Advanced Databases

Spring 2020

Lecture #08:

Query Evaluation

Milos Nikolic

THE UNIVERSITY of EDINBURGH

Query Evaluation

Challenges lurking behind a SQL query

```
SELECT C.cust_id, C.name, SUM(O.total) AS revenue
  FROM Customer AS C JOIN Order AS O
    ON C.cust_id = O.cust_id
 WHERE C.zipcode BETWEEN 8000 AND 8999
 GROUP BY C.cust_id, C.name
 ORDER BY C.name
```

projection

aggregation

join

grouping

selection

sorting

DBMS query processors do not execute a query as a large monolithic block...

... but split the query into a number of specialised routines, the **query operators**

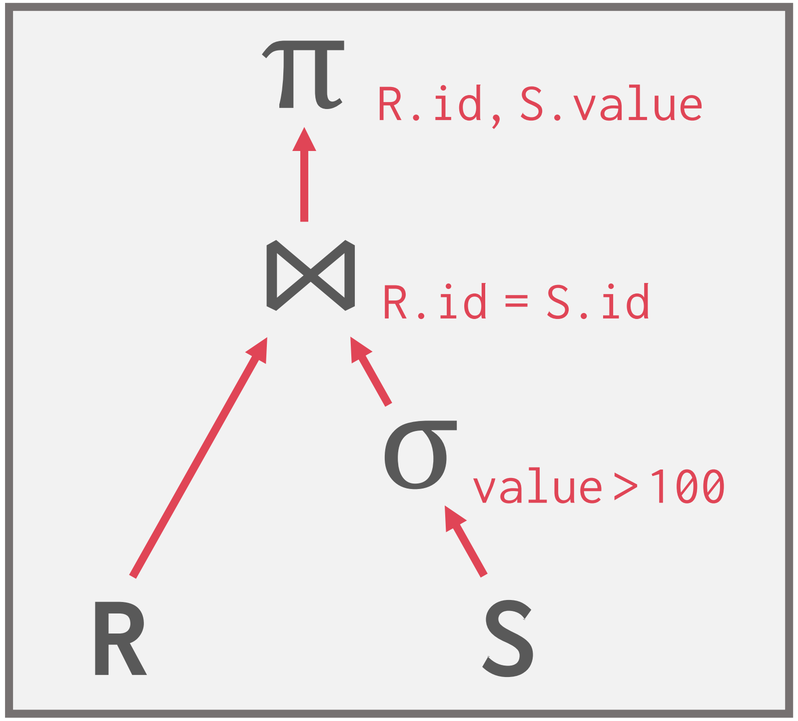
Query Evaluation

The operators from (extended) RA are arranged in a **tree** called **query plan**

- Edges indicate data flow (I/O of operators)
- Data flows from the leaves towards the root

