

Architecture of DB Systems

06 Query Processing

Matthias Boehm

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

Last update: Nov 18, 2020

Announcements/Org

■ #1 Video Recording

- Link in [TeachCenter](#) & [TUBE](#) (lectures will be public)
- Optional attendance (independent of COVID)



■ #2 COVID-19 Restrictions (HS i5)

- Corona Traffic Light: **RED**
- Temporarily webex lectures and recording



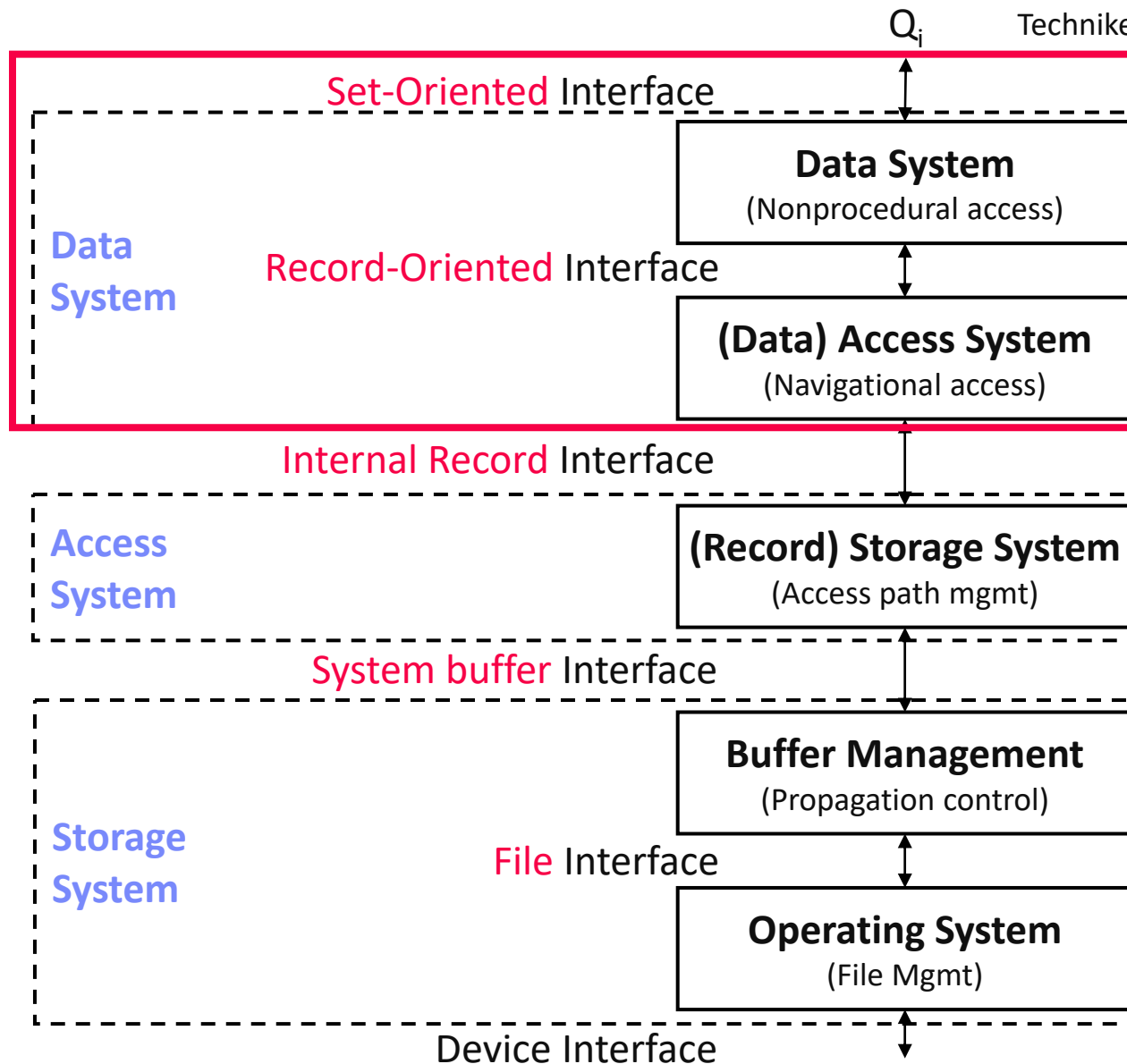
Agenda

- Overview Query Processing
- Plan Execution Strategies
- Physical Plan Operators

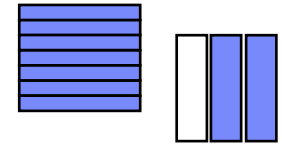
Overview Query Processing

DBMS Architecture, cont.

[Theo Härder, Erhard Rahm:
Datenbanksysteme: Konzepte und
Techniken der Implementierung, **2001**]

