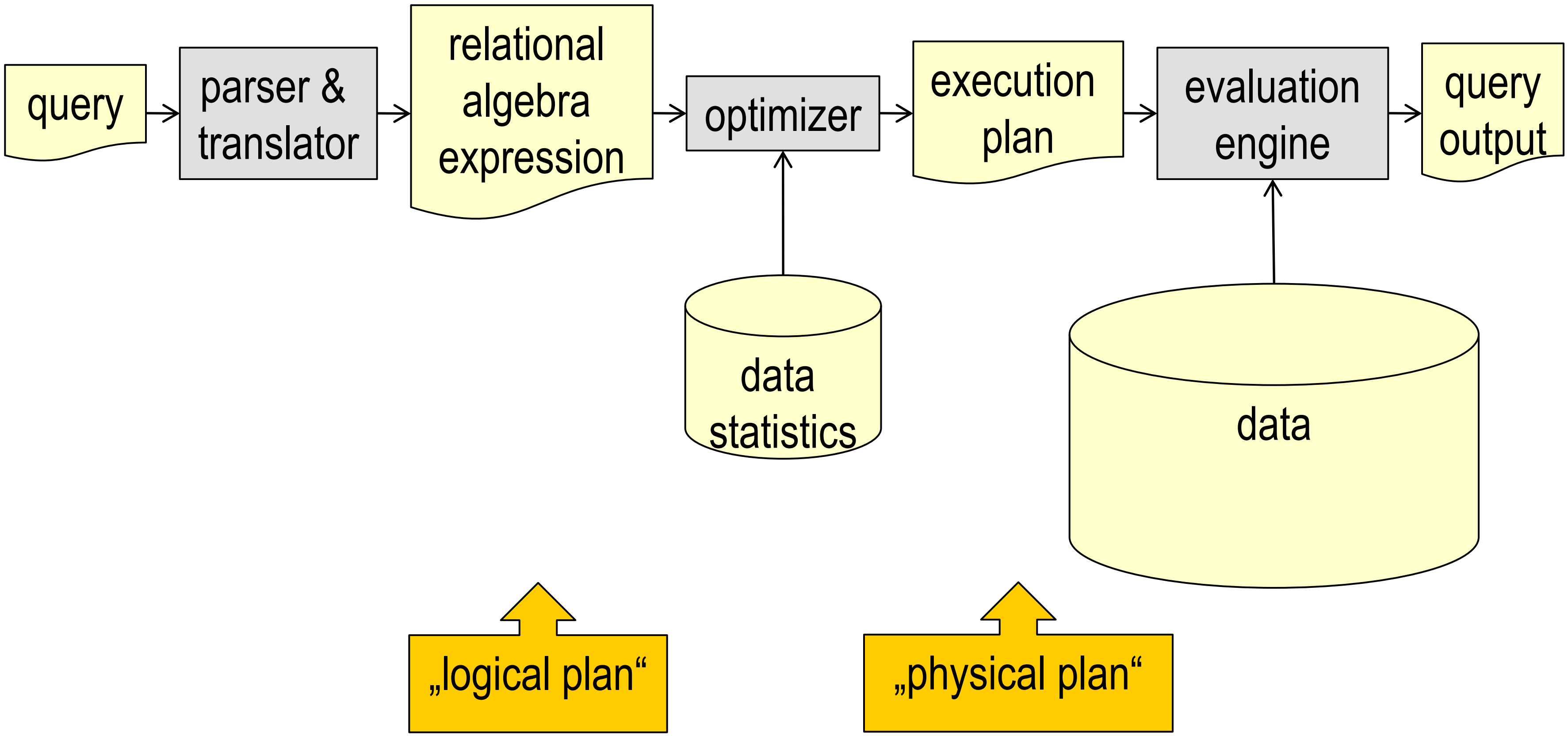
Query Processing: Evaluation of Relational Operations

Jennifer Widom

Steps in Database Query Processing

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – Optim2 - Execution



Running Example

- Tables (what are the keys?):

```
Student(ID, Name, Major)
Course(Num, Dept)
Taking(ID, Num)
```

- Query to find all EE students taking at least one CS course:

```
SELECT Name                         π
FROM   Student, Course, Taking      ×
WHERE  Taking.ID = Student.ID       σ
   AND Taking.Num = Course.Num
   AND Major = 'EE'
   AND Dept = 'CS'
```

… plus subqueries, aggregates, NULL, duplicates, …

Checker (Validation)

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Verifies query tree against database schema
  - All tables in FROM clause exist
  - All columns of tables exist
  - No ambiguities in table references or unqualified attribute references (table names usually added at this point)
  - All comparisons, aggregations, etc. are type-compatible
- Where does info come from?
  - System catalog