The output of the root is the query result

```
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



RA operators: selection (**σ**), projection (**π**), union (**∪**), intersection (**∩**), difference (**–**), product (**×**), join (**⋈**), renaming (**ρ**), assignment (**R ← S**), duplicate elimination (**δ**), aggregation (**γ**), sorting (**τ**), division (**R / S**)

Query Evaluation Operators

For RA operator ⍟, a typical DBMS query engine may provide

- **different implementations** ⍟′, ⍟″, … all semantically equivalent to ⍟
- with different performance characteristics

Variants (⍟′, ⍟″, …) are called **physical** operators

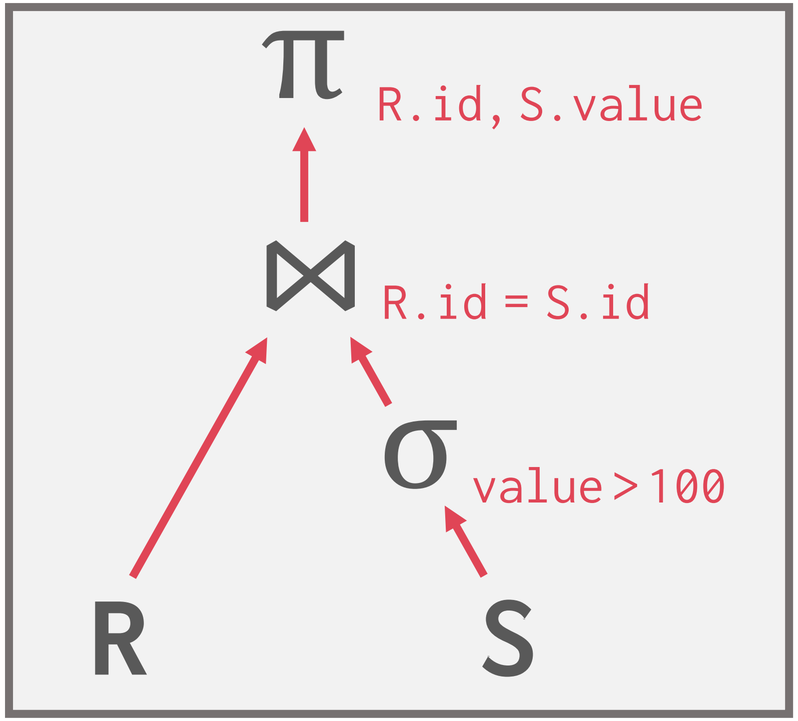
- implement the **logical** operator ⍟ of the relational algebra

Physical operators exploit properties such as:

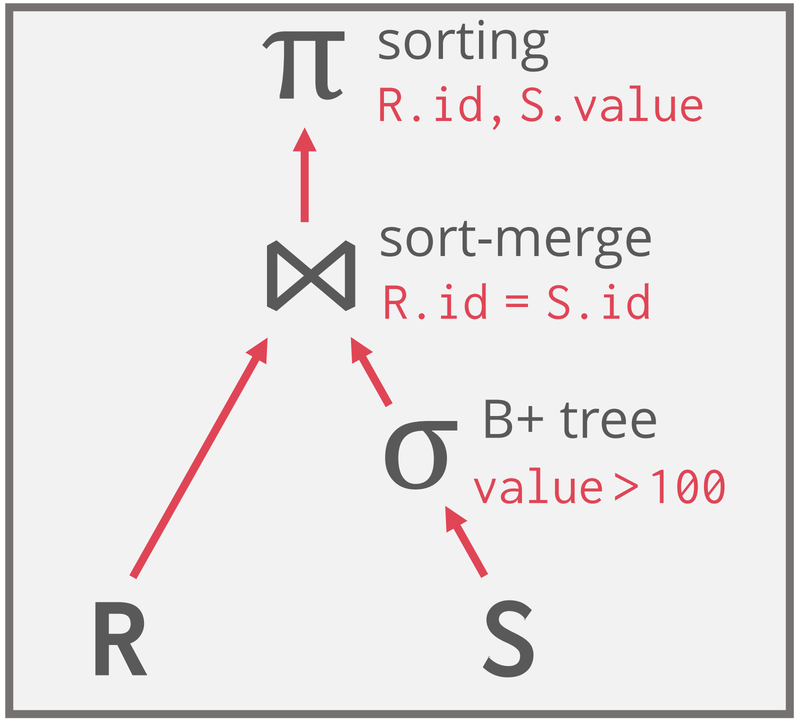
- presence or absence of indexes on the input file(s),
- sortedness and size of the input file(s),
