Database Management Systems 2010/11

– Chapter 4: Query Processing –

J. Gamper

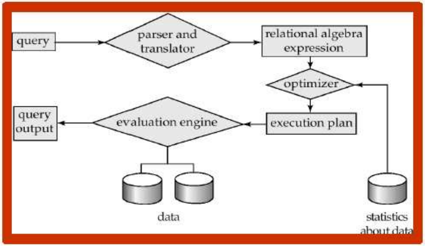
- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Basic Steps in Query Processing

- One of the most important tasks of a DBMS is to figure out an efficient **evaluation plan** (also termed execution plan or access plan) for high level statements.
  - It is particularly important to have evaluation strategies for
    - Selection (search conditions)
    - Joins (combining information in relational database)

- Query processing is a 3-step process:
  1. Parsing and translation
  2. Optimization
  3. Evaluation



Basic Steps in Query Processing ...

- **Step 1: Parsing and translation**
  - Translate the query into its internal form (query tree), which is then translated into **relational algebra (RA)**
  - Parser checks syntax and verifies relations
  - **Example:**
    - `SELECT balance FROM account WHERE balance < 2500`
      might be translated into $\sigma_{balance<2500}\ (\Pi_{balance}(\text{account}))$

Basic Steps in Query Processing ...

- **Step 2: Optimization**
  - An RA-expression may have many equivalent expressions
    - $\sigma_{balance<2500}\ (\Pi_{balance}(\text{account}))$ is equivalent to $\Pi_{balance}\ (\sigma_{balance<2500}(\text{account}))$
  - Each RA-operation can be evaluated using one of several different algorithms.
  - Correspondingly, an RA-expression can be evaluated in many ways.

  - **Evaluation plan:** Annotated RA-expression that specifies for each operator detailed instructions on how to evaluate it.
    - e.g., can use an index on balance to find accounts with balance < 2500
    - or can perform complete relation scan and discard accounts with balance ≥ 2500

$\Pi_{balance}$
|
$\sigma_{balance<2500}$; use index 1
|
account

  - **Goal of query optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
    - Cost is estimated using statistical information from the database catalog, e.g., number of tuples in each relation, size of tuples, etc.

Basic Steps in Query Processing . . .

- **Step 3: Evaluation**
  - The query-execution engine takes an evaluation plan, executes that plan, and returns the answers.

Overview

- In this chapter we study
  - How to measure query cost
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 5 we study
  - How to optimize queries, that is, how to find an evaluation plan with lowest estimated cost.

Measures of Query Cost

- **Query cost** is generally measured as the **total elapsed time** for answering a query.
- Many factors contribute to time cost and are considered in real DBMS, including
  - CPU cost and network communication
  - Disk access
    - Difference between sequential and random I/O
  - Buffer Size
    - Having more memory reduces need for disk access
    - Amount of real memory available to buffer depends on other concurrent OS procsesses, and hard to determine ahead of actual execution.
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Measures of Query Cost . . .

- Typically **disk access is the predominant cost**, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block, since data is read back after being written to ensure that the write was successful
- For simplicity ,
  - we just use **number of block transfers from disk** as the cost measure, and
  - we do not include cost to **writing output to disk**

Selection Evaluation Strategies

- Here we study the evaluation of the **selection operator**:
  - `SELECT * FROM r WHERE` θ
  - $\sigma_\theta(r)$
- The strategy/algorithm for the evaluation of the selection operator depends mainly on the
  - type of the selection condition
  - available index structures

Selection Evaluation Strategies . . .

- **File scan**
  - Class of search algorithms that locate and retrieve records that fulfill a selection condition, i.e., $\sigma_\theta(r)$
  - Lowest-level operator to access data

Selection Evaluation Strategies ...

- **A1 – search:** Scan each file block and test all records to see whether they satisfy the selection condition.
  - Expensive, but always applicable (regardless of indexes, ordering, selection condition ($\sigma_\theta(r)$), etc.)
  - Cost estimate (b_r = number of blocks in file):
    - Worst case: $Cost = b_r$
    - Selection is on a key attribute: $Average\ cost = b_r/2$ (stop on finding record)

Selection Evaluation Strategies . . .

- **A2 – Binary search:** Apply binary search to locate records that satisfy selection condition θ.
  - Only applicable if
    - the blocks of a relation are stored contiguously and
    - the selection condition is an equality comparison on the attribute on which the file is ordered, i.e., $\sigma_{A=v}(r)$
  - Cost estimate:
    - $\lceil \log_2 b_r \rceil$ – cost of locating the first tuple by a binary search on the blocks;
    - plus the number of blocks containing records that satisfy θ.

