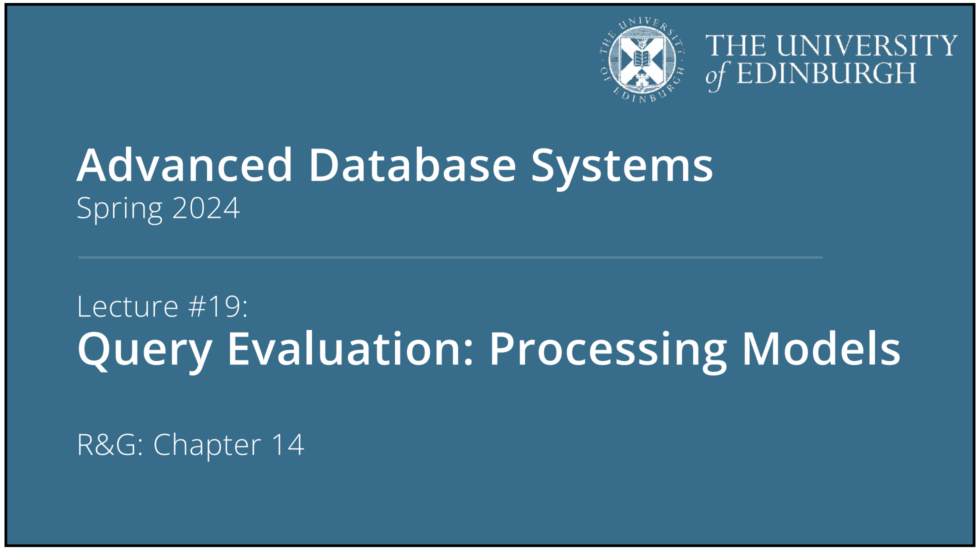


THE UNIVERSITY *of* EDINBURGH

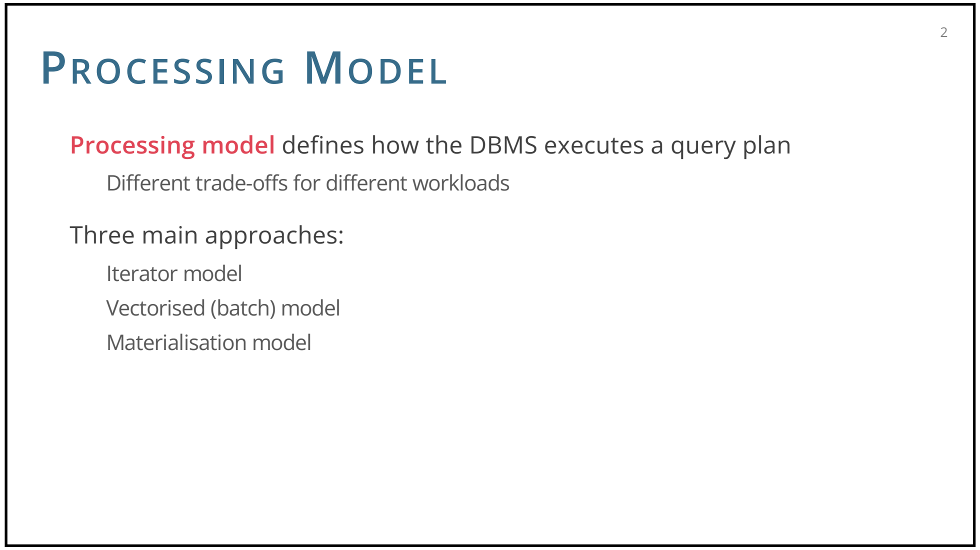
Advanced Database Systems

Spring 2024

Lecture #19:

Query Evaluation: Processing Models

R&G: Chapter 14



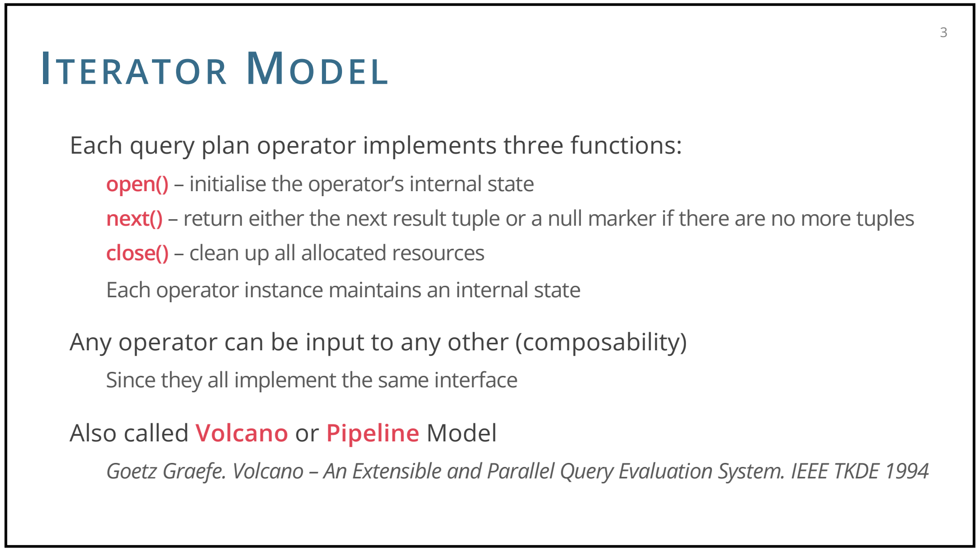
PROCESSING MODEL

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:

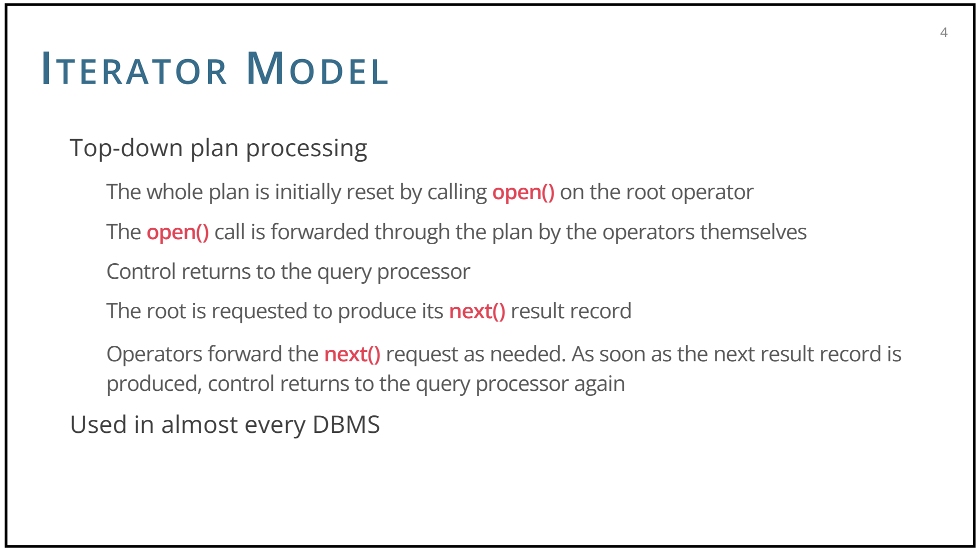
- **open()** – initialise the operator's internal state
- **next()** – return either the next result tuple or a null marker if there are no more tuples
- **close()** – clean up all allocated resources
- Each operator instance maintains an internal state

Any operator can be input to any other (composability)

