

# Data Management

## 08 Query Processing

**Matthias Boehm, Patrick Damme**

Graz University of Technology, Austria  
Computer Science and Biomedical Engineering  
Institute of Interactive Systems and Data Science  
BMK endowed chair for Data Management



# Announcements/Org

## ■ #1 Video Recording

- Link in **TeachCenter** & **TUbe** (lectures will be public)
- Hybrid: HSi13 / <https://tugraz.webex.com/meet/m.boehm>
- **Apr 25:** no more COVID restrictions at TU Graz



## ■ #2 Exercises

- **Exercise 2:** due May 03 + 7 late days  
**Last chance** to submit in TeachCenter: May 10 (**tomorrow**)
- **Exercise 3:** published May 09 (tonight), due May 31

# Query Optimization and Query Processing

```
SELECT * FROM TopScorer
WHERE Count >= 4
```

```
CREATE VIEW TopScorer AS
SELECT P.Name, Count(*)
FROM Players P, Goals G
WHERE P.Pid=G.Pid
AND G.GOwn=FALSE
GROUP BY P.Name
ORDER BY Count(*) DESC
```

WHAT

Yes, but **HOW** do  
we get there  
efficiently

2014

Name	Count
James Rodríguez	6
Thomas Müller	5
Robin van Persie	4
Neymar	4

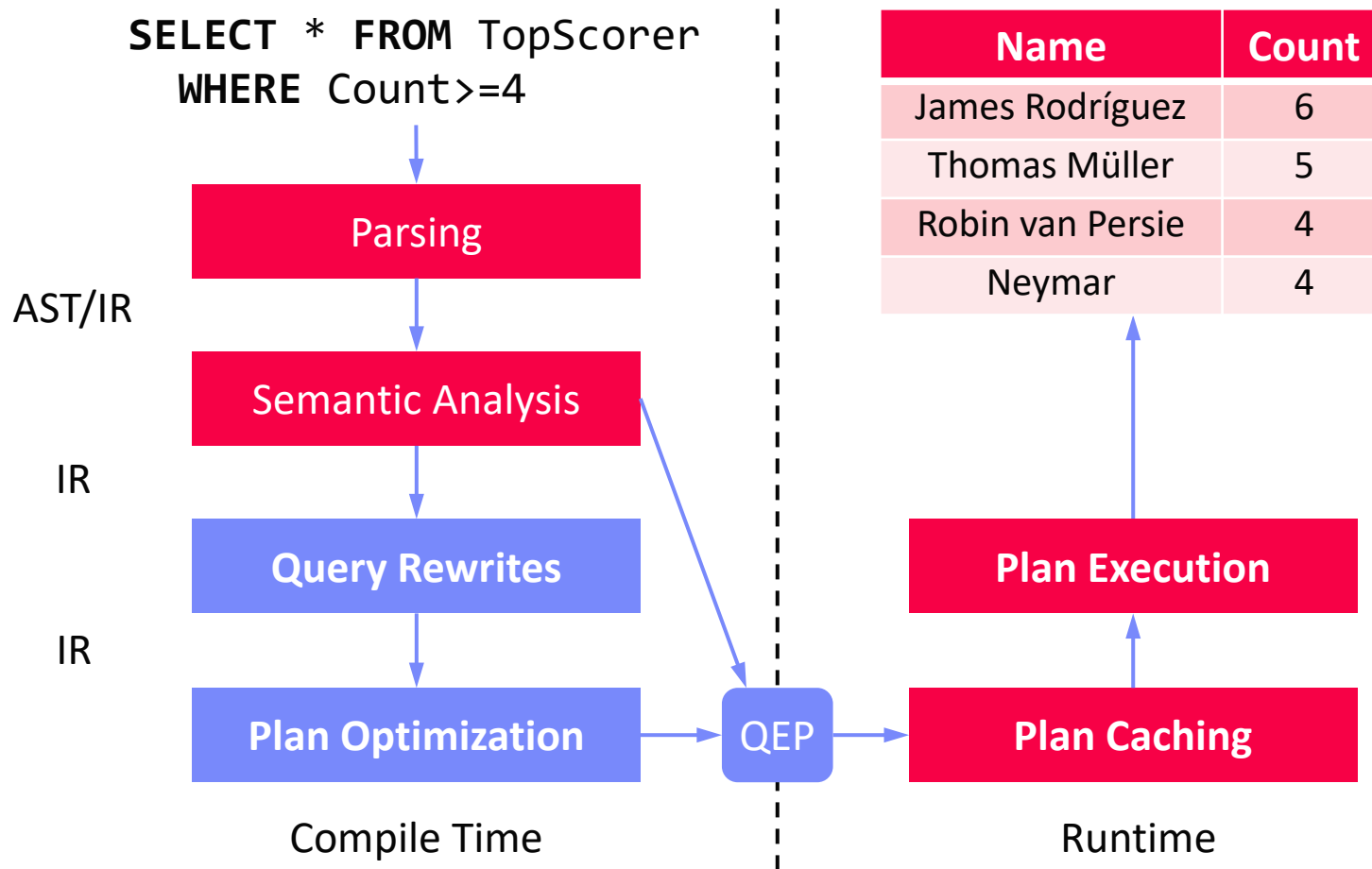
- **Goal: Basic Understanding of Internal Query Processing**
  - Query rewriting and query optimization
  - Query processing and physical plan operators
  - ➔ **Performance debugging & reuse of concepts and techniques**
  - ➔ Overview, detailed techniques discussed in **ADBS**

# Agenda

- **Query Rewriting and Optimization**
- **Plan Execution Strategies**
- **Physical Plan Operators**

# Query Rewriting and Optimization

# Overview Query Optimization



# Query Rewrites

## ■ Query Rewriting

- Rewrite query into semantically equivalent form that may be **processed more efficiently** or **give the optimizer more freedom**
- **#1 Same query can be expressed differently**, avoid hand-tuning
- **#2 Complex queries may have redundancy**

## ■ A Simple Example

- Catalog meta data:  
custkey is unique

```
SELECT DISTINCT custkey, name
FROM TPCH.Customer
```



```
SELECT custkey, name
FROM TPCH.Customer
```

## ■ 25+ years of experience on query rewriting

[**Hamid Pirahesh**, T. Y. Cliff Leung, Waqar Hasan:  
A Rule Engine for Query Transformation in  
Starburst and IBM DB2 C/S DBMS. **ICDE 1997**]



# Standardization and Simplification

## Normal Forms of Boolean Expressions

- **Conjunctive** normal form  $(P_{11} \text{ OR } \dots \text{ OR } P_{1n}) \text{ AND } \dots \text{ AND } (P_{m1} \text{ OR } \dots \text{ OR } P_{mp})$
- **Disjunctive** normal form  $(P_{11} \text{ AND } \dots \text{ AND } P_{1q}) \text{ OR } \dots \text{ OR } (P_{r1} \text{ AND } \dots \text{ AND } P_{rs})$