Selection Evaluation Strategies . . .

- **Index scan**
  - Class of search algorithms that **use an index**
  - Selection condition must be on the search-key of the index
  - Assume B^+-tree index and equality conditions, i.e., $\sigma_{A=v}(r)$

Selection Evaluation Strategies . . .

- **Equality queries:** $\sigma_{A=v}(r)$

- **A3 – Primary index + equality on candidate key**
  - Retrieve a single record that satisfies the equality condition
  - $Cost = HT_i + 1$ (i.e., height of B^+-tree index blocks + 1 data block)

- **A4 – Primary index + equality on non-key**
  - Retrieve multiple records, where records are on consecutive blocks
  - $Cost = HT_i + \#blocks\ with\ retrieved\ records$

- **A5 – Secondary index + equality on search-key**
  - 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
    - $Cost = HT_i + \#retrieved\ records + \#buckets\ with\ search\text{-}key\ value$
    - Can be very expensive, since each record may be on a different block
    - Linear file scan may be cheaper if many records are to be fetched!

Selection Evaluation Strategies . . .

- **Range queries:** $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$
  - Can be implemented by using
    - linear file scan (A1)
    - binary search (A2)
    - or using indices (see below)

- **A6 – Primary index on A + comparison condition**
  - $\sigma_{A \geq v}$: Use index to find first tuple with $A \geq v$; then scan relation sequentially
  - $\sigma_{A \leq v}$: Scan relation sequentially till first tuple with $A > v$; do not use index.

- **A7 – Secondary index on A + comparison condition**
  - $\sigma_{A \geq v}$: Use index to find first index entry with $A \geq v$; scan index sequentially from there, to find pointers to records.
  - $\sigma_{A \leq v}$: Scan leaf pages of index finding rec. pointers till first entry with $A > v$
  - Requires in the worst case one I/O for each record; linear file scan may be cheaper if many records are to be fetched!

Selection Evaluation Strategies ...

- **Conjunctive selection:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$

- **A8 – Conjunctive selection using one index**
  - Choose a θ_i and one of the algorithms A3 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
  - Test the other conditions on tuple after fetching it into memory buffer.
  - Cost = cost of selected algorithm from A3 to A7

- **A9 – Conjunctive selection using multiple-key index**
  - Use appropriate composite (multiple-key) index if available.

- **A10 – Conjunctive selection by intersection of identifiers**
  - Requires indexes with record pointers/identifiers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file that are in the intersection
  - If some conditions do not have indexes, apply test in memory.
  - Cost = sum of individual index scans + cost of retrieving records

Selection Evaluation Strategies . . .

- **Disjunctive selection:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$

- **A11 – Disjunctive selection by union of identifiers**
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file.
  - Applicable only if **all** conditions have available indices; otherwise use linear scan.

Selection Evaluation Strategies . . .

- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$ and an index is applicable to θ, find satisfying records using index and fetch from file.

Sorting

- **Sorting** is important for for several reasons:
  - SQL queries can specify that the output be sorted
  - Several relational operations can be implemented efficiently if the input relations are first sorted, e.g. joins
- We may build an index on the relation, and then use the index to read the relation in sorted order.
  - Sorting is only logically and not physically, which might lead to one disk block access for each tuple (can be expensive!)
    - $\Rightarrow$ It may be desirable to order the records physically.
- Relation fits in memory: Use techniques like **quicksort**
- Relation does not fit in main memory: Use external sorting, e.g., **external sort-merge** is a good choice

External Sort-Merge

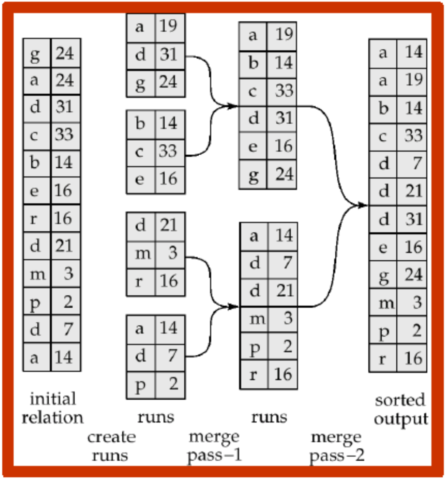
- **Step 1: Create N sorted runs** (M is # blocks in buffer)
  1. $i \leftarrow 0$
  2. Repeatedly do the following till the end of the relation
     - 2.1 Read M blocks of the relation (or the rest) into memory
     - 2.2 Sort the in-memory blocks
     - 2.3 Write sorted data to run file R_i;
     - 2.4 Increment i.
- **Step 2: Merge runs (N-way merge)** (assume $N < M$)
  (Use N blocks in memory to buffer input runs, and 1 block to buffer output)
  1. Read the first block of each run R_i into its buffer page
  2. **Repeat until** all input buffer pages are empty
     - 2.1 Select the first record (in sort order) among all buffer pages
     - 2.2 Write the record to the output buffer; if output buffer is full write it to disk.
     - 2.3 Delete the record from its input buffer page.
     - 2.4 **If** the buffer page becomes empty
       then read the next block (if any) of the run into the buffer

External Sort-Merge ...

