SCIENCE
PASSION
TECHNOLOGY

Database Systems
08 Query Processing

Matthias Boehm

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

Last update: June 24, 2019

Announcements/Org

- **#1 Video Recording**
  - Since lecture 03, video/audio recording
  - Link in TeachCenter & TUbe (**video recorder fixed?**)

- **#2 Statistics Exercise 1**
  - All submissions accepted (submitted/draft)
  - In progress of **grading**, but understaffed

77.4%

- **#3 Exercise 2**
  - Submission is crucial (modified rule: 1 exercise 2%-50%)
  - Modified deadline: **May 07 11.59pm**
  - **Please, submit correct file names**
    (avoid wrong IDs, wrong naming scheme)

Announcements/Org, cont.

- **#4 Study Abroad Fair 2019**

➢ **Study Abroad Fair**
May 22, 2019

✓ Your opportunity to find out about exchange programmes and scholarships offered by TU Graz

✓ Information booths

✓ Short presentations concerning various study abroad possibilities

tu4u.tugraz.at/go/study-abroad-fair-2019

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 to we get there efficiently

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

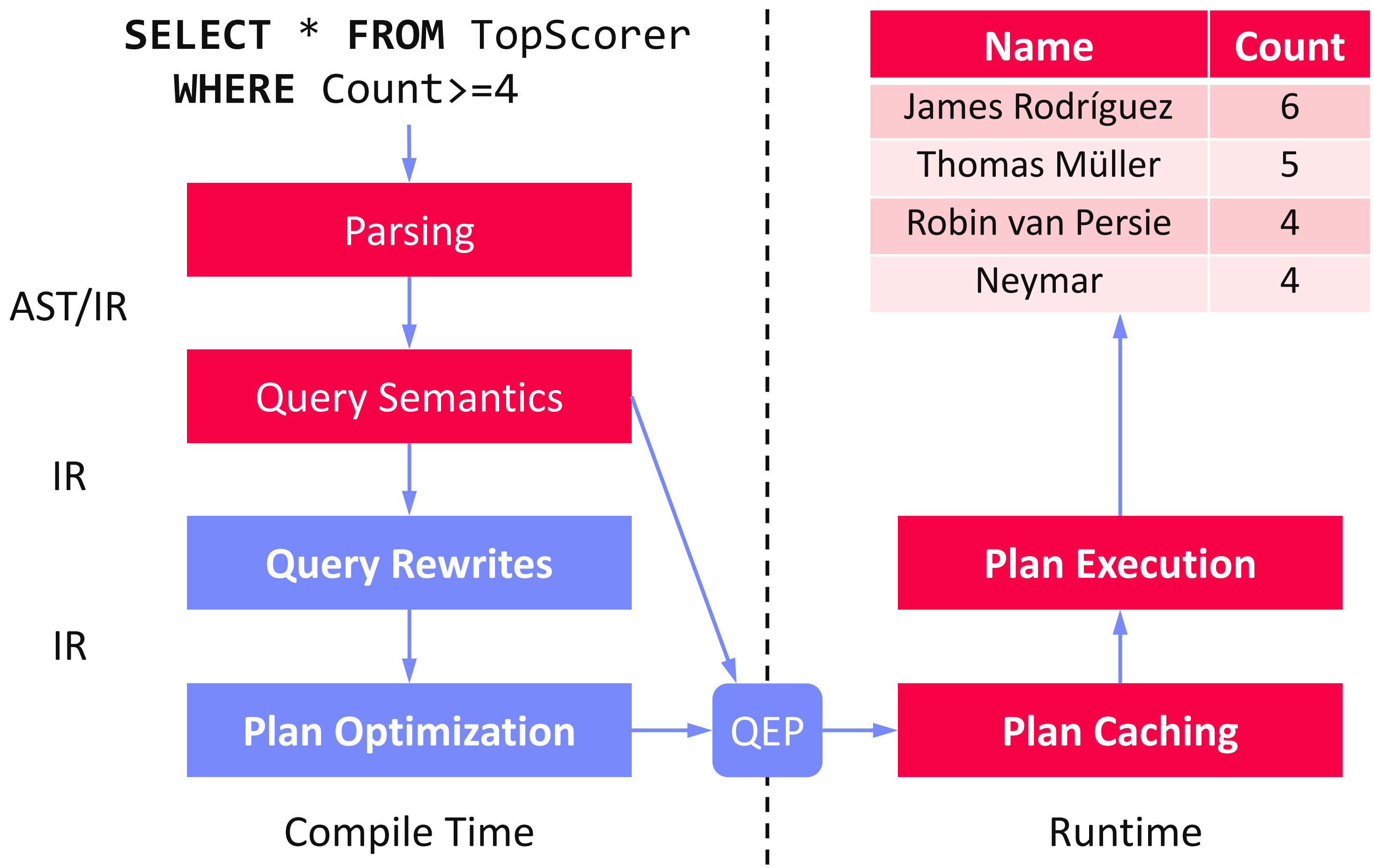
- **Goal: Basic Understanding of Internal Query Processing**
  - Query rewriting and query optimization
  - Query processing and physical plan operators
  - ➔ **Performance debugging & reuse of concepts and techniques**
  - ➔ Overview, detailed techniques discussed in **ADBS**

