TECHNISCHE UNIVERSITÄT DRESDEN

Dresden Database Research Group

5

Dirk Habich

Scalable Data Management (SDM)

Modern Database System Architecture

Previous focus

Dresden Database Research Group

1 Foundations
- Relational Data and SQL
- Traditional Database Architecture
- Transactions

2 Parallelism
- Data-oriented Architecture
- Coarse- and Fine-grained parallelism
- Data Fragmentation

3 Transactions
- Data Replication
- Distributed Transactions
- CAP-Theorem

4 Additional
- Map/Reduce and Hadoop
- Spark and its Ecosystem
- Database in the cloud

New focus – Modern Database Systems

Dresden Database Research Group

1 Foundations

- Relational Data and SQL
- Traditional Database Architecture
- Transactions

2 Parallelism

- Data-oriented Architecture
- Coarse- and Fine-grained parallelism
- Data Fragmentation

3 Transactions

- Data Replication
- Distributed Transactions
- CAP-Theorem

4 Additional

- Map/Reduce and Hadoop
- Spark and its Ecosystem
- Database in the cloud

Importance of Complex Data Analyses

Dresden Database Research Group

Industry 4.0

Electronic Patient Records

Business Intelligence

Trends to Face

Dresden Database Research Group

Application trends

Complex analytical queries

Example: Star Schema Benchmark (SSB) Query 3.1

```
SELECT c_nation, s_nation, d_year, SUM(lo_revenue)
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_orderdate = d_datekey
  AND c_region = 'ASIA' AND s_region = 'ASIA'
  AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year ASC, revenue DESC;
```

Hardware trends

Main Memory

Increasingly large capacities

Access only a few columns

In-Memory Column-Store

Store all data in main memory

Recap: Traditional Row-based Storage

Physical Data Storage

- Properties: The unit of a tuple with all attributes is preserved

Pro (+) / Cons (-)

\+ Easy to add/modify a tuple

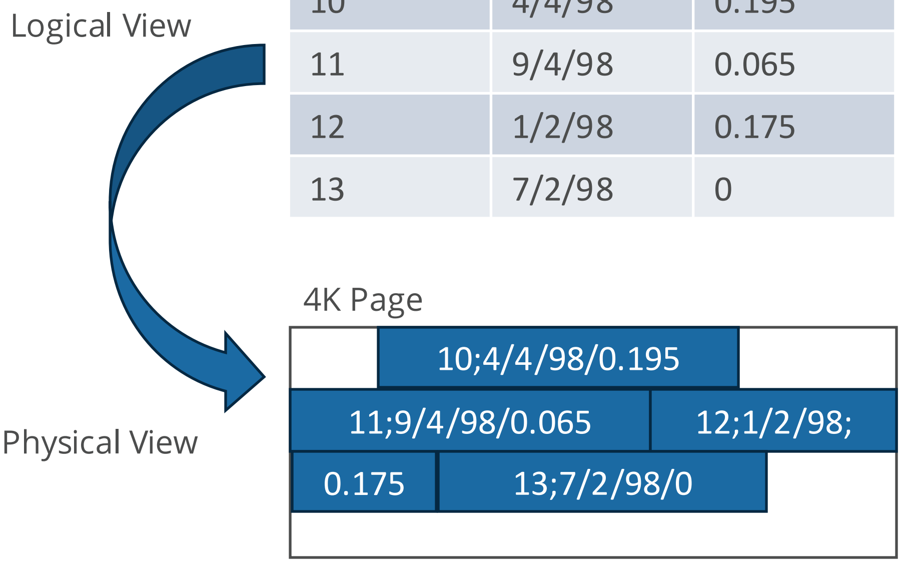
\- Even if only a few attributes are needded, all attributes have to be read
→ unnecessary I/O-cost

\- Compression is hard since consecutive values are from different domains (e.g., name, age, street addr, zip code, etc).

Illustration

Table A

ID	Day	Discount
10	4/4/98	0.195
11	9/4/98	0.065
12	1/2/98	0.175
13	7/2/98	0



Modern Column-based Storage

Dresden Database Research Group

Physical Data Storage

- Storage model based on vertical fragmentation
- Instead of storing all attributes of each relational tuple in one record, each column is stored in a separate table called BAT
- BAT = Binary Association Tuple
  - The left column is called head and is required to reconstruct the relational tuples
  - The right column is called tail and is holding the actual attribute

Illustration

Table A

ID	Day	Discount
10	4/4/98	0.195
11	9/4/98	0.065
12	1/2/98	0.175
13	7/2/98	0

Vertical Fragmentation

OID	ID
100	10
101	11
102	12
103	13

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98

OID	ID
100	0.195
101	0.065
102	0.175
103	0

Modern Column-based Storage/2

Physical Data Storage

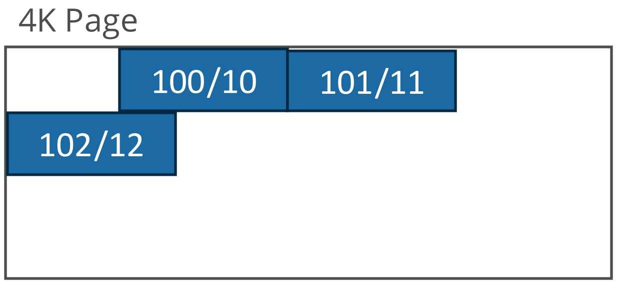
- Every relational table is internally represented as a collection of BATS
- For a table of k attributes, there exists k BATs and each BAT stores the attribute as (OID, value) pairs
- System generated OID values identifies the relational tuple that the attribute value belongs to, i.e., all attribute values of a single tuple are assigned the same OID.
- BATs are stored in regular 4K-pages → each BAT considers as single table with two attributes

Illustration

OID	ID
100	10
101	11
102	12
103	13

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98

OID	ID
100	0.195
101	0.065
102	0.175
103	0



Modern Column-based Storage/3

Pro / Cons

+ Only need to read in relevant data, more efficient scan operators

- tuple writes require multiple accesses

- Reading all columns of a single row requires expensive row-reconstruction (1:1 join)

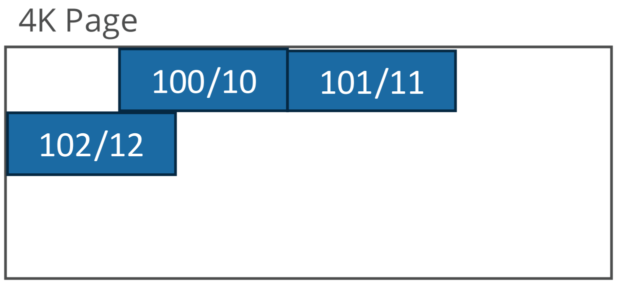
- High storage overhead due to additional OID attribute

Illustration

OID	ID
100	10
101	11
102	12
103	13

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98

OID	ID
100	0.195
101	0.065
102	0.175
103	0



Columnar Storage Optimization

Storage Optimization

Dresden Database Research Group

(1) Elimination of OIDs

- Implicit instead of explicit OIDs
- The order of the tuples is now crucial

Advantages

- Each attribute is now a simple array without any additional information
- Each array has a unique data type
- OID could be on-the-fly generated based on the position within the array