- If $N \geq M$, **several merge passes** (step 2) 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, each run being 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

External Sort-Merge . . .

- **Example**: M = 3, 1 block = 1 tuple



External Sort-Merge ...

- **Cost analysis**
  - Initial number of runs: b_r/M
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$
    - The number of runs decreases by a factor of $M-1$ in each merge pass
  - Disk accesses for initial run creation and in each pass is $2b_r$
    - Exception: 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
  - Total number of disk accesses: $Cost = b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$

- **Example:** Cost analysis of previous example
  - 12 (2 * 2 + 1) = 60 disk block transfers

Join Operation

- Several different algorithms for the evaluation of join operation
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- The choice of the algorithm is based on a cost estimate
- Examples use the following relations:
  - Relation customer:
    - Schema: customer = (customer-name, customer-street, customer-city)
    - Number of records: $n_c = 10,000$
    - Number of blocks: $b_c = 400$
  - Relation depositor:
    - Schema: depositor = (customer-name, account-number)
    - Number of records: $n_d = 5,000$
    - Number of blocks: $b_d = 100$

Nested-Loop Join

- Compute the theta join: $r \bowtie_\theta s$

foreach *tuple* t_r *in* r **do**
    foreach *tuple* t_s *in* s **do**
        if *pair* (t_r, t_s) *satisfies* θ **then**
            Add $t_r \circ t_s$ to the result
        end
    end
end

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

Nested-Loop Join ...

- Order of r and s are important: Relation r is read once, relation s is read up to $|r|$ times
  - **Worst case:** Only one block of each relation fits in main memory

$$Cost = n_r * b_s + b_r$$

  - If the smaller relation fits entirely in memory, use that as the inner relation

$$Cost = b_r + b_s$$

- **Example:** Assuming worst case memory availability
  - Depositor as outer relation: 5,000 * 400 + 100 = 2,000,100 block access.
  - Customer as outer relation: 10,000 * 100 + 400 = 1,000,400 block accesses.
  - Smaller relation *(depositor)* fits entirely in memory: 400 + 100 = 500 block accesses.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of the inner relation is paired with every block of the outer relation.

foreach *block* B_r *of* r **do**
    foreach *block* B_s *of* s **do**
        foreach *tuple* t_r *in* B_r **do**
            foreach *tuple* t_s *in* B_s **do**
                if *pair* (t_r, t_s) *satisfies* θ **then**
                    Add $t_r \circ t_s$ to the result
                end
            end
        end
    end
end

Block Nested-Loop Join ...

- Worst case: $Cost = b_r * b_s + b_r$
  - Each block in the inner relation s is read once for each block in the outer relation (instead of once for each tuple in the outer relation)
- Best case: $Cost = b_r + b_s$
- Improvements to nested loop and block nested loop algorithms (M is the number of main memory blocks):
  - Block nested-loop: Use $M - 2$ disk blocks for outer relation and two blocks to buffer inner relation and output; join each block of the inner relation with $M - 2$ blocks of the outer relation.
    - $Cost = \lceil b_r/(M-2) \rceil * b_s + b_r$
  - If equi-join attribute forms a key on 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).

Indexed Nested-Loop Join

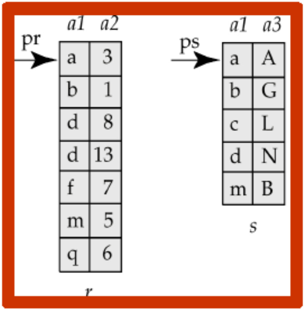
- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - can construct an index just to compute a join
- For each tuple t_r in the outer relation r, use the index to look up the tuples in s that satisfy the join condition with tuple t_r.
- Worst case: Buffer has space for only one page of r, and, for each tuple in r perform an index lookup on s.
  - $Cost = n_r * c + b_r$
    - where c is the cost of traversing index and fetching all matching s tuples for one tuple of r
    - c can be estimated as cost of a single selection on s using the join condition.
- If indexes are available on join attributes of both r and s, use relation with fewer tuples as the outer relation.

