Query Processing

Olaf Hartig

David R. Cheriton School of Computer Science
University of Waterloo

CS 640
Principles of Database Management and Use
Winter 2013

Some of these slides are based on
a slide set provided by Ulf Leser.

Notes

Outline

Notes

Query Processing Steps

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Parsing)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- Input: SQL query
- Output: internal representation (e.g. based on relational algebra)

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Query Validation)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

Check whether:

- mentioned tables, attributes, etc. exist,
- comparisons are feasible (e.g. comparability of attribute types),
- aggregation queries have a valid SELECT clause,
- etc.

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (View Resolution)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- Substitute references to views by the view definitions
- (More on views in a later lecture)

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Optimization)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- Considers possible *query execution plans* (QEPs)
  - Different QEPs have different costs (that is, resources needed for their execution)
- Output: an *efficient* QEP
  (i.e. *estimated* cost is the lowest or comparatively low)
- This task is all but trivial...

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Optimization, cont'd)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- The set of all possible QEPs is huge.
  - Optimizers enumerate only a restricted subset (*search space*)
- A desirable optimizer:
  - cost estimation is accurate
  - search space includes low cost QEPs
  - enumeration algorithm is efficient
- (Query optimization will be the main topic of our discussion next week.)

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Plan Compilation)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- Translate the selected QEP into a representation ready for execution (e.g. compiled machine code, interpreted language)

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Processing Steps (Execution)

Parsing
↓
Query Validation
↓
View Resolution
↓
Optimization
↓
Plan Compilation
↓
Execution

- Execute the compiled plan
- Return the query result via the respective interface

As given in:
Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys 25*(2): 73–170 (1993).

Notes

Query Representations

While it is processed by a DBMS, a query goes through multiple representations.

Usually, these are:

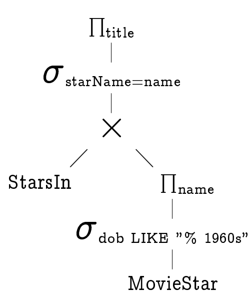
1. an expression in the query language (e.g. SQL query)
2. parse tree
3. logical plan
4. physical plan
5. compiled program

Notes

Logical Plans

- represented as an expression in a logical algebra (such as the relational algebra)
- closely related to the logical data model
- can be visualized as a tree of logical operators

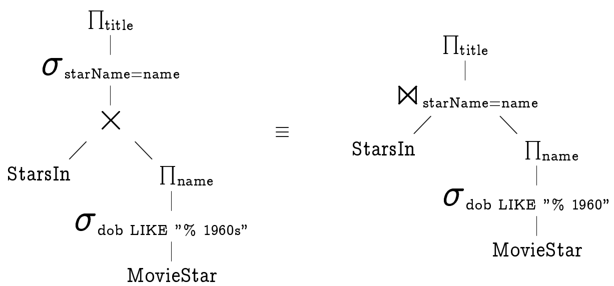
```
SELECT title
FROM StarsIn
WHERE starName IN (
      SELECT name
      FROM MovieStar
      WHERE dob LIKE "% 1960" )
```



Notes

Logical Plans (cont'd)

Remember, algebra expressions may be rewritten into *semantically equivalent* expressions.



Notes

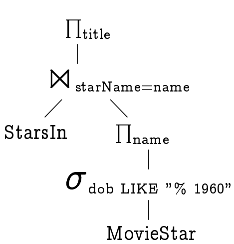
Physical Plans

- Also often called *query execution plan* (QEP)
- Represented as an expression in a *physical* algebra

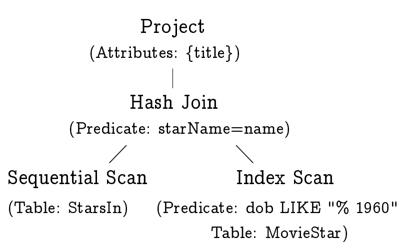
- Physical operators come with a specific algorithm.
  - e.g. a nested loop join

Notes

Physical Plans (Example)



A possible physical plan for the given logical plan:



Notes

Logical Operators vs. Physical Operators

- Logical operators and physical operators do not necessarily map directly into one another.
  - e.g. most join algorithms can project out attributes (without duplicate elimination)
  - e.g. a (physical) duplicate removal operator implements only a part of the (logical) projection operator
  - e.g. a (physical) sort operator has no counterpart in a (set based) logical algebra
- Physical operators can be associated with a cost function (to compute the amount of resources needed for their execution).

Notes

Examples for Physical Operators

- Table scan
- Sorting
- Duplicate elimination
- Selection
- Projection

Notes

Table Scan

- The leaves of each physical operator tree are (physical) tables.
- Accessing them completely implies a *sequential scan*.
  - Load each page of the table.
  - Sequentially scanning a table that occupies n pages has n I/O cost.

Combining the scan with the next operation in the plan is often better.

Example: `SELECT A, B FROM t WHERE A = 5`

Selection: If index on `A` available, perform an index scan instead (i.e. obtain relevant tuples by accessing the index)
- Especially effective if `A` is a key

Projection: Integrate into the table scan (i.e. read all tuples but only pass on attributes that are needed)
- Can also be combined with an index scan.

Notes

Sorting

- Many physical operators require input to be sorted.
- However, the (unsorted) input may not fit into main memory.
- We need an *external sorting* algorithm.
  - Intermediate results are temporarily stored on secondary memory.

