Advanced DB

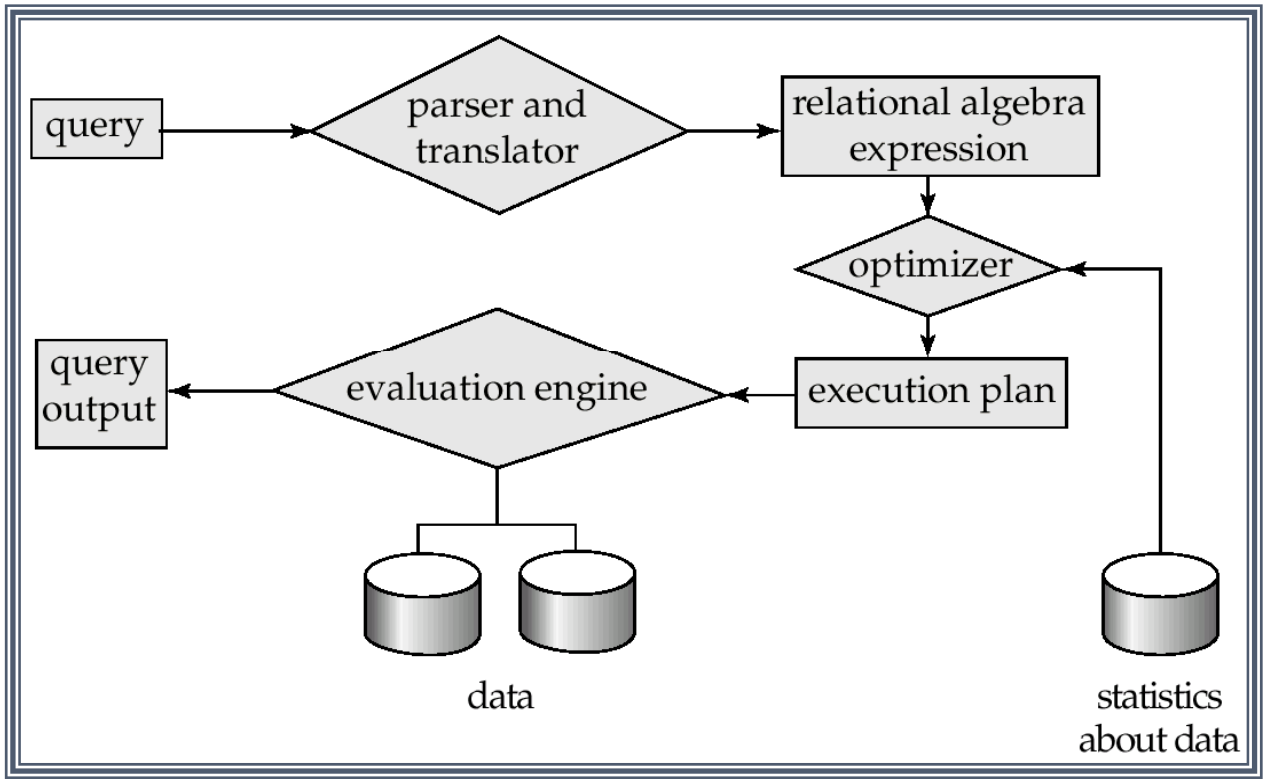
CHAPTER 13
QUERY PROCESSING

Chapter 13: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps

- Parsing and translation
  - translate the query into an internal form (eg., relational algebra)
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
- More than one way to evaluate a query
  - equivalence of expressions:

    SELECT *balance* FROM *account* WHERE *balance*>2500

    $\Pi_{balance}(\sigma_{balance>2500}(account))$ OR $\sigma_{balance>2500}(\Pi_{balance}(account))$

