Insight into DBMS: Design and Implementation

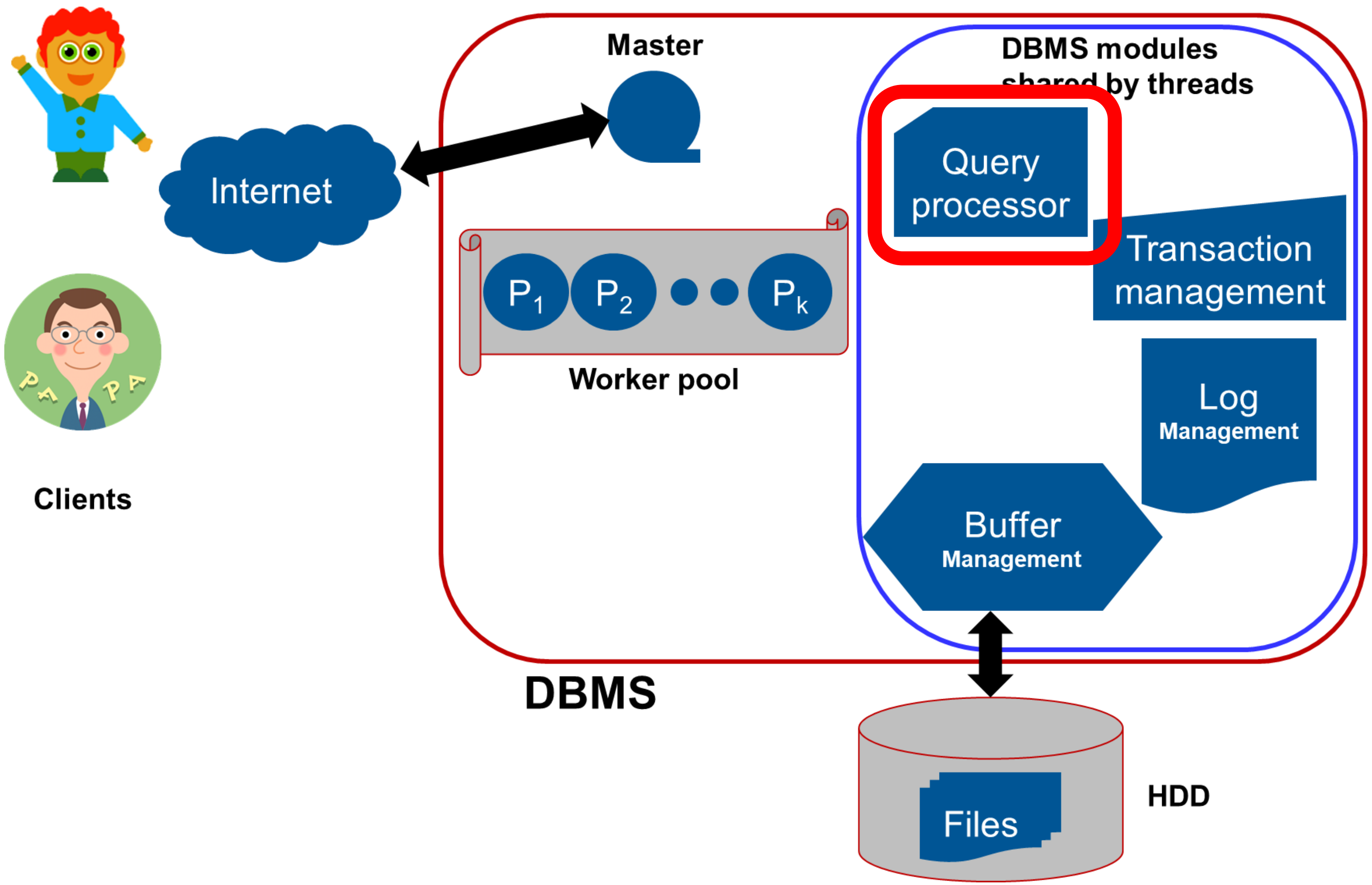
Chapter 4 B: SQL to File Operations
RA → File Operations

孔令波
mlinking@126.com
+86 15010255486

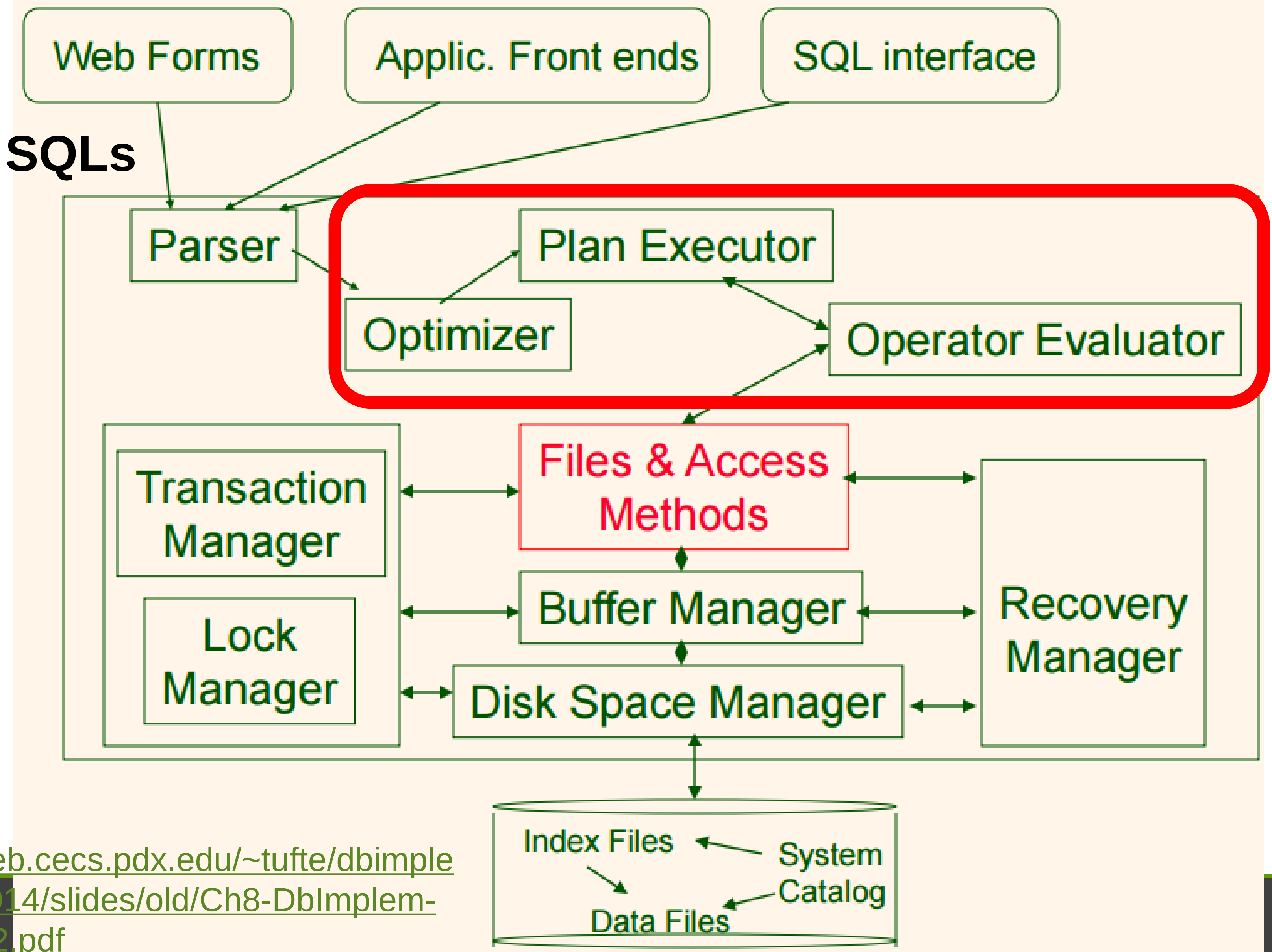
Outline of the chapters covered

- **Introduction**
- **Overview of the projects**
- **Demonstration of Development environment**
  - Watch and practice by yourself
- **My understanding about (R)DBMS**
  - History and D&I
- **SQL translation with 2 conversions**
  - SQL → RA (Relational Algebra)
  - RA → Sequence of File operations
- **Transaction control**
- **Deeper**
  - File, (R)DBMS, ERP, DW, Big Data (No SQL, SQL again)
  - SQL on MPP and Hadoop (Greenplum, **HAWQ**)

- **Sit back, but not relax!**
  - We'll start the most exciting yet challenging part – SQL translator !


Master
DBMS modules
shared by threads
Internet
Query
processor
Transaction
management
P1
P2
Pk
Worker pool
Log
Management
Clients
Buffer
Management
DBMS
HDD
Files

Database Architecture



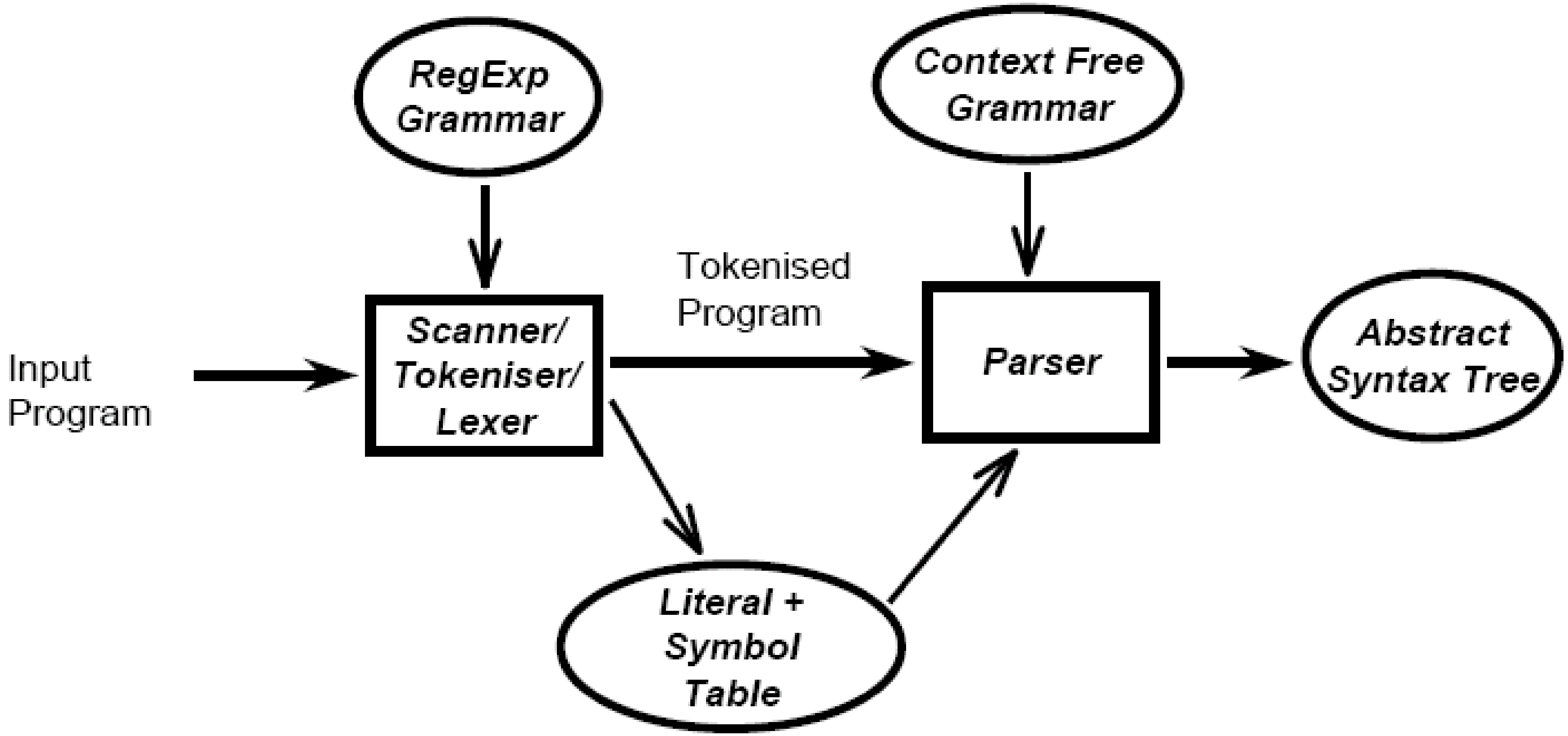
http://web.cecs.pdx.edu/~tufte/dbimplem-spr2014/slides/old/Ch8-DbImplem-Fall2012.pdf