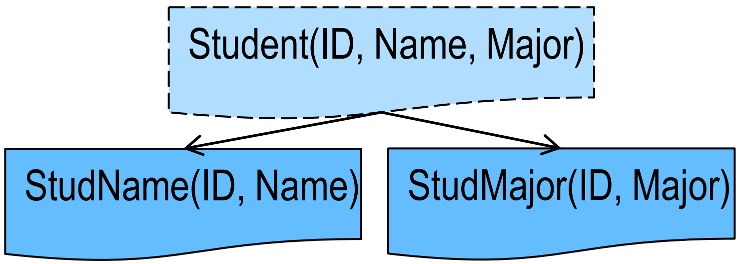
CONSTRUCTOR UNIVERSITY

View Expander

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – Optim2 - Execution

- Suppose Student is view:

```
CREATE VIEW Student AS
SELECT StudName.ID, Name, Major
FROM   StudName, StudMajor
WHERE  StudName.ID = StudMajor.ID
```



- Via view expander original query becomes:

```
SELECT Name
FROM   Student, Course, Taking
WHERE  Taking.ID = Student.ID
   AND  Taking.Num = Course.Num
   AND  Major = 'EE'
   AND  Dept = 'CS'
```

```
SELECT Name
FROM   Course, Taking, Student AS ( SELECT StudName.ID, Name, Major
FROM StudName, StudMajor WHERE  StudName.ID = StudMajor.ID )
WHERE  Taking.ID = Student.ID AND Taking.Num = Course.Num AND
          Student.Major = 'EE' AND Course.Dept = 'CS‘ AND StudName.ID = StudMajor.ID
```

- "flattened":

```
SELECT Name
FROM   Course, Taking, StudName, StudMajor
WHERE  Taking.ID = StudName.ID AND Taking.Num = Course.Num AND
StudMajor.Major = 'EE' AND Course.Dept = 'CS' AND StudName.ID = StudMajor.ID
```

Logical Plan

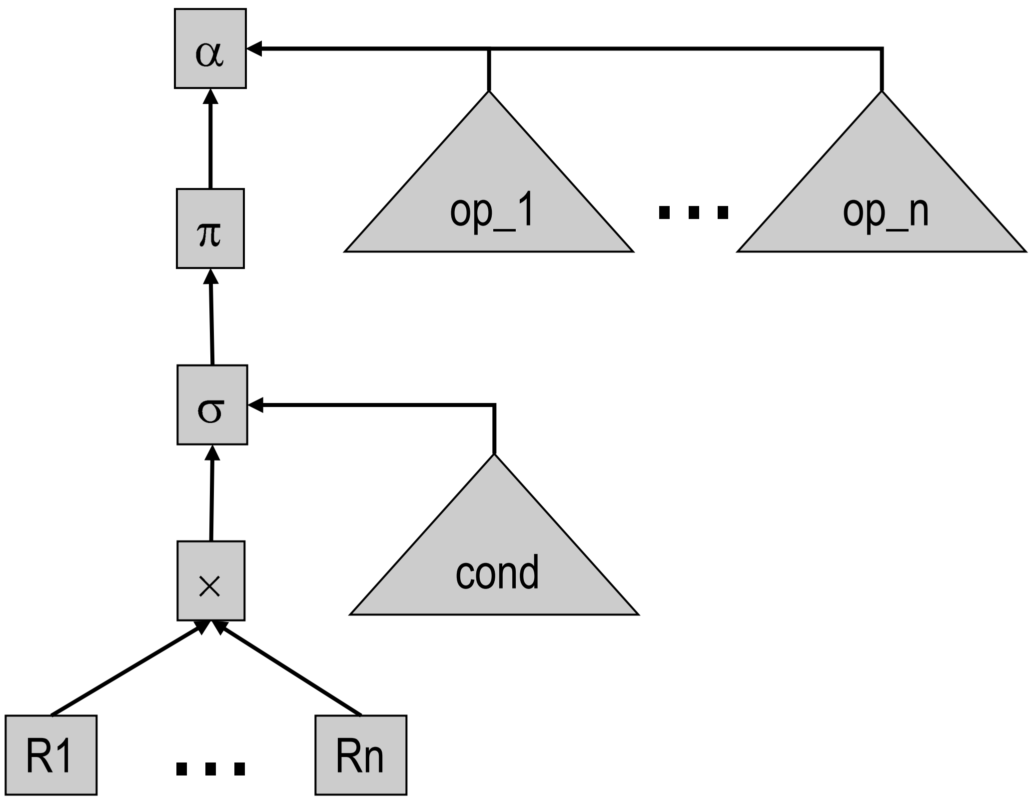
Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Extended relational algebra
  - Problem: SQL more than relational algebra → additional complexity
- Leaf of logical plan = data source = table name
- Inner nodes:
  - Basic operators: SELECT, PROJECT, CROSS-PRODUCT, UNION, DIFFERENCE
  - Abbreviations: NATURAL-JOIN, THETA-JOIN, INTERSECT
  - Extensions: RENAME, AGGREGATE/GROUP-BY, DISTINCT (+ others)
- Usually straightforward mapping
  parse tree → "naive" logical query plan
  - Optimizer may rewrite to "better" plan