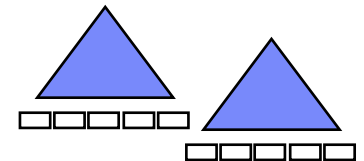


SELECT *
FROM R

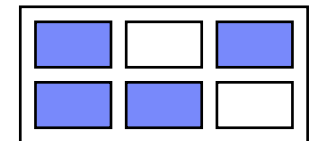


FIND NEXT
record

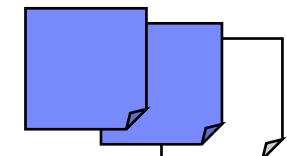
B-Tree
getNext



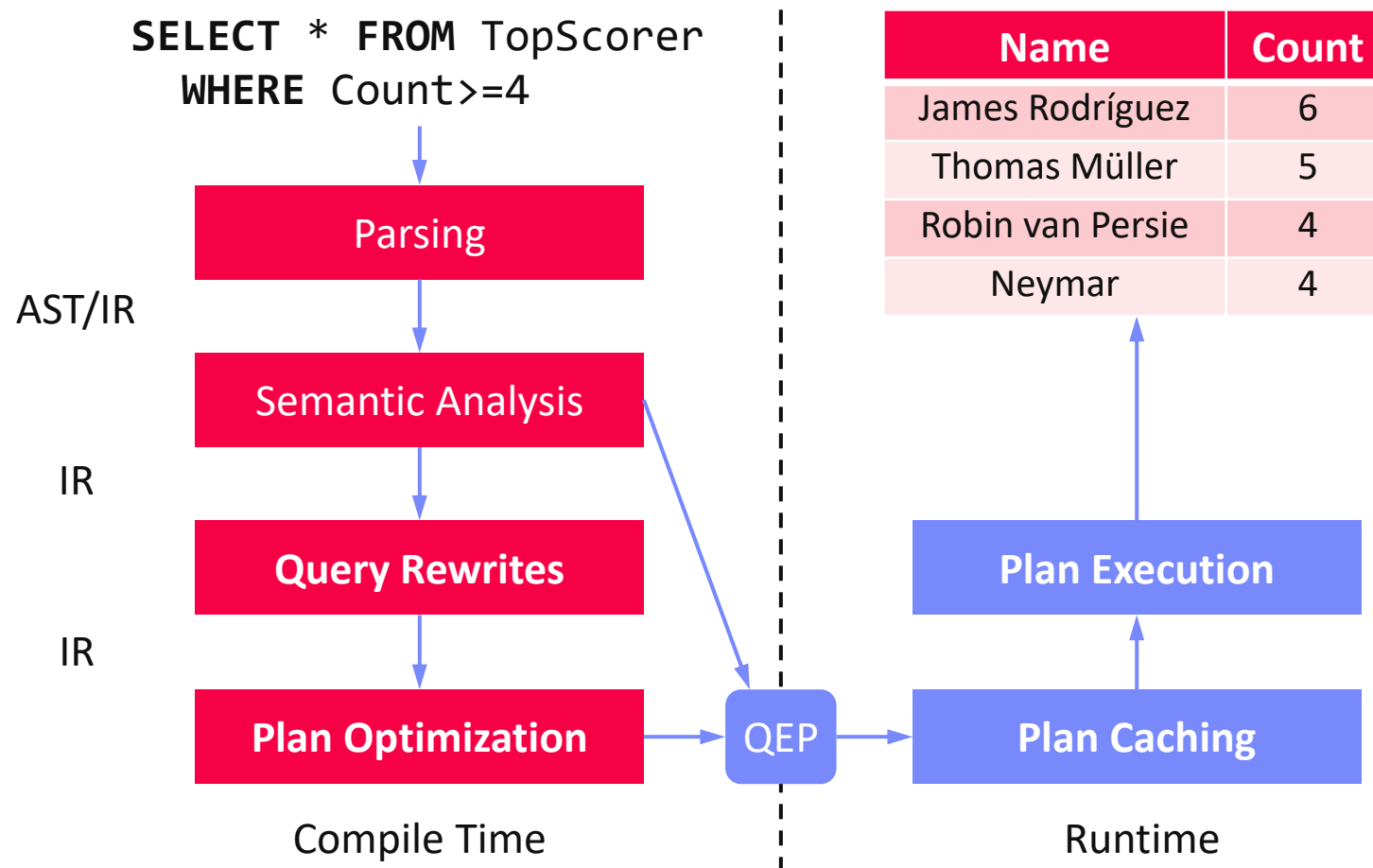
ACCESS
page j



READ
block k



Overview Query Processing



Database Catalog

[Meikel Poess: **TPC-H**. Encyclopedia of Big Data Technologies 2019]

■ Catalog Overview

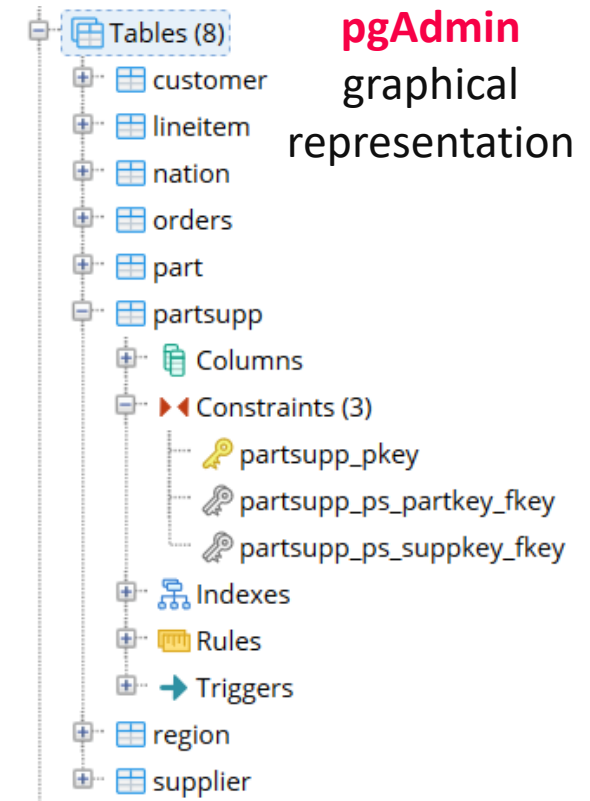
- **Meta data** of all database objects (tables, constraints, indexes) → **mostly read-only**
- **Accessible through SQL**, but internal APIs
- Organized by schemas (**CREATE SCHEMA tpch;**)

■ SQL Information_Schema

- Schema with tables for all tables, views, constraints, etc
- **Example:** check for existence of accessible table

```
SELECT 1 FROM information_schema.tables
WHERE table_schema = 'tpch'
      AND table_name = 'customer'
```

(defined as views over PostgreSQL catalog tables)



Plan Caching

Motivation

- Query rewriting, optimization and plan generation is expensive
- Cache and reuse compile plans**
(stored procedures, prepared/parameterized statements, ad-hoc queries)

Structure

- SQL query test
- Compiled query plans
- Statistics
 - Usage counts
 - Last run timestamp
 - Max/avg runtime
 - Compile time

#1 Probe Plan Cache

hash(SQL)

SQL	Plan	Stats
SELECT * FROM	QEP1: x-y-z	#5, 23ms
SELECT a FROM	QEP7: a-b	#100, 6ms
CREATE PROC	...	...