Chapter 3 B: How to get File operations for SQL really

- **My understanding about DBMS**
  - 3 stages before SQL is really executed
    - Parse tree → Logical Query Plan (**LQP**)/RA tree
    - Optimize LQP: Select optimal plan
      - Algebraic Optimization
    - LQP → Physical Query Plan (**PQP**)
      - Physical optimization
    - Execution of PQP
      - Materialize or pipeline
- **Advanced project in reading PostgreSQL**
  - Parse and Execute SQL in PostgreSQL

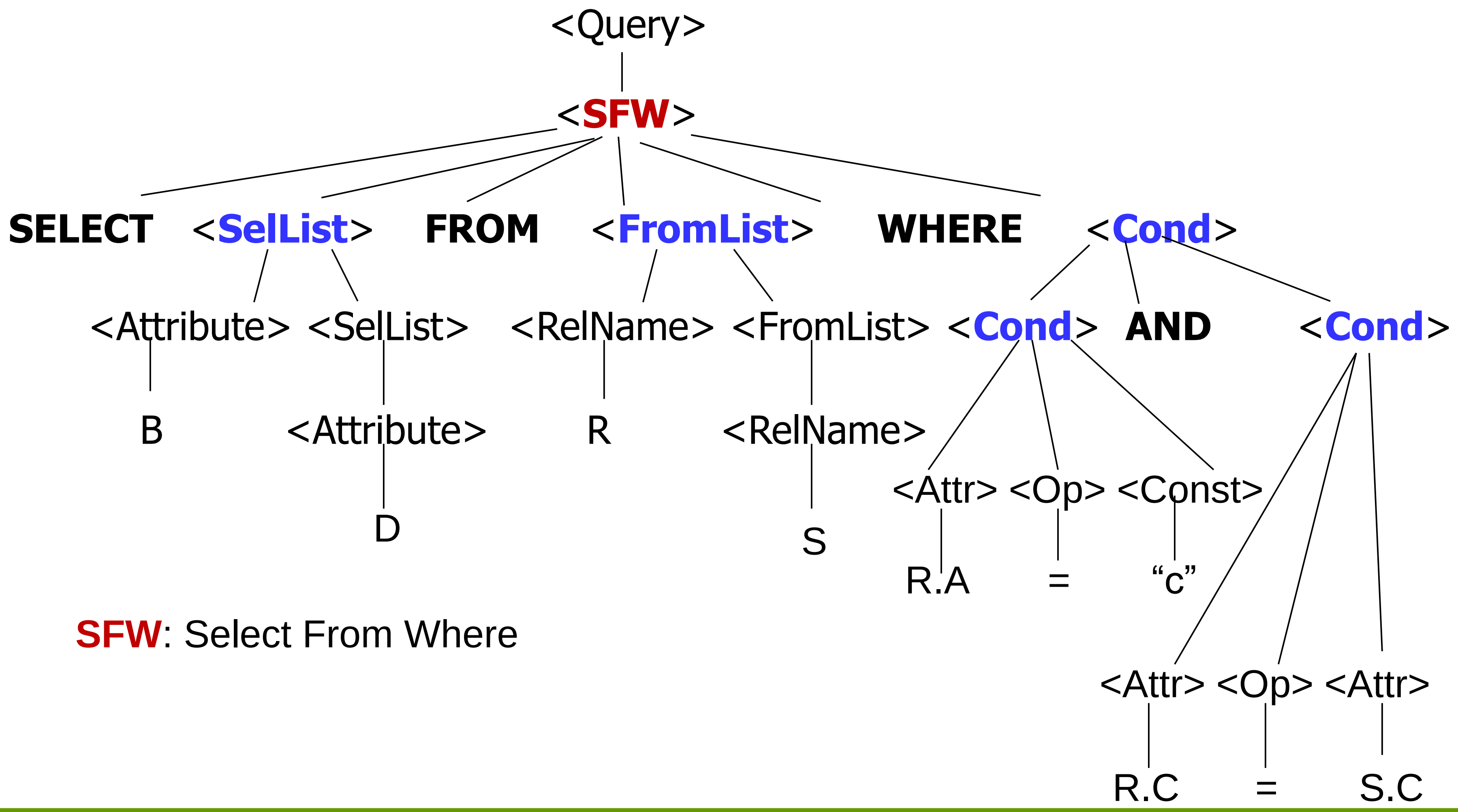
Parse Tree

Select B,D
From R,S
Where R.A = “c” $\wedge$ R.C=S.C



Parse Tree

Select B,D
From R,S
Where R.A = “c” $\wedge$ R.C=S.C



SFW: Select From Where

Relational Algebra [关系代数] is needed to carry out SQL's translation

The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management.

(E. F. Codd)

Enrolled

cid	grade	sid
Carnatic101	C	53666
Reggae203	B	53666
Topology112	A	53650
History105	B	53666

Students

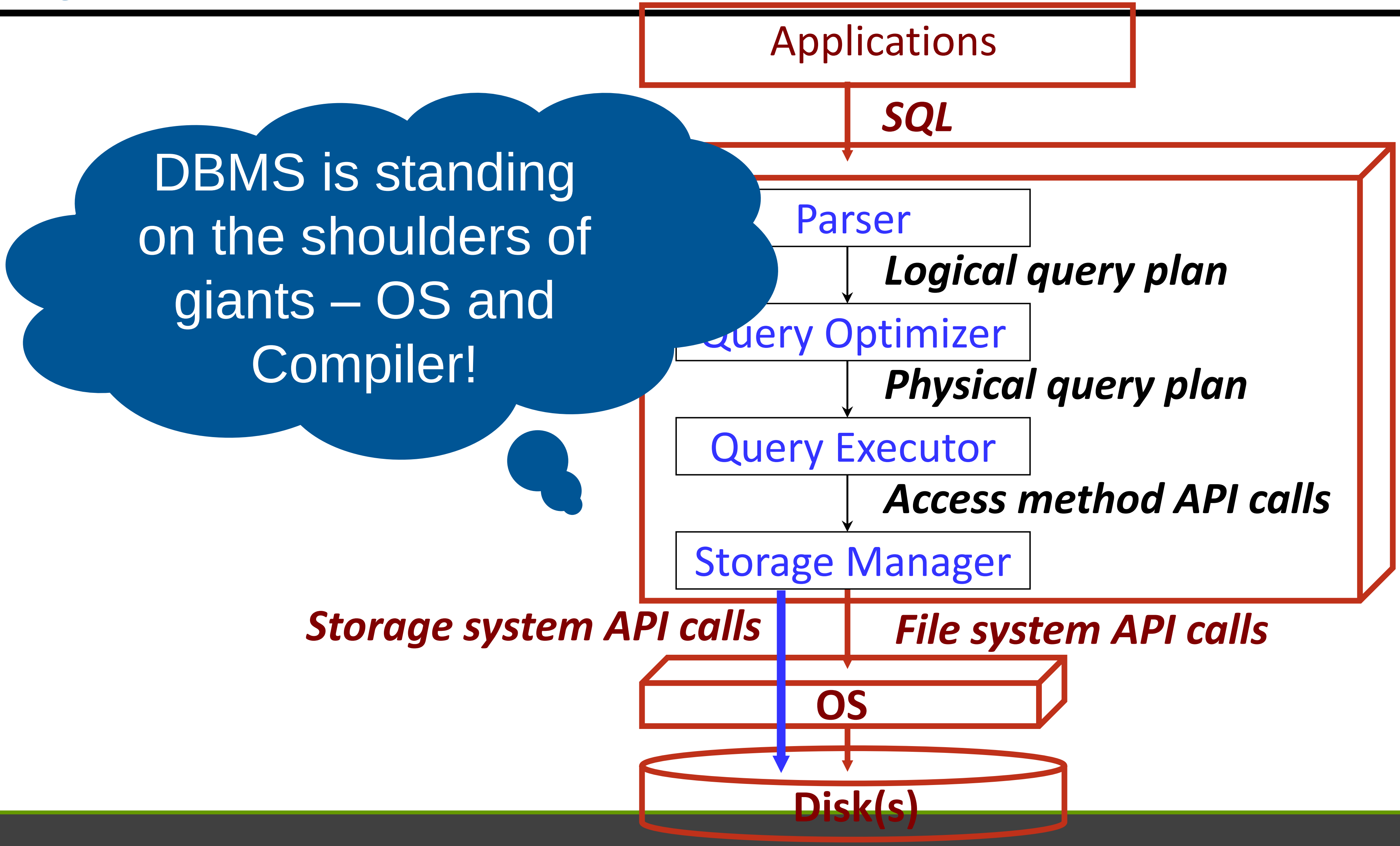
sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- **Pre-Relational:**
  write a program
- **Relational SQL**
  Select name, cid from students s, enrolled e where s.sid = e.sid
- **Relational Algebra**
  $$\pi_{name,cid}(Students \bowtie_{sid} Enrolled)$$
- **Relational Calculus**
  {R | R.name = S.name ∧ R.cid = S.cid ∧ ∃S(S∈Students∧ ∃E(E∈Enrolled ∧ E.sid = S.sid))}

With RA (Relational Algebra)

