪ Bitmap: one for each foreign column
☞ Precomputed query results (materialized views)?

Variant vs. traditional indexes

- ❖ What is the more glaring problem of these variant indexes that makes them not as widely applicable as the B^+-tree?
  - ▪ Difficult to update
- ❖ How did the paper get away with that?
  - ▪ OLAP with periodic batch updates