## Transformation Rules for Boolean Expressions

Rule Name	Examples
Commutativity rules	$A \text{ OR } B \Leftrightarrow B \text{ OR } A$ $A \text{ AND } B \Leftrightarrow B \text{ AND } A$
Associativity rules	$(A \text{ OR } B) \text{ OR } C \Leftrightarrow A \text{ OR } (B \text{ OR } C)$ $(A \text{ AND } B) \text{ AND } C \Leftrightarrow A \text{ AND } (B \text{ AND } C)$
Distributivity rules	$A \text{ OR } (B \text{ AND } C) \Leftrightarrow (A \text{ OR } B) \text{ AND } (A \text{ OR } C)$ $A \text{ AND } (B \text{ OR } C) \Leftrightarrow (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$
De Morgan's rules	$\text{NOT } (A \text{ AND } B) \Leftrightarrow \text{NOT } (A) \text{ OR } \text{NOT } (B)$ $\text{NOT } (A \text{ OR } B) \Leftrightarrow \text{NOT } (A) \text{ AND } \text{NOT } (B)$
Double-negation rules	$\text{NOT}(\text{NOT}(A)) \Leftrightarrow A$
Idempotence rules	$A \text{ OR } A \Leftrightarrow A$ $A \text{ AND } A \Leftrightarrow A$ $A \text{ OR } \text{NOT}(A) \Leftrightarrow \text{TRUE}$ $A \text{ AND } \text{NOT } (A) \Leftrightarrow \text{FALSE}$ $A \text{ AND } (A \text{ OR } B) \Leftrightarrow A$ $A \text{ OR } (A \text{ AND } B) \Leftrightarrow A$ $A \text{ OR } \text{FALSE} \Leftrightarrow A$ $A \text{ AND } \text{TRUE} \Leftrightarrow A$ $A \text{ AND } \text{FALSE} \Leftrightarrow \text{FALSE}$ $A \text{ OR } \text{TRUE} \Leftrightarrow \text{TRUE}$

# Standardization and Simplification, cont.

## ■ Elimination of Common Subexpressions

$$(A_1=a_{11} \text{ OR } A_1=a_{12}) \text{ AND } (A_1=a_{12} \text{ OR } A_1=a_{11}) \rightarrow A_1=a_{11} \text{ OR } A_1=a_{12}$$

## ■ Propagation of Constants

$$A \geq \mathbf{B} \text{ AND } B = \mathbf{7} \rightarrow A \geq \mathbf{7} \text{ AND } B = \mathbf{7} \quad R \bowtie_{a=b} (\sigma_{b>\theta}(S)) \rightarrow (\sigma_{a>\theta}(R)) \bowtie_{a=b} (\sigma_{b>\theta}(S))$$

## ■ Detection of Contradictions

$$A \geq B \text{ AND } B > C \text{ AND } C \geq A \rightarrow \mathbf{A > A} \rightarrow \mathbf{FALSE}$$

## ■ Use of Constraints

- A is primary key/unique:  $\pi_A \rightarrow$  no duplicate elimination necessary
- Rule  $\text{MAR\_STATUS} = \text{'married'} \rightarrow \text{TAX\_CLASS} \geq 3$ :  
 $(\text{MAR\_STATUS} = \text{'married'} \text{ AND } \text{TAX\_CLASS} = 1) \rightarrow \mathbf{FALSE}$

## ■ Elimination of Redundancy (set semantics)

- $R \bowtie R \rightarrow R$ ,  $R \cup R \rightarrow R$ ,  $R - R \rightarrow \emptyset$
- $R \bowtie (\sigma_p R) \rightarrow \sigma_p R$ ,  $R \cup (\sigma_p R) \rightarrow R$ ,  $R - (\sigma_p R) \rightarrow \sigma_{\neg p} R$
- $(\sigma_{p_1} R) \bowtie (\sigma_{p_2} R) \rightarrow \sigma_{p_1 \wedge p_2} R$ ,  $(\sigma_{p_1} R) \cup (\sigma_{p_2} R) \rightarrow \sigma_{p_1 \vee p_2} R$

# Query Unnesting

[Won Kim: On Optimizing an SQL-like Nested Query. **ACM Trans. Database Syst.** 1982]



## ■ Case 1: Type-A Nesting

- Inner block is not correlated and computes an aggregate
- Solution:** Compute the aggregate once and insert into outer query

```
SELECT OrderNo FROM Order
WHERE ProdNo =
  (SELECT MAX(ProdNo)
   FROM Product WHERE Price<100)
```



```
$X = SELECT MAX(ProdNo)
      FROM Product WHERE Price<100
SELECT OrderNo FROM Order
WHERE ProdNo = $X
```

## ■ Case 2: Type-N Nesting

- Inner block is not correlated and returns a set of tuples
- Solution:** Transform into a symmetric form (via join)

```
SELECT OrderNo FROM Order
WHERE ProdNo IN
  (SELECT ProdNo
   FROM Product WHERE Price<100)
```



```
SELECT OrderNo
FROM Order O, Product P
WHERE O.ProdNo = P.ProdNo
AND P.Price < 100
```

# Query Unnesting, cont.

[Won Kim: On Optimizing an SQL-like Nested Query. **ACM Trans. Database Syst.** 1982]



## Case 3: Type-J Nesting

- Un-nesting of correlated sub-queries w/o aggregation

```
SELECT OrderNo FROM Order O
WHERE ProdNo IN
  (SELECT ProdNo FROM Project P
   WHERE P.ProjNo = O.OrderNo
    AND P.Budget > 100,000)
```



```
SELECT OrderNo
FROM Order O, Project P
WHERE O.ProdNo = P.ProdNo
AND P.ProjNo = O.OrderNo
AND P.Budget > 100,000
```

## Case 4: Type-JA Nesting

- Un-nesting of correlated sub-queries w/ aggregation

```
SELECT OrderNo FROM Order O
WHERE ProdNo IN
  (SELECT MAX(ProdNo)
   FROM Project P
   WHERE P.ProjNo = O.OrderNo
    AND P.Budget > 100,000)
```



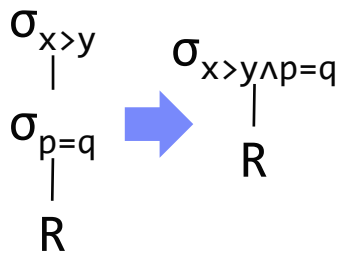
```
SELECT OrderNo FROM Order O
WHERE ProdNo IN
  (SELECT ProjNo, MAX(ProdNo)
   FROM Project
   WHERE Budget > 100.000
   GROUP BY ProjNo) P
WHERE P.ProjNo = O.OrderNo)
```

- Further un-nesting via case 3 and 2

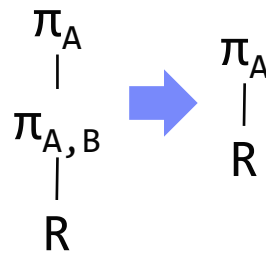
# Selections and Projections

## Example Transformation Rules

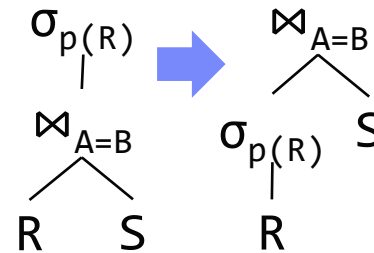
1) Grouping of Selections



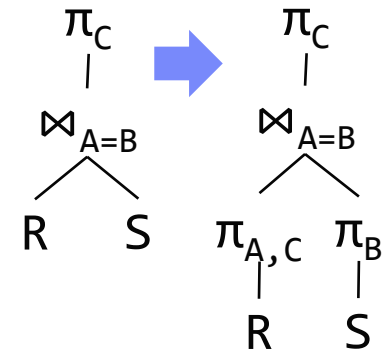
2) Grouping of Projections



3) Pushdown of Selections



4) Pushdown of Projections



## Restructuring Algorithm

- #1 Split n-ary joins into binary joins
- #2 Split multi-term selections
- #3 Push-down selections as far as possible
- #4 Group adjacent selections again
- #5 Push-down projections as far as possible