- **Three issues are concerned to carry out SQL's translation into File operations**
  - SQL → Parse Tree
    - We have learned before
  - Parse Tree → Relational Algebra expression (RA)/Logical Query plan (**LQP**)
    - Optimizing will be sketched
  - LQP → Physical Query Plan (**PQP**)
    - Map RA into corresponding functions [File Operations] – iterators?
      - ✓ TableScan(a), JoinScan(a, b), …
  - Executing
    - Derive operation sequence and execute

Leave SQL execution later



Chapter 3 B: How to get File operations for SQL really

My understanding about DBMS

- 3 stages before SQL is really executed
  - Parse tree → Logical Query Plan (<u>LQP</u>)/RA tree
  - Optimize LQP: Select optimal plan
    - Algebraic Optimization
  - LQP → Physical Query Plan (<u>PQP</u>)
    - Physical optimization
  - Execution of PQP
    - Materialize or pipeline

Advanced project in reading PostgreSQL

- Parse and Execute SQL in PostgreSQL

- **Mapping from an SQL query to an RA expression requires:**
  - a collection of **templates** for particular kinds of queries
  - a matching process to ...
    - determine what kind of query we have (i.e. choose a template)
    - bind components of actual query to slots in the template
  - mapping rules to ...
    - convert the matched query into relational algebra
    - filling slots in RA expression from matched components
- **May need to use multiple templates to map whole SQL statement.**

Parse Tree to Relational Algebra

- **This first rule allows a simple parse tree to be transformed into relational algebra**
  - A <**Query**> with a <**Condition**> without sub-queries is replaced by a relational algebra expression

- **The relational algebra expression consists of**
  - The **product** ($\times$) of all the relations in the <**FromList**>, which is an argument to
  - A **selection** σ_C where C is the <**Condition**>, which is an argument to
  - A **projection** π_L where L consists of the attributes in the <**SelList**>

- **There are still many other rules … your turn!**

How the initial tree is built:

- Apply **Cartesian cross** for relations in the **FROM** clause
- Then the conditions (join & selections) from **WHERE** clause are added
- Then **projections** from the **SELECT** clause

Canonical Query Translation

Canonical translation of SQL queries into algebra expressions.

Structure:

select distinct $a_1, \ldots, a_n$
from $R_1, \ldots, R_k$
where p

(Simplified) Canonical translation of SQL to algebra

1. Let $R_1 \ldots R_k$ be the entries in the **from** clause of the query. Construct the expression

$$F := \begin{cases} R_1 & \text{if } k = 1 \\ ((\ldots(R_1 \times R_2) \times \ldots) \times R_k) & \text{else} \end{cases}$$

2. The **where** clause is optional in SQL. Therefore, we distinguish the cases that there is no **where** clause and that the **where** clause exists and contains a predicate p. Construct the expression

$$W := \begin{cases} F & \text{if there is no } \textbf{where} \text{ clause} \\ \sigma_p(F) & \text{if the } \textbf{where} \text{ clause contains } p \end{cases}$$

3. Let s be the content of the **select distinct** clause. For the canonical translation it must be of either '*' or a list $a_1, \ldots, a_n$ of attribute names. Construct the expression

$$S := \begin{cases} W & \text{if } s = \text{'*'} \\ \Pi_{a_1,\ldots,a_n}(W) & \text{if } s = a_1, \ldots, a_n \end{cases}$$

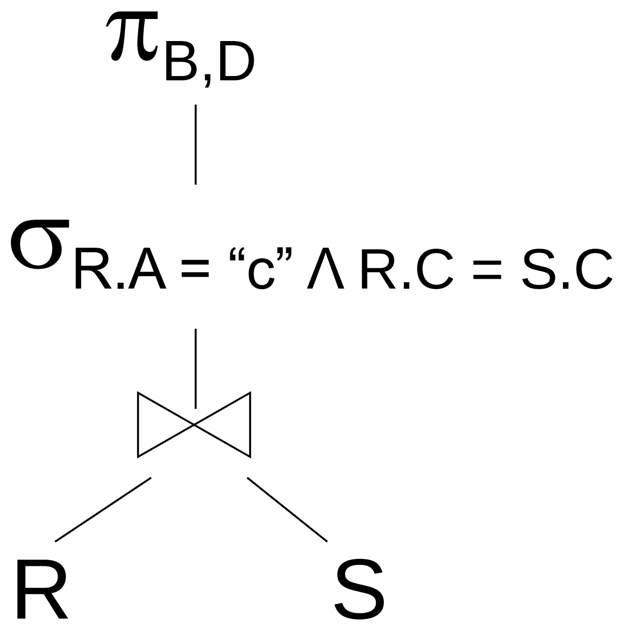
4. Return S.

Initial Logical Plan

$\bowtie$

Cartesian product of R and S

Select B,D
From R,S
Where R.A = “c” $\wedge$ R.C=S.C



Relational Algebra: $\Pi_{B,D}\,[\,\sigma_{R.A=“c”\wedge R.C = S.C}\,(RXS)]$

Chapter 3 B: How to get File operations for SQL really

My understanding about DBMS

- 3 stages before SQL is really executed
  - Parse tree → Logical Query Plan (**LQP**)/RA tree
  - Optimize LQP: Select optimal plan
    - ✓ Algebraic Optimization
  - LQP → Physical Query Plan (**PQP**)
    - ✓ Physical optimization

Advanced project in reading PostgreSQL

- Parse and Execute SQL in PostgreSQL

One SQL could be represented as different logical plans

- **Optimization of Query trees**
  - Making the initial tree more efficient
    - Many different relational algebra expressions correspond to the same query
    - The trick is to take the initial tree and convert it into an equivalent, more efficient tree
  - There are rules to help do this
    - Basically, we want to move select and project as far down the tree as possible and still maintain equivalence
      - This is because they limit the size of the relation

- **Expression Rewriting Rules**
  - Since RA is a **well-defined formal system**, there exist many algebraic **laws** on RA expressions, which can be used as a basis for expression rewriting in order to produce **equivalent** (more-efficient) expressions

- **Expression transformation based on such rules can be used**
  - to simplify/improve SQL→RA mapping results
  - to generate new plan variations during query optimization

Relational Algebra Laws

Commutative and Associative Laws:

- $R \times S \leftrightarrow S \times R$, $(R \times S) \times T \leftrightarrow R \times (S \times T)$
- $R \bowtie S \leftrightarrow S \bowtie R$, $(R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$ (natural join)
- $R \cup S \leftrightarrow S \cup R$, $(R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \cap S \leftrightarrow S \cap R$, $(R \cap S) \cap T \leftrightarrow R \cap (S \cap T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R$ (theta join)

But it is **not** true in general that

- $(R \bowtie_{Cond1} S) \bowtie_{Cond2} T \leftrightarrow R \bowtie_{Cond1} (S \bowtie_{Cond2} T)$

Example: $R(a, b)$, $S(b, c)$, $T(c, d)$, $(R\ Join_{[R.b>S.b]}\ S)\ Join_{[a<d]}\ T$

Cannot rewrite as $R\ Join_{[R.b>S.b]}\ (S\ Join_{[a<d]}\ T)$ because neither $S.a$ nor $T.a$ exists.

... Relational Algebra Laws

Selection commutativity (where c is a condition):

- $\sigma_c(\sigma_d(R)) \leftrightarrow \sigma_d(\sigma_c(R))$

Selection splitting (where c and d are conditions):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c(\sigma_d(R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$ (but only if R is a set)

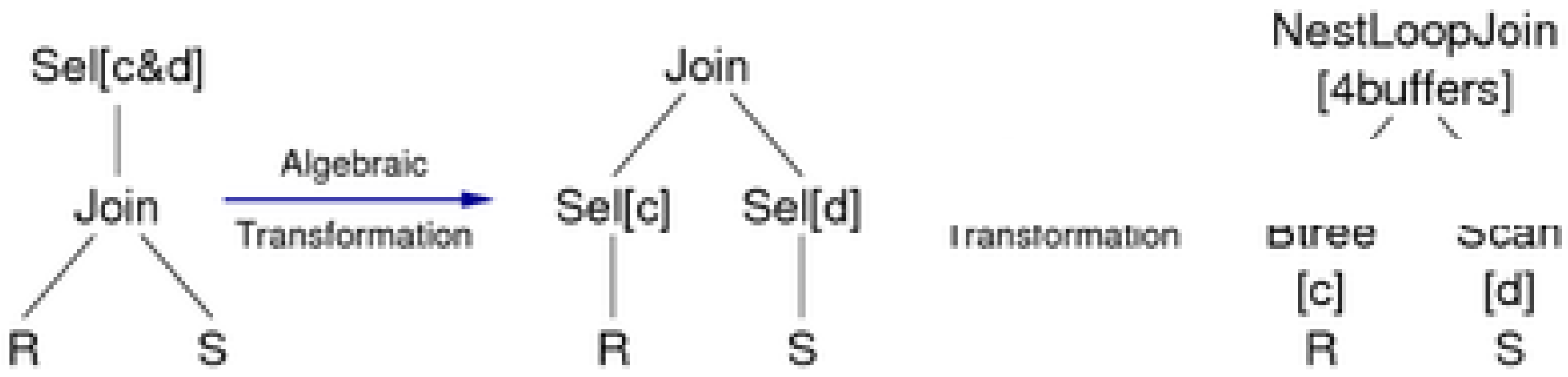
Selection pushing ($\sigma_c(R \; op \; S)$):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S$
  (must be pushed into both arguments of union)
- $\sigma_c(R - S) \leftrightarrow \sigma_c R - S$, $\sigma_c(R - S) \leftrightarrow \sigma_c R - \sigma_c S$
  (must be pushed into left branch of difference, may be pushed to right branch)

Algebraic/Logical Optimization [代数/逻辑优化]

- Make use of algebraic equivalences:
  - examine query expression
  - search for applicable transformation rules (**heuristics**)
  - generate equivalent (and "better") algebraic expressions
- Most commonly used heuristic:
  - Apply **Select** and **Project** before **Join**
- Rationale: minimizes size of intermediate relations.
  - Can potentially be done during the SQL→RA mapping phase.
- Algebraic optimization cannot assist with finding good join order.

Example of optimisation transformations:



For join, may also consider sort/merge join and hash join.