#2 Check schema valid,
statistics up-to-date

#3 Reuse or
Recompilation

- Examples:** MS SQL Server, IBM DB2

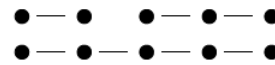
Query and Plan Types

[Guido Moerkotte, Building Query Compilers
(Under Construction), 2020,
<http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf>]

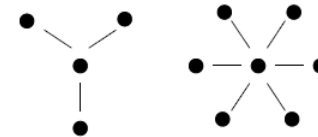


Query Types

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



Chains



Stars

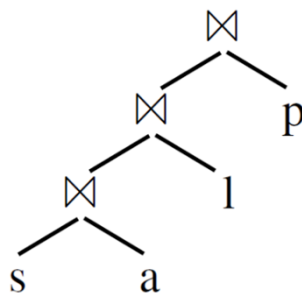


Cliques

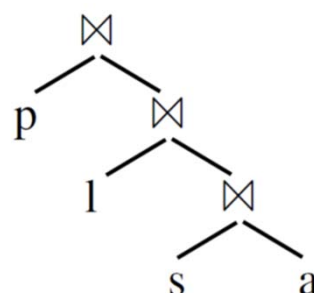
Join Tree Types / Plan Types

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

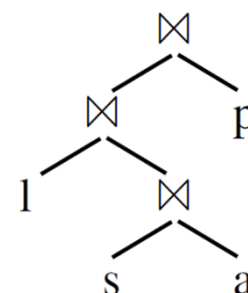
Left-Deep Tree



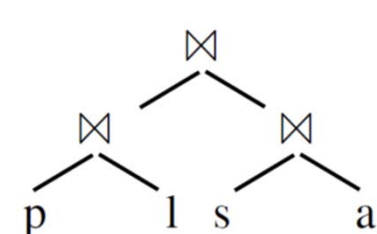
Right-Deep Tree



Zig-Zag Tree



Bushy Tree



Result Caching

■ Motivation

- Read-mostly data and same queries over unchanged inputs
- **Cache and reuse small result sets** (e.g., aggregation queries, distinct)

■ Structure

- Similar to materialized-views (cached intermediates)
- Store results of queries w/ `result_cache` hint in subarea of buffer pool, reuse via hint
- Drop cached results if underlying base data changes
- Also: Function result cache (memoization)

```
SELECT /*+ result_cache*/ *
FROM TopScorer
WHERE Count >= 4
```



Buffer pool
pages

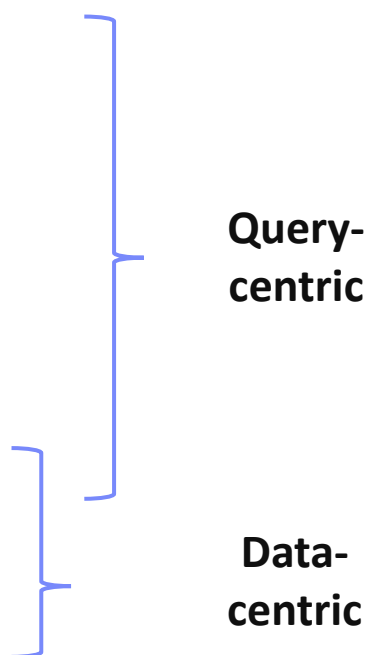
■ Examples: Oracle (from 11g)

[<https://oracle.readthedocs.io/en/latest/plsql/cache/alternatives/result-cache.html>]

Plan Execution Strategies

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)
- #5 **Data-Centric Processing** (row stores)



Query-
centric

Data-
centric

Iterator Model

Scalable (small memory)

High CPI measures

Volcano Iterator Model

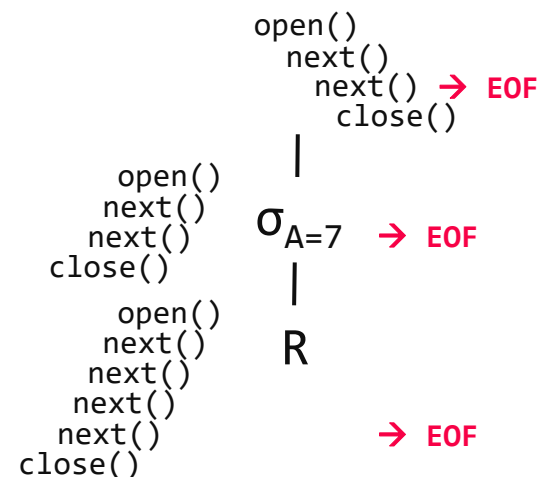
- Pipelined & no global knowledge
- Open-Next-Close (ONC) interface
- Query execution from root node (pull-based)

[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

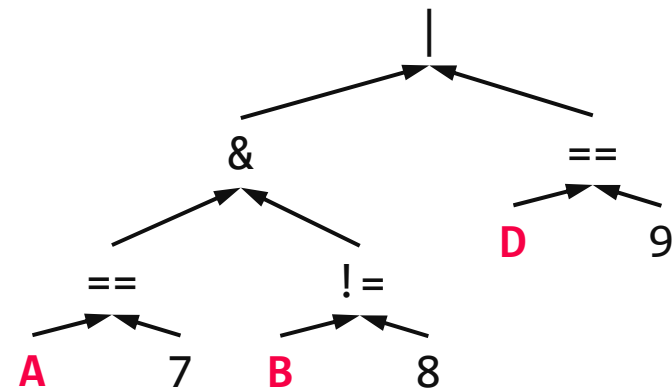
Operator Predicates

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

Example Selection σ

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

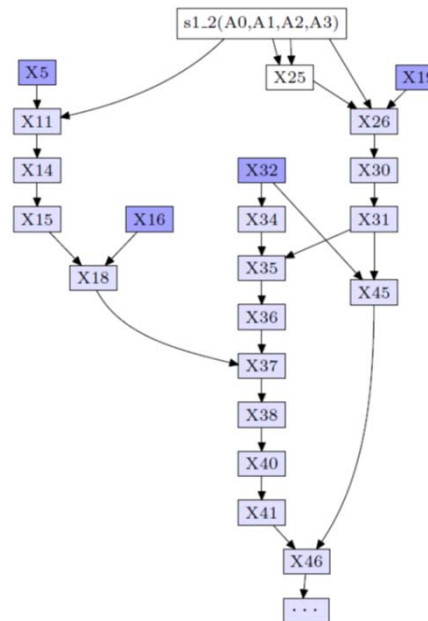
A	B	C	D
7	8	Product 1	10
14	8	Product 3	11
7	3	Product 7	7
3	3	Product 2	1