(2) Compression

- Compression easily possible for each single array

OID	ID
100	10
101	11
102	12
103	13

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98

OID	ID
100	0.195
101	0.065
102	0.175
103	0

ID
10
11
12
13

Day
4/4/98
9/4/98
1/2/98
7/2/98

ID
0.195
0.065
0.175
0

Compression Overview

Storing Relational Data Column-wise

- Each column as a sequence of values

	Integers	Fixed Point	Strings
ID	**Number of Items**	**Total Order Prize**	**Shipping Mode**
1	5	173 665.47	“Mail”
2	10	46 929.18	“Air”
3	3	193 846.25	“Rail”

✓ **Only need to read in relevant data attributes during query processing**

– Tuple writes require multiple accesses

Reducing Memory Footprint

- All logical data is mapped to integer data on the physical level

All Integers (64-Bit)

ID	Number of Items	Total Order Prize	Shipping Mode
1	5	17 366 547	10
2	10	4 692 918	20
3	3	19 384 625	30

DICT

768-Bit

Lightweight Integer Compression

ID	Number of Items	Total Order Prize	Shipping Mode
01	010	11 000 001 0 110 001 0 01 011 101	01010
10	111	01 000 111 1 001 100 1 10 110 010	10100
11	000	11 100 000 0 010 110 1 01 111 011	11110

102-Bit

Preprocessing Technique: DICT

Dresden Database Research Group

Dictionary Coding – DICT

- replaces each (string-)value by its unique key in a dictionary
- achieve small integer values

City
Berlin
Berlin
Berlin
Ilmenau
Ilmenau
Ilmenau
Magdeburg
Rostock
Rostock

City-Code
1
1
1
2
2
2
3
4
4

Dictionary

City	City-Code
Berlin	1
Ilmenau	2
Magdeburg	3
Rostock	4

Preprocessing Technique: FOR

Dresden Database Research Group

Frame-of-Reference – FOR

- represents each value as the difference to a certain given reference value (FOR)
- Achieve smaller integer values

Preis
45
54
48
55
51
53
40
50
49

FOR = 40 →

Preis (FOR)
5
14
8
15
11
13
0
10
9

Reference values (FOR) has to be stored to be able to decompress data

Preprocessing Technique: DELTA

Dresden Database Research Group

DELTA Coding – DELTA

- represents each value as the difference to its predecessor value
- achieve smaller integer values
- Sorting helpful for optimized compression ratio

Preis
10
15
20
21
23
25
30

DELTA
10
5
5
1
2
2
5

Compression Technique: RLE

Dresden Database Research Group

Run Length Encoding – RLE

- tackles uninterrupted sequences of occurrences of the same value, so called runs
- each run is represented by its value and length

City
Berlin
Berlin
Berlin
Ilmenau
Ilmenau
Ilmenau
Magdeburg
Rostock
Rostock

Run Value	Run Length
Berlin	3
Ilmenau	3
Magdeburg	1
Rostock	2

Compression Technique: NS

Dresden Database Research Group

Null Suppression – NS

- addresses the physical level of bits or bytes to reduce the number of bits per value

