THE BASICS OF QUERY PROCESSING

A DW designer must understand the principles and methods of query processing in order to produce better BI applications.

SQL is a declarative rather than a procedural language.

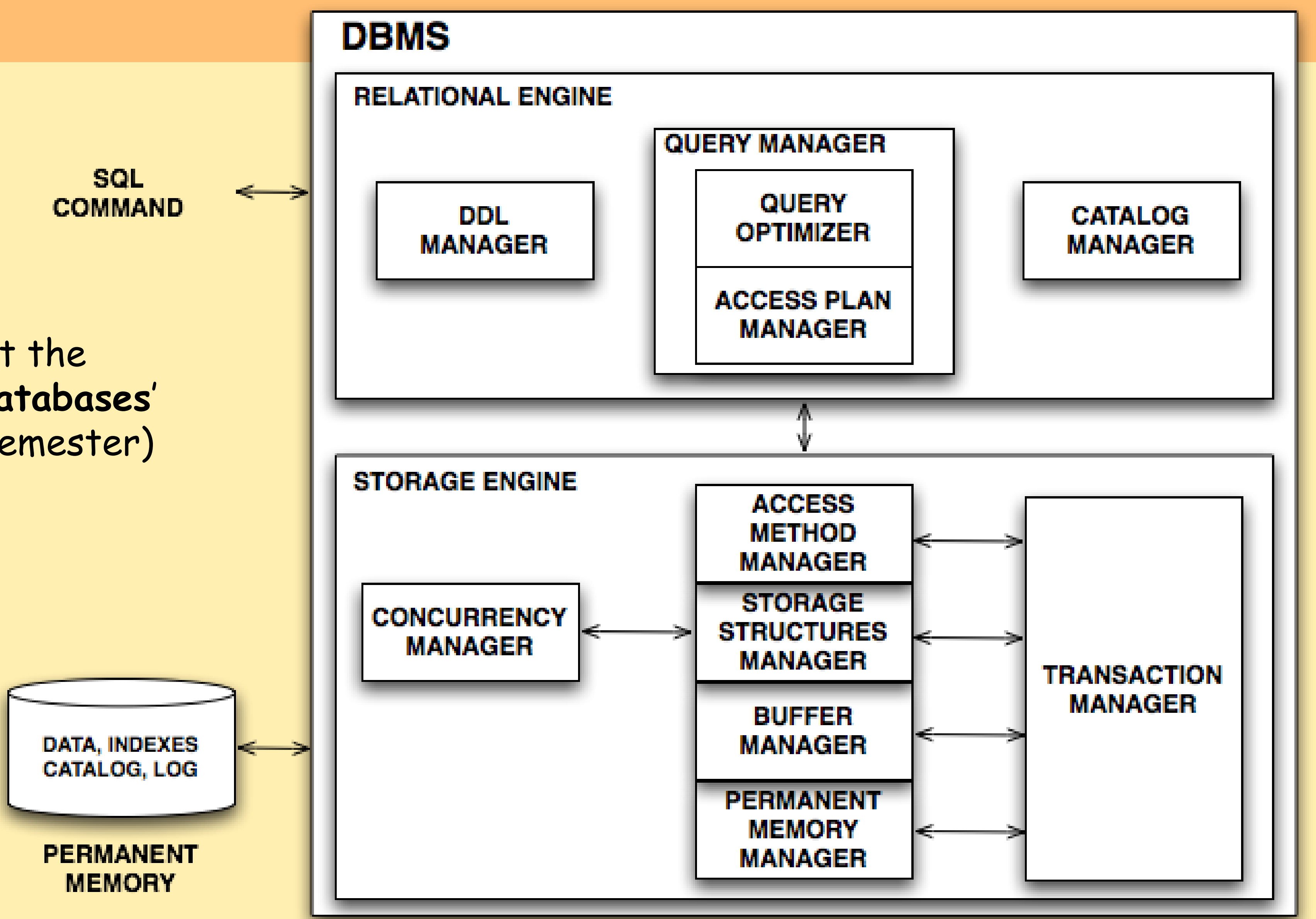
It describes **WHAT** we are looking for, but not how to get it.

Relational Algebra describes **HOW** to get results
with "logical query plan" of relational operators.

A naive way to evaluate an expression would be to compute the results of the relational operators directly as specified.

DBMS ARCHITECTURE

Details at the '**Advanced Databases**' course (2^{nd} semester)



THE BASICS OF QUERY PROCESSING

The **query optimizer** chooses an appropriate algorithm to execute the query expressed as "**physical query plan**", composed of a few basic **physical operators**, which **implement an algorithm** to compute each relational operator.

Several alternative implementation techniques exist for each relational operator.

