L10: Query Processing – Other Operations, Pipelining and Materialization

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
• **Join with a conjunctive condition:**

\[ r \ JOIN_{\theta_1 \land \theta_2 \land \ldots \land \theta_n} s \]

1. Either use block nested loops, or
2. Compute the result of one of the simpler joins \( r \ JOIN_{\theta_i} s \)
   - final result comprises those tuples in the intermediate result that satisfy the remaining conditions
   \[ \theta_1 \land \ldots \land \theta_{i-1} \land \theta_{i+1} \land \ldots \land \theta_n \]

• **Join with a disjunctive condition**

\[ r \ JOIN_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n} s \]

1. Either use block nested loops, or
2. Compute as the union of the records in individual joins:
   \[ (r \ JOIN_{\theta_1} s) \cup (r \ JOIN_{\theta_2} s) \cup \ldots \cup (r \ JOIN_{\theta_n} s) \]
   - useful only if all conditions are restrictive (selective)
Consider the following projection query

```sql
SELECT DISTINCT R.bid
FROM Reserves R
```

An approach based on **external sorting** for duplicate elimination:

- Modify Pass 0 of external sort to eliminate unwanted fields.
  - Thus, sorted runs contain smaller records
  - Size ratio depends on # and size of fields that are dropped.

- Modify merging passes to eliminate duplicates
  - Thus, number of result tuples smaller than input
  - Difference depends on # of duplicates.

- **Cost:** In Pass 0, read original relation, write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass.
Partitioning phase: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h_1$ to choose one of $M-1$ output buffers ($M$ is the number available main memory pages)
- Result is $M-1$ partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.

Duplicate elimination phase: For each partition, read it and build an in-memory hash table, using hash function $h_2$ ($<> h_1$) on all fields, while discarding duplicates.

Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.
• Sort-based approach is the standard; better handling of skew, and result is sorted

• If an index on the relation contains all wanted attributes in its search key, can do index-only scan
  - Apply projection techniques to data entries (much smaller!)

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

- Set operations can be handled by join algorithms. They should also remove duplicates.

**Sorting based approach:**
- Sort both relations (on the same attribute)
- The merging phase depends on the operation
  - **Intersection**: report a tuple only if it belongs to both files
  - **Union**: report all tuples removing duplicates
  - **Set difference**: report the tuples that belong to the first file but not the second one

**Hash based approach:**
- Partition files \( r \) and \( s \) using hash function \( h \) (on all attributes). \( s \) is the build input
- For each \( s \)-partition, build in-memory hash table (using \( h_2 \)), scan corresponding \( r \)-partition (page-by-page) and for each tuple \( t \) of \( R \)
  - **Intersection**: report \( t \) only if it also belongs to \( s \)
  - **Union**: report \( t \) if it does not belong to \( s \). At the end report all tuples of \( s \)
  - **Set difference**: report \( t \) only if it does not belong to \( s \) (computes \( R-S \); how do we compute \( S-R \)?)
Without grouping:
- In general, they require scanning the relation
- Given an index whose search key includes all attributes in the SELECT or WHERE clauses, we can do index-only scan (e.g., “find the average age of all sailors given an index on age”)

With grouping (assuming that grouping attribute values do not fit in memory):
- Sort on group-by attributes, then scan relation and compute aggregate for each group. (E.g., “compute the average rating for each age value”; what about “compute the average age for each rating value”?)
- Similar approach based on hashing on group-by attributes
- Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order
• So far we have seen algorithms for individual operations

• Alternatives for evaluating an entire expression tree
  - **Materialization**: Materialize (i.e., store into temporary relations in the disk) intermediate results from lower-level operations, and use them as inputs to upper-level operations
  - **Pipelining**: Pass on tuples to parent operations even as an operation is being executed
**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.