DICT

City
Berlin
Berlin
Berlin
Ilmenau
Ilmenau
Ilmenau
Magdeburg
Rostock
Rostock

→

City-Code
1
1
1
2
2
2
3
4
4

3 Bits per value

NS →

Column Code
001
001
001
010
010
010
011
100
100

Null Suppression Algorithms

Dresden Database Research Group

Bit-Aligned

- compress an integer value with a minimal number of bits

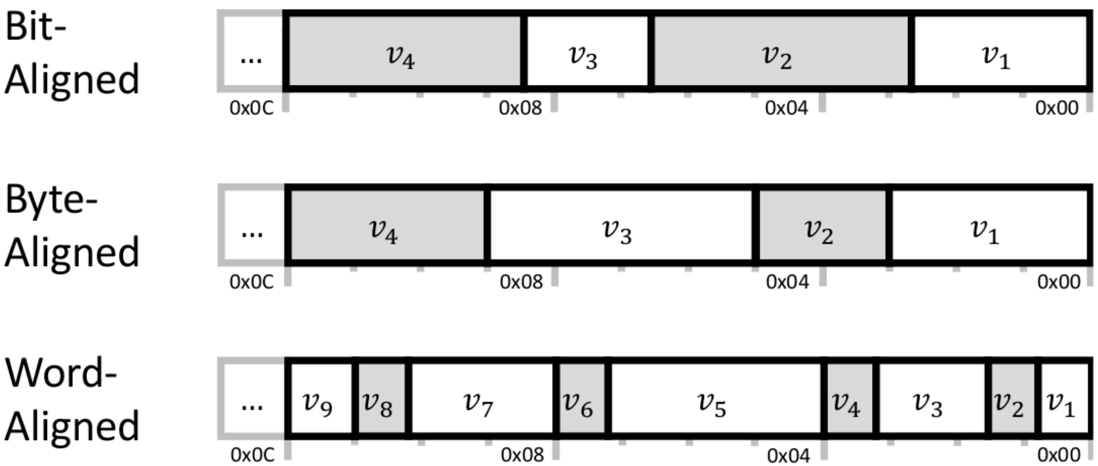
Byte-Aligned

- Compress an integer value with a minimal number of bytes

Word-Aligned

- Encode as many integer values as possible into 32-bit or 64-bit words

Alignment Examples

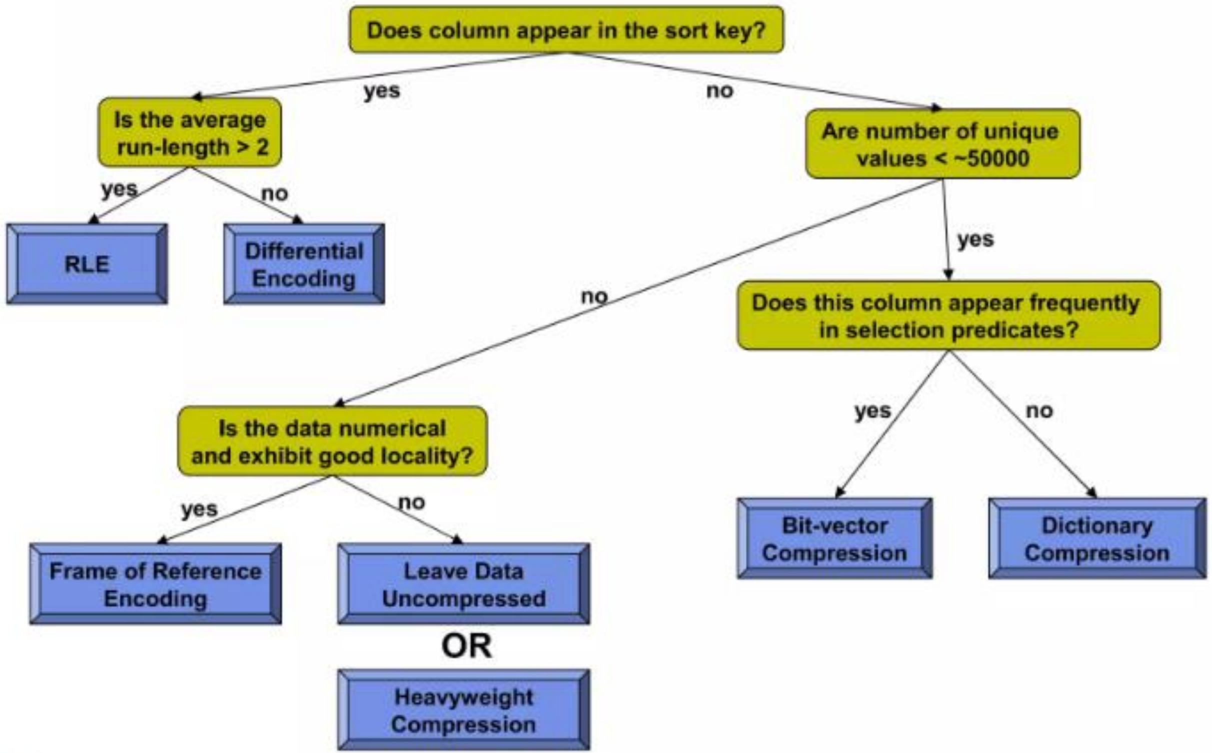


Compression Techniques Summary

RLE	DELTA	FOR	DICT	NS
Run Length Encoding Replace run by value & length	**Delta Coding** Replace data elem. by difference to predecessor	**Frame-of-Reference** Replace data elem. by difference to reference value	**Dictionary Coding** Replace data elem. by key in dictionary	**Null Suppression** Eliminate leading zero-bits
Repetitions	Sorting	Value range	# distinct values	Small integers

Logical	Level	Physical
natural numbers	data	bits and bytes
preprocessing	*role*	*actual compression*

What Compression Scheme To Use?



Storage Summary

Dresden Database Research Group

Most Important Properties

- Every column is encoded as a sequence of integer values
- Every column is stored in a contiguous memory area
- Heavily use of integer compression
  - to fit more data in main memory
  - To increase effective bandwidth

Impact on Query Processing

- Only relevant columns have to be accessed
- Query processing is mostly done on integer sequences

Processing Model

Processing Model

Dresden Database Research Group

A processing model

- Defines how the system execute a query plan and moves data from one operator to the next

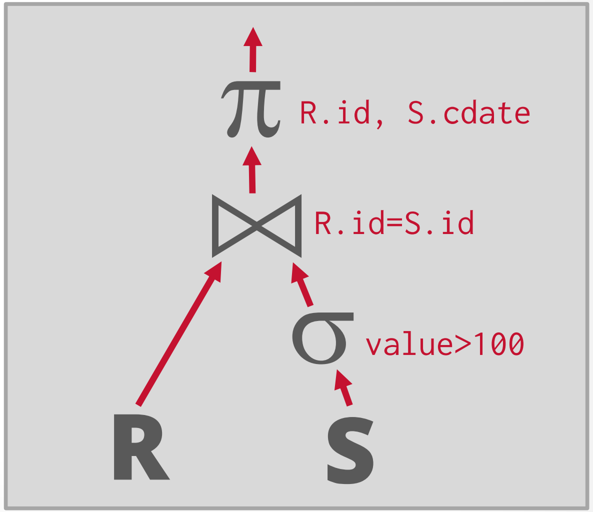
Each processing model

- Is comprised of two types of execution paths
  → Control Flow: How the system invokes an operator
  → Data Flow: How an operator send its result

The output

- of an operator can be either whole tuples (row-storage) or subset of columns (columnar storage)

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Evolution of Processing Models

Dresden Database Research Group

Iterator Model → Materialization Model → Batch Model

TECHNISCHE UNIVERSITÄT DRESDEN

Iterator Model

Dresden Database Research Group

Each query plan operator implements a Next() function.

- On each invocation, the operator returns either a single tuple or a EOF marker if there are no more tuples.
- The operator implements a loop that calls next on is children to retrieve their tuples and then process them.

Each operator implementation

- also has Open() and Close() functions.
- Analogous to constructors/destructors, but for operators.

Also called Volcano or Pipeline Model

Iterator Model - Example

Dresden Database Research Group

Control Flow →
Data Flow →

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

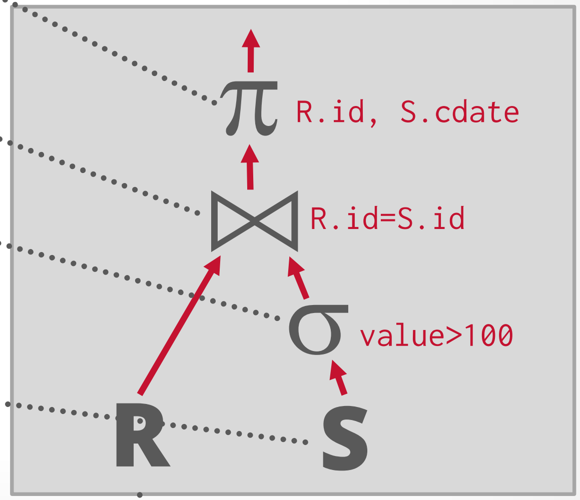
Next()
```
for t in child.Next():
  emit(projection(t))
```

Next()
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

Next()
```
for t in child.Next():
  if evalPred(t): emit(t)
```

Next()
```
for t in R:
  emit(t)
```

Next()
```
for t in S:
  emit(t)
```



Control Flow →
Data Flow →

1

```
for t in child.Next():
  emit(projection(t))
```

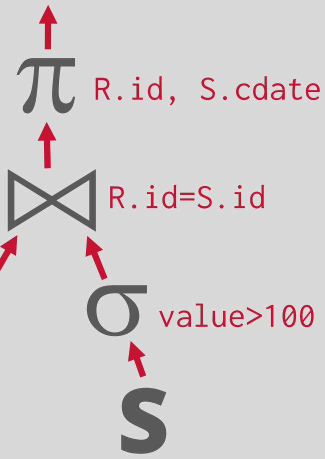
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for t in R:
  emit(t)
```

```
for t in S:
  emit(t)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Control Flow →
Data Flow →

1

```
for t in child.Next():
  emit(projection(t))
```

2

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

Single Tuple

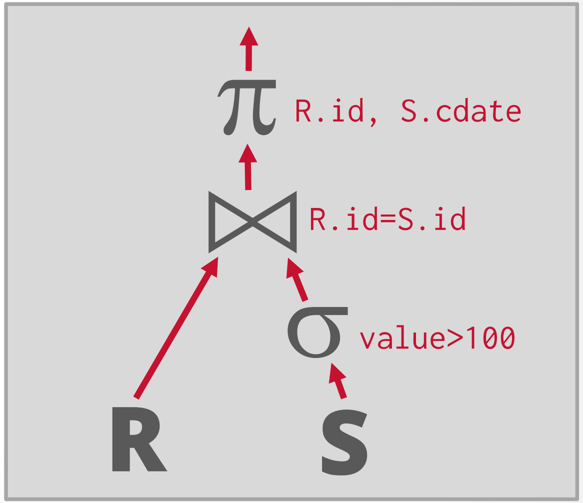
```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