STORAGE STRUCTURES and RIDs

For simplicity, let us assume that:

A table is stored in a Heap File

a file for each table, with tuples stored in the insertion order

When a record is stored in a database, it is identified

internally by a **record identifier (RID).**

A RID has the property

that identify **the disk address of the page containing the record**.

Table

StudCode	City	BirthYear
100	MI	2002
101	PI	2000
102	PI	2001
104	FI	2000
106	MI	2000
107	PI	2002

Indexes can be defined on attributes of a table.

WHAT IS AN INDEX?

An index is a mapping of attribute(s) (key) values to RID of records.

Definition. An index I on an attribute K of a relational table F is an **ordered table** I(K, RID), with (|I| = |F|). A tuple of the index is a pair **(K :=k_i, RID := r_i)**, where **k_i** is a (key) value for a record, and **r_i** is a **reference** (RID) to the corresponding record.

EXAMPLES

CREATE UNIQUE INDEX PK_StudCode
ON Students (StudCode)
CREATE INDEX S_BirthYear
ON Students (BirthYear)

Students

RID	StudCode	City	BirthYear
1	100	MI	2002
2	101	PI	2000
3	102	PI	2001
4	104	FI	2000
5	106	MI	2000
6	107	PI	2002

```
SELECT *
FROM Students
WHERE BirthYear = 2001
```