Agenda

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

Query Rewriting and Optimization

Overview Query Optimization



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

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**, prevent hand optimization
  - **#2 Complex queries may have redundancy**

- **A Simple Example**
  - Catalog meta data: custkey is unique

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

⬇ rewrite

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

- **20+ 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} OR ... OR P_{1n}) **AND ... AND** (P_{m1} OR ... OR P_{mp})
  - **Disjunctive** normal form (P_{11} AND ... AND P_{1q}) **OR ... OR** (P_{r1} AND ... AND P_{rs})

- **Transformation Rules for Boolean Expressions**

Rule Name	Examples	
Commutativity rules	A OR B ⇔ B OR A A AND B ⇔ B AND A	
Associativity rules	(A OR B) OR C ⇔ A OR (B OR C) (A AND B) AND C ⇔ A AND (B AND C)	
Distributivity rules	A OR (B AND C) ⇔ (A OR B) AND (A OR C) A AND (B OR C) ⇔ (A AND B) OR (A AND C)	
De Morgan's rules	NOT (A AND B) ⇔ NOT (A) OR NOT (B) NOT (A OR B) ⇔ NOT (A) AND NOT (B)	
Double-negation rules	NOT(NOT(A)) ⇔ A	
Idempotence rules	A OR A ⇔ A A OR NOT(A) ⇔ TRUE A AND (A OR B) ⇔ A A OR FALSE ⇔ A A AND FALSE ⇔ FALSE	A AND A ⇔ A A AND NOT (A) ⇔ FALSE A OR (A AND B) ⇔ A A OR TRUE ⇔ TRUE

Standardization and Simplification, cont.

- **Elimination of Common Subexpressions**
  - $(A_1=a_{11}$ **OR** $A_1=a_{12})$ **AND** $(A_1=a_{12}$ **OR** $A_1=a_{11}) \rightarrow A_1=a_{11}$ **OR** $A_1=a_{12}$
- **Propagation of Constants**
  - A ≥ **B** **AND** B = **7** → A ≥ **7** AND B = **7**
- **Detection of Contradictions**
  - A ≥ B **AND** B > C **AND** C ≥ A → **A > A** → **FALSE**
- **Use of Constraints**
  - A is primary key/unique: π_A → no duplicate elimination necessary
  - Rule MAR_STATUS = 'married' → TAX_CLASS ≥ 3:
    (MAR_STATUS = 'married' **AND** TAX_CLASS = 1) → **FALSE**
- **Elimination of Redundancy**
  - $R \bowtie R \rightarrow R$, $R \cup R \rightarrow R$, $R - R \rightarrow \emptyset$
  - $\mathbf{R \bowtie (\sigma_p R) \rightarrow \sigma_p R}$, $R \cup (\sigma_p R) \rightarrow R$, $R - (\sigma_p R) \rightarrow \sigma_{\neg p} R$
  - $\mathbf{(\sigma_{p1} R) \bowtie (\sigma_{p2} R) \rightarrow \sigma_{p1 \wedge p2} R}$, $(\sigma_{p1} R) \cup (\sigma_{p2} R) \rightarrow \sigma_{p1 \vee p2} R$