- Since they all implement the same interface

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

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

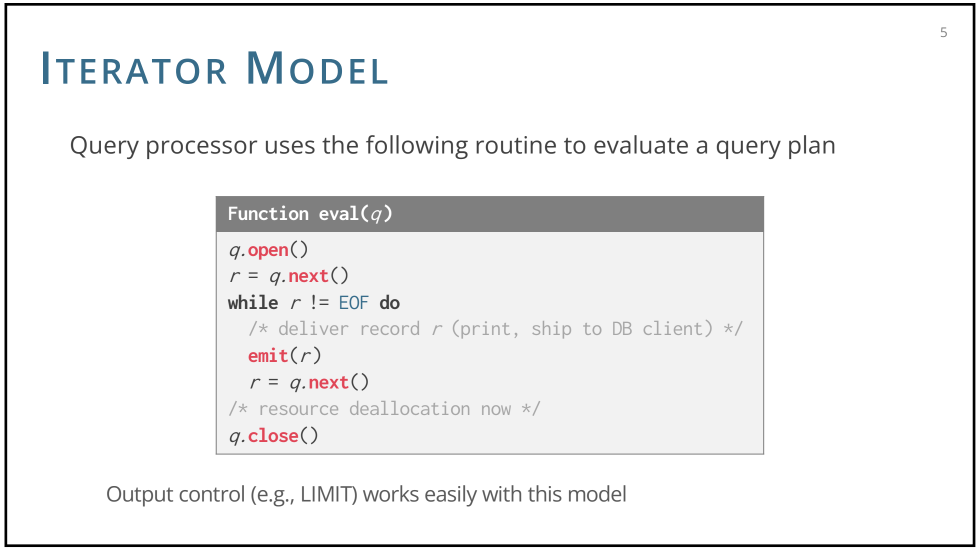


ITERATOR MODEL

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

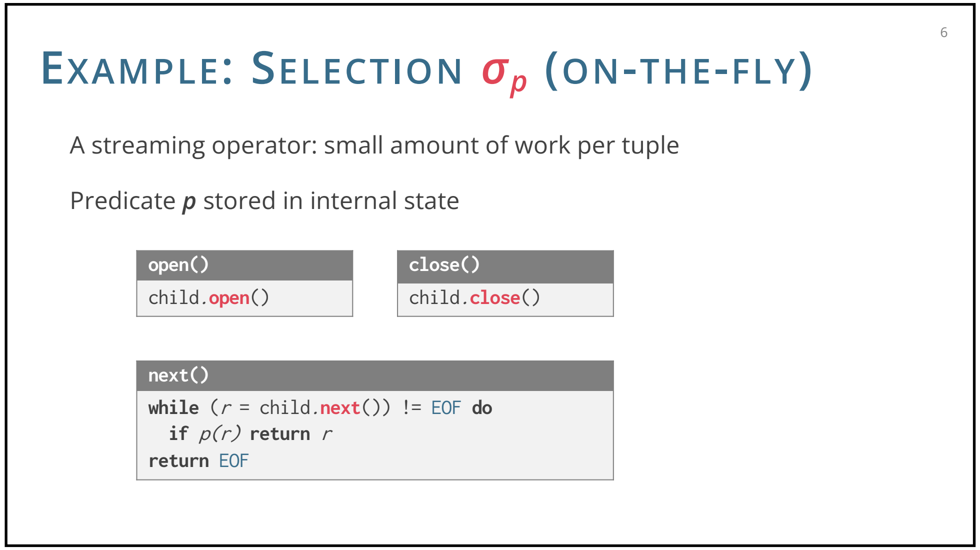


Iterator Model

Query processor uses the following routine to evaluate a query plan

```
Function eval(q)
q.open()
r = q.next()
while r != EOF 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



Example: Selection σ_p (on-the-fly)

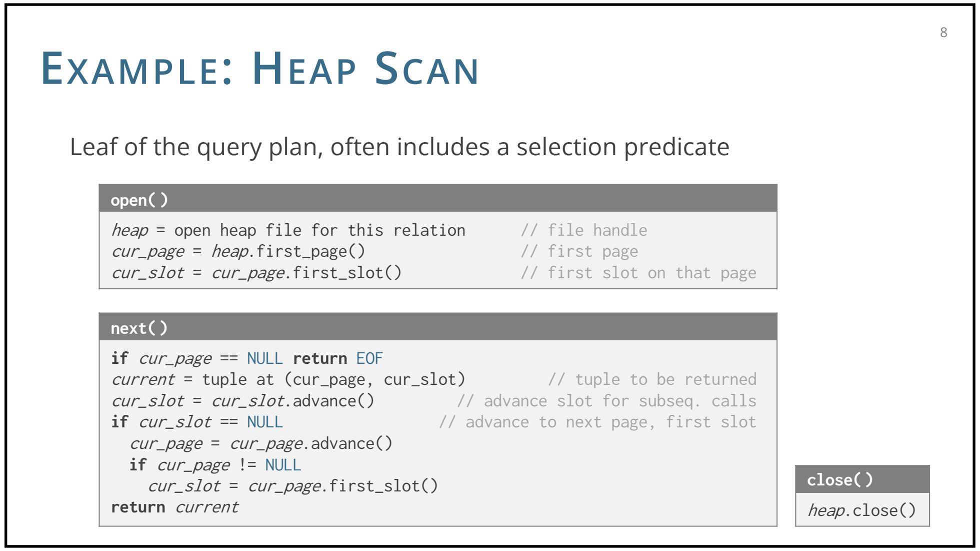
A streaming operator: small amount of work per tuple

Predicate ***p*** stored in internal state

