Physical Operators

How the operators of the relational algebra for bags are implemented?

Outline

the data

a relation R is stored on disk

- in $B(R)$ blocs
- with $T(R)$ tuples
- with $V(R,A)$ values for attribute A

we have M memory buffers (blocks) available

assume that

- the arguments of the operators are found on disk
- the result is left in main memory
- the data are accessed one block at a time

a first basic operator

table scan: get the blocks of R one by one

- if R is clustered: costs $B(R)$ I/O's
- if it is not: costs $T(R)$ I/O's

implemented as an iterator:

- Open: initializes the data structures needed to perform the scan
- GetNext: returns the next tuple in the result
- Close: ends the iteration after all tuples have been obtained

different types of operators (1)

- simple scan
- sorting based methods: sort R first
- hash-based methods: hash R first
- index-based methods: use an index on R to scan it (see latter)

different types of operators (2)

- one pass algorithm
  - for small size relations or tuple at a time operations
  - read the data only once
- two pass algorithm
  - for relations not fitting in main memory
  - read once, process, write on disk, read again
- multi pass algorithm
  - no limit on the size of the data
  - generalization of two pass algorithms

different types of operators (3)

- tuple at a time unary operations
  - σ and π
  - do not require the entire relation in memory at once
- full relation, unary operations
  - γ and δ
  - require to have all or almost all tuples in main memory at once
- full relation, binary operations
  - all the other operations
  - require to have all or almost all tuples in main memory at once

one pass, tuple at a time

$\sigma(R)$, $\pi(R)$

- read one block, process the tuples
- requires that $M \geq 1$
- costs $B(R)$ if R is clustered

one pass for unary, full relation

$\delta(R)$

- read one block, keep in memory one copy of each tuple seen
- use 1 memory block for the block read and $M-1$ for the seen tuples
- needs an appropriate in memory data structure (balance tree, hashtable)
- requires that $B(\delta(R)) \leq M-1$
- requires $B(R)$ I/O's

one pass for unary, full relation

$\gamma(R)$

- read one block, keep in memory entries for each group
  - min, max, sum, count require only one entry
  - avg requires two
- use 1 memory block for the block read and $M - 1$ for the groups
- needs an appropriate in memory data structure (balance tree, hashtable)
- memory requirement depends on the size of entries
- requires $B(R)$ I/O's

one pass binary, full relation

$R \cup_B S$

- output R then output S
- requires $M \geq 1$
- requires $B(R) + B(S)$ I/O's

one pass binary, full relation

$\cup_S$, $\cap_S$, $\setminus_S$, $\cap_B$, $\setminus_B$, $\times$, $\bowtie$

- read the smaller of R, S in memory
- build a suitable in memory data structure
- read one block of the bigger table
- requires $B(R) + B(S)$ I/O's
- requires that $min(B(R),B(S)) \leq M - 1$

example of $\cap_B$

assume $B(S) = min(B(R), B(S))$

record the count for each $t \in S$

1. read one block of R, for each t'
2. if $t' \in S$
   2.1 decrement the count of t', output t'
3. when count reaches 0, no more output t'

two pass algorithms based on sorting

why sorting?

- order by needs it
- operators more efficient when parameters are sorted

when?

- $B(R) > M$

two pass algorithms based on sorting

the basic idea

repeat:

1. read M blocks of R
2. sort these blocks in main memory (time to sort will not exceed disk I/O time)
3. write this sorted sublist on disk

second pass: read the sorted sublists to process the relational operation

two pass algorithms based on sorting

$\delta(R)$,$\gamma(R)$

for δ:

1. sort the blocks and write the sublists on disk
2. read one block of each sublist
3. copy each tuple to the output ignoring duplicates

two pass algorithms based on sorting

example for $\delta(R)$

assume $M = 3$, tuples are integers, 2 tuples per blocks

$R = 2,5,2,1,2,2,4,5,4,3,4,2,1,5,2,1,3$

step 1: 3 sorted sublists on disk

- $R_1 = 1,2,2,2,2,5$
- $R_2 = 2,3,4,4,4,5$
- $R_3 = 1,1,2,3,5$

two pass algorithms based on sorting

step 2 and 3 (1):

step 2:

sublist	in memory	waiting on disk
R_1	1,2	2,2,2,5
R_2	2,3	4,4,4,5
R_3	1,1	2,3,5

step 3: output 1

two pass algorithms based on sorting

step 2 and 3 (2):

step 2:

sublist	in memory	waiting on disk
R_1	2	2,2,2,5
R_2	2,3	4,4,4,5
R_3	2,3	5

step 3: output 2

two pass algorithms based on sorting

step 2 and 3 (3):

step 2:

sublist	in memory	waiting on disk
R_1	5	
R_2	3	4,4,4,5
R_3	3	5

step 3: output 3 (and so on..)