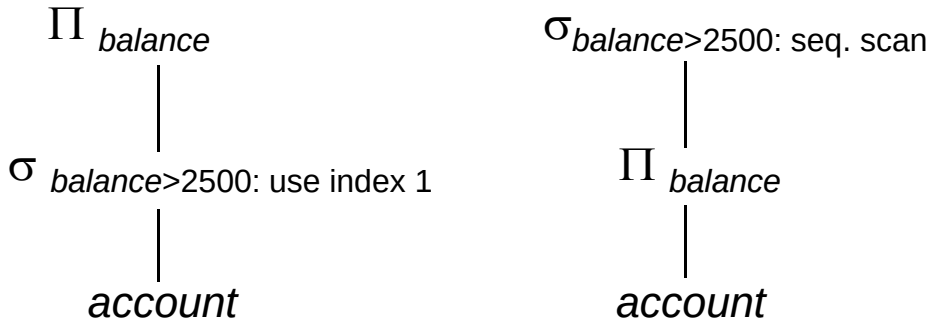
  - Different strategies and algorithms

    $\sigma_{balance>2500}(account)$: *sequential scan* OR

    index scan on balance

Query Plan

- Evaluation primitive
  - a relational algebra expression annotated with instructions on how to evaluate it
    - $\sigma_{balance>2500}(account)$: use index 1
    - $\sigma_{balance>2500}(account)$: use table scan
- Query evaluation (execution) plan
  - a sequence of primitive operations that can be used to evaluate a query
- Example:
  - SELECT *balance* FROM *account* WHERE *balance*>2500



Query Optimization

- Equivalence of Expressions

  Given a DB schema *S*, a query *Q* on *S* is *equivalent* to another query *Q´* on *S*, if the answer sets of *Q* and *Q´* are the same in *any* instances of the DB.

  $\Pi_{b\_name,\ asset}(\sigma_{c\_city=\text{“PC”}}(customer \bowtie depositor \bowtie branch))$ vs

  $\Pi_{b\_name,\ asset}((\sigma_{c\_city=\text{“PC”}}(customer)) \bowtie depositor \bowtie branch)$

- Query optimization is the process of *selecting* the *most efficient query evaluation plan* for a given query
  - Most efficient? How do we know?

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- In a centralized system, disk access is the predominant cost
  - average-seek-cost, average-block-read-cost, average-block-write-cost
- For simplicity we use *number of block transfers from/to disk* as the cost measure
  - ignore the difference in cost between sequential and random I/O
  - Ignore the difference in cost between disk reads and disk writes
  - ignore CPU costs
  - ignore writing final output to disk
- Real systems take all of these measures into account

Selection Operation

- File scan
  - locate and retrieve records that fulfill a selection condition
  - Full file scan: retrieve all records of a file (relation)

- A1: *linear search*
  - Scan each file block and test all records on the selection condition
  - Cost estimate (number of disk blocks scanned) = b_r
    - # of blocks containing records from relation *r*
  - If selection is on a key attribute, cost = (b_r /2)
    - stop on finding record
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Selection Operation (Cont.)

- A2: *binary search*
  - Applicable if selection is an equality comparison on the attribute on which file is ordered
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - $\lceil \log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
    - *Plus* number of blocks containing records that satisfy selection condition

- A3: *primary index on candidate key, equality*
  - Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = $HT_i + 1$

Selection Operation (Cont.)

- A4: *primary index on nonkey, equality*
  - Retrieve multiple records
  - Records will be on consecutive blocks
  - *Cost* = HT_i + number of blocks containing retrieved records

- A5: *equality on search-key of secondary index*
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = HT_i + 1
  - Retrieve multiple records if search-key is not a candidate key
    - *Cost* = HT_i + *number of records retrieved*
      - Can be very expensive!
    - each record may be on a different block
      - one block access for each retrieved record

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:

- A6: *primary index, comparison*
  - (Relation is sorted on A)
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

- A7: *secondary index, comparison*
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case,
    - Retrieving records that are pointed to requires an I/O for each record
    - Linear file scan may be cheaper if many records are to be fetched!

