CSE544
Data Management

Lecture 8

Query Execution – Part 2

Outline

- Steps involved in processing a query
- Main Memory Operators
- Query execution
- External Memory Operators

Query Execution

Interpret RA

- Pros/cons?

- dominant 1980-2010
  - Why?

Compile RA

- Pros/Cons?

- Renewed interest
  - Why?

Query Execution

Interpret RA

- Pros/cons?
  - Portable, simple
  - Slow
- dominant 1980-2010
  - Why?

Compile RA

- Pros/Cons?
  - Faster
  - Architecture specific
- Renewed interest
  - Why?

Query Execution

Interpret RA

- Pros/cons?
  - Portable, simple
  - Slow
- dominant 1980-2010
  - Why?
  - I/O cost dominates

Compile RA

- Pros/Cons?
  - Faster
  - Architecture specific
- Renewed interest
  - Why?
  - Large buffer pool

Operator Interface

Volcano model:

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

Operator Interface

Volcano model:

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

Data-driven model:

- open(),produce(), consume(),close()
- Push model
- Introduced by Thomas Neumann in Hyper (at TU Munich), later acquired by Tableau
- Reading for Wednesday

Key Takeaway

- Compiled/interpreted & Volcano/data-driven are somewhat independent dimensions
  - We discuss the volcano/data-driven models
- Paper uses Futamura's project to explain the *compiled code* of each model
  - Less important for databases, won't discuss much

Recap: Volcano Model

Each operator exports three methods:

- Open()
- Next()
- Close()

```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Recap: Hash Join

Supply $\bowtie_{sid=sid}$ Supplier

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

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

Build phase

Probe phase

`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Volcano Model

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

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

open()

(On the fly) π_{sname}

(On the fly) $\sigma_{scity= \text{'Seattle'} \text{ and } sstate= \text{'WA'} \text{ and } pno=2}$

(Hash Join) $\bowtie_{sid = sid}$

Supply (File scan)

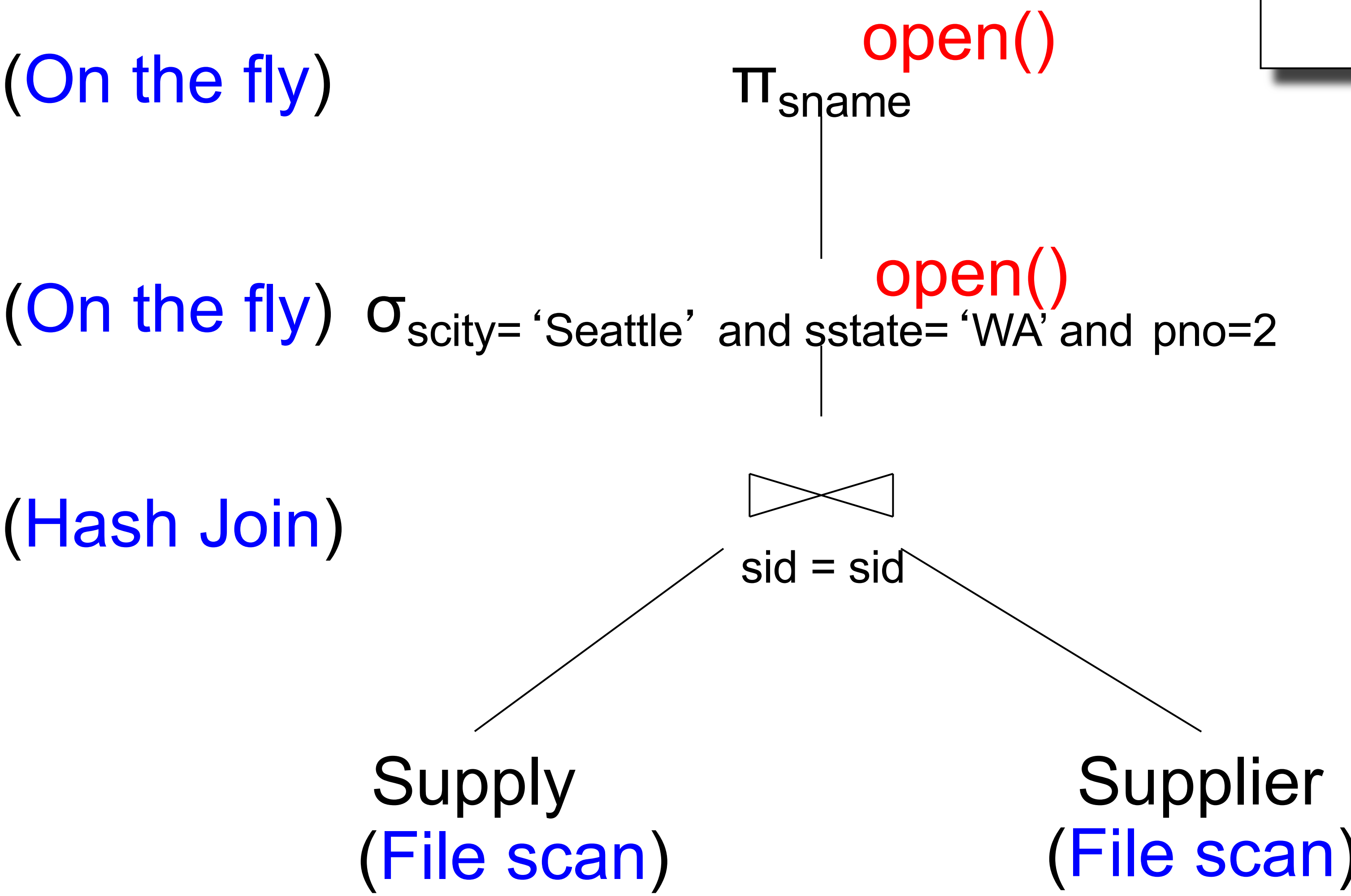
Supplier (File scan)

`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Volcano Model

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

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

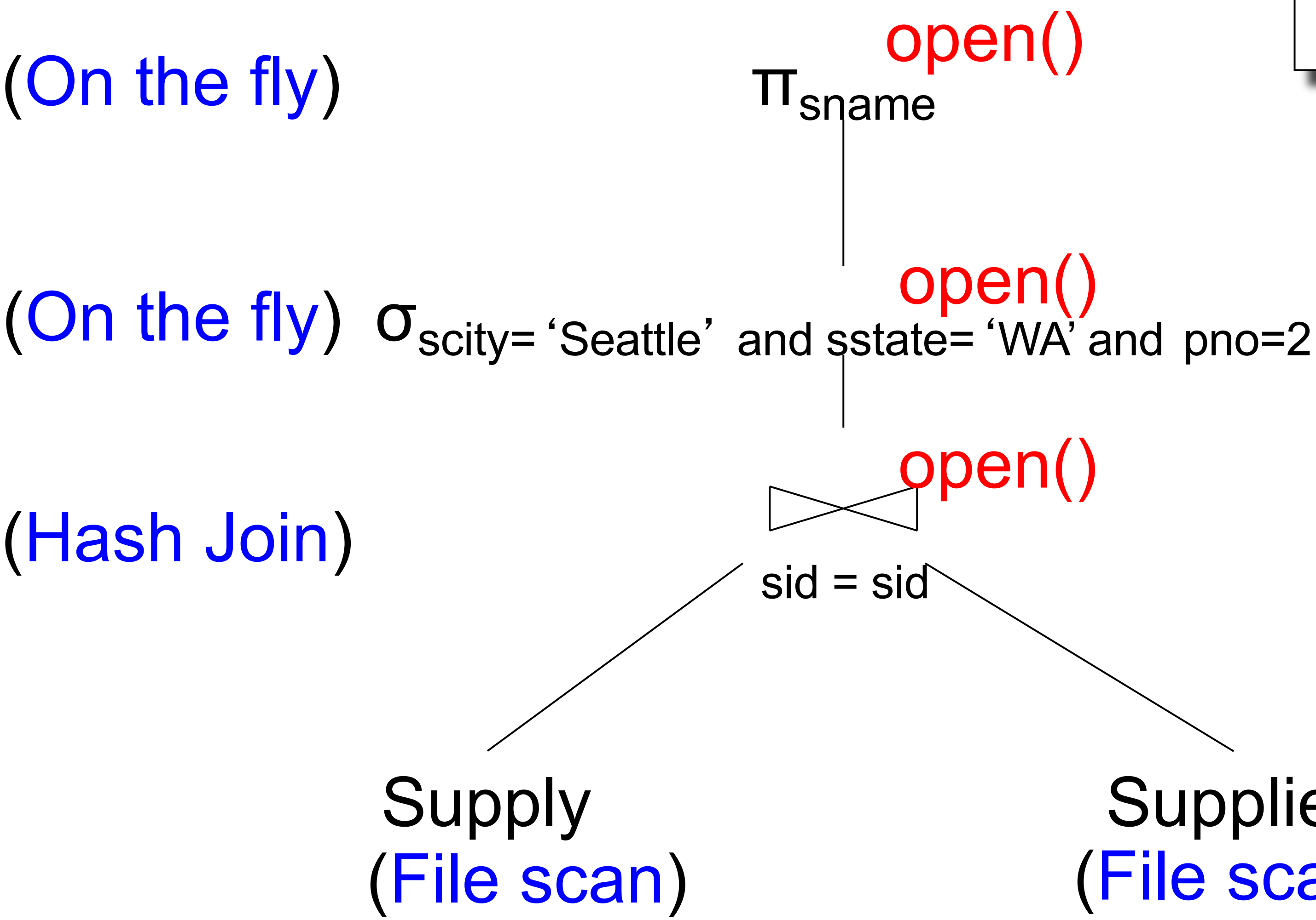


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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



```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Volcano Model

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

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

(On the fly) π_{sname} open()

(On the fly) $\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$ open()

(Hash Join) $\bowtie_{sid = sid}$ open()

Supply (File scan)

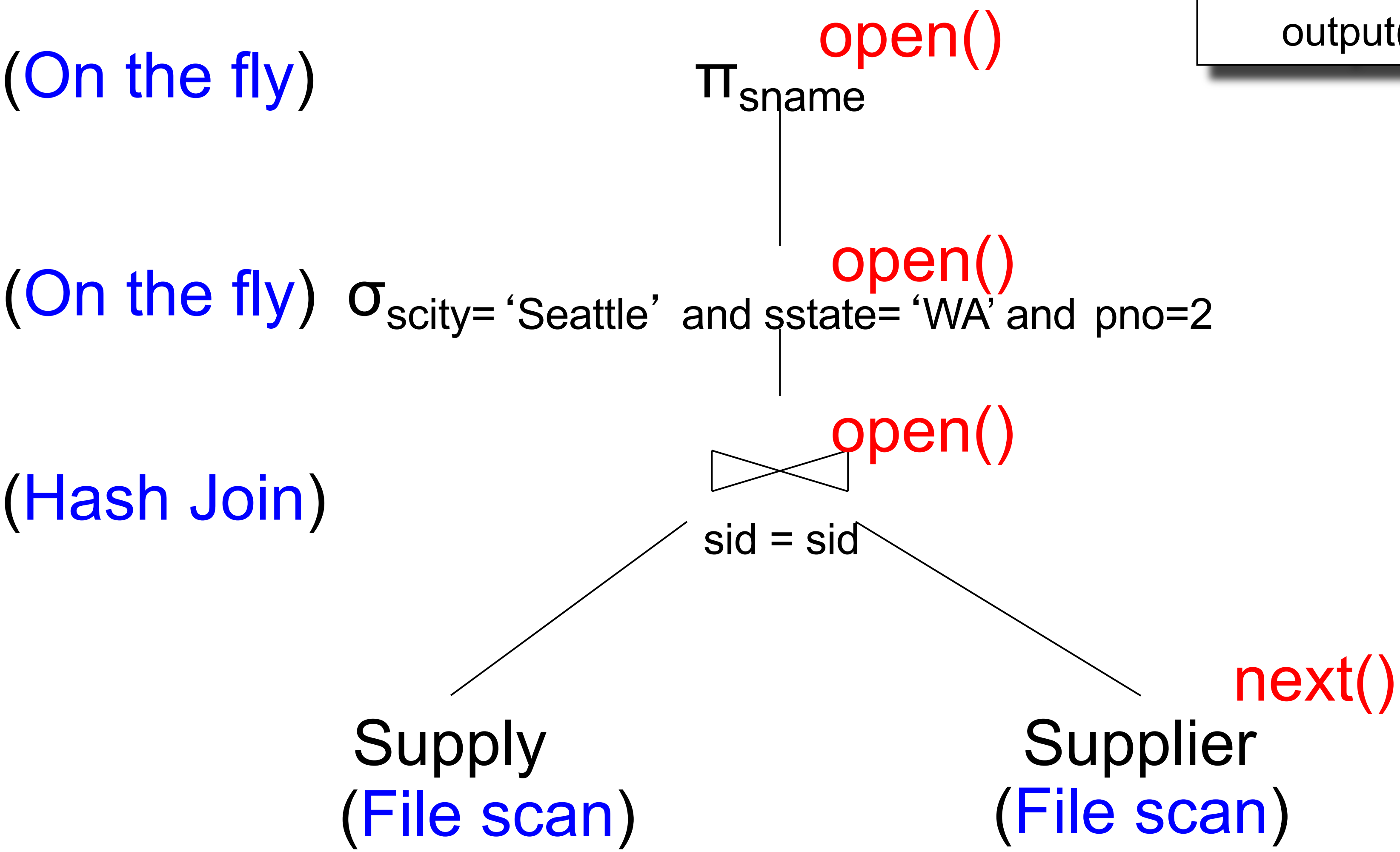
Supplier (File scan) open()

