CSE544
Data Management

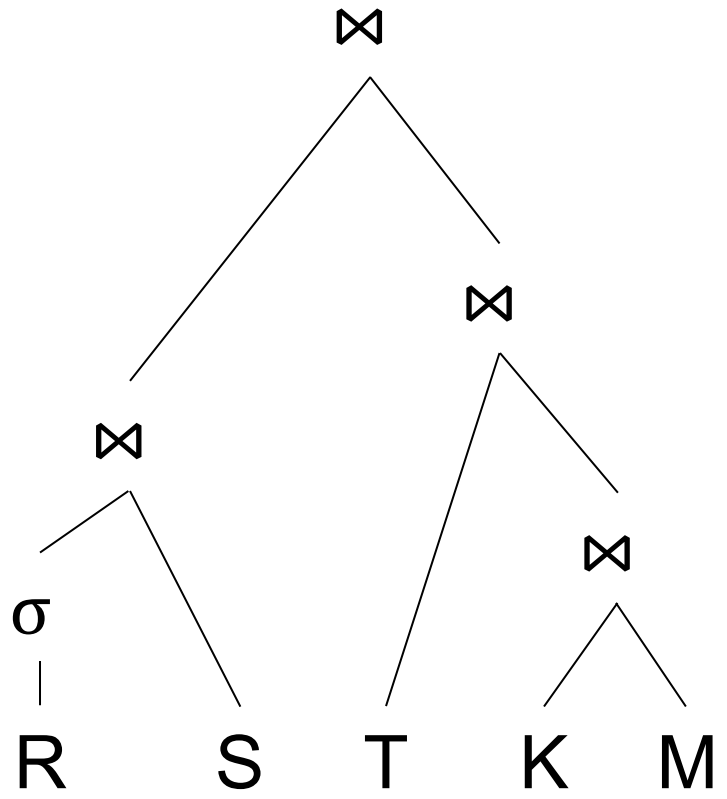
Lecture13
Query Optimization – Part 2

Announcements

- Project proposals feedback posted
- Start working on the project!
- HW3 due next Friday, 11/14

Iterator Model
or the Pull Model

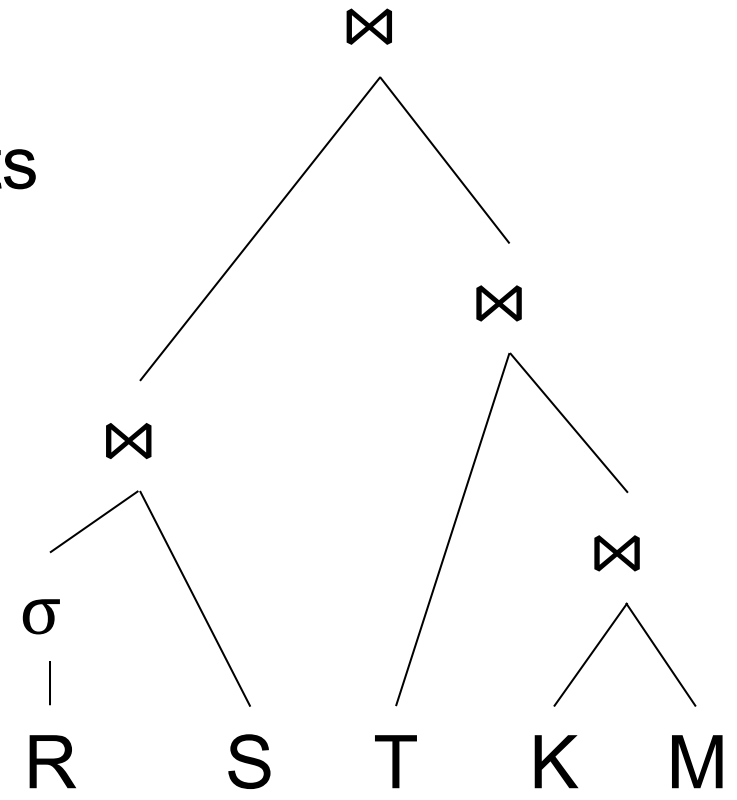
How Do We Combine Them?



How Do We Combine Them?

Option 1:
materialize intermediate results

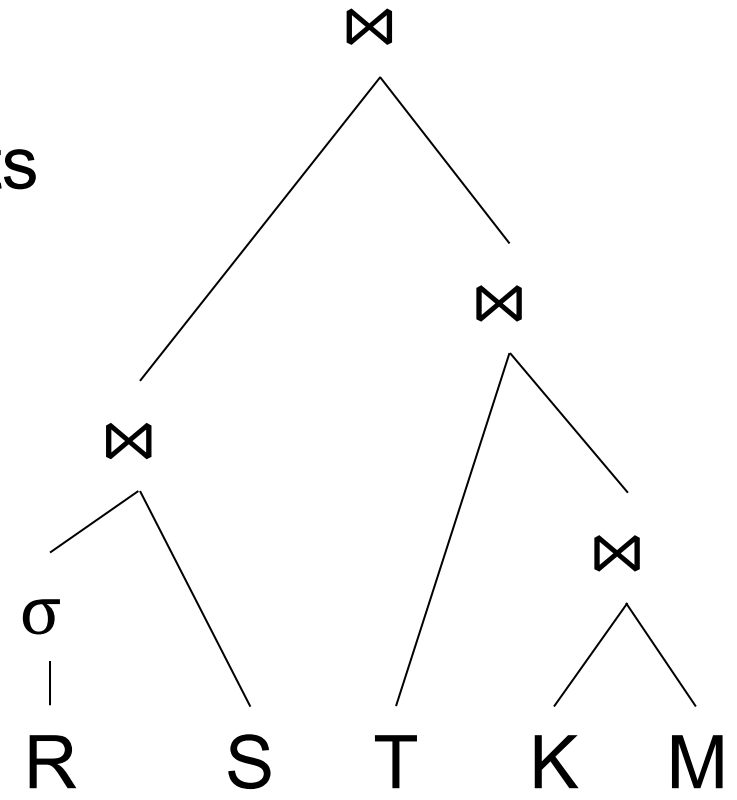
Option 2:
Pipeline tuples btw. ops



How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops



Implementation:
Operator Interface

Operator Interface

Volcano model:

- open(), next(), close()
- Pull model
- Volcano optimizer: G. Graefe's (Wisconsin) → SQL Server
- Supported by most DBMS today

Operator Interface

Volcano model:

- open(), next(), close()
- Pull model
- Volcano optimizer: G. Graefe's (Wisconsin) → SQL Server
- Supported by most DBMS today

Data-driven model:

- open(),produce(), consume(),close()
- Push model
- Introduced by Thomas Neumann in Hyper (at TU Munich), later acquired by Tableau
- "How to architect…"

Volcano Model

Open()

- Calls open() on the children
- Creates any local data structures

Next()

- May call next() repeatedly on children
- Returns exactly 1 tuple, or EOF

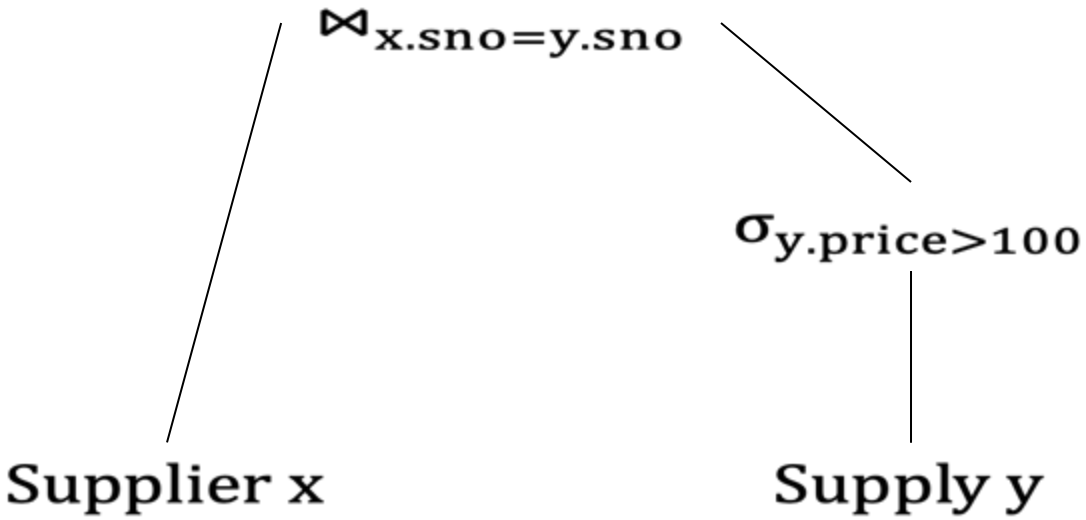
Close()

- Free any local memory

Volcano Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



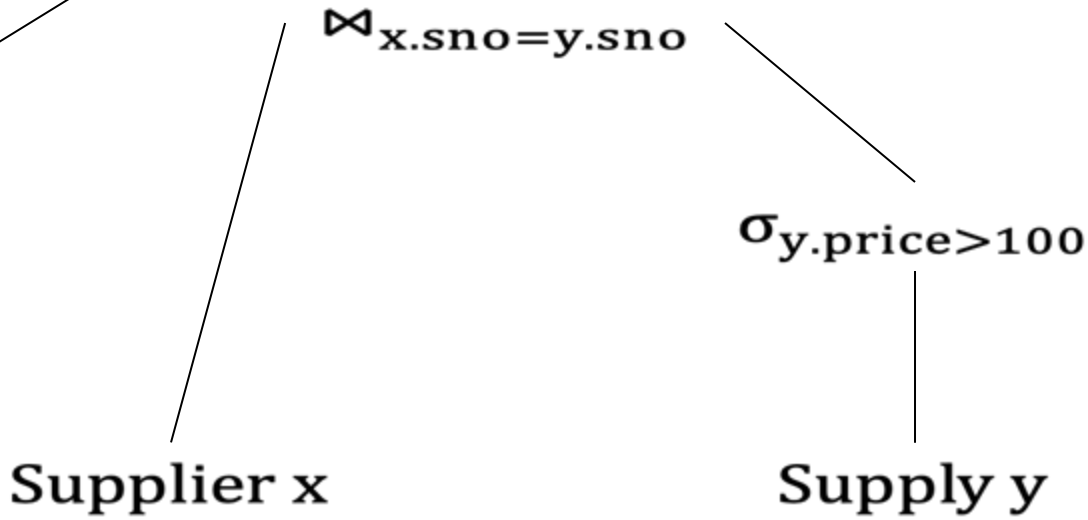
Volcano Example

Should be

$\sigma_{y.price>100}(Supply)$

”Normal” hash-join

```
for x in Supplier do
    insert(x.sno, x)

for y in Supply do
    x = find(y.sno);
    output(x,y);
```

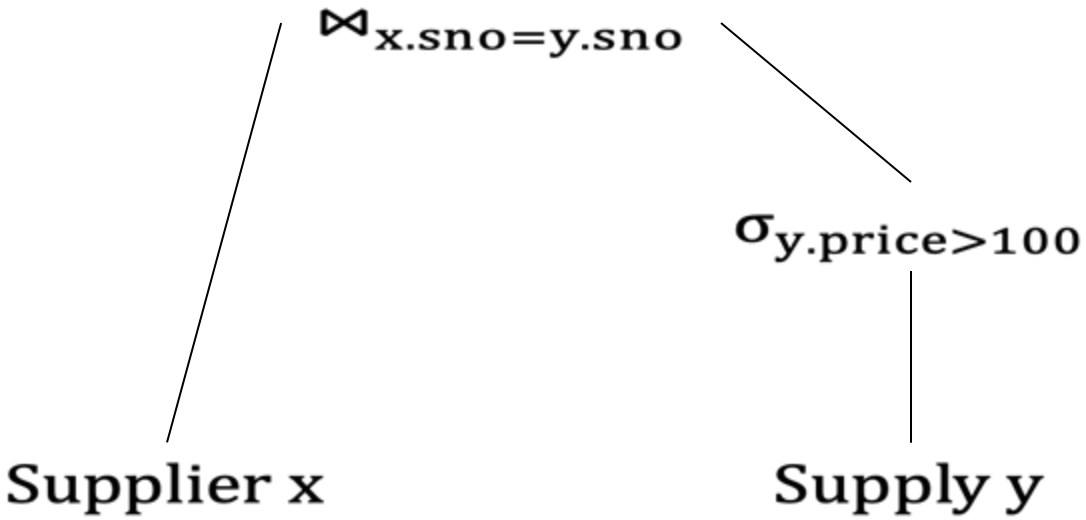


Volcano Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```

Pipelining changes
the code significantly

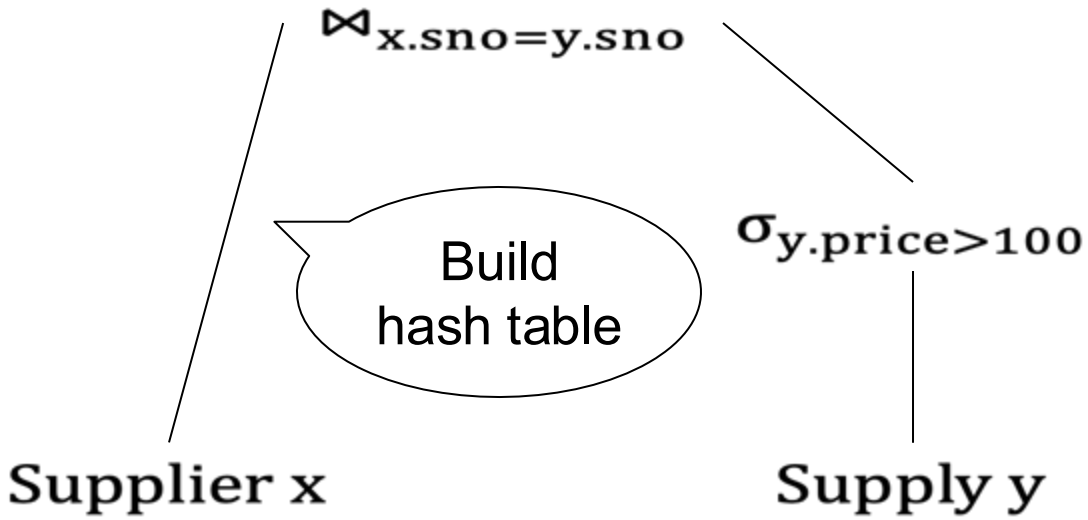


Volcano Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```

Pipelining changes
the code significantly

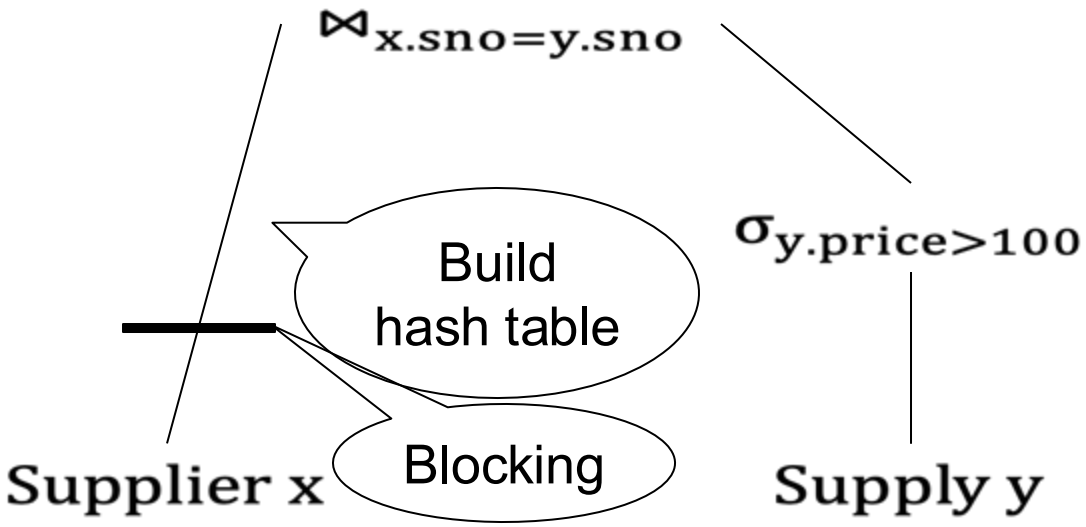


Volcano Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```

Pipelining changes
the code significantly

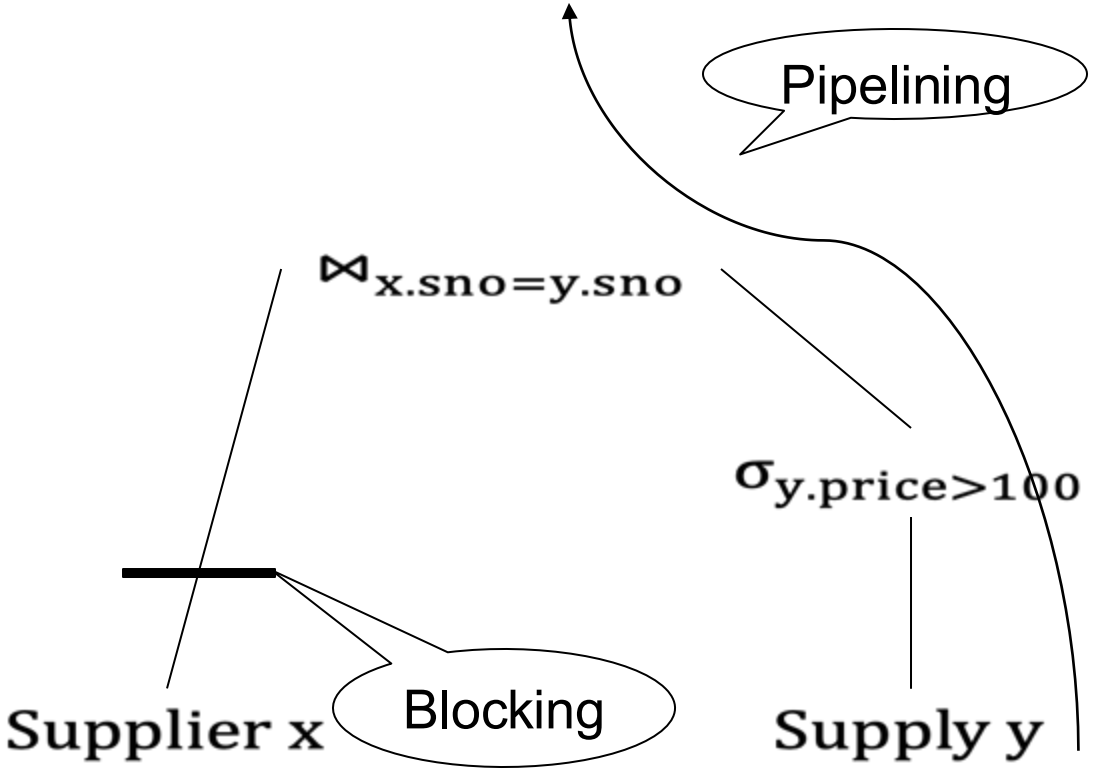


Volcano Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```

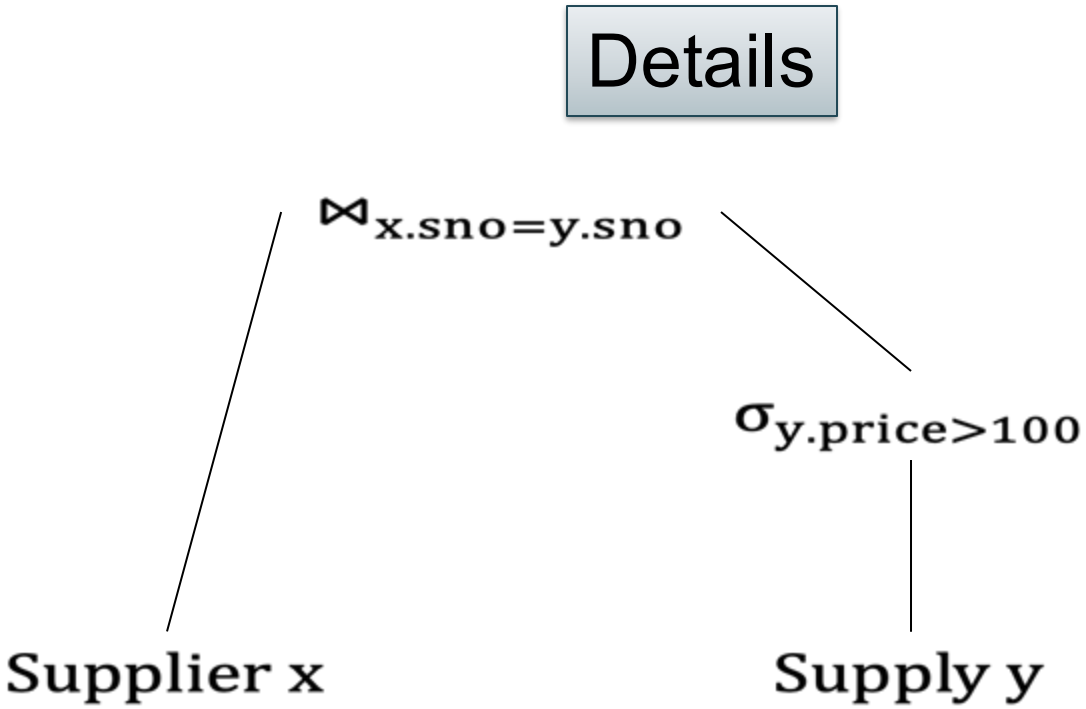
Pipelining changes the code significantly



Volcano Example

”Normal” hash-join

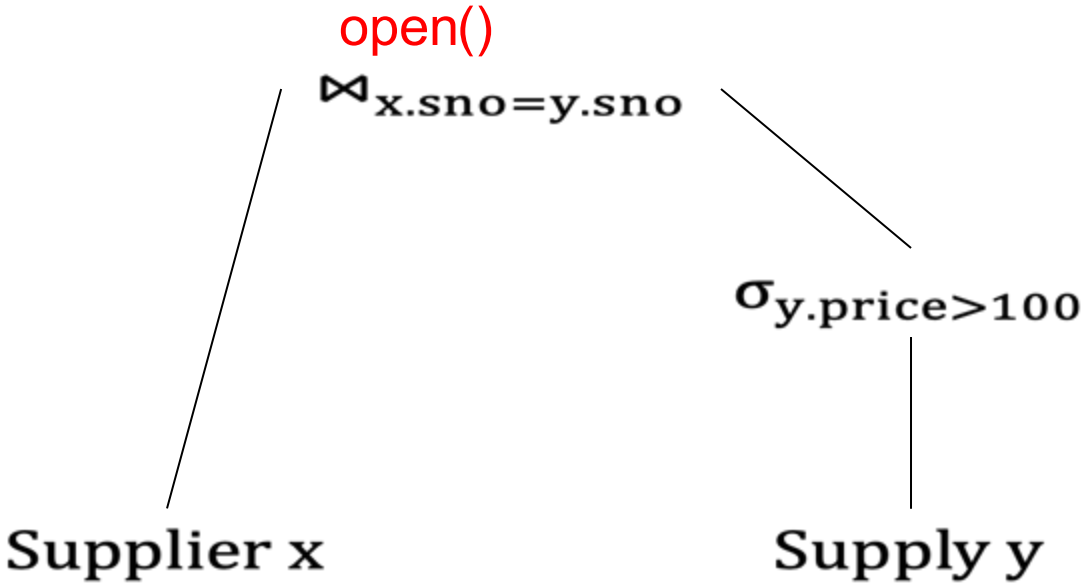
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