- space in the buffer pool, buffer replacement policy, etc.

Query Evaluation Plans

Logical Plan



Physical Plan



Query optimization = choose "best" physical plan

(among many alternatives)

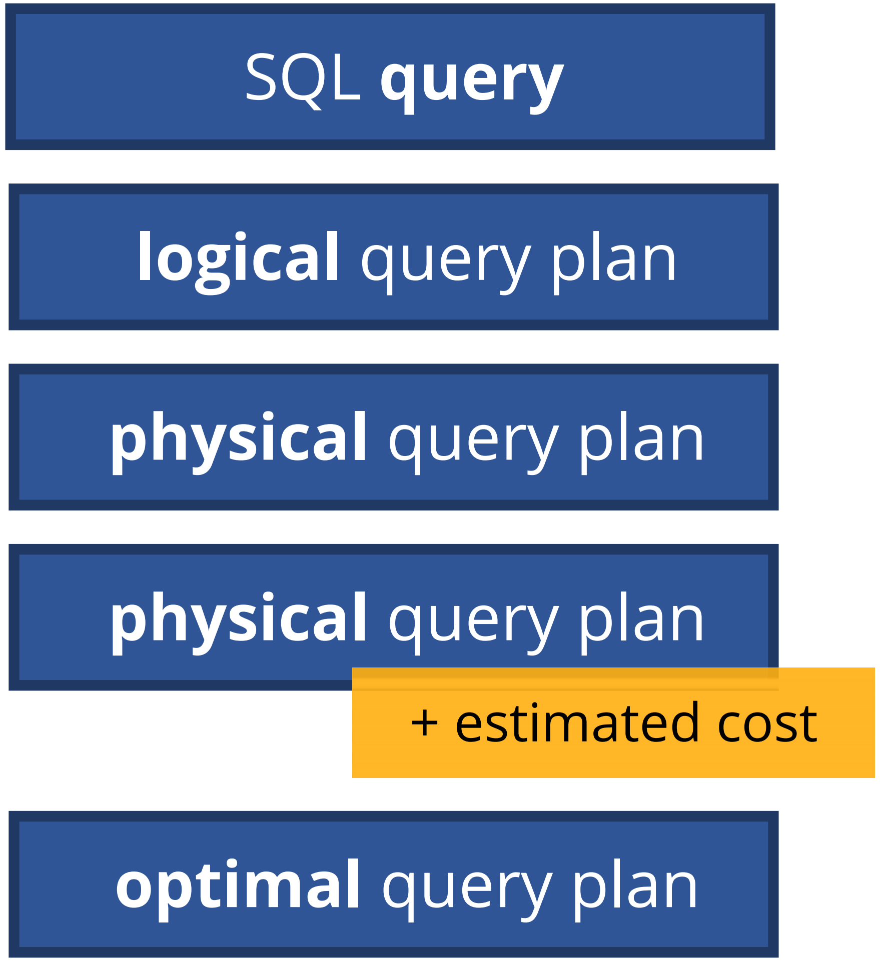
Query Evaluation Workflow

1. Parse given query
2. Translate query to RA
3. Enumerate plans by selecting physical operators and order of operators
4. Determine cost of physical query plans
5. Select the “optimal” query plan

   space of possible plans far too large

   some type of approximation is used

   no guarantee to find optimal query plan



Access Methods

An **access method** (path) is a way the DBMS can access the data stored in a table

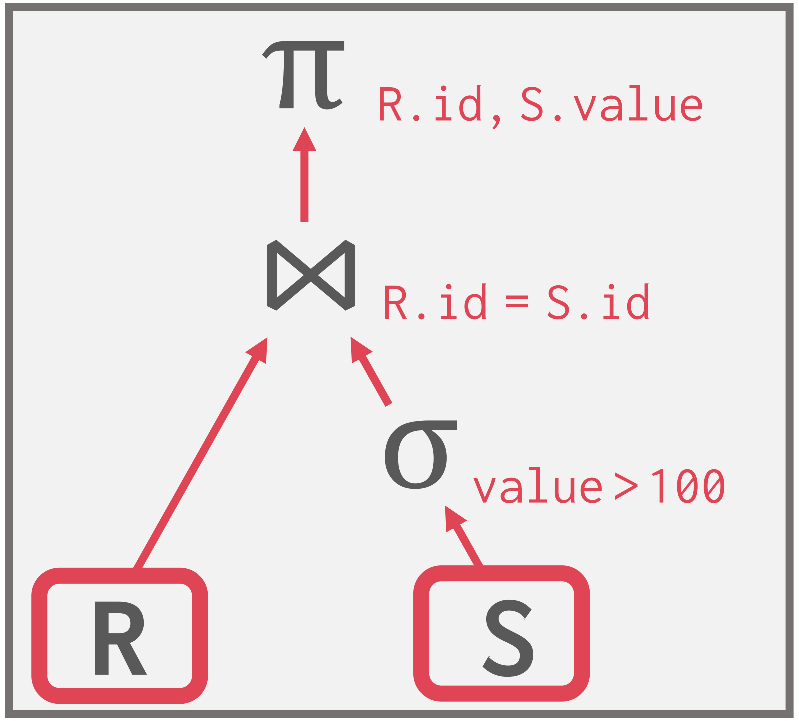
- Not defined in relational algebra
- Includes selection **predicates**

Three basic approaches:

- Sequential scan
- Index scan
- Multi-Index / “Bitmap” scan

Choice depends on #pages needed to read