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';
```

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.useselect(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;
```

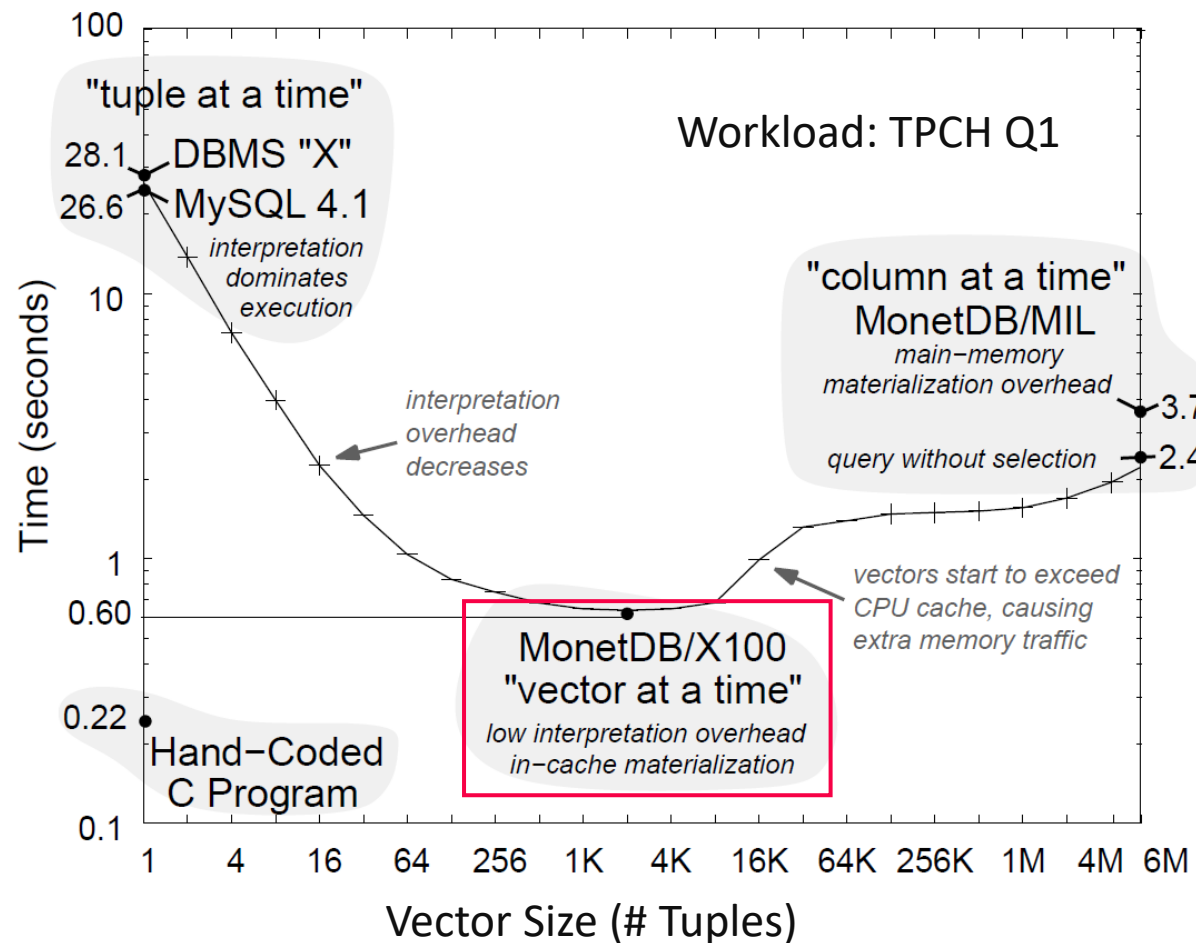
**Binary
Association
Tables**
(BATs:=OID/Val)

[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]

Vectorized Execution (vector-at-a-time), cont.

■ Motivation

- **Iterator Model:** many function calls, no instruction-level parallelism
- **Materialized:** mem-bandwidth-bound

■ Hyper-Pipelining

- Operators work on vectors
- Pipelining of vectors (sub-columns)
- Vector sizes according to cache size
- Pre-compiled function primitives

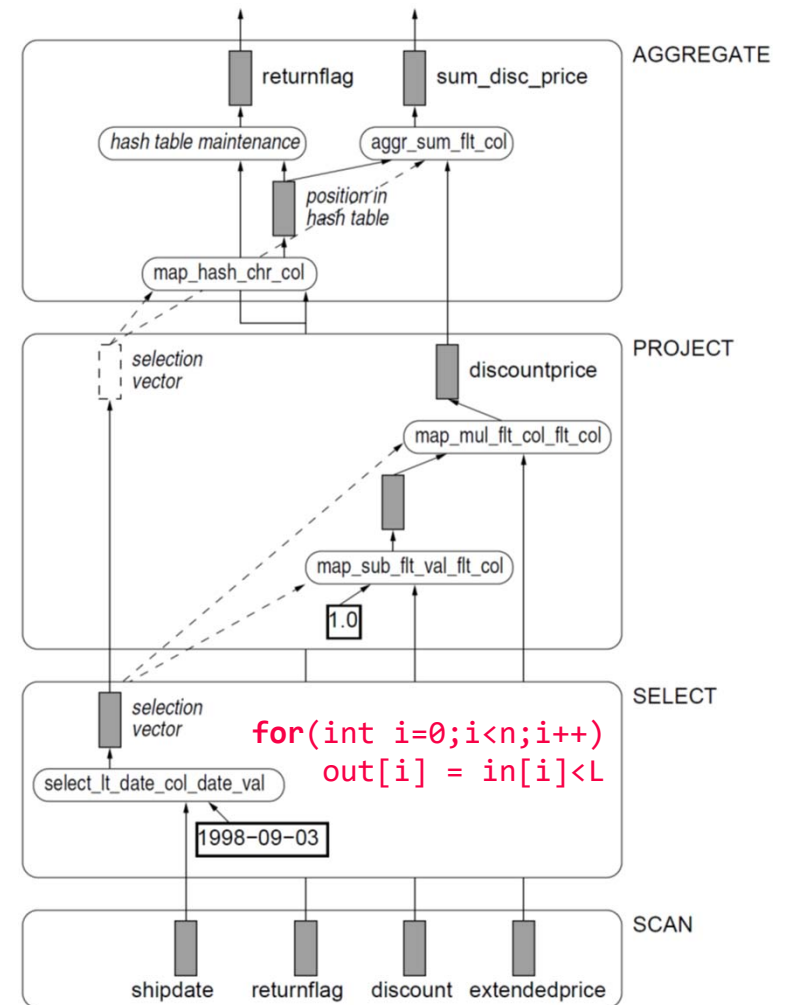
➔ Generalization of execution strategies