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)
```

→

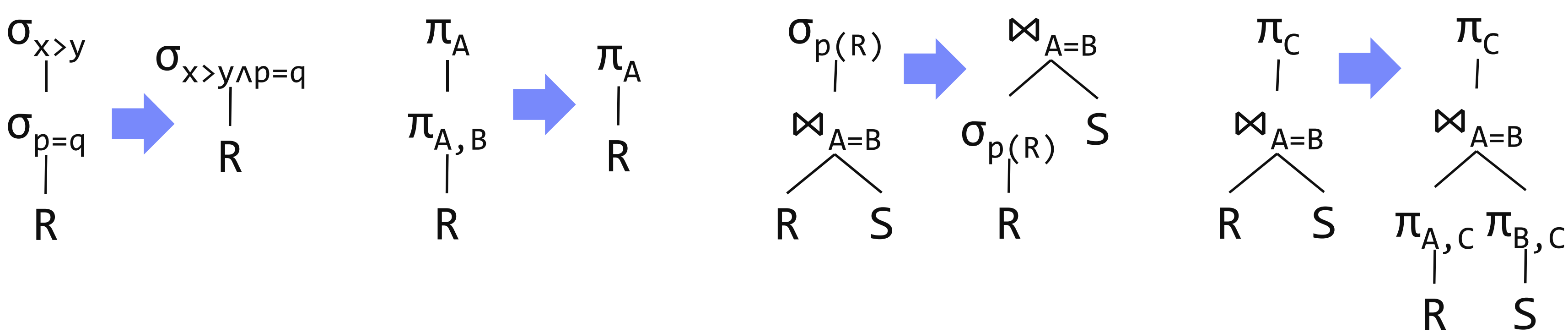
```
SELECT OrderNo FROM Order O
  WHERE ProdNo IN
   (SELECT ProdNo FROM
     (SELECT ProjNo, MAX(ProdNo)
       FROM Project
       GROUP BY ProjNo) P
    WHERE P.ProjNo = O.OrderNo
       AND P.Budget > 100.000)
```

  - 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

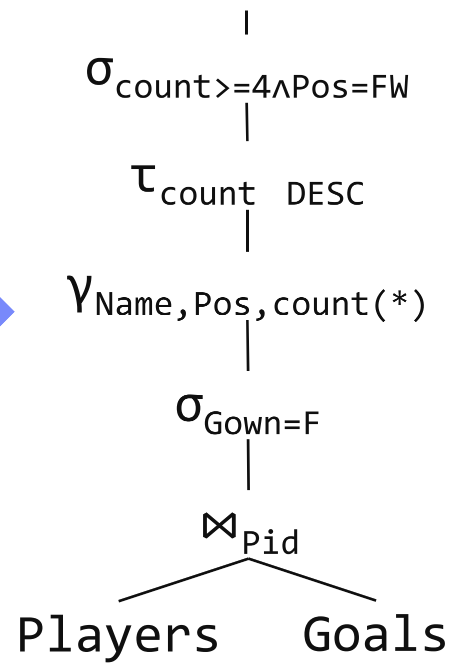
Output: Restructured query graph

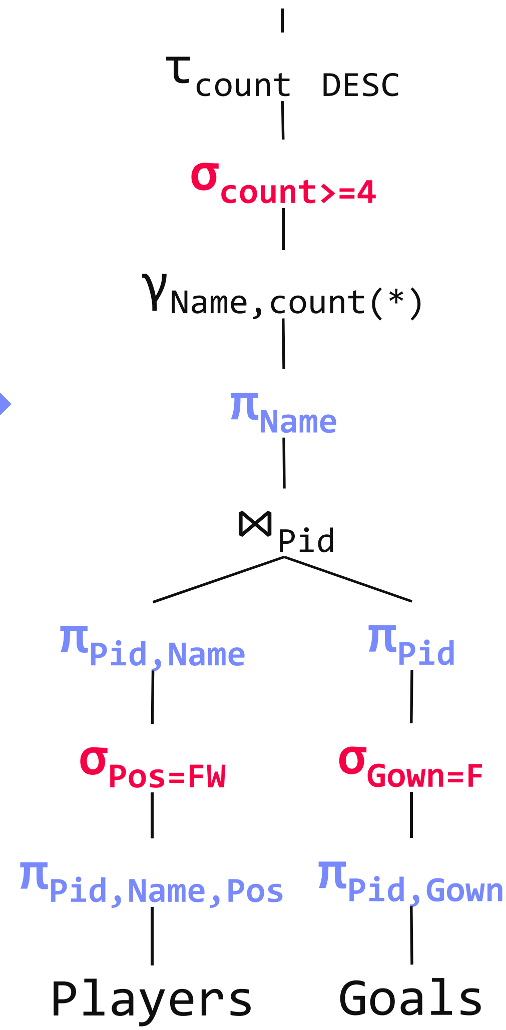
Example Query Restructuring

```
SELECT * 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

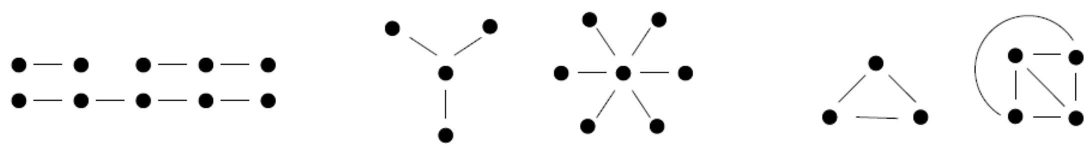
$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

  - **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)

	(estimated)	**(real)**
|	10	**590**
$\sigma_{\text{Model= 'Golf'}}$		
|	1,000	**5,000**
$\sigma_{\text{Make='VW'}}$		
|		
Cars	10,000	10,000