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

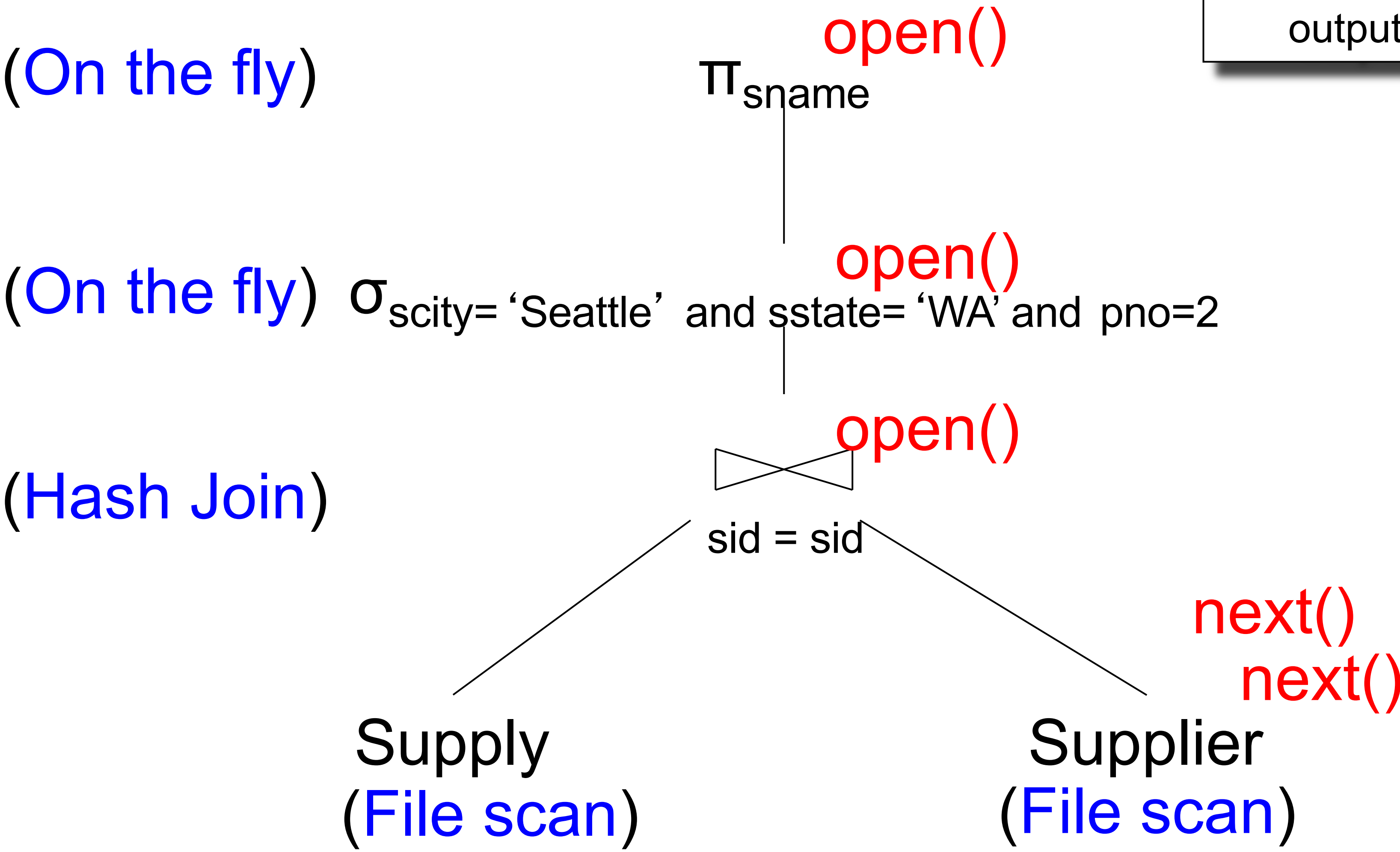


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

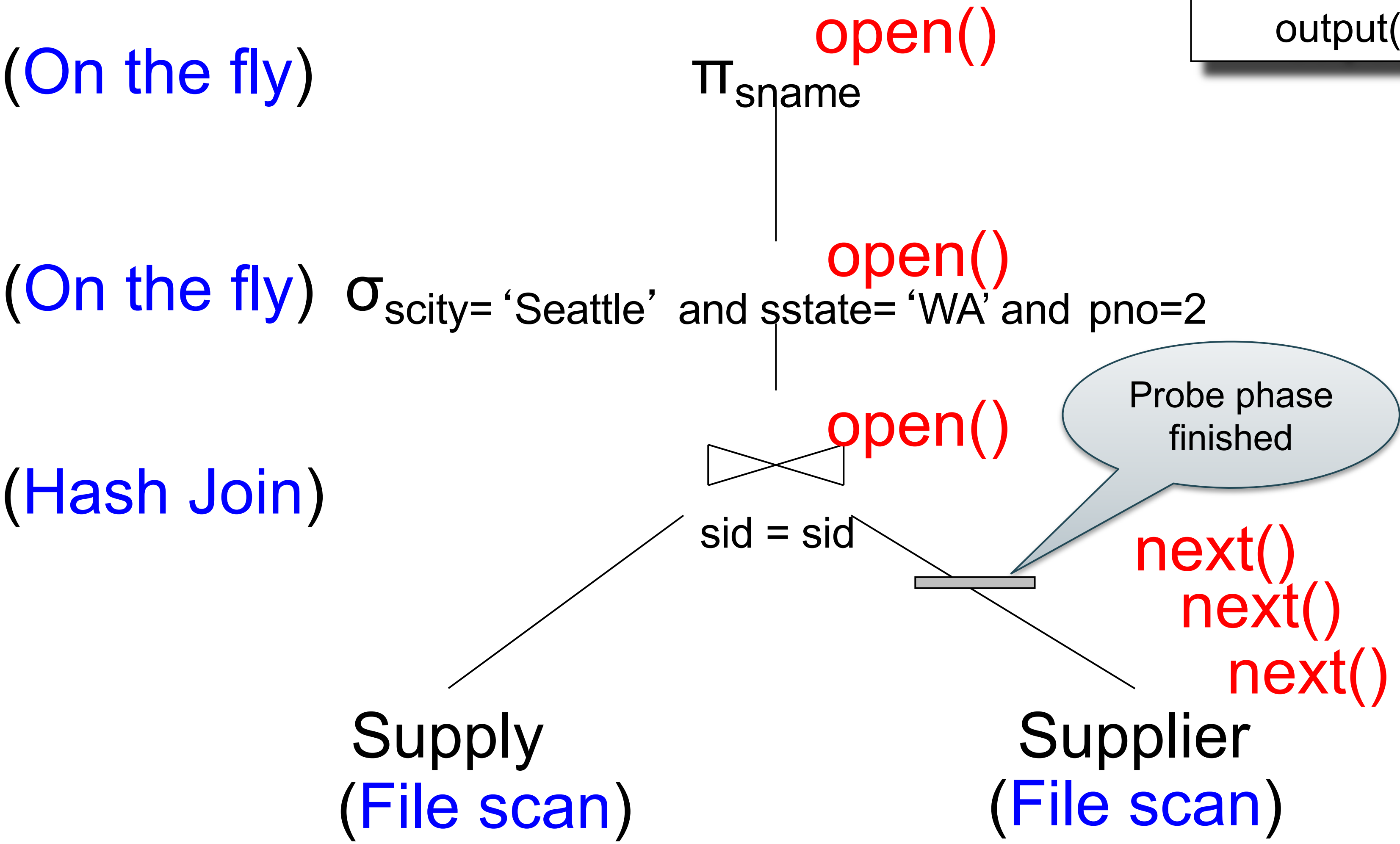


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

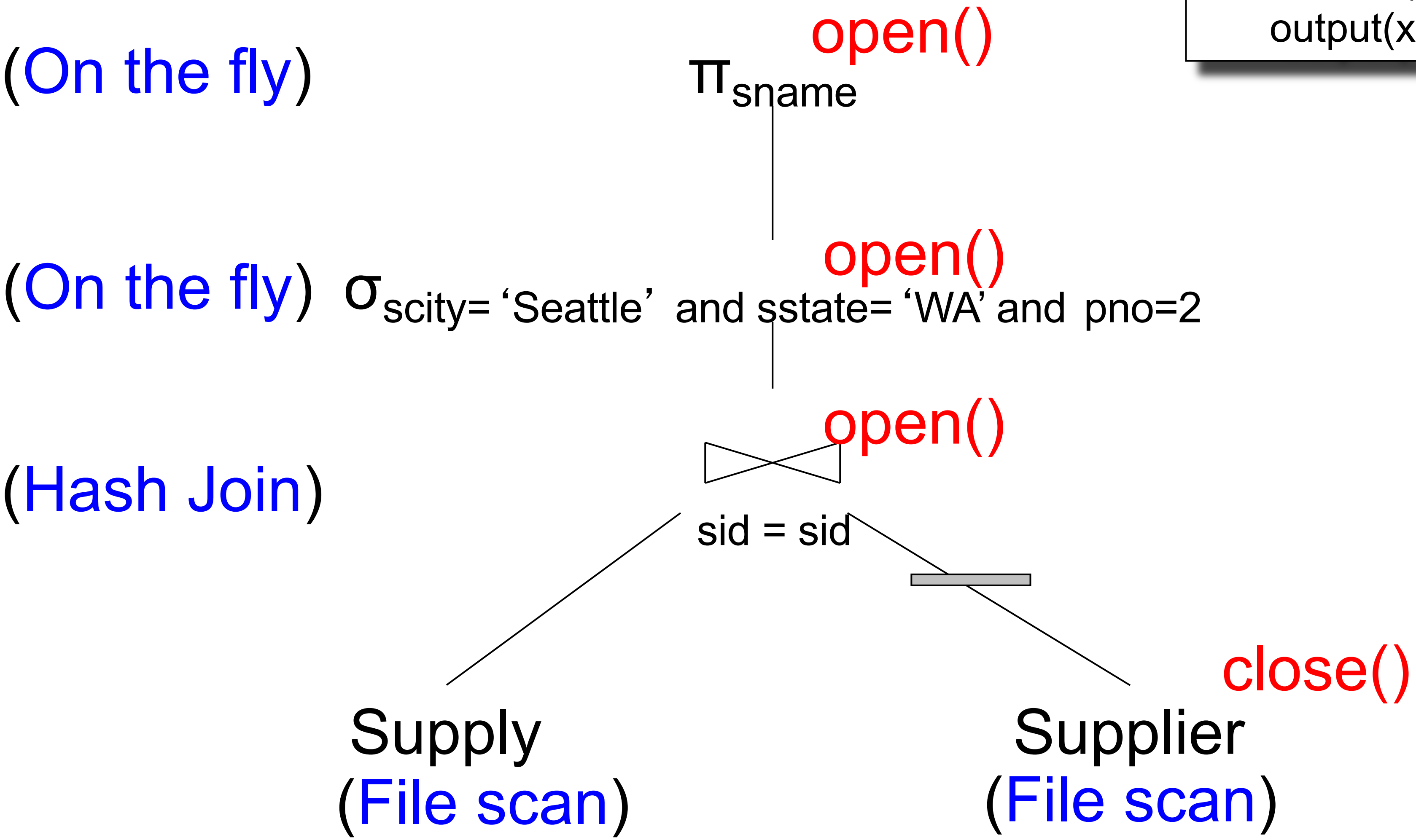


`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Volcano Model

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

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

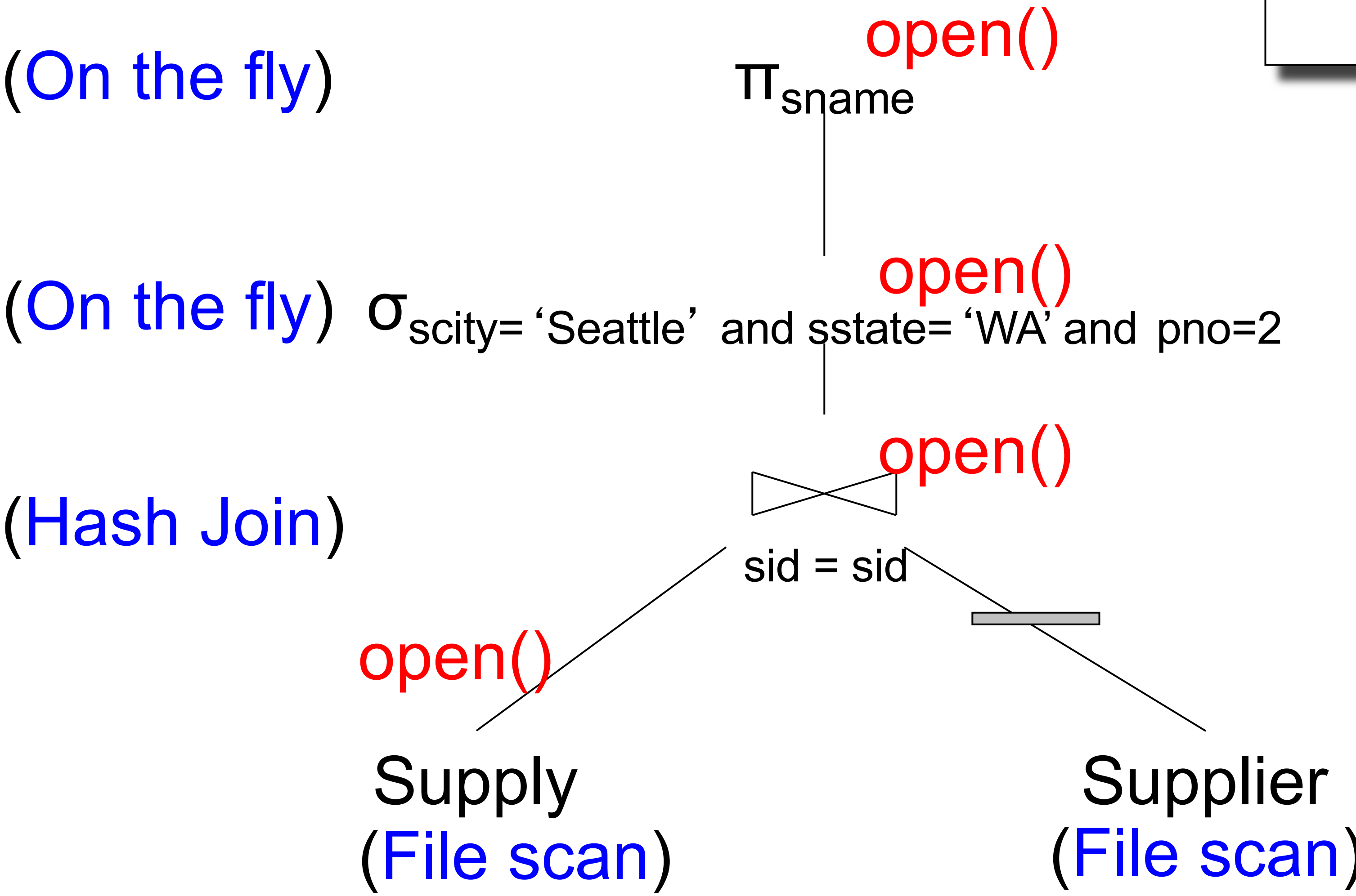


Supplier(<u>sid</u>, sname, scity, sstate)
Supply(<u>sid, pno</u>, quantity)