**Input:** Standardized, simplified, and un-nested query graph

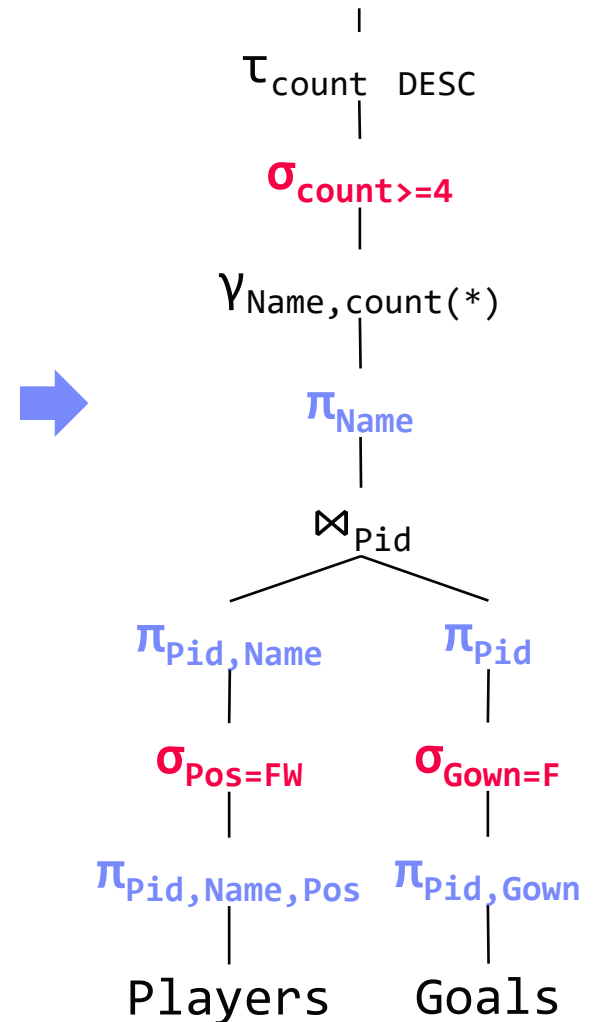
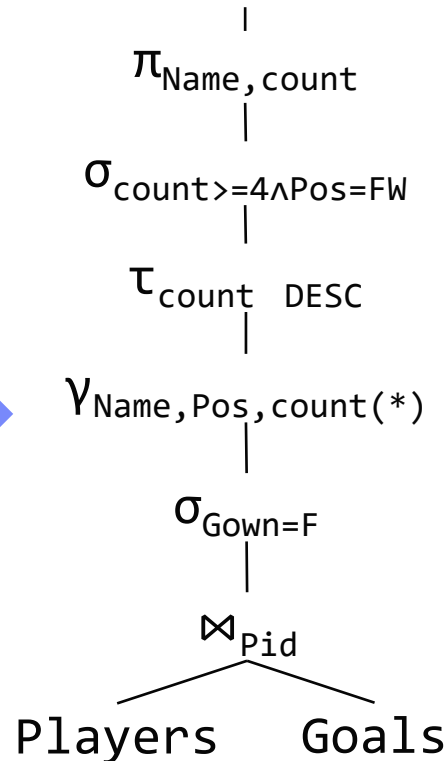
**Output:** Restructured query graph

# Example Query Restructuring

```
SELECT Name, count
FROM TopScorer
WHERE count >= 4
AND Pos = 'FW'
```

```
CREATE VIEW TopScorer AS
SELECT P.Name, P.Pos, count(*)
FROM Players P, Goals G
WHERE P.Pid=G.Pid
AND G.GOwn=FALSE
GROUP BY P.Name, P.Pos
ORDER BY count(*) DESC
```

Additional metadata:  
P.Name is unique



# Plan Optimization Overview

## Plan Generation

- Selection of **physical access path and plan operators**
- Selection of **execution order** of plan operators
- Input:** logical query plan → **Output:** optimal physical query plan
- Costs of query optimization should not exceed yielded improvements

## Different Cost Models

- Relies on statistics (cardinalities, selectivities via histograms + estimators)
- Operator-specific and general-purpose cost models

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases} \quad \begin{matrix} \text{(estimated)} & \text{(real)} \end{matrix}$$

- I/O costs** (number of read pages, tuples)
- Computation costs** (CPU costs, path lengths)
- Memory** (temporary memory requirements)
- Beware assumptions of optimizers**  
(no skew, independence, no correlation)

	10	590
$\sigma_{\text{Model}='Golf'}$		
	1,000	5,000
$\sigma_{\text{Make}='VW'}$		
Cars	10,000	10,000

# Query and Plan Types

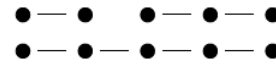
[Guido Moerkotte, Building Query Compilers  
(Under Construction), 2020,

<http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf>

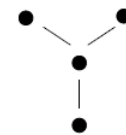


## Query Types

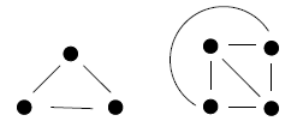
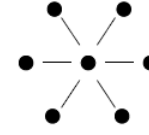
- Nodes: Tables
- Edges: Join conditions
- Determine **hardness of query optimization** (w/o cross products)



**Chains**



**Stars**

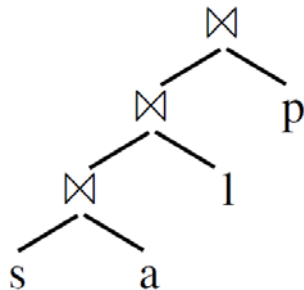


**Cliques**

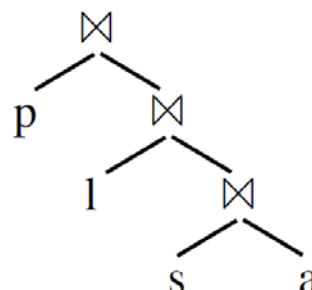
## Join Tree Types / Plan Types

- Data flow graph of tables and joins (logical/physical query trees)
- Edges: data dependencies (fixed execution order: bottom-up)