Logical Query Tree: Notation Overview

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

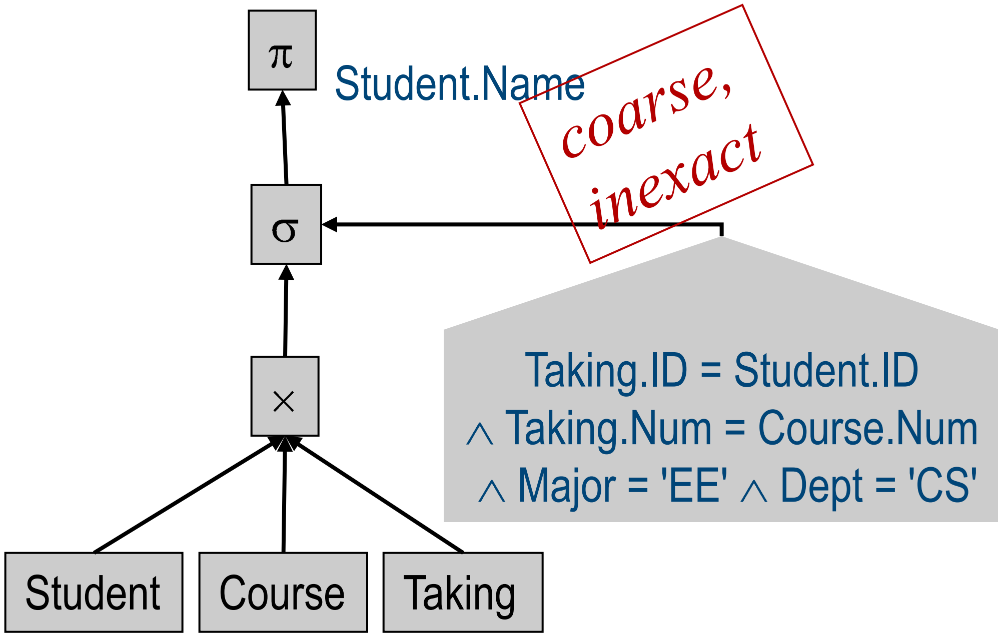
- Logical query tree
  = Logical plan = parsed query, translated into relational algebra
- Equivalent to relational algebra expression (why not calculus?) using:
  - × cross product
  - σ selection from set, based on condition *cond*
  - π projection to attributes
  - α application of an expression to arguments
  - ▷◁ joins...