```
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



Sequential Scan

For each page in the table

- Retrieve it from the buffer pool
- Iterate over each tuple and check if it matches (arbitrary) predicate *p*

```
for page in table.pages:
  for t in page.tuples:
    if evalPred(p,t):
      // do something!
```

The DBMS keeps an internal **cursor** that tracks the last examined page

I/O cost = N pages to read + $sel(p) \cdot N$ pages to write

- *sel(p)* – **selectivity** of predicate *p* is the fraction of tuples satisfying predicate *p*
- The selection operator often processes tuples “on-the-fly” (no writing to disk)

Index Scan

The DBMS picks an index to find the tuples that the query needs

Which index to use depends on:

- What attributes the index contains
- What attributes the query references
- The attributes' value domains
- Predicate composition
- Whether the index has unique or non-unique keys

Index Scan

Suppose that a single table has two indexes

Tree index 1 on **age**

Index 2 on **dept**

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'UK'
```

Scenario #1

There are 99 people under the age of 30 but only 2 people in the CS department

Scenario #2

There are 99 people in the CS department but only 2 people under the age of 30

Clustered B+ Tree Scan

A **clustered B+ tree** index whose search key matches the selection predicate *p* is clearly the superior method

Reminder: Tree index with search key **<*a*, *b*, *c*>** accepts a conjunction of terms that make a **prefix** of search keys

a = *1* and *b* = *3* and *c* < *5* ✓   *a* = *1* and *b* > *5* ✓   *a* = *1* and *c* > *9* **X**

I/O cost =

3-4 to access B+ tree +

sel(p) · *N* to scan (sorted) leaf pages +

sel(p) · *N* to write output pages

Unclustered B+ Tree Scan

Accessing an unclustered B+ tree index can be expensive

- I/O cost ≈ # of matching **leaf index entries**

If ***sel(p)*** indicates a large number of qualifying records, it pays off to

- read the matching index entries <*k*, *rid*>
- sort those entries on their *rid* field
- access the pages in sorted *rid* order

Lack of clustering is a minor issue if ***sel(p)*** is close to 0

Hash Index Scan

A selection predicate *p* matches a hash index only if:

1) *p* contains a term of the form ***A* = *c***, and
2) the hash index has been built over column ***A***

Use index to jump to the bucket of qualifying tuples

- 1-2 I/O cost to retrieve the right index bucket page in practice

sel(p) is likely to be close to 0 for equality predicates

I/O cost = # pages to read in matching bucket (= length of chain) +
sel(p) · *N* pages to write

Multi-Index Scan

- If there are multiple indexes that the DBMS can use for a query:
  - Compute sets of record IDs using each matching index
  - Combine these sets based on the query's predicates (union vs. intersect)
  - Retrieve the records and apply any remaining terms
- Set intersection can be done with bitmaps, hash tables, or Bloom filters

Multi-Index Scan

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'UK'
```

Suppose that a single table has two indexes

- Tree Index 1 on **age**
- Index 2 on **dept**

DBMS may decide to use both indexes

- Retrieve the record ids satisfying **age < 30** using Tree Index 1
- Retrieve the record ids satisfying **dept = 'CS'** using Index 2
- Take their intersection
- Retrieve records and check **country = 'UK'**