Indexes

StudCode	RID
100	1
101	2
102	3
104	4
106	5
107	6

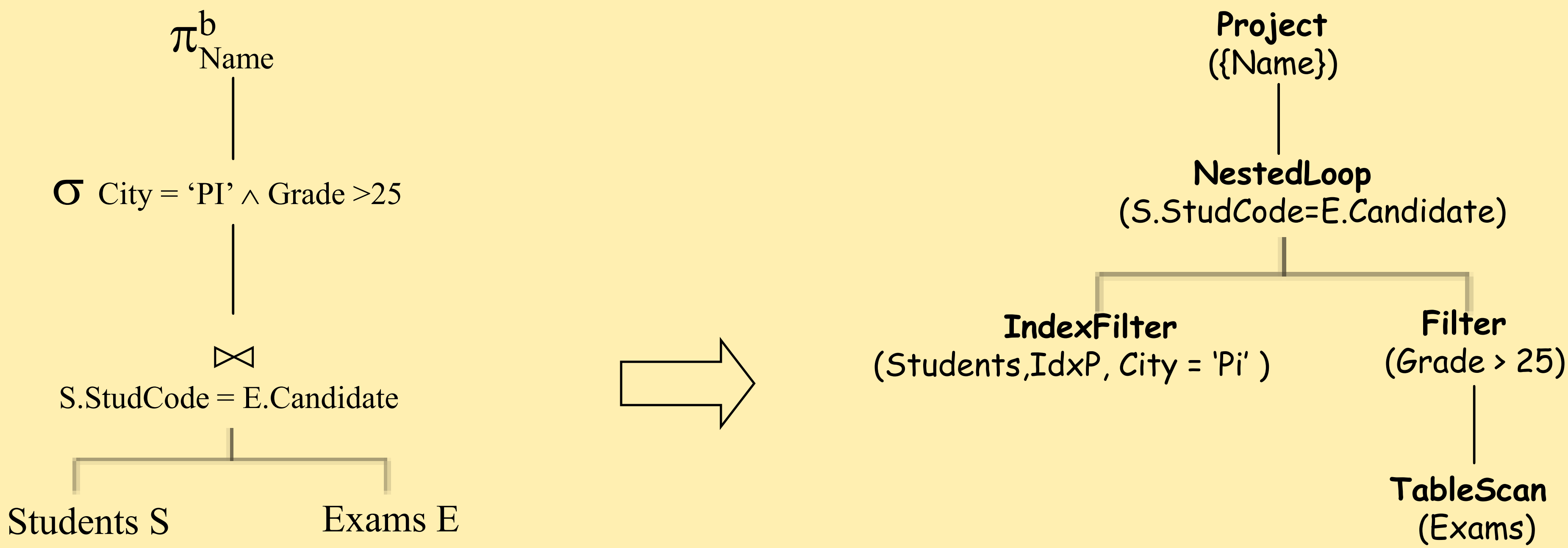
Index on **StudCode**

BirthYear	RID
2000	2
2000	4
2000	5
2001	3
2002	1
2002	6

Index on **BirthYear**

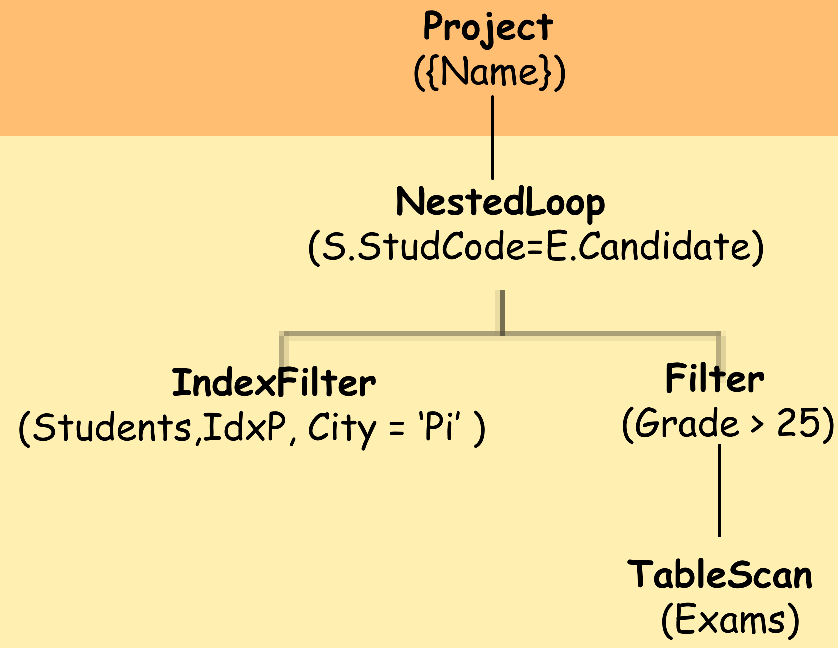
QUERY EXECUTION STEPS

```
SELECT  Name
FROM    Students S, Exams E
WHERE   S.StudCode = E.Candidate AND City='PI' AND Grade>25
```



LOGICAL PLAN **PHYSICAL (ACCESS) PLAN**

PHYSICAL QUERY PLAN EXECUTION



What is the result of a physical operator?

Materialization vs **pipelining**?

Each operator is typically implemented as an iterator using a 'pull' interface: when an operator is 'pulled' for the next output record, it 'pulls' on its inputs and computes them.

The interface provide methods **open**, **next**, **isDone**, and **close**.

QUERY EXECUTION

```
# SQL COMMAND Q ANALYSIS
SQLCommand parseTree = Parser.parseStatement(Q)

# COMMAND CHECK
Type type = parseTree.check()

# QUERY OPTIMIZATION
Value accessPlan = parseTree.Optimize()

# ACCESS PLAN EXECUTION
accessPlan.open();
while not accessPlan.isDone():
      Record rec = accessPlan.next()
      print(rec)
accessPlan.close()
```

IMPLEMENTATION OF RELATIONAL OPERATIONS

We will consider how to implement:

- **Projection**
- **Selection** (Restriction)
- **Group by**
- **Join**

Java Relational System (JRS) physical operators

PHYSICAL OPERATORS FOR TABLES AND SORT

Operators for R :

TableScan (R): to scan R;

SortScan (R, $\{A_i\}$): to scan R sorted on the $\{A_i\}$;

Operator to sort (τ):

Sort (O, $\{A_i\}$): to sort records of the operand **O** on the $\{A_i\}$;

EXAMPLE

```
SELECT   *
FROM     R ;
```

R ⇒ TableScan (R)

LOGICAL PLAN — PHYSICAL PLAN

```
SELECT   *
FROM     R
ORDER BY A ;
```

$\tau_{\{A\}}$
|
R

⇒ SortScan (R, {A})

⇒ Sort ({A})
|
TableScan (R)

PHYSICAL OPERATORS FOR δ , π^b

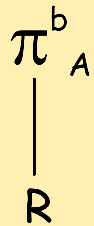
Project (O, $\{A_i\}$): to project the records of O without duplicates elimination;

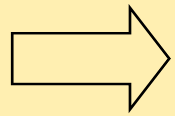
Distinct (O): **to eliminate duplicated** from **sorted** records of O;

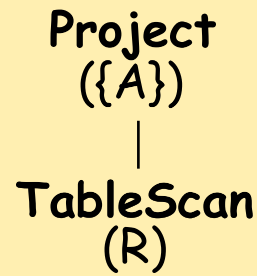
HashDistinct(O): **to eliminate duplicated** from records of O;