Indexed Nested-Loop Join ...

- **Example:** Compute *depositor* $\bowtie$ *customer*, with depositor as the outer relation.
  - Let *customer* have a primary B^+-tree index on the join attribute *customer-name*, which contains 20 entries in each index node.
  - Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
  - *depositor* has 5,000 tuples and 100 blocks
  - Indexed nested loops join:
    - $Cost = 5,000 * 5 + 100 = 25,100$ disk accesses.
  - Block nested loops join:
    - $Cost = 100 * 400 + 100 = 40,100$ disk accesses assuming worst case memory
    - May be significantly less with more memory

Merge-join

- Basic idea of **merge-join (sort-merge join)**: Use two pointers *pr* and *ps* that are initialized to the first tuple in *r* and *s* and move in a synchronized way through the sorted relations.
- Algorithm
  1. Sort both relations on their join attributes (if not already sorted on the join attr.).
  2. Scan *r* and *s* in sort order and return matching tuples.
  3. Move the tuple pointer of the relation that is less far advanced in sort order (more complicated if the join attributes are not unique - every pair with same value on join attribute must be matched).



Merge-join ...

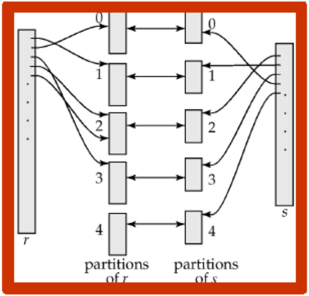
- Applicable for equi-joins and natural joins only
- If all tuples for any given value of the join attributes fit in memory
  - One file scan of r and s is enough
  - $Cost = b_r + b_s$ (+ the cost of sorting if relations are not sorted)
- Otherwise, a block nested-loop join must be performed between the tuples with the same attributes

Merge-join ...

- Variations of merge-join exist
- **Secondary indexes** on join attribute(s) exist for both relations.
  - Scan the records through the indexes.
  - Drawback: Records may be scattered throughout the file blocks
- **Hybrid merge-join:** One relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree.
  - Intermediate result contains tuples of sorted relation and addresses of the tuples of the unsorted relation.
  - Sort the intermediate result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples.

Hash-Join

- Applicable for equi-joins and natural joins only.
- Partition tuples of r and s using the same hash function h, which maps the values of the join attributes to the set $\{0, 1, ..., n\}$
  - Partitions of r-tuples: $r_0, r_1, ..., r_n$
    - all $t_r \in r$ with $h(t_r[JoinAttrs]) = i$ are put in r_i
  - Partitions of s-tuples: $s_0, s_1, .., s_n$
    - all $t_s \in s$ with $h(t_s[JoinAttrs]) = i$ are put in s_i
- r-tuples in r_i need only to be compared with s-tuples in s_i
- an r-tuple and s-tuple that satisfy the join condition have the same hash value i, and are mapped to r_i and s_i, respectively.



Hash-Join ...

- **Algorithm** for the hash-join of r and s
  1. Partition the relation s using hash 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:
     3.1 Load s_i into memory and build an in-memory hash index on it using the join attribute (this hash index uses a different hash function than h).
     3.2 Read the tuples in r_i from the disk (block by block). 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 as result tuple.

- Relation s is called the build input and r is called the probe input.

Hash-Join . . .

- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- Two ways to handle overflow
  - **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.
  - **Overflow avoidance** performs partitioning carefully to avoid overflows
    - Initially, build many small partitions of s, then combine some partitions s.t. they still fit into main memory. Partition r in the same way, but the size of partitions r_i does not matter.
- Both approaches fail with large numbers of duplicates
  - Fallback option: Block nested loops join on overflowed partitions

Hash-Join ...

- **Cost analysis** of hash join
  - Partitioning of the two relations: $2 * (b_r + b_s)$
    - Complete reading of the two relations plus writing back
  - The build and probe phases read each of the partitions once: $b_r + b_s$
  - The number of blocks occupied by the n_h partitions could be slightly more than $b_r + b_s$ due to partially filled blocks (at most one in each partition)
    $\Rightarrow$ Overhead: $2 * n_h$ for each of the two relations
  - Total cost: $Cost = 3 * (b_r + b_s) + 4 * n_h$
    - Overhead $4 * n_h$ is quite small compared to $b_r + b_s$ and can be ignored.

Hash-Join ...