Sorting

- Main memory based algorithms
  - Bubblesort, quicksort, etc.
  - Rely on frequent exchange of elements
  - Not good for relations that don't fit in memory
- Use index on sort key
  - If one exists, simply access records in order
  - Even if one does not exist, we may build one, and then use the index to read the relation in sorted order
    - Index building cost + index sequential access cost
  - May be quite effective
- In general external sort-merge is a good choice

External Sort-Merge

- Let *M* be the memory (buffer) size in blocks

1. Create *sorted runs*.

   Let *i* = 1
   repeat until all blocks of relation have been read:
   (a) Read next *M* blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment *i.*
   Let the total number of runs be *N*

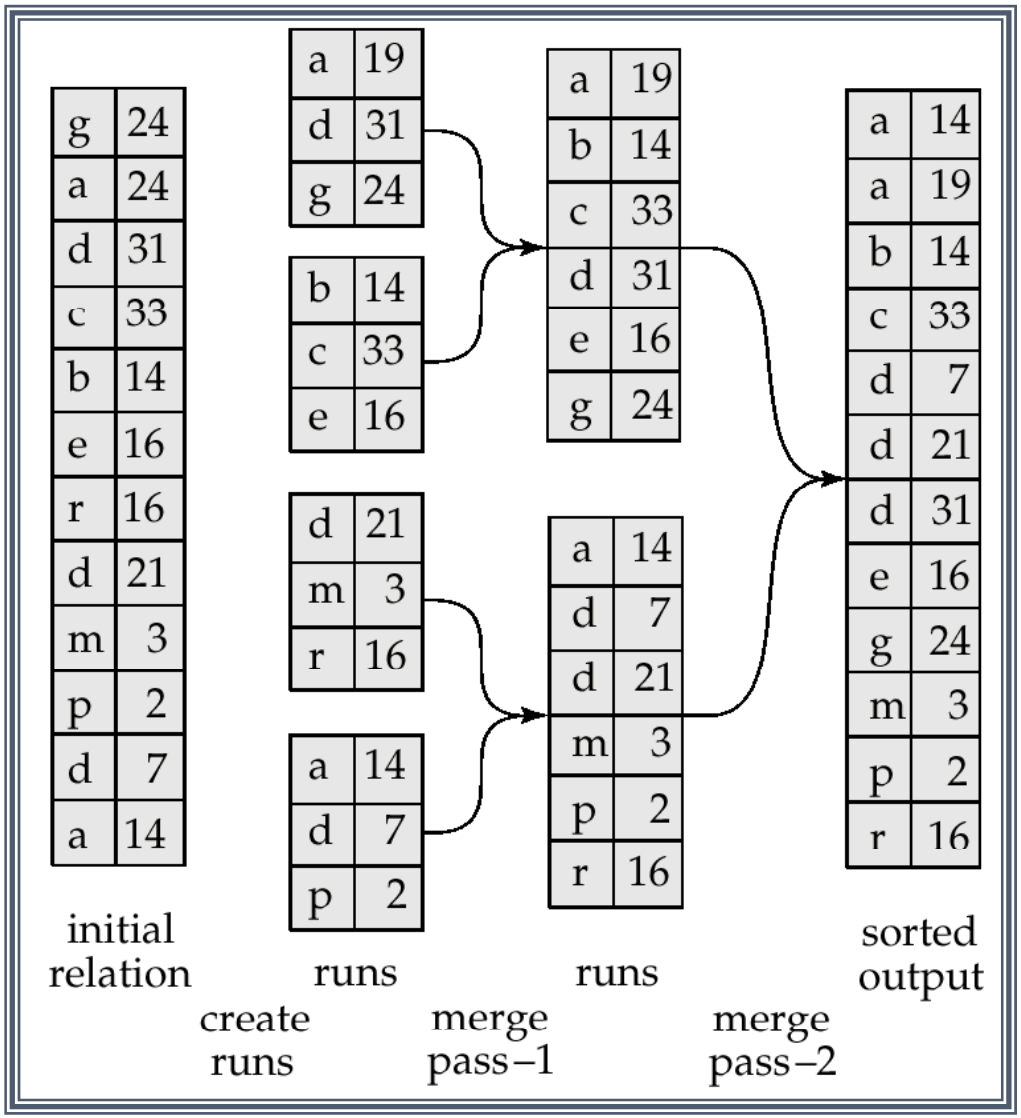
2. Merge the runs (*N*-way merge). (assume for now that *N* < *M*)
   1. Use *N* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.
      3. Delete the record from its input buffer page.
         If the buffer page becomes empty **then**
         read the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9
  - Repeated passes are performed till all runs have been merged into one.