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



[Marcin Zukowski, Peter A. Boncz, Niels Nes, Sándor Héman: MonetDB/X100 - A DBMS In The CPU Cache. **IEEE Data Eng. Bull. 28(2), 2005**]



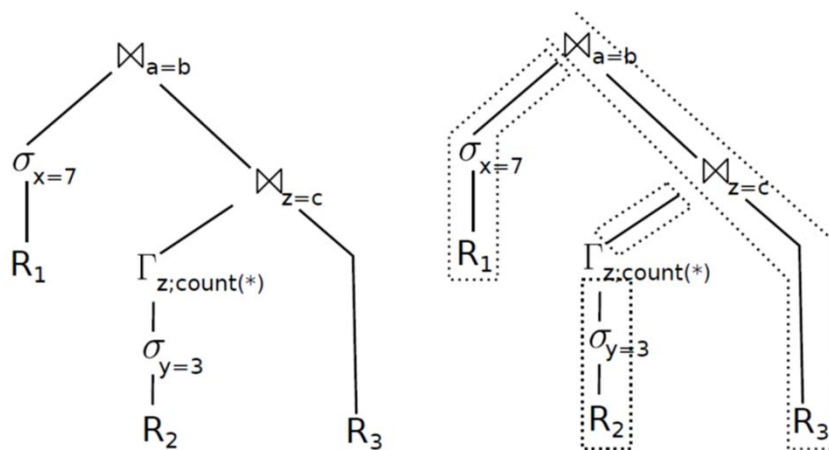
Query Compilation

07 Query Compilation and Parallelization

- 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)

```

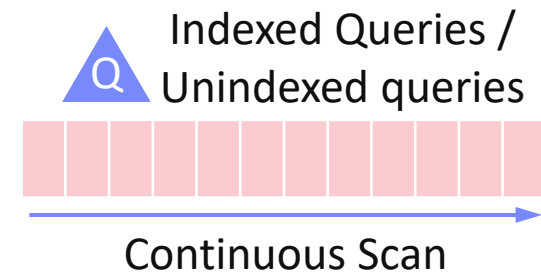
initialize memory of  $\bowtie_{a=b}$ ,  $\bowtie_{c=z}$ , and  $\Gamma_z$ 
for each tuple  $t$  in  $R_1$ 
  if  $t.x = 7$ 
    materialize  $t$  in hash table of  $\bowtie_{a=b}$ 
for each tuple  $t$  in  $R_2$ 
  if  $t.y = 3$ 
    aggregate  $t$  in hash table of  $\Gamma_z$ 
for each tuple  $t$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

Data-Centric / Continuous Scan Processing

■ Crescendo (ETH Zurich)

- **Amadeus use case:** latency <2s, freshness <2s, query diversity/update load, linear scale-out/scale-up
- **ClockScan:** cooperative scan
- **Index Union Update Join:** update-data join (write, and read cursor)

[Philipp Unterbrunner et al.: Predictable Performance for Unpredictable Workloads. **PVLDB 2(1) 2009**]

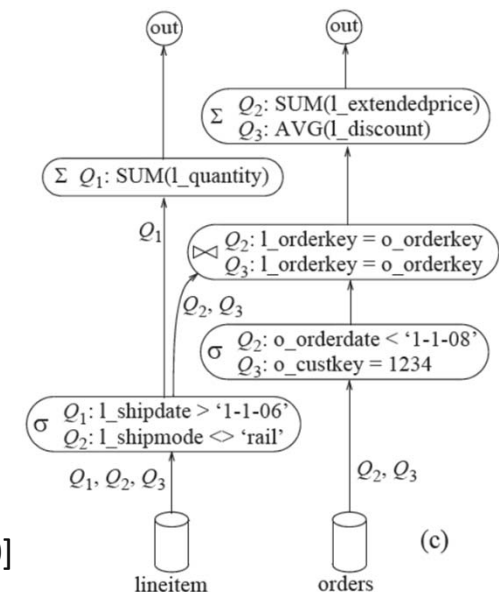


■ DataPath System (Rice University)

- Push-based, data-centric processing model
- Multi-query optimization → DAG of operations (tuple bit-string to relate tuples to queries)
- I/O system pushed chunks to operators
- Load shedding on overload and explicit scheduling



[Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, Luis Leopoldo Perez: The DataPath system: a data-centric analytic processing engine for large data warehouses. **SIGMOD 2010**]



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)

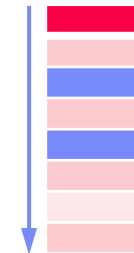
Logical Plan Operators	Selection $\sigma_p(R)$	Projection $\pi_A(R)$	Grouping $\gamma_{G:agg(A)}(R)$	Join $R \bowtie_{R.a=S.b} S$
				
Physical Plan Operators	TableScan IndexScan ALL	ALL	SortGB HashGB	NestedLoopJN SortMergeJN HashJN

Table and Index Scan

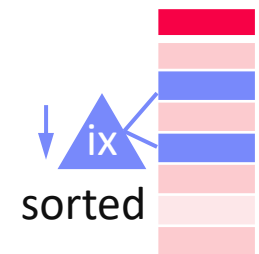
Table Scan vs Index Scan

- For highly selective predicates, index scan **asymptotically much better** than table scan
- Index scan **higher per tuple overhead** (break even ~5% output ratio)

Table Scan



Index Scan



Index Scan

Example $\sigma_{7 \leq A \leq 106}(R)$

- IX ASC on A

RID List Handling

- IX often returns TIDs
- Fetch, Sort + Fetch
- AND:** $RIDs(x) \cap RIDs(y)$
- OR:** $RIDs(x) \cup RIDs(y)$

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

