COMP 3311 DATABASE MANAGEMENT SYSTEMS

LECTURE 17 QUERY PROCESSING: EXPRESSION EVALUATION

PROJECTION OPERATION: USING SORTING

```
select distinct boatId
from Reserves;
```

Reserves(sailorId, boatId, rDate)

- For duplicate elimination we can use a modified external sorting approach.

Modify Pass 0 of external sorting to eliminate unwanted attributes.

- Thus, sorted runs contain smaller tuples. (The size reduction depends on the number and the size of the attributes that are eliminated.)

Modify merge passes to eliminate duplicates.

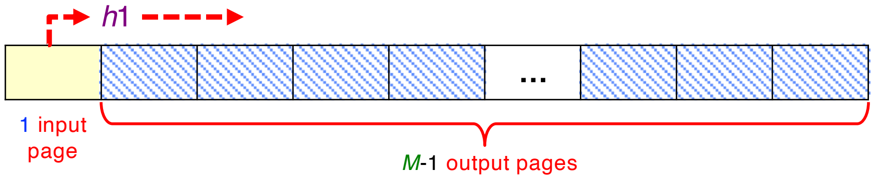
- Thus, the number of result tuples is smaller than the input. (The difference depends on the number of duplicates.)

Cost: Pass 0: the original relation is read and the same number of smaller tuples are written out ⟹ fewer write page I/Os than read I/Os.

Merge passes: smaller tuples are read and fewer tuples are written out in each pass ⟹ fewer write page I/Os than read I/Os.

PROJECTION OPERATION: USING HASHING

Partitioning Phase



- Read *r* using one input page.
- For each tuple, discard unwanted attributes and apply hash function h_1 to the remaining attributes to hash to one of *M*-1 output pages where *M* is the available main memory pages.
- The result is *M*-1 partitions (of tuples with no unwanted attributes) in which two tuples from different partitions are guaranteed to be distinct.

Duplicate Elimination Phase

- Read each partition and build an in-memory hash table using hash function h_2 ($\neq h_1$) on all attributes; discard duplicates in the pages.

Cost:

For partitioning: Read *r* and write out each tuple, but with fewer attributes ⟹ fewer write page I/Os than read I/Os.

For duplicate elimination: Read the result of the partitioning phase.

PROJECTION OPERATION: DISCUSSION

- The sort-based approach is the standard since it handles skewed data better *and* the result is sorted.

- If an index on the relation contains all wanted attributes in its search key, we can do an *index-only* scan.
  - Apply projection on the index data entries, which are much smaller.

- If an ordered (i.e., tree) index contains all wanted attributes as the *prefix* of the search key, we can do even better.
  - Retrieve data entries in order using an index-only scan, discard unwanted attributes and compare the adjacent entries to check for duplicates.

☞ **For an index-only scan, the index must be dense. Why?**

SET OPERATIONS: USING SORTING

1. Sort both relation *r* and *s* on the same attribute eliminating duplicates.

2. The merge phase depends on the operation.

Intersection (*r* ∩ *s*):	Output a tuple *only if* it is in *both* relations.
Union (*r* ∪ *s*):	Output a tuple *only once* if it belongs to both relations.
Set difference (*r* – *s*):	Output a tuple *only if* it is in the first relation *but not* in the second one.

SET OPERATIONS: USING HASHING

1. Partition relation *r* and *s* using hash function h_1 on all attributes.
   - *s* is the build input.

2. For each *s*-partition,
   - build an in-memory hash table (using h_2) on all attributes while discarding duplicates.
   - scan the corresponding *r*-partition (page-by-page) and, for each tuple t_r of *r*, probe the *s*-partition and do the following.

Intersection ($r \cap s$): Output t_r *only if* it is also in *s.*

Union ($r \cup s$): Output t_r if it *is not* in *s*; also output all tuples of *s.*

Set difference ($r - s$): Output t_r *only if it is not* in *s*.
This computes $r - s$. How to compute $s - r$?

If t_r is in *s*, then remove all matching t_s tuples from the *s*-partition. Output all remaining *s* tuples.

AGGREGATE OPERATIONS

Without Grouping

- In general, requires scanning the relation.
- If we have a tree index whose search key includes *all attributes* in the select or where clauses, we can do an index-only scan (e.g., “Find the average age of all sailors” given a dense index on age).

With Grouping

- Use the duplicate elimination techniques (i.e., sorting or hashing), but instead of eliminating duplicates, *gather them into groups and apply aggregate operations on each group*.

  ☞ **Cost is the same as for duplicate elimination.**

  ☞ **Do not need to actually form groups—can form groups *and* compute aggregate for each group on-the-fly.**

- If we have a tree index whose search key includes *all attributes* in select, where and group by clauses, we can do an index-only scan.
- If the group-by attributes form a prefix of the search key, we can retrieve index data entries/tuples in group-by order.