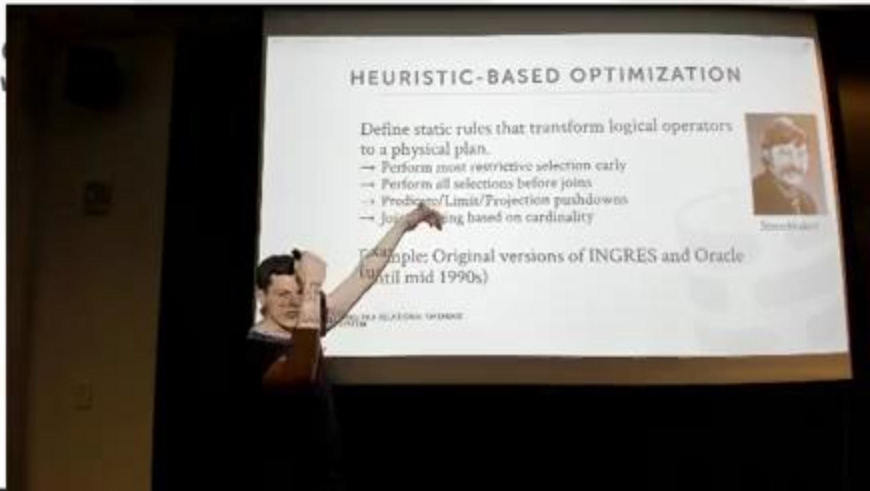
HEURISTIC-BASED OPTIMIZATION

Define static rules that transform logical operators to a physical plan.

→ Perform most restrictive selection early
→ Perform all selections before joins
→ Predicate/Limit/Projection pushdowns
→ Join ordering based on cardinality

Stonebraker

Example: Original versions of INGRES
(until mid 1990s)



QUERY PROCESSING IN A RELATIONAL DATABASE MANAGEMENT SYSTEM
VLDB 1979

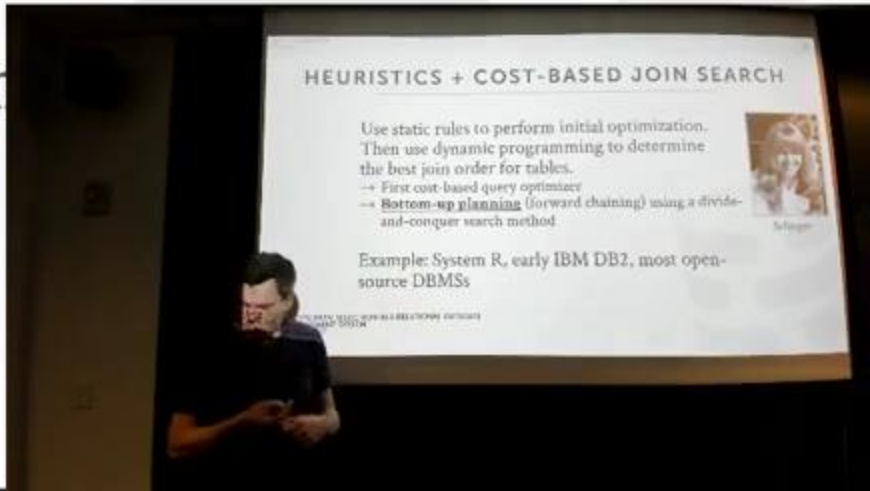
HEURISTICS + COST-BASED JOIN SEARCH

Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables.

→ First cost-based query optimizer
→ **Bottom-up planning** (forward chaining) using a divide-and-conquer search method

Selinger

Example: System R, early IBM DB2, m
source DBMSs



ACCESS PATH SELECTION IN A RELATIONAL DATABASE MANAGEMENT SYSTEM
SIGMOD 1979

Chapter 3 B: How to get File operations for SQL really

My understanding about DBMS

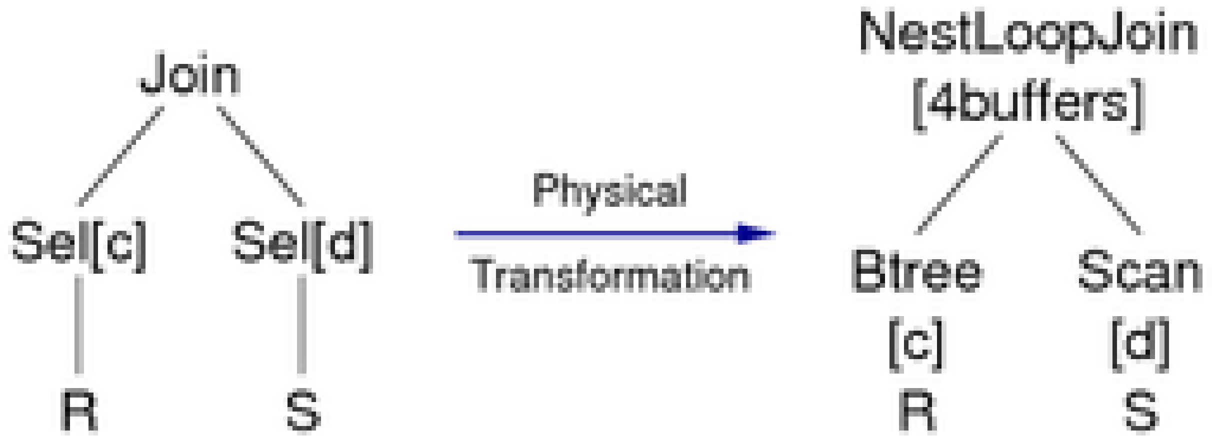
- 3 stages before SQL is really executed
  - Parse tree → Logical Query Plan (LQP)/RA tree
  - Optimize LQP: Select optimal plan
    - Algebraic Optimization
  - LQP → Physical Query Plan (PQP)
    - Physical optimization
  - Execution of PQP
    - Materialize or pipeline

Advanced project in reading PostgreSQL

- Parse and Execute SQL in PostgreSQL

LQP→PQP

- **Here we have another conversion/translation from RA/LQP to PQP**
  - Mapping Relational operators (Select, Product, Project, …) to **file functions** (TableScan(…), Join(…), Btree(…), …)
- Rules
  - leaves of LQP (stored relations) become **scan** operators
  - internal nodes of LQP (operators) become one or more physical operations (algorithms)
  - edges of LQP are marked as "**pipeline**" or "**materialize**"



Example Physical Query Plans

Filter(x=1 AND z<5)

IndexScan(R,y=2)

$\sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5}(R)$

two-pass
hash-join
101 buffers

materialize

two-pass
hash-join
101 buffers

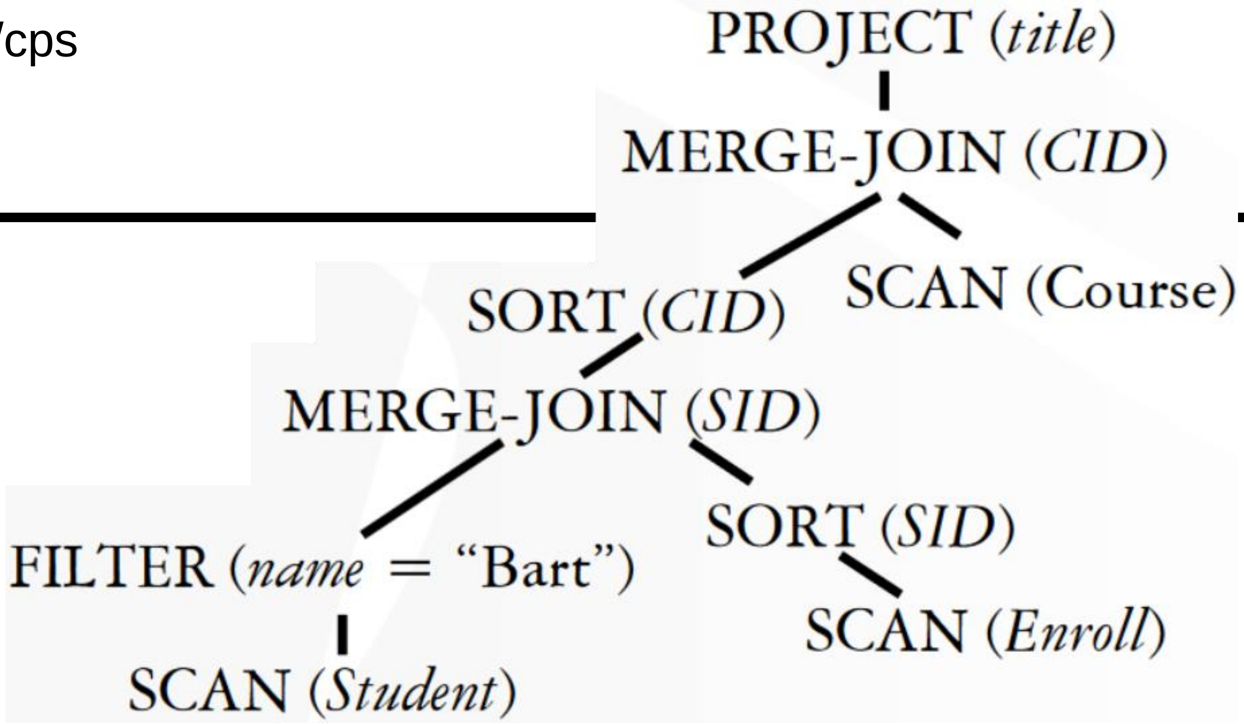
TableScan(U)

TableScan(R)

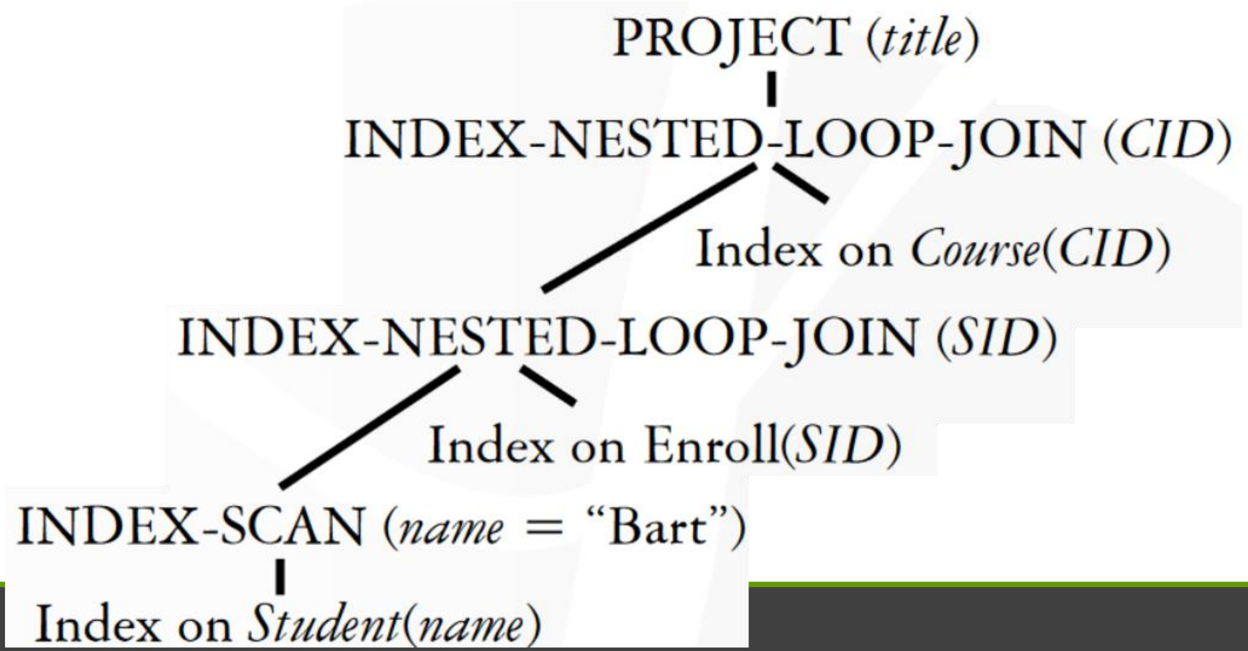
TableScan(S)

$R \bowtie S \bowtie U$

https://www.cs.duke.edu/courses/fall02/cps196.3/lectures/22-qp.pdf



- **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



Physical Optimization [物理优化]

- Makes use of execution cost analysis [based on statistics]:
  - examine query evaluation plan
  - determine efficient join sequences
  - select access method for each operation   (e.g. index for select)
  - for distributed DB, select best sites   (closest, best bandwidth)
  - determine total cost for evaluation plan
  - repeat for all possible plans and choose best
- Physical optimization is also called query evaluation plan generation.