- **Example:** Join *customer* $\bowtie$ *depositor*
  - Assume that memory size is 20 blocks
  - $b_d = 100$ and $b_c = 400$
  - *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
  - Similarly, partition customer into five partitions, each of size 80. This is also done in one pass.
    - Partition size of probe relation needs not to fit into main memory!
  - Therefore total cost: $3 * (100 + 400) = 1500$ block transfers
    - Ignores cost of writing partially filled blocks
    - Compare this to 25,100 block transfers of indexed nested loop join!

Complex Joins

- Join with a **conjunctive** condition: $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$ using a more efficient strategy; final result comprises those tuples in the intermediate result that satisfy the remaining conditions $\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$
- Join with a **disjunctive** condition: $r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$, i.e., $(r \bowtie_{\theta_1} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$

Other Operations

- **Duplicate elimination:** Can be implemented via hashing or sorting
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - Hashing is similar - duplicates will come into the same bucket.
- **Projection:** Implemented by performing projection on each tuple followed by duplicate elimination.
- **Aggregation:** Can be implemented in a manner similar to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together; then the aggregate functions are applied on each group.

Other Operations ...

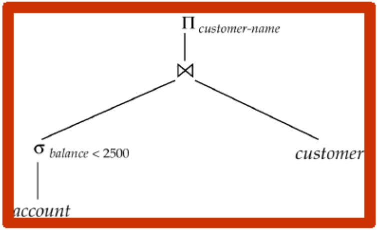
- **Set operations** ($\cup, \cap$ *and* $-$): Can either use variant of merge-join after sorting or variant of hash-join.
- Algorithm for set operations on r and s using hashing:
  1. Partition r and s using the same hash function, thereby creating $r_1, \ldots, r_n$ and $s_1, \ldots, s_n$
  2. Process each partition i as follows:
     - Read r_i and build an in-memory hash index on r_i using a different hash function.
     - Read s_i and do the following:
       - $r \cup s$: Add tuples in s_i to the hash index if they are not already in it. At end of s_i add the tuples in the hash index to the result.
       - $r \cap s$: Output tuples in s_i to the result if they are in the hash index.
       - $r \setminus s$: For each tuple in s_i, if it is in the hash index, delete it from the index. At end of s_i add the remaining tuples in the hash index to the result.

Evaluation of Expressions

- Evaluation of complex expressions that contain multiple operations (represented as expression tree)
  - **Materialization:** Generate results of an (sub-)expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat this process in a bottom up fashion.
  - **Pipelining:** Evaluate several operations simultaneously by passing on tuples to parent operations even as an operation is being executed.

Materialization

- **Materialized evaluation**
  - Evaluate one operation at a time, starting at the lowest-level.
  - Use intermediate results materialized into temporary relations to evaluate next-level operations.
- **Example:** Evaluate the expression in the figure below.
  1. Compute and store $\sigma_{balance<2500}$
  2. Then compute and store its join with *customer*
  3. Finally, compute the projections on *customer-name*.

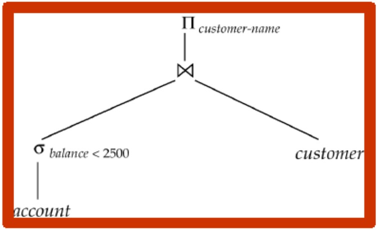


Materialization ...

- Materialized evaluation is always applicable.
- Cost analysis
  - Cost = cost of individual operations +
    cost of writing intermediate results to disk
  - 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
- **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

- **Pipelined evaluation:** Evaluate several operations simultaneously, passing the results of one operation on to the next
- **Example:**
  - Do not store the result of $\sigma_{balance<2500}$
  - Instead, pass tuples directly to the join.
  - Similarly, don't store result of join, pass tuples directly to projection



Pipelining ...

- Much cheaper than materialization: No need to store a temporary relation to disk.
- Pipelining may not always be possible
  - Some evaluation algorithms are not able to generate result tuples even as they get input tuples, e.g., merge join, or hash join
  - They produce intermediate results being written to disk and then read back always
  - Algorithm variants are possible to generate (at least some) results on the fly, as input tuples are read in.
- Two types of pipelining
  - Demand driven
  - Producer driven

Pipelining . . .

- **Demand driven** (or **lazy**) evaluation
  - System repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain the "**state**" to know what to return next
  - Each operation is implemented as an **iterator** with the following operations
    - **open()**:
      e.g., for file scan: initialize file scan, store pointer to beginning of file as state
      e.g., for merge join: sort relations and store pointers to beginning of sorted relations as state
    - **next()**:
      e.g., for file scan: Output next tuple, and advance and store file pointer
      E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**

Pipelining ...

- **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 schedules operations that have space in output buffer and can process more input tuples.