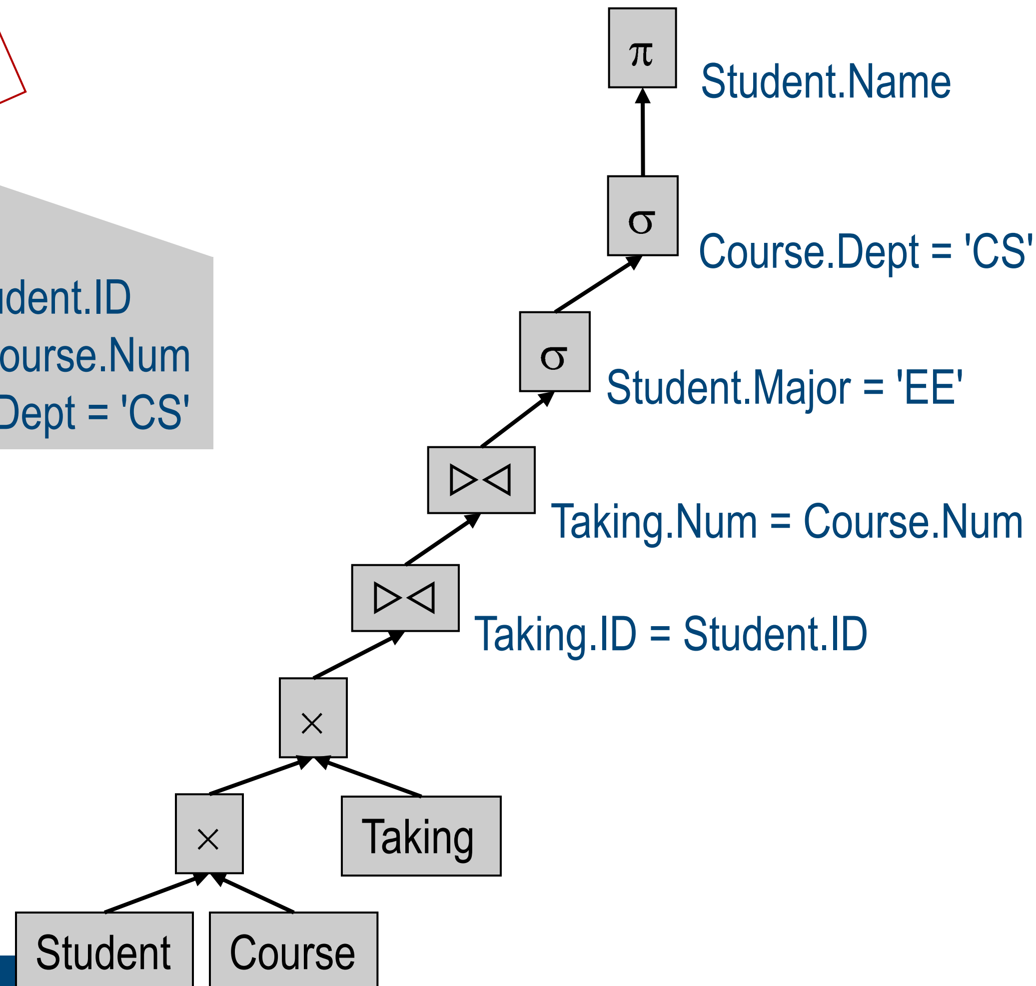


SELECT α(op_1(R1,R2,…)),op_2(R1,R2,…), …)
FROM R1, R2, …
WHERE σ(R1,R2,…)

Logical Query Tree: Example

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution






```
SELECT  Name
FROM    Student, Course, Taking
WHERE   Taking.ID = Student.ID
   AND  Taking.Num = Course.Num
   AND  Major = 'EE'
   AND  Dept = 'CS'
```

Query Optimization

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – Optim2 - Execution

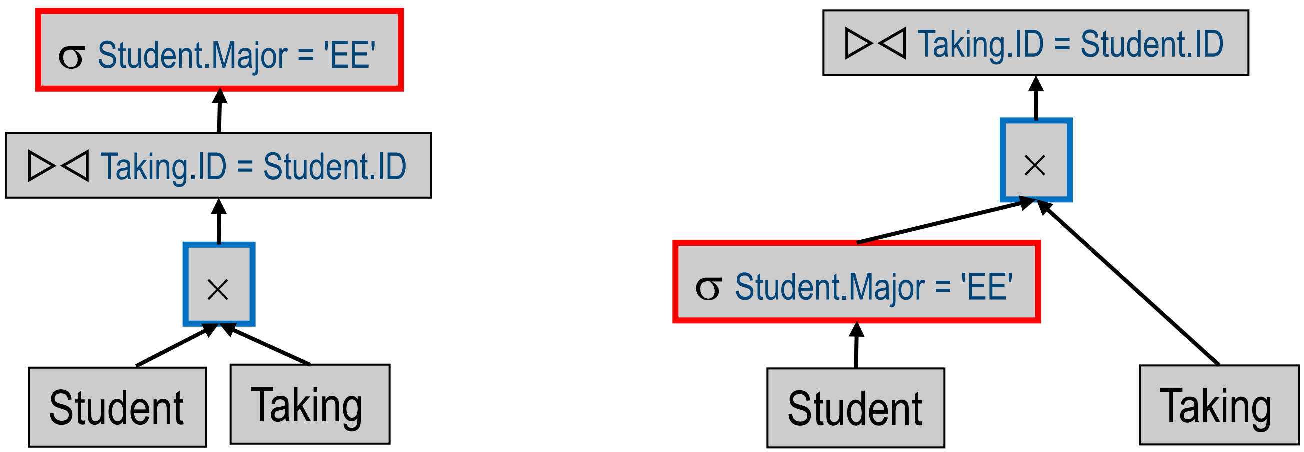
- Optimization = find better, equivalent plan
  - Equivalent = produces same result
  - Logical level optimization = aka heuristic optimization
  - Physical level optimization = aka cost-based optimization
- Two main issues:
  - For a given query, how to find cheapest plans?
  - How is cost of a plan estimated?

Logical („Heuristic“) Optimization

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – Optim2 - Execution

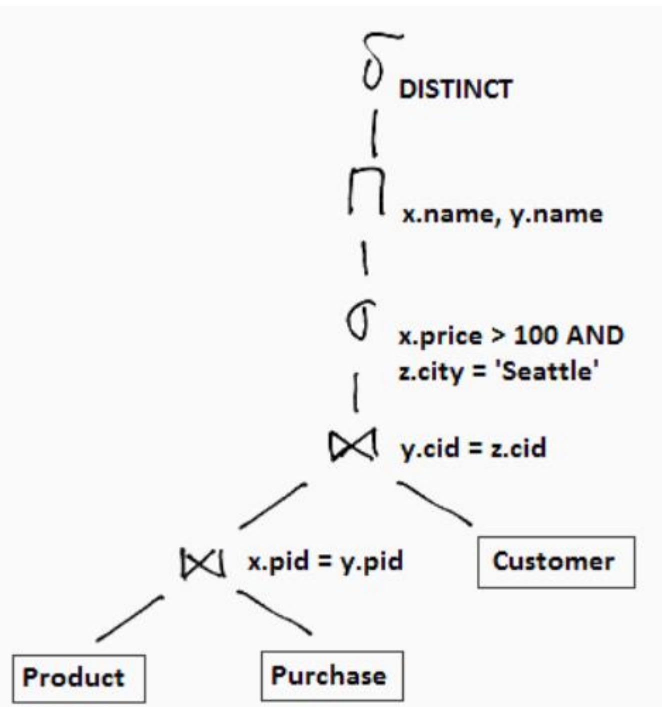
- logical tree $\rightarrow$ (more efficient) logical tree
  - heuristically apply algebraic equivalences
    - *heuristics* = *"looks good, let's try it!"*
- Ex: “push down predicates”