External Sort-Merge - Example

M=3



External Sort-Merge (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:

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

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

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Running Example
  - Number of records of *customer*: 10,000 *depositor*: 5000
  - Number of blocks of *customer*: 400 *depositor*: 100

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  for each tuple t_r in r do
    for each tuple t_s in s do
      if pair (t_r, t_s) satisfy the join condition θ
        add $t_r \bullet t_s$ to the result.

  - r is called the outer relation
  - s the inner relation
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- Worst case
  - there is memory only to hold one block of each relation

    $$n_r * b_s + b_r$$

    disk accesses.
- Best case
  - the smaller relation fits entirely in memory: use it as the inner relation

    $$b_r + b_s$$
- Example
  - Worst case
    - 5,000 * 400 + 100 = 2,000,100 (*depositor* as outer relation)
    - 1,000 * 100 + 400 = 1,000,400 (*customer* as the outer relation)
  - If smaller relation (*depositor)* fits entirely in memory
    - 100 + 400 = 500

Block Nested-Loop Join

- Variant of nested-loop join
  - every block of inner relation is paired with every block of outer relation

    for each block B_r of r do
    for each block B_s of s do
    for each tuple t_r in B_r do
    for each tuple t_s in B_s do
    if (t_r, t_s) satisfy the join condition
    then add $t_r \bullet t_s$ to the result
- Worst case estimate
  - Each block in the inner relation s is read once for each *block* in the outer relation
  - $b_r * b_s + b_r$

Block Nested-Loop Join (Cont.)

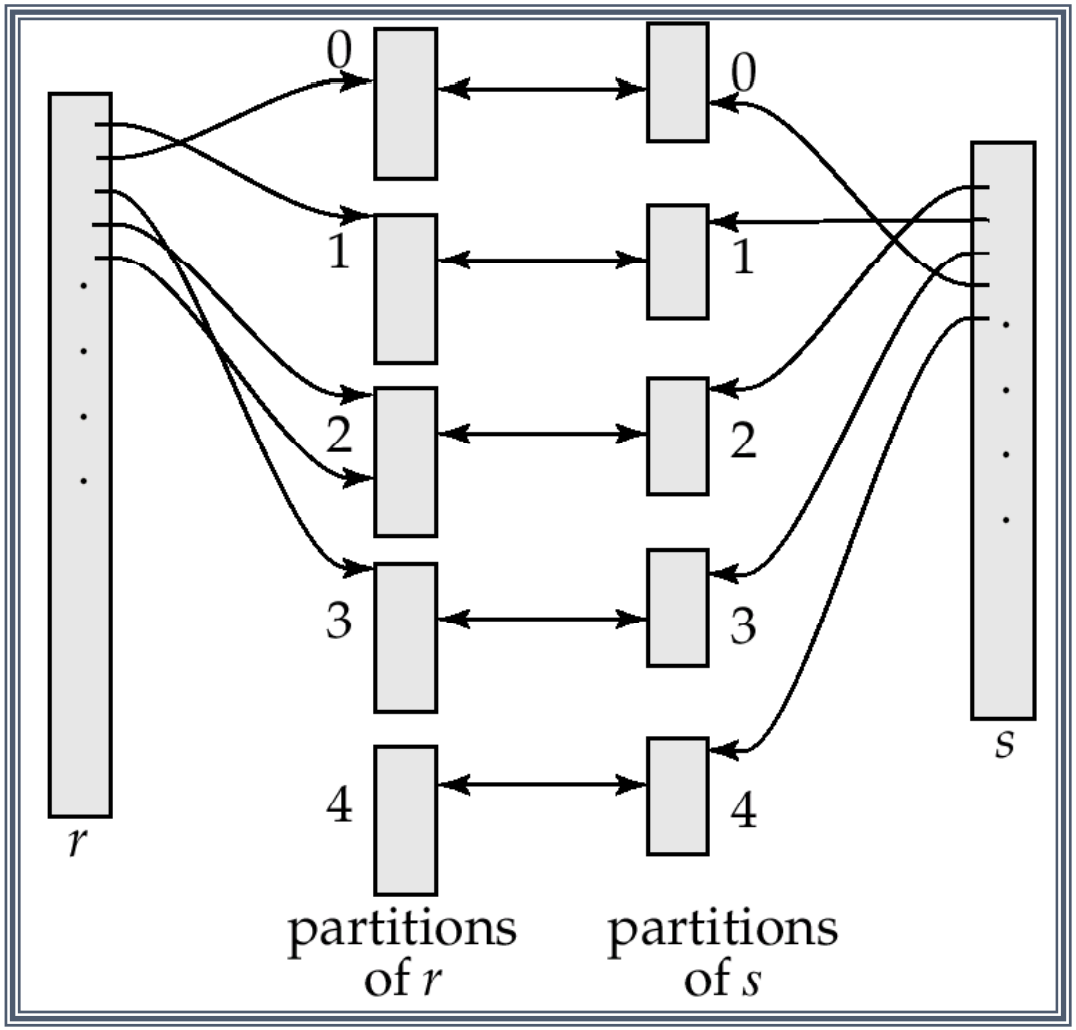
- Best case
  - $b_r + b_s$ block accesses.
- Improvements
  - Use M-2 disk blocks as blocking unit for outer relations, and remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$
  - If equi-join attribute forms a key of inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$: partitions of *r*
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_n$: partitions of *s*
- *r* tuples in r_i need to be compared with *s* tuples in s_i only