Join Ordering Problem

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



Chains Stars Cliques

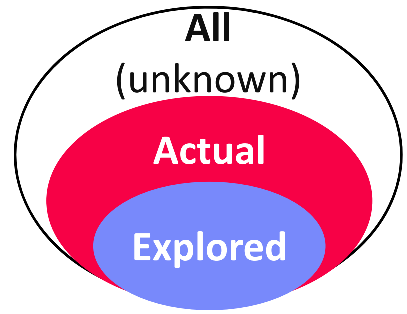
- **Search Space**

	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

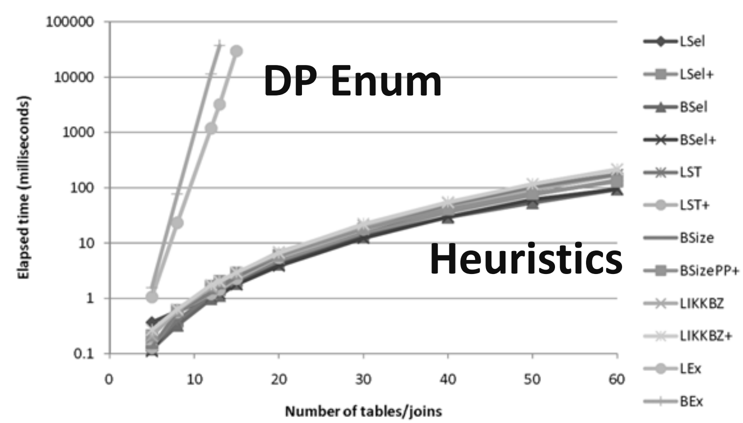
C(n) … Catalan Numbers

[Guido Moerkotte, Building Query Compilers (Under Construction), **2019**, http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf]

Join Order Search Strategies



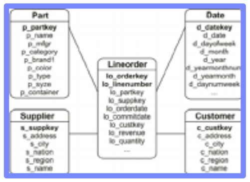
- **Tradeoff: Optimal (or good)** plan vs **compilation time**
- **#1 Naïve Full Enumeration**
  - Infeasible for reasonably large queries (long tail up to 1000s of joins)
- **#2 Exact Dynamic Programming**
  - Guarantees optimal plan, often too expensive (beyond 20 relations)
  - Bottom-up vs top-down 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 ⋈ Lineorder ⋈ Supplier ⋈ σ(Customer) ⋈ σ(Date), **left-deep plans**

#	Plan	Costs
1	Lineorder ⋈ Part	30M
	Lineorder ⋈ Supplier	20M
	Lineorder ⋈ σ(Customer)	90K
	Lineorder ⋈ σ(Date)	**40K**
	~~Part ⋈ Customer~~	N/A
	…	…
2	(Lineorder ⋈ σ(Date)) ⋈ Part	150K
	(Lineorder ⋈ σ(Date)) ⋈ Supplier	100K
	(Lineorder ⋈ σ(Date)) ⋈ σ(Customer)	**75K**

#	Plan	Costs
3	((Lineorder ⋈ σ(Date)) ⋈ σ(Customer)) ⋈ Part	120K
	((Lineorder ⋈ σ(Date)) ⋈ σ(Customer)) ⋈ Supplier	**105M**
4	**(((Lineorder ⋈ σ(Date)) ⋈ σ(Customer)) ⋈ Supplier) ⋈ 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	Plan
{C}	Tbl, IX
{D}	Tbl, IX
{L}	…
{P}	…
{S}	…

Q1+Q1

Q2	Plan
{C,L}	L⋈C, ~~C⋈L~~
{D,L}	L⋈D, ~~D⋈L~~
{L,P}	~~L⋈P~~, P⋈L
{L,S}	~~L⋈S~~, S⋈L
~~{C,D}~~	~~N/A~~
…	…

Q1+Q2, Q2+Q1

Q3	Plan
{C,D,L}	(L⋈C)⋈D, ~~D⋈(L⋈C)~~, ~~(L⋈D)⋈C~~, ~~C⋈(L⋈D)~~
{C,L,P}	~~(L⋈C)⋈P~~, P⋈(L⋈C), ~~(P⋈L)⋈C~~, ~~C⋈(P⋈L)~~
{C,L,S}	…
{D,L,P}	…
{D,L,S}	…
{L,P,S}	…