Volcano Model

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

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

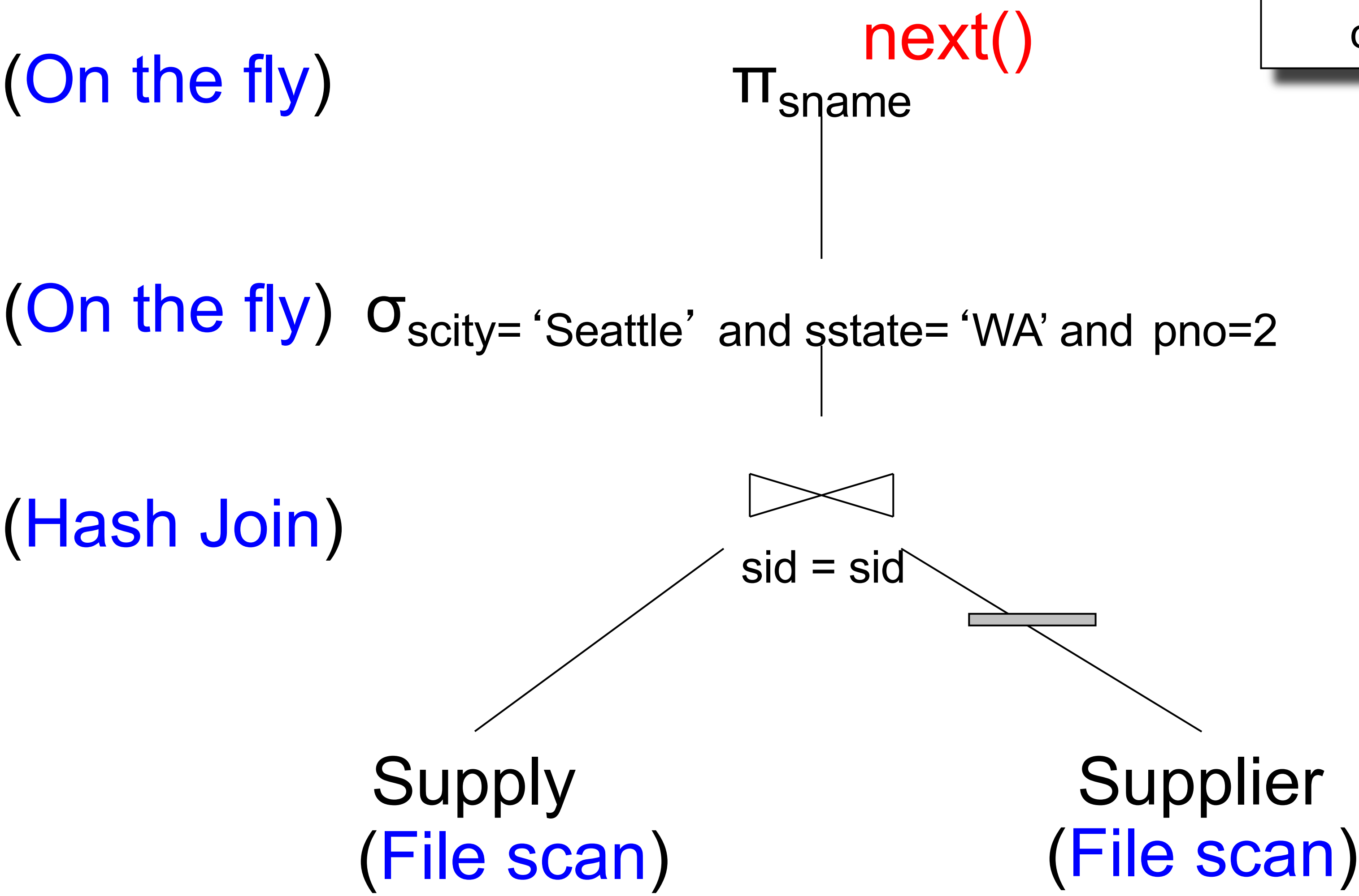


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

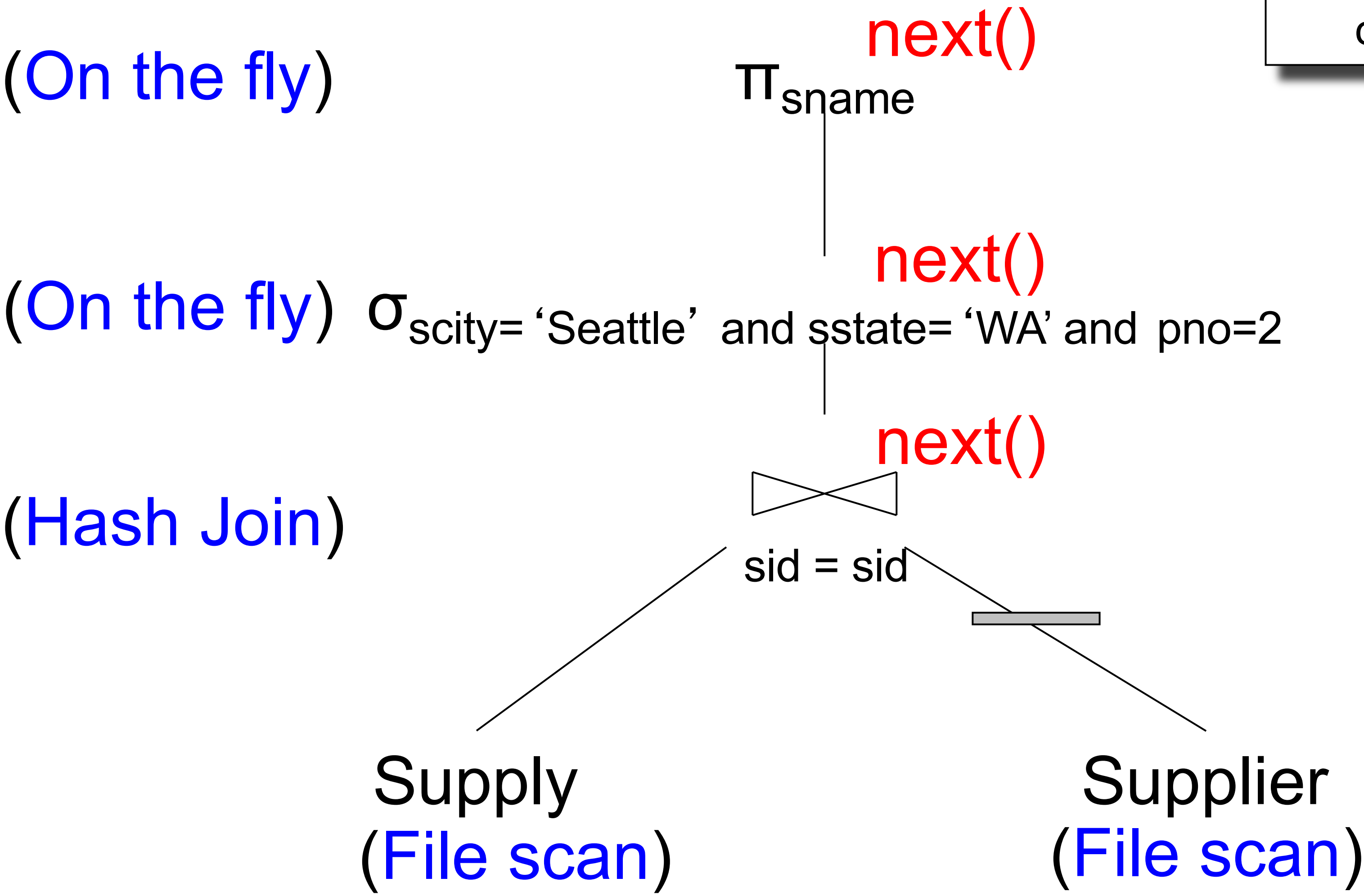


Volcano Model

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

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

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



```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Volcano Model

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

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

(On the fly) π_{sname} next()

(On the fly) $\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$ next()

(Hash Join) $\bowtie_{sid = sid}$ next()

next() Supply (File scan)

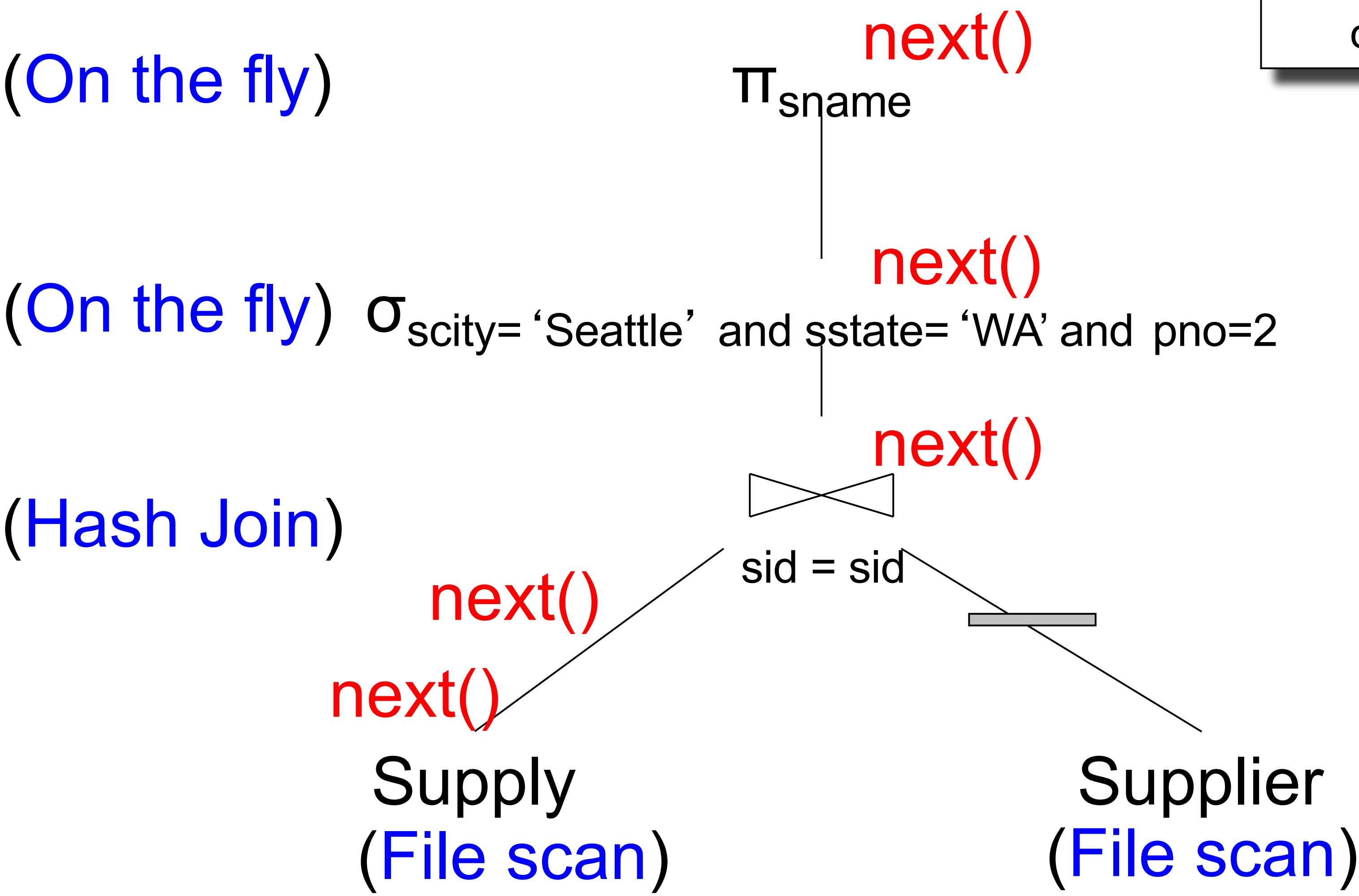
Supplier (File scan)

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

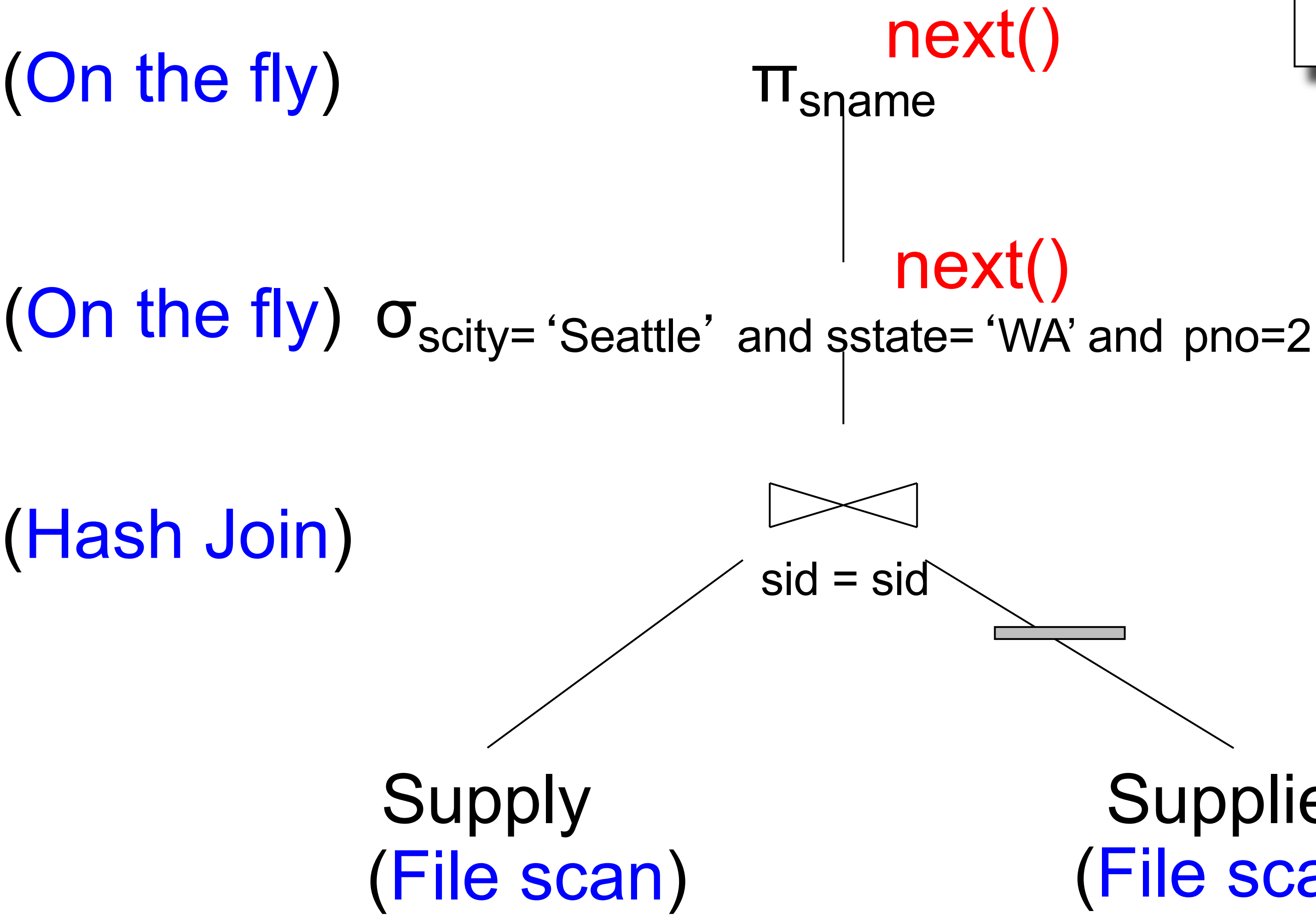


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Volcano Model

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

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

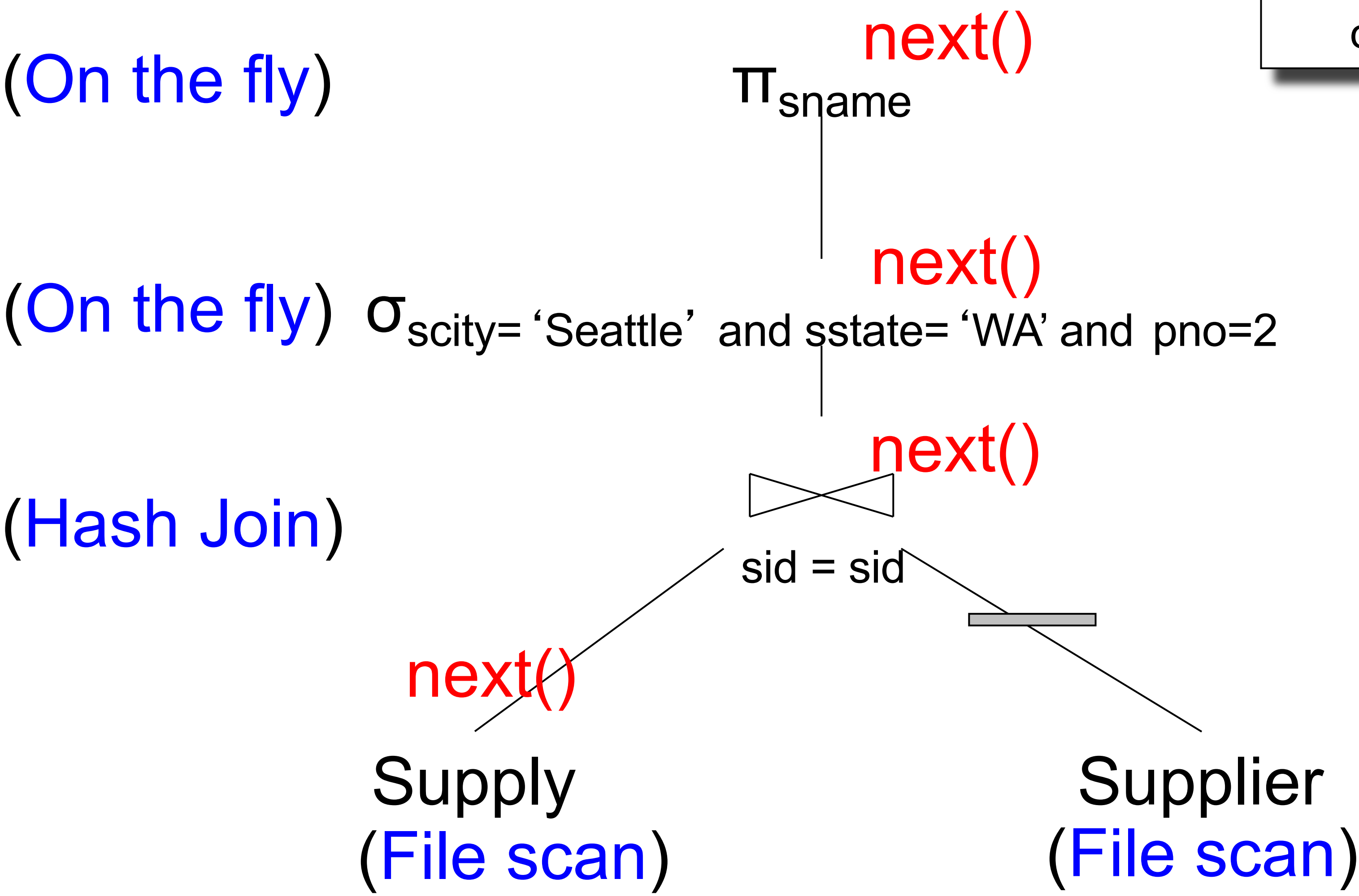


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

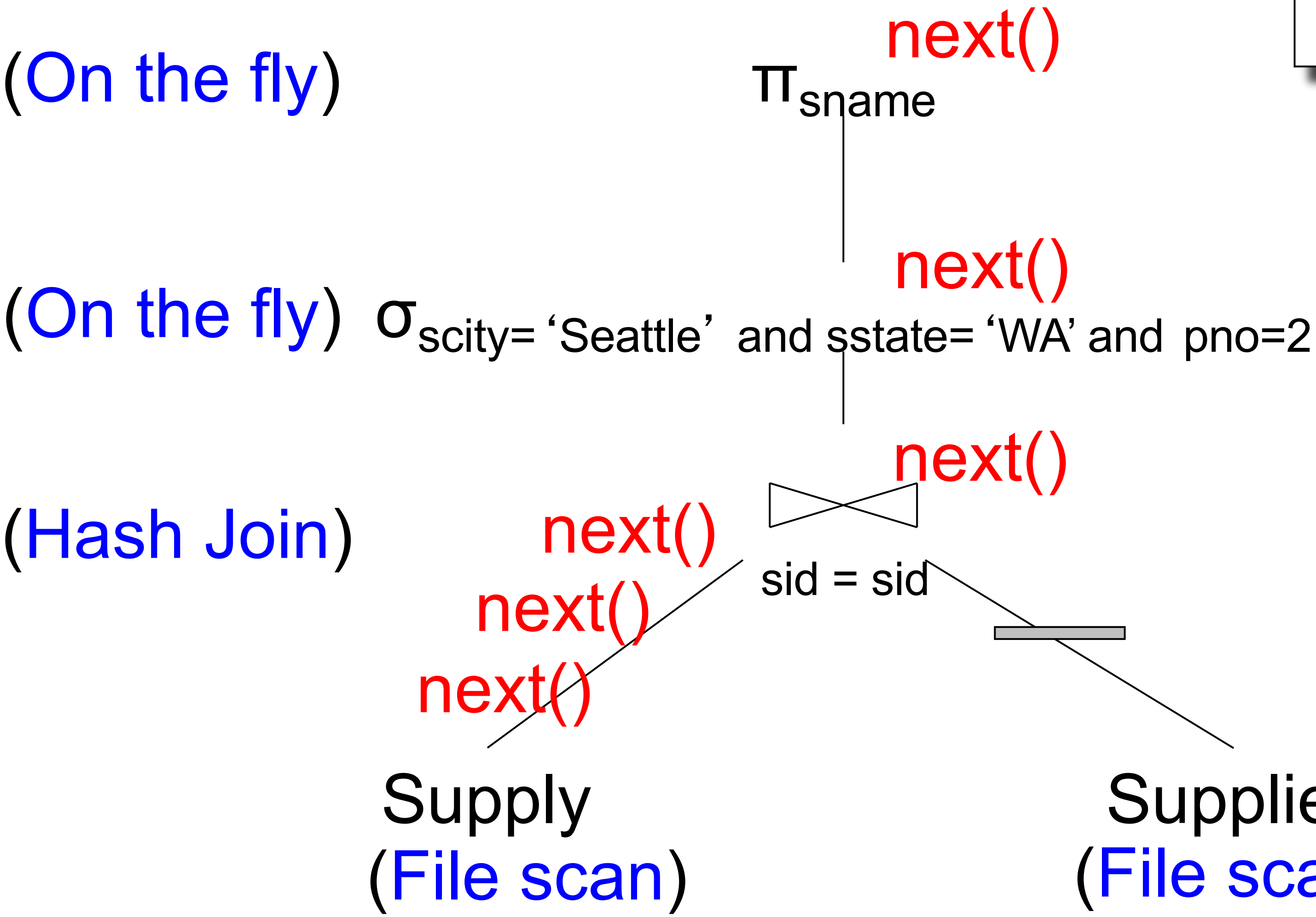


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Volcano Model

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

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

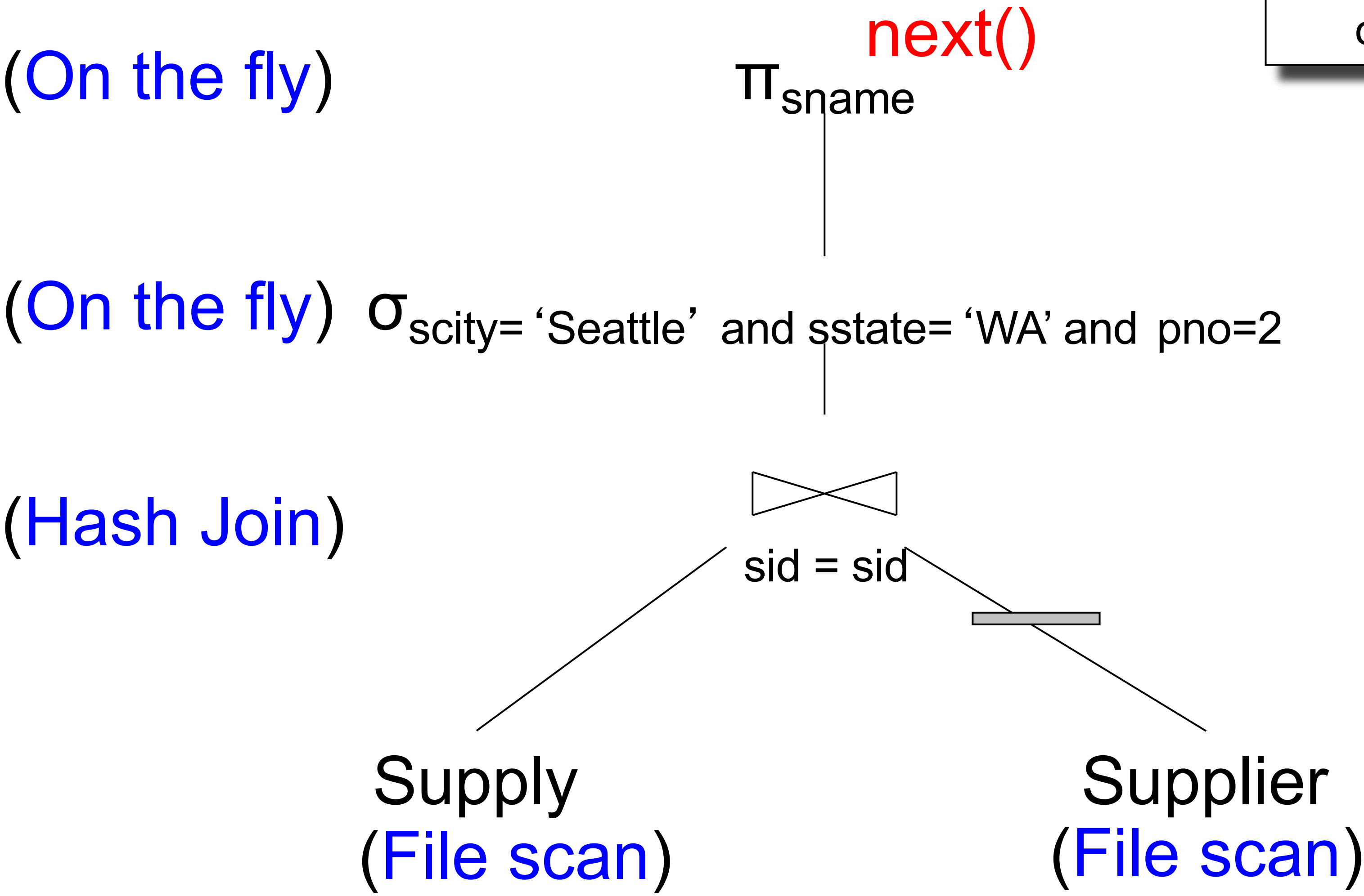


Volcano Model

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

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

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



Data-Driven Model

Each operator exports four methods:

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

called **once** by parent

called **repeatedly** by children