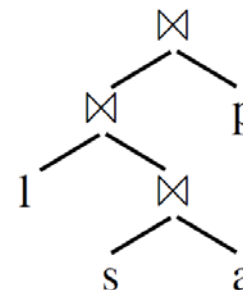
**Left-Deep Tree**



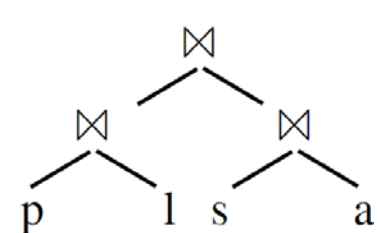
**Right-Deep Tree**



**Zig-Zag Tree**



**Bushy Tree**



# Join Ordering Problem

[Guido Moerkotte, Building Query Compilers  
(Under Construction), 2020,

<http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf>



## Join Ordering

- Given a join query graph, find the optimal join ordering
- In general, **NP-hard**; but polynomial algorithms exist for special cases

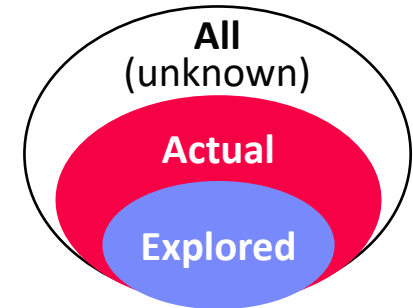
## Search Space

- Dependent on query and plan types
- Note:** if we allow cross products similar to cliques (fully connected)

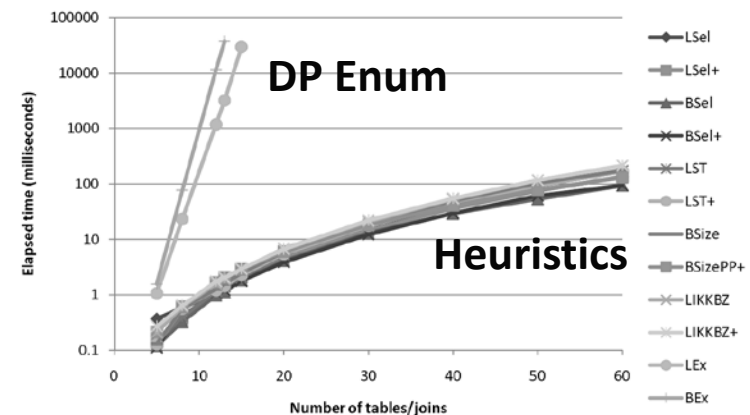
	Chain (no CP)			Star (no CP)		Clique / CP (cross product)		
	left-deep	zig-zag	bushy	left-deep	zig-zag/ bushy	left-deep	zig-zag	bushy
n	$2^{n-1}$	$2^{2n-3}$	$2^{n-1}C(n-1)$	$2(n-1)!$	$2^{n-1}(n-1)!$	$n!$	$2^{n-2}n!$	$n! C(n-1)$
5	16	128	224	48	384	120	960	1,680
10	512	~131K	~2.4M	~726K	~186M	~3.6M	~929M	~17.6G

$C(n)$  ... Catalan Numbers

# Join Order Search Strategies



- **Tradeoff: Optimal (or good) plan vs compilation time**
- **#1 Naïve Full Enumeration**
  - Infeasible for reasonably large queries (long tail up to 1000s of joins)
- **#2 Exact Dynamic Programming**
  - Guarantees optimal plan, often too expensive (beyond 20 relations)
  - Bottom-up vs top-down approaches
- **#3 Greedy / Heuristic Algorithms**
- **#4 Approximate Algorithms**
  - E.g., Genetic algorithms, simulated annealing
- **Example PostgreSQL**
  - Exact optimization (DPSize) if < 12 relations (geqo\_threshold)
  - Genetic algorithm for larger queries
  - Join methods: NLJ, SMJ, HJ

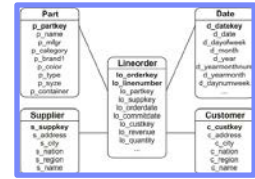


[Nicolas Bruno, César A. Galindo-Legaria, Milind Joshi: Polynomial heuristics for query optimization. **ICDE 2010**]



# Greedy Join Ordering

Star Schema  
Benchmark



## Example

- Part  $\bowtie$  Lineorder  $\bowtie$  Supplier  $\bowtie$   $\sigma$ (Customer)  $\bowtie$   $\sigma$ (Date), **left-deep plans**

#	Plan	Costs
1	Lineorder $\bowtie$ Part	30M
	Lineorder $\bowtie$ Supplier	20M
	Lineorder $\bowtie$ $\sigma$ (Customer)	90K
	Lineorder $\bowtie$ $\sigma$ (Date)	40K
	<del>Part <math>\bowtie</math> Customer</del>	N/A
	...	...

2	(Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ Part	150K
	(Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ Supplier	100K
	(Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ $\sigma$ (Customer)	75K

#	Plan	Costs
3	((Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ $\sigma$ (Customer)) $\bowtie$ Part	120M
	((Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ $\sigma$ (Customer)) $\bowtie$ Supplier	105M
4	((((Lineorder $\bowtie$ $\sigma$ (Date)) $\bowtie$ $\sigma$ (Customer)) $\bowtie$ Supplier) $\bowtie$ Part	135M

**Note:** Simple  $O(n^2)$  algorithm for left-deep trees;  
 $O(n^3)$  algorithms for bushy trees existing (e.g., GOO)

# Dynamic Programming Join Ordering

## Exact Enumeration via Dynamic Programming

- #1: **Optimal substructure** (Bellman's Principle of Optimality)
- #2: **Overlapping subproblems** allow for memoization

➔ Approach DPSize: Split in independent subproblems (optimal plan per set of quantifiers and interesting properties), solve subproblems, combine solutions

## Example

Q1+Q1		Q1+Q2, Q2+Q1		Q1+Q3, Q2+Q2, Q3+Q1	
Q1	Plan	Q2	Plan	Q3	Plan
{C}	Tbl, IX	{C,L}	$L \bowtie C, C \bowtie L$	{C,D,L}	$(L \bowtie C) \bowtie D, D \bowtie (L \bowtie C), (L \bowtie D) \bowtie C, C \bowtie (L \bowtie D)$
{D}	Tbl, IX	{D,L}	$L \bowtie D, D \bowtie L$	{C,L,P}	$(L \bowtie C) \bowtie P, P \bowtie (L \bowtie C), (P \bowtie L) \bowtie C, C \bowtie (P \bowtie L)$
{L}	...	{L,P}	$L \bowtie P, P \bowtie L$	{C,L,S}	...
{P}	...	{L,S}	$L \bowtie S, S \bowtie L$	{D,L,P}	...
{S}	...	{C,D}	N/A	{D,L,S}	...
		...	...	{L,P,S}	...

Q4	Plan
{C,D,L,P}	$((L \bowtie C) \bowtie D) \bowtie P, P \bowtie ((L \bowtie C) \bowtie D)$
{C,D,L,S}	...
{C,L,P,S}	...
{D,L,P,S}	...

Q5	Plan
{C,D,L,P,S}	...

