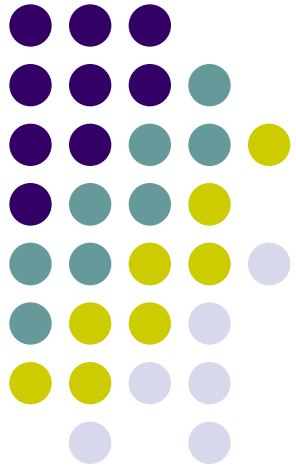
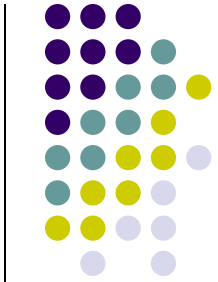
CMSC424: Database Design

Instructor: Amol Deshpande

amol@cs.umd.edu

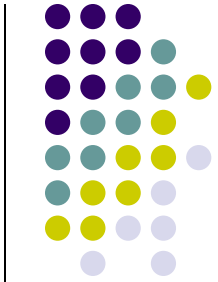


Spring 2020 – Online Instruction Plan



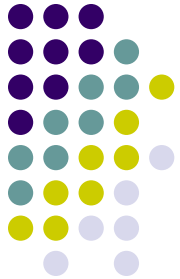
- Week 1: File Organization and Indexes
- Week 2 (Homework due April 6, noon):
  - Overview, Measures of Cost, Selections
  - Join Operation
  - Sorting, and Other Operators
- Week 3: Query Optimization; Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

Sorting and Other Issues



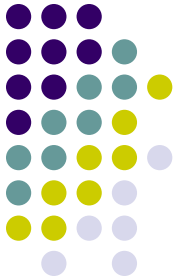
- Book Chapters
  - 12.4, 12.6, 12.7
- Key topics:
  - How to sort when data doesn't fit in memory
  - How to execute other operations like duplicate elimination, group by aggregates, etc.
  - How to put it all together in a query plan
  - Pipelining vs Materialization
  - Iterator Interface

Sorting



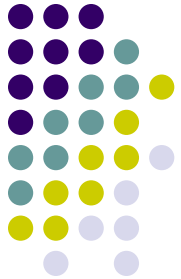
- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough…
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

External sort-merge



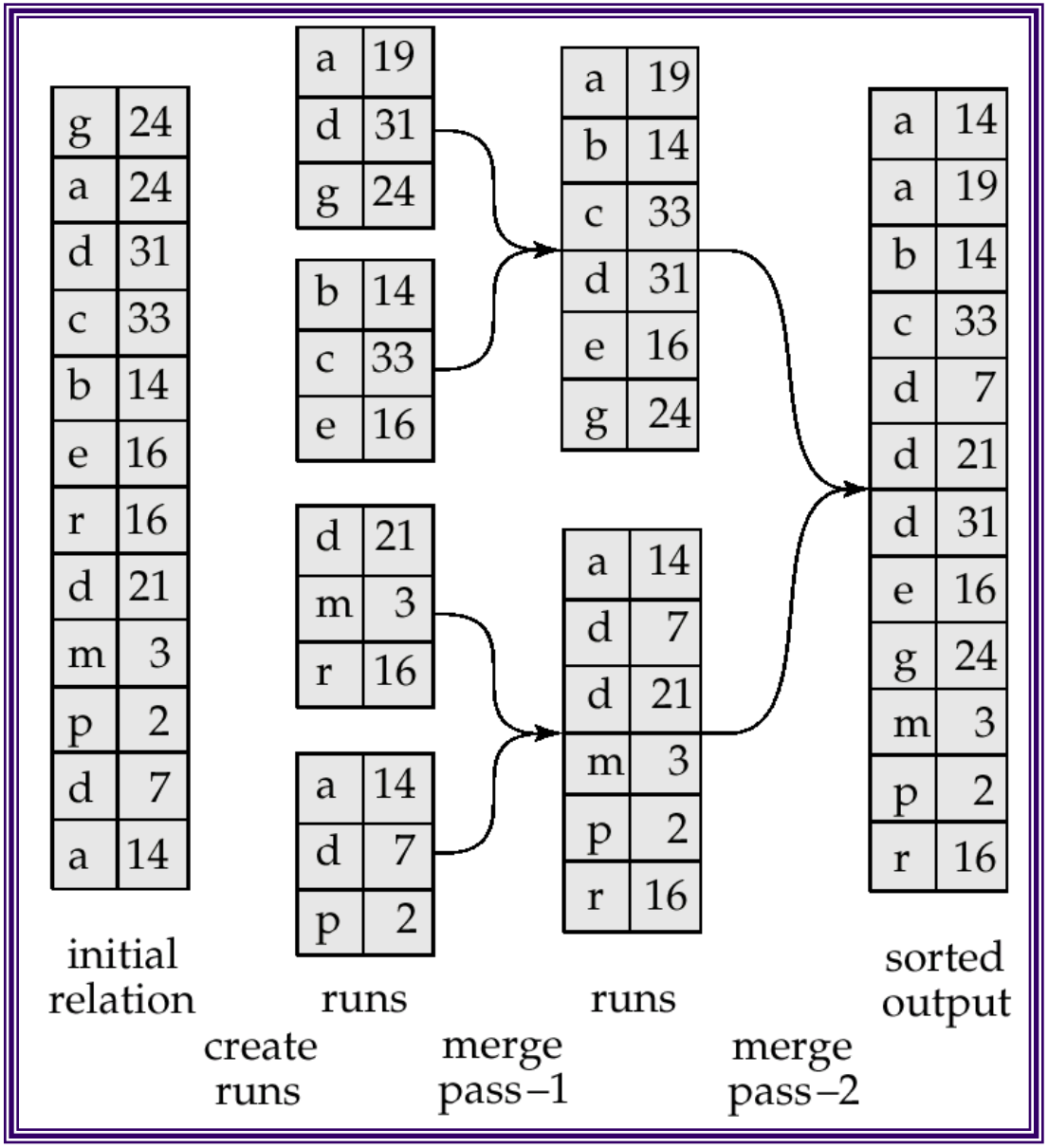
- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this “N” times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*