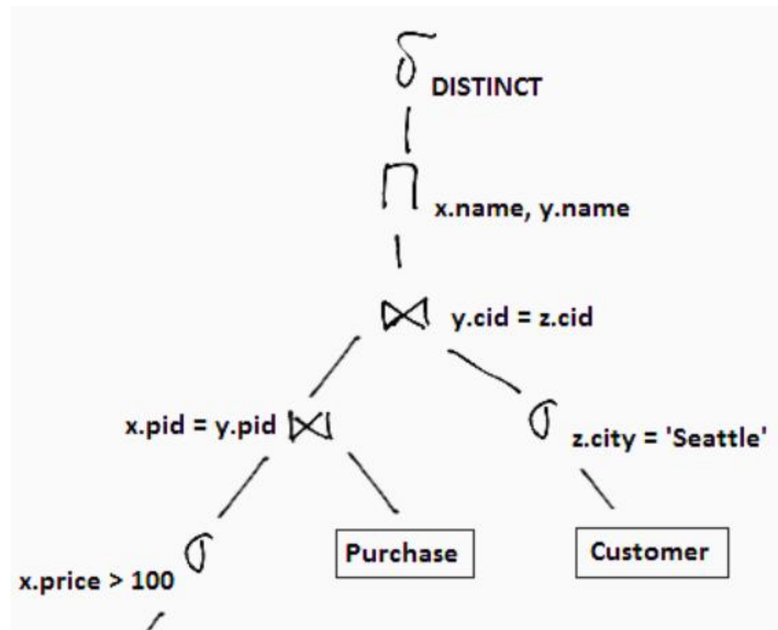
$\sigma_{major='EE'}(\triangleright\triangleleft_{Taking.ID=Student.ID}(Taking,Student)) \equiv \triangleright\triangleleft_{Taking.ID=Student.ID}(Taking,\sigma_{major='EE'}(Student))$



Heuristic Optimization: Another Example [src]

```sql
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid AND y.cid = z.cid AND
      x.price > 100 AND z.city = 'Seattle'
```





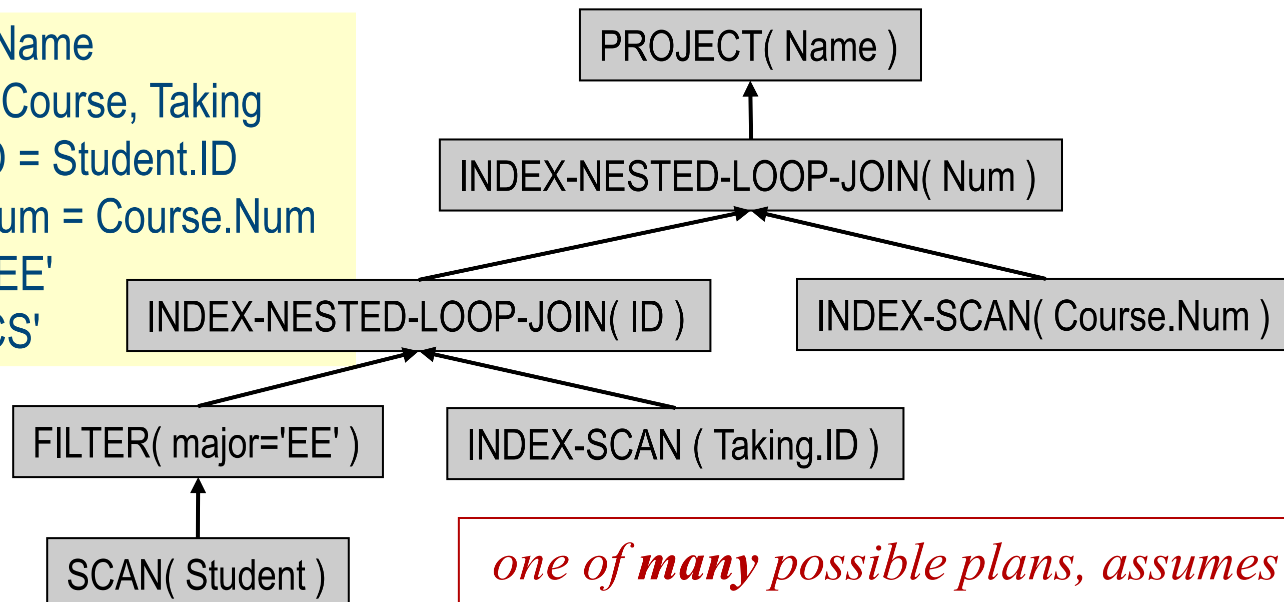
Why better?