# BREAK (and Test Yourself)

- Rewrite the following RA expressions – assuming two relations  $R(a, b, c)$  and  $S(d, e, f)$  – into equivalent expressions with lower costs. (5 points)

$$\sigma_{b=7}(R \bowtie S) \rightarrow \sigma_{b=7}(R) \bowtie S$$

$$(\sigma_{e>3}(S)) \cap (\sigma_{f<7}(S)) \rightarrow \sigma_{e>3 \wedge f<7}(S)$$






$$\pi_{a,b}(R \bowtie_{a=d} S) \rightarrow \pi_{a,b}(R) \bowtie_{a=d} S$$

$$R \cup (\sigma_{d<e \wedge e<f \wedge f<d}(S)) \rightarrow R \cup \emptyset \rightarrow R$$

$$\sigma_{b=3}(\gamma_{b,\max(c)}(R)) \rightarrow \gamma_{3,\max(c)}(\sigma_{b=3}(R))$$

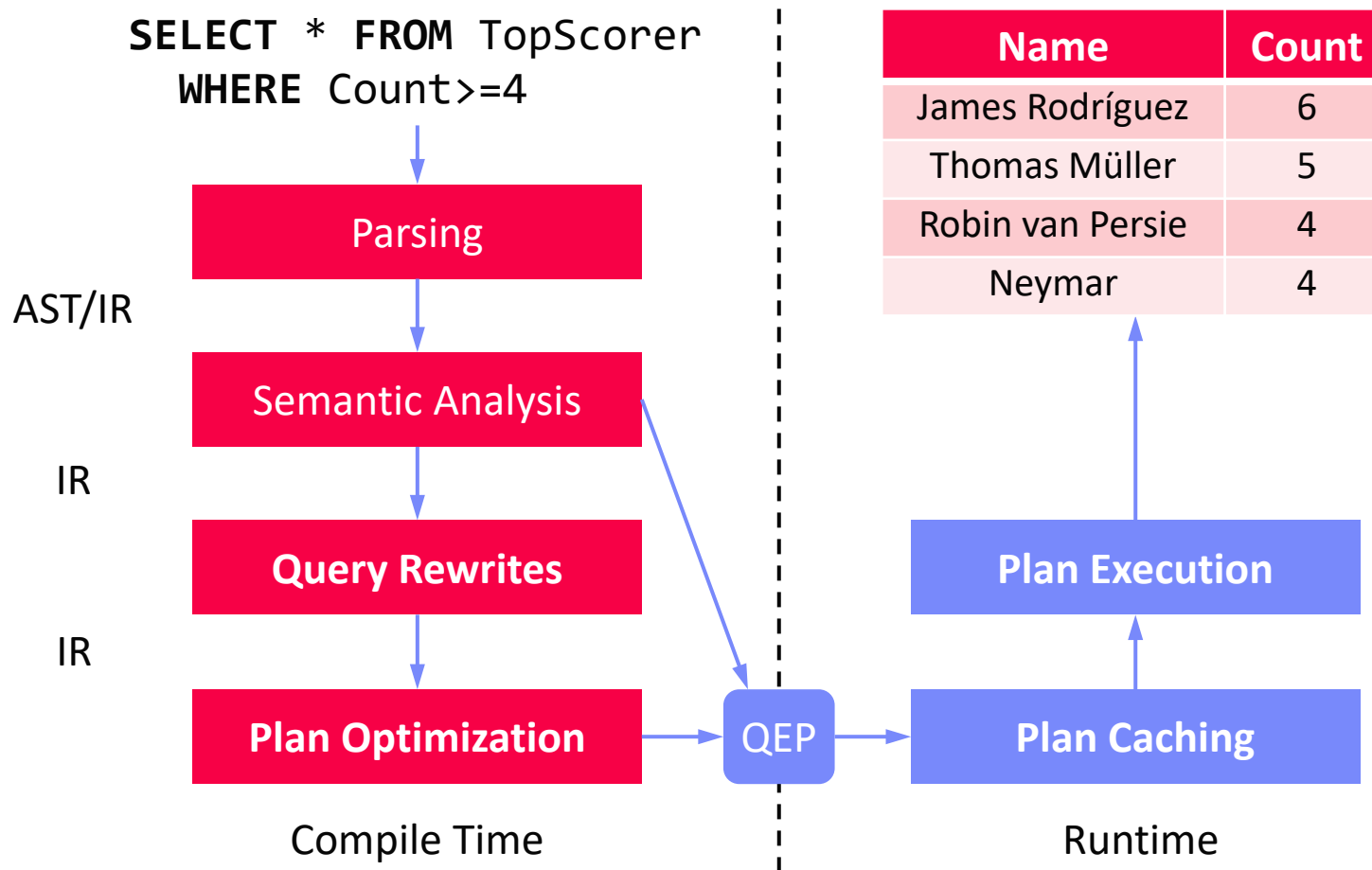
# BREAK (and Test Yourself), cont.

- Assume relations  $R(a,b,c)$  and  $S(d,e)$ , and indicate in the table below whether or not the two RA expressions per row are equivalent in bag semantics. For non-equivalent expressions briefly explain why. (5 points)

Expression 1	Expression 2	Equivalent?
$\sigma_{c=3}(\sigma_{b=7}(R))$	$\sigma_{c=3}(\sigma_{c=3 \vee b=7}(R))$	
$R \bowtie_{a=e} S$	$\sigma_{a=e}(R \times S)$	
$(\sigma_{b<3}(R)) \cap (\sigma_{b \geq 3}(R))$	$R$	
$\pi_{b,d}(R \bowtie_{a=e} S)$	$(\pi_{a,b}(R)) \bowtie_{a=e} (\pi_{d,e}(S))$	
$\pi_{a,b}(\sigma_{c=3}(\sigma_{b=7}(R)))$	$\sigma_{b=7}(\pi_{a,b}(\sigma_{c=3}(R)))$	

# Plan Execution Strategies

# Overview Query Processing



# Overview Execution Strategies

- Different execution strategies (processing models) with different pros/cons (e.g., memory requirements, DAGs, efficiency, reuse)
- #1 **Iterator Model** (mostly row stores)
- #2 **Materialized Intermediates** (mostly column stores)
- #3 **Vectorized (Batched) Execution** (row/column stores)
- #4 **Query Compilation** (row/column stores)

High-level  
overview,  
details in  
**ADBS**

## High CPI measures

[Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. **IEEE Trans. Knowl. Data Eng.** 1994]



- **Example  $\sigma_{A=7}(R)$**

Diagram illustrating the execution of a recursive function with base cases. The diagram shows three levels of function calls: `open()`, `next()`, `next()`, `close()` at the top; `open()`, `next()`, `next()`, `close()` in the middle; and `open()`, `next()`, `next()`, `next()`, `close()` at the bottom. A vertical line connects the first `next()` of each level. To the right of this line, the label  $\sigma_{A=7}$  is shown, followed by a red arrow pointing to **EOF**. Below  $\sigma_{A=7}$ , the letter **R** is shown. At the bottom right, another red arrow points to **EOF**.