Processing Model

Processing model defines how the DBMS executes a query plan

- Different trade-offs for different workloads

Three main approaches:

- Iterator Model
- Vectorised (batch) Model
- Materialisation Model

Iterator Model

Each query plan operator implements three functions:

- **open()** – initialise the operator's internal state
- **next()** – return either the next result tuple or a null marker if there are no more tuples
- **close()** – clean up all allocated resources
- Each operator instance maintains an internal state

Operators implement a loop that calls **next()** on its children to retrieve their tuples and then process them

Also called **Volcano** or **Pipeline** Model

Goetz Graefe. Volcano – An Extensible and Parallel Query Evaluation System. IEEE TKDE 1994

Iterator Model

- Top-down plan processing
  - The whole plan is initially reset by calling **open()** on the root operator
  - The **open()** call is forwarded through the plan by the operators themselves
  - Control returns to the query processor
  - The root is requested to produce its **next()** result record
  - Operators forward the **next()** request as needed. As soon as the next result record is produced, control returns to the query processor again
- Used in almost every DBMS

Iterator Model

Query processor uses the following routine to evaluate a query plan

Function eval(q)

```
q.open()
r = q.next()
while r ≠ NULL do
  /* deliver record r (print, ship to DB client) */
  emit(r)
  r = q.next()
/* resource deallocation now */
q.close()
```

Output control (e.g., LIMIT) works easily with this model

Iterator Model

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

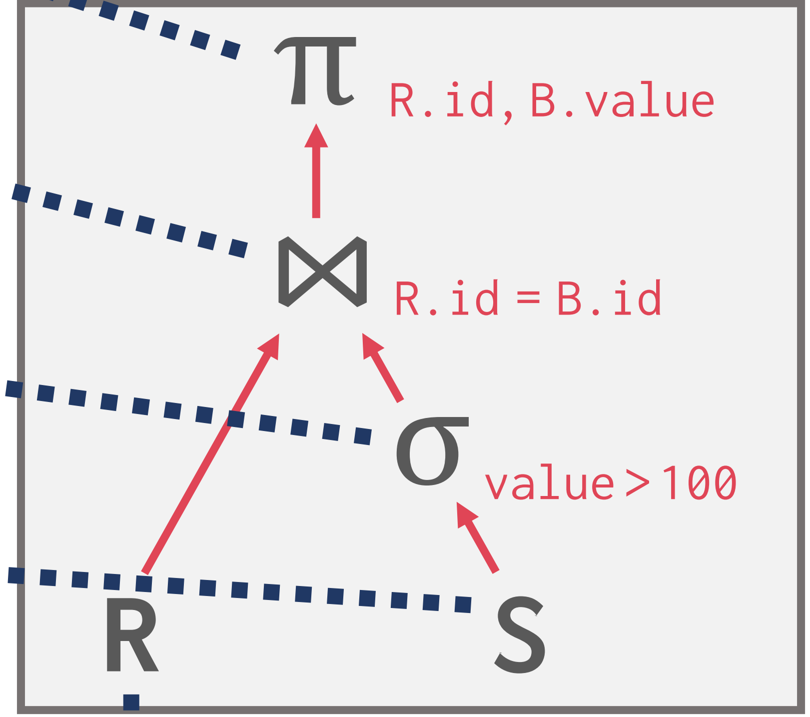
```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

```
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.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