- **Then physical optimization is carried out**
  - Among many pre-optimal PQPs
  - Based on catalog information

Tables

name	file	#tuples	size
Tables	…	6	1
Attributes	…	23	1
Views	…	1	1
Indexes	…	0	1
Sailors	…	40,000	500
Reserves	…	100,000	1000

Attributes

name	table	type	pos
name	Tables	string	1
file	Tables	string	2
#tuples	Tables	integer	3
size	Tables	integer	4
name	Attributes	string	1
table	Attributes	string	2
type	Attributes	string	3
pos	Attributes	integer	4
⋮	⋮	⋮	⋮
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	date	3
rname	Reserves	string	4

Views

name	text
Captains	SELECT * FROM Sailors WHERE…

Indexes

name	file	type	#keys	size
Boats	…	B+Tree	100	1

Query Evaluation Example

Example database schema (multi-campus university):

Employee(**eid**, ename, status, city)
Department(**dname**, city, address)
Subject(**sid**, sname, syllabus)
Lecture(**subj**, **dept**, **empl**, time)

Example database instance statistics:

Relation	*r*	*R*	*C*	*b*
Employee	1000	100	10	100
Department	100	200	5	50
Subject	500	95	10	100
Lecture	2000	100	10	200

Query: *Which depts in Sydney offer database subjects?*

```
select dname
from   Department D, Subject S, Lecture L
where  D.city = 'Sydney' and S.sname like '%Database%'
       and D.dname = L.dept and S.sid = L.subj
```

Additional information (needed to determine query costs):

- 5 departments are located in Sydney
- 80 subjects are about databases
- 300 lectures are on databases
- 100 lectures are in Sydney
- 3 of these are on databases

... Query Evaluation Example

Relation	r	R	C	b
Employee	1000	100	10	100
Department	100	200	5	20
Subject	500	95	10	100
Lecture	2000	100	10	200

Strategy #1 for answering query:

1. *TMP1* ← *Subject × Lecture*
2. *TMP2* ← *TMP1 × Department*
3. *TMP3* ← *Select[check](TMP2)*

check = (city='Sydney' & sname='Databases' & dname=dept & sid=subj)

4. *RESULT* ← *Project[dname](TMP3)*

Costs involved in using strategy #1:

Reln	r	R	C	b
TMP1	1000000	195	5	20000
TMP2	100000000	395	2	50000000
TMP3	3	395	2	2
RESULT	3	100	10	1

Total I/Os

$= Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4}$

$= (100+100*200+ 200000 + (20+20*20000+ 5*10^7 + 5*10^7)+2) + 2$

$= 100,440,124$

Strategy #2 for answering query:

1. *TMP1* ← *Join[sid=subj](Subject, Lecture)*
2. *TMP2* ← *Join[dept=dname](TMP1, Department)*
3. *TMP3* ← *Select[city='Sydney' & sname='Databases'](TMP2)*
4. *RESULT* ← *Project[dname](TMP3)*

Costs involved in using strategy #2:

Reln	*r*	*R*	*C*	*b*
TMP1	2000	195	5	400
TMP2	2000	395	2	1000
TMP3	3	395	2	2
RESULT	3	100	10	1

Total I/Os

$= Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4}$

$= (100+100*200+400) + (20+20*400+1000) + (1000+2) + 2$

$= 30,524$

Strategy #3 for answering query:

1. *TMP1* ← *Join[sid=subj](Subject, Lecture)*
2. *TMP2* ← *Select[sname='Databases'](TMP1)*
3. *TMP3* ← *Join[dept=dname](TMP2, Department)*
4. *TMP4* ← *Select[city='Sydney'](TMP3)*
5. *RESULT* ← *Project[dname](TMP4)*

Costs involved in using strategy #3:

Reln	*r*	*R*	*C*	*b*
TMP1	2000	195	5	400
TMP2	20	195	5	4
TMP3	20	395	2	10
TMP4	3	395	2	2
RESULT	3	100	10	1

Total I/Os

$= Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$

$= (100+100*200+400) + (400+4) + (4+4*20+10) + (10+2) + 1$

$= 21,011$

Strategy #4 for answering query:

1. *TMP1* ← *Select[sname='Databases'](Subject)*
2. *TMP2* ← *Select[city='Sydney'](Department)*
3. *TMP3* ← *Join[sid=subj](TMP1, Lecture)*
4. *TMP4* ← *Join[dept=dname](TMP3, TMP2)*
5. *RESULT* ← *project[dname](TMP4)*

Costs involved in using strategy #4:

Reln	*r*	*R*	*C*	*b*
TMP1	80	95	5	16
TMP2	5	200	5	1
TMP3	300	195	5	60
TMP4	3	395	2	2
RESULT	3	100	10	1

Total I/Os

$= Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$

$= (100+16) + (20+1) + (16+16*200+60) + (1+1*60+2) + 2$

$= 3478$

Strategy #5 for answering query:

1. *TMP1* ← *Select[sname='Databases'](Subject)*
2. *TMP2* ← *Select[city='Sydney'](Department)*
3. *TMP3* ← *Join[dept=dname](TMP2,Lecture)*
4. *TMP4* ← *Join[sid=subj](TMP3,TMP1)*
5. *RESULT* ← *project[dname](TMP4)*

... Query Evaluation Example

Costs involved in using strategy #5:

Reln	*r*	*R*	*C*	*b*
TMP1	80	95	5	16
TMP2	5	200	5	1
TMP3	100	295	3	34
TMP4	3	395	2	2
RESULT	3	100	10	1

Total I/Os

$= Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$

$= (100+16) + (20+1) + (1+1*200+34) + (16+16*34+2) + 2$

$= 936$

Chapter 3 B: How to get File operations for SQL really

- **My understanding about DBMS**
  - 3 stages before SQL is really executed
    - Parse tree → Logical Query Plan (**LQP**)/RA tree
    - Optimize LQP: Select optimal plan
      - Algebraic Optimization
    - LQP → Physical Query Plan (**PQP**)
      - Physical optimization
    - Execution of PQP
      - Materialize or pipeline
- **Advanced project in reading PostgreSQL**
  - Parse and Execute SQL in PostgreSQL

Query Execution: Just like the execution of math expression

- **The query execution engine is like a virtual machine that runs physical query plans**
  - Either **materialize** or **pipeline**

- **Two architectural considerations:**
  - Granularity of processing
  - Control flow:
    - Sequential
      - In-order Traversal, and one after another
    - Or
    - Concurrent
      - Cut into segments, and execute concurrently

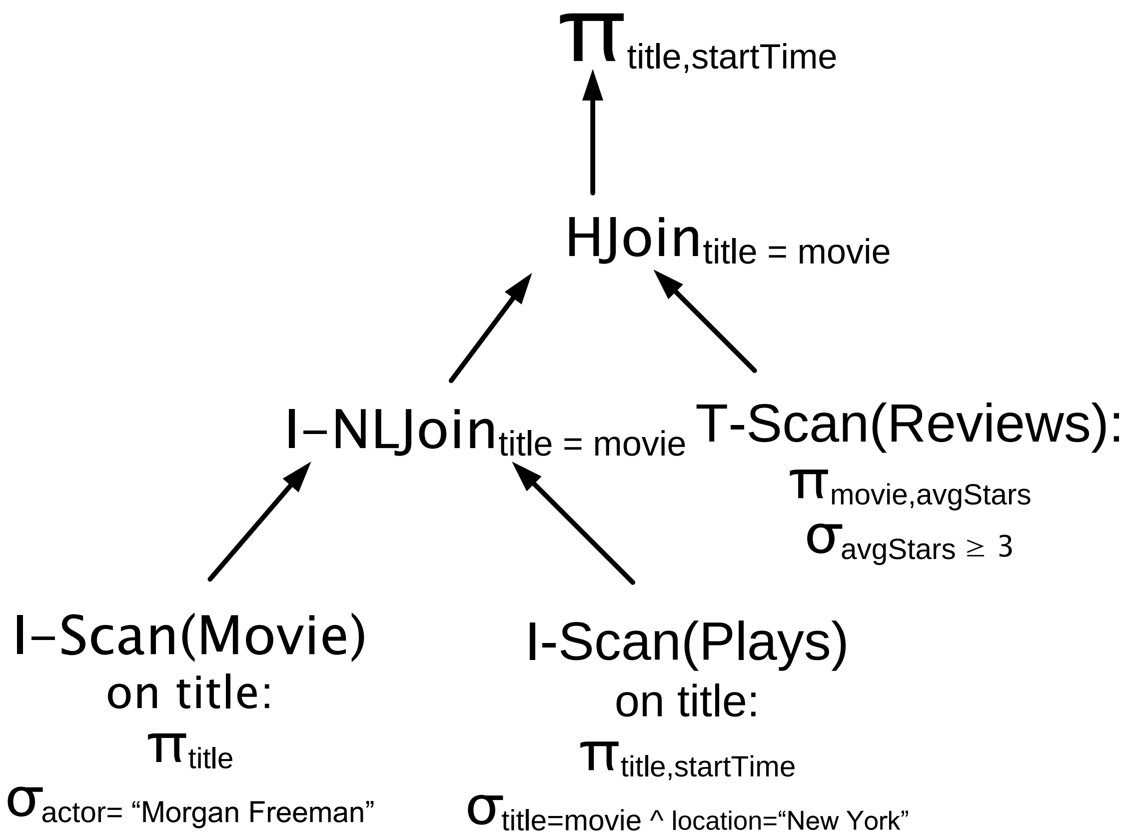
Granularity of Processing

Options for granularity:

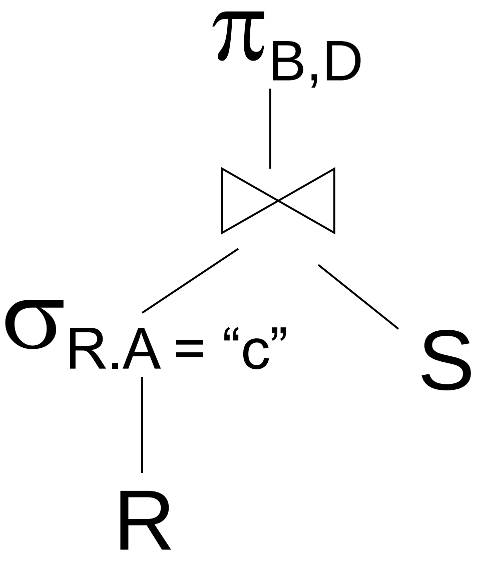
- Operators process one relation at a time
- Operators process one tuple at a time (**pipelining**) – better responsiveness and parallelism

Blocking algorithms can't pipeline output, e.g., GROUP BY over unsorted data

Some operators are partly blocking: e.g., hash join (HJoin) must buffer one input before joining



Operator Plumbing [水管工 – 组装…]

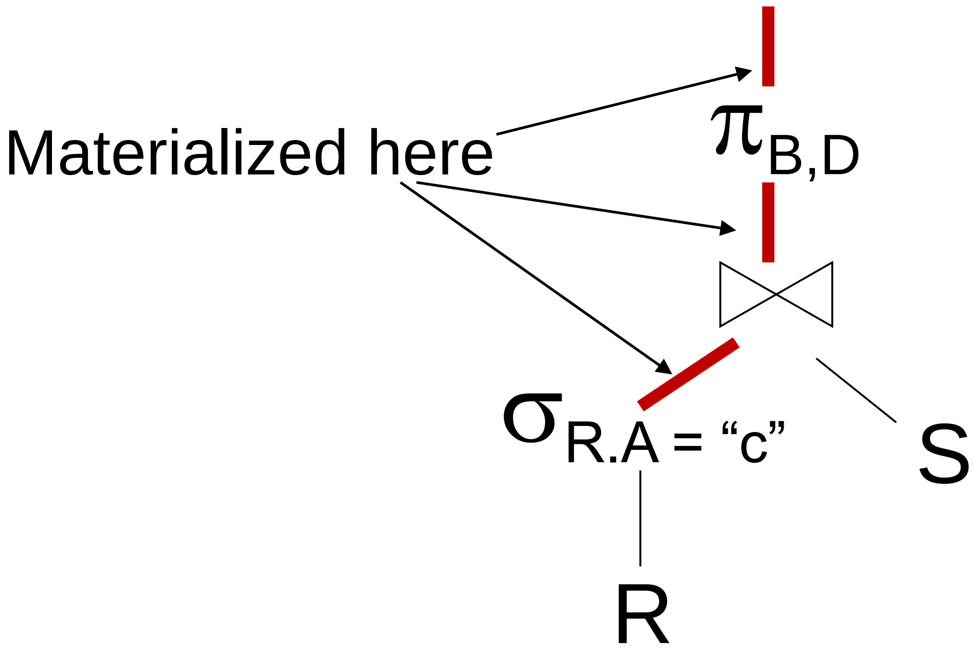


- **Materialization**:
  - output of one operator written to disk, next operator reads from the disk
  - perform operations in series and write intermediate results to disk
- **Pipelining**:
  - output of one operator directly fed to next operator

Two ways to transfer intermediate results among the operations

Materialization

- Produce intermediate result [中间结果]



https://www.cs.duke.edu/courses/fall08/cps216/Lectures/03_iterators_and_rewrites.ppt

Materialization Example:

- select s.name, s.id, e.course, e.mark
- from   Student s, Enrolment e
- where e.student = s.id and
-         e.semester = '05s2' and s.name = 'John';

might be executed as

Temp1 = BtreeSelect[semester=05s2](Enrolment)

Temp2 = **BtreeSelect**[name=John](Student)

-- indexes on name and semester

-- produce sorted Temp1 and Temp2

Temp3 = **SortMergeJoin**[e.student=s.id](Temp1,Temp2)

-- SMJoin especially effective, since

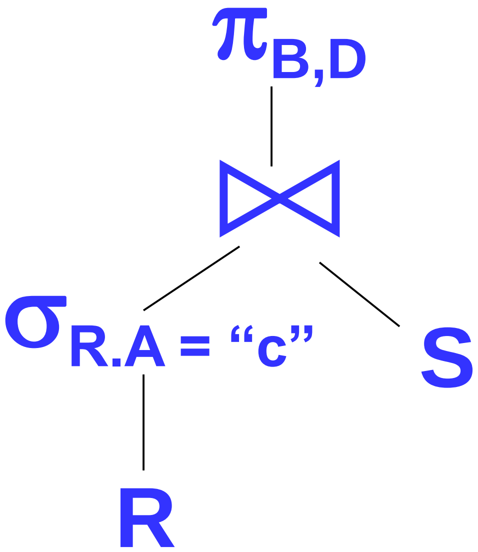
-- Temp1 and Temp2 are already sorted

Result = **Project**[name,id,course,mark](Temp3)

Pipelining

- How pipelining is organized between two operators:
  - blocks execute "concurrently" **as producer/consumer** pairs
    - first operator acts as producer; second as consumer
  - structured as interacting **iterators** (open; while(next); close)
- Advantage:
  - no requirement for disk access (results passed via memory buffers)
- Disadvantage:
  - each operator accesses inputs via linear scan

Iterators: Pipelining



➔ Each operator supports:

- Open()
- GetNext()
- Close()

➔ Parameters are those “List”s

https://www.cs.duke.edu/courses/fall08/cps216/Lectures/03_iterators_and_rewrites.ppt

Implementation of single-relation selection iterator:

```
typedef struct {
    File   inf;  // input file
    Cond   cond; // selection condition
    Buffer buf;  // buffer holding current page
    int    curp; // current page during scan
    int    curr; // index of current record in page
} Iterator;
```

Iterator structure contains information:

- related to operation being performed (e.g. `cond`)
- information giving current execution state (e.g. `curp`, `curr`)

Implementation of single-relation selection iterator (cont):

```
Iterator *open(char *relName, Condition cond) {
    Iterator *iter = malloc(sizeof(Iterator));
    iter->inf  = openFile(fileName(relName),READ);
    iter->cond = cond;
    iter->curp = 0;
    iter->curr = -1;
    readBlock(iter->inf, iter->curp, iter->buf);
    return iter;
}
void close(Iterator *iter) {
    closeFile(iter->inf);
    free(iter);
}
```

Implementation of single-relation selection iterator (cont):

```
Tuple next(Iterator *iter) {
    Tuple rec;
    do {
        // check if reached end of current page
        if (iter->curr == nRecs(iter->buf)-1) {
            // check if reached end of data file
            if (iter->curp == nBlocks(iter->inf)-1)
                return NULL;
            iter->curp++;
            iter->buf = readBlock(iter->inf, iter->curp);
            iter->curr = -1;
        }
        iter->curr++;
        rec = getRec(iter->buf, iter->curr);
    } while (!matches(rec, iter->cond));
    return rec;
}
// curp and curr hold indexes of most recently read page/record
```

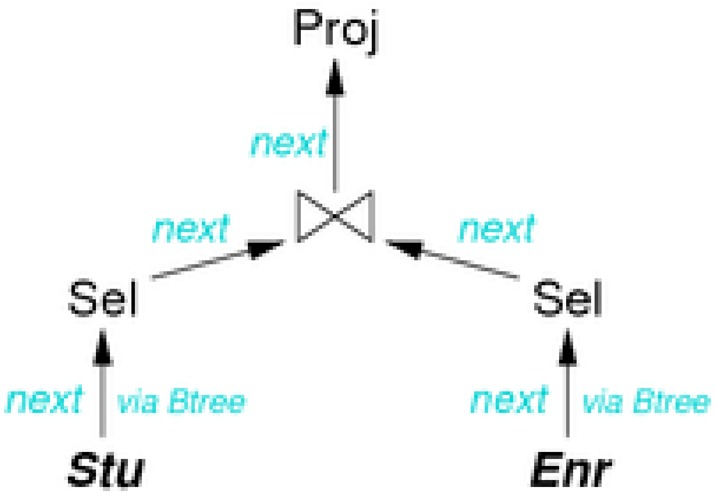
Consider the query:

```
select  s.id,  e.course,  e.mark
from    Student s,  Enrolment e
where   e.student = s.id and
        e.semester = '05s2' and s.name = 'John';
```

which maps to the RA expression

Proj[id,course,mark](Join[student=id](Sel[05s2](Enr),Sel[John](Stu)))

Modelled as communication between RA tree nodes:



This query might be executed as

```
System:
    iter0 = open(Result)
    while (Tup = next(iter0)) { display Tup }
    close(iter0)
Result:
    iter1 = open(Join)
    while (T = next(iter1))
        { T' = project(T); return T' }
    close(iter1)
Sel1:
    iter4 = open(Btree(Enrolment,'semester=05s2'))
    while (A = next(iter4)) { return A }
    close(iter4)
```

```
Join: -- nested-loop join
    iter2 = open(Sel1)
    while (R = next(iter2) {
        iter3 = open(Sel2)
        while (S = next(iter3))
            { if (matches(R,S) return (RS) }
        close(iter3) // better to reset(iter3)
    }
    close(iter2)
Sel2:
    iter5 = open(Btree(Student,'name=John'))
    while (B = next(iter5)) { return B }
    close(iter5)
```