- Sorting, grouping/aggregation, build-phase of (simple) hash joins

**PostgreSQL:** `Init()`,  
`GetNext()`, `ReScan()`, `MarkPos()`,  
`RestorePos()`, `End()`

# Iterator Model – Predicate Evaluation

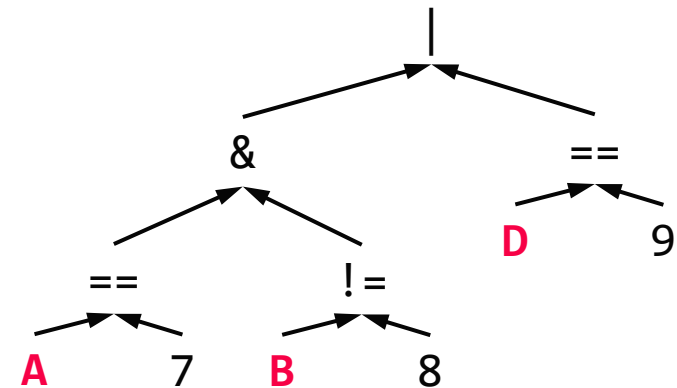
## Operator Predicates

- Examples: arbitrary selection predicates and join conditions
- Operators parameterized **with in-memory expression trees/DAGs**
- Expression evaluation engine** (interpretation)

## Example Selection $\sigma$

- $(A = 7 \wedge B \neq 8) \vee D = 9$

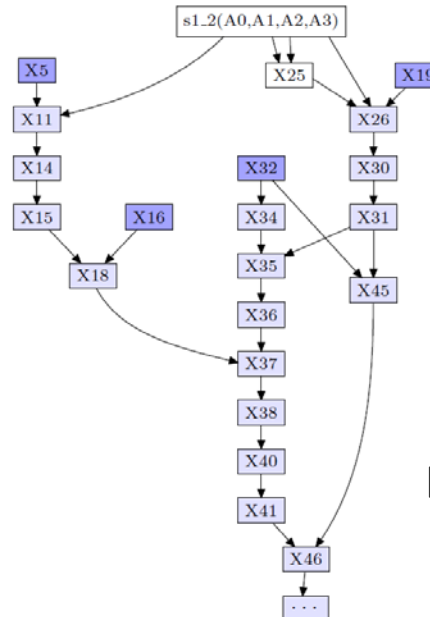
A	B	C	D
7	8	Product 1	10
14	8	Product 3	11
7	3	Product 7	7
3	3	Product 2	1



# Materialized Intermediates (column-at-a-time)

```
SELECT count(DISTINCT o_orderkey)
FROM orders, lineitem
WHERE l_orderkey = o_orderkey
      AND o_orderdate >= date '1996-07-01'
      AND o_orderdate < date '1996-07-01'
        + interval '3' month
      AND l_returnflag = 'R';
```

Column-oriented storage  
Efficient array operations  
DAG processing  
Reuse of intermediates  
Memory requirements  
Unnecessary read/write  
from and to memory



```
function user.s1_2(A0:date,A1:date,A2:int,A3:str):void;
  X5 := sql.bind("sys","lineitem","l_returnflag",0);
  X11 := algebra.uselect(X5,A3);
  X14 := algebra.markT(X11,0@0);
  X15 := bat.reverse(X14);
  X16 := sql.bindIdxbat("sys","lineitem","l_orderkey_fkey");
  X18 := algebra.join(X15,X16);
  X19 := sql.bind("sys","orders","o_orderdate",0);
  X25 := mtime.addmonths(A1,A2);
  X26 := algebra.select(X19,A0,X25,true,false);
  X30 := algebra.markT(X26,0@0);
  X31 := bat.reverse(X30);
  X32 := sql.bind("sys","orders","o_orderkey",0);
  X34 := bat.mirror(X32);
  X35 := algebra.join(X31,X34);
  X36 := bat.reverse(X35);
  X37 := algebra.join(X18,X36);
  X38 := bat.reverse(X37);
  X40 := algebra.markT(X38,0@0);
  X41 := bat.reverse(X40);
  X45 := algebra.join(X31,X32);
  X46 := algebra.join(X41,X45);
  X49 := algebra.selectNotNil(X46);
  X50 := bat.reverse(X49);
  X51 := algebra.kunique(X50);
  X52 := bat.reverse(X51);
  X53 := aggr.count(X52);
  sql.exportValue(1,"sys.orders","L1","wrd",32,0,6,X53);
end s1_2;
```

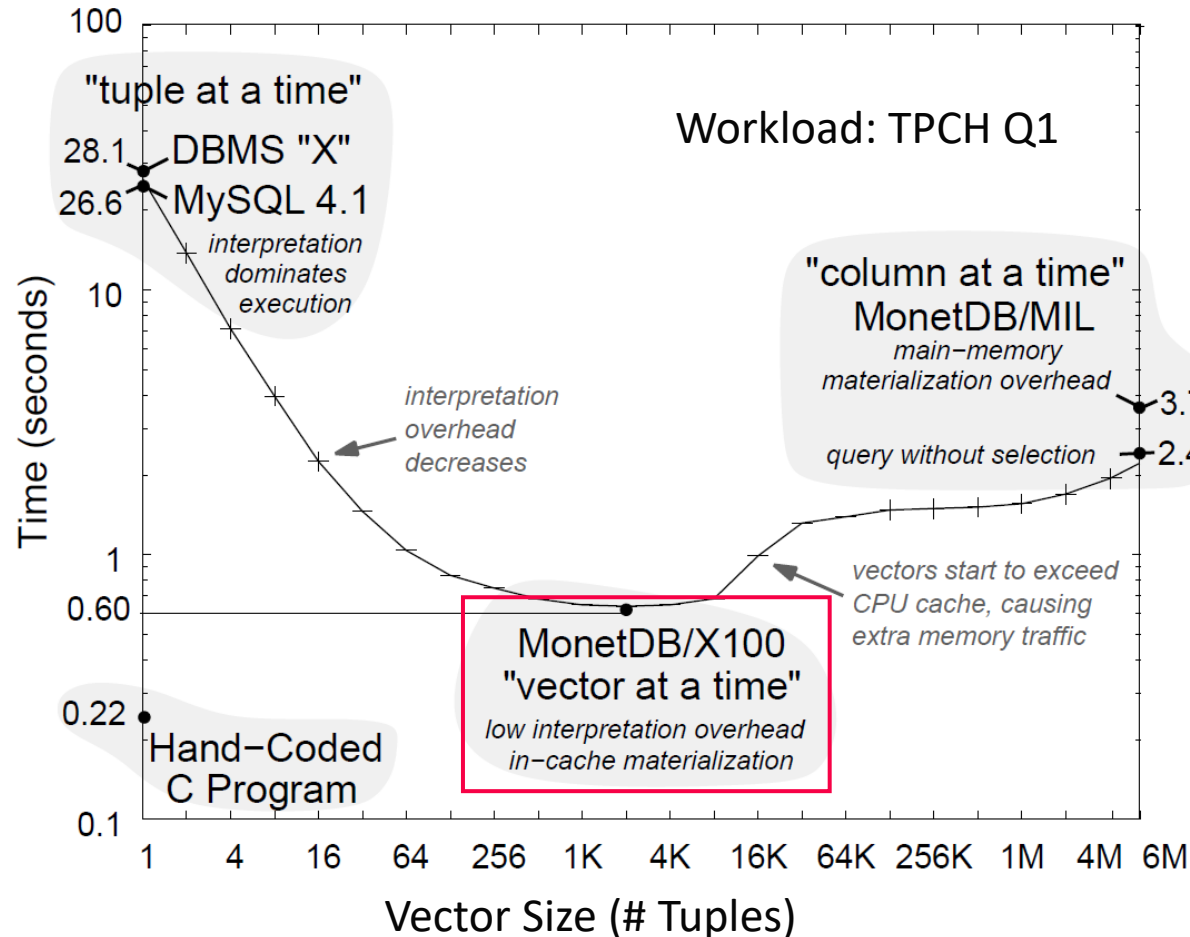
Binary  
Association  
Tables  
(BATs:=OID/Val)