```
open()
child.open()
```

```
close()
child.close()
```

```
next()
while (r = child.next()) != EOF do
  if p(r) return r
return EOF
```



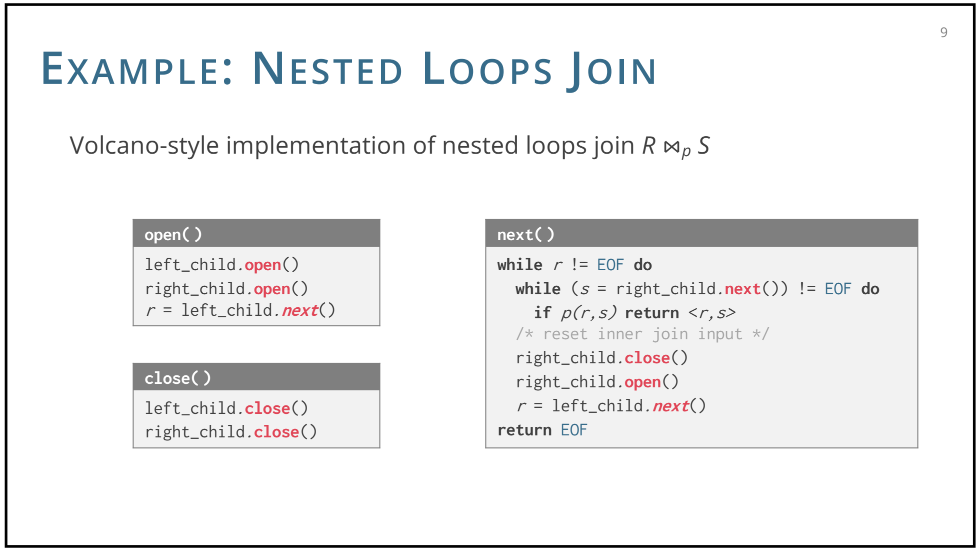
Example: Heap Scan

Leaf of the query plan, often includes a selection predicate

```
open()
heap = open heap file for this relation       // file handle
cur_page = heap.first_page()                  // first page
cur_slot = cur_page.first_slot()              // first slot on that page
```

```
next()
if cur_page == NULL return EOF
current = tuple at (cur_page, cur_slot)          // tuple to be returned
cur_slot = cur_slot.advance()           // advance slot for subseq. calls
if cur_slot == NULL                    // advance to next page, first slot
  cur_page = cur_page.advance()
  if cur_page != NULL
    cur_slot = cur_page.first_slot()
return current
```

```
close()
heap.close()
```



Example: Nested Loops Join

Volcano-style implementation of nested loops join $R \bowtie_p S$