```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

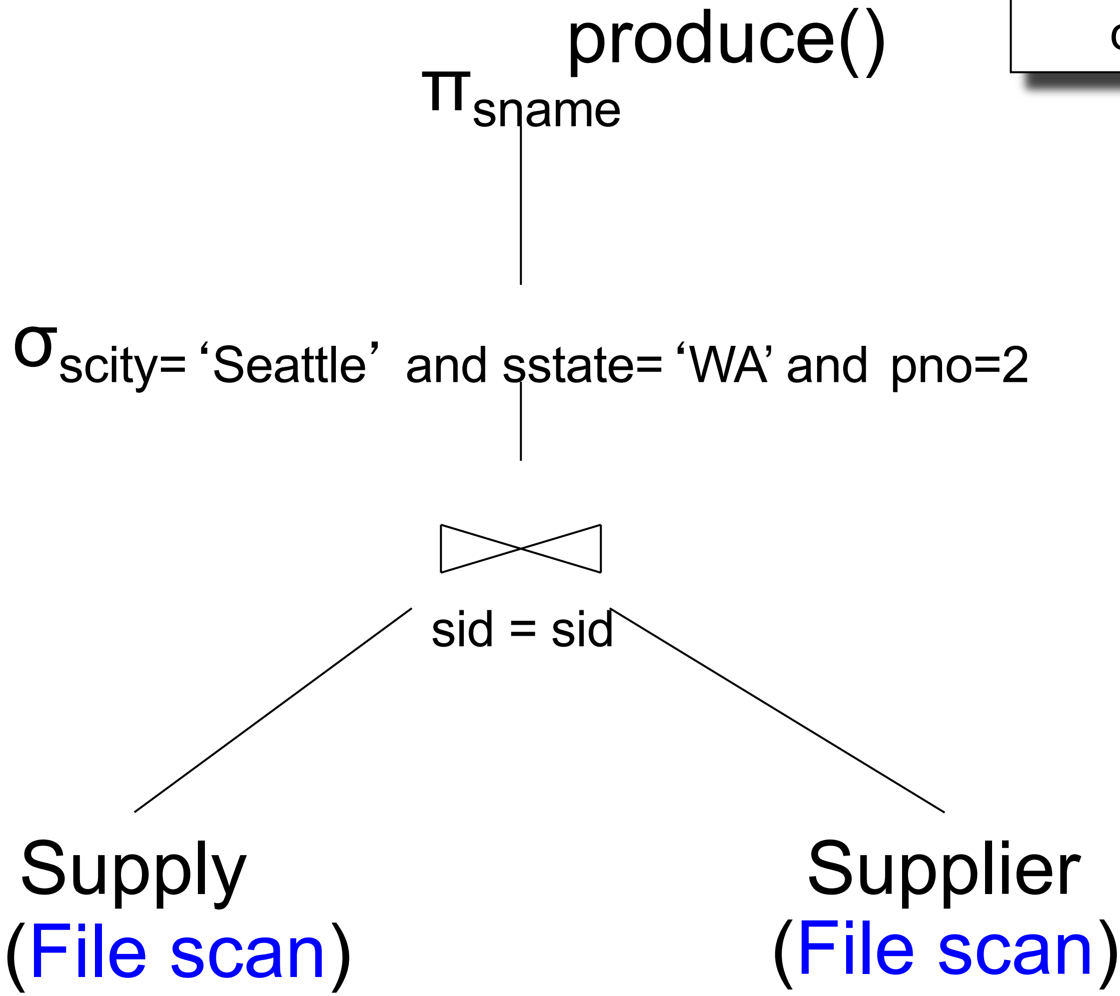
```
for x in Supplier do
    insert(x.sid, x)

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

(On the fly)

(On the fly)

(Hash Join)

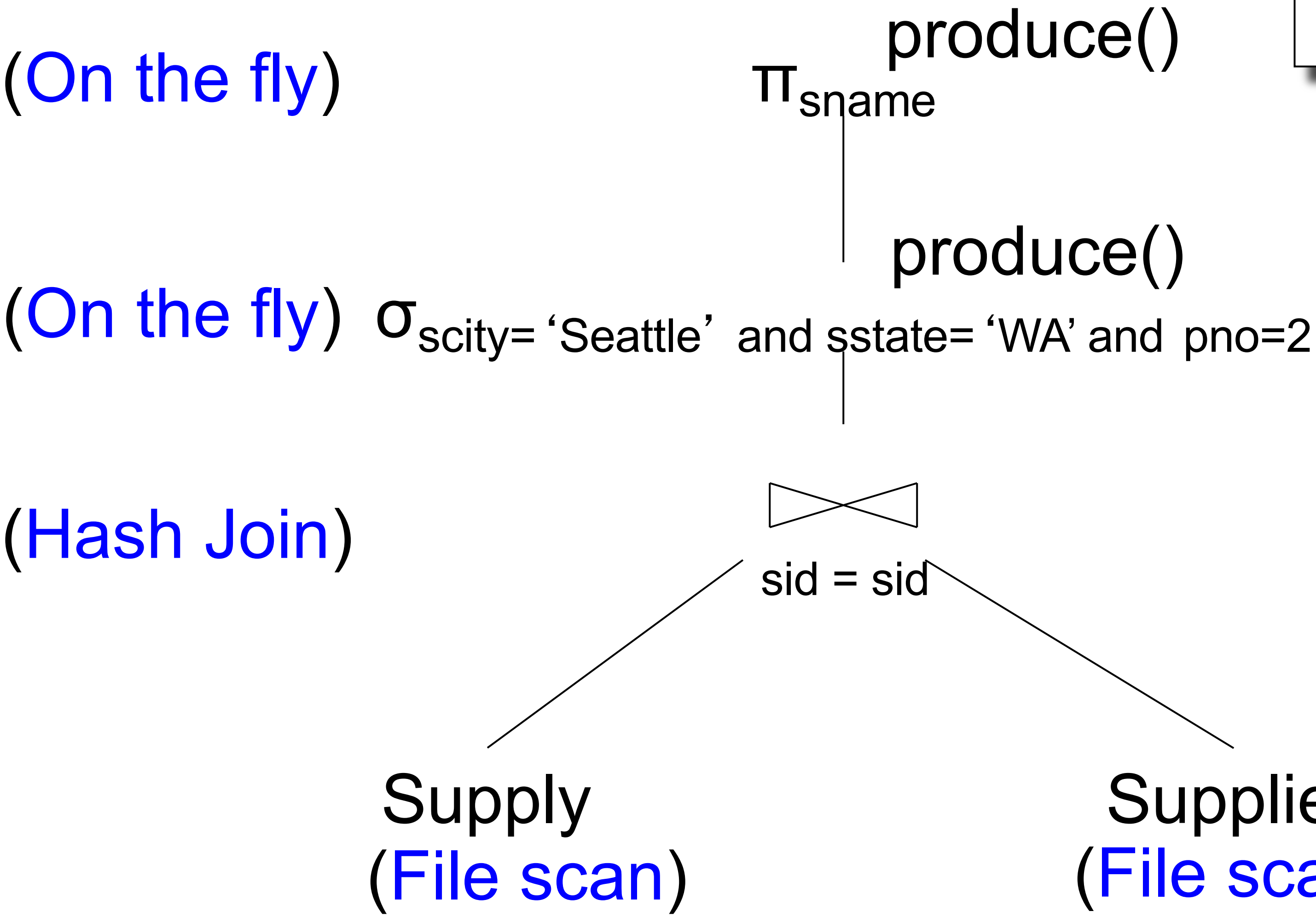


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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



```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

for x in Supplier do
    insert(x.sid, x)

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

produce()

(On the fly) π_{sname}

produce()

(On the fly) $\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$

produce()

(Hash Join) $\bowtie$ sid = sid

Supply (File scan)

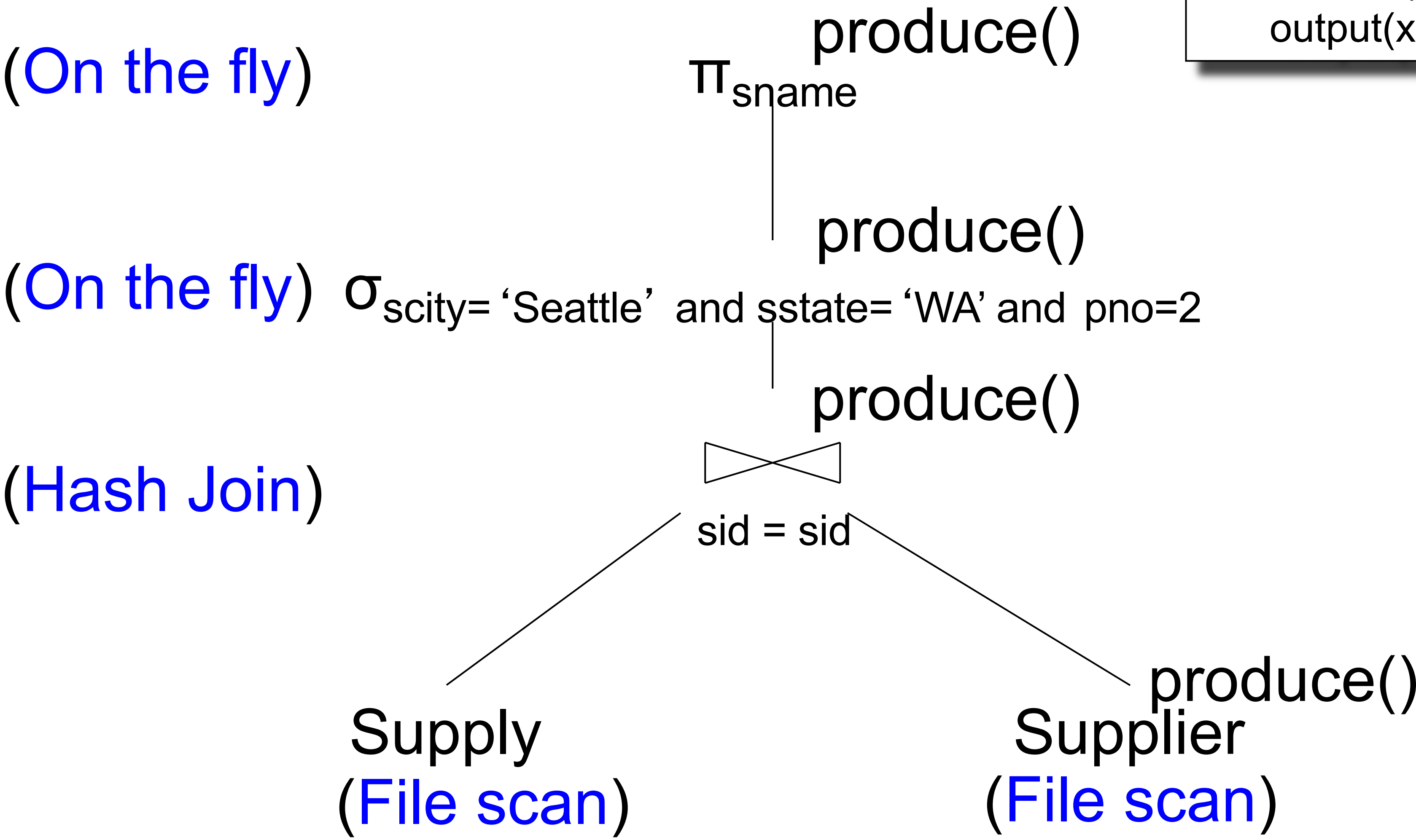
Supplier (File scan)

`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Data-Driven

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

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

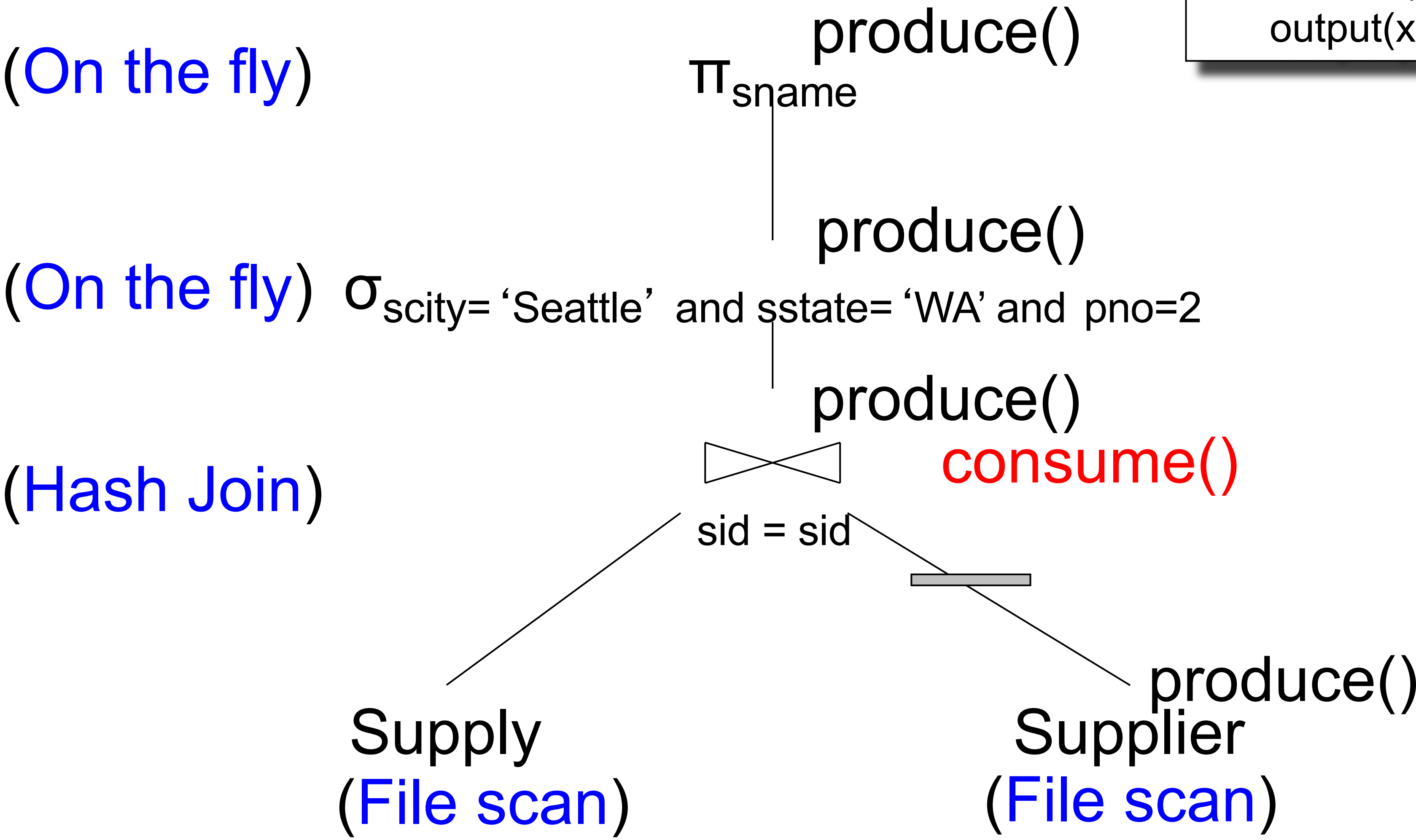


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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