EXAMPLE

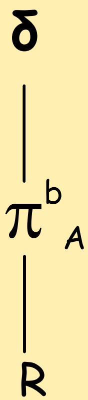
```
SELECT   A
FROM     R ;
```

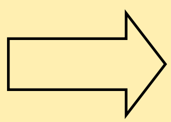


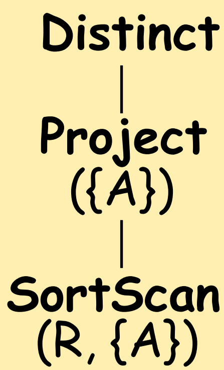


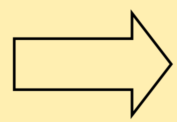


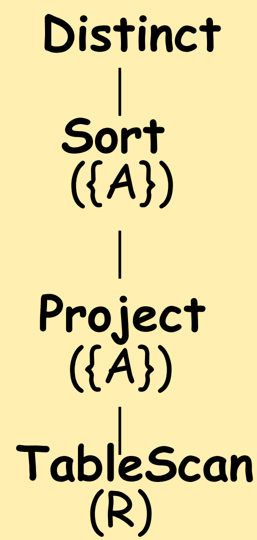
```
SELECT   DISTINCT A
FROM     R ;
```











PHYSICAL OPERATORS FOR σ

Filter (O, ψ): selection of the records of O;

The selection operator applied to a **relation** can be implemented with an **index.**

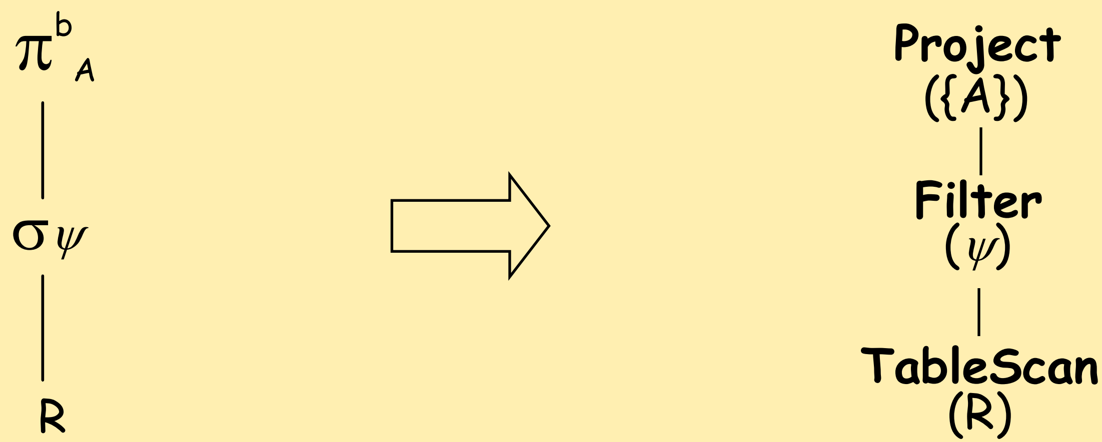
IndexFilter (R, Idx, ψ): selection with an index Idx on the ψ attributes of the records of R;

= **RidIndexFilter**(Idx, ψ): to retrieve the RIDs from an index

\+ **TableAccess**(O, R), to retrieve records from R using the RID in O;

1) PHYSICAL PLAN EXAMPLE: SFW

SELECT *A*
FROM R
WHERE *A* **BETWEEN** 50 **AND** 100;



LOGICAL PLAN **PHYSICAL PLAN**

ψ = *A* **BETWEEN** 50 **AND** 100

2) PHYSICAL PLAN EXAMPLE: SFW WITH INDEX

```
SELECT  *
FROM    R
WHERE   A BETWEEN 50 AND 100;
```

Idx an index on A

σ_{ψ}

R

⇒

IndexFilter
(R, Idx, ψ)

LOGICAL PLAN **PHYSICAL PLAN**

ψ = A **BETWEEN** 50 **AND** 100

EXERCISES

```
SELECT   A, B
FROM     R
WHERE    (A BETWEEN 50 AND 100) AND B > 20;
```

Idx an index on A

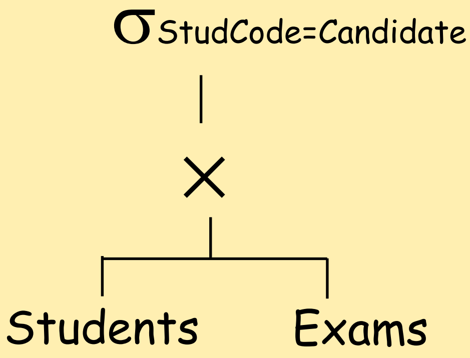
```
SELECT   A, B
FROM     R
WHERE    (A BETWEEN 50 AND 100) AND B > 20
ORDER BY A;
```

Idx an index on A, B

```
SELECT  *
FROM    Students S, Exams E
WHERE   S.StudCode = E.Candidate
```

Simple, but it must be carefully optimized :

(Students x Exams) is large; so



is inefficient.