(*Note:* In the textbook, r_i is denoted as H_{ri}, s_i is denoted as H_{si}, and *n* is denoted as n_h.)

Hash-Join - Example



Hash-Join Algorithm

1. Partition the relation *s* using hashing function *h*.
   - When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory
      - build an in-memory hash index on it using the join attribute (using a different hash function than the earlier one *h*)
   (b) Read the tuples in r_i from the disk one by one
      - For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

- Relation *s* is called the build input and

  r is called the probe input

Hash-Join algorithm (Cont.)

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory.
- *Hash-table overflow* occurs in partition s_i if s_i does not fit in memory.
  - Many tuples in *s* with same value for join attributes or bad hash function
  - Overflow resolution can be done in build phase
    - Partition s_i is further partitioned using different hash function.
    - Partition r_i must be similarly partitioned.
  - Fails with large numbers of duplicates
    - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- Cost of hash join (without recursive partitioning)

  $3(b_r + b_s) + 4n$ ($4n$ can be ignored)

  - If the entire build input can be kept in main memory, n can be set to 0 and the algorithm does not partition the relations into temporary files. Cost estimate goes down to $b_r + b_s$.
- Example:
  - *customer* ⋈ *depositor*
  - memory size: 20 blocks; $b_{depositor}$= 100 and $b_{customer}$ = 400.
  - *depositor* is build input
  - Partition it into 6 partitions, each of size less than 20 blocks
  - Similarly, partition *customer* into 6 partitions each of size about 70
  - Therefore total cost:

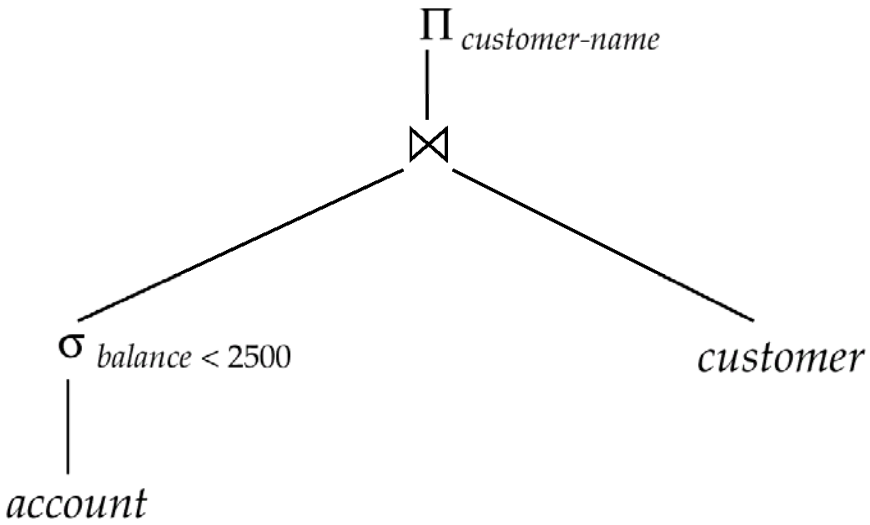
    3(100 + 400) + 4*6 = 1,524 block transfers
  - ignores cost of writing partially filled blocks

Evaluation of Expressions

- So far, we have seen algorithms for individual operations
- How do you evaluate an entire expression tree?
- *Materialization*
  - generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
- *Pipelining*
  - pass on tuples to parent operations even as an operation is being executed

Materialization

- Evaluate one operation at a time, starting at the lowest-level.
- Use intermediate results *materialized into temporary relations* to evaluate next-level operations.
- E.g., in figure below
  - compute and store $\sigma_{balance>2500}(account)$
  - then compute and store its join with *customer*
  - and finally compute the projections on *customer-name.*



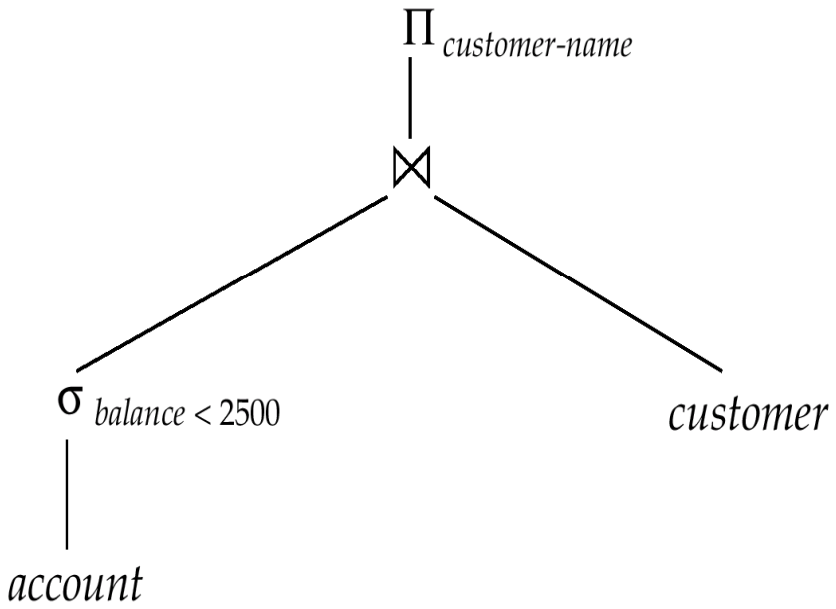
Materialization (Cont.)

- 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
  - *Overall cost* = *Sum* of costs of individual operations + cost of writing intermediate results to disk

Pipelining

- Evaluate several operations simultaneously
  - passing the results of one operation on to the next.

  e.g.: in expression tree
  - don't store result of selection
  - instead, pass tuples directly to the join
  - similarly, don't store result of join but pass tuples directly to projection



Pipelining (cont.)

- Much cheaper than materialization
  - no need to store a temporary relation to disk.
- Pipelining may not always be possible
  - e.g., sort, hash-join: must wait for entire input to materialize
  - Very difficult to achieve a lengthy chain of pipeline
- For pipelining to be effective
  - use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation

END OF CHAPTER 13