RELATIONAL ALGEBRA TREE

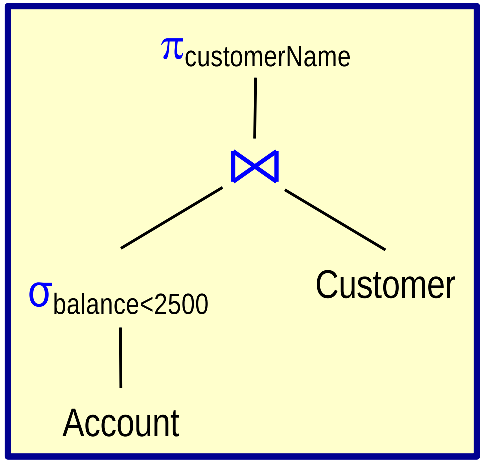
- Recall that a relational algebra (operator) tree represents a relational algebra expression (an SQL query) as a tree.
- It is a graphical way of representing the evaluation order of a relational algebra expression. (No need for parenthesis!)
- Evaluation is bottom up using materialization or pipelining.

☞ **Can annotate the tree to indicate access methods to use for each relation and implementation method to use for each relational operator.**

SQL query

```
select customerName
from Customer natural join Account
where balance<2500;
```

Relational algebra tree



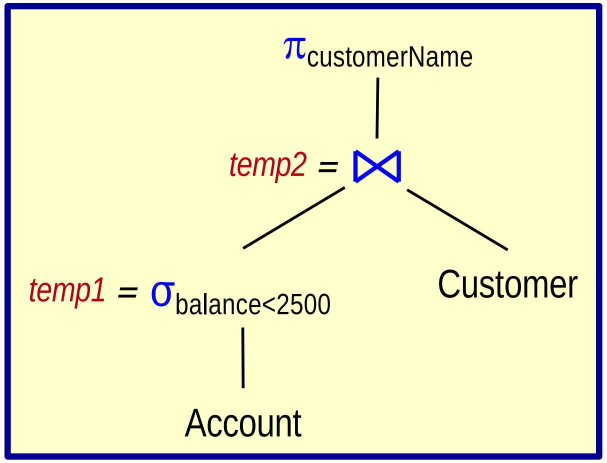
Relational algebra expression

$\pi_{customerName}(\sigma_{balance<2500}(\text{Customer JOIN Account})$

EVALUATION USING MATERIALIZATION

- Evaluate one operation at a time, store intermediate result into a temporary relation and use it to evaluate the next operation.

- For the query in the figure:
  1. compute and store the selection $\sigma_{balance<2500}$(Account) into *temp1*.
  2. compute and store the join of *temp1* with Customer in *temp2*.
  3. compute the projection of *temp2* on customerName.



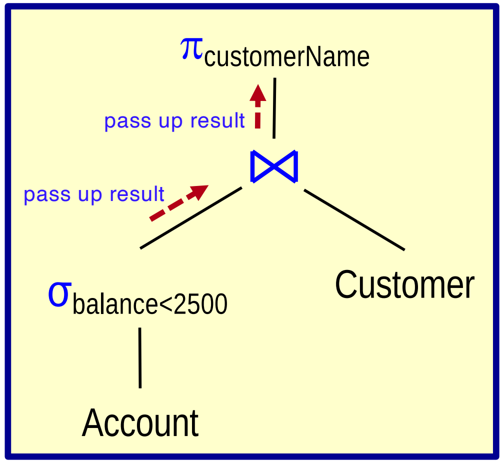
- Materialized evaluation is always applicable, but the cost of writing and reading intermediate results can be quite high.

Overall cost: sum of costs of individual operations + cost of writing intermediate results to disk

EVALUATION USING PIPELINING

- Evaluate several operations simultaneously, passing the results to the next operation (lazy or eager).

- For the query in the figure:
  - compute the selection $\sigma_{balance<2500}$(Account), but do not store the result; instead, pass the tuples directly to the join operation.
  - Similarly, compute the join, but do not store the result; instead, pass the tuples directly to the projection operation.



- Pipelining is much cheaper than materialization since there is no need to store temporary relations to disk.

- Need evaluation algorithms that generate output tuples even as tuples are received for input to the operation.

EVALUATION USING PIPELINING (CONT'D)

Demand-driven or Lazy

- The system repeatedly requests the next tuple from the top level operation.
- Each operation requests the next tuple from its pipelined inputs as required in order to output its next tuple.
- In between calls, the operation has to maintain "state" so it knows what to return next.

Producer-driven or Eager

- Operations produce tuples continuously and pass them into their pipeline output.
  - A buffer is needed between operations – one operation puts tuples into the buffer, while another operation removes tuples from the buffer.
  - If the buffer is full, the operation waits until there is space in the buffer before generating more tuples.

EVALUATION USING PIPELINING (CONT'D)

- Algorithms that are not able to output tuples as they get input tuples are called blocking.
  - ☞ Sort-merge join or hash join are blocking since they require intermediate results to be written to disk and then read back.

- There are algorithm variants that generate (at least some) results on-the-fly as input tuples are read.
  - ☞ Hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read.

Pipelined Join: Hybrid hash join can be modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and outputting matching result tuples between partition 0 tuples.

- ➢ When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0.
- ➢ This can also be done symmetrically for s_0 tuples.

QUERY PROCESSING: SUMMARY

Query Processing

1. Transform the SQL query into a relational algebra expression.
2. Generate different query execution plans and evaluate their cost.
3. Select an optimal query execution plan.

Operation Processing

Selection: linear scan; index lookup.

Sorting: external sort-merge if too large to fit into memory

Join: block nested-loop; indexed nested-loop; merge join; hash join

Projection, duplicate elimination, set operations: sorting; hashing

Aggregate operations: linear scan; index-only scan; sorting; hashing

Expression evaluation: materialization; pipelining.