External sort-merge

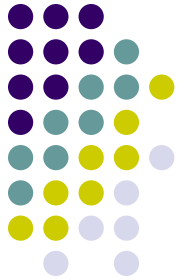


- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M ?*
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000 blocks = 4MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

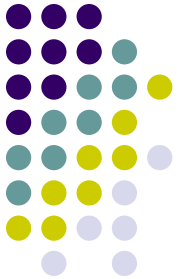


- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk accesses for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

  Thus total number of disk accesses for external sorting:

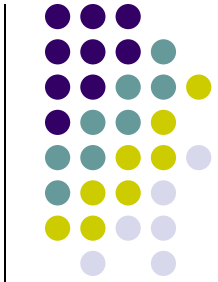
  $$b_r (2\lceil \log_{M-1}(b_r / M) \rceil + 1)$$

Query Processing



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

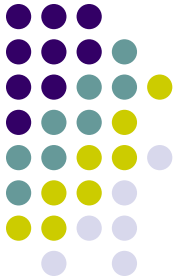
Group By and Aggregation



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

- Hash-based algorithm
- Steps:
  - Create a hash table on *a,* and keep the *count(b) 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

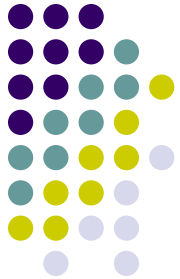
Group By and Aggregation



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

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

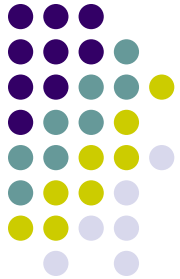
Group By and Aggregation



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

- 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(): can do efficiently with 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 ;

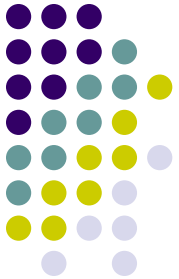
- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev = r*
    - else
      - Skip *r*