Q1+Q3, Q2+Q2, Q3+Q1

Q4	Plan
{C,D,L,P}	~~((L⋈C)⋈D)⋈P~~, P⋈((L⋈C)⋈D)
{C,D,L,S}	…
{C,L,P,S}	…
{D,L,P,S}	…

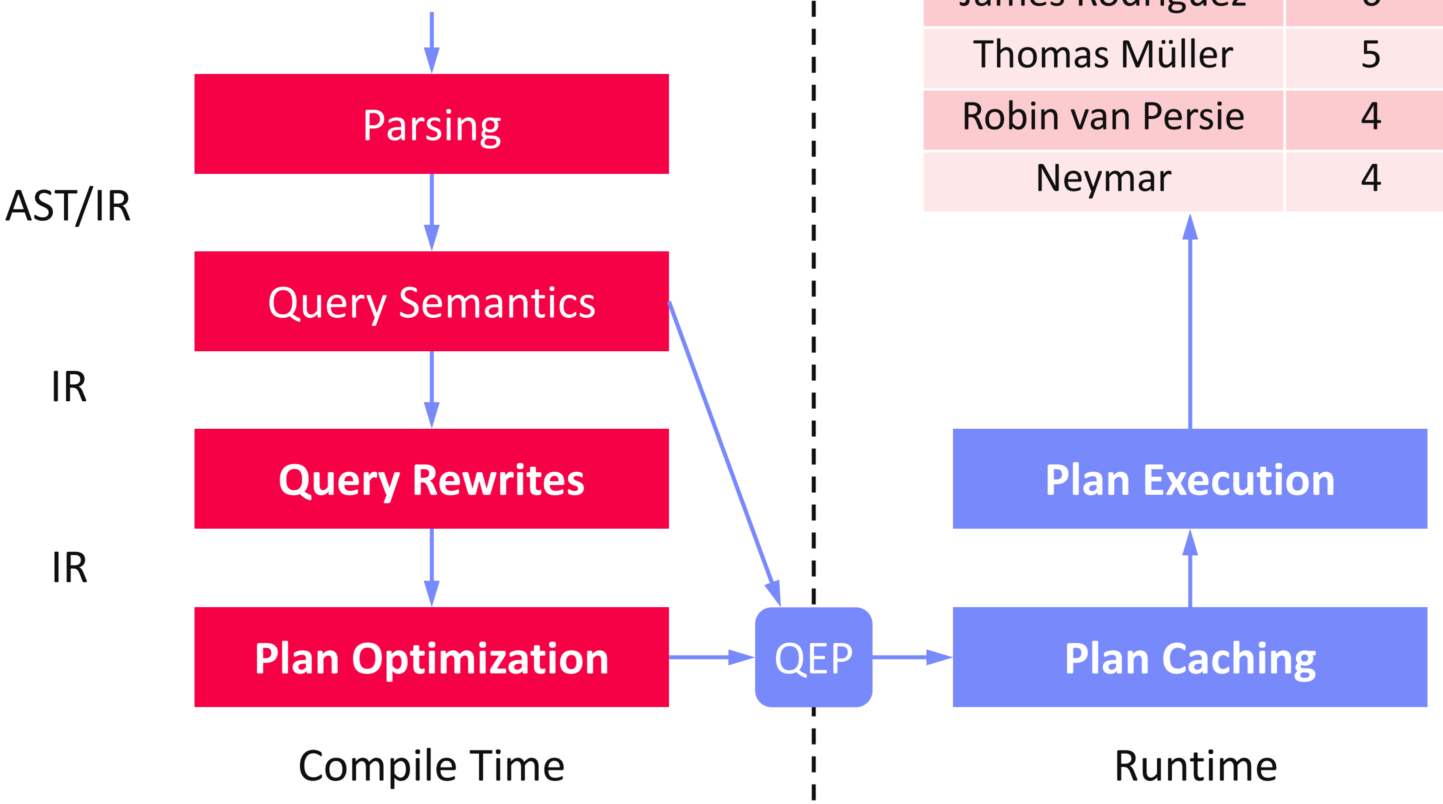
Q1+Q4, Q2+Q3, Q3+Q2, Q4+Q1

Q5	Plan
{C,D,L,P,S}	…

Plan Execution Strategies

Overview Query Processing

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



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

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)

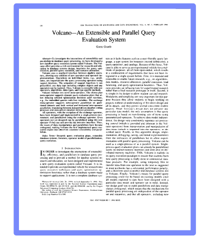
High-level overview, details in **ADBS** (applies to #2–#4)

Iterator Model

Scalable (small memory)
High CPI measures

- **Volcano Iterator Model**
  - **Pipelined & no global knowledge**
  - **Open-Next-Close** (ONC) interface
  - Query execution from root node (pull-based)