```
for t in R:
  emit(t)
```

```
for t in S:
  emit(t)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Control Flow →
Data Flow →

1
```
for t in child.Next():
  emit(projection(t))
```

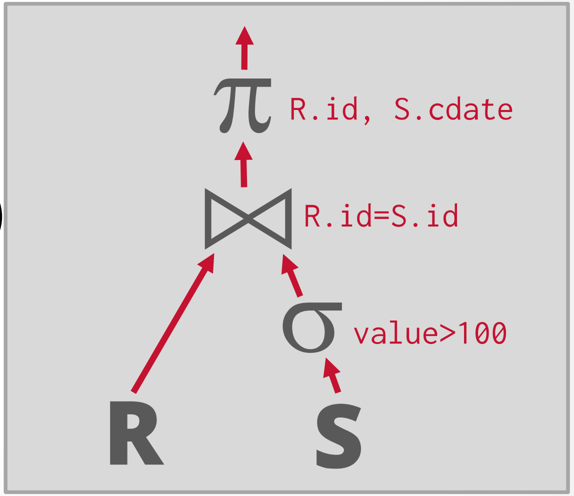
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

3
```
for t in R:
  emit(t)
```

4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

5
```
for t in S:
  emit(t)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Control Flow →
Data Flow →

1
```
for t in child.Next():
  emit(projection(t))
```

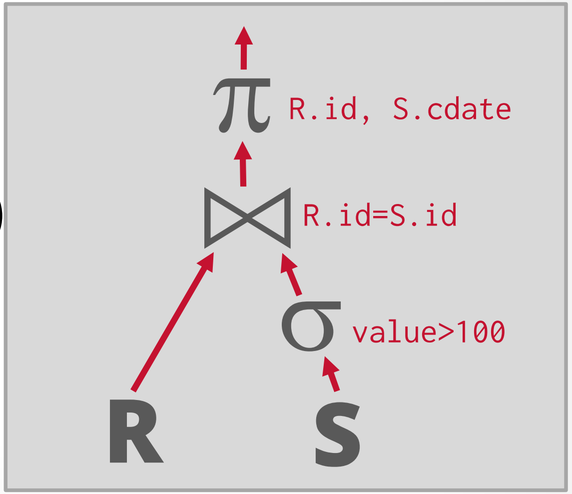
2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

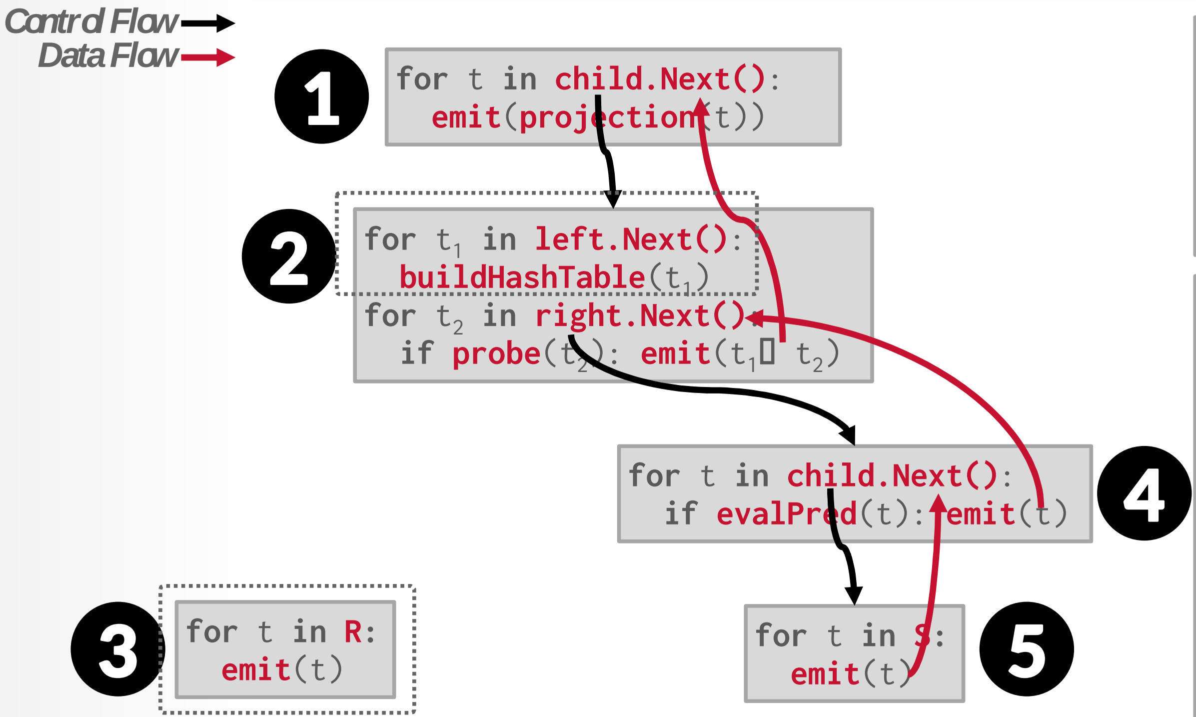
3
```
for t in R:
  emit(t)
```

4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

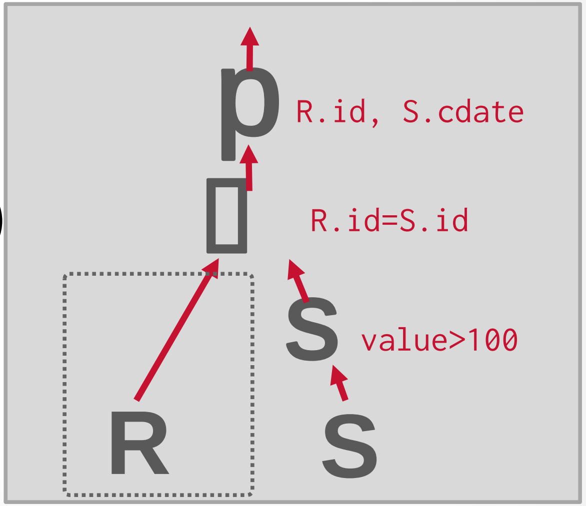
5
```
for t in S:
  emit(t)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```






```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Pipeline #1

Control Flow →
Data Flow →

1
```
for t in child.Next():
  emit(projection(t))
```

2
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈ t2)
```