NESTED LOOPS

```
foreach  r in R do
  foreach  s in S do
    if r.r1 = s.s1  then
       add <r, s> to result
```

$$R(r1,\ r2) \bowtie_{r1=s1} S(s1,\ s2)$$

R

r_1	r_2
2	a
3	d
4	a
8	c

S

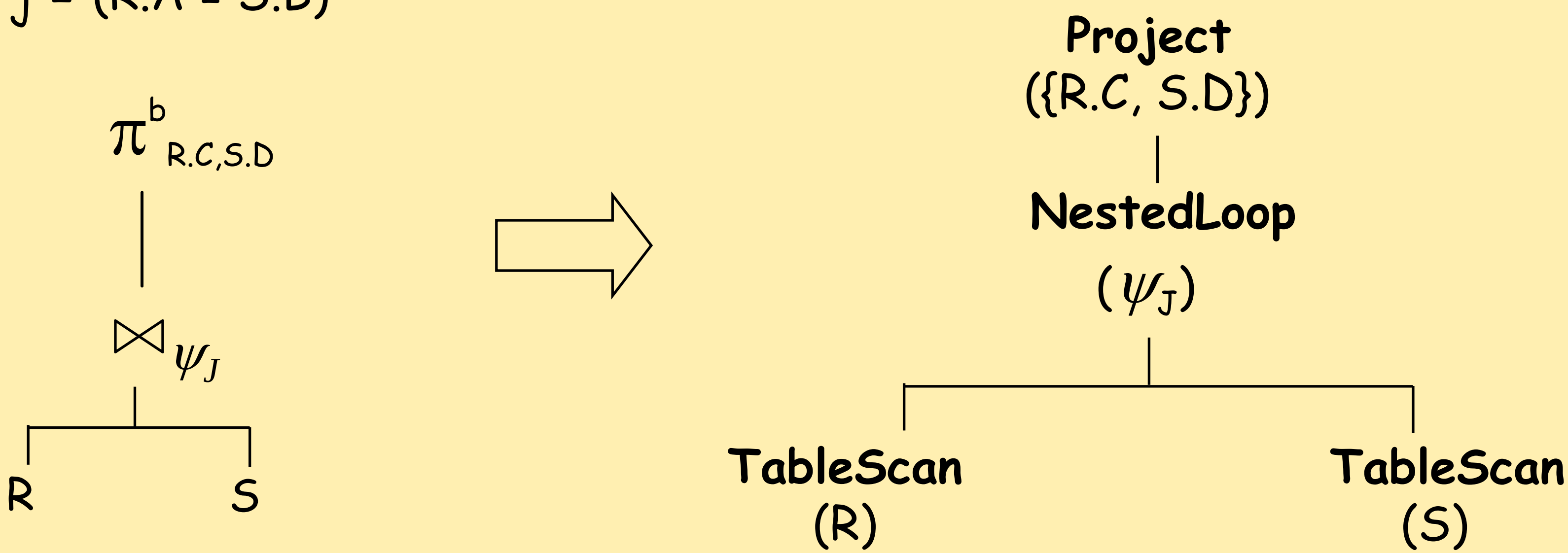
s_1	s_2
2	a
4	d
2	a
8	c
8	c
4	d
3	c

EXAMPLE: JOIN PHYSICAL PLAN

NestedLoop (O_E,O_I, ψ_J): join with nested loop and ψ_J as join condition;

```
SELECT  R.C, S.D
FROM    R , S
WHERE   R.A = S.B;
```

ψ_J = (R.A = S.B)