two pass algorithms based on sorting

conclusion, for δ and γ

- I/O cost: $3 \times B(R)$
- can handle larger files than one pass version
- requires that $B(R) \leq M^2$
  - no more than M sublists
  - each at-most M blocks long

two pass algorithms based on sorting

$R \cup_S S$ (two passes not needed for $\cup_B$)

1. create sorted sublists for R and S
2. read one block of each sublist
3. find the first remaining t, output t, remove all copies of t
4. read the next block of the sublist when the current block is exhausted

two pass algorithms based on sorting

requirements for $R \cup_S S$

- I/O cost: $3 \times (B(R) + B(S))$
- requires that $B(R) + B(S) \leq M^2$
  - no more than M sublists for R and S
  - each at-most M blocks long

same requirements for $\cap$ and $\setminus$

two pass algorithms based on sorting

example for $R \setminus_B S$

assume $M = 3$, tuples are integers, 2 tuples per blocks

$R =$ 2,5,2,1,2,2,4,5,4,3,4,2 and $S =$ 1,5,2,1,3

sublist	in memory	waiting on disk
R_1	1,2	2,2,2,5
R_2	2,3	4,4,4,5
S_1	1,1	2,3,5

remove 1 since $1 \notin R \setminus_B S$

two pass algorithms based on sorting

sublist	in memory	waiting on disk
R_1	2	2,2,2,5
R_2	2,3	4,4,4,5
S_1	2,3	5

output 2 four times

two pass algorithms based on sorting

sublist	in memory	waiting on disk
R_1	5	
R_2	3	4,4,4,5
S_1	3	5

remove 3

two pass algorithms based on sorting

sublist	in memory	waiting on disk
R_1	5	
R_2	4,4	4,5
S_1	5	

output 4 three times

two pass algorithms based on hashing

hashing?

- partition relation into buckets so that every bucket has the same number of tuples
- needs a (carefully chosen) function h that associates each tuple with its bucket number

what for?

- instead of performing the operation on every block of R, use one bucket at a time

two pass algorithms based on hashing

principle: hash R into $M-1$ buckets with h

- use $M-1$ block of memory (one per bucket)
- read one block of R
- hash the tuples to bucket $h(t)$
- when a bucket is full, write it on disk

two pass algorithms based on hashing

$\delta(R)$,$\gamma(R)$

example for $\delta(R)$

- hash R into $M - 1$ buckets
- for each tuple in each bucket, output t and remove duplicates

cost and requirement

- I/O cost is $3 \times B(R)$
- $B(R) \leq M^2$
  - R partitioned into buckets of size $B(R)/M - 1$
  - that number no larger than M

two pass algorithms based on hashing

set operations

- same h function for hashing R and S
- $M-1$ buckets for R : $R_1, \ldots, R_{M-1}$
- $M-1$ buckets for S : $S_1, \ldots, S_{M-1}$
- for all i, compute the one pass operation for R_i with S_i

cost and requirement

- I/O cost is $3 \times (B(R) + B(S))$
- $min(B(R), B(S)) \leq M^2$
  - R (resp. S) partitioned into buckets of size $B(R)/M-1$ (resp. $B(S)/M-1$)
  - one pass operation requires operand of size $\leq M-1$

two pass algorithms based on hashing

hashing or sorting?

- size requirement for hash-based binary operations depends only on the smaller relation
- sorted sublist on consecutive blocks reduces rotational latency or seek time
- only sort based algo can do ORDER BY!

what's next?

- indexing
- query plans, cost estimation
- the join operation:
  - join algorithms
  - join order
  - choosing the join method

see you next semester