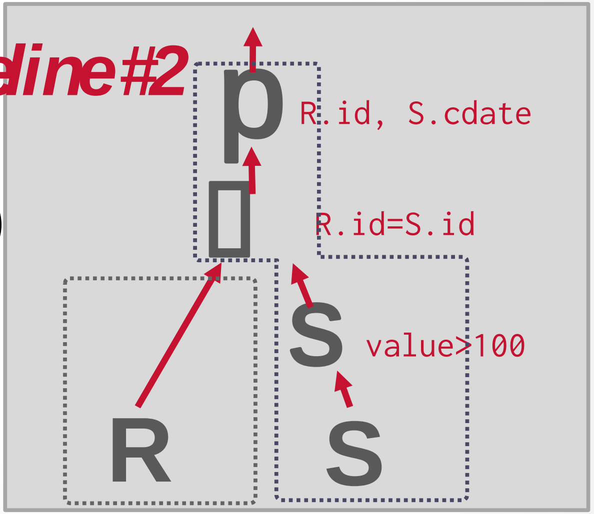
3
```
for t in R:
  emit(t)
```

4
```
for t in child.Next():
  if evalPred(t): emit(t)
```

5
```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Pipeline #2



Iterator Model

The Iterator model is used in almost every DBMS.

- Easy to implement / debug.
- Output control works easily with this approach.

Allows for pipelining

- where the DBMS tries to process each tuple through as many operators as possible before retrieving the next tuple.

A pipeline breaker

- is an operator that cannot finish until all its children emit all their tuples.
- → Joins (Build Side), Subqueries, Order By

Systems

SQLite, mongoDB, Microsoft SQL Server

IBM DB2, PostgreSQL, ORACLE

MySQL

Evolution of Processing Models

Iterator Model → Materialization Model → Batch Model

Materialization Model

Dresden Database Research Group

Description

- Each operator processes its input all at once and then emits its output all at once.
- The operator "materializes" its output as a single result.
- Can send either a materialized row or a single column.

Origin

- Originally developed in MonetDB in the 1990s to process entire columns at a time instead of single tuples.
- MonetDB = 1st pure database system with a columnar storage approach

Example

Control Flow
Data Flow

1

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

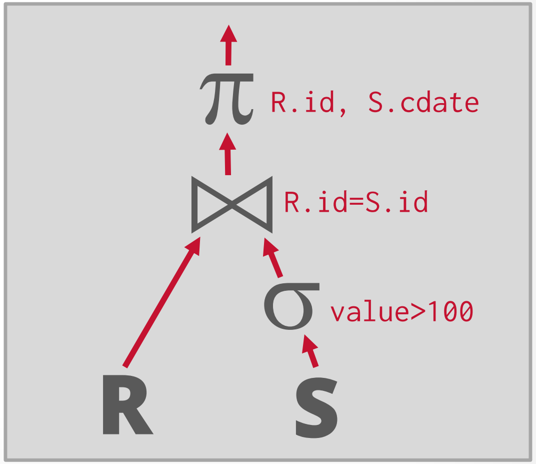
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow
Data Flow

1

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

2

```
out = [ ]
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
return out
```

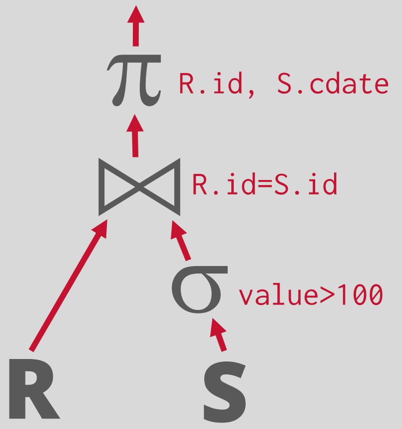
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

3

```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow →
Data Flow →

❶
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

❷
```
out = [ ]
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
return out
```

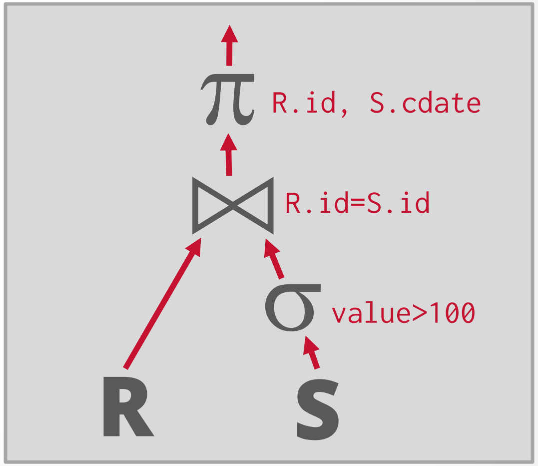
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

All Tuples

❸
```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow
Data Flow

1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

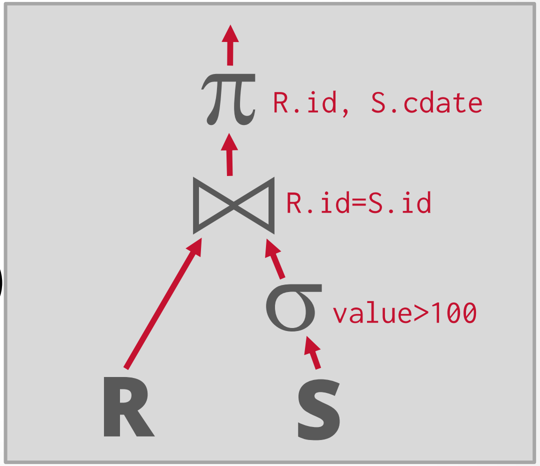
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

4
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

3
```
out = [ ]
for t in R:
  out.add(t)
return out
```

5
```
out = [ ]
for t in S:
  out.add(t)
return out
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

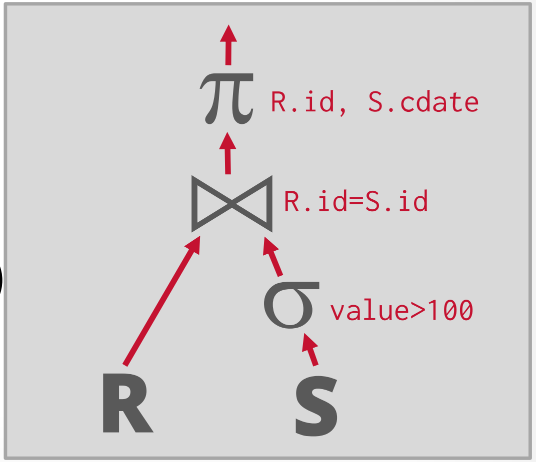
2
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

3
```
out = [ ]
for t in R:
  out.add(t)
return out
```

4
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

5
```
out = [ ]
for t in S:
  out.add(t)
return out
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow →
Data Flow →

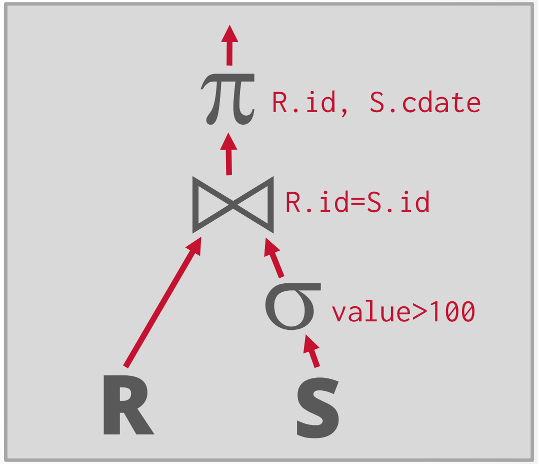
1
```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