ITERATOR MODEL

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

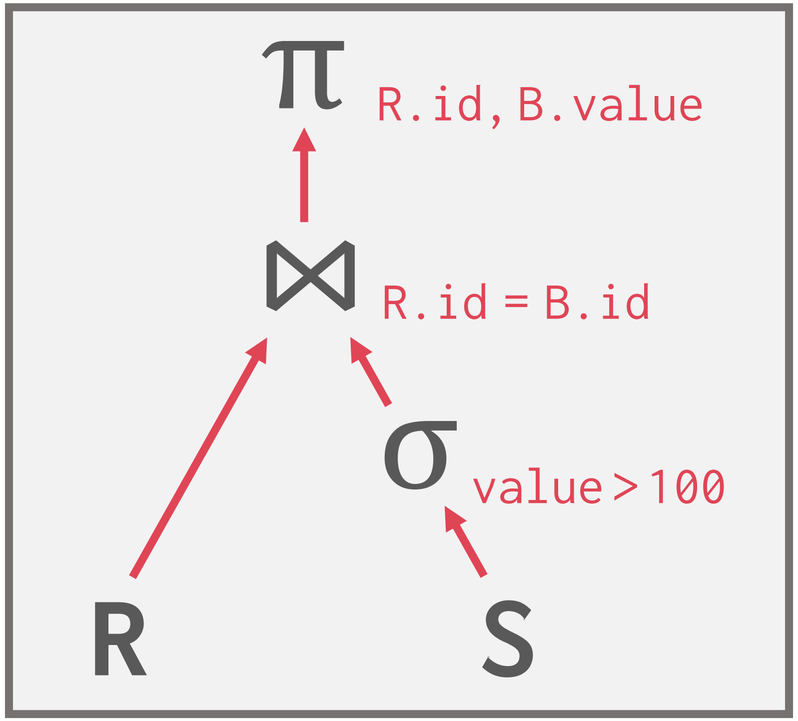
2
```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

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.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



Iterator Model

- Allows for tuple **pipelining**
  - The DBMS process a tuple through as many operators as possible before having to retrieve the next tuple
  - Reduces memory requirements and response time since each chunk of input is propagated to the output immediately
- Some operators will **block** until children emit all of their tuples
  - E.g., sorting, hash join, grouping and duplicate elimination over unsorted input, subqueries
  - The data is typically buffered (“materialised”) on disk

Iterator Model

+ **Nice** & **simple** interface

+ Allows for **easy** combination of operators

– Next called for **every single** tuple & operator

– **Virtual** call via function pointer

  Degrades branch prediction of modern CPUs

– **Poor** code locality and **complex** bookkeeping

  Each operator keeps state to know where to resume

Vectorisation Model

Like Iterator Model, each operator implements a **next()** function

Each operator emits a **batch of tuples** instead of a single tuple

- The operator's internal loop processes multiple tuples at a time
- The size of the batch can vary based on hardware and query properties

Ideal for OLAP queries

- Greatly reduces the number of invocations per operator
- Operators can use vectorised (SIMD) instructions to process batches of tuples

Vectorisation Model

1

```
out = { }
for t in child.output():
  out.add(projection(t))
  if |out| > n: emit(out)
```

2

```
out = { }
for t1 in left.output():
  buildHashTable(t1)
for t2 in right.output():
  if probe(t2): out.add(t1⋈ t2)
  if |out| > n: emit(out)
```

4

```
out = { }
for t in child.output():
  if evalPred(t): out.add(t)
  if |out| > n: emit(out)
```

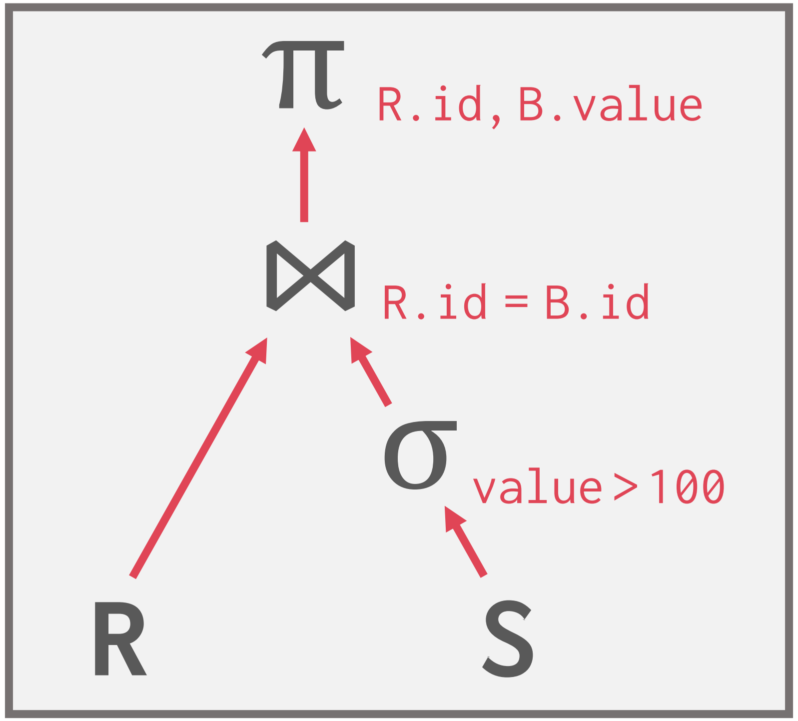
3

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

5

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