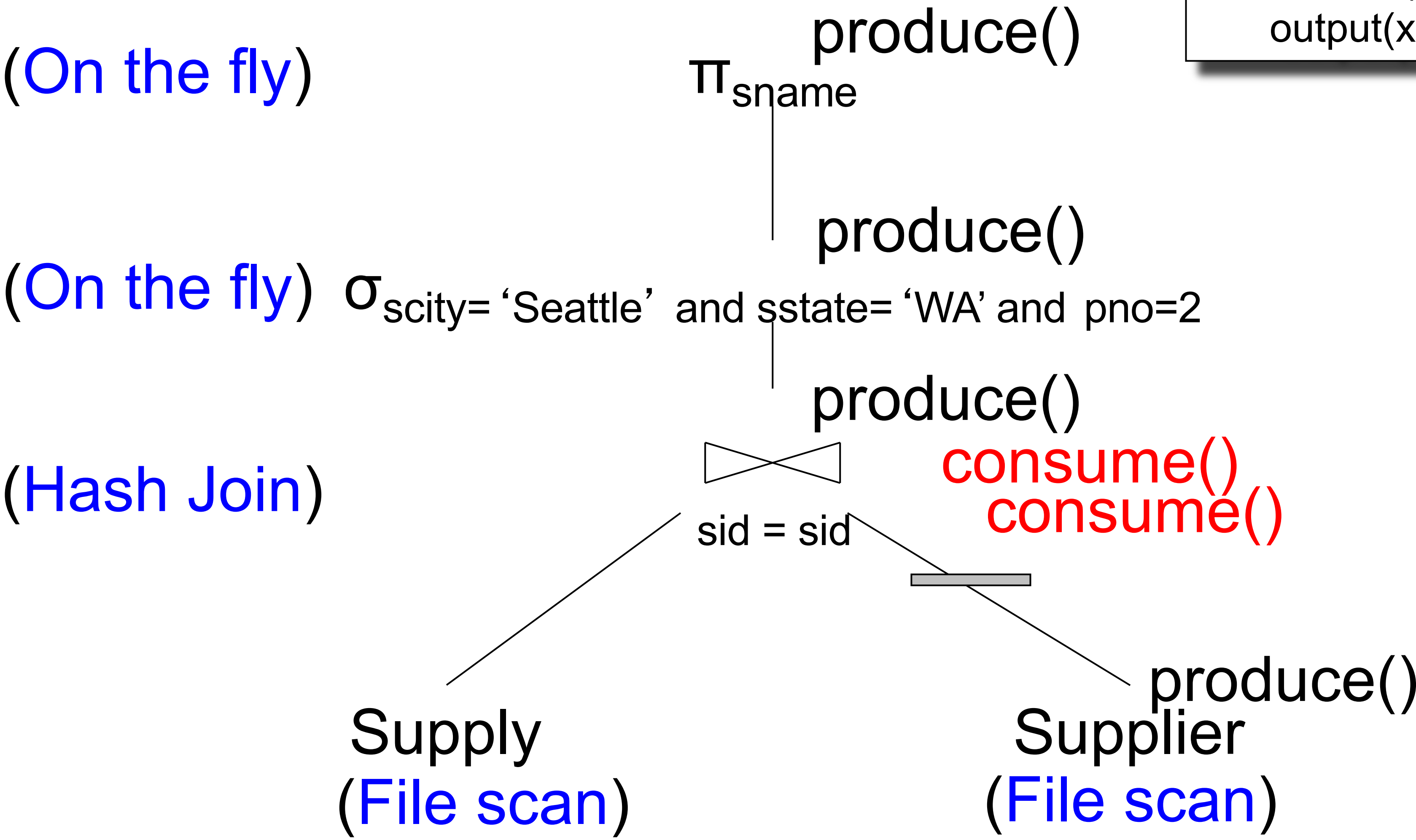


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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

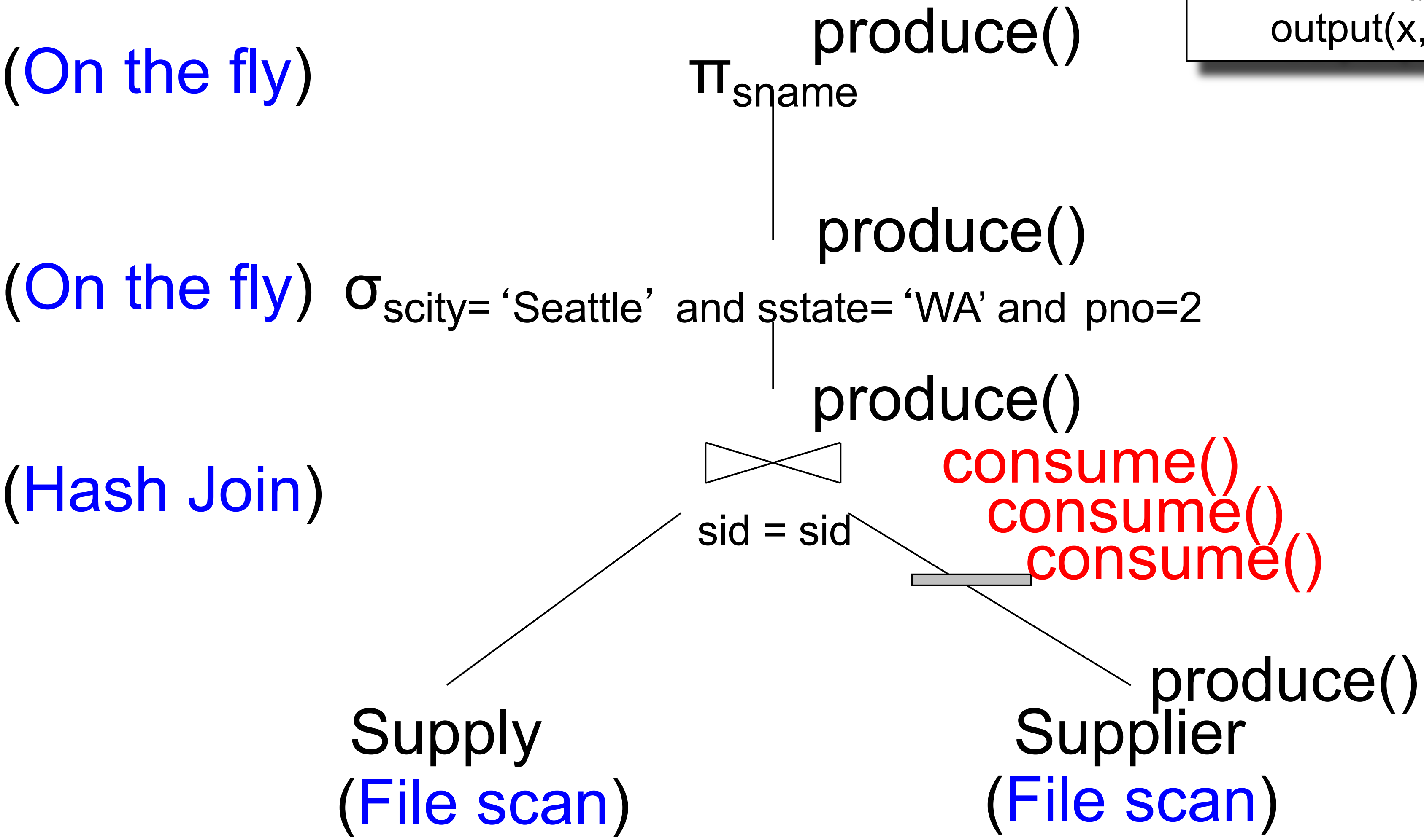


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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

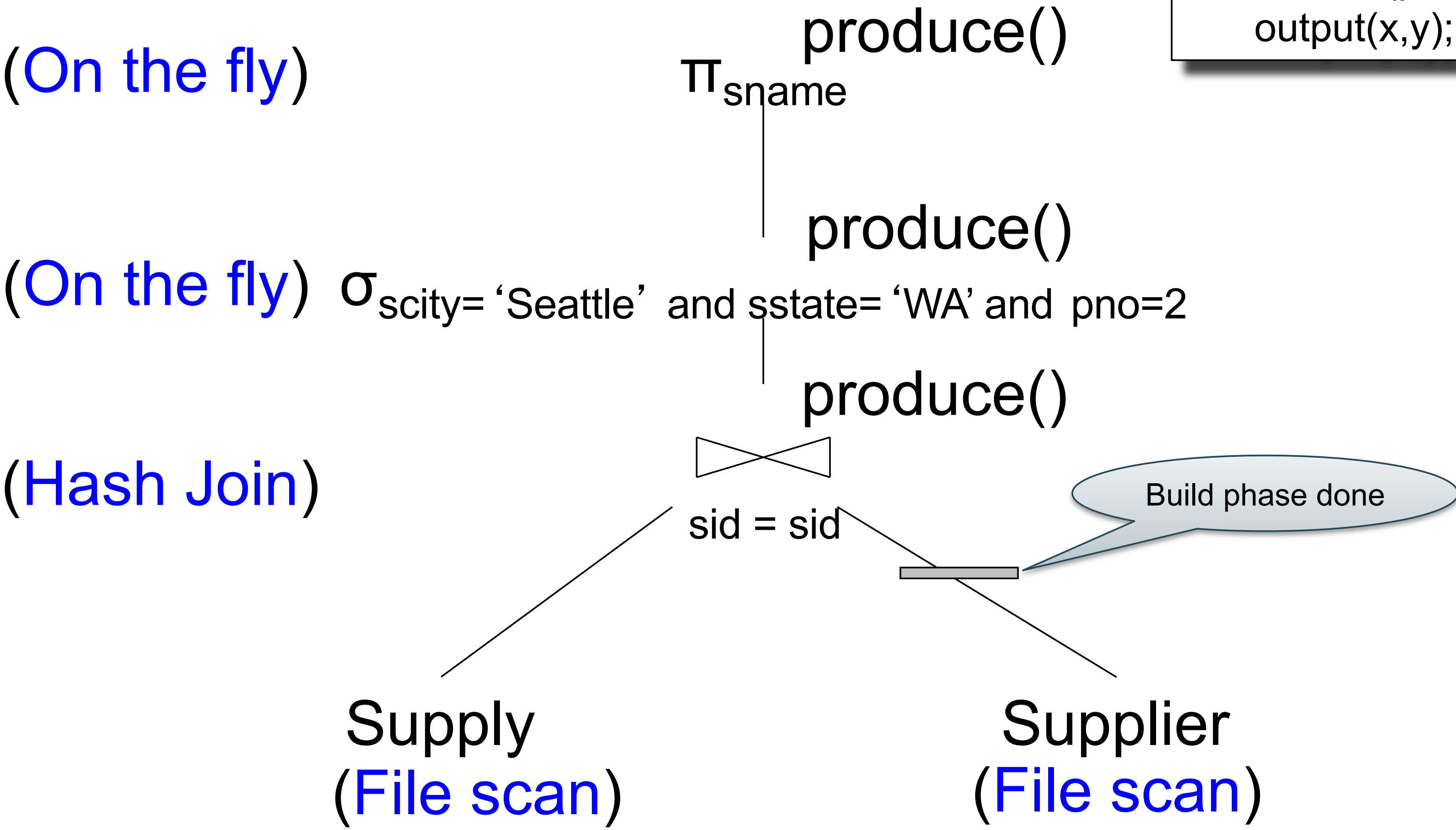


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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

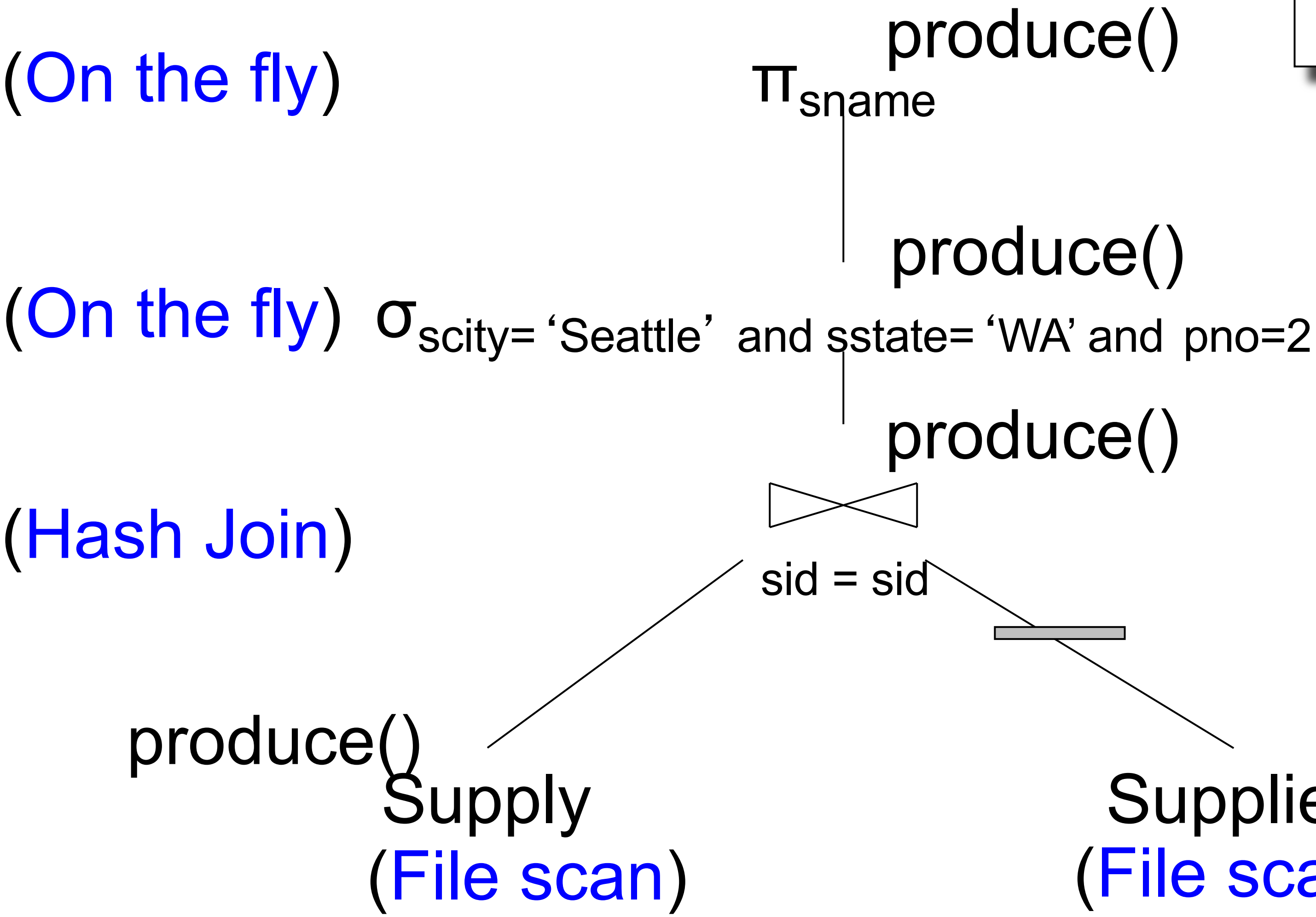


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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