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

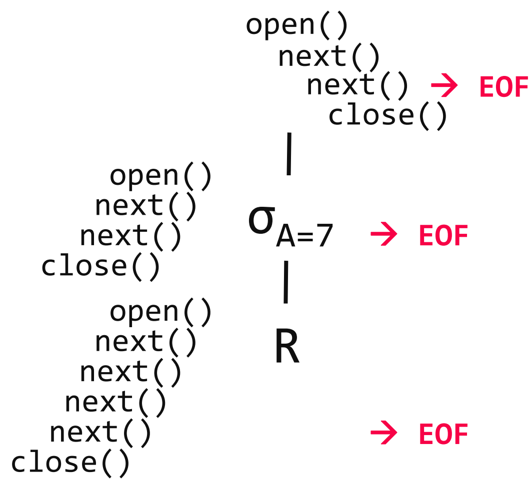


- **Example $\sigma_{A=7}(R)$**

```
void open() { R.open(); }

void close() { R.close(); }

Record next() {
  while( (r = R.next()) != EOF )
    if( p(r) ) //A==7
      return r;
  return EOF;
}
```



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

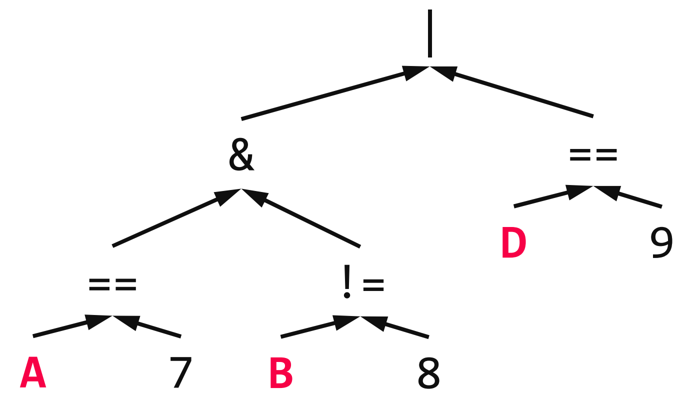
PostgreSQL: `Init()`, `GetNext()`, `ReScan()`, `MarkPos()`, `RestorePos()`, `End()`

