Query Processing

- Overview
- Sorting
- Join operators
- Other operators
- Selection operation
- Putting it all together…

Group By and Aggregation

select a, count(a)
from R
group by a;

- Hash-based algorithm:
  - Create a hash table on *a,* and keep the *count(a) so far*
  - Read *R* tuples one by one
  - For a new *R tuple, "r"*
    - Check if *r.a* exists in the hash table
    - If yes, increment the count
    - If not, insert a new value

Group By and Aggregation

select a, count(b)
from R
group by a;

- Sort-based algorithm:
  - Sort *R* on *a*
  - Now all tuples in a single group are contiguous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

Group By and Aggregation

Summary:

- sum(), count(), min(), max(): only need to maintain one value per group
  - Called "distributive"
- average() : need to maintain the "sum" and "count" per group
  - Called "algebraic"
- stddev(): algebraic, but need to maintain some more state
- median(): efficiently via sort, but need two passes (called "holistic")
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- count(distinct b): must do duplicate elimination before the count

Duplicate Elimination

select distinct a
from R ;

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev = r*
    - else
      - Skip *r*

Set operations

*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"*union": append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based algorithm*

Query Processing

- Overview
- Join execution
- Selection operation
- More join execution
- Sorting
- Other operators
- Putting it all together…

"Cost"

- Complicated to compute
- We will focus on disk:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for *b* block transfers plus *S* seeks

    $$b * t_T + S * t_S$$

  - Measured in *seconds*

Selection Operation

- SELECT * FROM person WHERE SSN = "123"
- Option 1: Sequential Scan
  - Read the relation start to end and look for "123"
    - Can always be used (not true for the other options)
  - Cost ?
    - *Let b_r = Number of relation blocks*
    - Then:
      - 1 seek and b_r block transfers
    - So:
      - $t_S + b_r * t_T$ *sec*
    - *Improvements:*
      - *If SSN is a key, then can stop when found*
        - *So on average, $b_r/2$ blocks accessed*

Selection Operation

- SELECT * FROM person WHERE SSN = "123"
- Option 2 : Binary Search:
  - Pre-condition:
    - *The relation is sorted on SSN*
    - *Selection condition is an equality*
      - E.g. can't apply to "*Name like '%424%'*"
  - Do binary search
    - Cost of finding the *first* tuple that matches
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
      - *All I/Os are random, so need a seek for all*
        - The last few are short hops, but we ignore such small effects
  - Not quite: What if 10000 tuples match the condition ?
    - Incurs additional cost

Selection Operation

- SELECT * FROM person WHERE SSN = "123"
- Option 3 : Use Index
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all pointers using the index

Selection w/ B+-Tree Indexes

why?	cost of reading the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b - 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

h_i = height of the index

Selection Operation

- Selections involving ranges
  - *select * from accounts where balance > 100000*
  - *select * from matches where matchdate between '10/20/06' and '10/30/06'*
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can't use hash indexes for this purpose

Selection Operation

- Complex selections
  - <u>Conjunctive</u>: *select * from accounts where balance > 100000 and SSN = "123"*
  - <u>Disjunctive</u>: *select * from accounts where balance > 100000 or SSN = "123"*
  - Option 1: Sequential scan
  - Option 2 *(Conjunctive only)*: Using an appropriate index *on one of the conditions*
    - E.g. Use SSN index to evaluate SSN = "123". Apply the second condition to the tuples that match
    - Or do the other way around (if index on balance exists)
    - Which is better ?
  - Option 3 *(Conjunctive only)* : Choose a multi-key index
    - Not commonly available

Selection Operation

- Complex selections
  - Conjunctive: *select * from accounts where balance > 100000 and SSN = "123"*
  - Disjunctive: *select * from accounts where balance > 100000 or SSN = "123"*

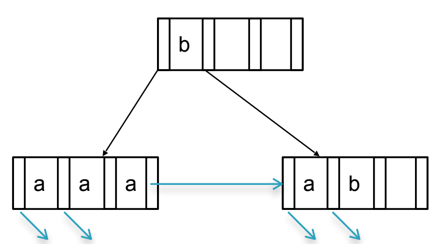
  - Option 4: Conjunction or disjunction of *record identifiers*
    - Use indexes to find all RIDs that match each of the conditions
    - Do an *intersection* (for conjunction) or a *union* (for disjunction)
    - Sort the records and fetch them in one shot
    - Called "Index-ANDing" or "Index-ORing"
  - Heavily used in commercial systems

Secondary B+-Tree Indexes *and many tuples*

- Assume secondary index `R` for non-candidate attribute '`name`'.
- How to hold all ptrs to matching tuples?
  - easiest way is to keep existing structure and duplicate the keys

```
select * from R where name="a"
```

- Assume tree w/ 3 matching tuples, so:




```
c = hR + extra_leaves + tuple_blocks = 1 + 1 + 3
         tree access       read tuples
```

- We assume number of extra leaves = $\left\lceil \frac{\#ptrs}{ptrs/node} \right\rceil - 1.$