Physical Query Plan

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Typically, several algorithm variants for implementing query node = operator
- Physical plan created by concretizing particular algorithm per node
  - Based on indexes, table sizing, predicate selectivity, ...
- Ex:

```
SELECT  Student.Name
FROM    Student, Course, Taking
WHERE   Taking.ID = Student.ID
    AND Taking.Num = Course.Num
    AND Major = 'EE'
    AND Dept = 'CS'
```



*one of **many** possible plans, assumes particular index situation!*

Sample Physical Plan, Textual

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

IBM Informix Dynamic Server

```
SET EXPLAIN ON AVOID_EXECUTE;
SELECT   C.customer_num, O.order_num
FROM     customer C, orders O, items I
WHERE    C.customer_num = O.customer_num
         AND O.order_num = I.order_num
```

```
for each row in the customer table do:
  read the row into C
  for each row in the orders table do:
    read the row into O
    if O.customer_num = C.customer_num then
      for each row in the items table do:
        read the row into I
        if I.order_num = O.order_num then
          accept the row and send to user
        end if
      end for
    end if
  end for
end for
```

In PostgreSQL:
EXPLAIN ANALYZE

Physical Plan Operators

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Usually: physical plan leaf = table, index
- Access methods for single tables:
  - Table scan: SCAN(table)
  - Index scan: INDEX-SCAN(index)
  - Condition-based index scan: INDEX-SCAN-P (index, predicate)
    (note: obviously the predicate must be compatible with the index to be scanned)
- Join methods:
  - NESTED-LOOP JOIN (various algorithms / improvements);
  - SORT-MERGE JOIN
  - HASH JOIN (various algorithms)
- In a parallel system: EXCHANGE
- In a distributed system: SHIP

Physical Plan Generation

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Even more possible physical query plans for a given logical plan
- physical plan generator tries to select "optimal" one
  - sometimes called "physical plan enumerator"
  - usually wrt response time or (in some cases) throughput
- How are intermediate results passed from children to parents?
  - Temporary files
  - Iterator interface (next)

Iterator Interface

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Every operator maintains its own execution state, implements the following methods:
  - open( ):
    Initialize state
  - getNext( ):
    Return next tuple (or null pointer); read more data when needed
  - close( ):
    Clean up
- "ONC protocol"

Iterator for Table Scan

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- open( )
  - Allocate buffer space
- getNext( )
  - If no block of R has been read yet: read first block from the disk; return first tuple in the block (or null pointer if R is empty)
  - If no more tuple left in current block: read next block of R from disk; return first tuple in block (or null pointer if no more blocks in R)
  - Return next tuple in block
- close( )
  - Deallocate buffer space

Iterator for Nested-Loop Join

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- **open( )**
  - R.open( ); S.open( );
  - r = R.getNext( );
- **getNext( )**
  - Repeat until r and s join:

```
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;
}
```

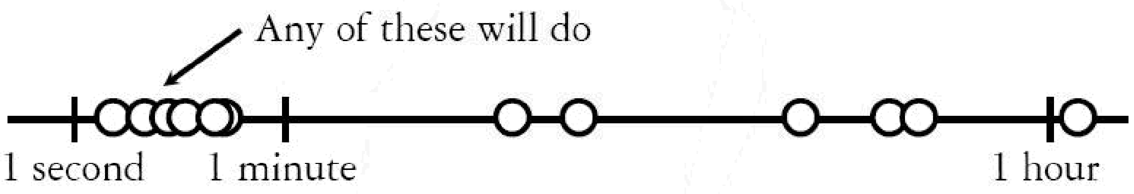
  - return rs;
- **close( )**
  - R.close( ); S.close( );

```
for r in R:
    for s in S:
        if r joins s
        then return rs
```

Physical („Cost-Based“) Optimization

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – Optim2 - Execution

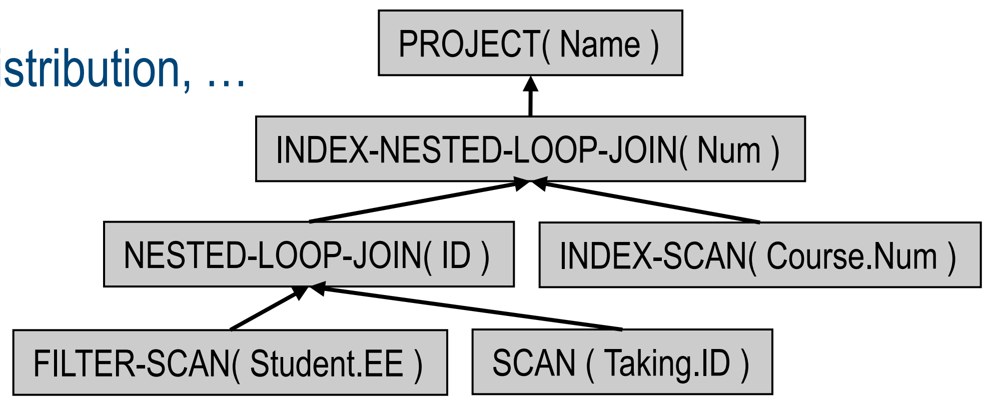
- Approach:
  - enumerate all (?) possible physical plans that can be derived from given logical plan
  - estimate cost for each plan
  - pick best (i.e., least cost) alternative
- Ideally: Want to find best plan; practically: Avoid worst plans!