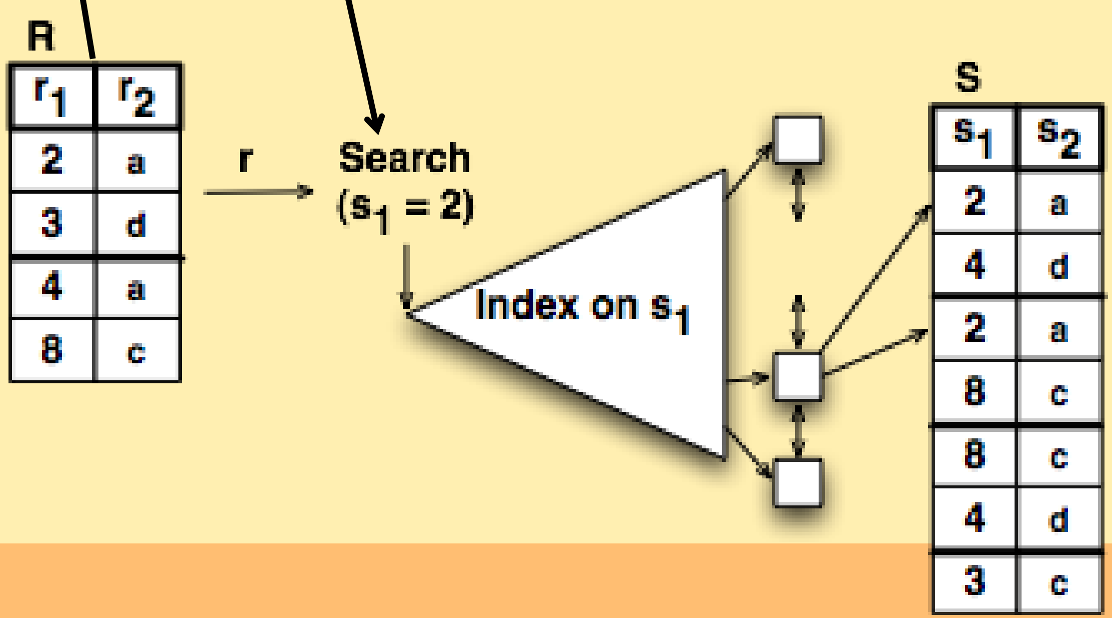
LOGICAL PLAN **PHYSICAL PLAN**

ANOTHER ALGORITHM

Index Nested loop : $R(r1, r2) \bowtie_{r1=s1} S(s1, s2)$

Hyp: There is an **index on the join column s1 of the internal relation** (S)

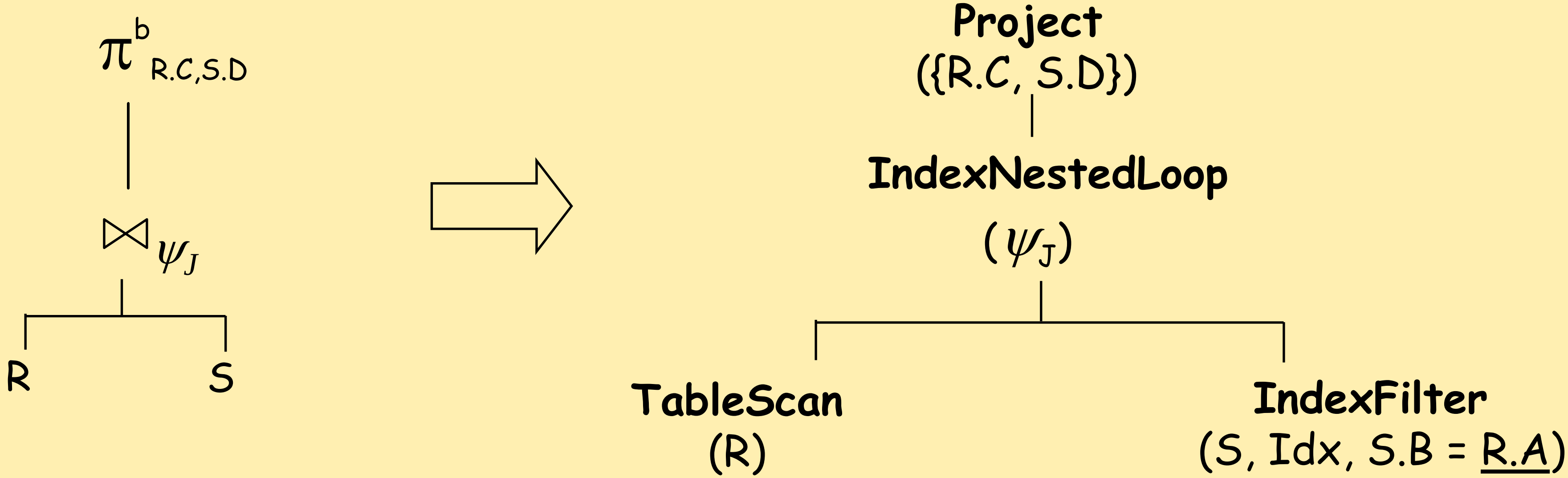
```
foreach r in R do
        foreach  s in S where s1 = r.r1 do
                add <r, s> to result
```



EXAMPLE: JOIN PHYSICAL PLAN WITH AN INDEX

SELECT R.C, S.D
FROM R , S
WHERE R.A = S.B;   Idx an index on S.B

$\psi_J = (R.A = S.B)$



LOGICAL PLAN   **PHYSICAL PLAN**

OTHER JOIN ALGORITHMS EXIST: MERGEJOIN, HASHJOIN,...