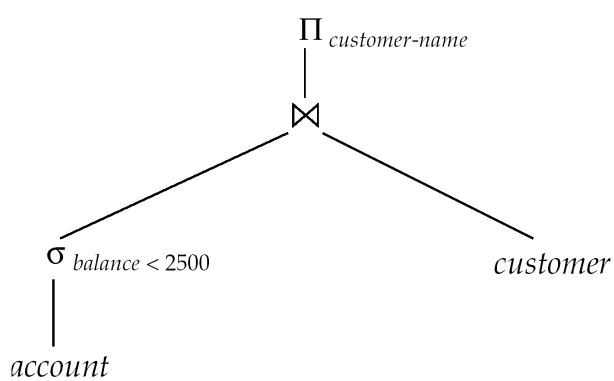
- Each is random read, so here: $c = (1 + 1 + 3) * (t_T + t_S)$

Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Putting it all together…

Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500



- Two options:
  - Materialization
  - Pipelining

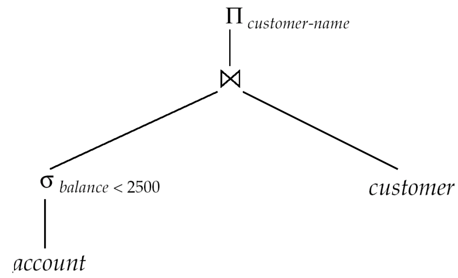
Evaluation of Expressions

- Materialization
  - Evaluate each expression separately
    - Store its result on disk in temporary relations
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
    - Do not go to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about

Materialization

- Materialized evaluation *always* works
- Can be expensive to write and read back from disk
  - Cost formulas ignore cost of writing final results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining



- Evaluate several operations at same time
- passing results from one to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - Instead, pass tuples directly to the join.
  - Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires more memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- Beware blocking operations:
  - must consume entire input before it starts producing output tuples

Pipelining

- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no.
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: *see duplicate elimination*

Pipelining: Demand-driven

- Iterator Interface
  - Each operator implements:
    - init(): Initialize the state (sometimes called open())
    - get_next(): get the next tuple from the operator
    - close(): Finish and clean up
  - Example: sequential scan:
    - init(): open the file
    - get_next(): get the next tuple from file
    - close(): close the file
- Execute by repeatedly calling get_next() at the root
  - root calls get_next() on its children, the children call get_next() on their children etc…
- The operators need to maintain internal state so they know what to do when the parent calls get_next()

Example: Hash-Join Iterator Interface

- open():
  - Call open() on the left and the right children
  - Decide if partitioning needed (if size of smaller relation > memory)
  - Create a hash table
- get_next(): (no partitioning)
  - First call:
    - Get all tuples from the right child one by one (using get_next()), and insert them into the hash table
    - Read the first tuple from the left child (using get_next())
  - All calls:
    - Probe into the hash table using the "current" tuple from the left child
      - Read a new tuple from left child if needed
    - Return exactly "one result"
      - Must keep track if more results need to be returned for that tuple

Hash-Join Iterator Interface

- close():
  - Call close() on the left and the right children
  - Delete the hash table, other intermediate state etc…
- get_next(): (partitioning)
  - First call:
    - Get all tuples from both children and create the partitions on disk
    - Read the first partition for the right child and populate the hash table
    - Read the first tuple from the left child from appropriate partition
  - All calls:
    - Once a partition is finished, clear the hash table, read in a new partition from the right child, and re-populate the hash table
  - Not that much more complicated
- Take a look at the postgreSQL codebase (or assignment 7)

Pipelining (Cont.)

- In producer-driven or *eager* pipelining:
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System runs operations that have space in output buffer and can process more input tuples

Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Quiz 7
- Query *optimization….*

not Homework 7

Q7
10 Points

Consider a relation R(A, B, C, D, E), and the following FDs on it:
B → DE
D → AC
AE → C

The decomposition of R into R1(A, B, C) and R2(C, D, E) is not lossless. Provide an instance of R (i.e., an example set of tuples) that demonstrates it. You don't need more than 2 tuples. Note that the instance must satisfy all the functional dependencies.

Explanation
r(a,b,c,d,e)
b->de
d->ac
ae->c

r1(a,b,c)
no FDs carry

r2(c,d,e)
no FDs carry

R=
1,1,1,2,2
2,2,1,1,1

R1:
1,1,1
2,2,1

R2:
1,2,2
1,1,1

Join back and get:
1,1,1,2,2
1,1,1,1,1
2,2,1,2,2
2,2,1,1,1

Q8
2 Points

For the above schema, which of the following is NOT a lossless decomposition?

- ○ R1(A, B, C), R2(B, C, D, E)
- ○ R1(A, C, D), R2(B, D, E)
- ● R1(A, C, D), R2(A, C, B, E)
- ○ R1(A, B, D, E), R2(B, C)

I deleted this question this year, maybe I'll put it on the final.