Control Flow

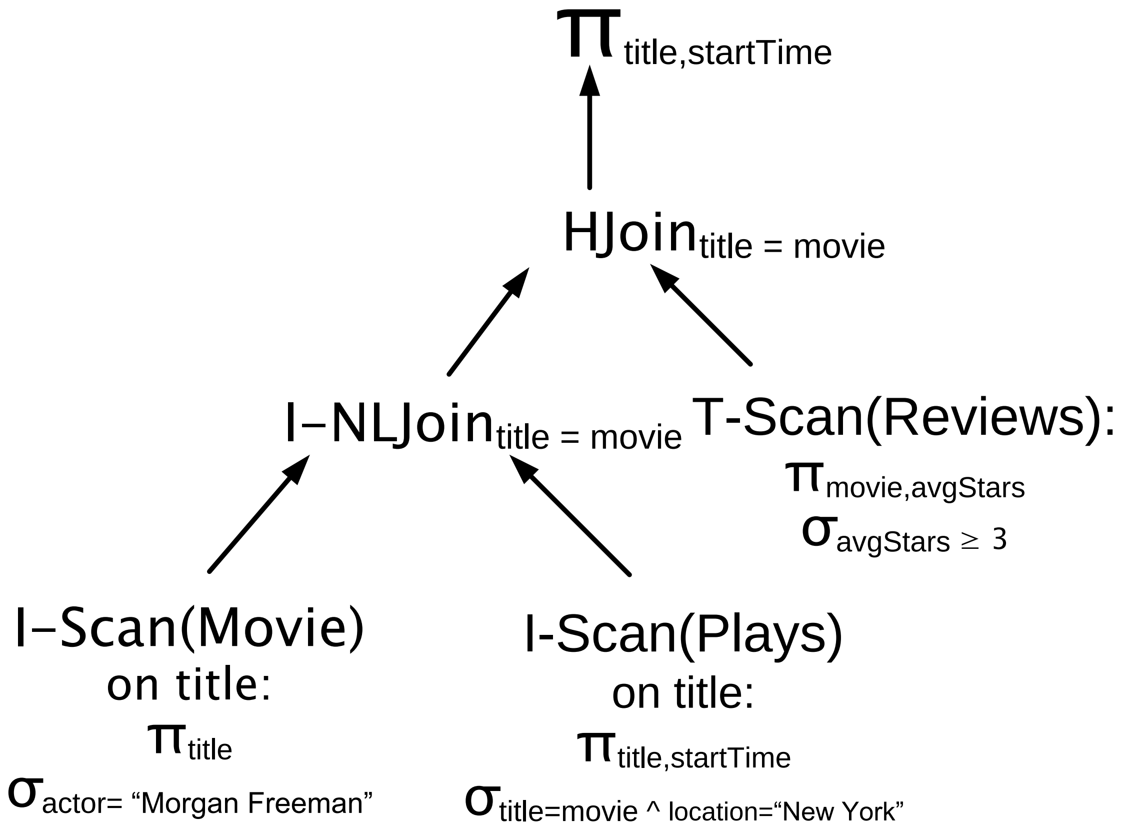
Options for control flow:

- **Iterator-based – execution begins at root**
  - *open, next, close*
  - Propagate calls to children

    May call multiple child *nexts*

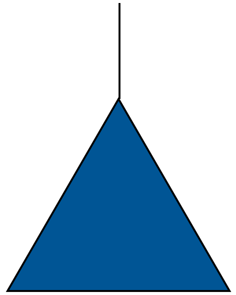
- **Dataflow-driven – execution driven by tuple arrival at leaves**

- **Hybrids of the two**



Iterator for Select

$\sigma_{R.A = \text{“c”}}$

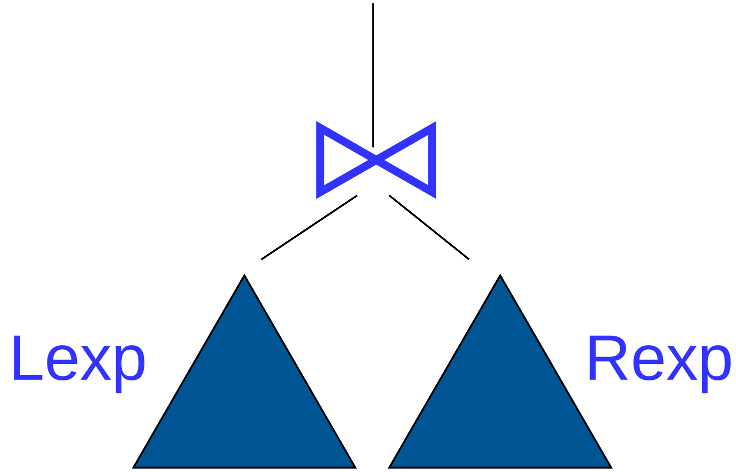


```
Open() {
  /** initialize child */
  Child.Open();
}

Close() {
  /** inform child */
  Child.Close();
}
```

```
GetNext() {
 LOOP:
     t = Child.GetNext();
     IF (t == EOT) {
       /** no more tuples */
       RETURN EOT;
     }
     ELSE IF (t.A == “c”)
       RETURN t;
 ENDLOOP:
}
```

Iterator for Tuple Nested Loop Join (TNLJ)



- TNLJ (conceptually)

  for each r $\in$ Lexp do

  for each s $\in$ Rexp do

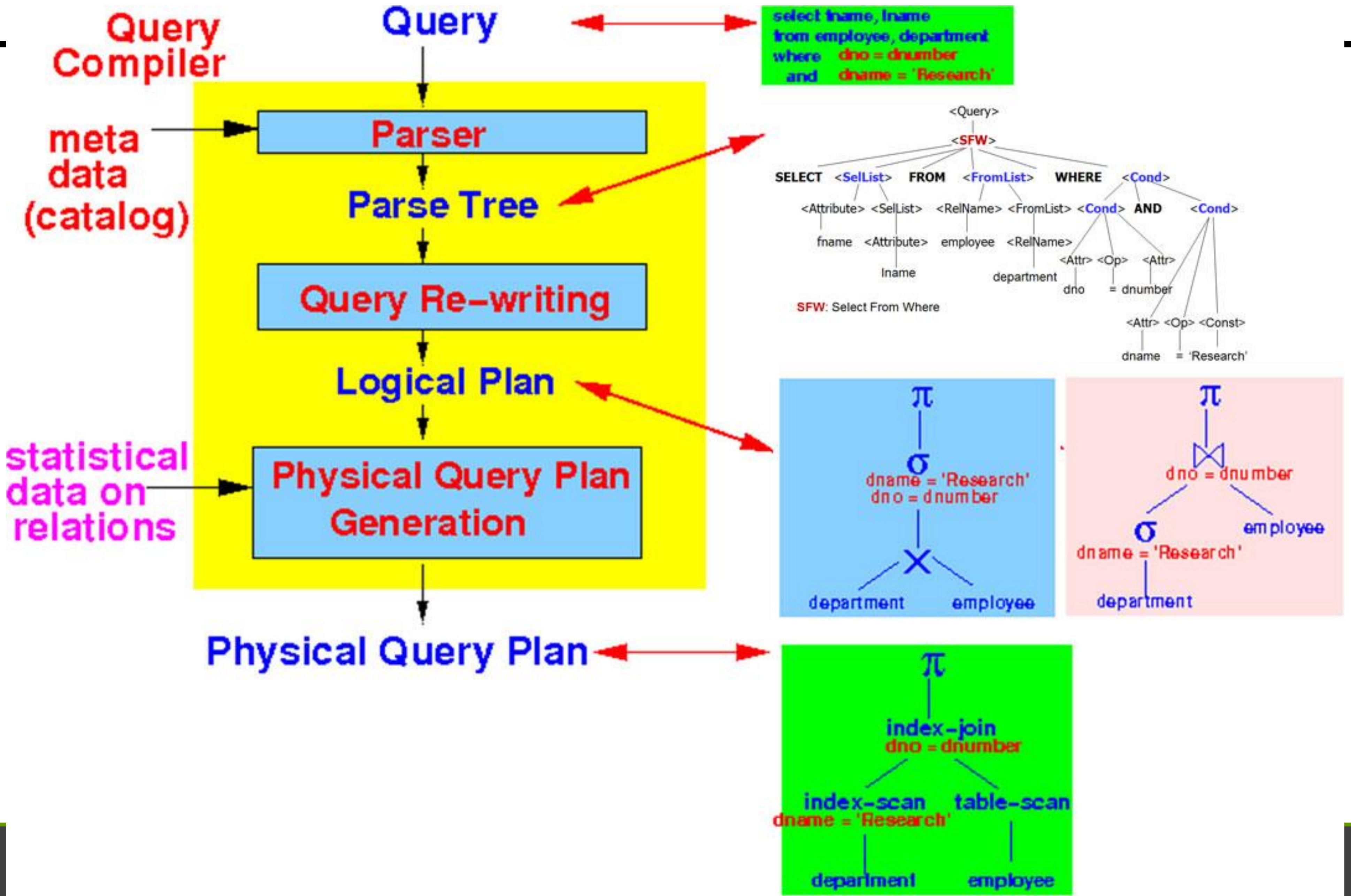
  if Lexp.C = Rexp.C, output r,s

https://www.cs.duke.edu/courses/fall08/cps216/Lectures/03_iterators_and_rewrites.ppt

In short

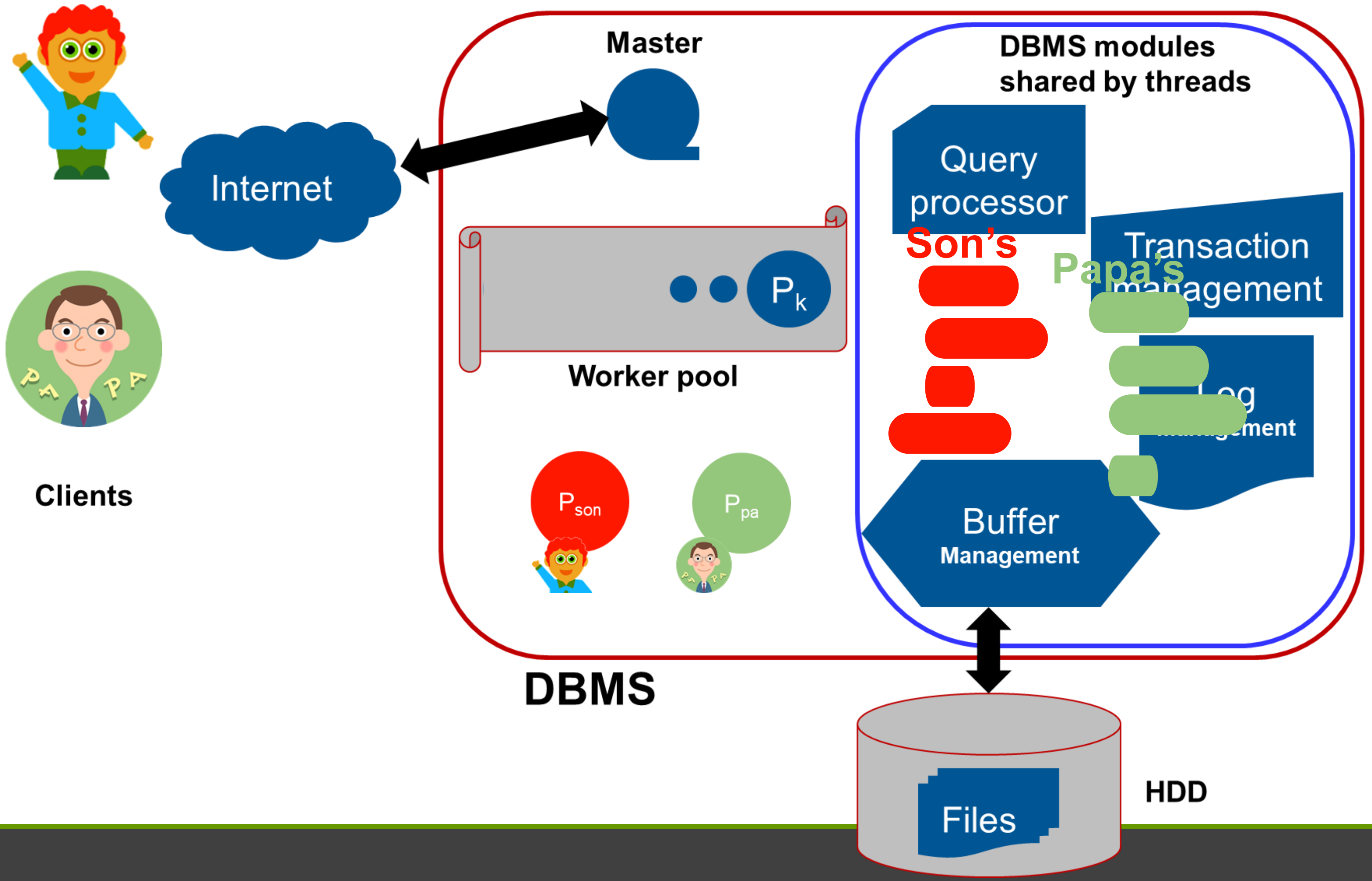
- **Math Expression → AST**
  - Helpful to understand SQL processing
- **SQL Processing**
  - SQL → Parse Tree [Math → AST]
  - Parse Tree → Canonical RA [Tree Traverse]
    - Canonical RA is converted into equivalent RAs (for Algebraic optimization)
  - RA → PQP (Physical Query Plan) [Math → AST]
    - RA could be mapped to many PQPs
    - With help of catalog information, physical optimization
  - The optimal PQP is executed [Tree Traverse][Iterator]
    - Materialize or Pipeline