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

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



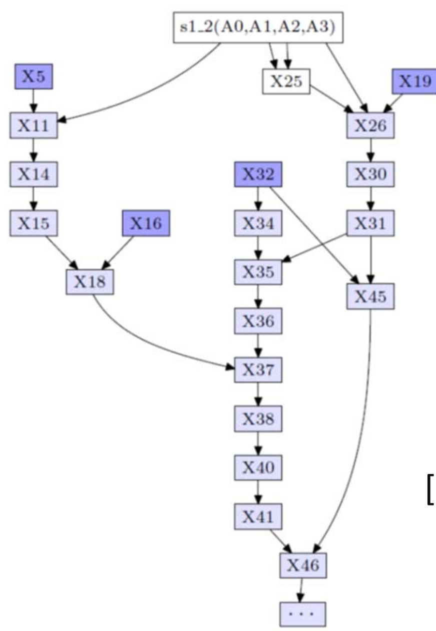
Materialized Intermediates (column-at-a-time)

```
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';
```

```
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)

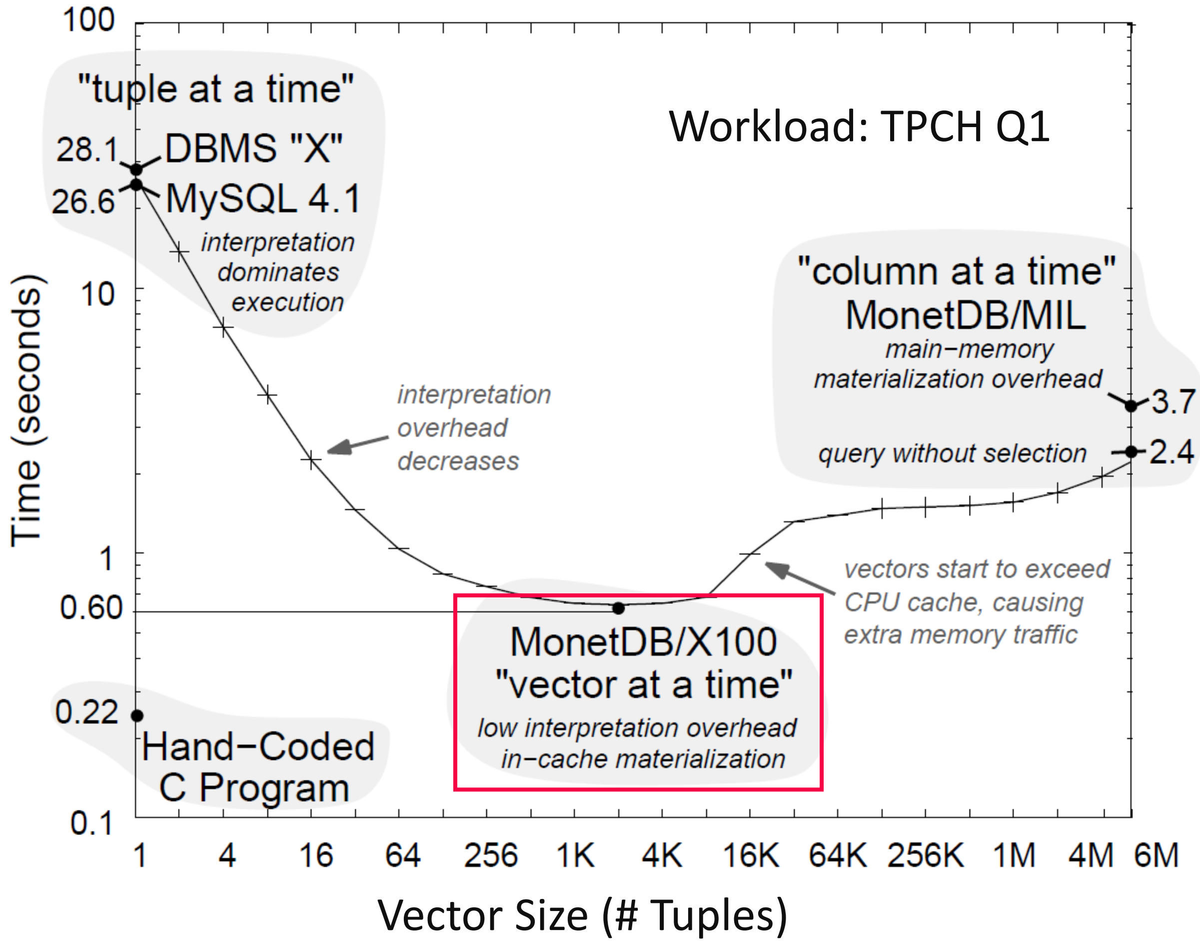
Column-oriented storage
Efficient array operations
DAG processing
Reuse of intermediates
Memory requirements
Unnecessary read/write from and to memory



[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**



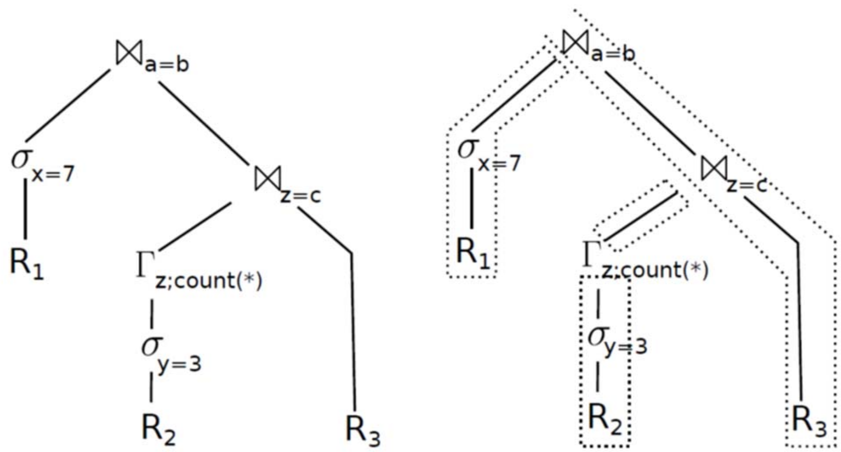
Column-oriented storage
Efficient array operations
Memory/cache efficiency
DAG processing
Reuse of intermediates

[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)



Compiled Query
(conceptual, not LLVM)

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_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 Γ_z
~~**for each** tuple t in Γ_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 $t_1 \circ t_2 \circ t_3$

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

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)

	Selection	Projection	Grouping	Join
Logical Plan Operators	$\sigma_p(R)$	$\pi_A(R)$	$\gamma_{G:agg(A)}(R)$	$R \bowtie_{R.a=S.b} S$

Physical Plan Operators	`TableScan`
	`IndexScan`
	ALL
		ALL	`SortGB`
			`HashGB`
				`NestedLoopJN`
				`SortMergeJN`
				`HashJN`

	Lecture 07			**This Lecture** **Exercise 3**

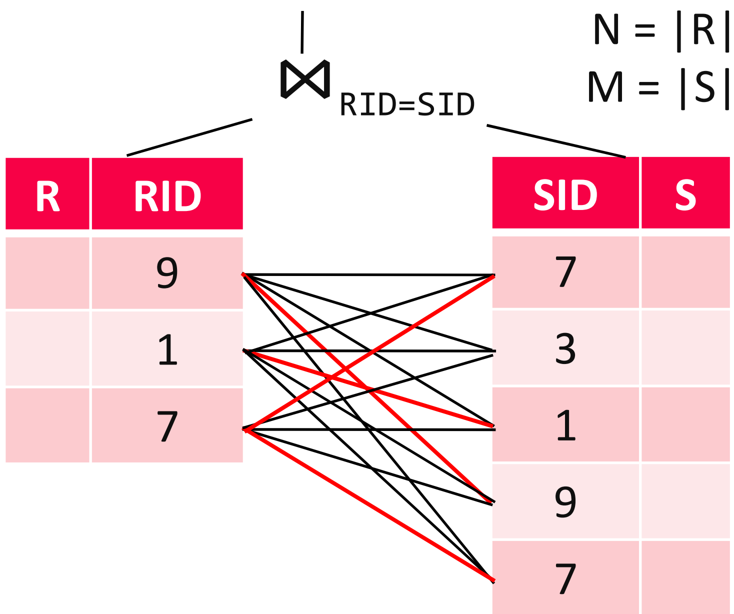
Nested Loop Join

- **Overview**
  - Most general join operator (no order, no indexes, arbitrary predicates θ)
  - Poor asymptotic behavior (very slow)

- **Algorithm** (pseudo code)