PHYSICAL OPERATORS FOR $\bowtie_{\psi_J}$

```
foreach r in R do
        foreach  s in S where s1 = r.r1 do
                add <r, s> to result
```

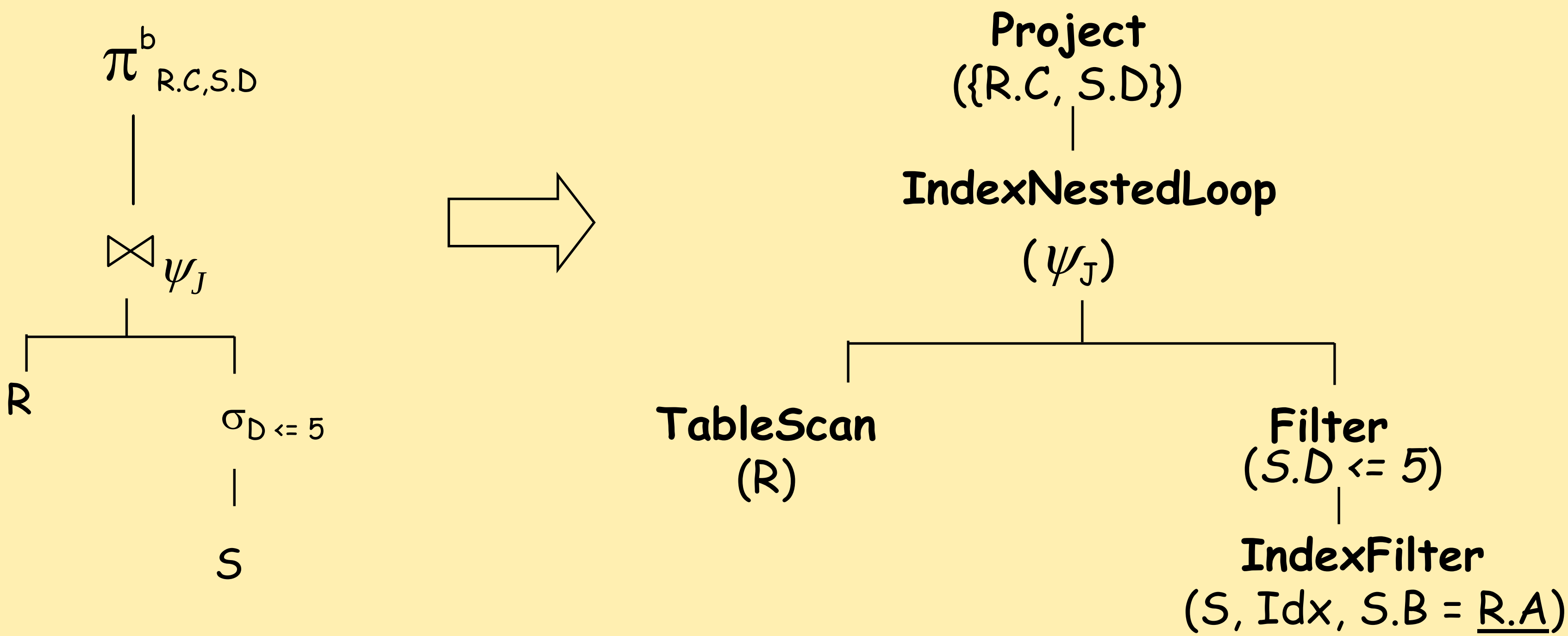
IndexNestedLoop (O_E,O_I, ψ_J): join with index nested loop. The **inner operand** O_I is an **IndexFilter**(R, Idx, ψ_J) or **Filter** (O, ψ_J) **with** O an **IndexFilter**(R, Idx, ψ ').

EXAMPLE: JOIN PHYSICAL PLAN WITH AN INDEX

SELECT R.C, S.D
FROM R , S
WHERE R.A = S.B AND S.D <= 5;

Idx an index on S.B

ψ_J = (R.A = S.B)



LOGICAL PLAN

PHYSICAL PLAN

PHYSICAL OPERATORS FOR γ

GroupBy $(O, \{A_i\}, \{f_i\})$: to group the **sorted** records of O on the $\{A_i\}$ using the aggregation function in $\{f_i\}$.