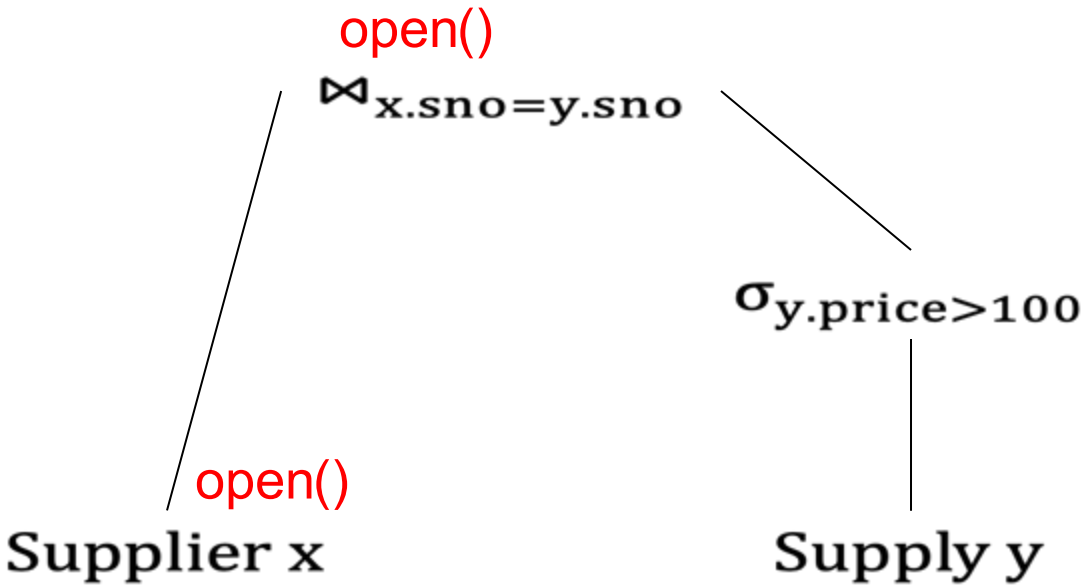
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

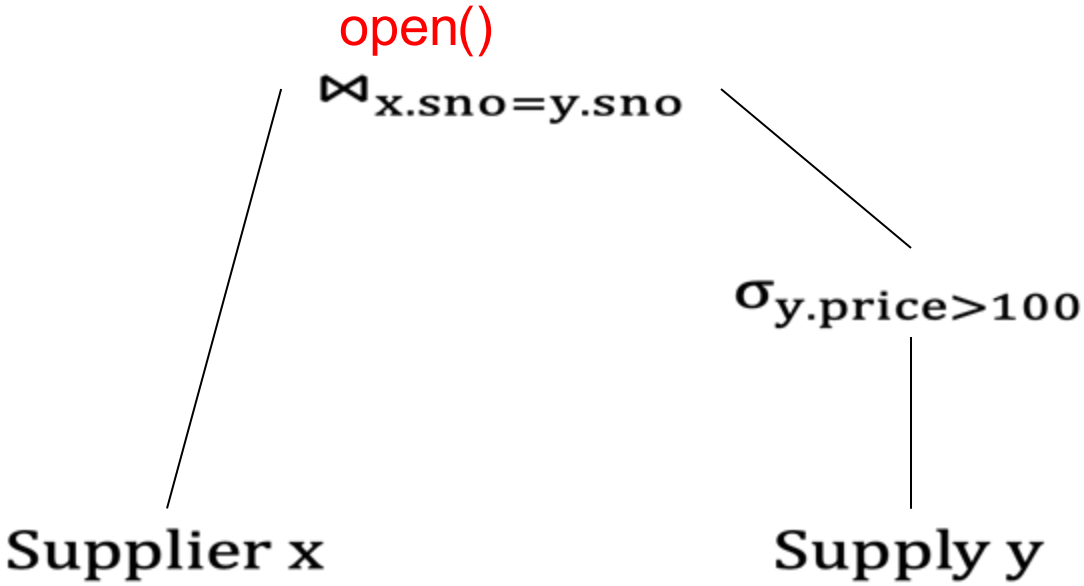
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

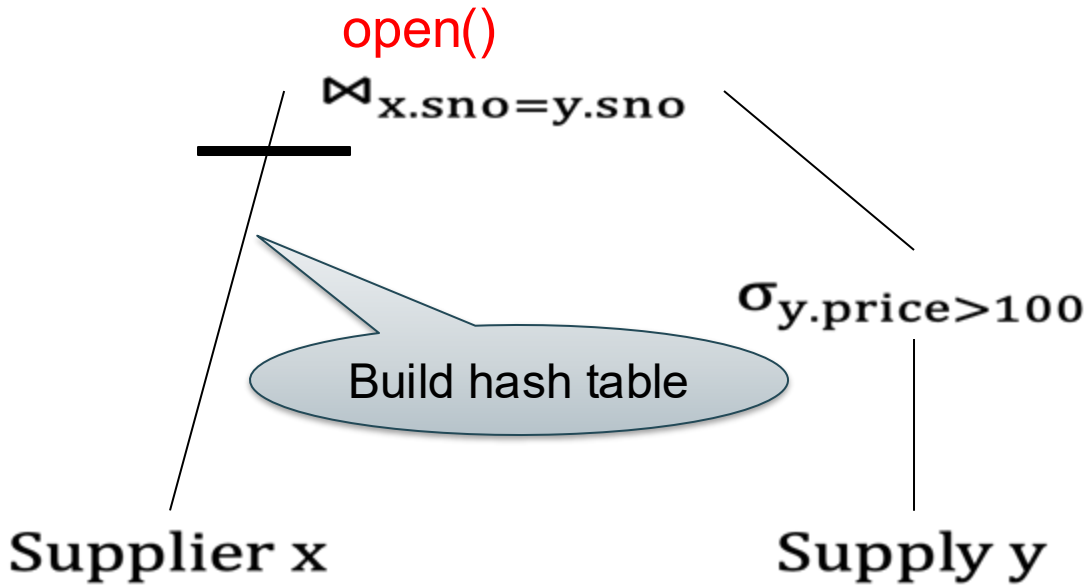
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

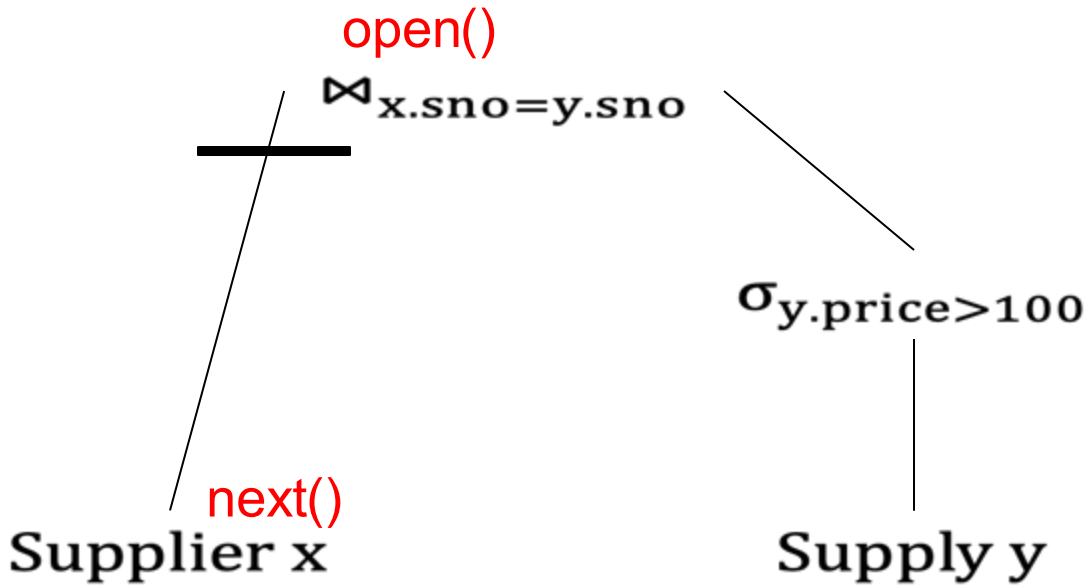
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

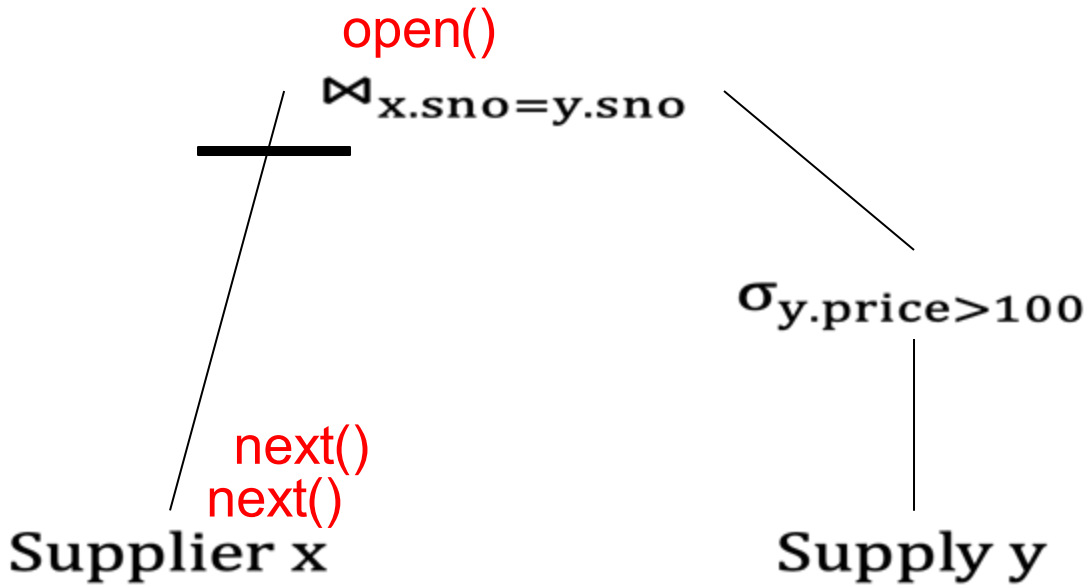
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

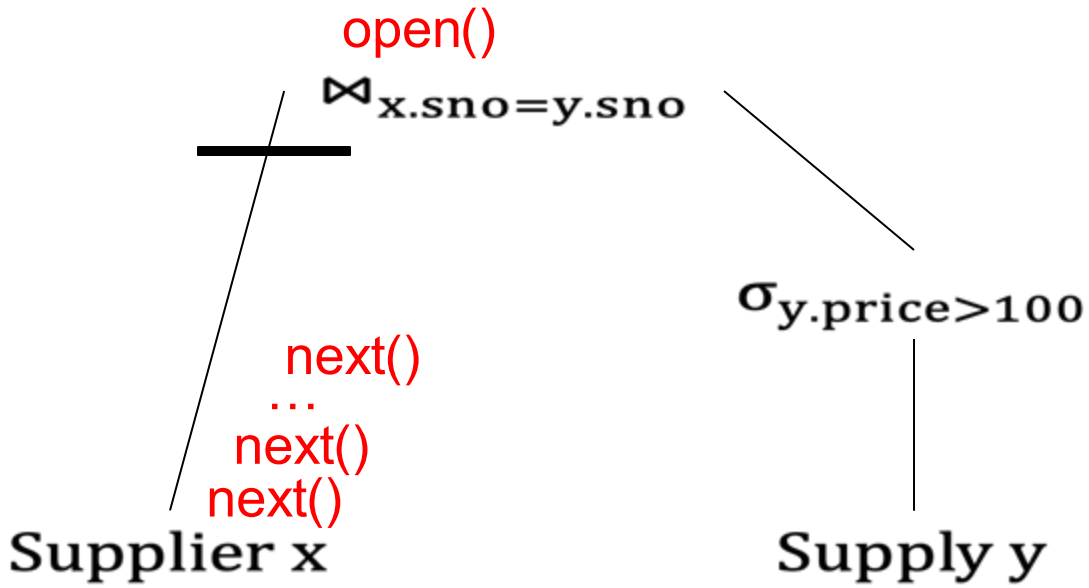
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

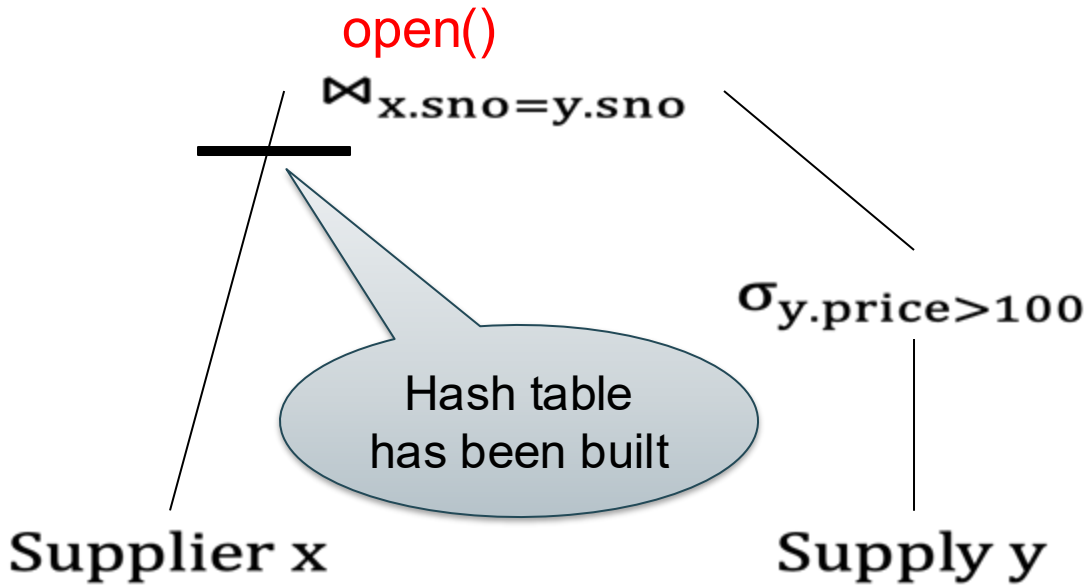
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

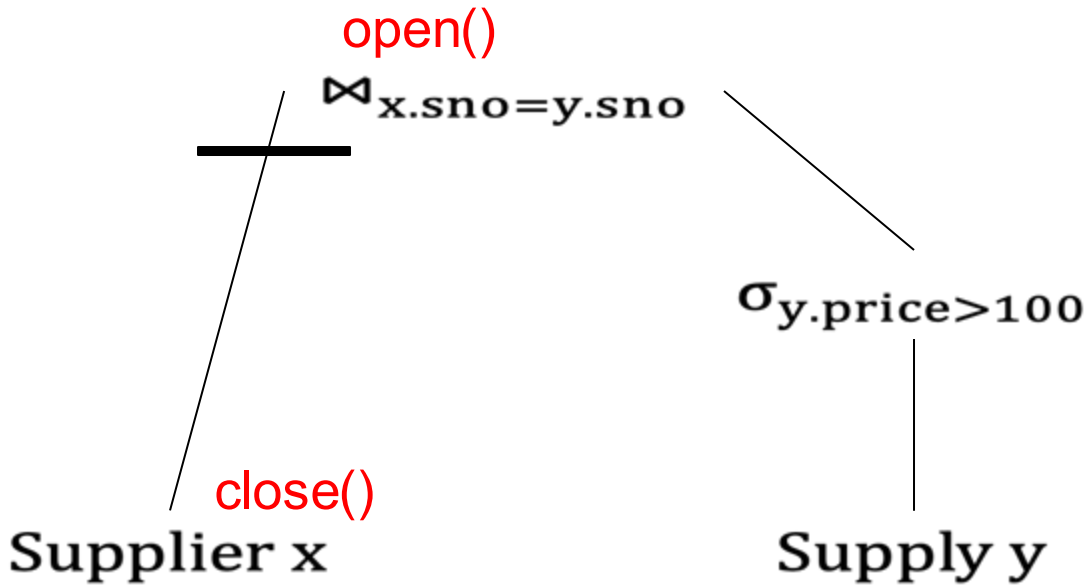
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

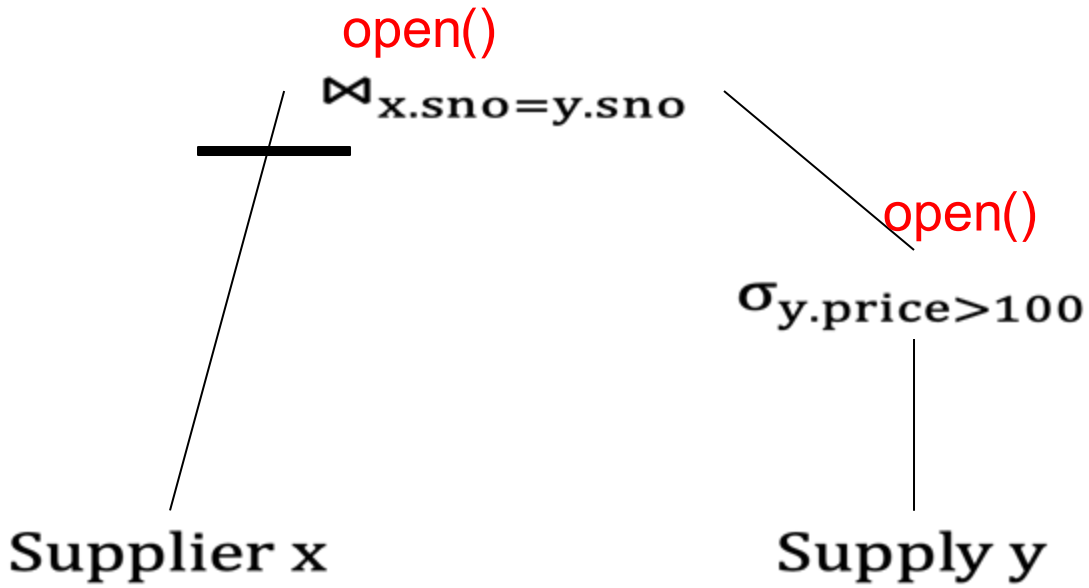
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

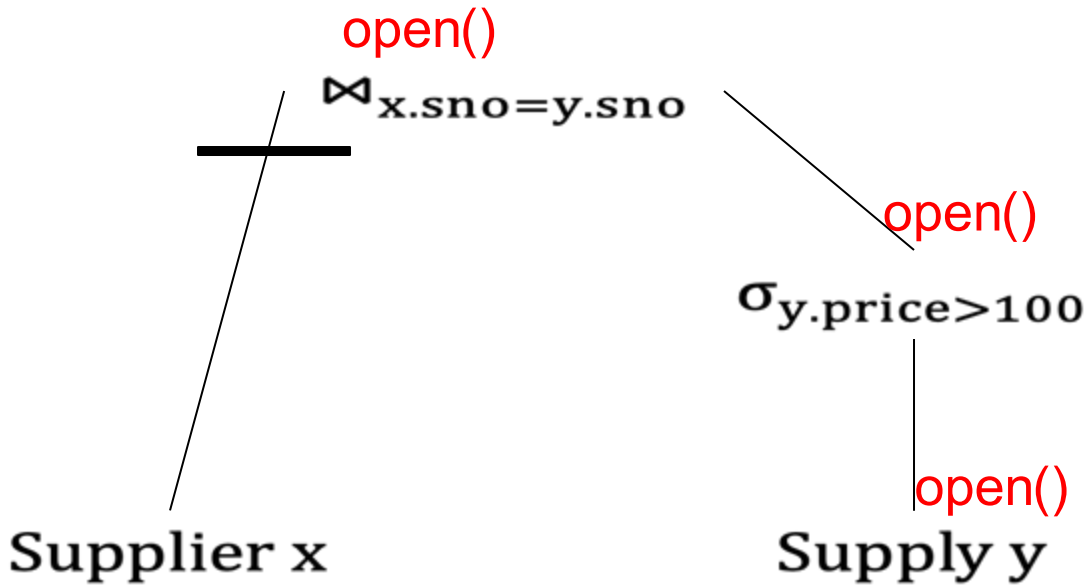
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

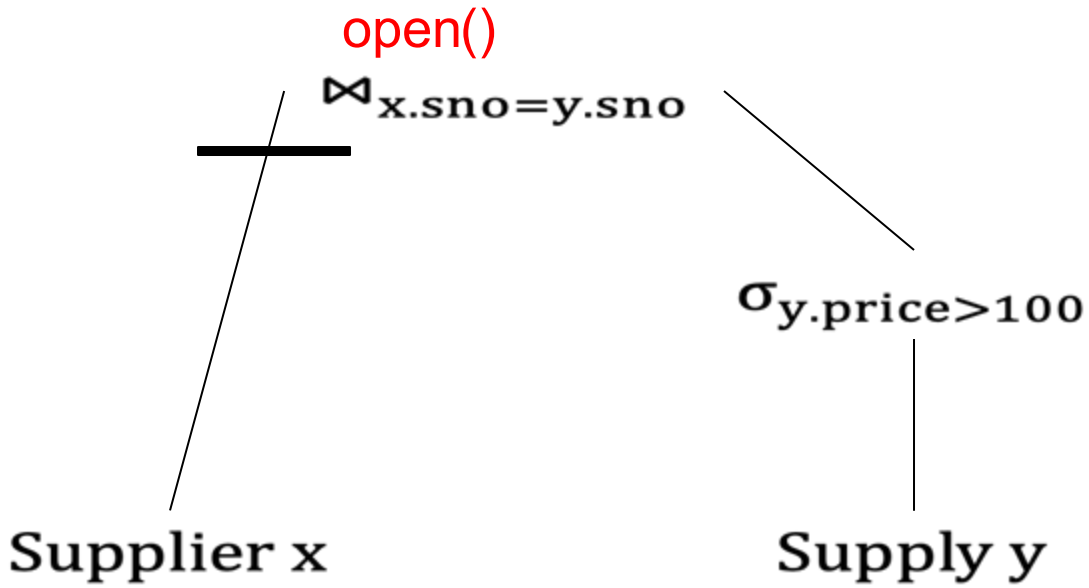
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

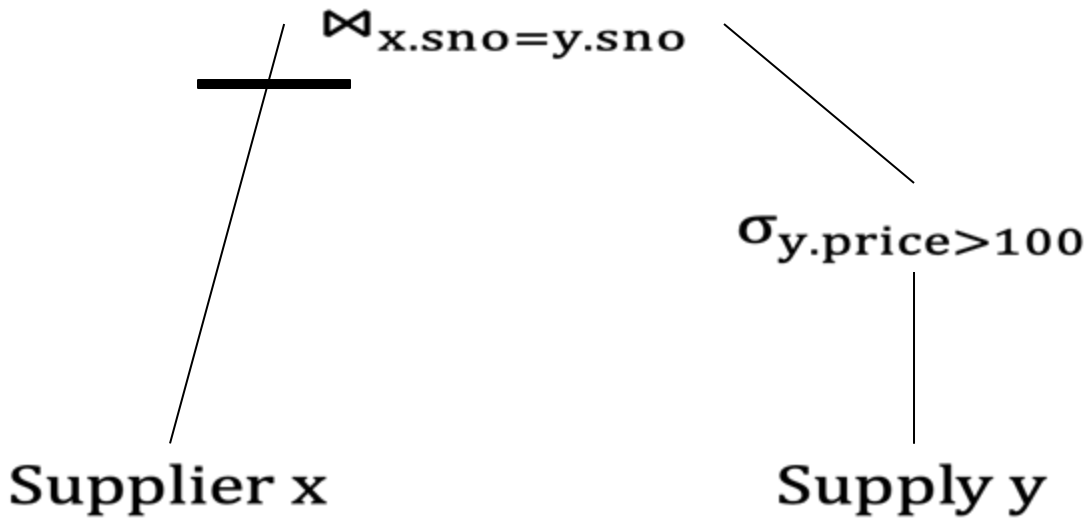
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