```
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



Materialisation Model

- Each operator processes its input all at once and then emits its output
  - The operator "materialises" its output as a single result
- Bottom-up plan processing
  - Data not pulled by operators but **pushed** towards them
  - Leads to better code and data locality
- Better for OLTP workloads
  - OLTP queries typically only access a small number of tuples at a time
  - Not good for OLAP queries with large intermediate results

Materialisation Model

5
```
out = { }
for t in child.output():
  out.add(projection(t))
```

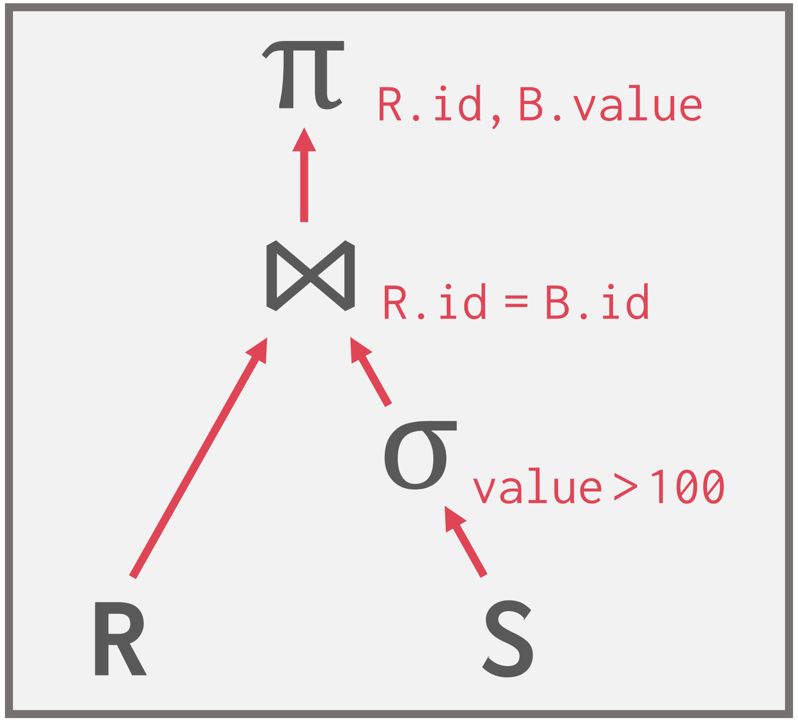
4
```
out = { }
for t1 in left.output():
  buildHashTable(t1)
for t2 in right.output():
  if probe(t2): out.add(t1⋈ t2)
```

3
```
out = { }
for t in child.output():
  if evalPred(t): out.add(t)
```

1
```
out = { }
for t in R:
  out.add(t)
```

2
```
out = { }
for t in S:
  out.add(t)
```

```sql
SELECT R.id, S.value
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



Processing Models: Summary

Iterator / Volcano

Direction: Top-Down

Emits: Single Tuple

Target: General Purpose

Vectorised

Direction: Top-Down

Emits: Tuple Batch

Target: OLAP

Materialisation

Direction: Bottom-Up

Emits: Entire Tuple Set

Target: OLTP