```
for each s in S
  for each r in R
    if( r.RID θ 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)

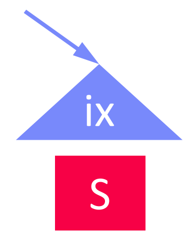
Block Nested Loop / Index Nested Loop Joins

- **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

```
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 θ 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)**

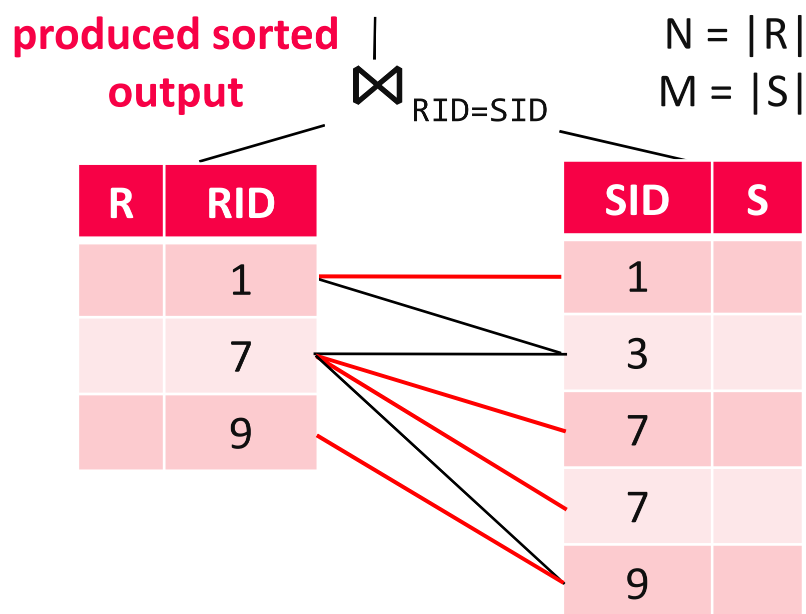
```
for each r in R
  for each s in S.IX(θ,r.RID)
    emit concat(r,s)
```



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)**

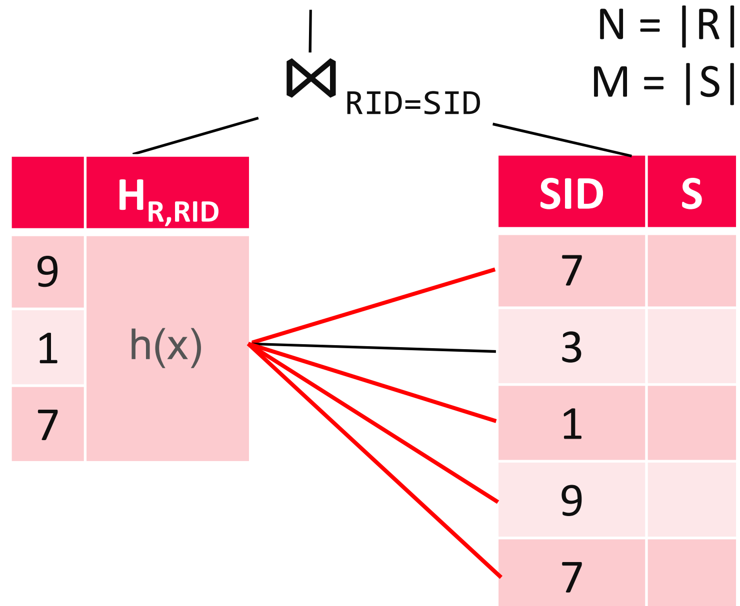
Hash Join

- **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)
  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

Conclusions and Q&A

- **Summary**
  - **Query rewriting and query optimization**
  - **Query processing and physical operators**

- **Exercise 2 Reminder**
  - Submission deadline: **May 07 11.59pm** (+ max 7 late days)
  - **Modified submission rules, but crucial to submit**

- **Next Lectures**
  - May 13: **09 Transaction Processing and Concurrency**