[Milena Ivanova, Martin L. Kersten, Niels J. Nes, Romulo Goncalves: An architecture for recycling intermediates in a column-store. **SIGMOD 2009**]



# Vectorized Execution (vector-at-a-time)

- Idea: Pipelining of vectors (sub columns) s.t. vectors fit in CPU cache



Column-oriented storage  
Efficient array operations  
Memory/cache efficiency  
DAG processing  
**Reuse of intermediates**



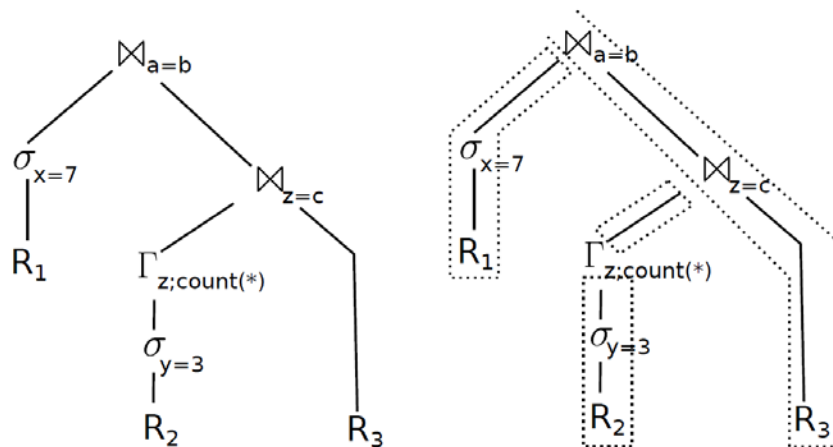
[Peter A. Boncz, Marcin Zukowski,  
Niels Nes: MonetDB/X100: Hyper-  
Pipelining Query Execution.  
CIDR 2005]

# Query Compilation

- Idea: Data-centric, not op-centric processing + LLVM code generation

## Operator Trees

(w/o and w/ pipeline boundaries)



[Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. **PVLDB 2011**]

## Compiled Query

(conceptual, not LLVM)

```

initialize memory of  $\bowtie_{a=b}$ ,  $\bowtie_{c=z}$ , and  $\Gamma_z$ 
for each tuple  $t$  in  $R_1$ 
    if  $t.x = 7$ 
        materialize  $t$  in hash table of  $\bowtie_{a=b}$ 
for each tuple  $t$  in  $R_2$ 
    if  $t.y = 3$ 
        aggregate  $t$  in hash table of  $\Gamma_z$ 
for each tuple  $t$  in  $\Gamma_z$ 
    materialize  $t$  in hash table of  $\bowtie_{z=c}$ 
for each tuple  $t_3$  in  $R_3$ 
    for each match  $t_2$  in  $\bowtie_{z=c}[t_3.c]$ 
        for each match  $t_1$  in  $\bowtie_{a=b}[t_3.b]$ 
            output  $t_1 \circ t_2 \circ t_3$ 
    
```





# Physical Plan Operators

# Overview Plan Operators

## Multiple Physical Operators

- Different physical operators for different data and query characteristics
- Physical operators can have vastly different costs

## Examples (supported in most DBMS)

Logical Plan Operators	Selection $\sigma_p(R)$	Projection $\pi_A(R)$	Grouping $\gamma_{G:agg(A)}(R)$	Join $R \bowtie_{R.a=S.b} S$
				
Physical Plan Operators	TableScan IndexScan ALL	ALL	SortGB HashGB	NestedLoopJN SortMergeJN HashJN

Lecture 07

This Lecture  
(Exercise 3)

# Nested Loop Join

## Overview

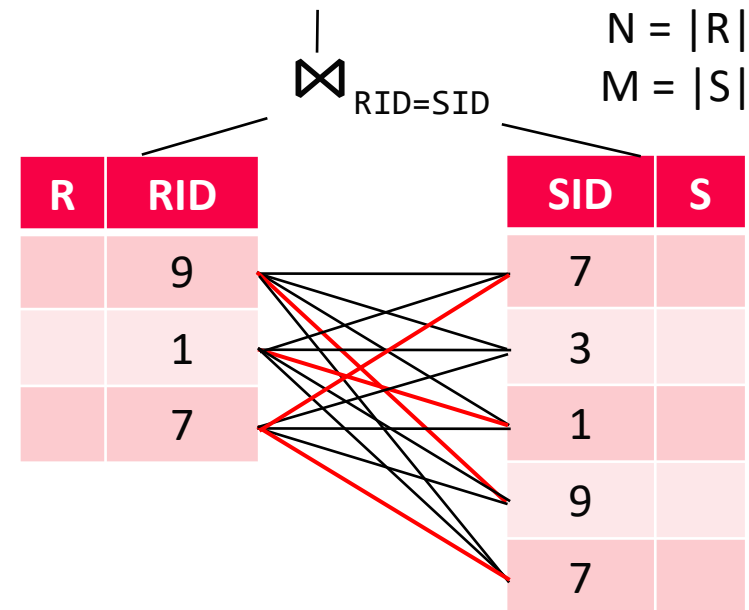
- Most general join operator (no order, no indexes, arbitrary predicates  $\theta$ )
- Poor asymptotic behavior (very slow)

## Algorithm (pseudo code)

```

for each s in S
  for each r in R
    if( r.RID  $\theta$  s.SID )
      emit concat(r, s)
  
```

How to implement **next()**?



## Complexity

- Complexity: Time:  $O(N * M)$ , Space:  $O(1)$
- Pick smaller table as inner if it fits entirely in memory (buffer pool)

# Block Nested Loop / Index Nested Loop Joins

## Block Nested Loop Join

- Avoid I/O by blocked data access
- Read blocks of  $b_R$  and  $b_S$  R and S pages
- Complexity unchanged but potentially much fewer scans

```

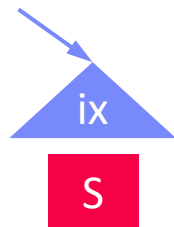
for each block  $b_R$  in R
  for each block  $b_S$  in S
    for each  $r$  in  $b_R$ 
      for each  $s$  in  $b_S$ 
        if(  $r.RID \theta s.SID$  )
          emit concat( $r, s$ )
  
```

## Index Nested Loop Join

- Use index to locate qualifying tuples  
( $=$ ,  $>=$ ,  $>$ ,  $<=$ ,  $<$ )
- Complexity (for equivalence predicates):  
Time:  $O(N * \log M)$ , Space:  $O(1)$

```

for each  $r$  in R
  for each  $s$  in  $S.IX(\theta, r.RID)$ 
    emit concat( $r, s$ )
  