In short


Query Compiler
Query
select fname, lname
from employee, department
where dno = dnumber
and dname = 'Research'
meta data (catalog)
Parser
Parse Tree
<Query>
<SFW>
SELECT <SelList> FROM <FromList> WHERE <Cond>
<Attribute> <SelList> <RelName> <FromList> <Cond> AND <Cond>
fname <Attribute> employee <RelName>
<Attr> <Op> <Attr>
lname
department
dno = dnumber
SFW: Select From Where
<Attr> <Op> <Const>
dname = 'Research'
Query Re-writing
Logical Plan
π
σ dname = 'Research' dno = dnumber
×
department
employee
π
dno = dnumber
σ dname = 'Research'
employee
department
statistical data on relations
Physical Query Plan Generation
Physical Query Plan
π
index-join dno = dnumber
index-scan dname = 'Research'
table-scan
department
employee

Now we get the corresponding file operations
SQLs → File Operation sequences

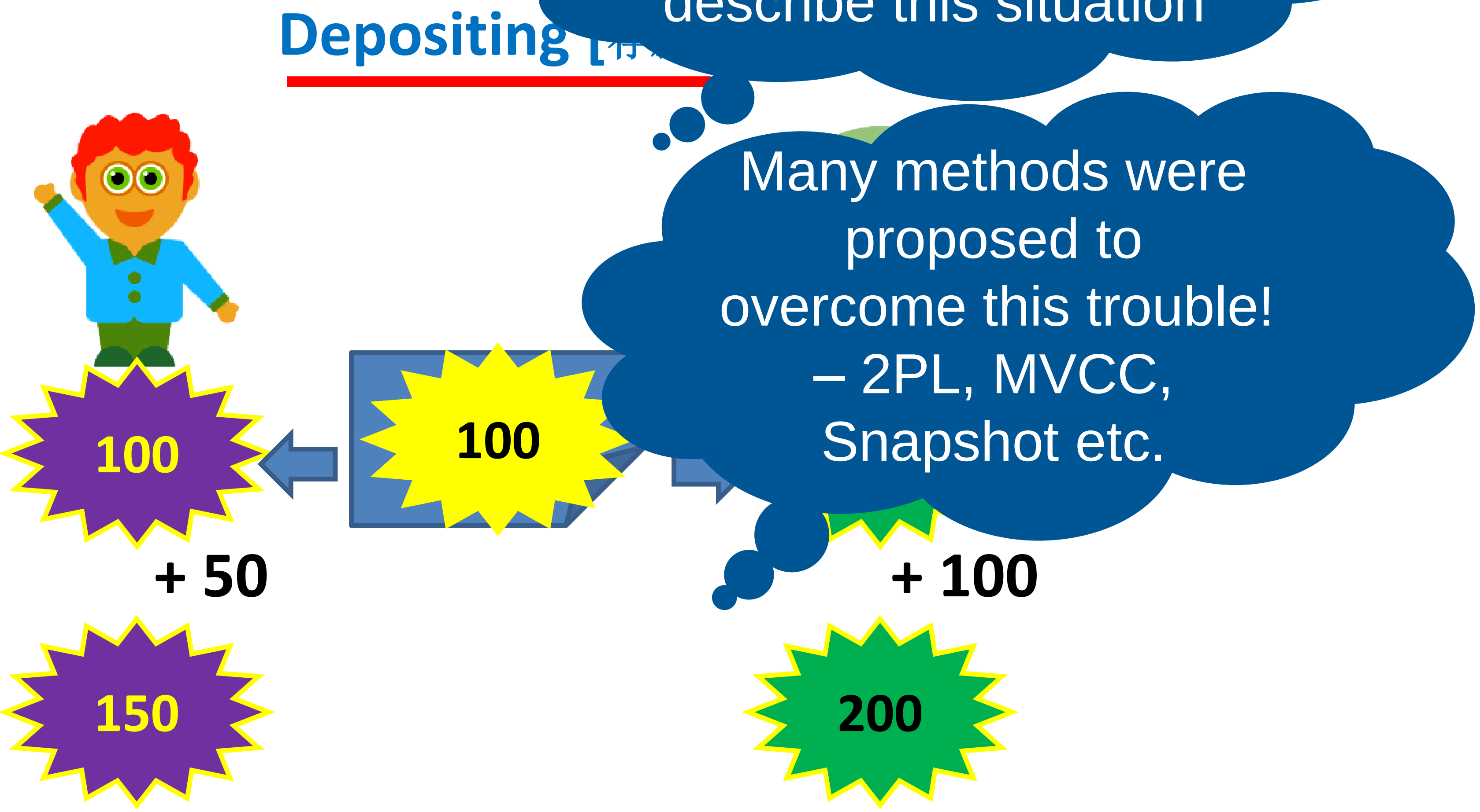
- But file operations of SQLs from concurrent clients/users could be **interleaved**



Data Inconsistency!

Transaction [事务] concept is used to describe this situation

Many methods were proposed to overcome this trouble! – 2PL, MVCC, Snapshot etc.



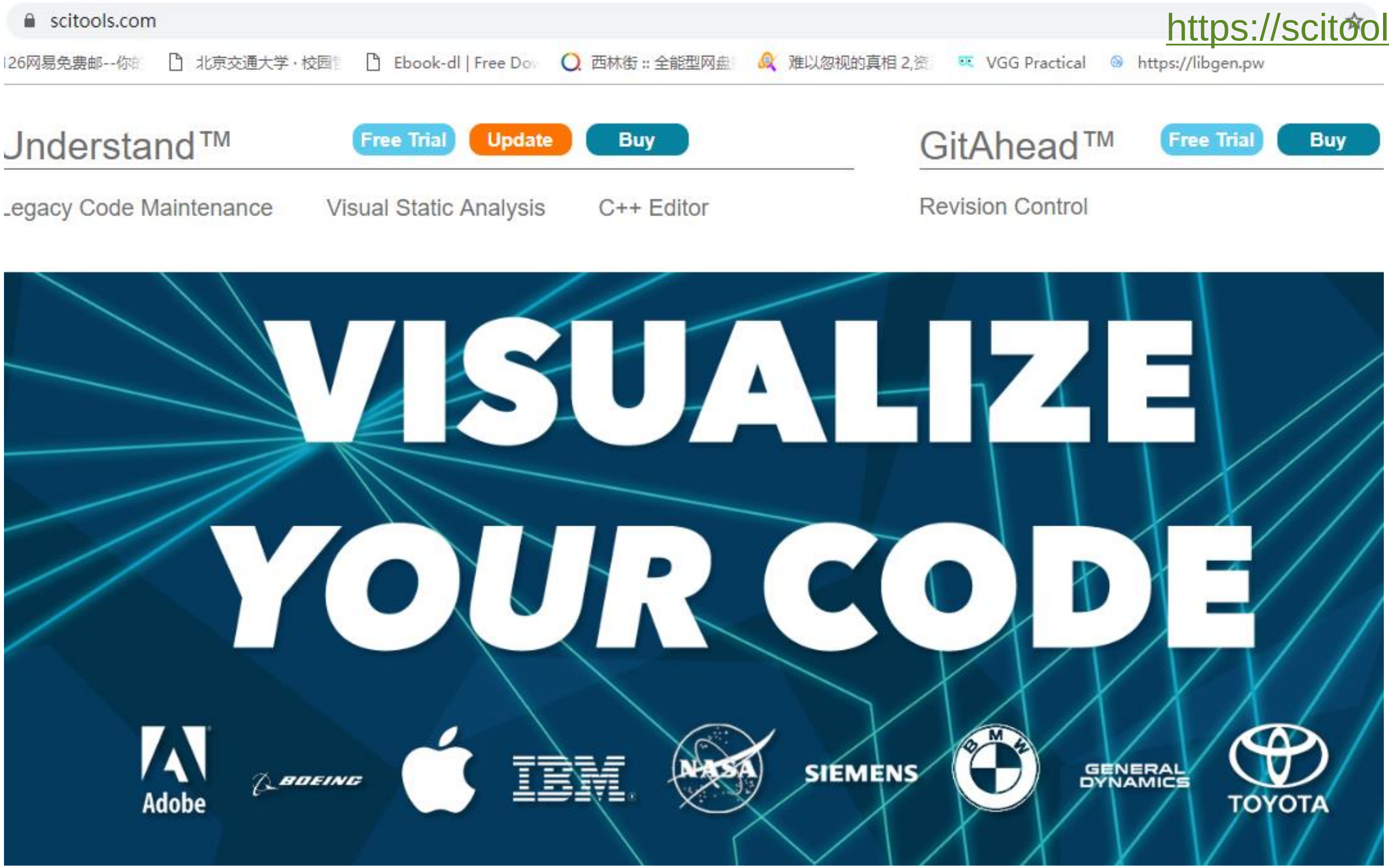
Chapter 3 B:
How to get File operations for SQL really

- **My understanding about DBMS**
  - 3 stages before SQL is really executed
    - Parse tree → Logical Query Plan (**LQP**)/RA tree
    - Optimize LQP: Select optimal plan
      - Algebraic Optimization
    - LQP → Physical Query Plan (**PQP**)
      - Physical optimization
    - Execution of PQP
      - Materialize or pipeline
- **Advanced project in reading HyperSQL**
  - Parse and Execute SQL in HyperSQL

- **Distill the workflow of SQL processing with an example**
  - Major classes/functions
  - + Data Structure
  - + Relationships of function calls
  - + your understanding about the mechanism

Reading codes – Understand

https://scitools.com/



Chapter 60

CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES

Handbook of DATA STRUCTURES and APPLICATIONS