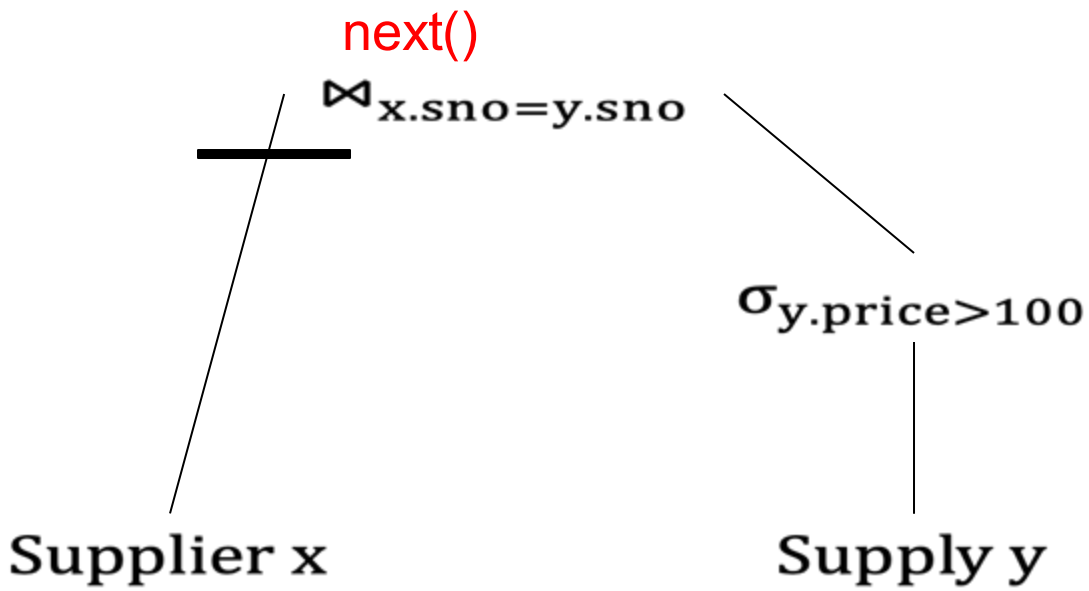
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

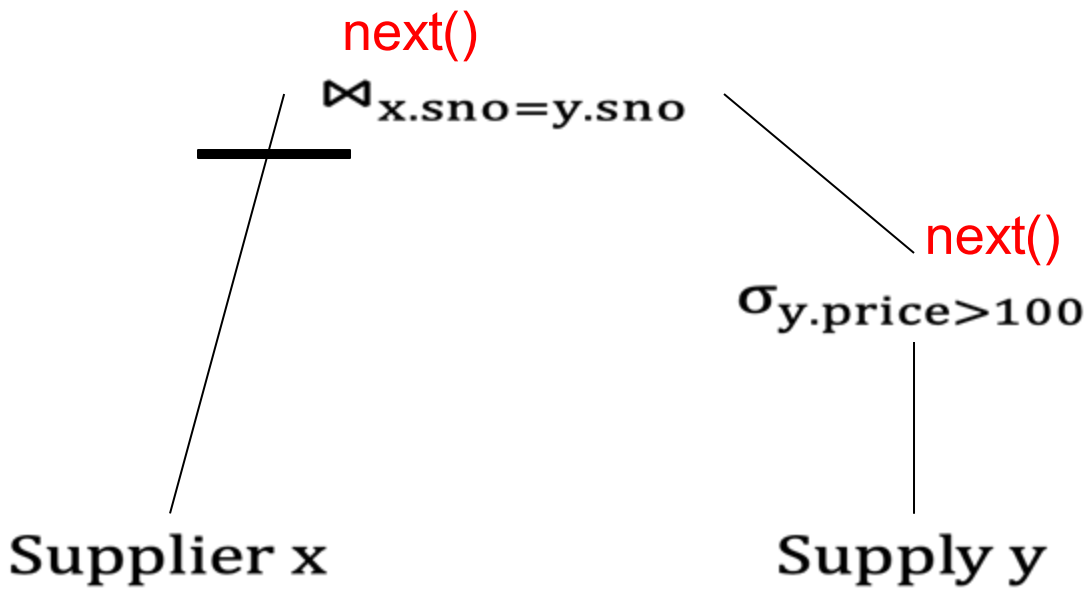
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

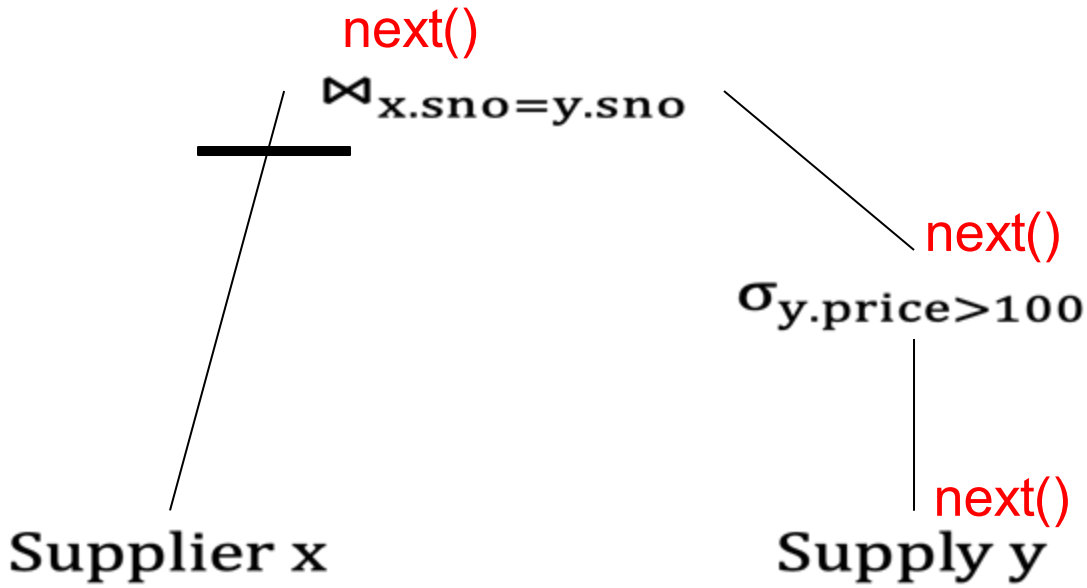
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

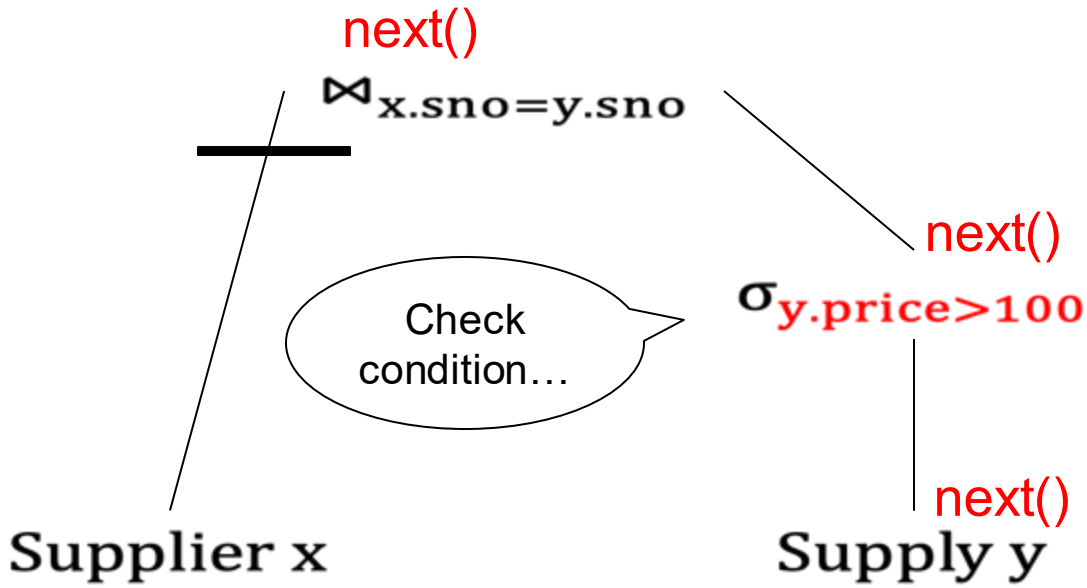
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

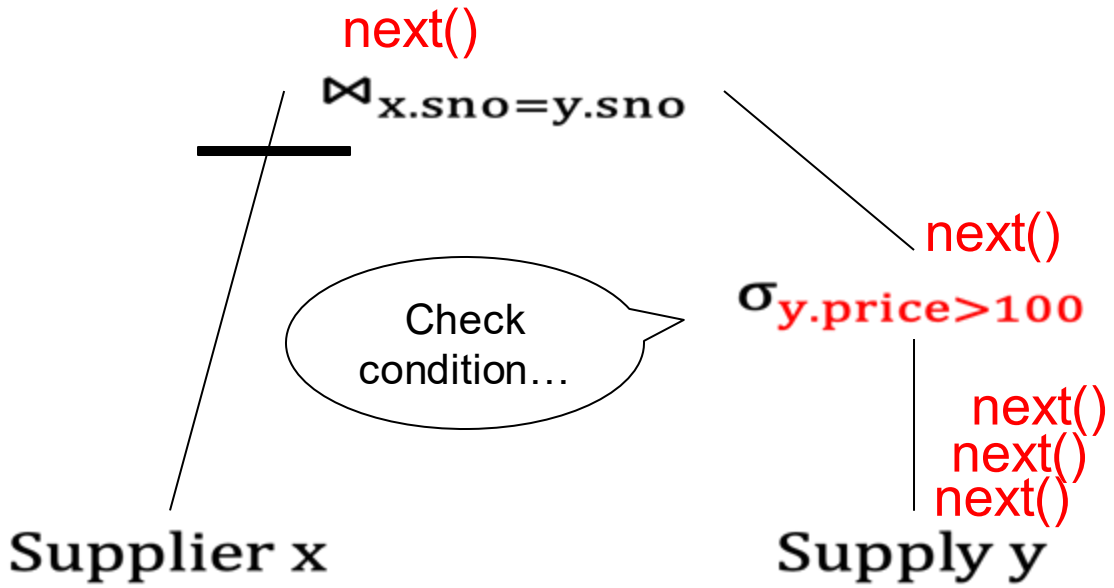
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

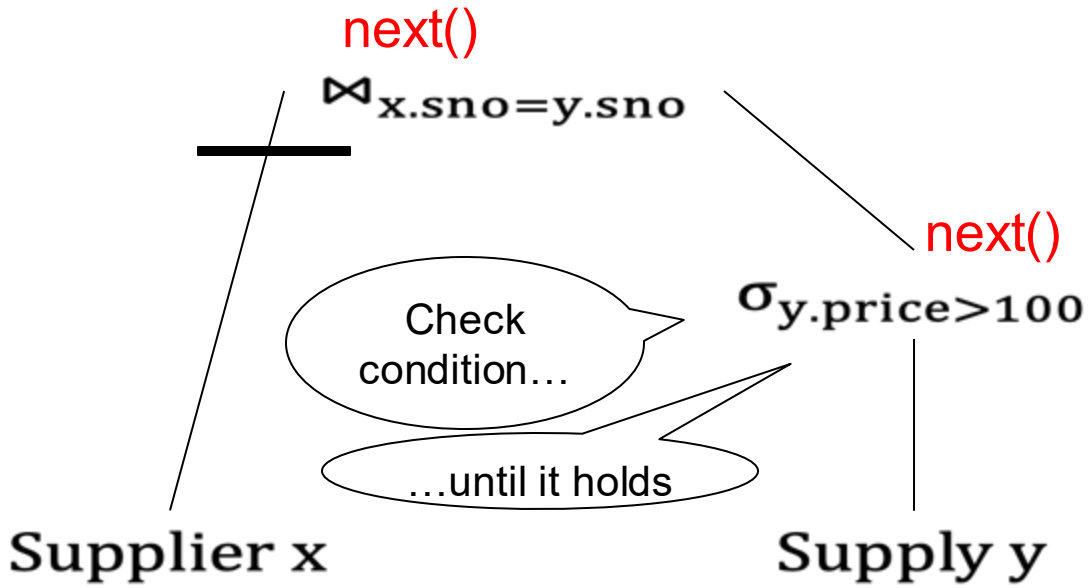
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

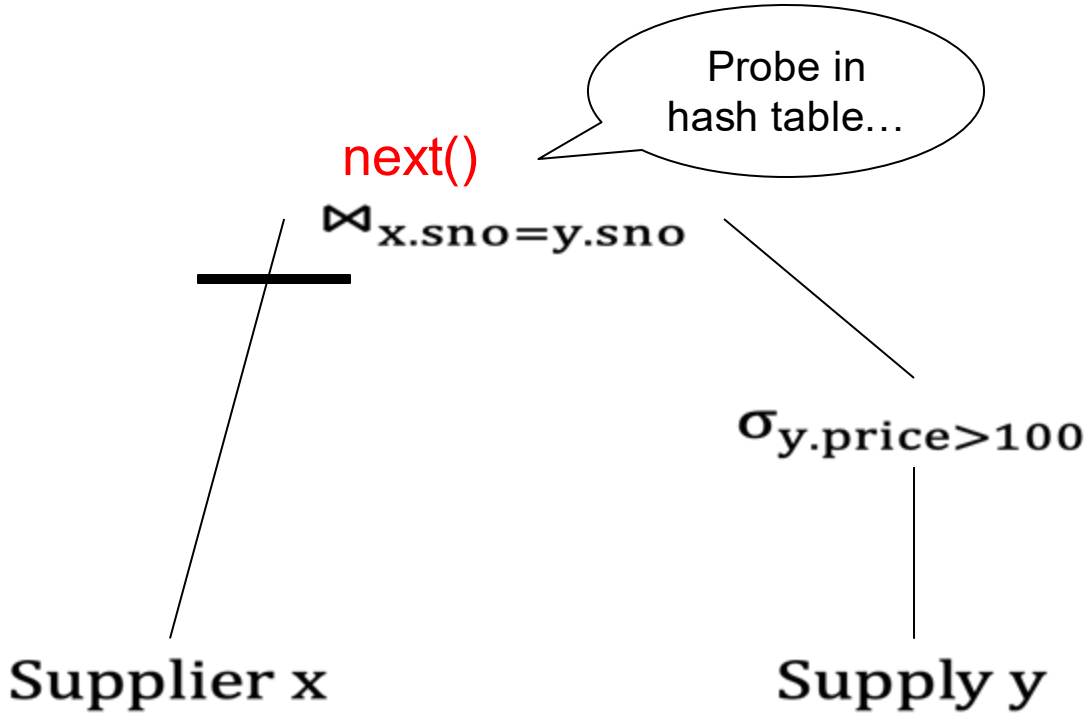
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

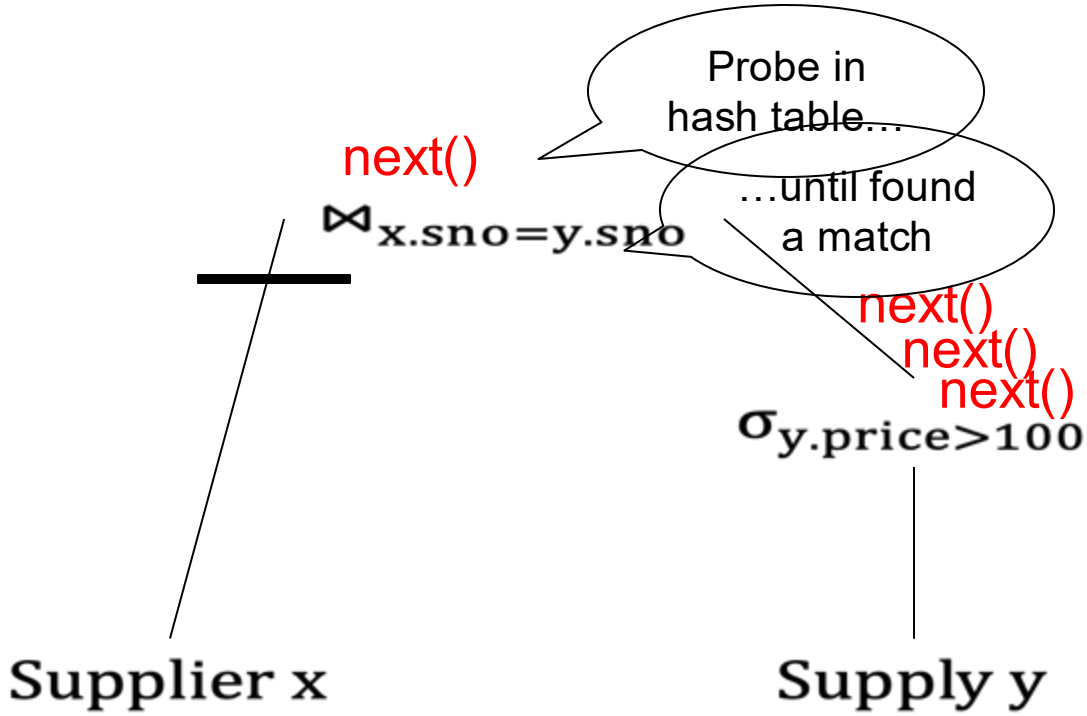
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

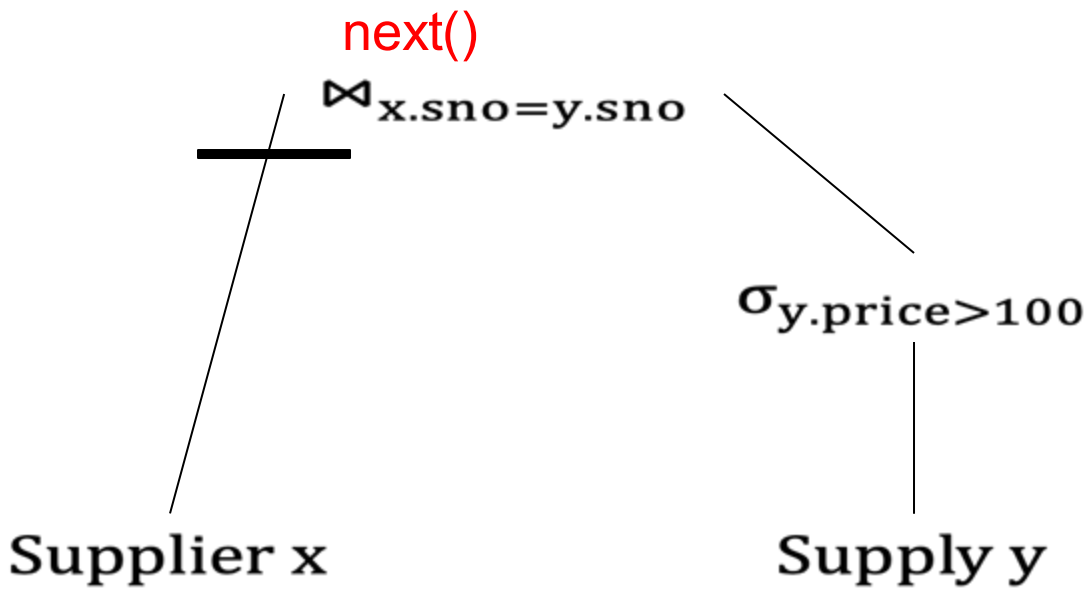
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

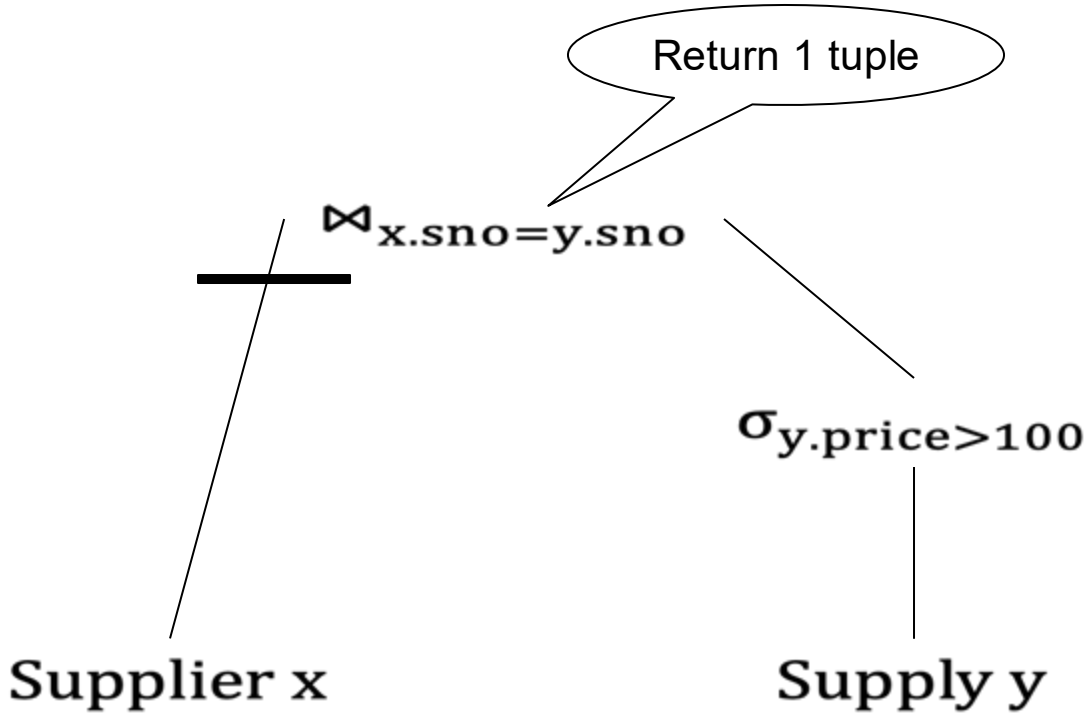
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