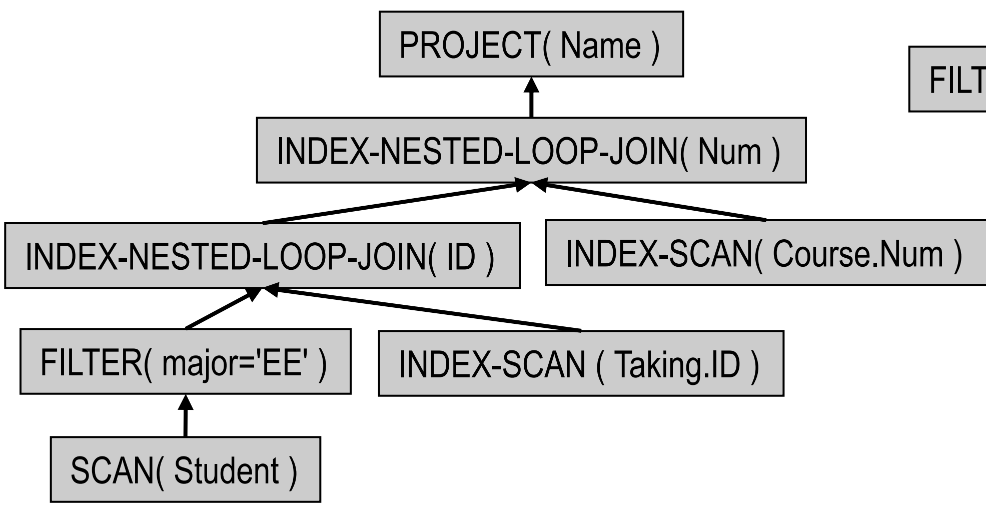


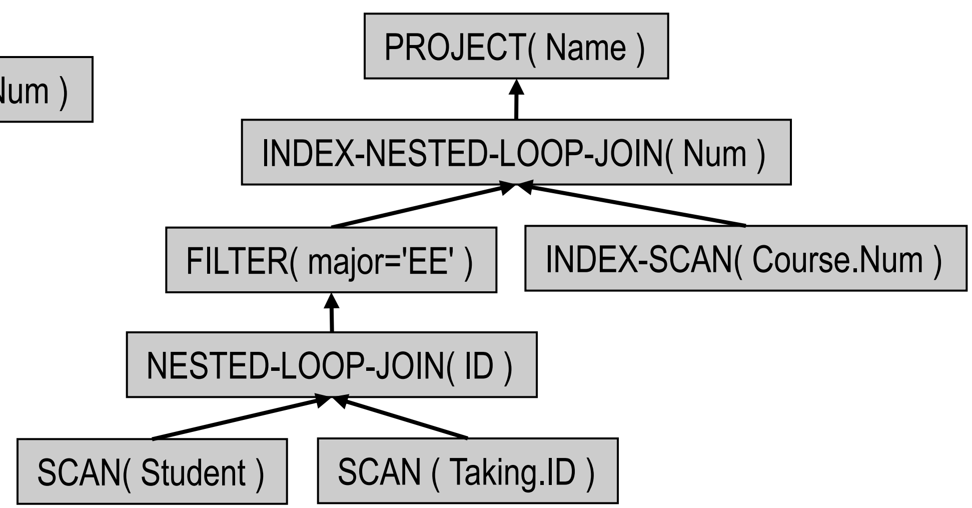
Physical („Cost-Based“) Optimization

Parser – Checker - Views - Logical plan – Optim1 - Physical plan – **Optim2** - Execution

- Estimate costs, based on physical situation
  - concrete table sizes, indexes, data distribution, …
  - Find cheapest plan







Summary: Logical vs Physical Query Plan

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Both are trees representing query evaluation
- Leaves of the tree represent data (table vs table/index)
- Internal nodes of the tree = "operators" over the data
- Logical vs physical plan:

	Level	Operators
Logical plan	higher-level, algebraic	query language constructs
Physical plan	lower-level, operational	"access methods"

