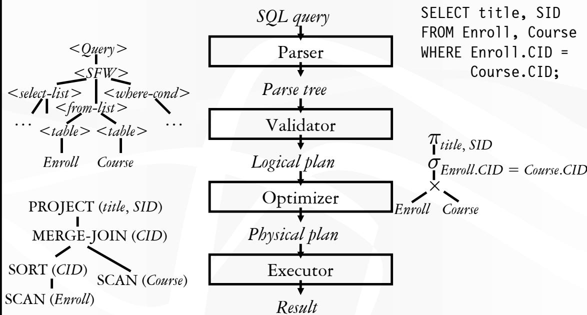
Query Processing: A Systems View

CPS 196.3

Introduction to Database Systems

A query's trip through the DBMS

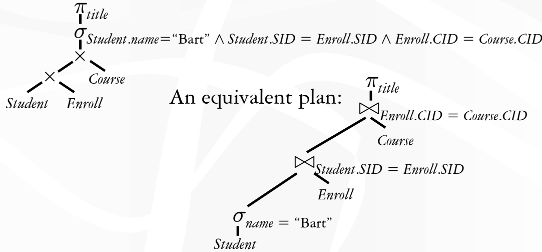


Parsing and validation

- Parser: SQL → parse tree
  - Good old lex & yacc
  - Detect and reject syntax errors
- Validator: parse tree → logical plan
  - Detect and reject semantic errors
    - Nonexistent tables/views/columns?
    - Insufficient access privileges?
    - Type mismatches?
      - Examples: `AVG(name)`, `name + GPA`, `Student UNION Enroll`
  - Also
    - Expand *
    - Expand view definitions
  - Information required for semantic checking is found in system catalog (contains all schema information)

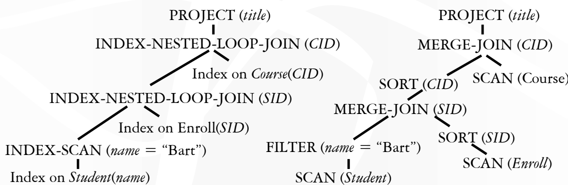
Logical plan

- Nodes are logical operators (often relational algebra operators)
- There are many equivalent logical plans



Physical plan

- Node are physical operators that implement particular algorithms (e.g., scanning, sorting, hashing…)
- There are even more equivalent physical plans
  - Even a single logical plan can have different physical plans
  - Equivalent semantics, but not costs or assumptions!
  - Optimizer: one logical plan → "best" physical plan



Physical plan execution

- Executor: physical plan → result
  - Detect and report run-time errors
    - Example: scalar subquery returns multiple tuples
- Recall a physical plan is a tree of operators
- How are intermediate results passed from children to parents?
  - Temporary files
    - Compute the tree bottom-up
    - Children write intermediate results to temporary files
    - Parents read temporary files
  - Iterator interface (next)

Iterator interface

- Every physical operator maintains its own execution state and implements the following methods:
  - `open()`: Initialize state and get ready for processing
  - `getNext()`: Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  - `close()`: Clean up

An iterator for table scan

- `open()`
  - Allocate a block of memory
- `getNext()`
  - If no block of *R* has been read yet, read the first block from the disk and return the first tuple in the block (or the null pointer if *R* is empty)
  - If there is no more tuple left in the current block, read the next block of *R* from the disk and return the first tuple in the block (or the null pointer if there are no more blocks in *R*)
  - Otherwise, return the next tuple in the memory block
- `close()`
  - Deallocate the block of memory

An iterator for nested-loop join

R: An iterator for the left subtree

S: An iterator for the right subtree

NESTED-LOOP-JOIN

- `open()`

```
R.open(); S.open(); r = R.getNext();
```

- `getNext()`

```
do {
   s = S.getNext();
   if (s == null) {
     S.close(); S.open(); s = S.getNext(); if (s == null) return null;
     r = R.getNext(); if (r == null) return null;
   }
} until (r joins with s);
return rs;
```

- `close()`

```
R.close(); S.close();
```

Execution of an iterator tree

- Call `root.open()`
- Call `root.getNext()` repeatedly until it returns null
- Call `root.close()`

- Requests go down the tree
- Intermediate result tuples go up the tree
- No intermediate files are needed
  - But still useful when an iterator is opened many times
    - Example: the inner iterator in a nested-loop join