”Normal” hash-join

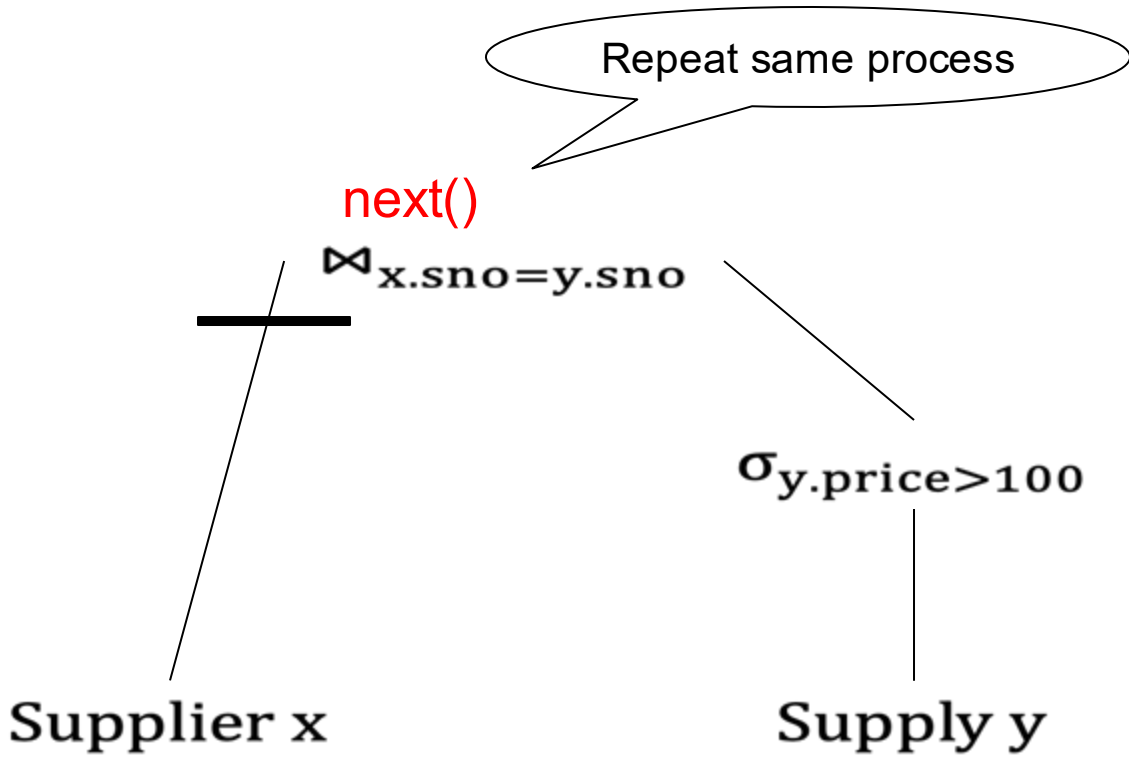
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Volcano Example

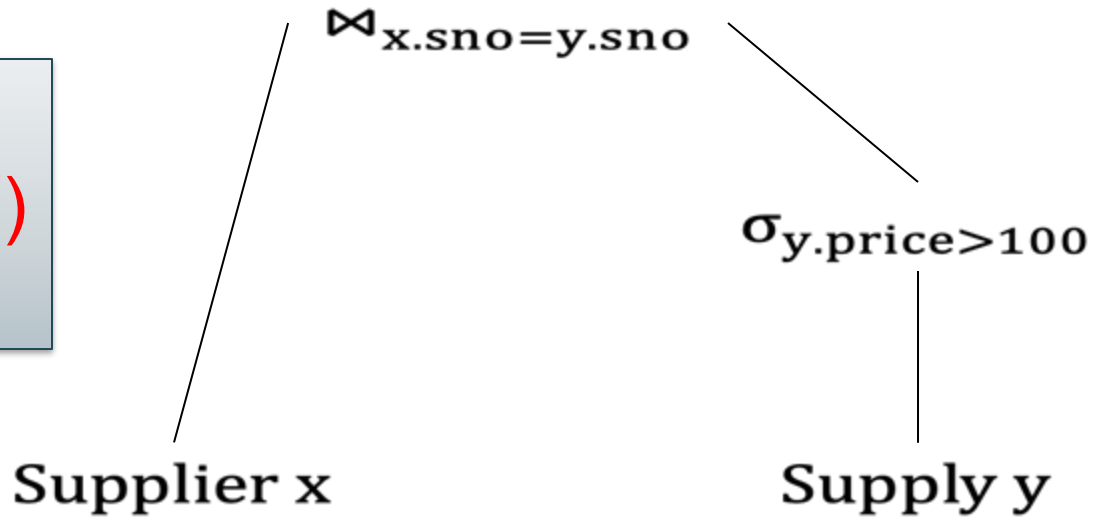
”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



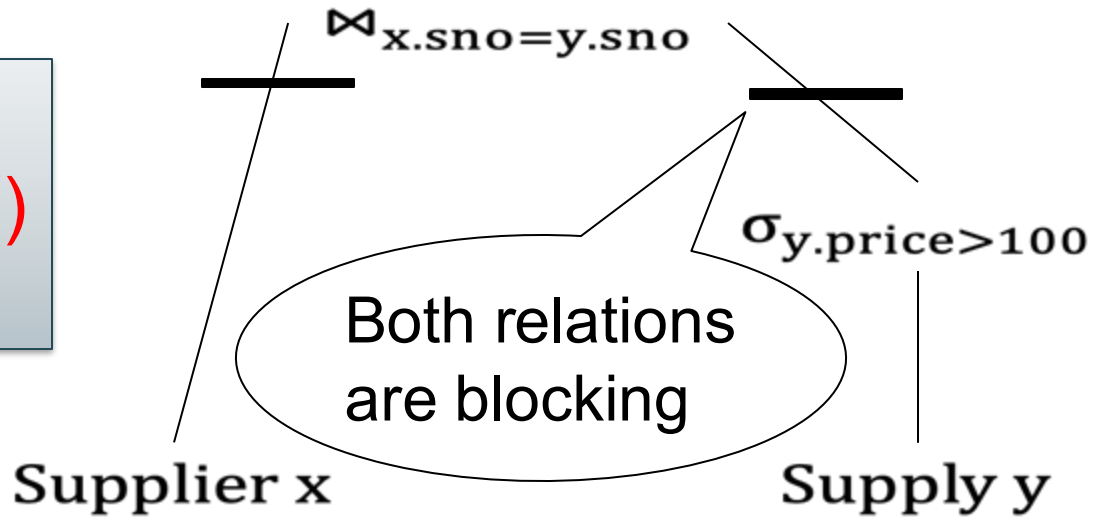
Volcano Example

Describe
open(), next(), close()
for merge-join



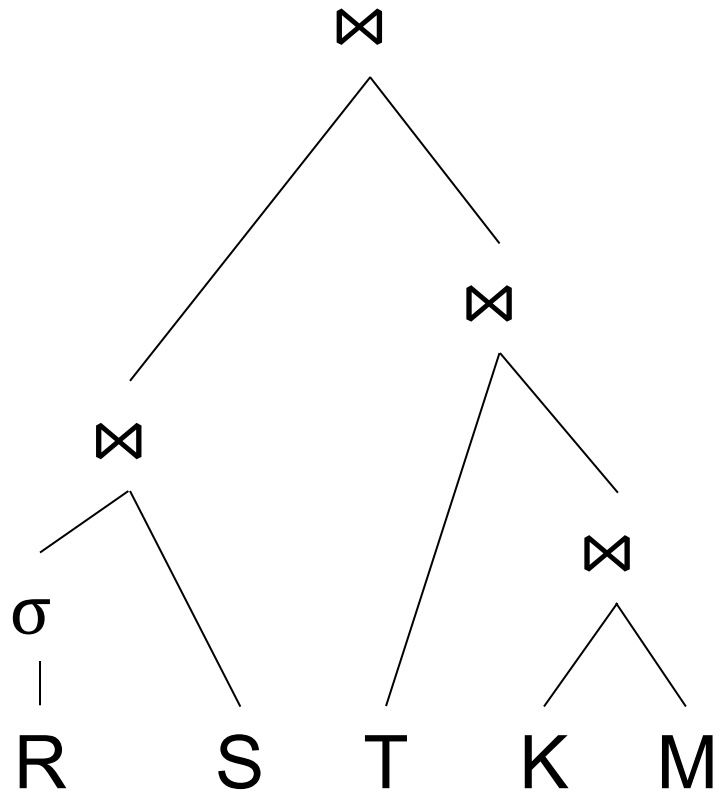
Volcano Example

Describe
open(), next(), close()
for merge-join



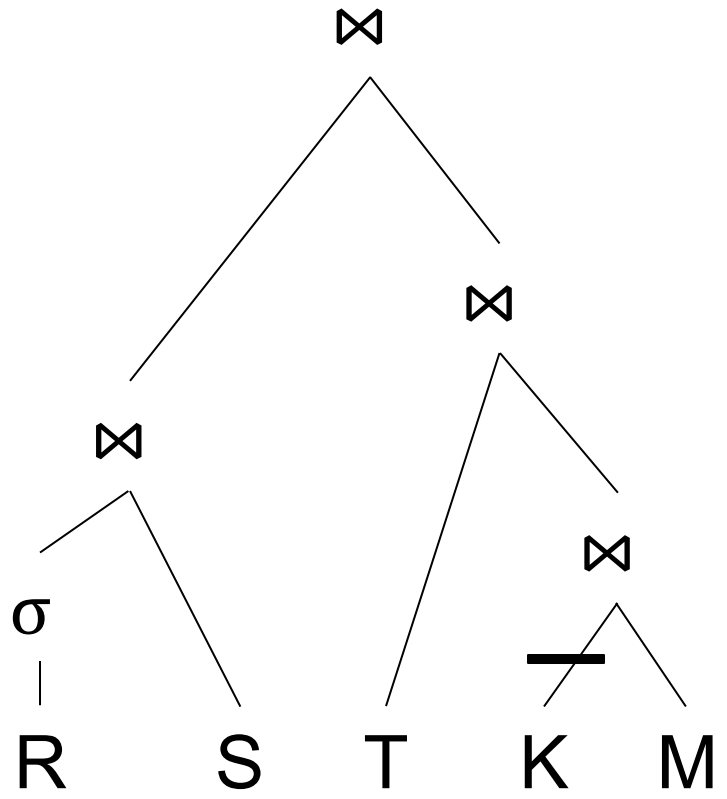
Volcano Model

Usually: block the outer relation, pipeline the inner relation



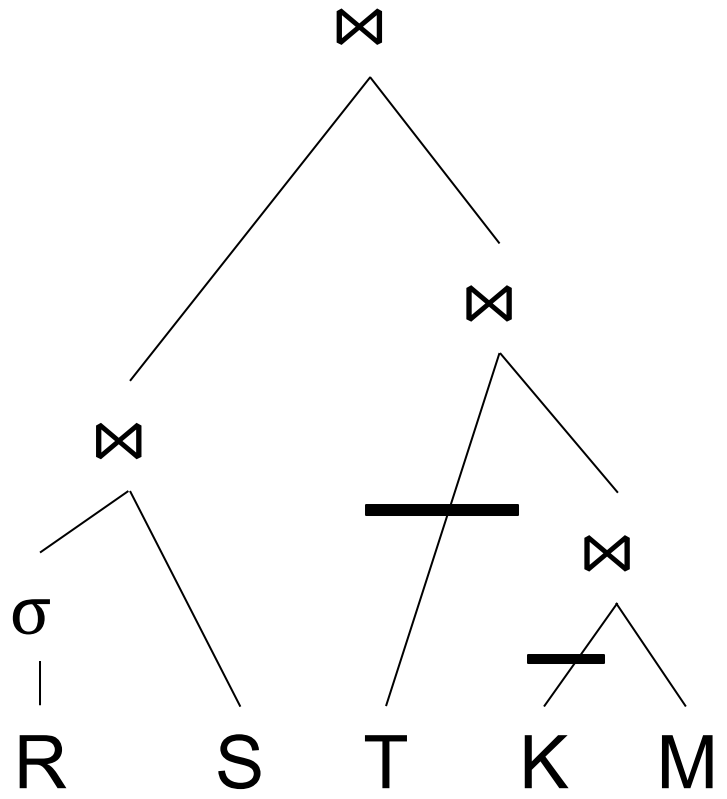
Volcano Model

Usually: block the outer relation, pipeline the inner relation



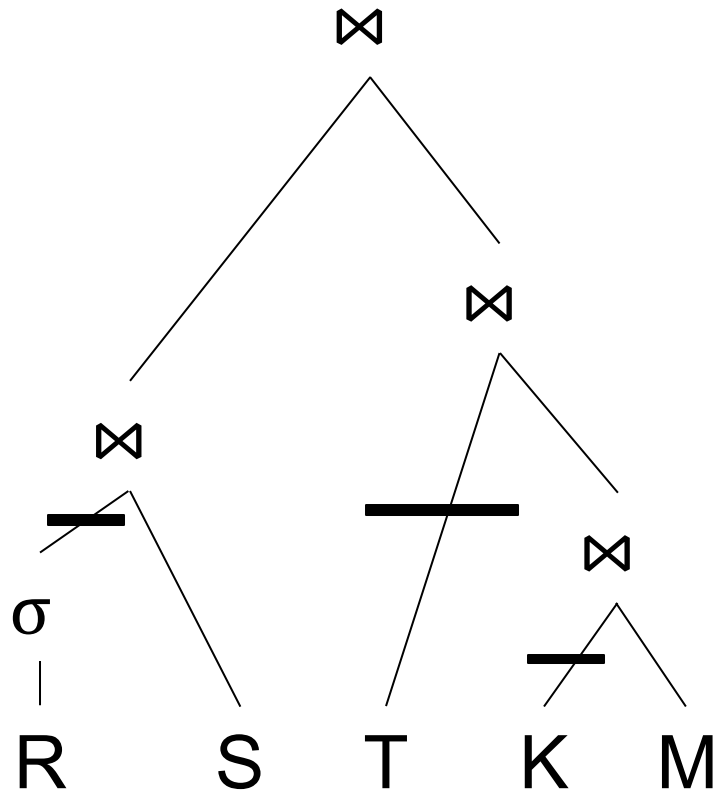
Volcano Model

Usually: block the outer relation, pipeline the inner relation



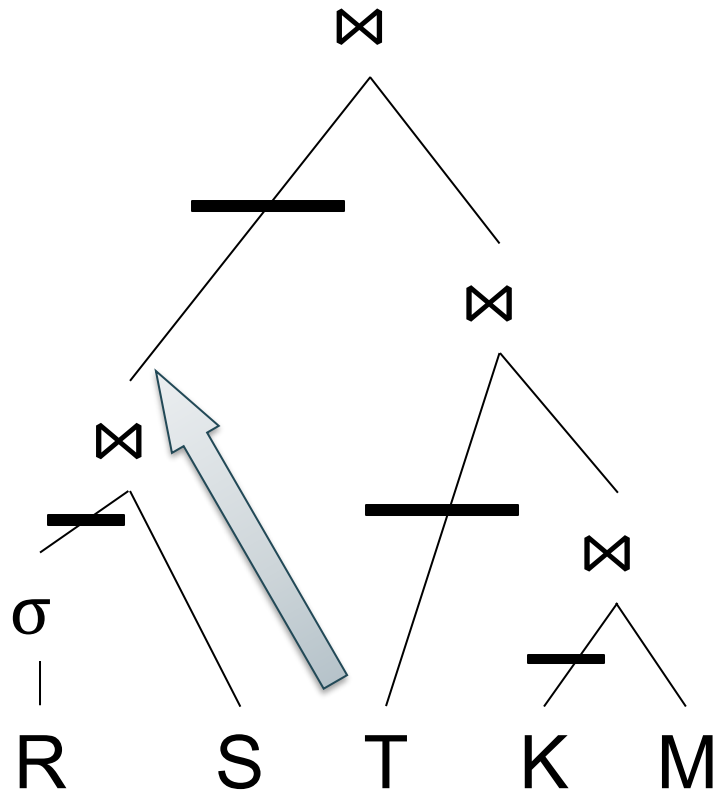
Volcano Model

Usually: block the outer relation, pipeline the inner relation



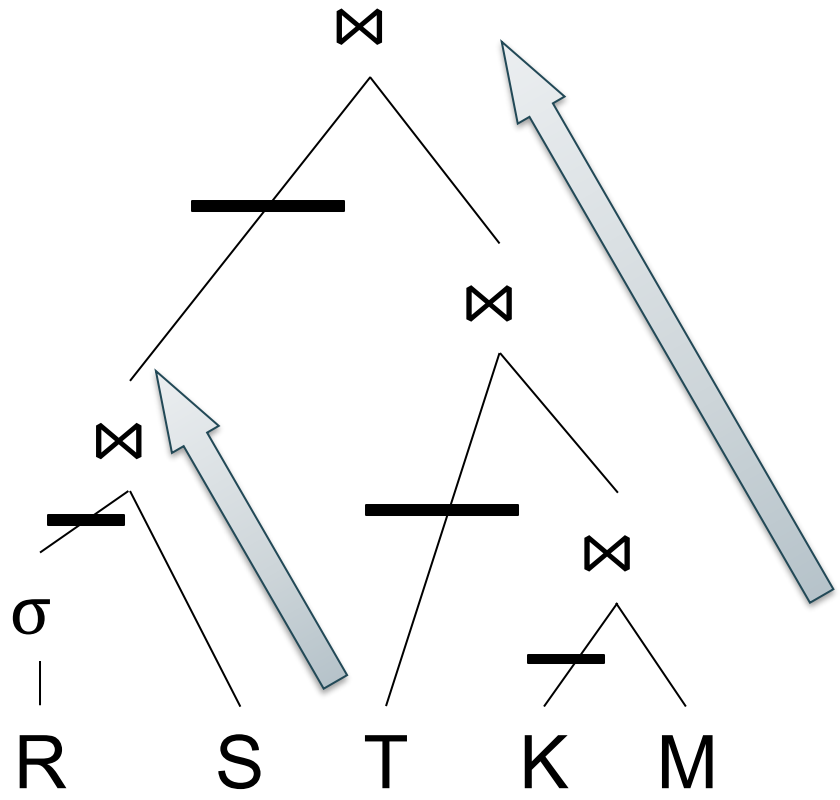
Volcano Model

Usually: block the outer relation, pipeline the inner relation



Volcano Model

Usually: block the outer relation, pipeline the inner relation

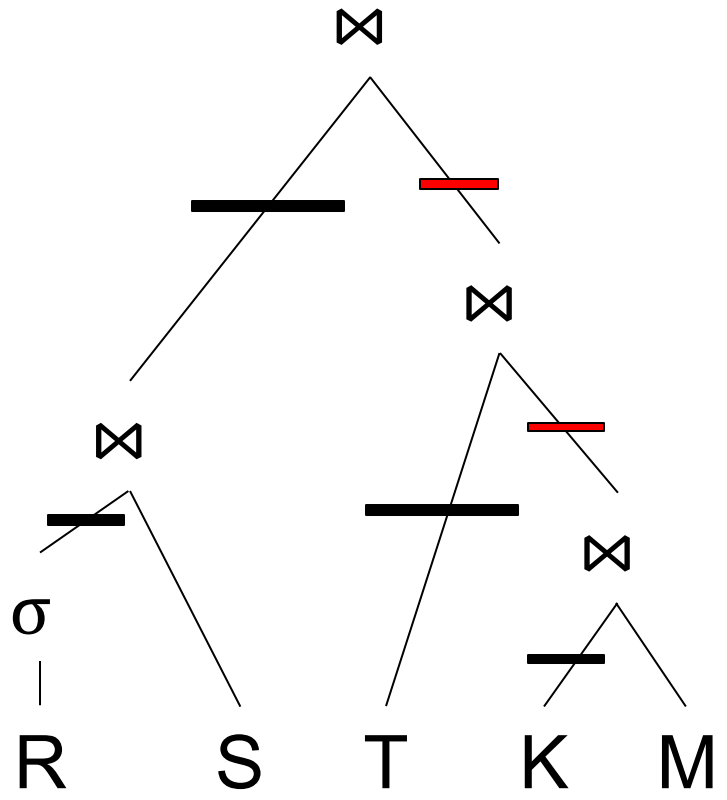


Volcano Model

Usually: block the outer relation, pipeline the inner relation

Lots of confusion in literature on whether the outer table is blocking, or the inner table is blocking.

(In class: outer table is more logical)



Volcano Model

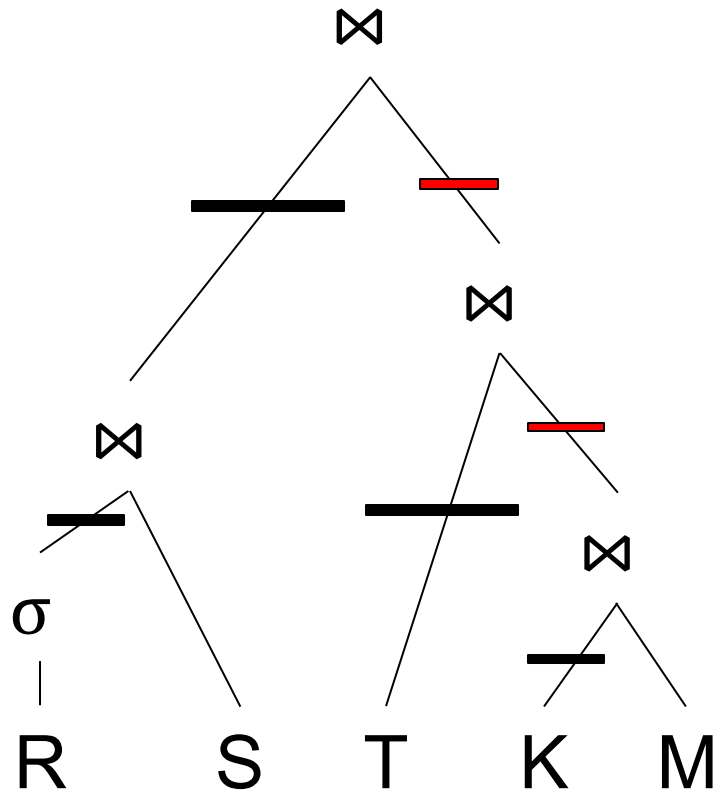
Usually: block the outer relation, pipeline the inner relation

Lots of confusion in literature on whether the outer table is blocking, or the inner table is blocking.