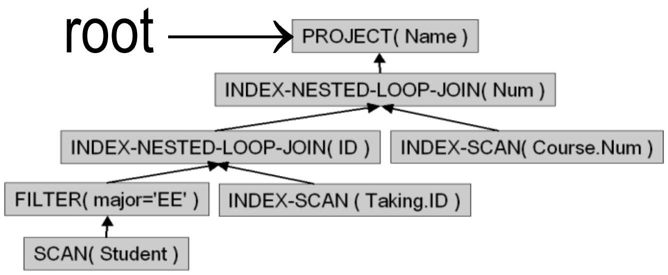
Optional: Code Generator

Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Code gen. - Execution

- Translates physical query plan tree into executable code
  - Possibly mixed hardware: CPU, GPU, FPGA, ...
- Often instead: compile into "database machine code" program
- Very system-specific
  - may instead use a query plan *interpreter* (see next)

Finale: Execution of Tree

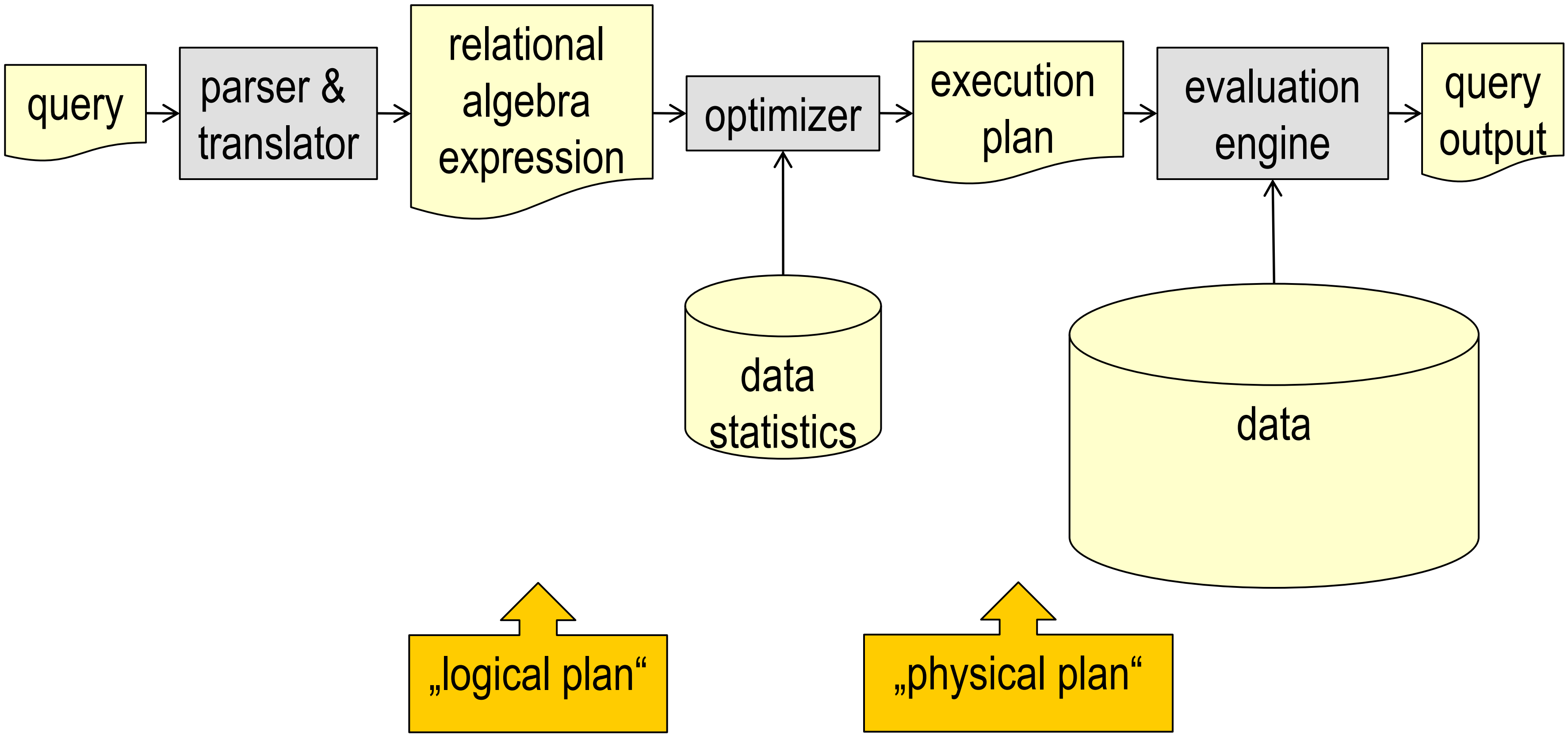
Parser – Checker - Views - Logical plan - Rewriter - Physical plan - Optim. - **Execution**




```
result = {};
root.open();
do
{
        tmp = root.getNext();
        result += tmp;
} while (tmp != NULL);
root.close();
return result;
```

- Recursive evaluation of tree
  - Requests go down
  - Intermediate result tuples go up
- Often instead: compile into "database machine code" program
  - CPU, GPU, FPGA, ...

Summary


query
parser & translator
relational algebra expression
optimizer
execution plan
evaluation engine
query output
data statistics
data
„logical plan“
„physical plan“