In figure below, compute and store

\[ \sigma_{balance < 2500}(account) \]

then compute the store its join with *customer*, and finally compute the projections on *customer-name*.
• Materialized evaluation is always applicable
• Cost of writing intermediate results to disk and reading them back can be quite high
  – **Overall cost**: Sum of costs of individual operations + cost of writing intermediate results to disk
  • Our cost formulas for operations ignore cost of writing final results to disk
• **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next

• E.g., in the previous expression tree, do *not* store the result of

\[ \sigma_{balance<2500}(account) \]

  - instead, pass tuples directly to the join. Similarly, do not store the result of join, but pass tuples directly to projection

• Much cheaper than materialization: no need to store a temporary relation to disk

• For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation

• Pipelines can be executed in two ways:
  - *demand driven*
  - *producer driven*
• In **demand driven** or **lazy** evaluation
  - The system repeatedly requests the next tuple from top level operation
  - Each operation requests next tuple from the children operations as required, in order to output its next tuple
  - In between calls, each operation has to maintain “**state**” so that it knows what to return next
• 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
Some algorithms are not able to output results even as they get input tuples. They are called **blocking**.

- E.g., merge-join, or hash-join
- These lead to **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

- E.g., hybrid hash-join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read

  - **Pipelined join technique**: Hybrid hash-join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches 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 $r_0$
    - Symmetrically for $s_0$ tuples
Consider the relations \( R_1(A,B,C) \), \( R_2(C,D,E) \), and \( R_3(E,F) \). Primary keys are underlined and foreign keys in italics. Foreign keys are not NULL.

Assume that:
- \( R_1 \) has 1000 tuples
- \( R_2 \) has 10000 tuples
- \( R_3 \) has 100000 tuples

What is the best way to join \( R_1, R_2, \) and \( R_3 \)?

\((R_1 \; \text{JOIN}_C \; R_2) \; \text{JOIN}_E \; R_3\) or \( R_1 \; \text{JOIN}_C \; (R_2 \; \text{JOIN}_E \; R_3)\)

What is the size (number of records) in intermediate result \( \text{JOIN}_C \) \( R_1 \; \text{JOIN}_C \; R_2 \)?

What is the size (number of records) in intermediate result \( \text{JOIN}_E \) \( R_2 \; \text{JOIN}_E \; R_3 \)?

What is the size (number of records) in the final result?

Cost difference between a good and a bad way of evaluating a query can be enormous.

How can the optimizer can choose the best evaluation plan for processing the query?

Different plans for a given query involve
- Different but equivalent algebra expressions
- Different algorithms for each operation
Given a query, the optimizer will first generate an algebra expression (tree)

```sql
SELECT sname
FROM Sailor S, Reserves R, Boats B
```

**Expression 1**

```
π S.sname
| σ R.date=1.1.2005 AND B.color=red
+-------------------------+-------------------------+-------------------------+
| JOIN                    | JOIN                    | JOIN                    |
| Boat B                 | Boat B                 | Boat B                 |
| Sailor S               | Sailor S               | Sailor S               |
| Reserves R             | Reserves R             | Reserves R             |
| sname S                | sname S                | sname S                |
```

**Expression 2**

```
π S.sname
| σ R.date=1.1.2005 AND B.color=red
+-------------------------+-------------------------+-------------------------+
| JOIN                    | JOIN                    |
| Boat B                 | Boat B                 |
| Sailor S               | Sailor S               |
| Reserves R             | Reserves R             |
| sname S                | sname S                |
```

**Expression 3**

```
π S.sname
| σ R.date=1.1.2005
+-------------------------+
| JOIN                    |
| Boat B                 |
| Sailor S               |
| Reserves R             |
| sname S                |
| σ B.color=red
+-------------------------+
| JOIN                    |
| Boat B                 |
```
• An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

**Example assuming that Sailor is sorted on sid**

```
\[ \pi_{S.sname} (\text{Merge Join} (\text{JOIN} (\sigma_{R.date=1.1.2005} (S), R), B), \sigma_{B.color=red} (B)) \]
```

- Use index on `R.date`
- Index nested loop
- Use index on `B.bid`
- Filter `B.color` condition in memory
- Materialize intermediate result
- Sort on `R.sid`
• Must consider the interaction of evaluation techniques when choosing evaluation plans: choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. For example,
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost of a subsequent operation (e.g., duplicate elimination).
  - nested-loop join may provide opportunity for pipelining
• Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
    • E.g., number of tuples, number of distinct values for join attributes, etc.
  - Need to estimate statistics for intermediate results to compute cost of complex expressions