```
open()
left_child.open()
right_child.open()
r = left_child.next()
```

```
close()
left_child.close()
right_child.close()
```

```
next()
while r != EOF do
  while (s = right_child.next()) != EOF do
    if p(r,s) return <r,s>
  /* reset inner join input */
  right_child.close()
  right_child.open()
  r = left_child.next()
return EOF
```

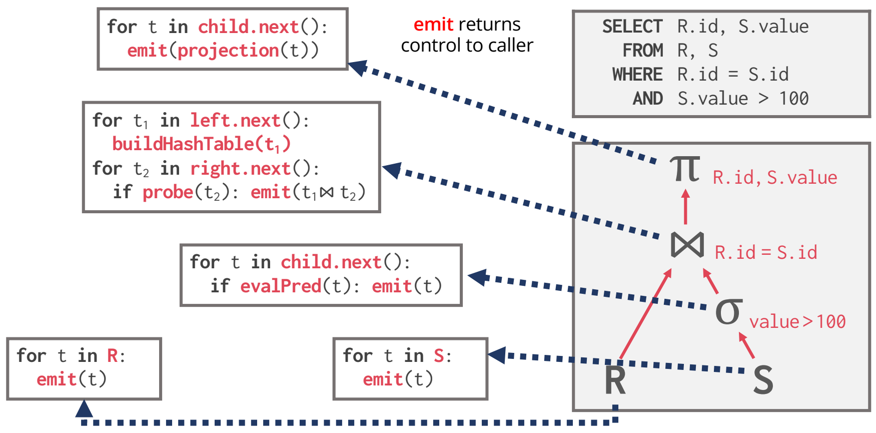
Example: Sort (2-pass)

```
open()
// first, all of pass 0, a blocking call
child.open()
repeatedly call child.next() and generate the sorted runs on disk, until child gives EOF
// second, set up for pass 1, assumes enough buffers to merge
open each sorted run file and load one page per run into input buffer for pass 1
```

```
next()                 // pass 1 merge (assumes enough buffers to merge)
output = min tuple across all buffers
if min tuple was last one in its buffer
  fetch next page from that run into buffer
return output          // (or EOF if no tuples remain)
```

```
close()
deallocate the runs files
child.close()
```

Iterator Model



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

emit returns control to caller

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

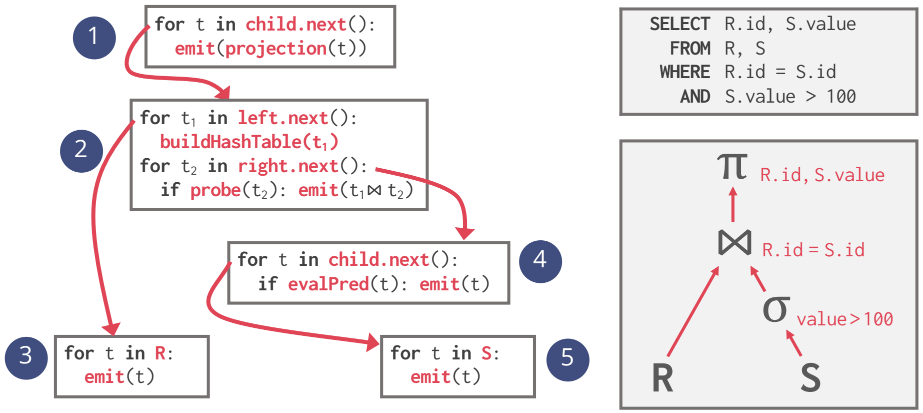
```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

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

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

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

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)
```

```
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

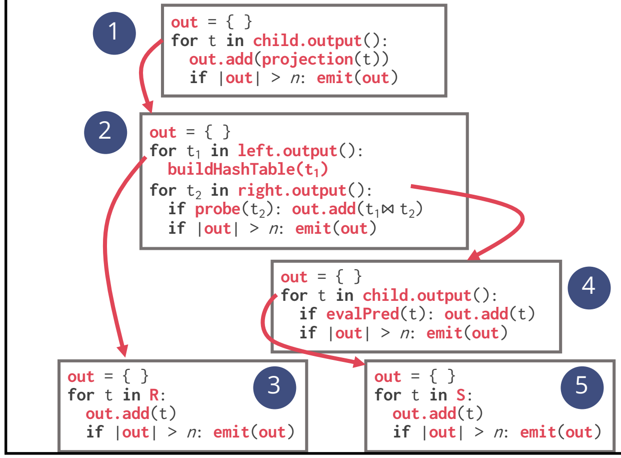
Each operator emits a batch of tuples instead of a single tuple