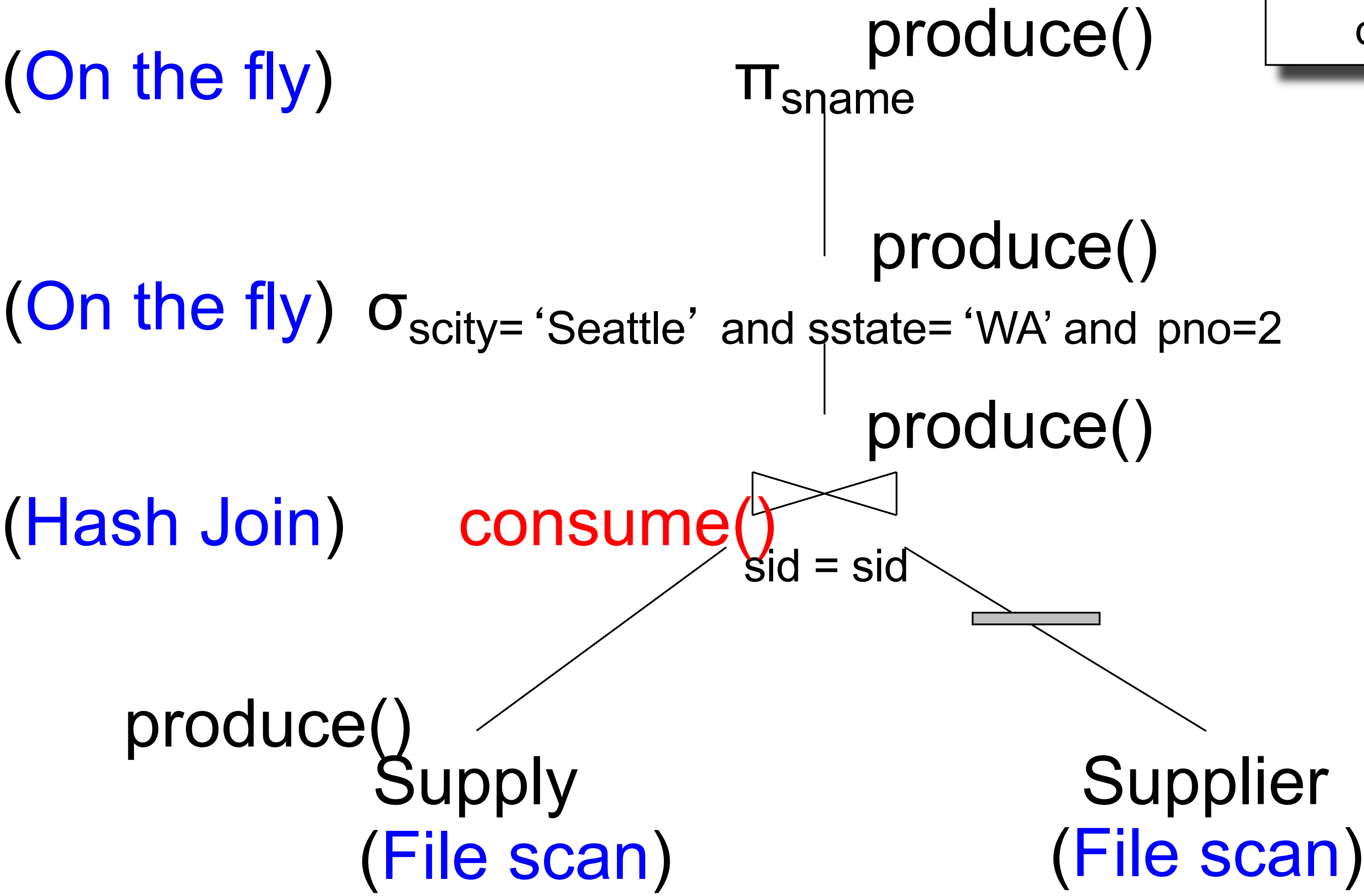


Supplier(<u>sid</u>, sname, scity, sstate)
Supply(<u>sid, pno</u>, quantity)

Data-Driven

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

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

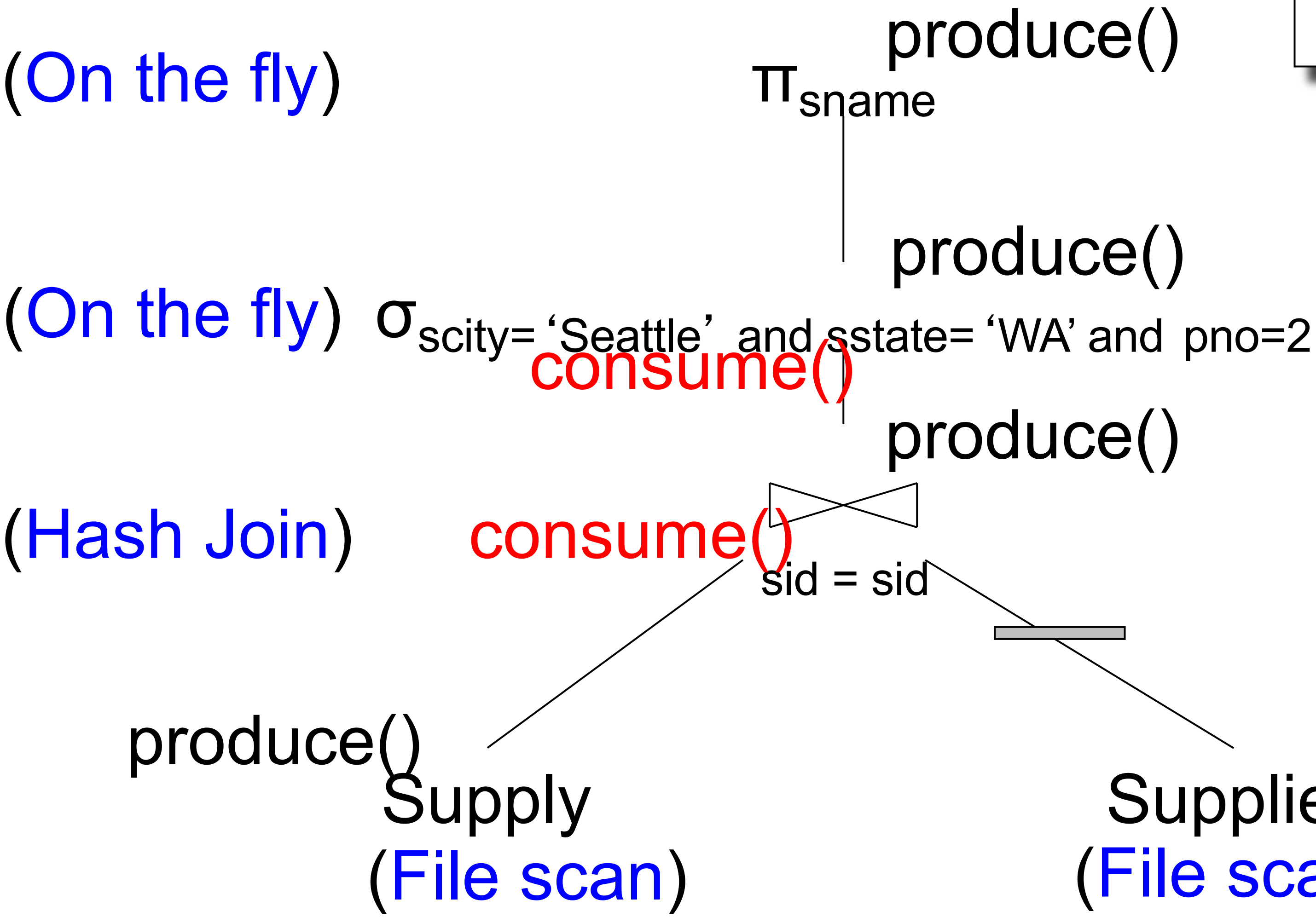


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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

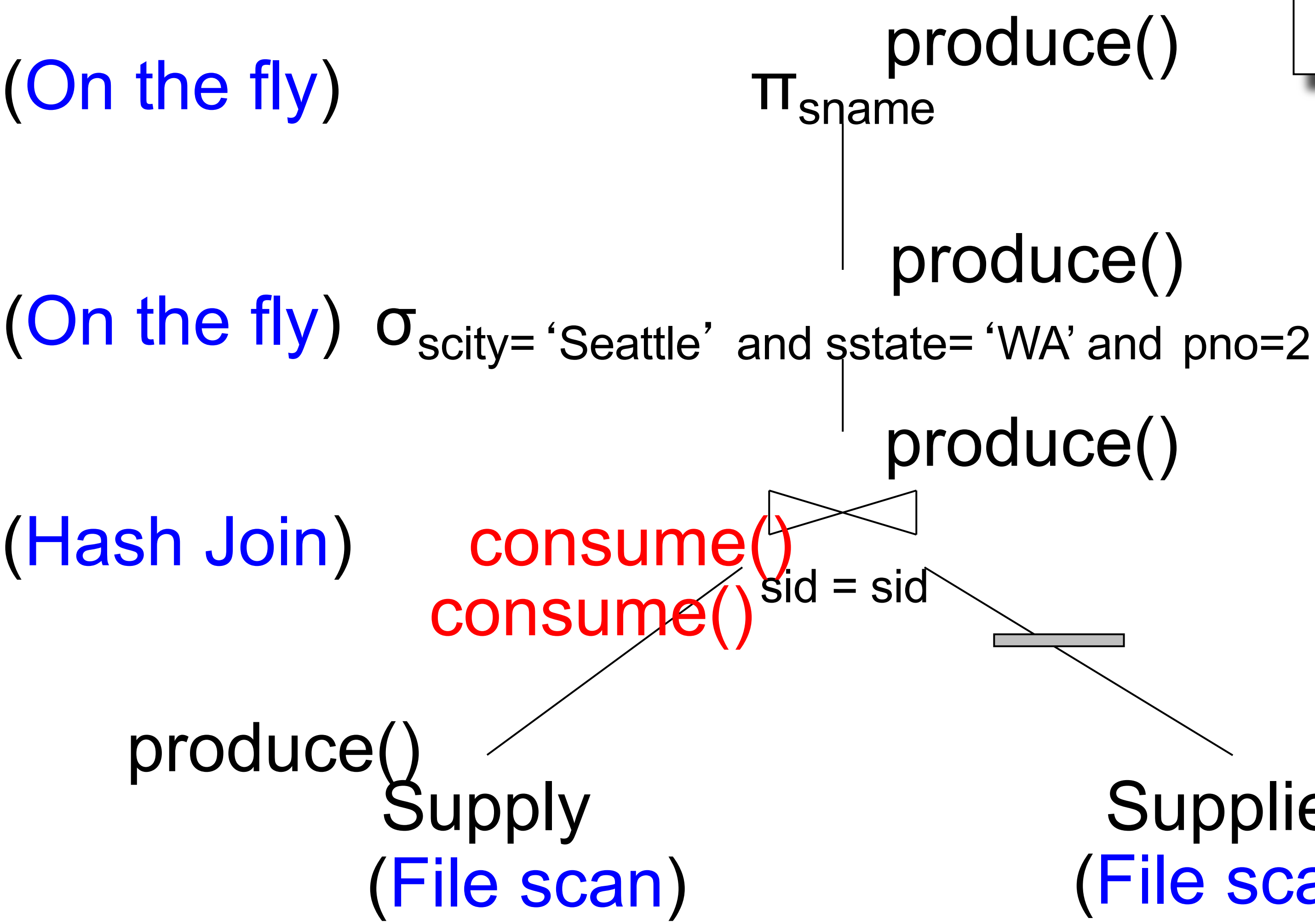


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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

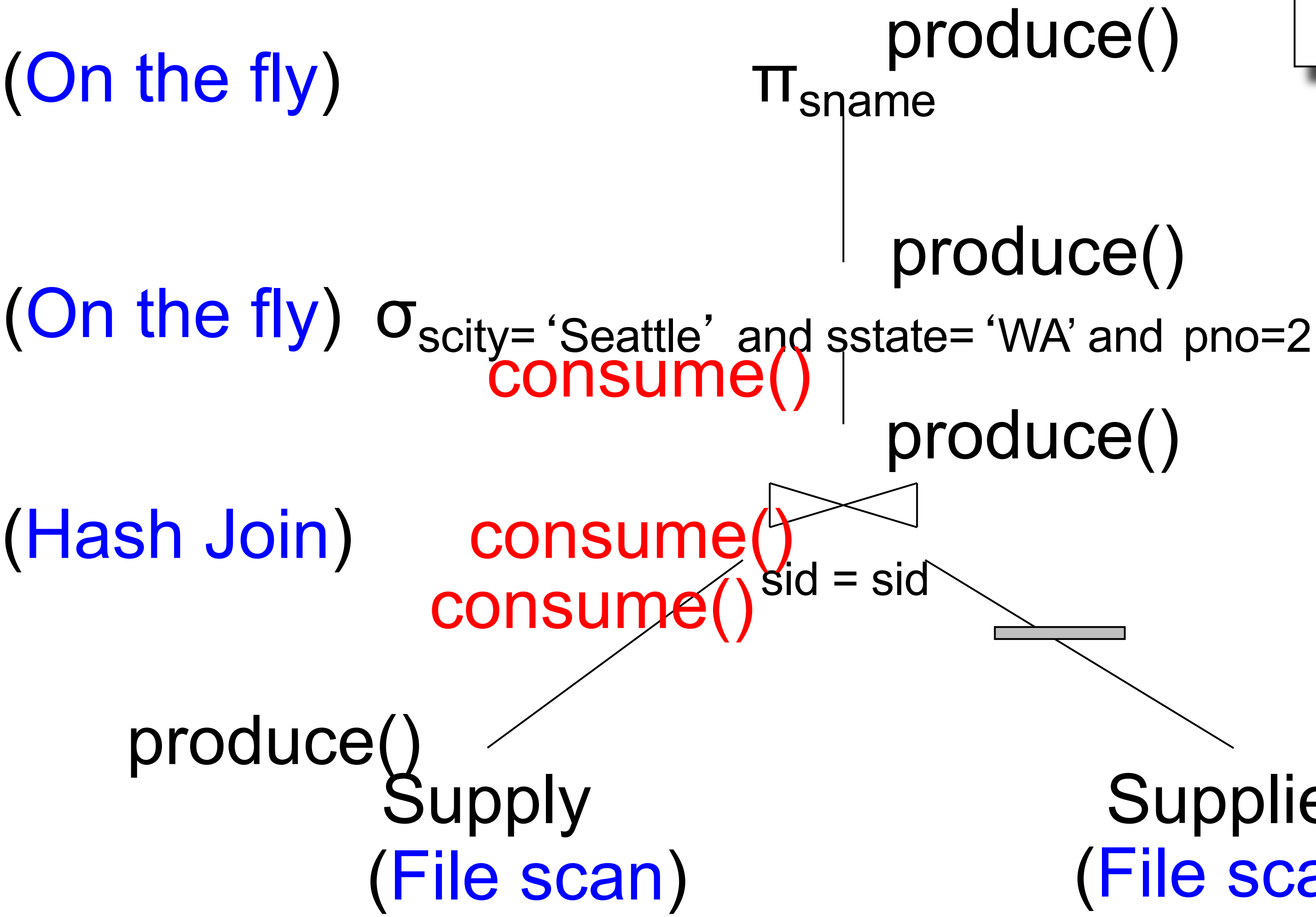


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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



Supplier(<u>sid</u>, sname, scity, sstate)
Supply(<u>sid, pno</u>, quantity)

Data-Driven

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

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

(On the fly) consume() π_{sname} produce()

(On the fly) consume() $\sigma_{scity='Seattle' \text{ and } sstate='WA' \text{ and } pno=2}$ produce()

(Hash Join) consume() consume() $\bowtie_{sid = sid}$ produce()

produce() Supply (File scan)

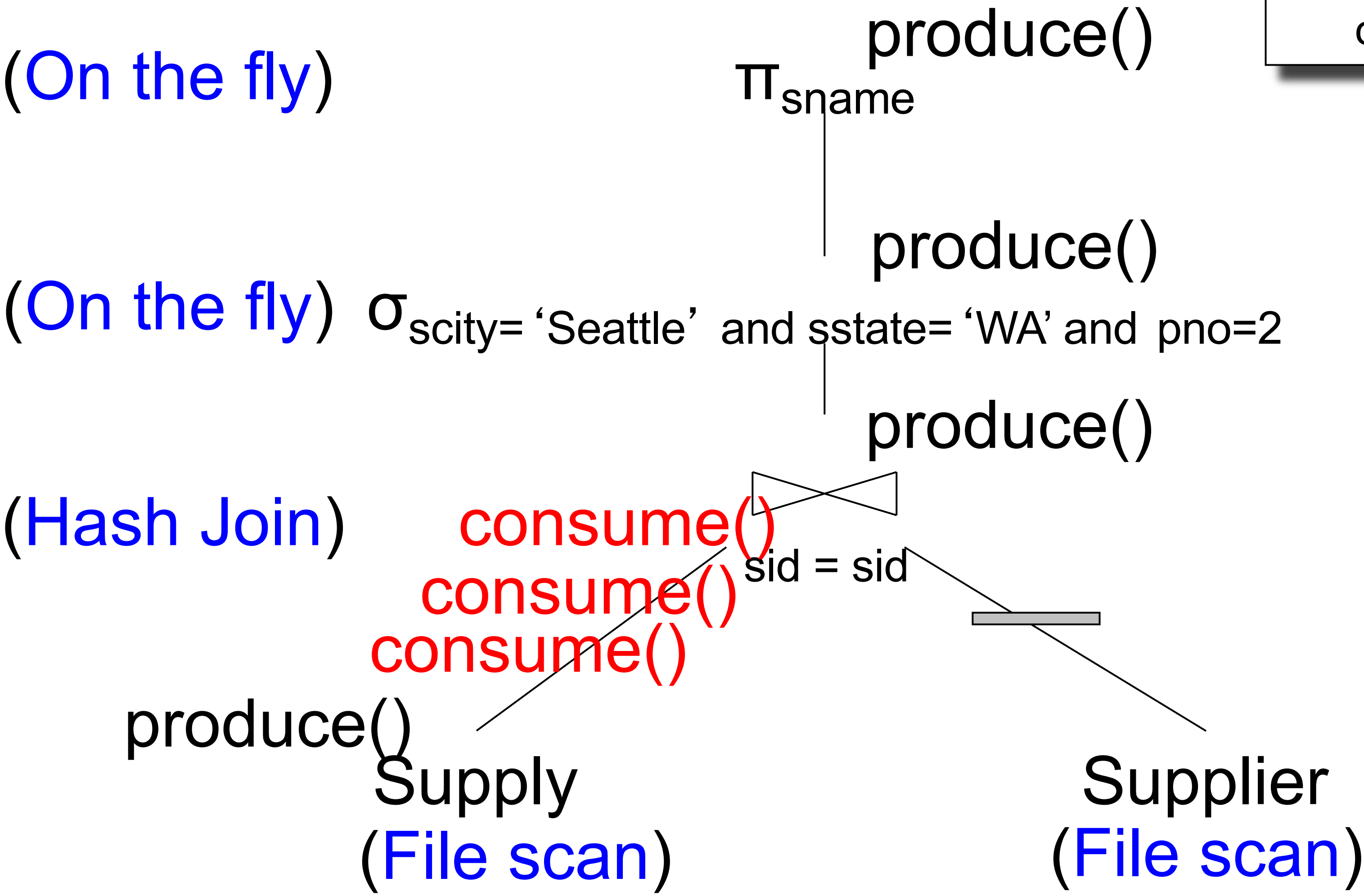
Supplier (File scan)