2
```
out = [ ]
for t₁ in left.Output():
  buildHashTable(t₁)
for t₂ in right.Output():
  if probe(t₂): out.add(t₁⋈t₂)
return out
```

```
out = [ ]
for t in S:
  if evalPred(t): out.add(t)
return out
```

3
```
out = [ ]
for t in R:
  out.add(t)
return out
```

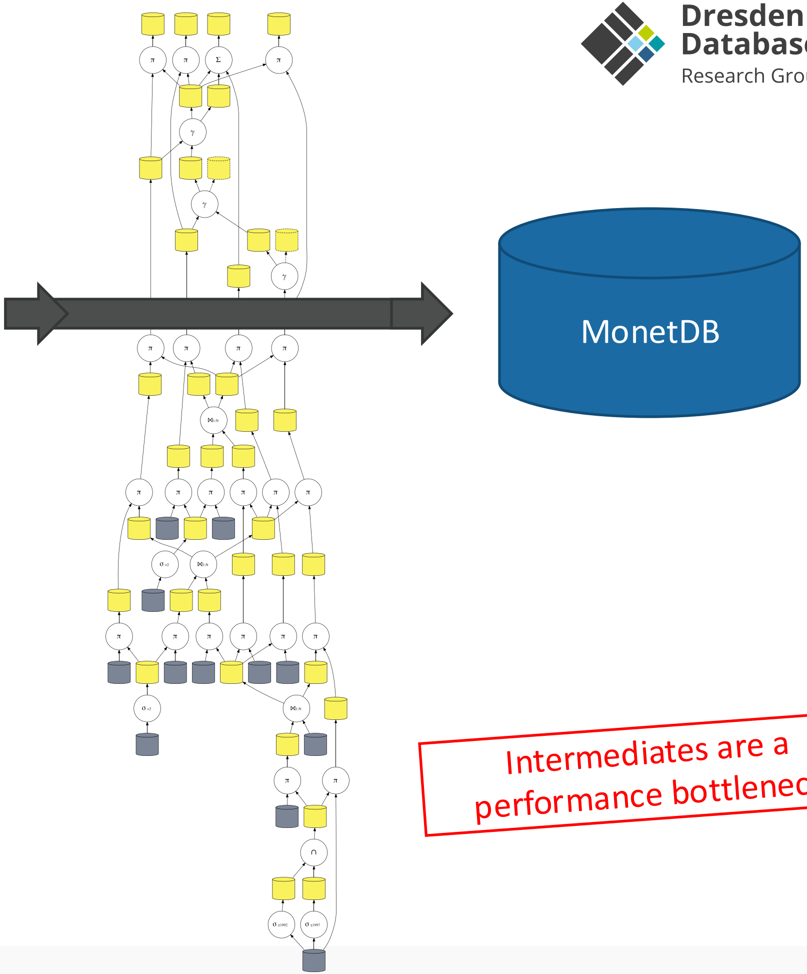
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Materialization Model

Example: Star Schema Benchmark (SSB) Query 3.1

```
SELECT c_nation, s_nation, d_year, SUM(lo_revenue)
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_orderdate = d_datekey
  AND c_region = 'ASIA' AND s_region = 'ASIA'
  AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year ASC, revenue DESC;
```



Number of occurrences

33x operator

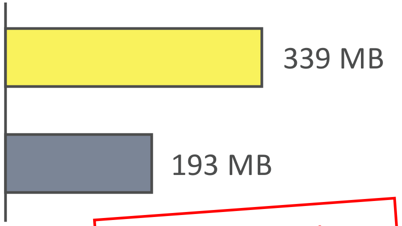
35x intermediate

12x base column

Many Intermediates

Total data size @sf 1

(64-bit per data element)



Large total size

Intermediates are a performance bottleneck

Dresden Database Research Group

Evolution of Processing Models

Dresden Database Research Group

Iterator Model

Materialization Model

Batch Model

TECHNISCHE UNIVERSITÄT DRESDEN

Batch Model

Like the Iterator Model where each operator implements a Next() function, but...

- Each operator emits a batch of tuples/columns instead of a single tuple.
- The operator's internal loop processes multiple tuples/column values at a time.
- The size of the batch can vary based on hardware or query properties.

Iterator Model

- Best-of-both- Worlds: Combination of Iterator Model and Materialization Model

Example

Control Flow →
Data Flow →

1

```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

2

```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

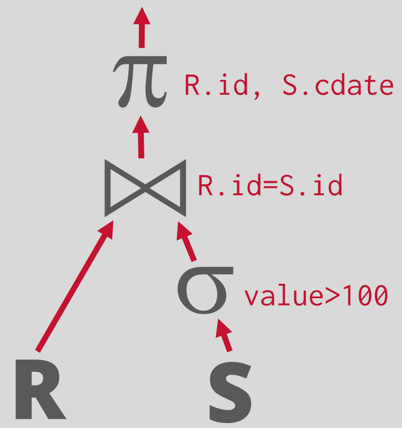
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3

```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

2
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

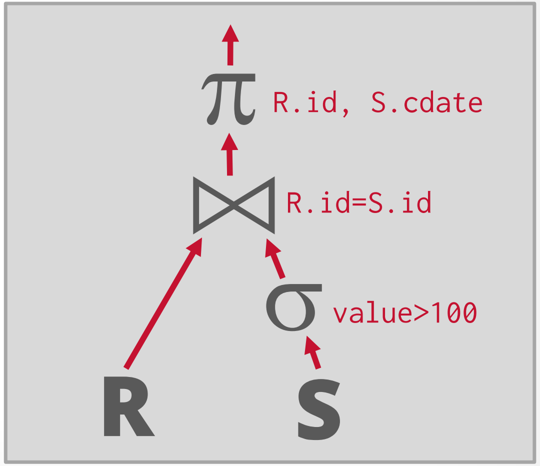
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example

Control Flow →
Data Flow →

1
```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

2
```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

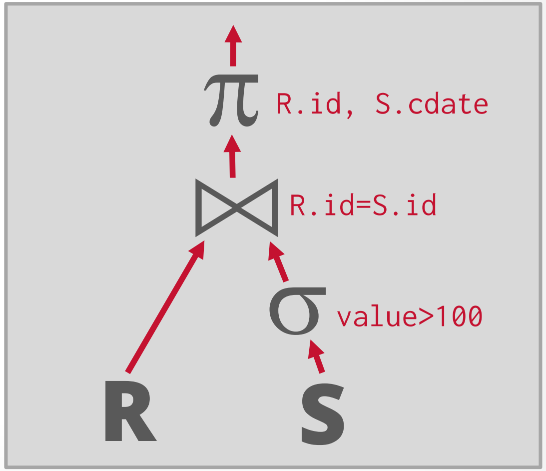
3
```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