Notes

(Simple) External Merge Sort

- First, sort each page internally.
- Group these sorted pages into pairs and for each pair merge its two pages.
- Each of these groups is then merged with another group, resulting in groups of four pages.
- And so on.
- The final group is the completely sorted file.

- This strategy makes use of 3 page buffers in main memory.
- If more buffers are available, we should exploit them ...

Notes

External Merge Sort

- Suppose:
  - $m+1$ page buffers are available in main memory;
  - the input occupies p pages.
- Stage 1:
  - Group the pages into $\frac{p}{m}$ groups.
  - Sort each group (using an m-way merge after sorting each of its pages internally).
- Each additional stage n ($n > 1$):
  - Group the sorted groups from stage n-1 into larger groups such that each larger group consists of m of the previous groups (and m^n pages).
  - For each of these new groups perform an m-way merge.
  - If $m^{n+1} \geq p$ we are done.
- Maximum number of stages: $\lceil \log_m(p) \rceil$
- Maximum number of pages read and written: $2 \times p \times \lceil \log_m(p) \rceil$

Notes

Duplicate Elimination (Option 1)

Two steps:

1. Sort input table (or intermediate result) on DISTINCT column(s)
   - Can be skipped if input is sorted already.
2. Scan sorted table and output unique tuples.

- Generated output is sorted
- Cannot be pipelined

Notes

Duplicate Elimination (Option 2)

- Idea: Scan the input and gradually populate an internal data structure that holds each unique input tuple once.
- For each input tuple check whether it has already been seen
  - No: insert tuple into the data structure and output the tuple
  - Yes: skip to the next input tuple
- Possible data structures:
  - Hash table – might be faster, needs good hash function
  - Binary tree – some cost for balancing, robust
- Does not sort output (but existing sorting would remain)
- Can be pipelined

Notes

Nested Loop Join

(We focus on equi joins and natural joins.)

- General idea:

```
FOR EACH tuple r in relation R DO
    FOR EACH tuple s in relation S DO
        IF r.A = s.A THEN output tuple r ∪ s.
```

- Of course, we do this for relations that are distributed over multiple pages:

```
FOR EACH page p of R DO
    FOR EACH page q of S DO
        FOR EACH tuple r on page p DO
            FOR EACH tuple s on page q DO
                IF r.A = s.A THEN output tuple r ∪ s.
```

- I/O cost: $\text{pages}(R) + \text{pages}(R) \times \text{pages}(S)$

Notes

Nested Loop Join (Possible Improvements)

- Block nested loop join
- Zig-zag join
- Index nested loop join

Notes

Sort-Merge Join

- Sort phase:
  - Sort *both* inputs on the join attributes
  - May need to use an external sorting algorithm
  - Sorting may be skipped for inputs that are already sorted
- Merge phase:
  - Synchronously iterate over both (sorted) inputs
  - Merge and output tuples that can be joined
  - Caution if join values may appear multiple times

Notes

Sort-Merge Join (Example)

R

A	**B**
a2	1
a4	0
a3	2
a4	1

⟶ sort(B)

A	**B**
a4	0
a2	1
a4	1
a3	2

⇐ ⇒

B	**C**
0	c2
2	c3
2	c1
3	c4

⟵ sort(B)

S

B	**C**
3	c4
2	c1
0	c2
2	c3

R $\bowtie$ **S**

A	**B**	**C**
a4	0	c2
a3	2	c3
a3	2	c1

Notes

Sort-Merge Join (Cost Estimation)

- I/O costs for sort phase are approximately:

$$\underbrace{2 \times \text{pages}(R) \times \left\lceil \log(\text{pages}(R)) \right\rceil}_{\text{sorting } R} + \underbrace{2 \times \text{pages}(S) \times \left\lceil \log(\text{pages}(S)) \right\rceil}_{\text{sorting } S}$$

- I/O costs for merge phase are:

$$\text{pages}(R) + \text{pages}(S)$$

Notes

Hash Join

- Idea: use join attributes as hash keys in both input relations
- Choose hash function for building hash tables with m buckets (where m is the number of page buffers available in main memory)
- **Partitioning phase:**
  - Scan relation R and populate its hash table.
  - Whenever the page buffer for a hash bucket is full, write it to disc and start a new page for that buffer.
  - Finally, write the remaining page buffers to disc.
  - Do the same for relation S (using the same hash function!).
  - Result: hash-partitioned versions of both relations on disc
  - Now, we only need to compare tuples in corresponding partitions.
- **Probing phase:**
  - Read in a complete partition from R (assuming $|R| < |S|$).
  - Scan over the corresponding partition of S and produce join tuples.
- I/O costs: $\underbrace{2\Big(\text{pages}(R) + \text{pages}(S)\Big)}_{\text{partitioning}} + \underbrace{\Big(\text{pages}(R) + \text{pages}(S)\Big)}_{\text{probing}}$

Notes

Summary

- For each query different physical operators can be combined into different, semantically equivalent QEPs (query execution plans).
- Each physical operator comes with an algorithm.
- Commonly used techniques for many of these algorithms:
  - Combining: Multiple tasks may be combined once some input data has been read in.
  - Partitioning: Either by sorting or by hashing, we can partition the input(s) and do less work by ignoring many irrelevant combinations.
  - Indexing: Existing indexes may be exploited for reducing work to relevant parts of the input.
- Each of these algorithms has a certain cost.
- Thus, different QEPs have different costs.
- The actual cost can only be estimated (as long as we do not execute the QEP).

Notes