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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

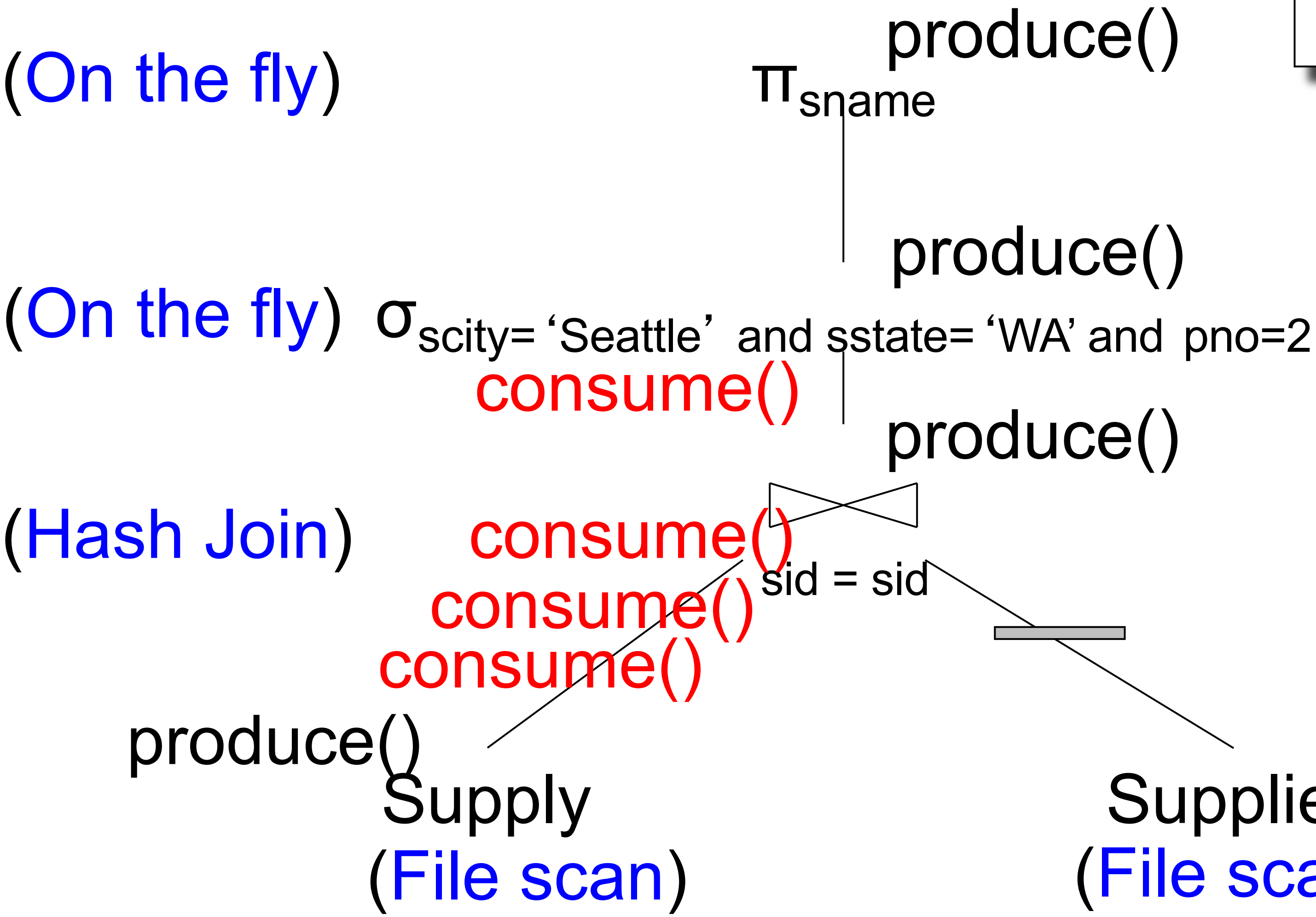


```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Data-Driven

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

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

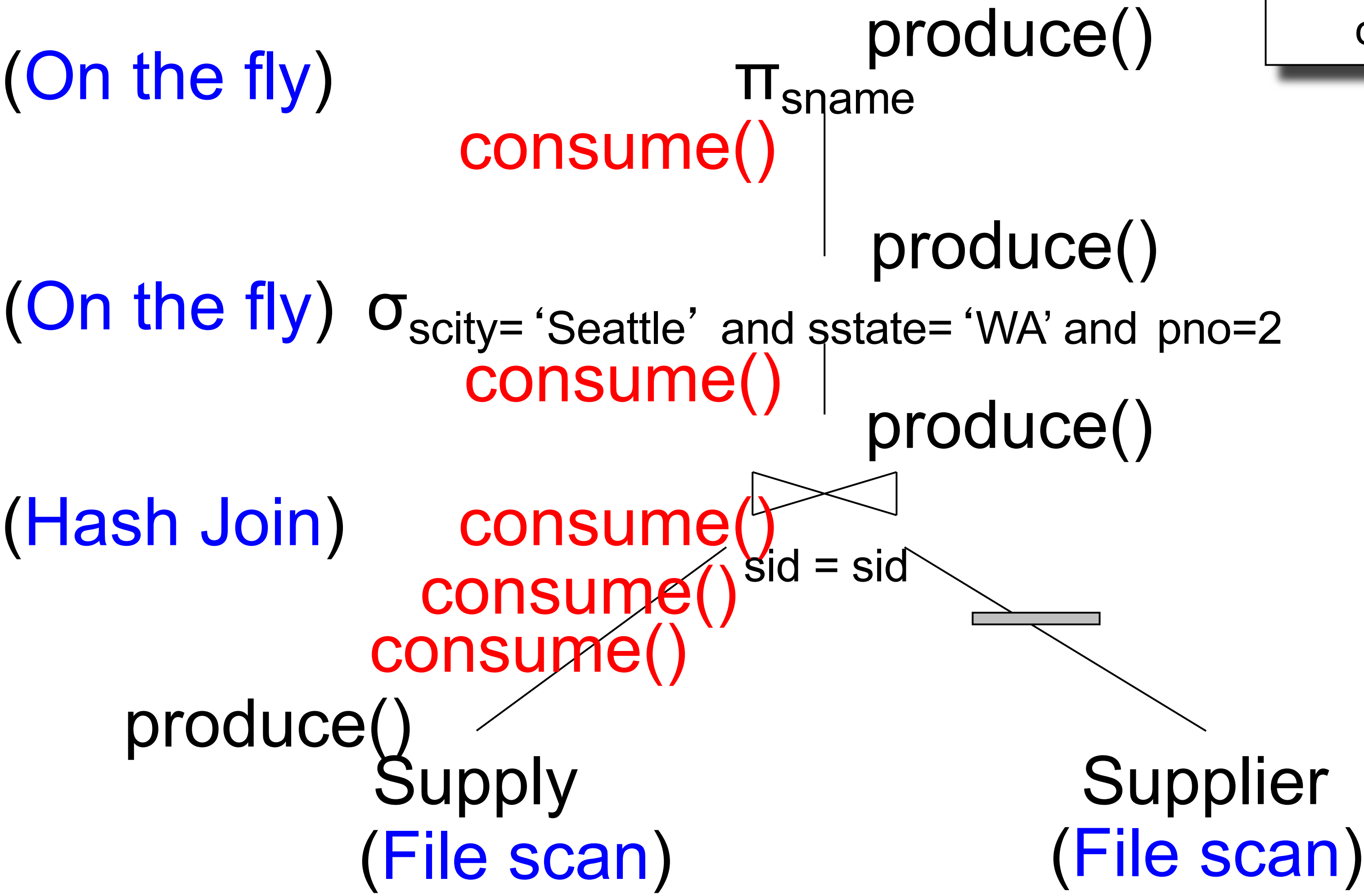


Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Data-Driven

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

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



Call-back

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

- Paper's solution: call-back function

[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))
  }
}
```

Left/right convention reversed

(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))
  }
}
```

Left/right convention reversed

(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))
  }
}
```

Left/right convention reversed

(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))
  }
}
```

Left/right convention reversed

(a) (b)

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

```
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)

```
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)
    }
    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)

```
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)

```
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)

Final Thoughts

- Volcano model:
  - next() returns single tuple – inefficient
- Vectorized model:
  - next() returns a bundle, e.g. 1000 tuples
- Partial evaluation:
  - specialize a function to some parameters
- Futamura projection:
  - specialize an interpreter to a program

Outline

- Steps involved in processing a query
- Main Memory Operators
- Query execution
- External Memory Operators

External Memory Algorithms

- Selection and index-join
- Nested loop join
- Partitioned hash-join, a.k.a. grace join
- Merge-join

Cost Parameters

- In database systems the data is on disk
- Parameters:
  - B(R) = # of blocks (i.e., pages) for relation R
  - T(R) = # of tuples in relation R
  - V(R, a) = # of distinct values of attribute a
  - M = # pages available in main memory
- Cost = total number of I/Os
- Convention: writing the final result to disk is *not included*

`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Cost Parameters

`Supplier(sid,sname,scity,sstate)`

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) =
- V(Supplier, sname) =
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Cost Parameters

Supplier(sid,sname,scity,sstate)

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) =
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Cost Parameters

```
Supplier(sid,sname,scity,sstate)
```

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Cost Parameters

Supplier(sid,sname,scity,sstate)

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) = 860
- V(Supplier, sstate) =
- M =

`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Cost Parameters

`Supplier(sid,sname,scity,sstate)`

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) = 860
- V(Supplier, sstate) = 50 why?
- M =

Cost Parameters

Supplier(sid,sname,scity,sstate)

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) = 860
- V(Supplier, sstate) = 50 why?
- M = 10,000,000 = 80GB why so little?

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R)

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  – cost = B(R)
- Index-based selection:
  – Unclustered index on a: cost = T(R) / V(R,a)

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R)
- Index-based selection:
  - Unclustered index on a: cost = T(R) / V(R,a)
  - Clustered index on a: cost = B(R) / V(R,a)

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R)
- Index-based selection:
  - Unclustered index on a: cost = T(R) / V(R,a)
  - Clustered index on a: cost = B(R) / V(R,a)
- Assumptions:
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

```sql
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

$B(R) = 2000$
$T(R) = 100{,}000$
$V(R, a) = 20$

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R)
- Index-based selection:
  - Unclustered index on a: cost = T(R) / V(R,a)
  - Clustered index on a: cost = B(R) / V(R,a)
- Assumptions:
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

B(R) = 2000
T(R) = 100,000
V(R, a) = 20

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R) — 2000
- Index-based selection:
  - Unclustered index on a: cost = T(R) / V(R,a)
  - Clustered index on a: cost = B(R) / V(R,a)
- Assumptions:
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

```
SELECT *
FROM Supplier
WHERE scity = 'Seattle'
```

Selection

B(R) = 2000
T(R) = 100,000
V(R, a) = 20

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

- Sequential scan:
  - cost = B(R) — 2000
- Index-based selection:
  - Unclustered index on a: cost = T(R) / V(R,a) — 5000
  - Clustered index on a: cost = B(R) / V(R,a) — 100
- Assumptions:
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

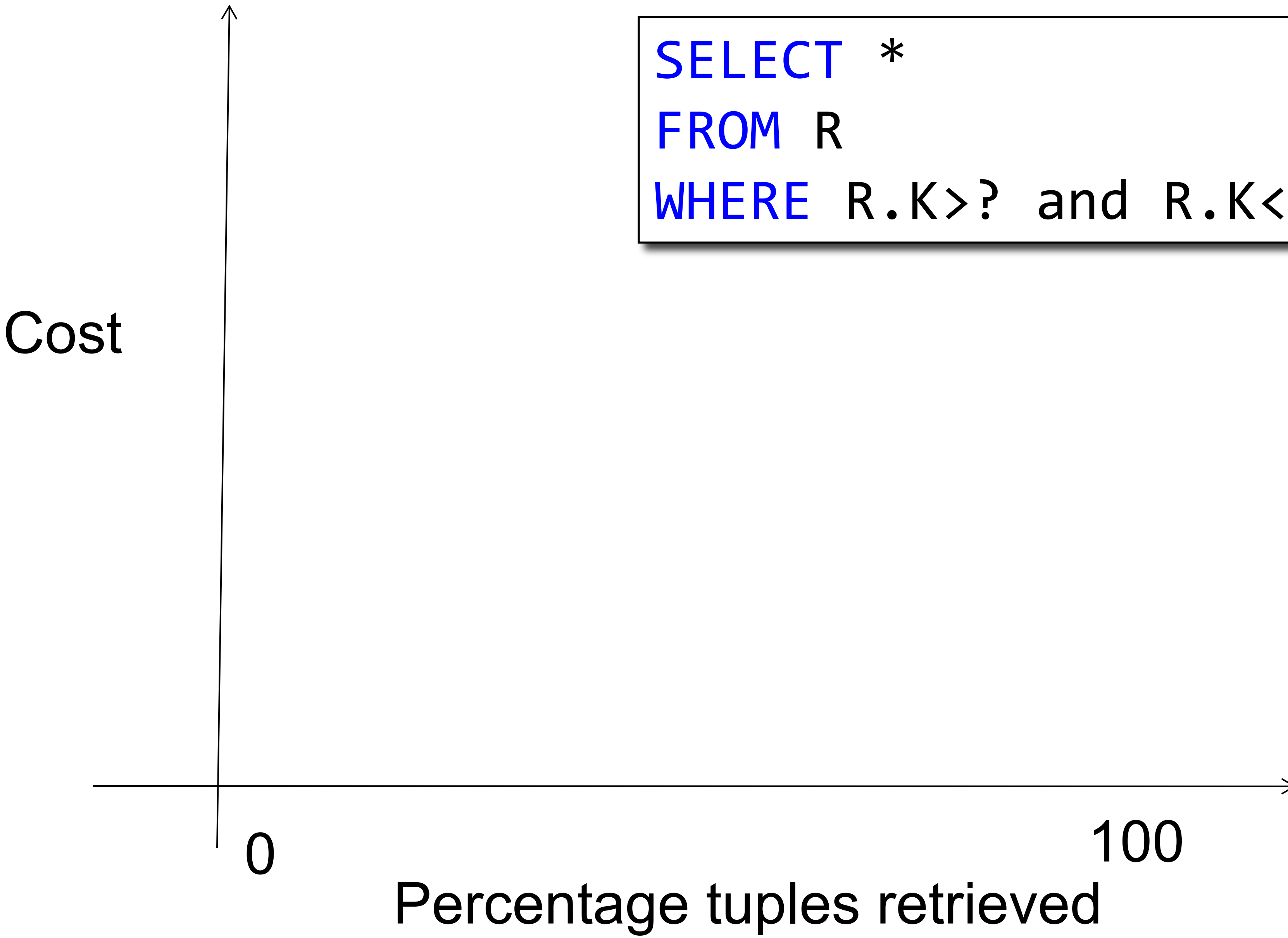
The 2% Rule

Rule of thumb:

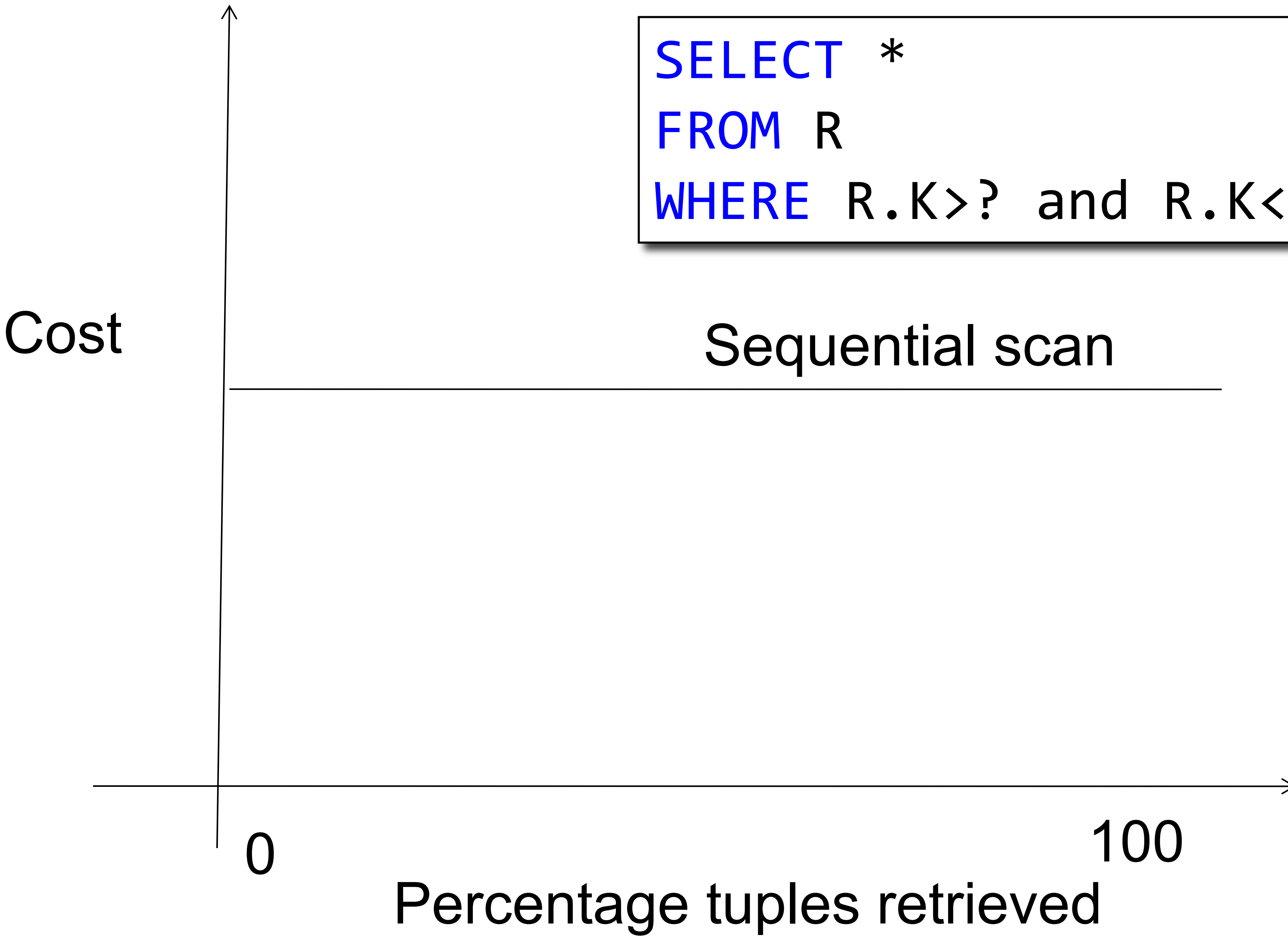
- If you read more than 2% of the data, then it's faster to do a full sequential scan than to use an unclustered index