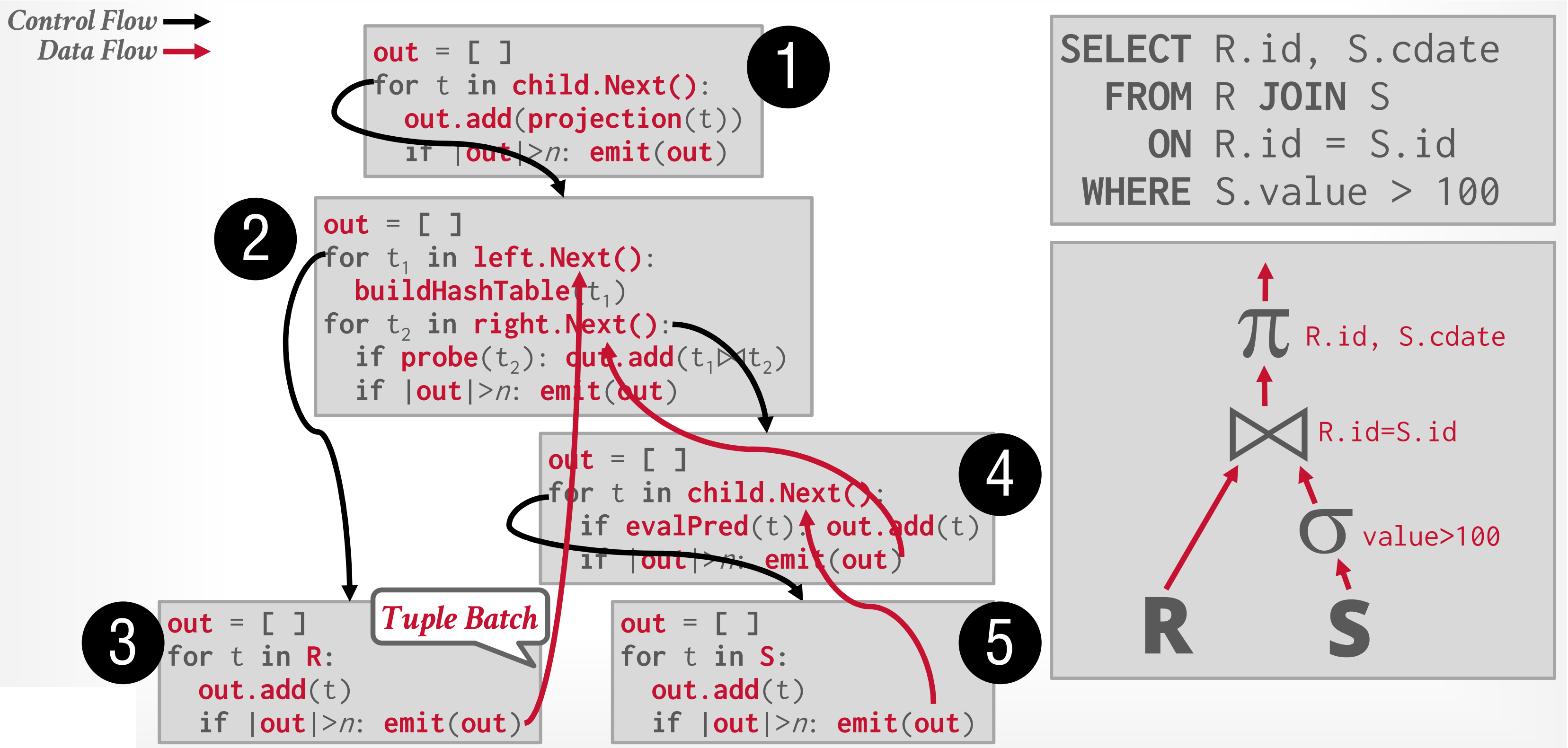
4
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

5
```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Example



Control Flow
Data Flow
out = [ ]
for t in child.Next():
out.add(projection(t))
if |out|>n: emit(out)
1
out = [ ]
for t_1 in left.Next():
buildHashTable(t_1)
for t_2 in right.Next():
if probe(t_2): out.add($t_1 \bowtie t_2$)
if |out|>n: emit(out)
2
out = [ ]
for t in child.Next():
if evalPred(t): out.add(t)
if |out|>n: emit(out)
4
3
out = [ ]
for t in R:
out.add(t)
if |out|>n: emit(out)
Tuple Batch
out = [ ]
for t in S:
out.add(t)
if |out|>n: emit(out)
5
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
π R.id, S.cdate
$\bowtie$ R.id=S.id
σ value>100
R
S

Batch Model

Dresden Database Research Group

Ideal for OLAP queries

- because it greatly reduces the number of invocations per operator.

Allows an out-of-order CPU

- to efficiently execute operators over batches of tuples.
- Operators perform work in tight for-loops over arrays, which compilers know how to optimize / vectorize.
- No data or control dependencies.
- Hot instruction cache.

Systems

presto
Google Big Query
ROCKSET
Yellowbrick
pinot
SingleStore
VERTICA
vectorwise
AlloyDB
ClickHouse
DuckDB
Velox
DATA FUSION

Processing Direction

Observation

- In the previous examples, the DBMS starts executing a query by invoking Next() at the root of the query plan and pulling data up from leaf operators.
- This is the how most DBMSs implement their execution engine.

Approach #1: Top-to-Bottom (Pull)

- Start with the root and "pull" data up from its children.
- Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

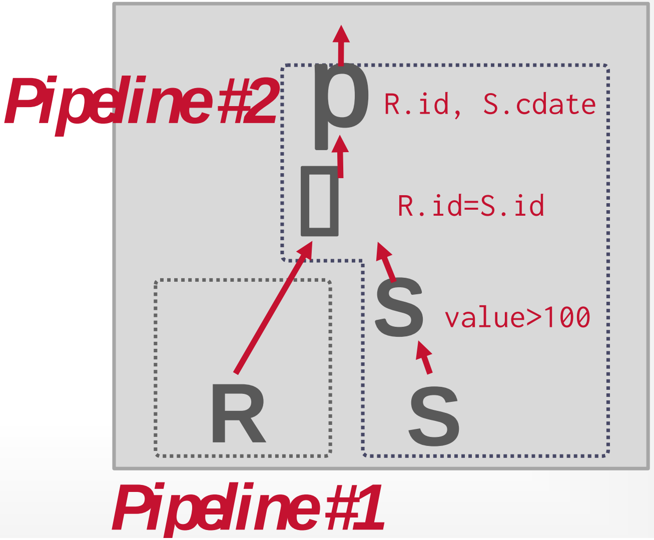
Approach #2: Bottom-to-Top (Push)

- Start with leaf nodes and "push" data to their parents.
- Can "fuse" operators together within a for-loop to minimize intermediate result staging.

Push-based Processing Direction

Control Flow →
Data Flow →

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Push-based Processing Direction

Control Flow →
Data Flow →

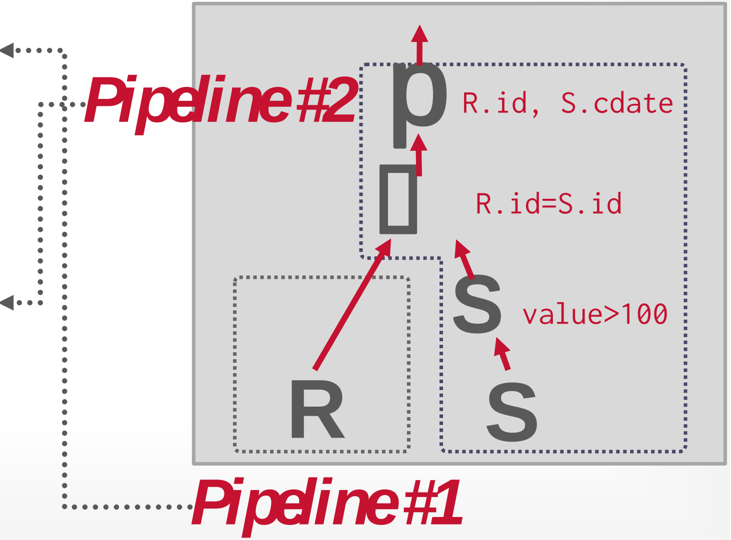
```sql
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