(In class: outer table is more logical)

But may also block everything

E.g. merge-join



Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
  Tuple next () {
    boolean found = false;
    Tuple r = null;
    while (!found) {
       r = c.next();
       if (r == null) break;
       found = p(r);
    }
    return r;
  }
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
  Tuple next () {
    boolean found = false;
    Tuple r = null;
    while (!found) {
       r = c.next();
       if (r == null) break;
       found = p(r);
    }
    return r;
  }
  void close () { c.close(); }
}
```

Volcano Model

Query plan execution

```
Operator q = parse(“SELECT ...”); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree
```

Volcano Model

Query plan execution

```
Operator q = parse("SELECT ...");  # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
```

Volcano Model

Query plan execution

```
Operator q = parse("SELECT ...”); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break; # end of results
  else printOnScreen(t); # output tuple
}
```

Volcano Model

Query plan execution

```
Operator q = parse("SELECT ...”); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break; # end of results
  else printOnScreen(t); # output tuple
}
q.close();
```

Pipeline v.s. Blocking

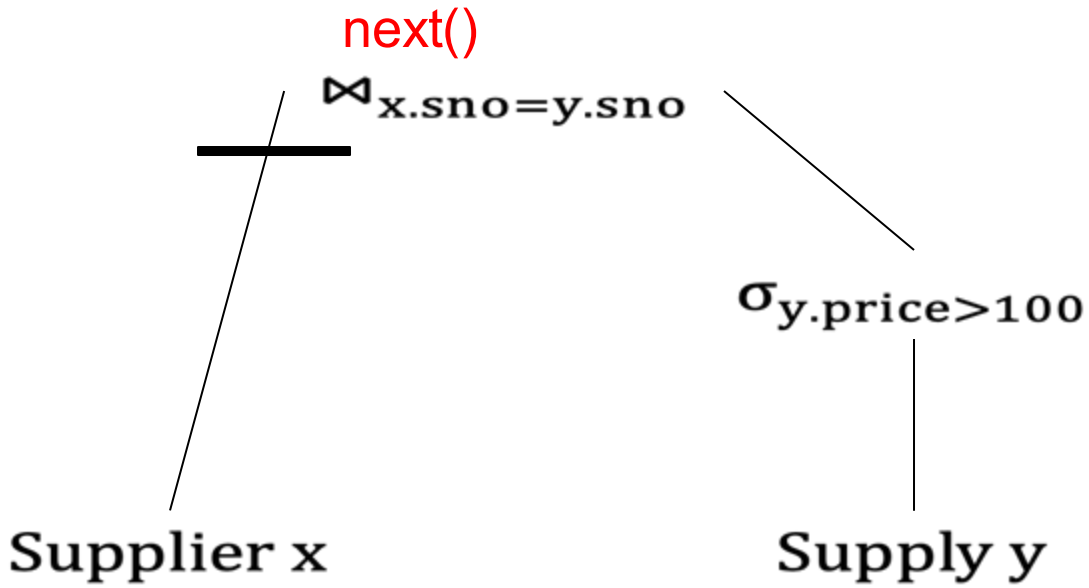
- Pipeline
  - Tuples move all the way through up the query plan
  - Advantages: speed
  - Disadvantage: need all hash tables in memory
- Blocking
  - Compute and store on disk entire subplan
  - Advantage: needs less memory
  - Disadvantage: slower

Vectorized Query Processing

- next() returns a batch of tuples
- Typically 1000 tuples
- This reduced the communication overhead

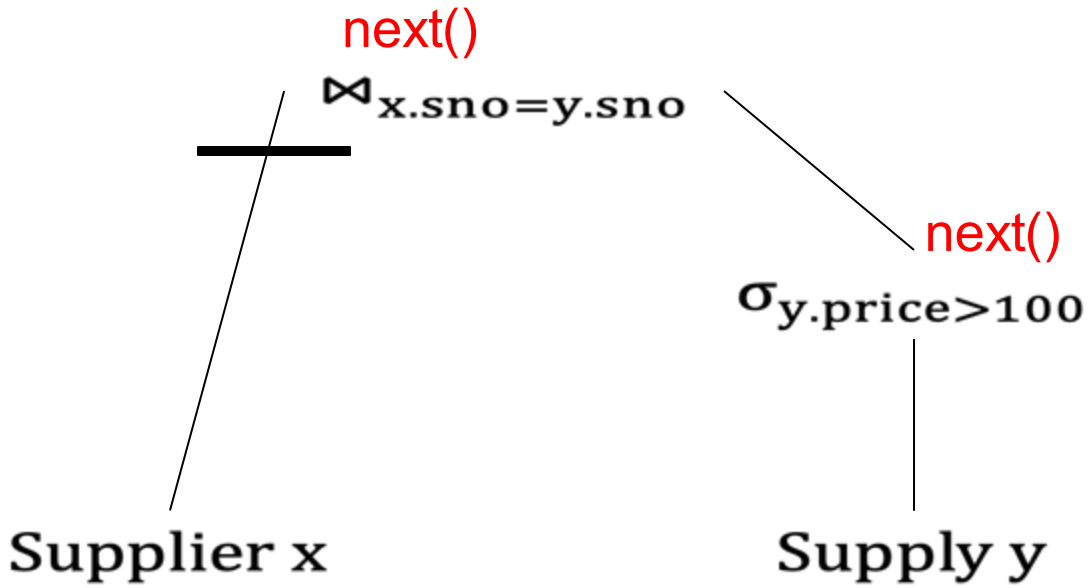
Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



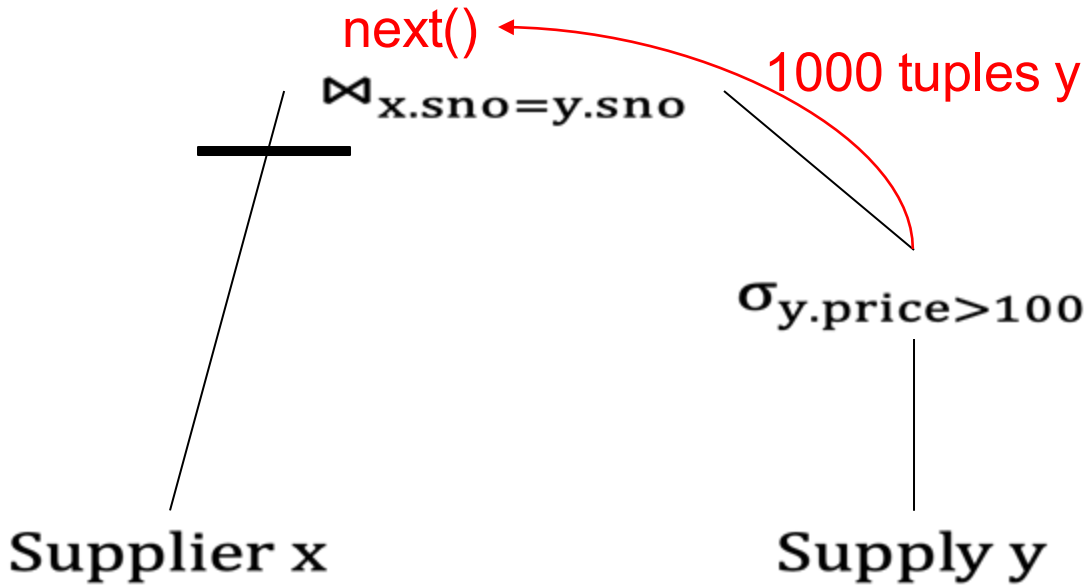
Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



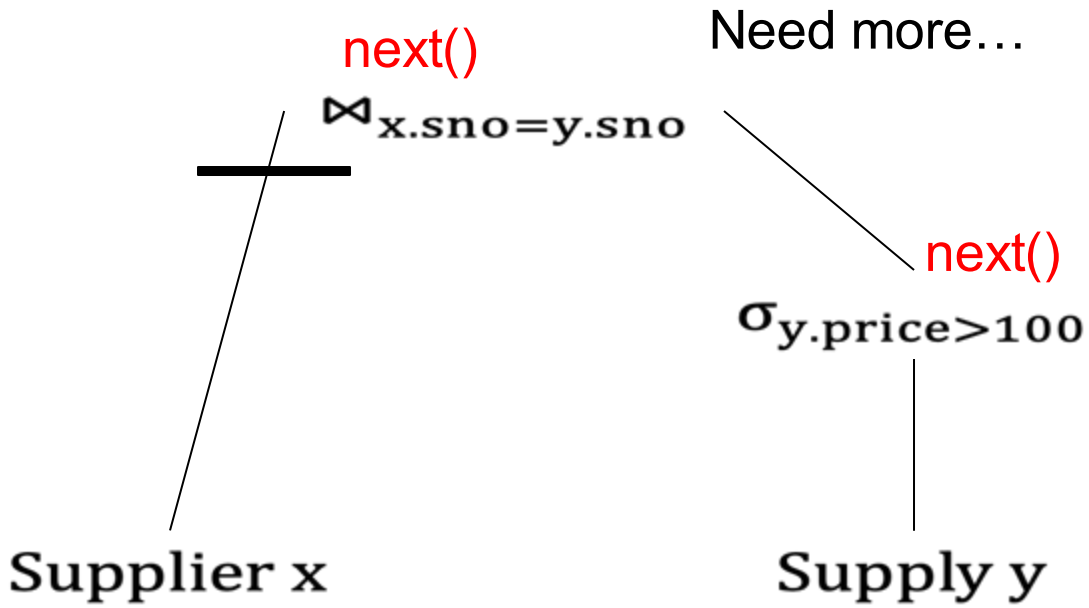
Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



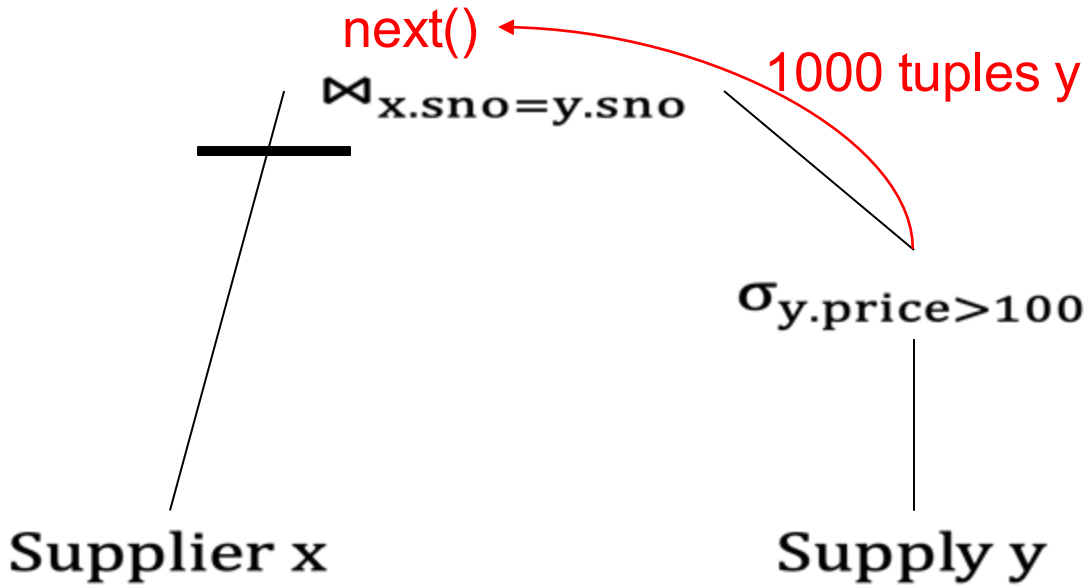
Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



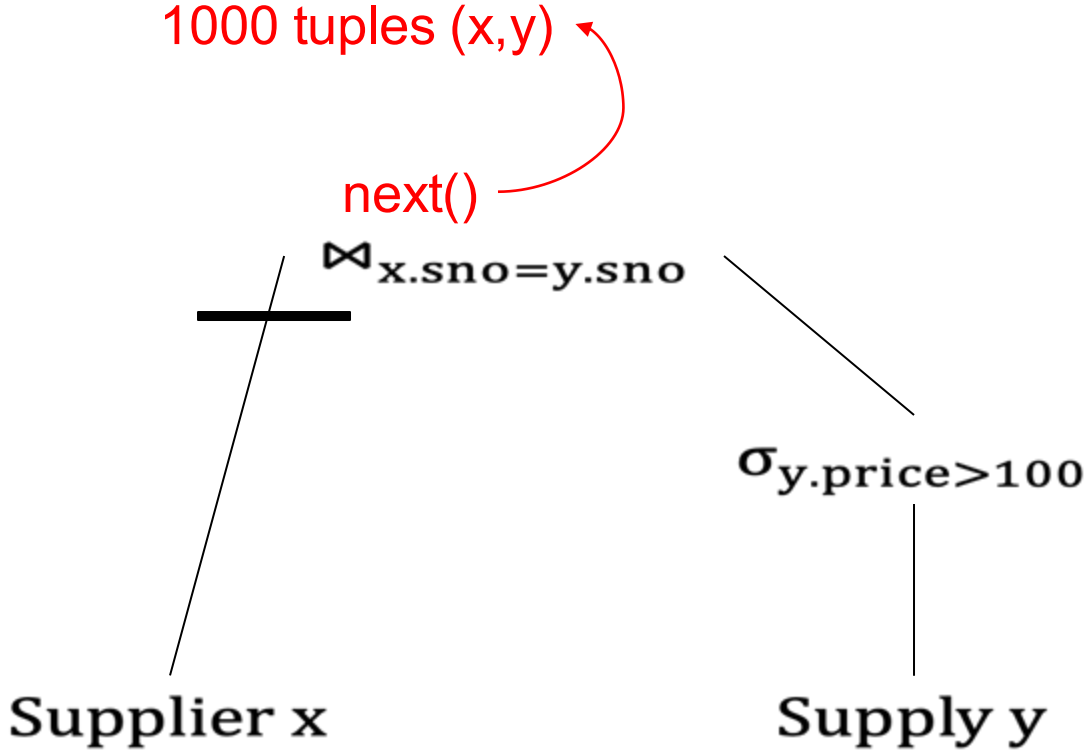
Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Vectorized Query Processing

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Discussion

- Volcano iterator model is simple
- Used by most RDBMS