```
void close() { IX.close(); }
```

```
Record next() {  
    if(r == null)
```

```
        return r=IX.get(Low);           // A=7
```

```
        if((r=IX.next()).K ≤ Upper)    // A≤106
```

```
            return r;
```

```
        return EOF;
```

```
}
```

Nested Loop Join

Overview

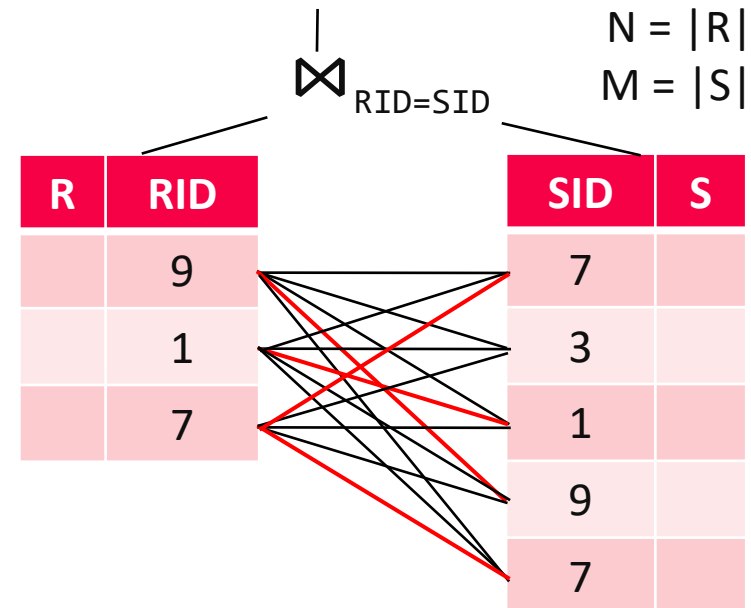
- **Most general join operator** (no order, no indexes, arbitrary predicates θ)
- **Poor asymptotic behavior** (very slow)

Algorithm (pseudo code)

```

for each s in S
  for each r in R
    if( r.RID  $\theta$  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

```

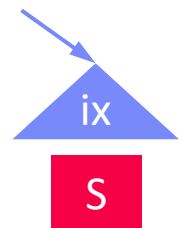
for each block  $b_R$  in R
  for each block  $b_S$  in S
    for each  $r$  in  $b_R$ 
      for each  $s$  in  $b_S$ 
        if(  $r.RID \theta s.SID$  )
          emit concat( $r, s$ )
  
```

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

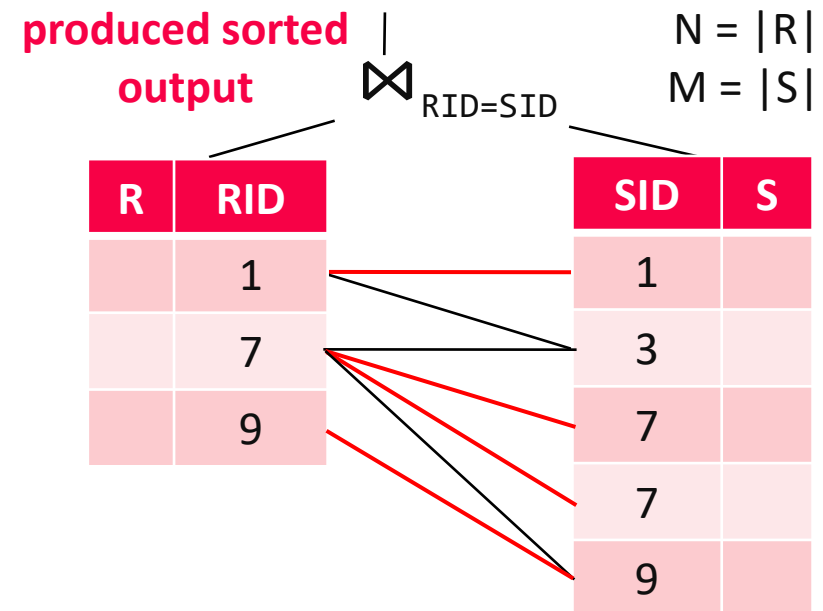
- **Sort Phase:** sort the input tables R and S (w/ external sort algorithm)
- **Merge Phase:** step-wise merge with lineage scan

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)