- 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)
```

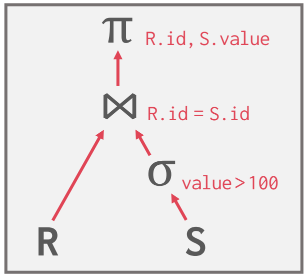
2
```
out = { }
for t₁ in left.output():
  buildHashTable(t₁)
for t₂ in right.output():
  if probe(t₂): out.add(t₁⋈ t₂)
  if |out| > n: emit(out)
```

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

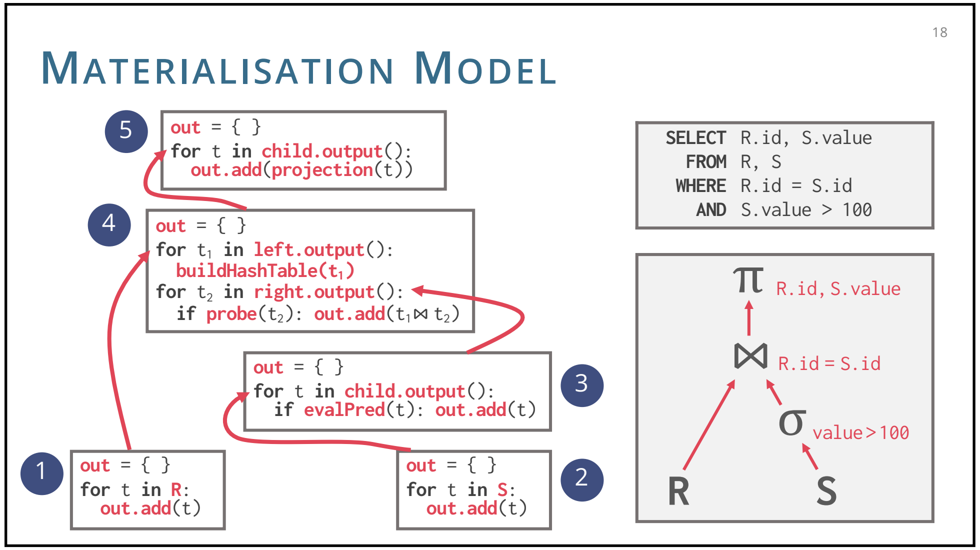
- The operator "materialises" its output as a single result

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))
```

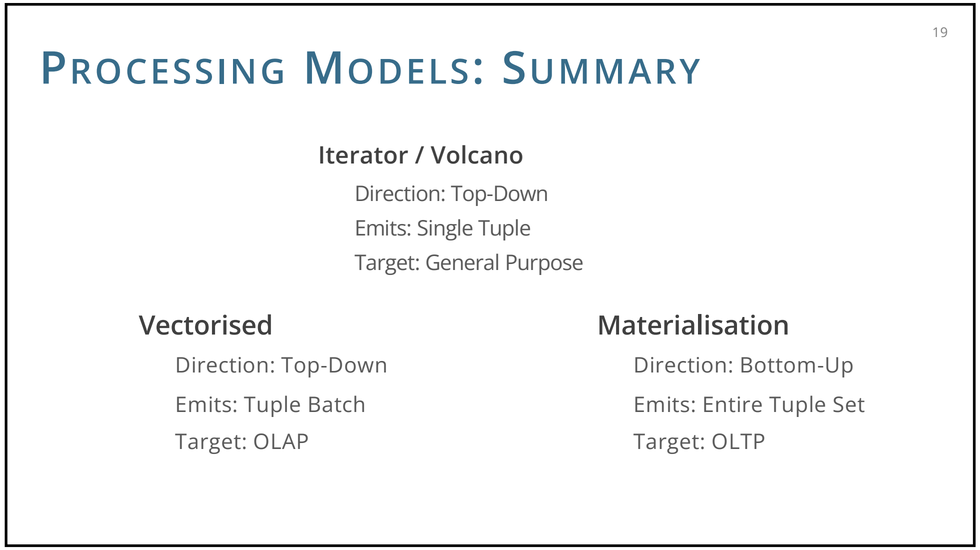
4
```
out = { }
for t1 in left.output():
  buildHashTable(t1)
for t2 in right.output():
  if probe(t2): out.add(t1⋈t2)
```

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)
```

```
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