```



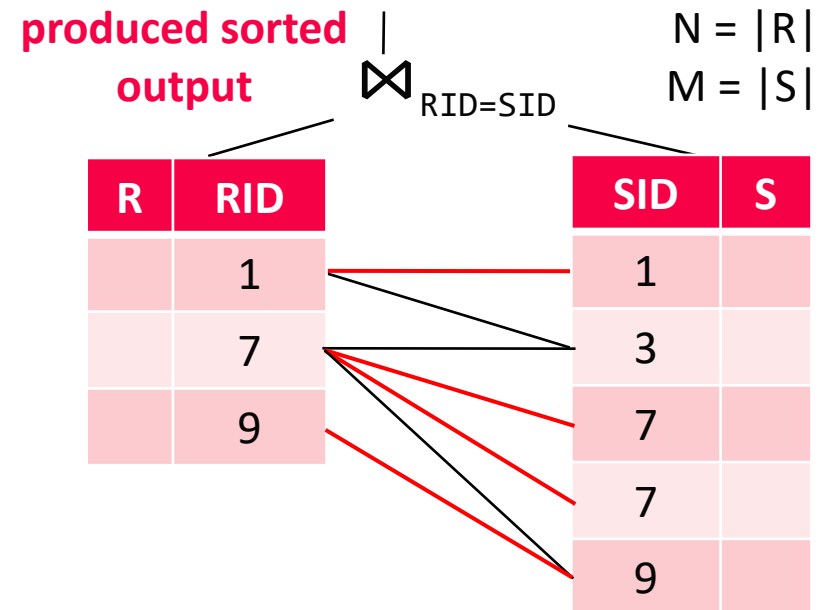
# Sort-Merge Join

## Overview

- **Sort Phase:** sort the input tables R and S (w/ external sort algorithm)
- **Merge Phase:** step-wise merge with lineage scan

## Algorithm (Merge, PK-FK)

```
Record next() {
  while( curR!=EOF && curS!=EOF ) {
    if( curR.RID < curS.SID )
      curR = R.next();
    else if( curR.RID > curS.SID )
      curS = S.next();
    else if( curR.RID == curS.SID ) {
      t = concat(curR, curS);
      curS = S.next(); //FK side
      return t;
    }
  }
  return EOF;
}
```



## Complexity

- Time (unsorted vs sorted):  $O(N \log N + M \log M)$  vs  $O(N + M)$
- Space (unsorted vs sorted):  $O(N + M)$  vs  $O(1)$

# Hash Join

## Overview

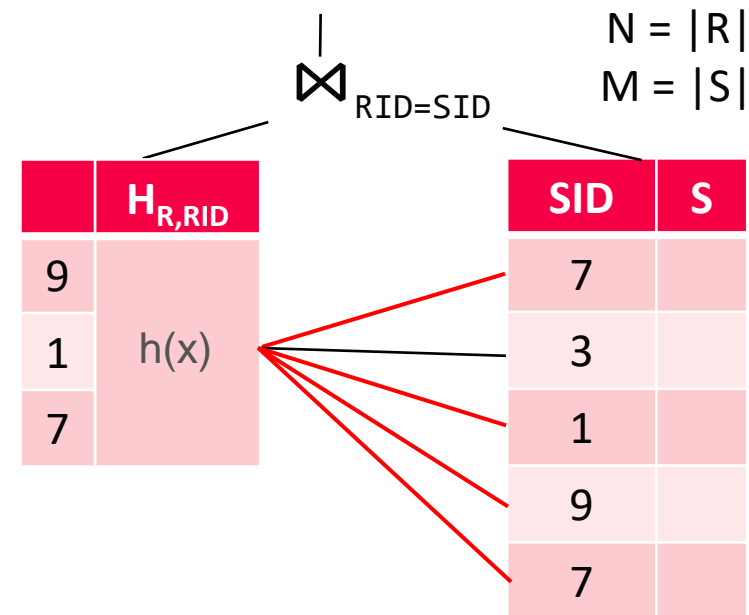
- **Build Phase:** read table S and build a hash table  $H_S$  over join key
- **Probe Phase:** read table R and probe  $H_S$  with the join key

## Algorithm (Build+Probe, PK-FK)

```
Record next() {
    // build phase (first call only)
    while( (r = R.next()) != EOF )
        Hr.put(r.RID, r);

    // probe phase
    while( (s = S.next()) != EOF )
        if( Hr.containsKey(s.SID) )
            return concat(Hr.get(s.SID), s);

    return EOF;
}
```



## Complexity

- Time:  $O(N + M)$ , Space:  $O(N)$
- Classic hashing: p in-memory partitions of Hr w/ p scans of R and S

# Sort-GroupBy and Hash-GroupBy

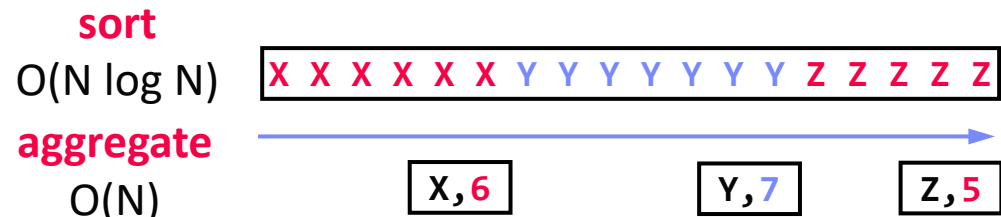
## Recap: Classification of Aggregates (04 Relational Algebra)

- Additive, semi-additive, additively-computable, others

$$\gamma_{A, \text{count}(*)}(R)$$

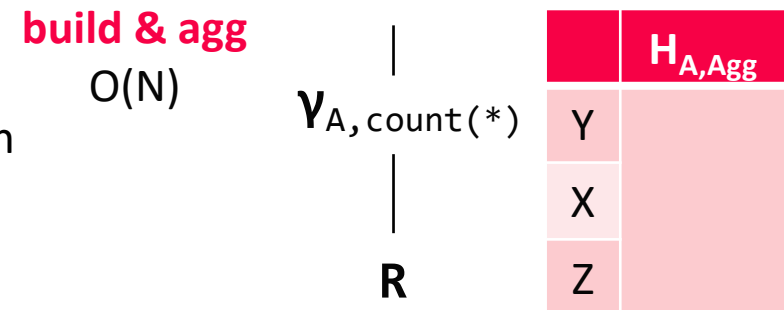
## Sort Group-By

- Similar to sort-merge join (Sort, GroupAggregate)
- Sorted group output



## Hash Group-By

- Similar to hash join (HashAggregate)
- Higher temporary memory consumption
- Unsorted group output
- #1 w/ tuple grouping
- #2 w/ direct aggregation (e.g., count)
- Beware:** cache-unfriendly if many groups ( $\text{size}(H) > \text{L2/L3 cache}$ )



# Summary and Q&A

- **Query Rewriting and Optimization**
- **Plan Execution Strategies**
- **Physical Plan Operators**
  
- **Next Lectures (Part A)**
  - **09 Transaction Processing and Concurrency** [May 16]
- **Next Lectures (Part B)**
  - 10 NoSQL (key-value, document, graph) [May 23]
  - 11 Distributed Storage and Data Analysis [May 30]
  - 12 Data Stream Processing Systems [Jun 13, Patrick]