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*

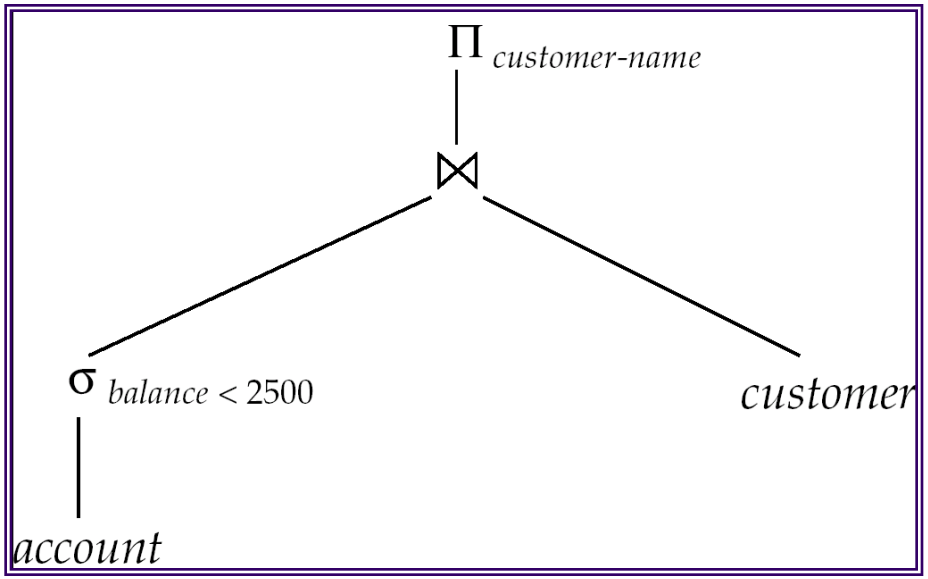
Query Processing



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

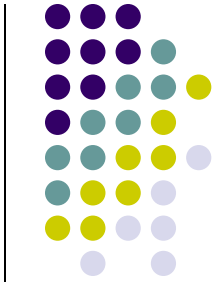
Evaluation of Expressions

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



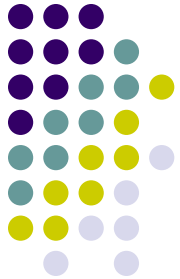
- Two options:
  - Materialization
  - Pipelining

Evaluation of Expressions



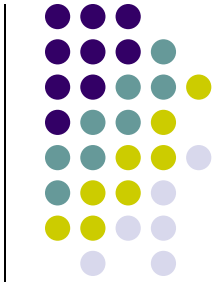
- Materialization
  - Evaluate each expression separately
    - Store its result on disk in *temporary relations*
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always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

Materialization



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing 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

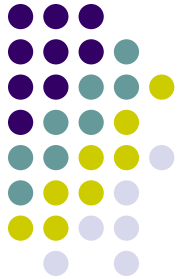


- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

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

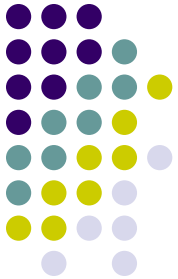
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited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

Pipelining



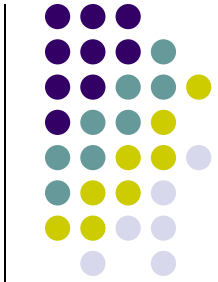
- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no. Need to wait for the entire input before producing output
    - However, there are tricks you can play here
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination

Pipelining: Demand-driven



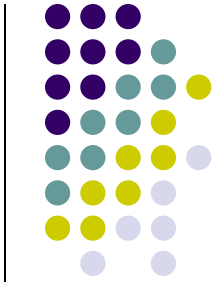
- Iterator Interface
  - Each operator implements:
    - *init(): Initialize the state (sometimes called open())*
    - *get_next(): get the next tuple from the operator*
    - *close(): Finish and clean up*
  - Sequential Scan:
    - *init():* open the file
    - *get_next():* get the next tuple from file
    - *close():* close the file
- Execute by repeatadly 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()

Hash-Join Iterator Interface



- open():
  - Call open() on the left and the right children
  - Decide if partitioning is needed (if size of smaller relation > allotted memory)
  - Create a hash table
- get_next(): (((assuming no partitioning needed)))
  - 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

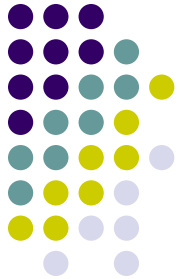
Hash-Join Iterator Interface



- close():
  - Call close() on the left and the right children
  - Delete the hash table, other intermediate state etc…
- get_next(): (((partitioning needed)))
  - 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

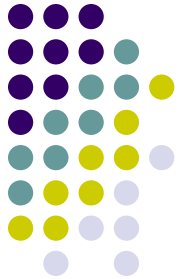
- Take a look at the postgreSQL codebase

Pipelining (Cont.)



- In produce-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 schedules operations that have space in output buffer and can process more input tuples

Recap: Query Processing



- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handles TBs (even PBs) of data
- However, consider how complex SQL is and how much you can do with it
  - Compared to that, this isn't much
- Most of it is very nicely modular
  - Especially through use of the *iterator()* interface
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase very easy to read and modify
- Having so many operators does complicate the codebase and the query optimizer though
  - But needed for performance