- Works great when query is interpreted
- Doesn't work well when it is compiled

Discussion

- Volcano iterator model is simple
- Used by most RDBMS
- Works great when query is interpreted
- Doesn't work well when it is compiled

What does interpreted or compiled mean?

The Push Model

Today's Paper

How to Architect a Query Compiler

Query execution: interpret v.s. compile

- What are the pros/cons?
- What was the traditional approach?
- What changed recently?

Data-Driven Model

Each operator exports four methods:

- Open()
- Produce()
- Consume()
- Close()

Data-Driven Model

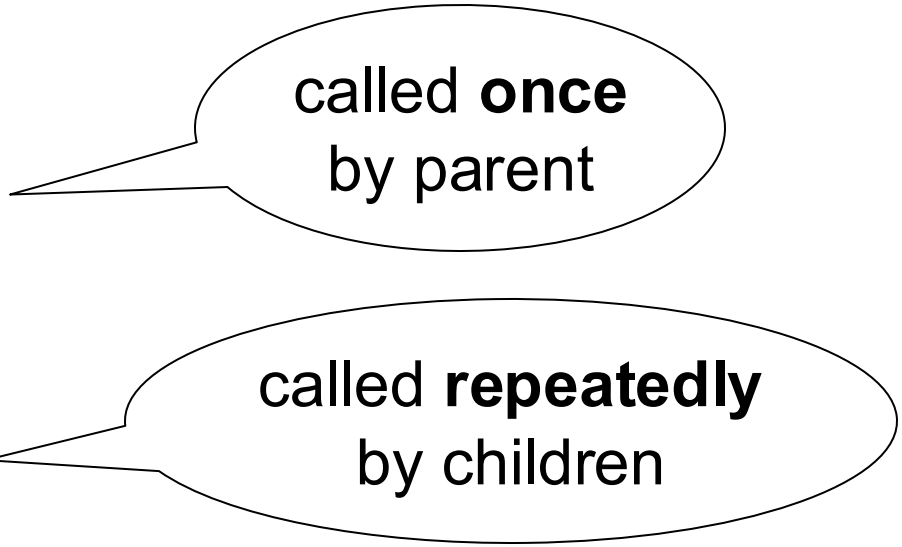
Each operator exports four methods:

- Open()
- Produce()
- Consume()
- Close()

Data-Driven Model

Each operator exports four methods:

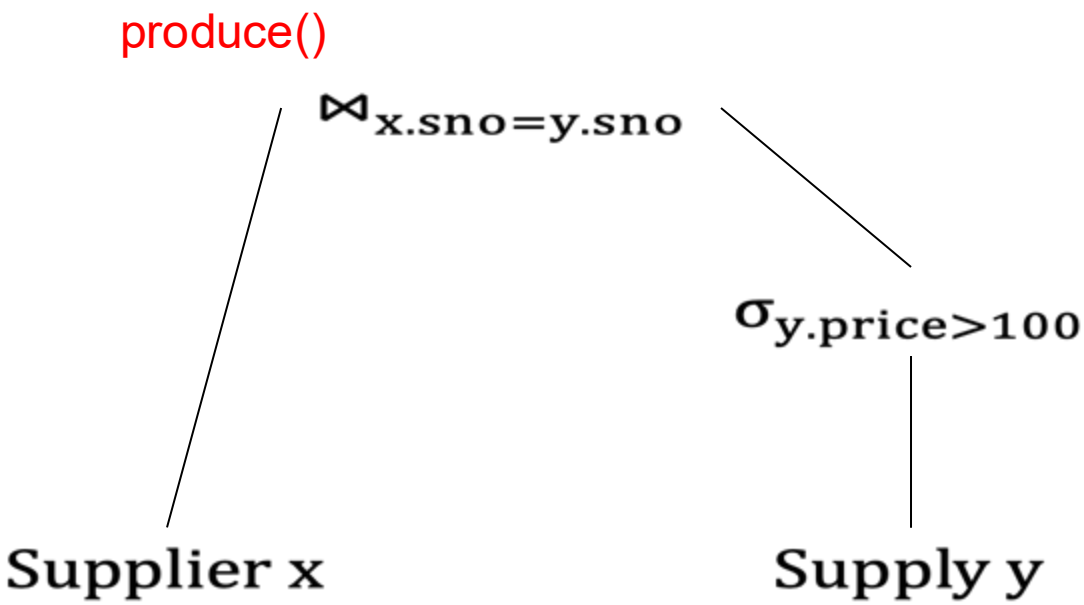
- Open()
- Produce()
- Consume()
- Close()



Example

”Normal” hash-join

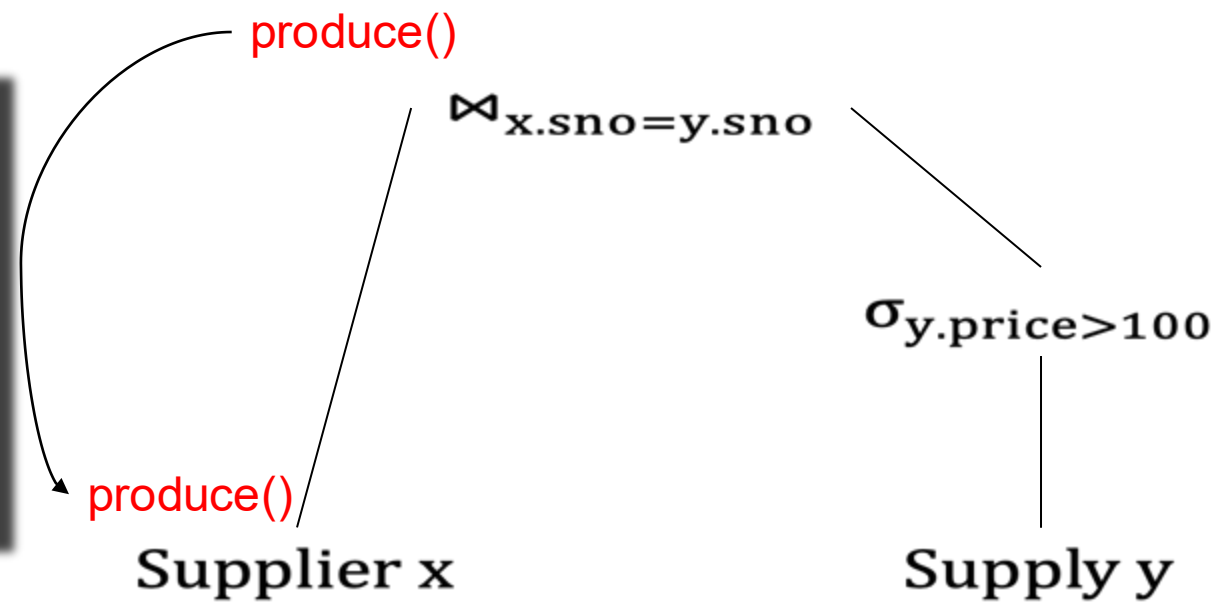
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

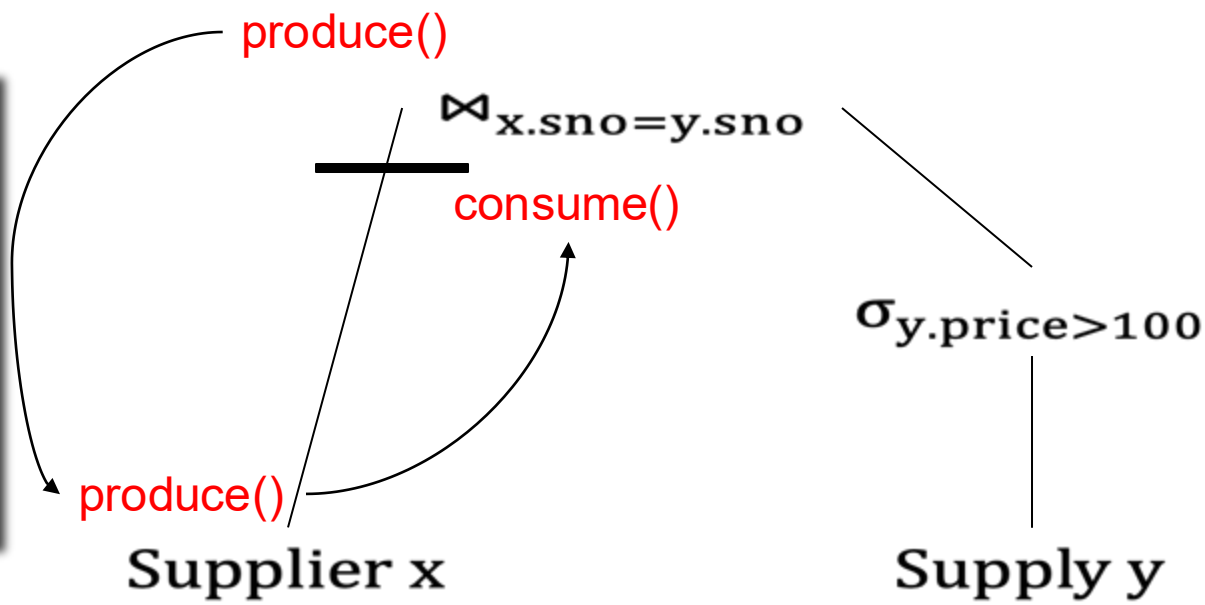
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

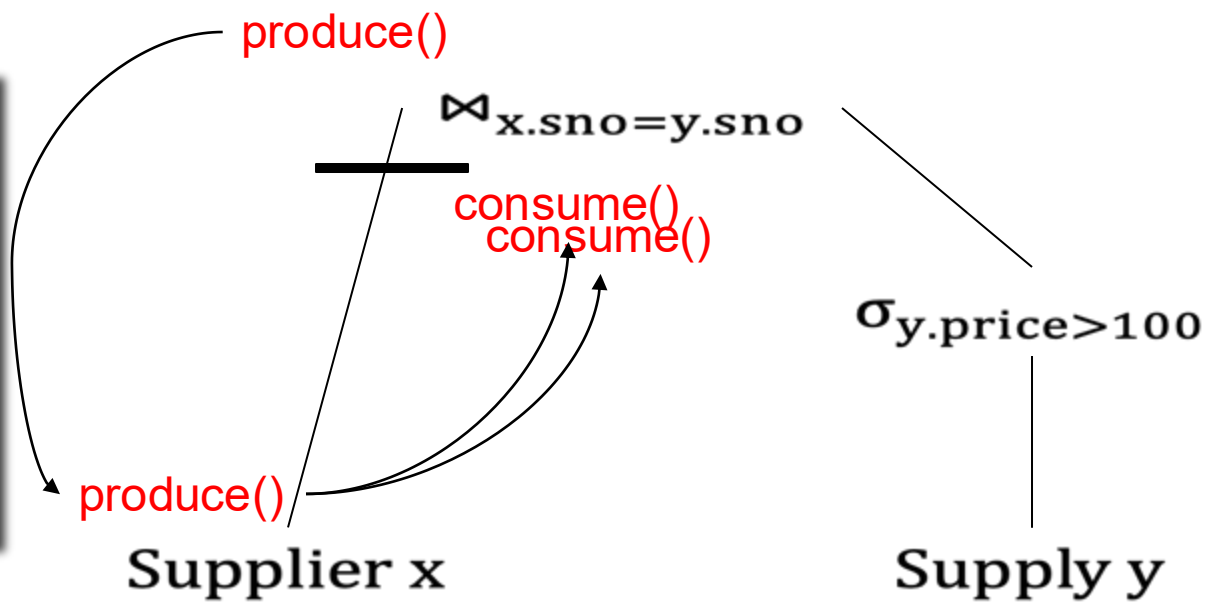
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

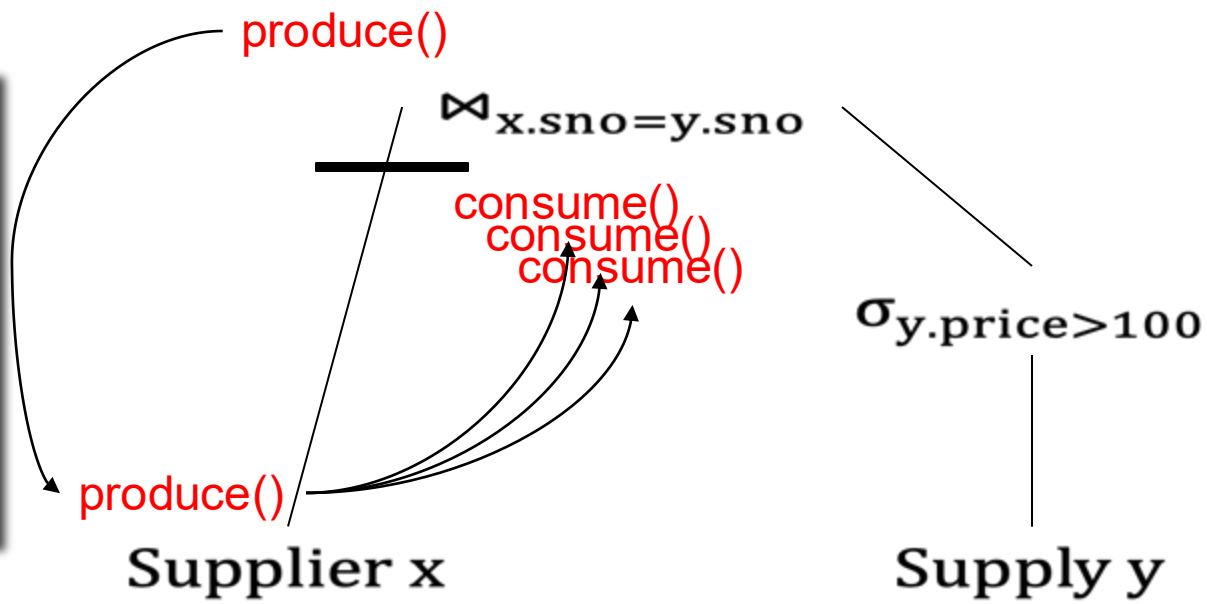
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

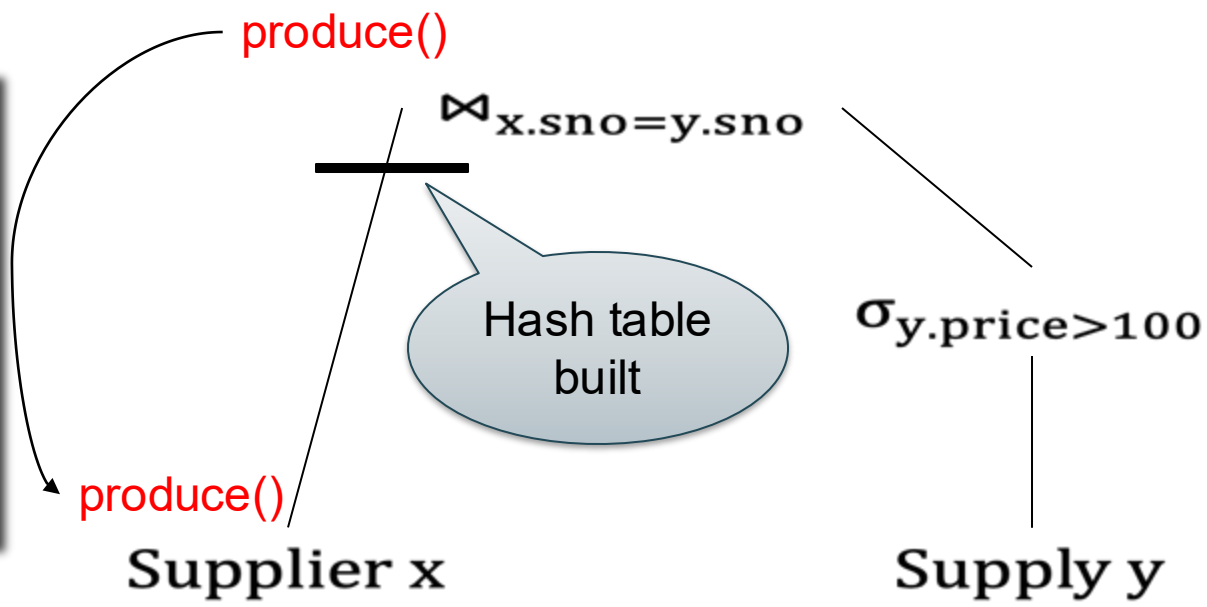
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

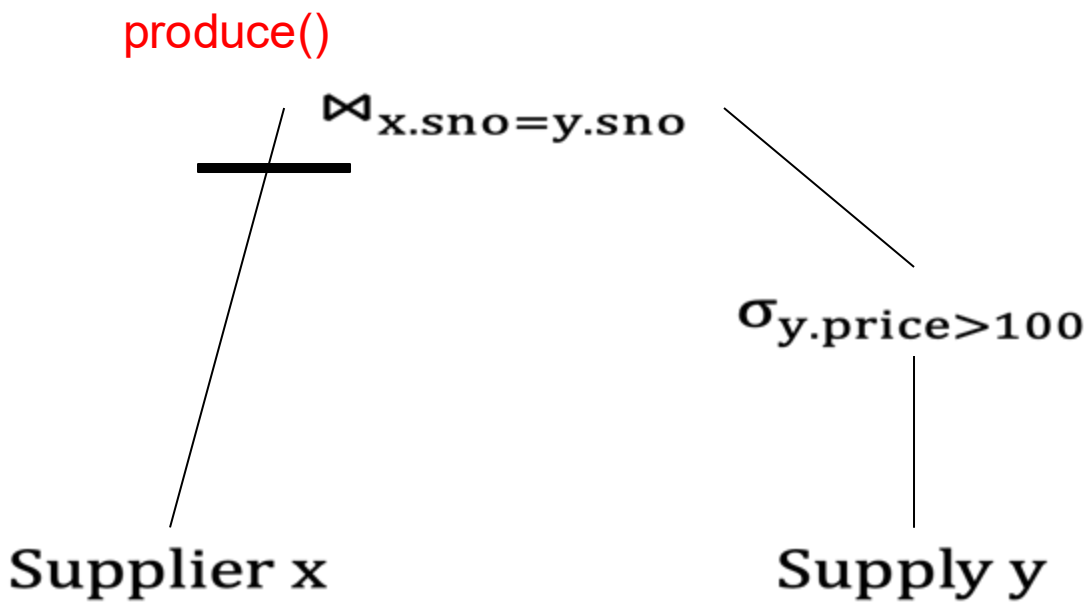
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

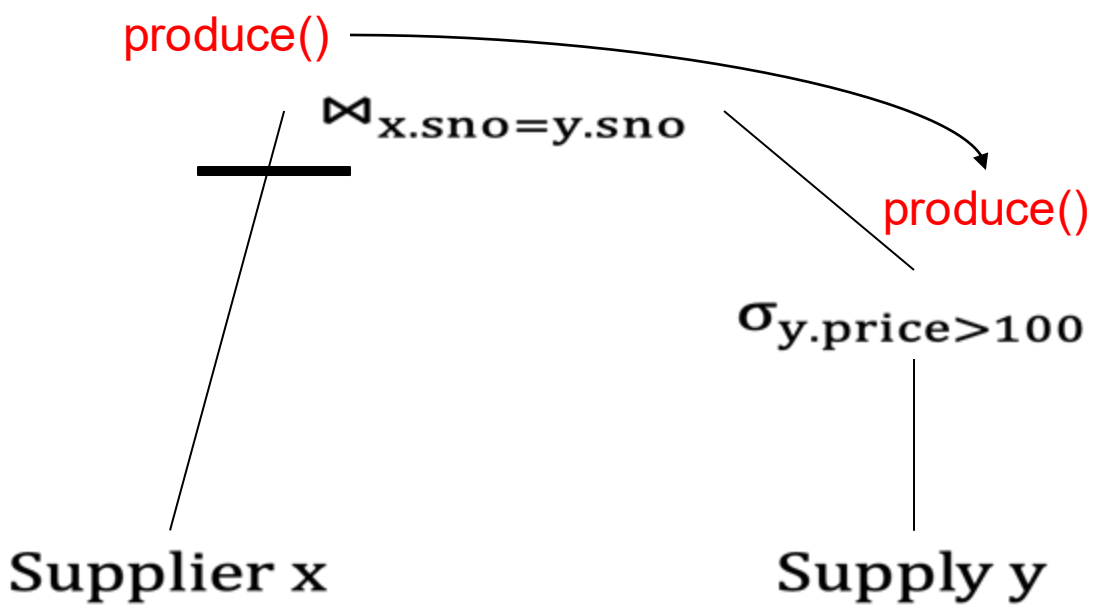
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

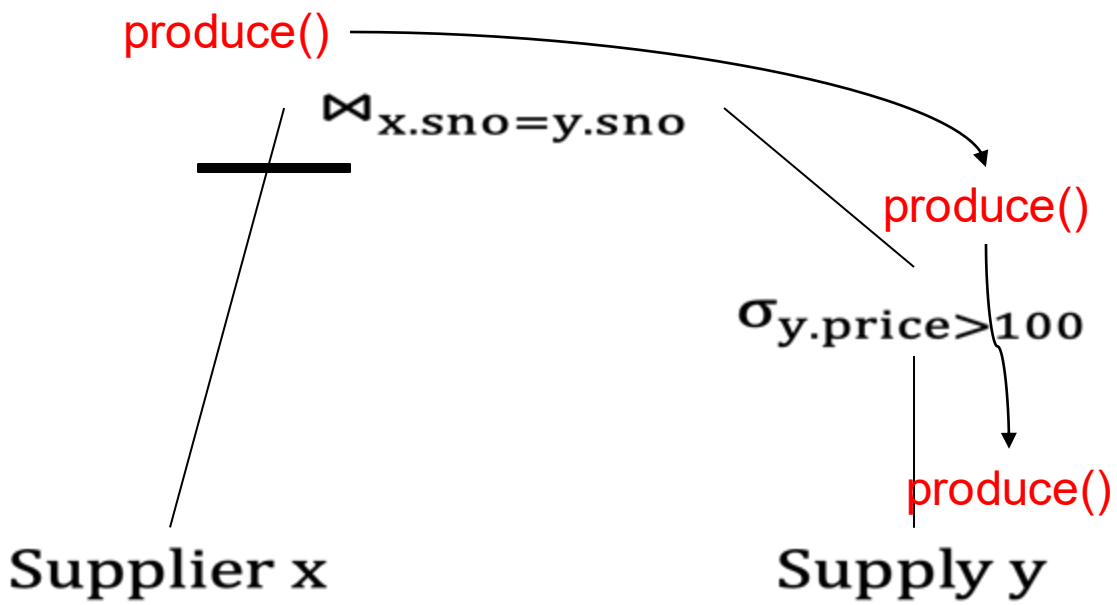
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

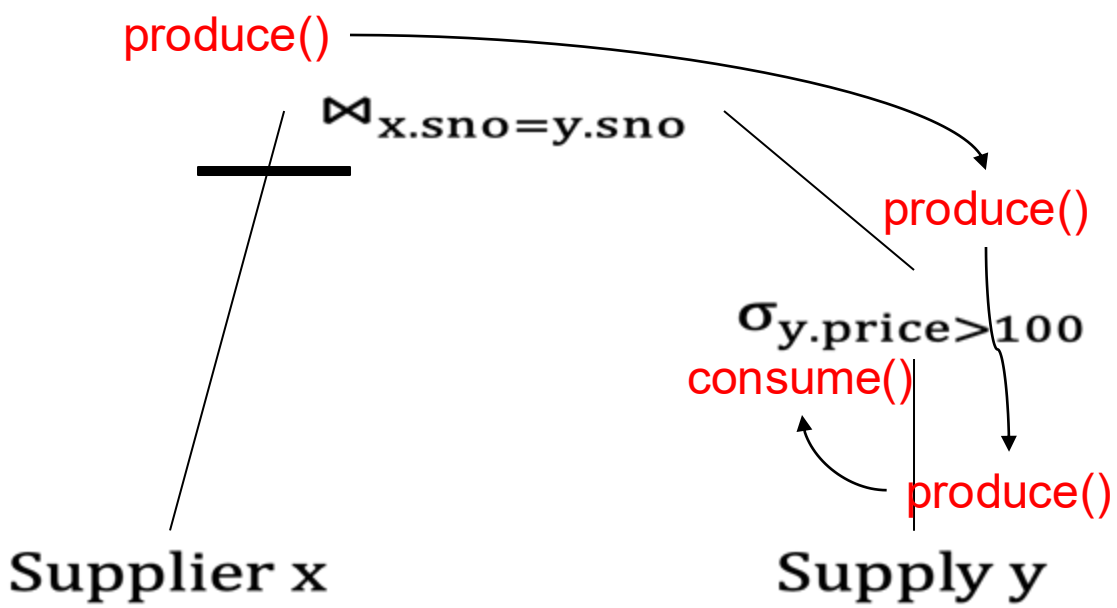
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

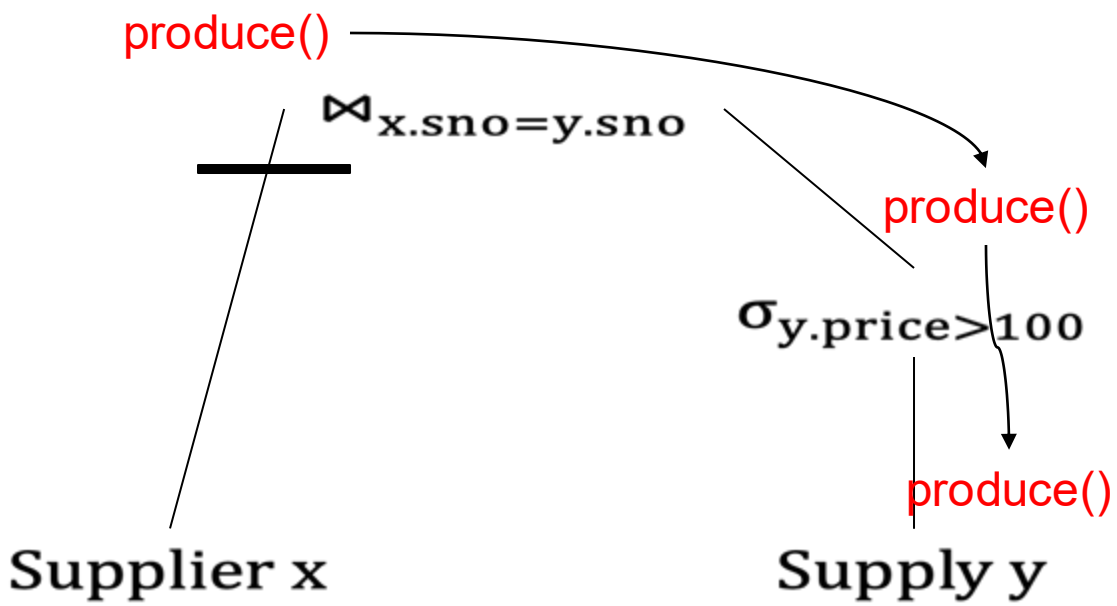
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

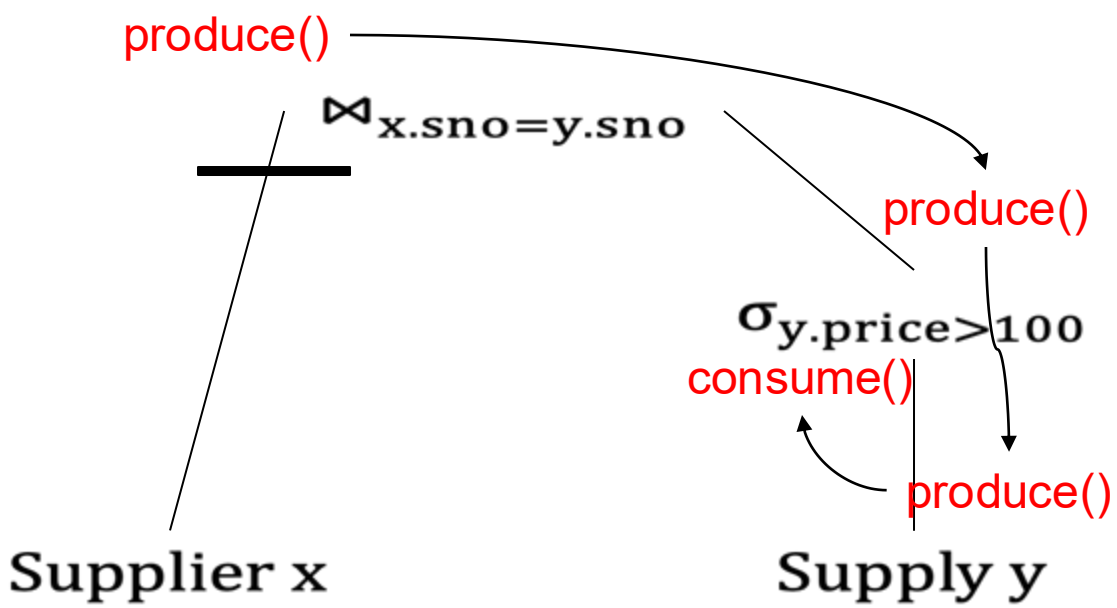
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

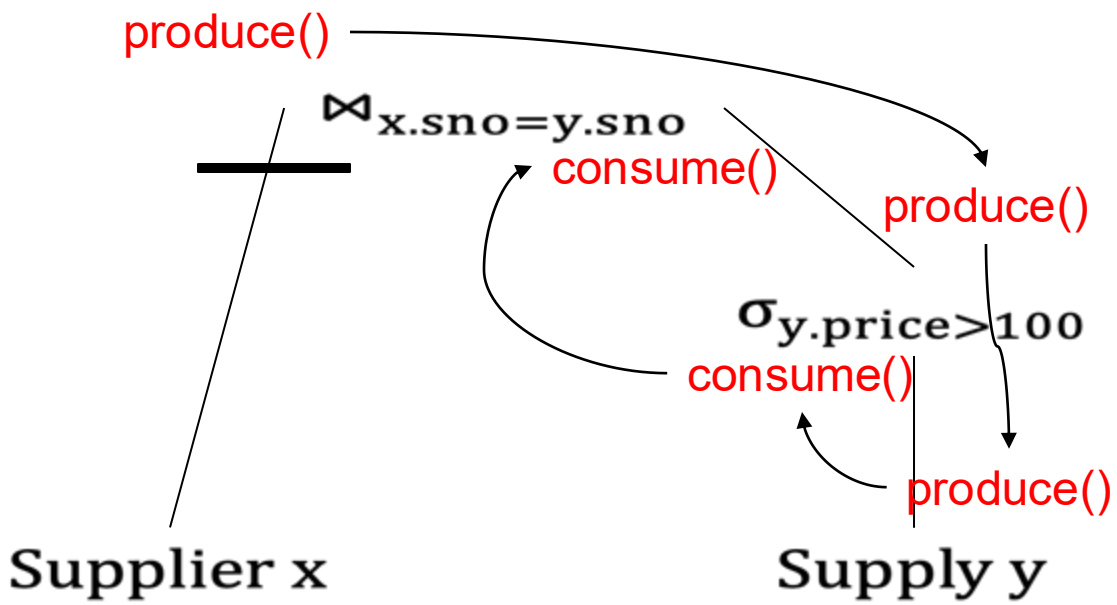
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

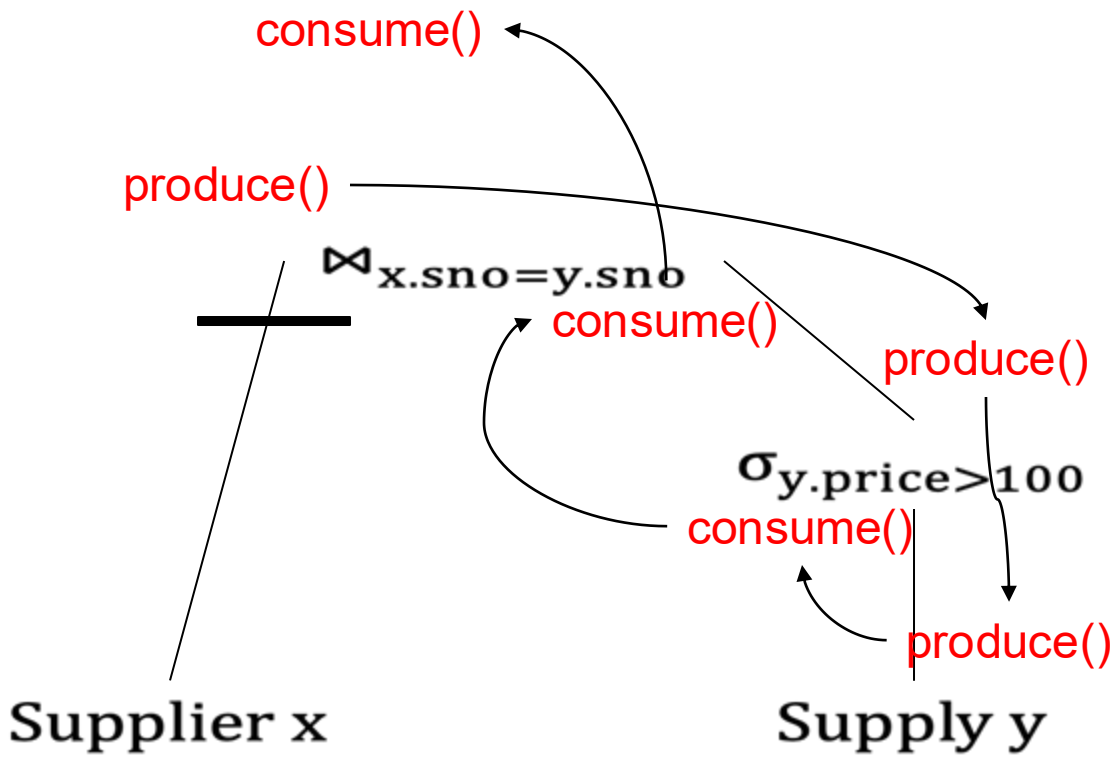
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

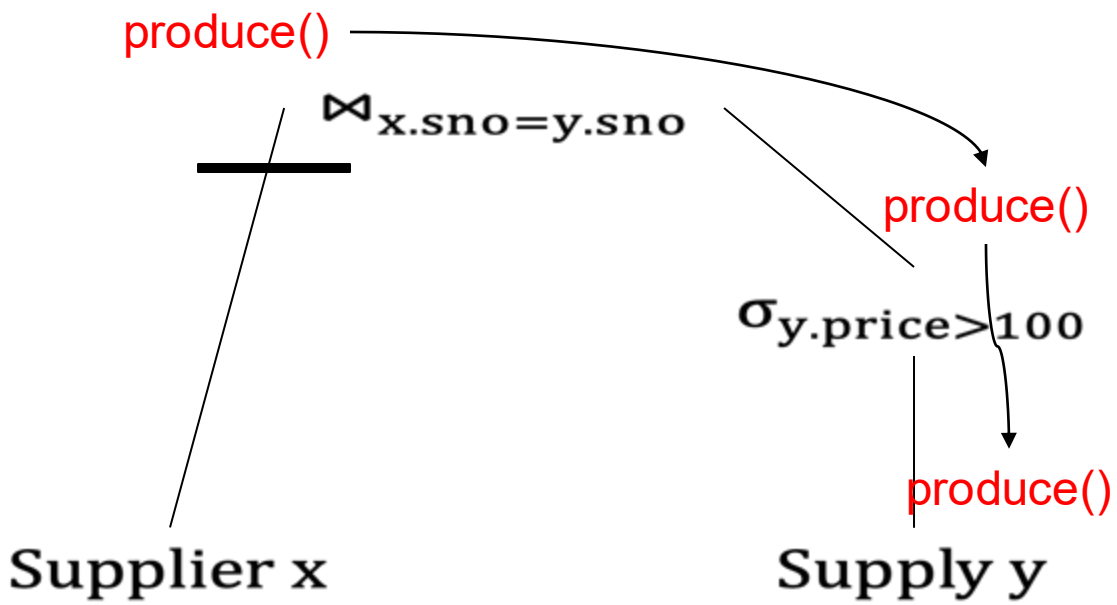
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



Example

”Normal” hash-join

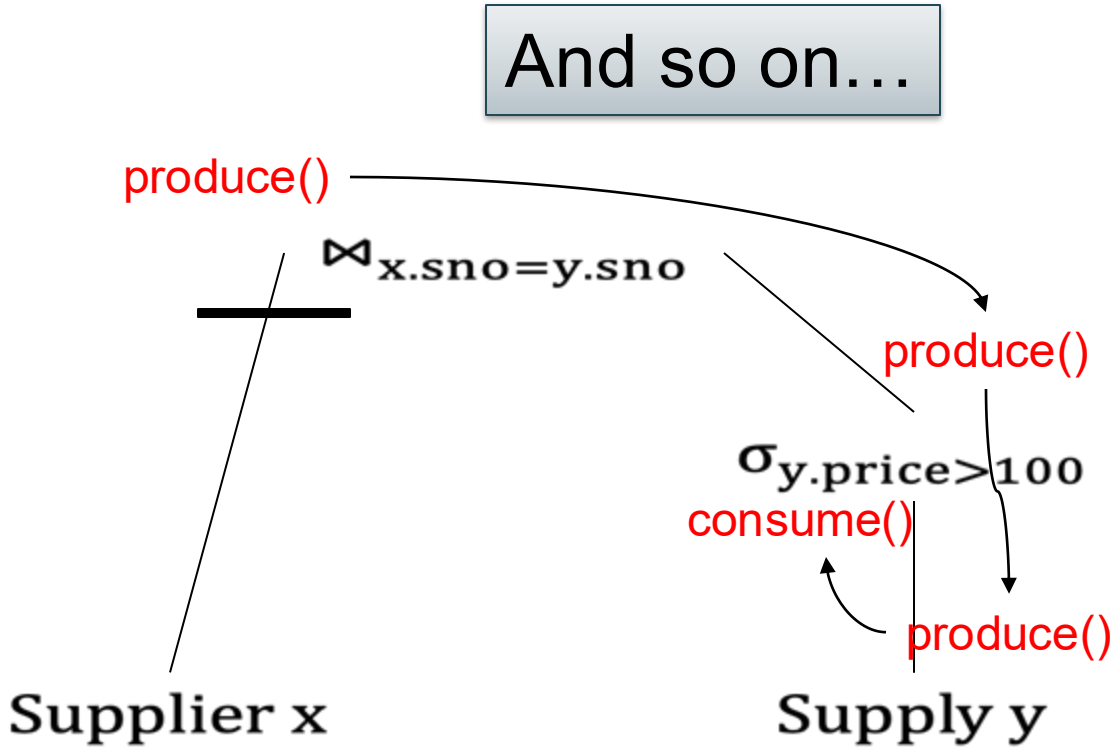
```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