- The operator returns records with attributes the $\{A_i\}$ and the functions in $\{f_i\}$.
- The records of O are **sorted** on the $\{A_i\}$;

HashGroupBy $(O, \{A_i\}, \{f_i\})$

PHYSICAL PLAN WITH GROUP BY

```
SELECT     A, SUM(B)
FROM       R
WHERE      A BETWEEN 50 AND 100
GROUP BY A
HAVING   COUNT(*) > 1;
```

$\pi^{b}_{A,SUM(B)}$

|

$\sigma_{COUNT(*) > 1}$

|

${}_{A}\gamma_{SUM(B),\ COUNT(*)}$

|

$\sigma_{A\ BETWEEN\ 50\ AND\ 100}$

|

R

LOGICAL PLAN

Project
({A, SUM(B)})

|

Filter
(COUNT(*) > 1)

|

GroupBy
({A}, {SUM(B),COUNT(*)})

|

Sort
({A})

|

Filter
(A BETWEEN 50 AND 100)

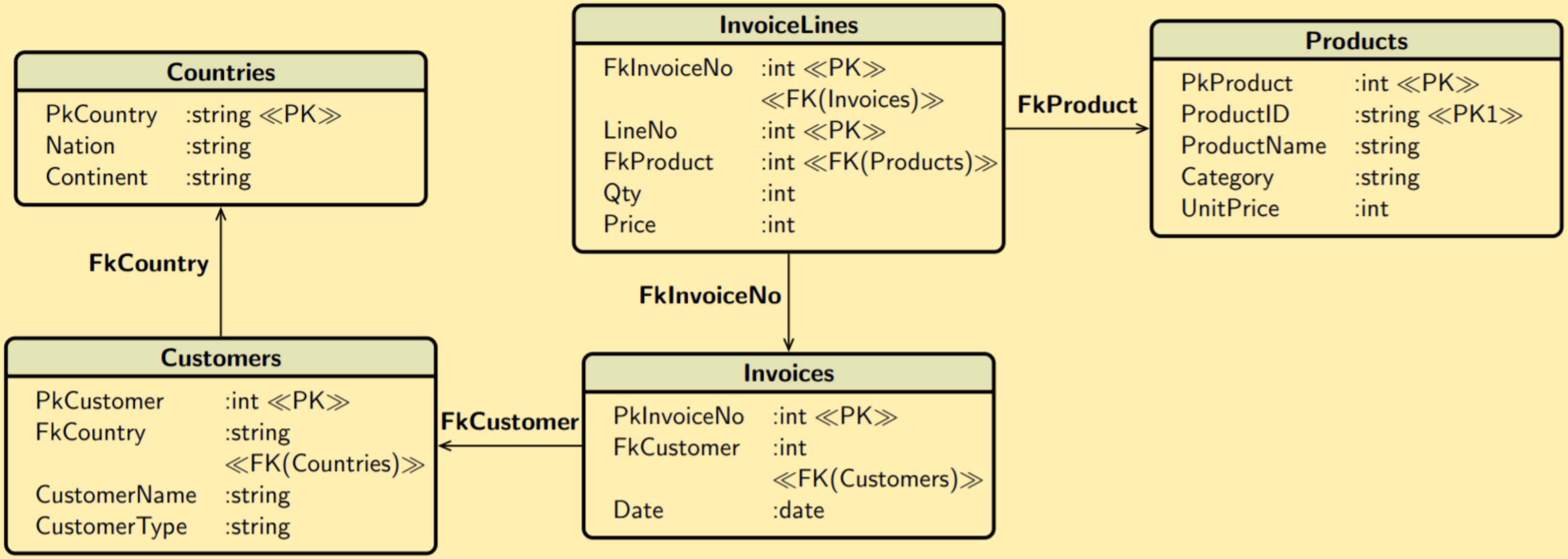
|

TableScan
(R)

PHYSICAL PLAN

Exercises from lesson 14: look at JRS physical query plans

- Using JRS on the database TestStar, write SQL queries and check their Logical Query Plans for:



1. Number of distinct Customers by Product

```
SELECT FkProduct, COUNT(DISTINCT FkCustomer) AS NCustomer
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo=PkInvoiceNo
GROUP BY FkProduct;
```

Physical Query Plans in Oracle

- **ORACLE:** EXPLAIN PLAN

```
SELECT /*+ GATHER_PLAN_STATISTICS */ count(*) from A, B, C WHERE
A.STATUS = B.STATUS AND   A.B_ID = B.ID AND   B.STATUS = 'OPEN' AND
B.ID = C.B_ID AND   C.STATUS = 'OPEN'

Plan hash value: 2966481601

----------------------------------------
| Id  | Operation                      |
----------------------------------------
|   1 |  SORT AGGREGATE                |
|*  2 |   HASH JOIN                    |
|*  3 |    HASH JOIN                   |
|   4 |     TABLE ACCESS BY INDEX ROWID|
|*  5 |      INDEX RANGE SCAN          |
|*  6 |     TABLE ACCESS FULL          |
|*  7 |    INDEX FAST FULL SCAN        |
----------------------------------------
```

Physical Query Plans in SQL Server (using Management Studio)

Estimated execution plan

Actual execution plan