1

```
for t1 in R:
  buildHashTable(t1)
```

2

```
for t2 in S:
  if evalPred(t):
    if probeHashTable(t2):
      emit(projection(t1⋈ t2))
```



Push-based Processing Direction

Dresden Database Research Group

Control Flow →
Data Flow →

Scheduler

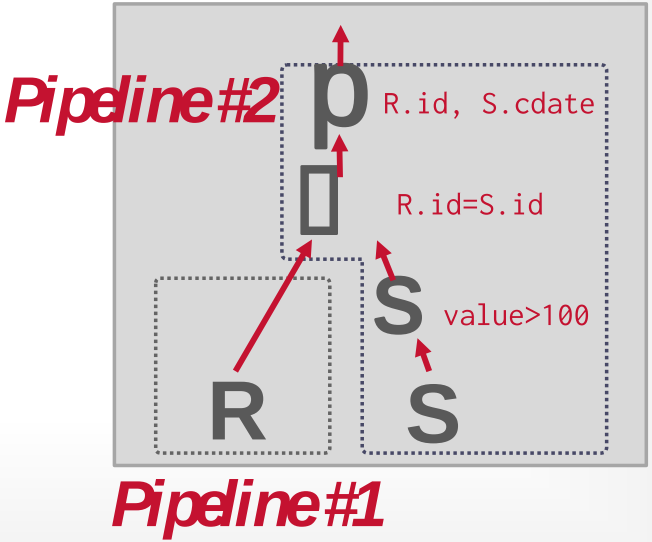
1

```
for t1 in R:
  buildHashTable(t1)
```

2

```
for t2 in S:
  if evalPred(t):
    if probeHashTable(t2):
      emit(projection(t1⨝ t2))
```

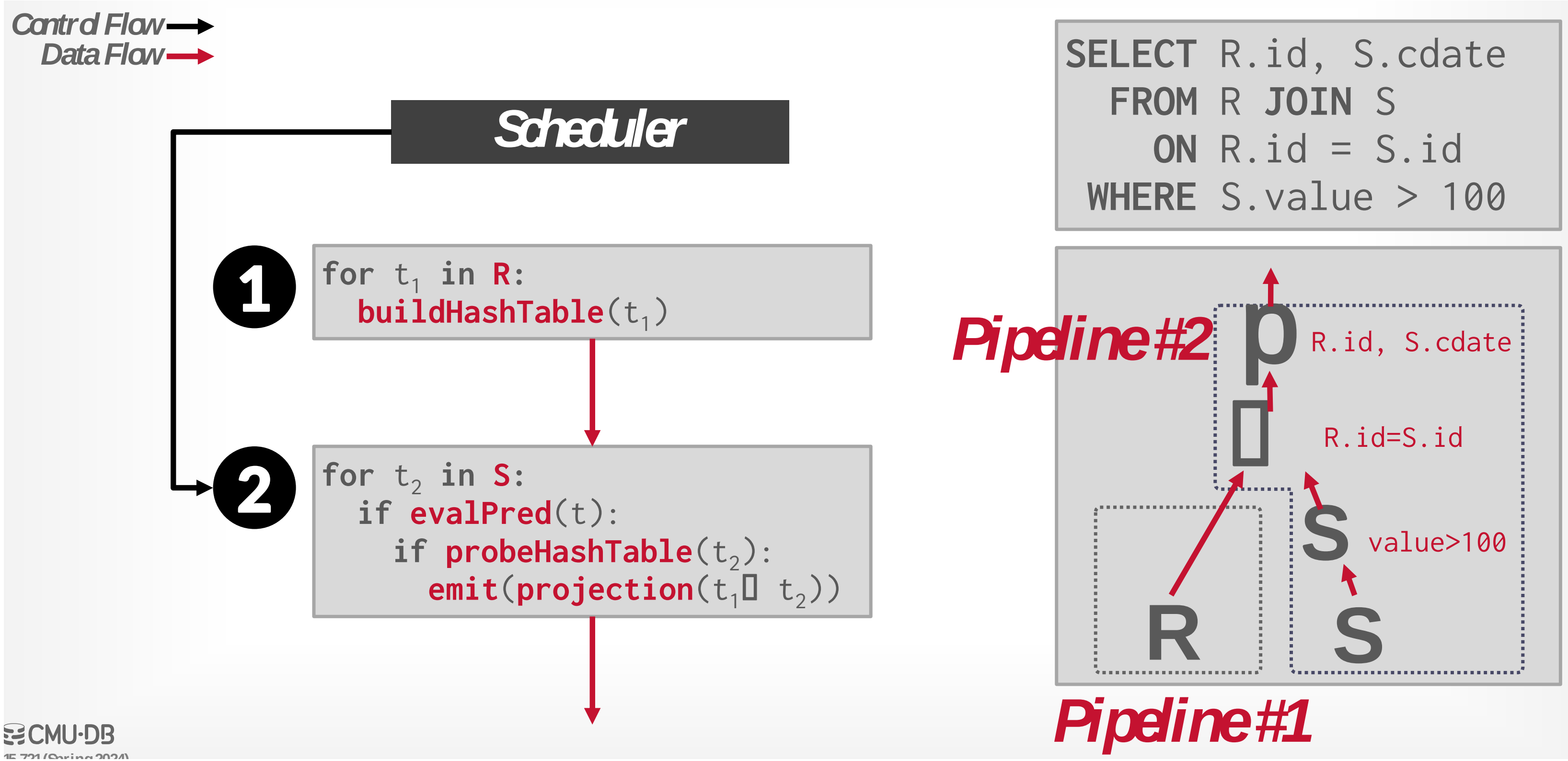
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



CMU-DB
15-721 (Spring 2024)

Push-based Processing Direction

Dresden Database Research Group



Summary

Modern In-Memory Database Systems

- Optimized for analytical queries
- Columnar Data Organization
- Batch-based Processing Model

Homework

- Please read the following paper (available on OPAL: 04-MonetDB.pdf)

DOI:10.1145/1409360.1409380

Breaking the Memory Wall in MonetDB

By Peter A. Boncz, Martin L. Kersten, and Stefan Manegold