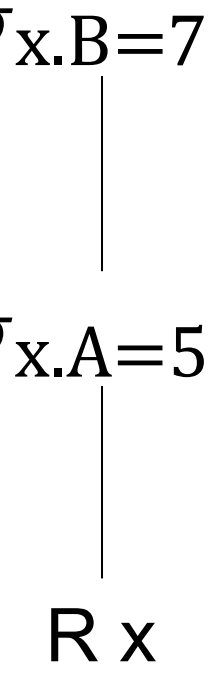
Example

”Normal” hash-join

```
for x in Supplier do
      insert(x.sno, x)

for y in Supply do
      x = find(y.sno);
      output(x,y);
```



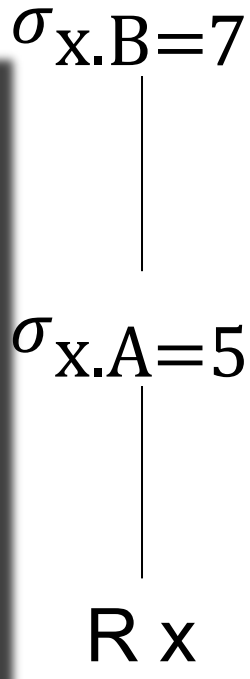
Volcano v.s. Data Driven



Volcano v.s. Data Driven

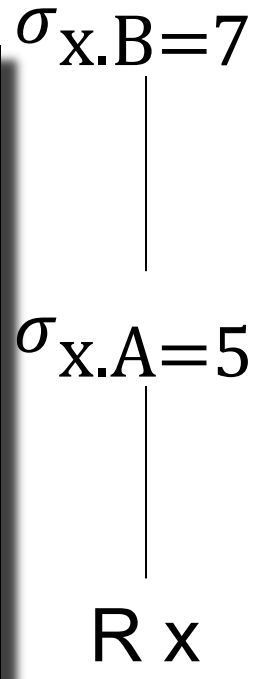
repeat /* $\sigma_B.next()$ */

output x
until x.isNull()



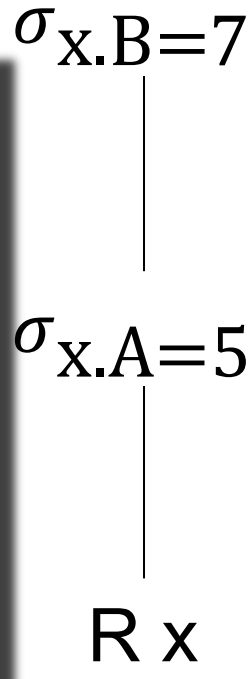
Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
 until x.isNull() or x.B=7
 output x
until x.isNull()
```



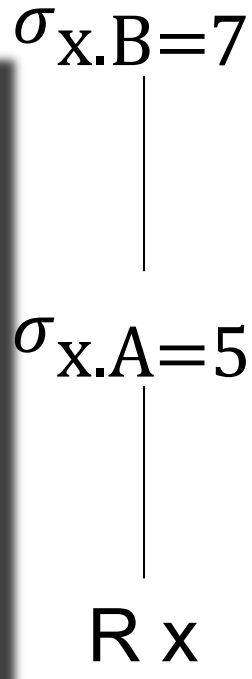
Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//


  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```



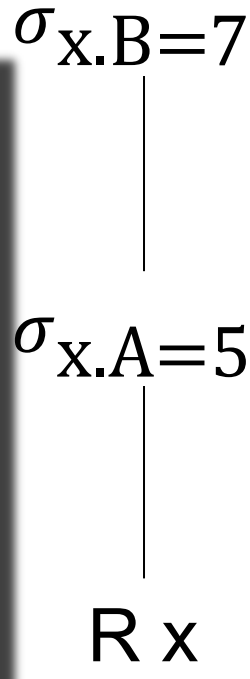
Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```



Volcano v.s. Data Driven

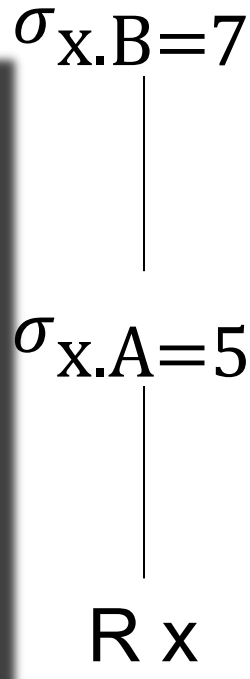
```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R    // R.produce()
```

Volcano v.s. Data Driven

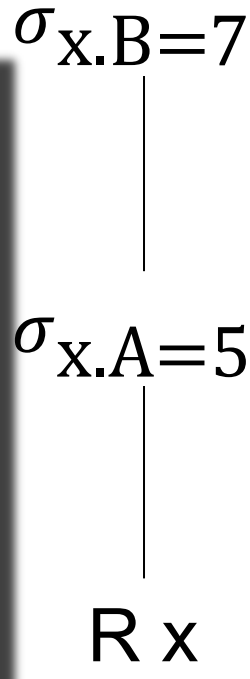
```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R    // R.produce()
  if x.A=5    // σ_A.consume()
```

Volcano v.s. Data Driven

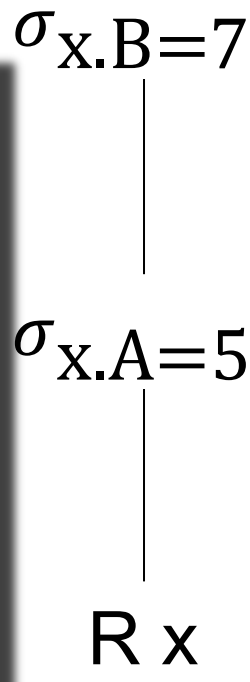
```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R     // R.produce()
  if x.A=5     // σ_A.consume()
    if x.B=7   // σ_B.consume()
```

Volcano v.s. Data Driven

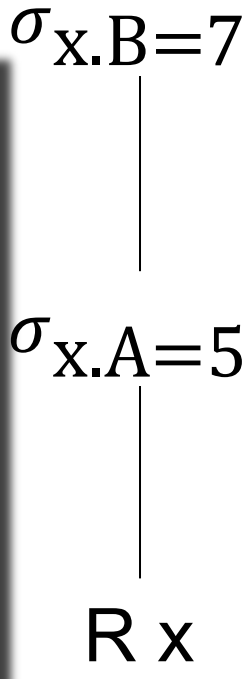
```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R     // R.produce()
  if x.A=5     // σ_A.consume()
    if x.B=7   // σ_B.consume()
      output x
```

Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```



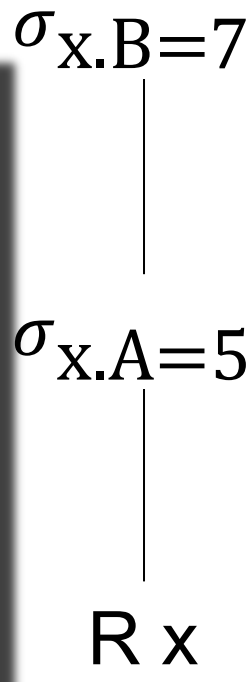

```
for x in R     // R.produce()
  if x.A=5     // σ_A.consume()
    if x.B=7   // σ_B.consume()
      output x
```

Three iterations.

isNull tested repeatedly

Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R     // R.produce()
  if x.A=5     // σ_A.consume()
    if x.B=7   // σ_B.consume()
      output x
```

Three iterations.

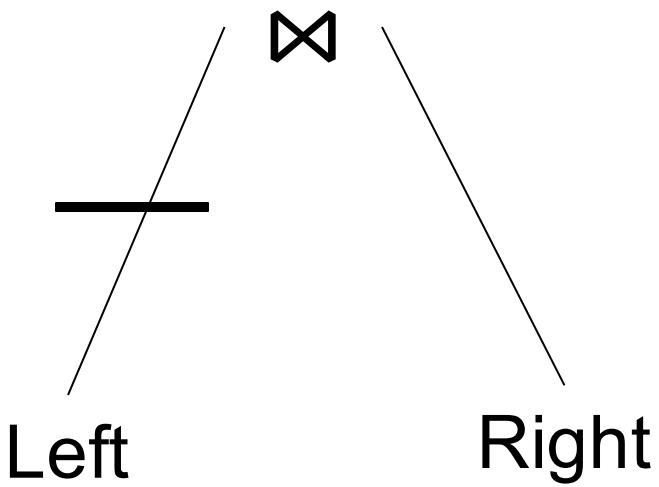
isNull tested repeatedly

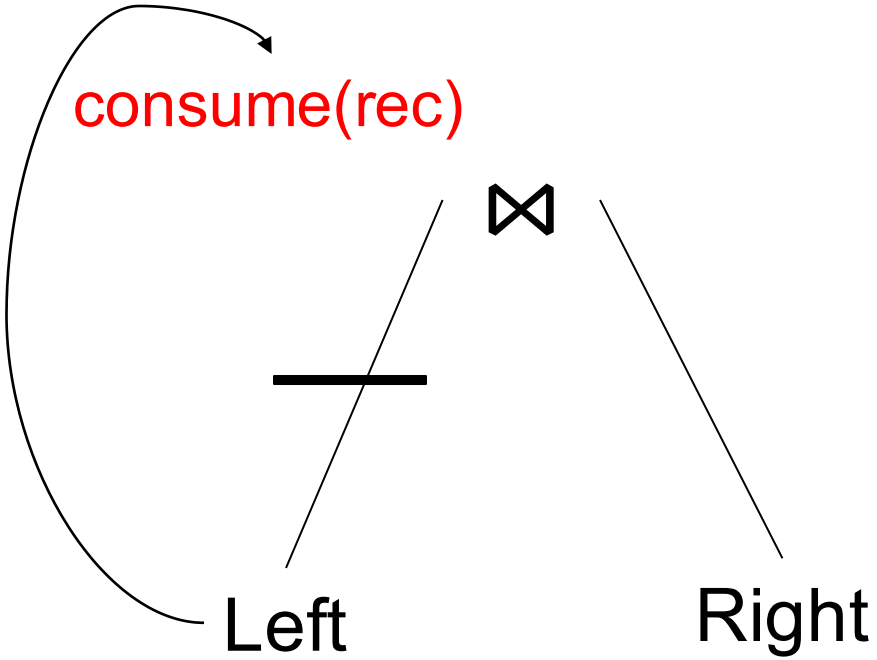
Simpler code. Better instruction cache locality

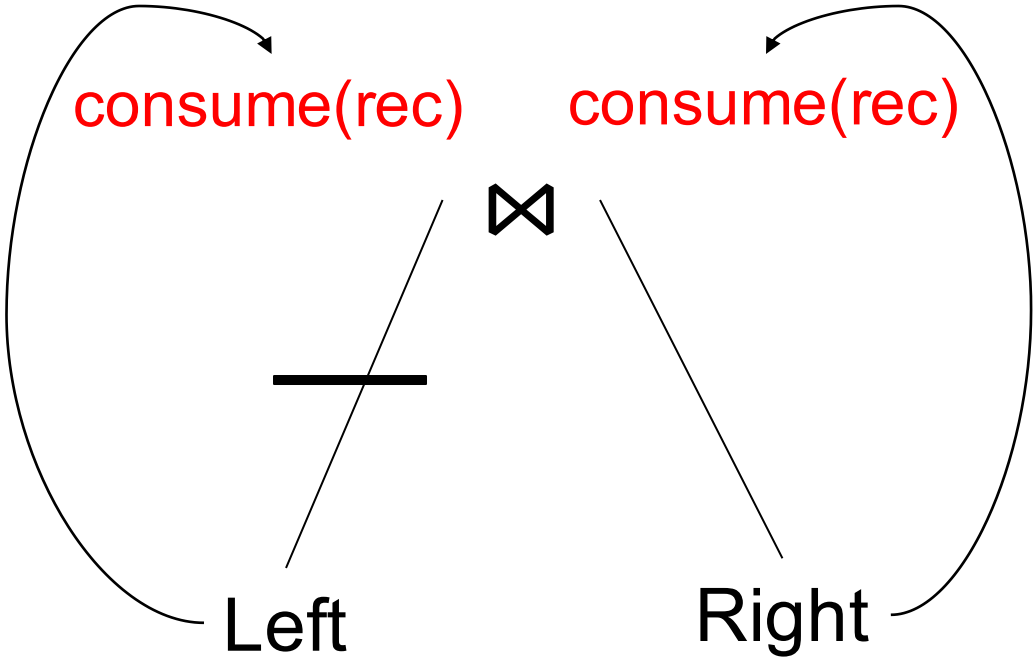
Call-back

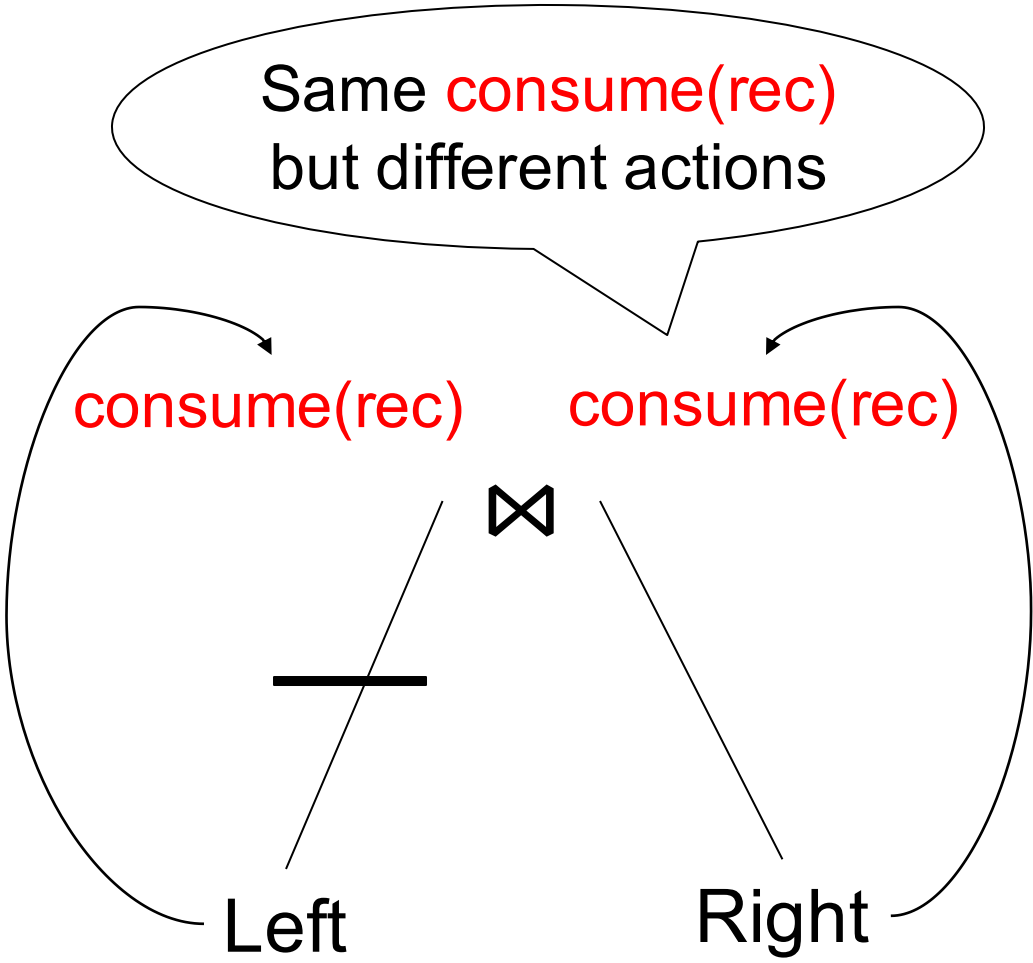
- For any non-commutative operator like hash-join, consume() must treat differently calls from left and right child

- Paper's solution (next)


⋈
Left
Right

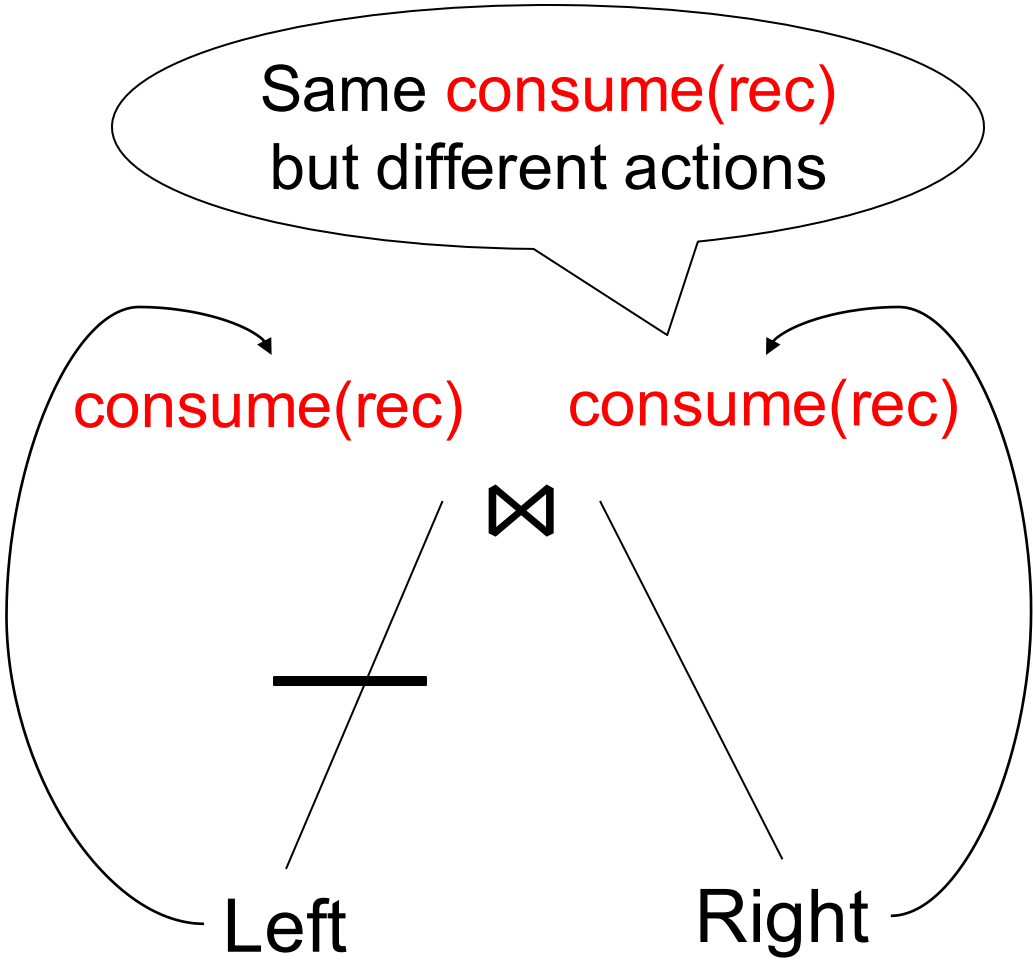

consume(rec)
⋈
Left
Right


consume(rec)
consume(rec)
⋈
Left
Right


Same consume(rec)
but different actions
consume(rec)
consume(rec)
⋈
Left
Right

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {



  def open() = {                    // Step 1


  }
  def produce() = {


  }
  def consume(rec: Record) = {









  }
}
```

(a)



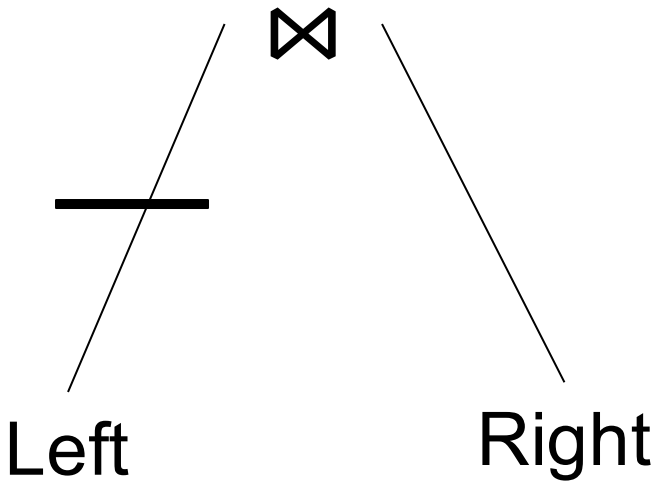
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {



  def open() = {                    // Step 1

  }
  def produce() = {


  }
  def consume(rec: Record) = {




  }
}
```