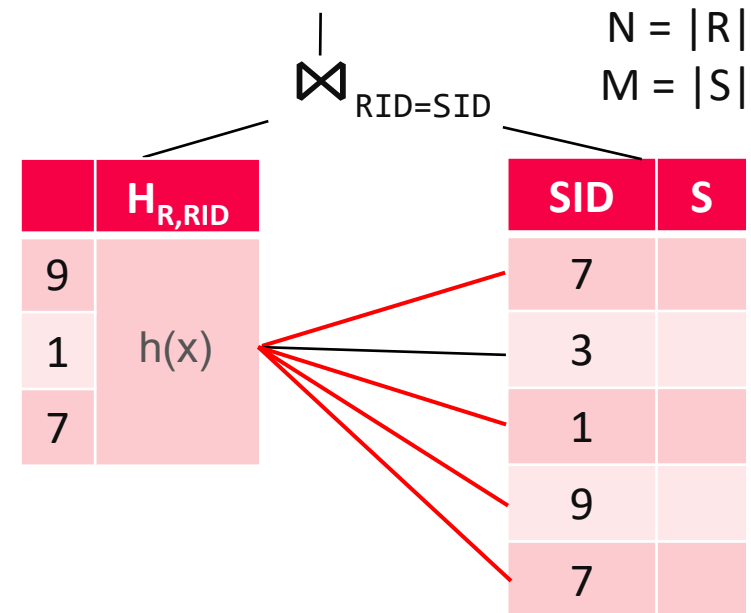
```

Record next() {
    // build phase (first call)
    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

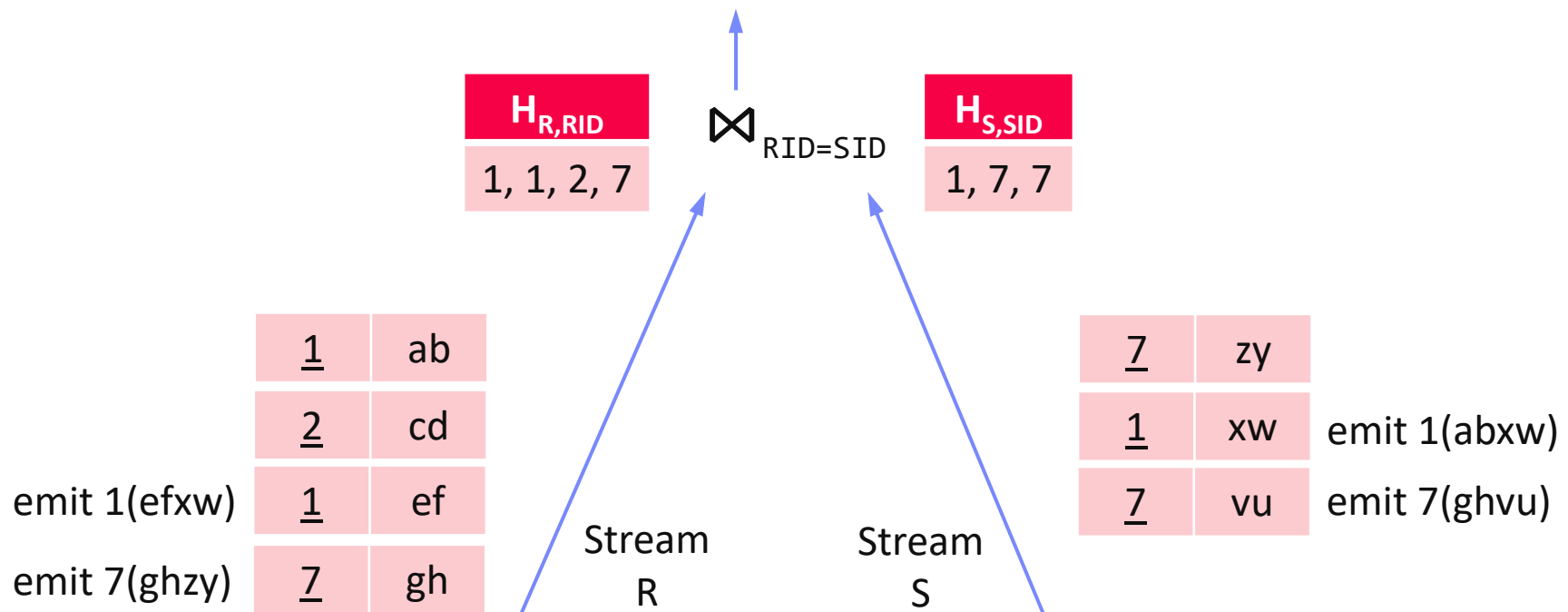
Double-Pipelined Hash Join

[Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, Daniel S. Weld:
An Adaptive Query Execution System
for Data Integration. **SIGMOD 1999**]



Overview and Algorithm

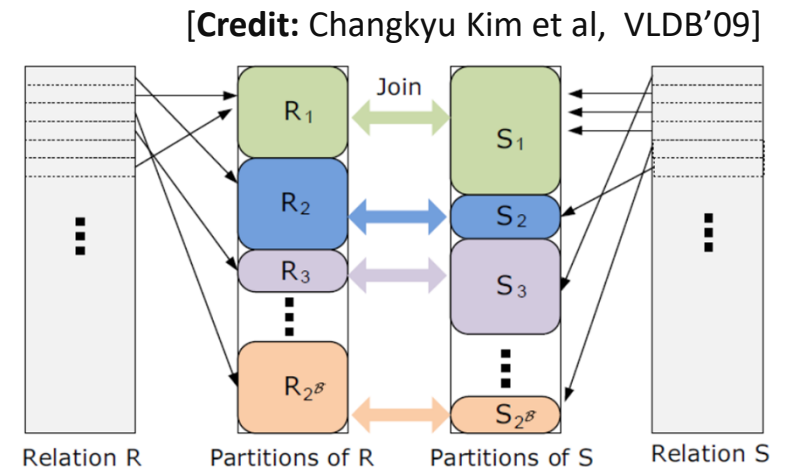
- Join of bounded streams (or unbounded w/ time-based invalidation)
- Equi join predicate**, **symmetric and non-blocking**
- For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



Partitioned Hash Join

Range-Partitioned

- Co-partitioning tuples from R and S into partitions defined by key ranges
- Local hash join over partitions
- Fit local hash table in caches
- Partitioning shuffles rows/RIDs

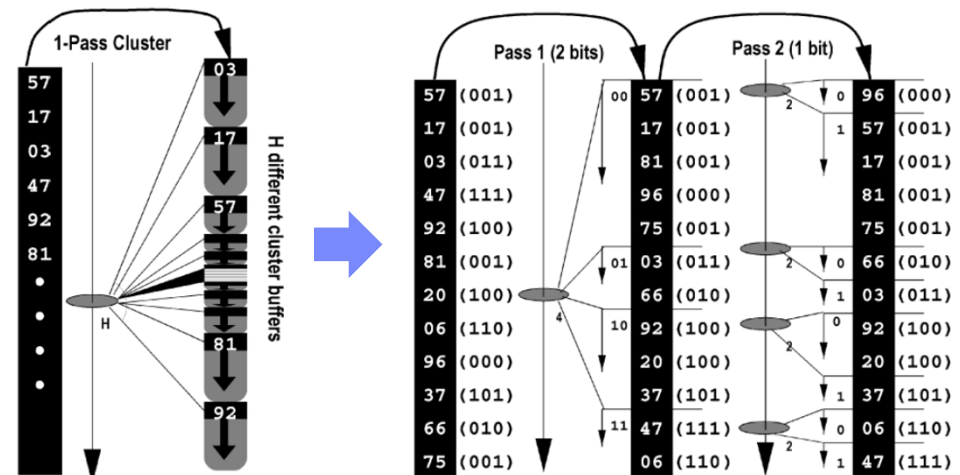


Radix Hash Join

- Multi-pass radix partitioning (first 2,3,etc bits of hash)
- Better locality during partitioning (TLB, L1/L2)



[Stefan Manegold, Peter A. Boncz, Martin L. Kersten: Optimizing Main-Memory Join on Modern Hardware. IEEE Trans. Knowl. **Data Eng.** 14(4) 2002]



Hash vs Sort-Merge Joins, Revisited ... Revisited

■ PVLDB'09



[Changkyu Kim et al: Sort vs. Hash **Revisited**: Fast Join Implementation on Modern Multi-Core CPUs. **PVLDB 2(2) 2009**]

■ PVLDB'12



[Martina-Cezara Albutiu et al: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. **PVLDB 5(10) 2012**]

■ PVLDB'13 / TKDE'15



[Cagri Balkesen, Gustavo Alonso, Jens Teubner, M. Tamer Özsu: Multi-Core, Main-Memory Joins: Sort vs. Hash **Revisited**. **PVLDB 7(1) 2013**]