(a)



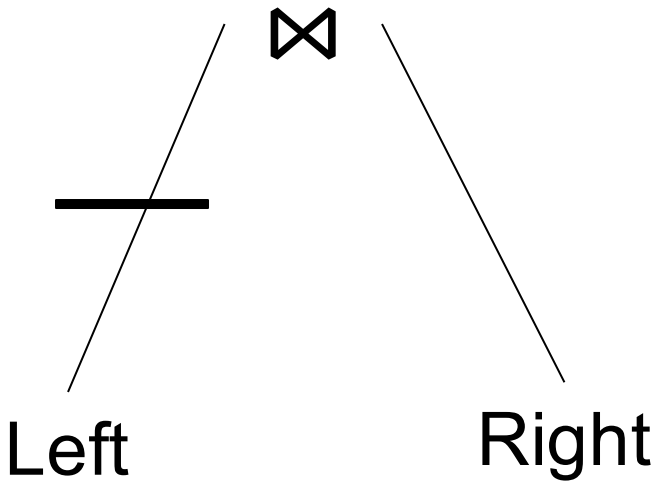
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {



  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {


  }
  def consume(rec: Record) = {









  }
}
```

(a)



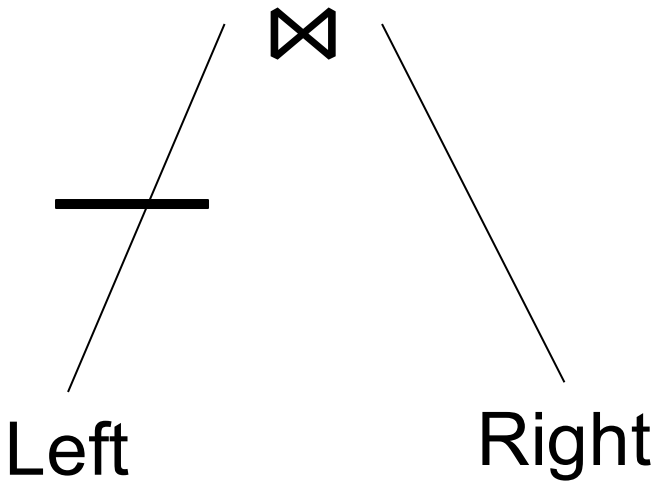
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()

  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {



  }
  def consume(rec: Record) = {









  }
}
```

(a)



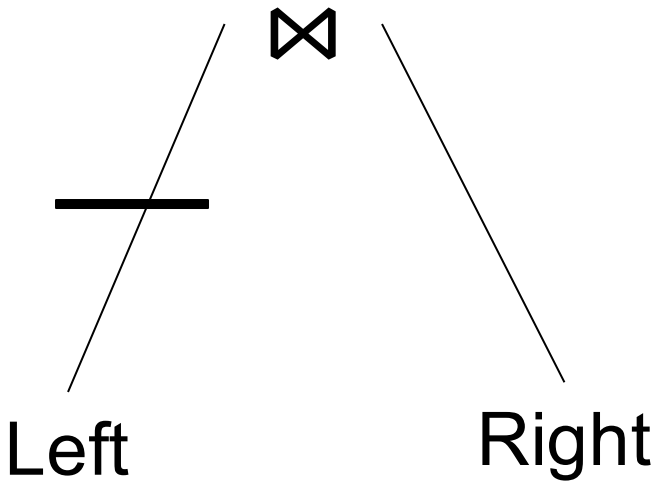
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {



  }
  def consume(rec: Record) = {









  }
}
```

(a)



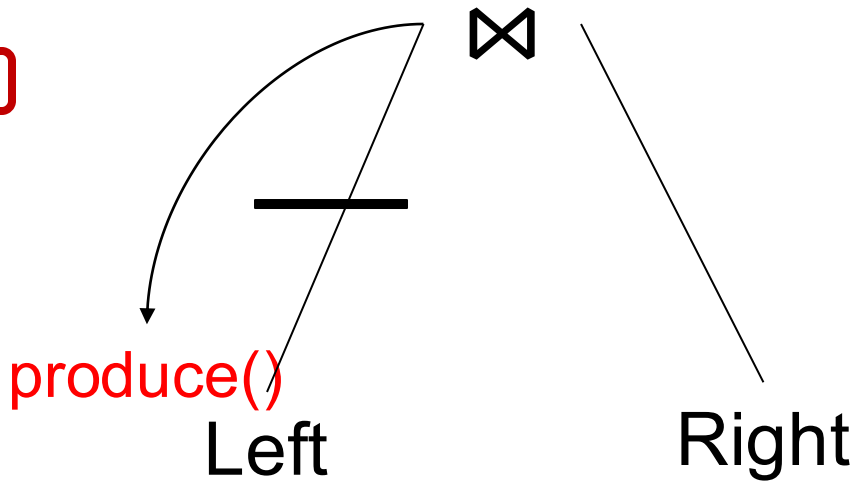
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {




  }
}
```

(a)



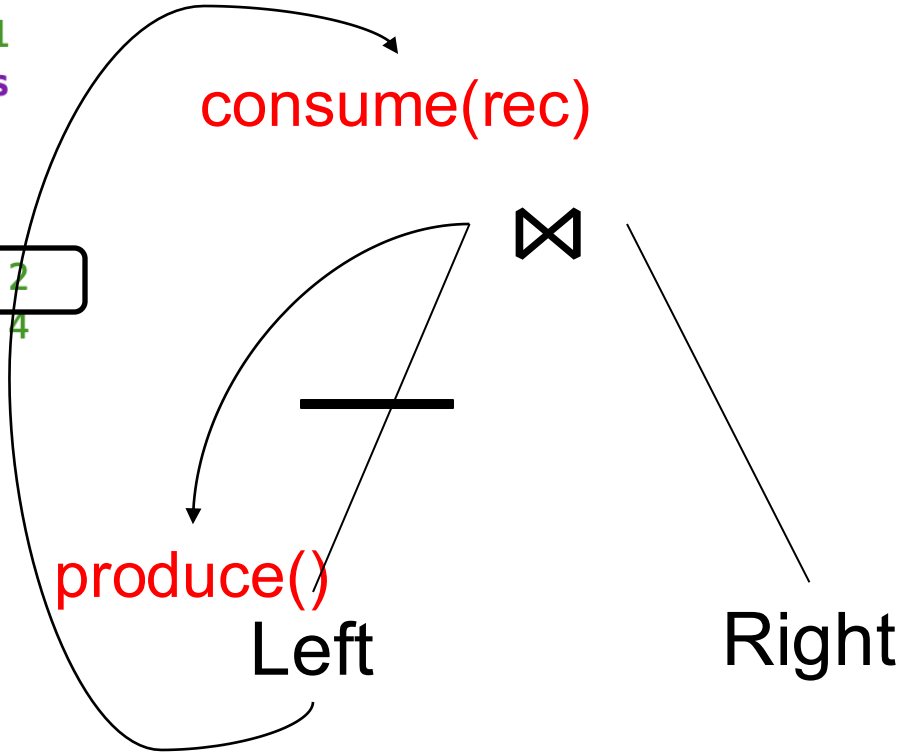
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {




  }
}
```

(a)



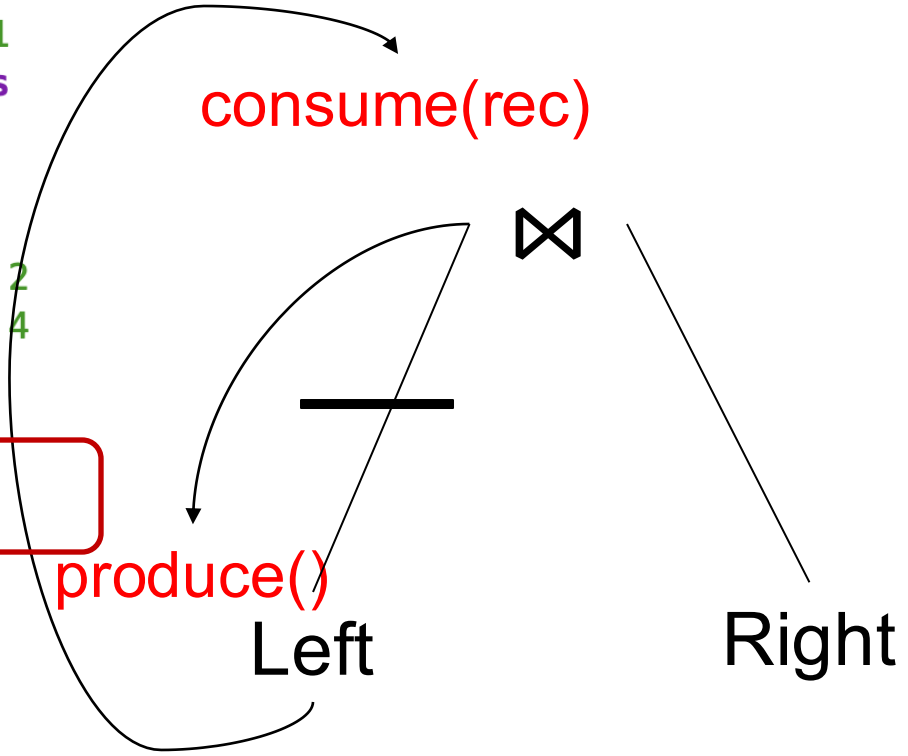
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)



  }
}
```

(a)



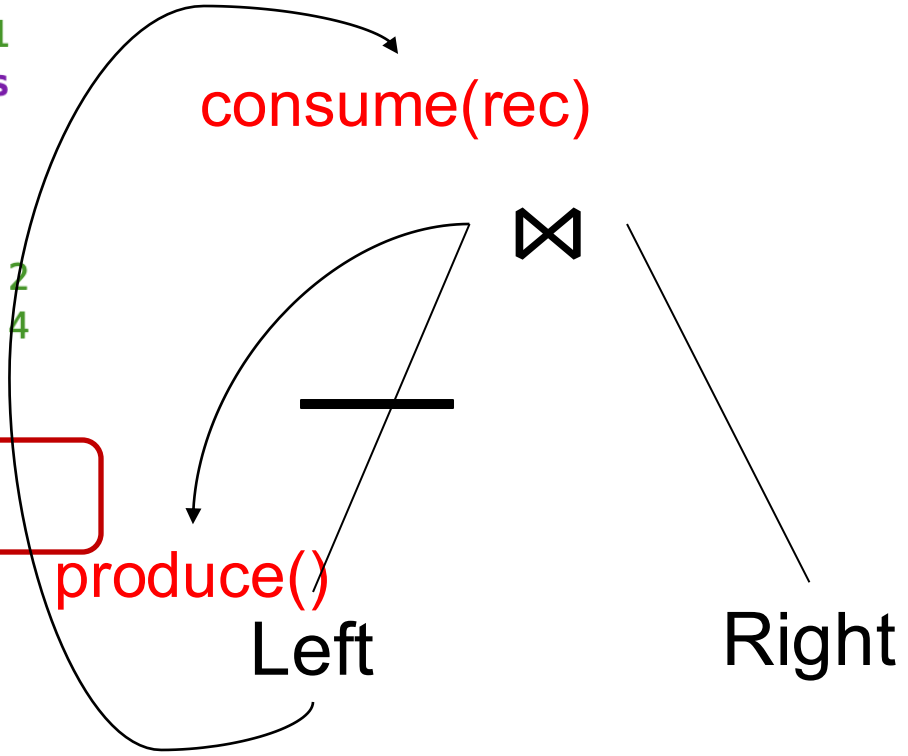
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)             // Step 3
      hm += (lkey(rec), rec)
    else                    // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)



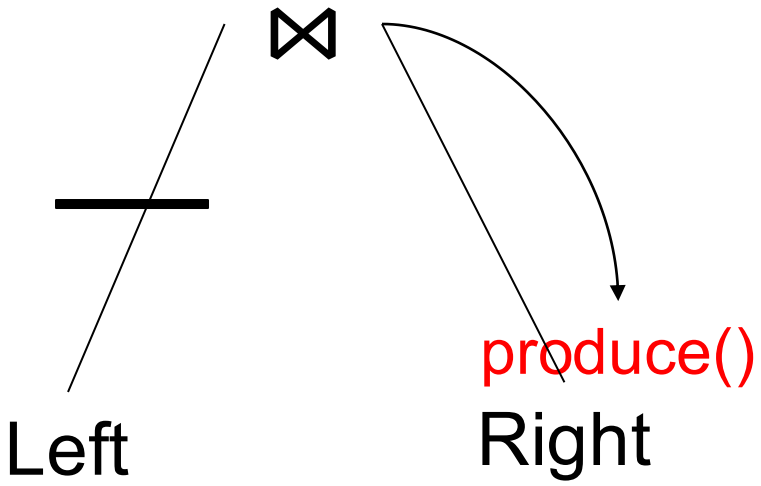
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)             // Step 3
      hm += (lkey(rec), rec)
    else                    // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)



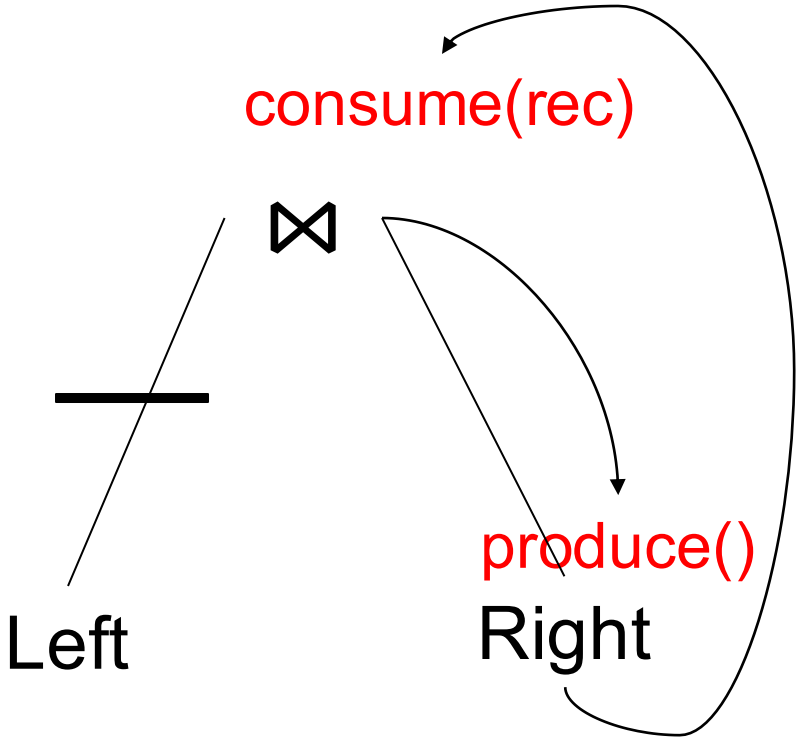
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)             // Step 3
      hm += (lkey(rec), rec)
    else                    // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)



(b)

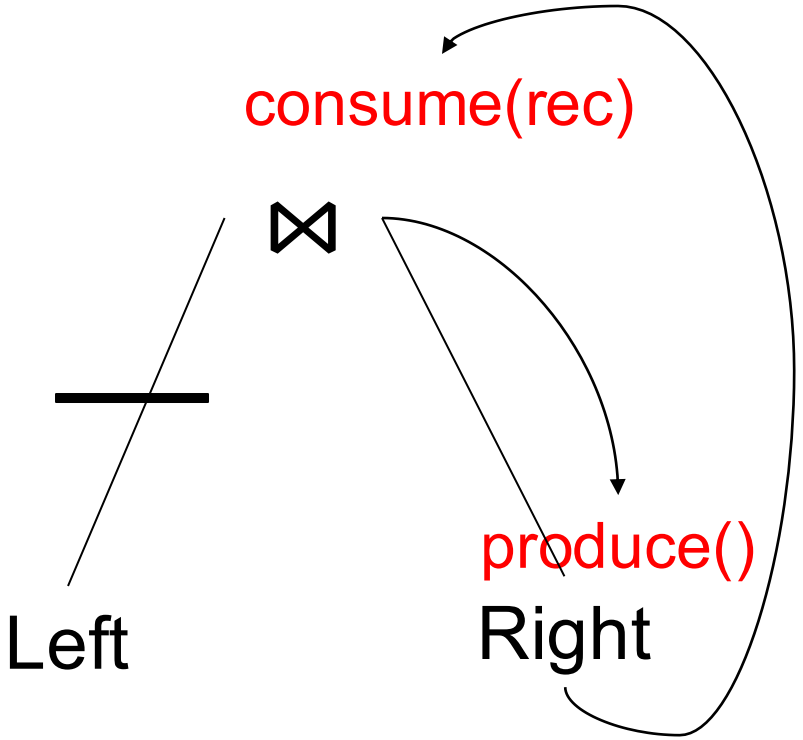
Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)             // Step 3
      hm += (lkey(rec), rec)
    else                    // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

Q: There is no for loop!
Where is the for loop?



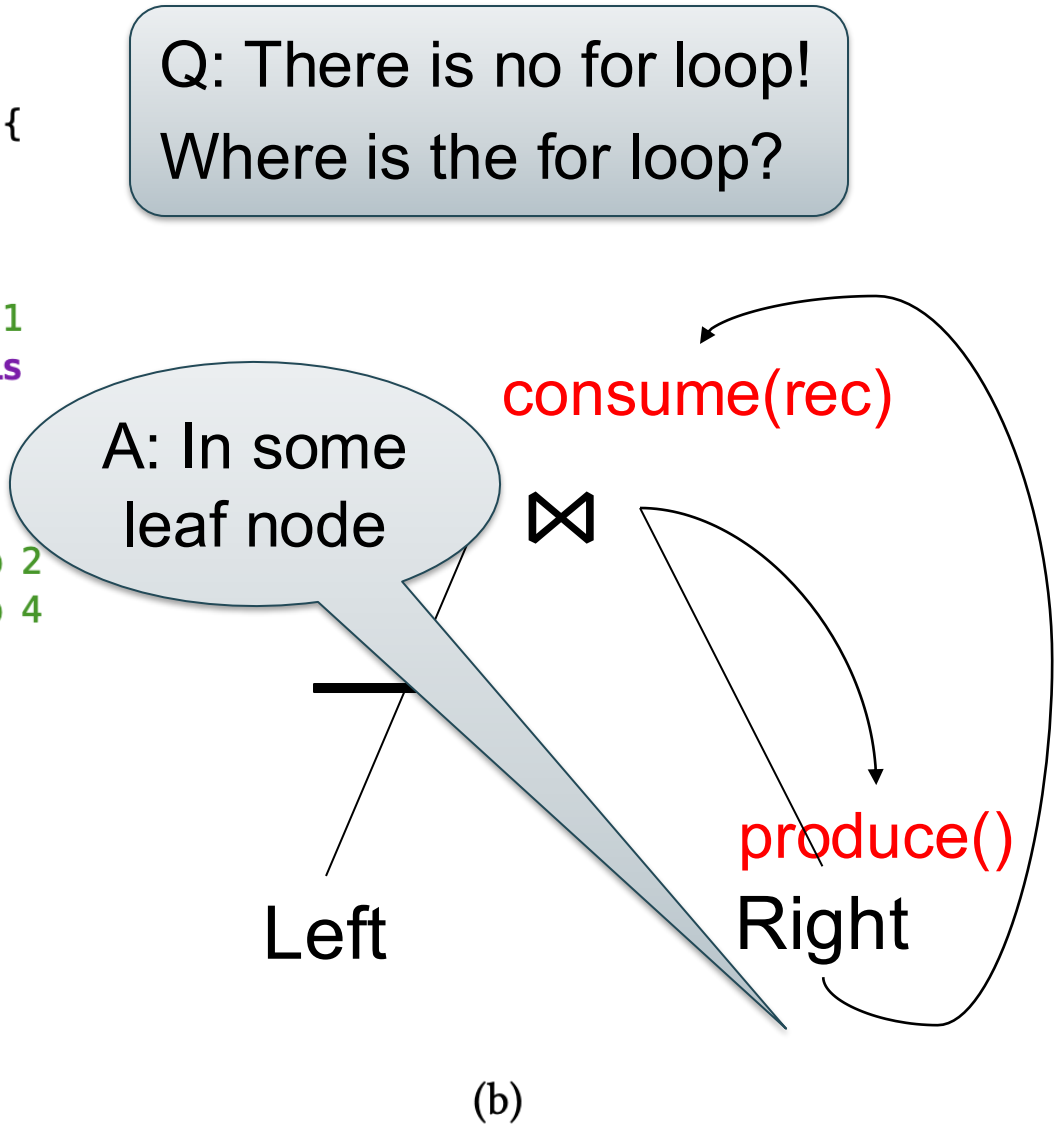
(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)             // Step 3
      hm += (lkey(rec), rec)
    else                    // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)



(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a) (b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {

      }
    }
  }
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {

    }
  }
}
```

cb = call-back function

This is consume(rec)

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()

      }
    }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }


    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }



    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

[How to Architect a Query Compiler]

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

cb is the call-back function of the parent

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Today: data in memory, cost dominated by CPU

- Compilation has significant advantage

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Today: data in memory, cost dominated by CPU

- Compilation has significant advantage
- Alternative: vectorized processing
  - Next() returns batch of 1000 tuples
  - Interpreter as good as compiled code