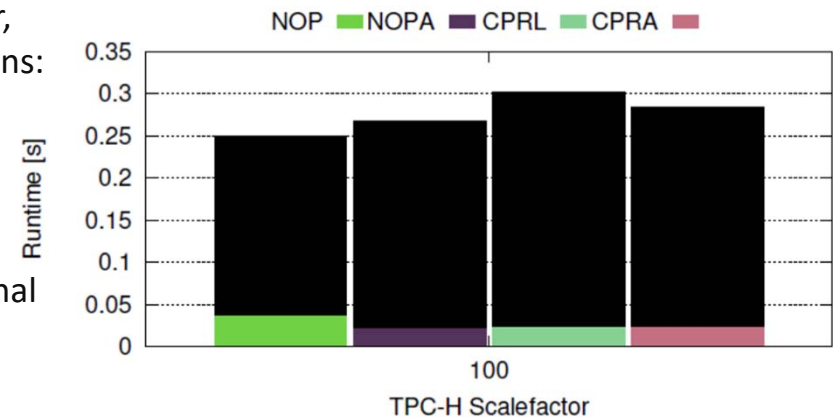
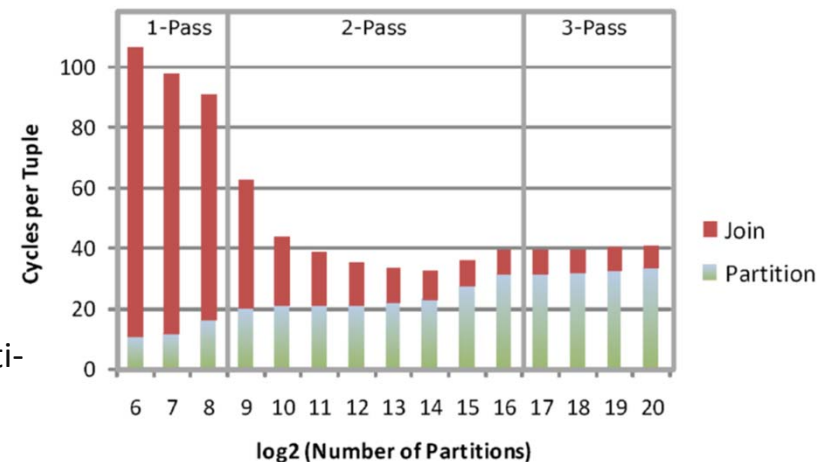
■ SIGMOD'16



[Stefan Schuh, Xiao Chen, Jens Dittrich: An Experimental Comparison of **Thirteen** Relational Equi-Joins in Main Memory. **SIGMOD 2016**]

■ Interesting Perspective

- Large-small table joins
- Comparison by query runtime



[Thomas Neumann: Comparing Join Implementations, <http://databasearchitects.blogspot.com/2016/04/comparing-join-implementations.html>, 04/2016]

Sort-GroupBy and Hash-GroupBy

Recap: Classification of Aggregates (DM, DIA)

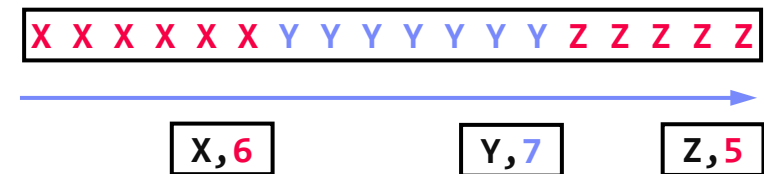
- Additive, semi-additive, additively-computable, others

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

Sort Group-By

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

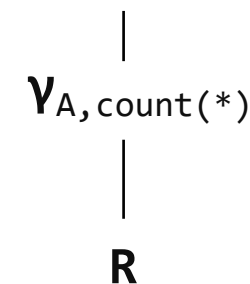
sort
 $O(N \log N)$
aggregate
 $O(N)$



Hash Group-By

- Similar to hash join (HashAggregate)
- Higher temporary memory consumption
- Unsorted group output

build & agg
 $O(N)$



	$H_{A, \text{Agg}}$
Y	
X	
Z	

- #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

- Overview Query Processing
- Plan Execution Strategies
- Physical Plan Operators

- Next Lectures (Part B)
 - **Nov 25:** no lecture, work on your programming projects
 - Sorry, SIGMOD Revision Nov 23, SIGMOD Industry Nov 25, BTW meta reviews Nov 20, SIGMOD first round reviews Nov 25
 - **07 Query Compilation and Parallelization** [Dec 02]
 - **08 Query Optimization I** (normalization, rewrites, unnesting) [Dec 09]
 - **09 Query Optimization II** (cost models, join ordering) [Dec 09]
 - **10 Adaptive Query Processing** [Dec 16]