Lesson: don't build unclustered indexes when V(R,a) is small

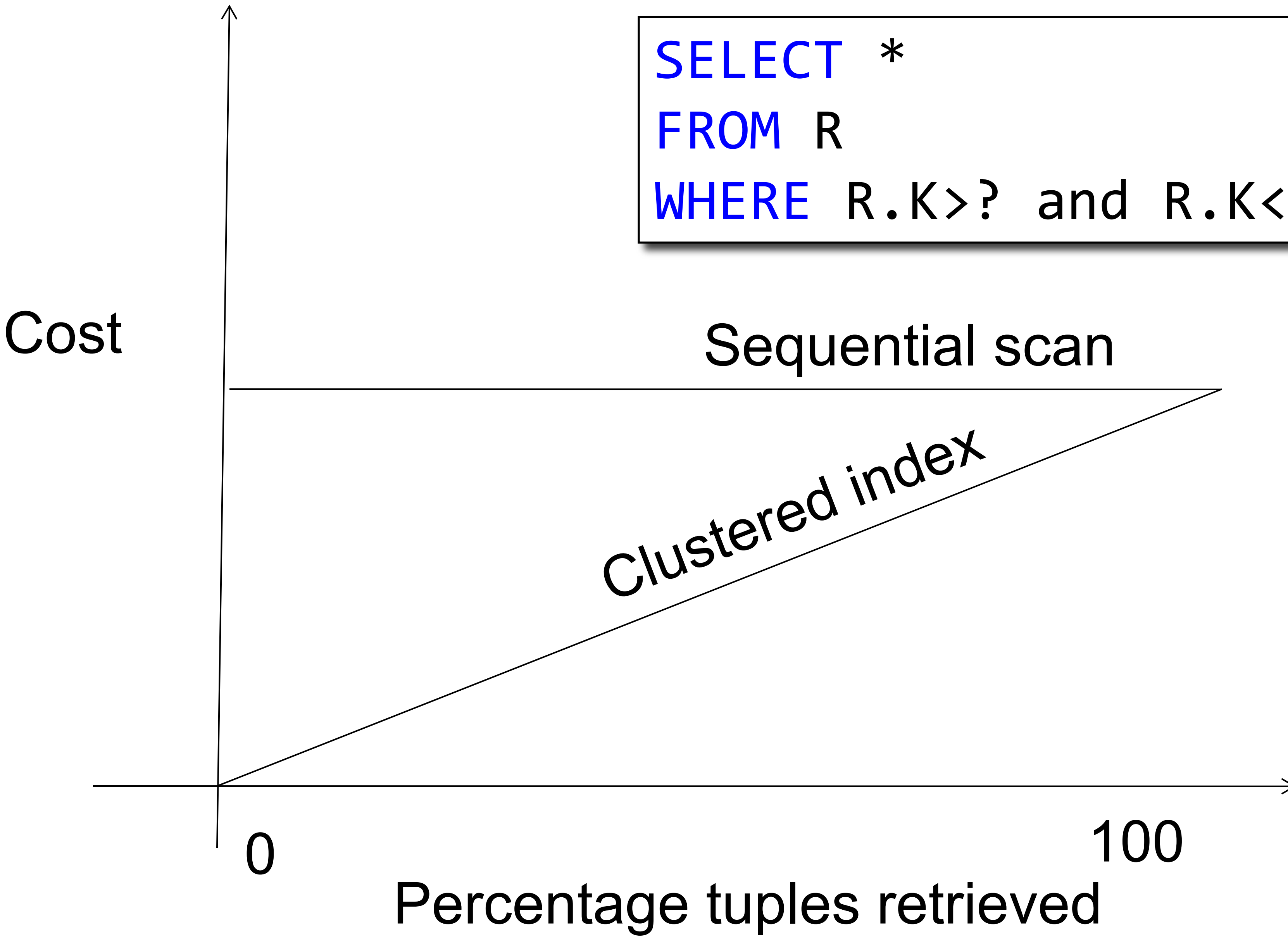
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



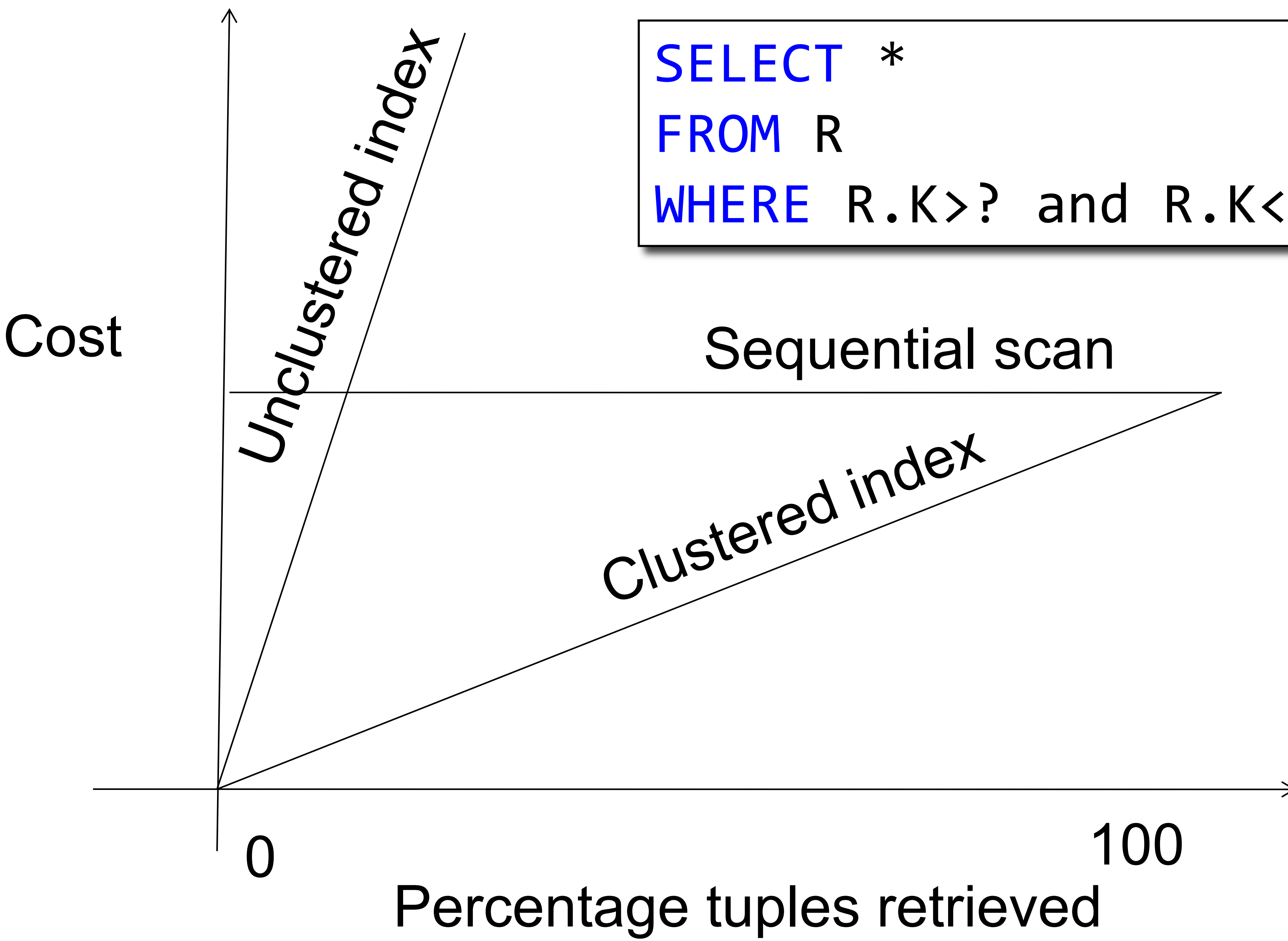
```sql
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```sql
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



Index Nested Loop Join

$R \bowtie S$

- Assume S has index on join attribute
- Iterate over R, probe each tuple in S
- Cost:
  - Clustered: $B(R) + T(R)B(S) / V(S,a)$
  - Unclustered: $B(R) + T(R)T(S) / V(S,a)$

External Memory Algorithms

- Selection and index-join
- Nested loop join
- Partitioned hash-join, a.k.a. grace join
- Merge-join

Nested Loop Joins

$R \bowtie S$

- Naïve nested loop join: $T(R) + T(R) * B(S)$
  - WHY?
- Switch order: $B(S) + B(R) * T(S)$

- We can be much cleverer by using the available main memory: M

Block Nested Loop Join

- Group of (M-2) pages of S is called a “block”

```
for each (M-2) pages ps of S do
  for each page pr of R do
    for each tuple s in ps
      for each tuple r in pr do
        if r and s join then output(r,s)
```

Main memory hash-join
$(M-1)ps \bowtie pr$

Block Nested Loop Join

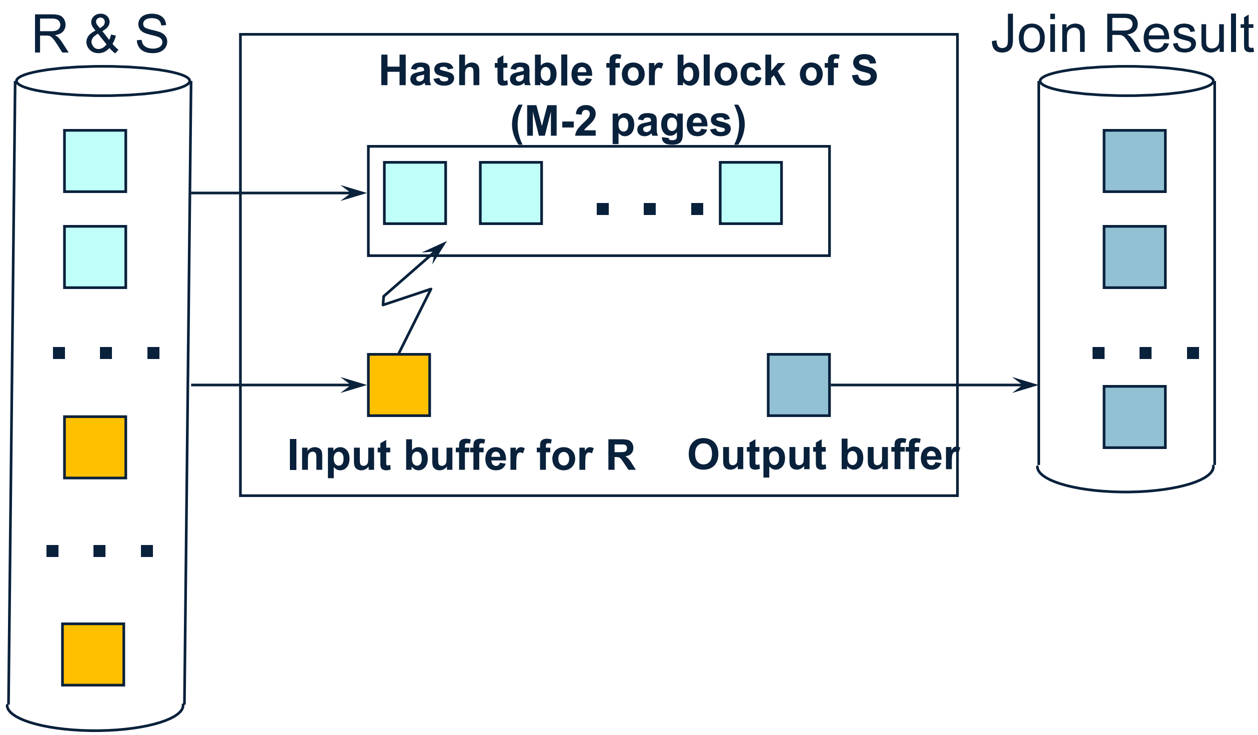
- Group of (M-2) pages of S is called a "block"

```
for each (M-2) pages ps of S do
  for each page pr of R do
    for each tuple s in ps
      for each tuple r in pr do
        if r and s join then output(r,s)
```

Main memory hash-join (M-1)ps ⋈ pr

B(S) + B(S)B(R)/(M-2) disk I/Os. WHY?

Block Nested Loop Join



B(S) + B(S)B(R)/(M-2) disk I/Os.

External Memory Algorithms

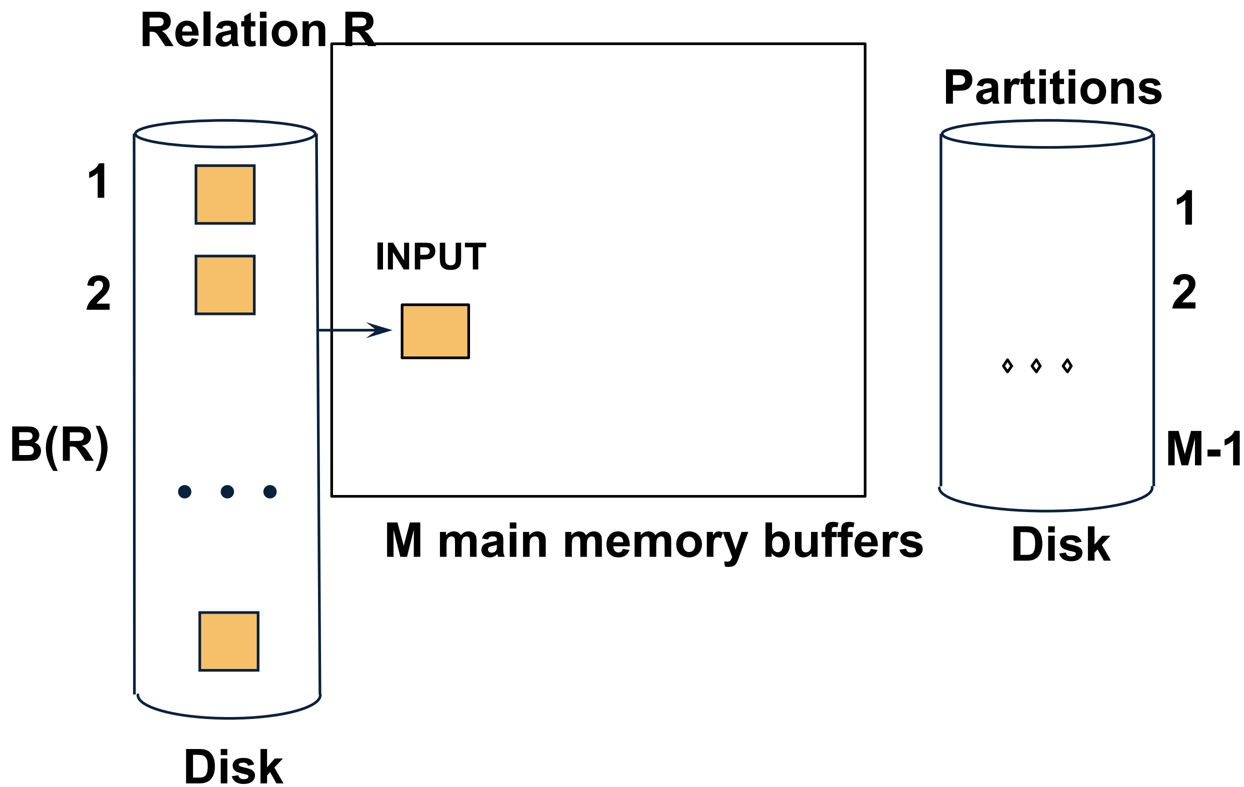
- Selection and index-join
- Nested loop join
- Partitioned hash-join, a.k.a. grace join
- Merge-join

Partitioned Hash-Join a.k.a. Grace Join

- R ⋈ S, both bigger than main memory
- Step 1:
  – Hash partition both R and S
  – Store buckets on disk
- Step 2:
  – Read one S-bucket in main memory
  – Join with corresponding R-bucket
  – Repeat for all buckets

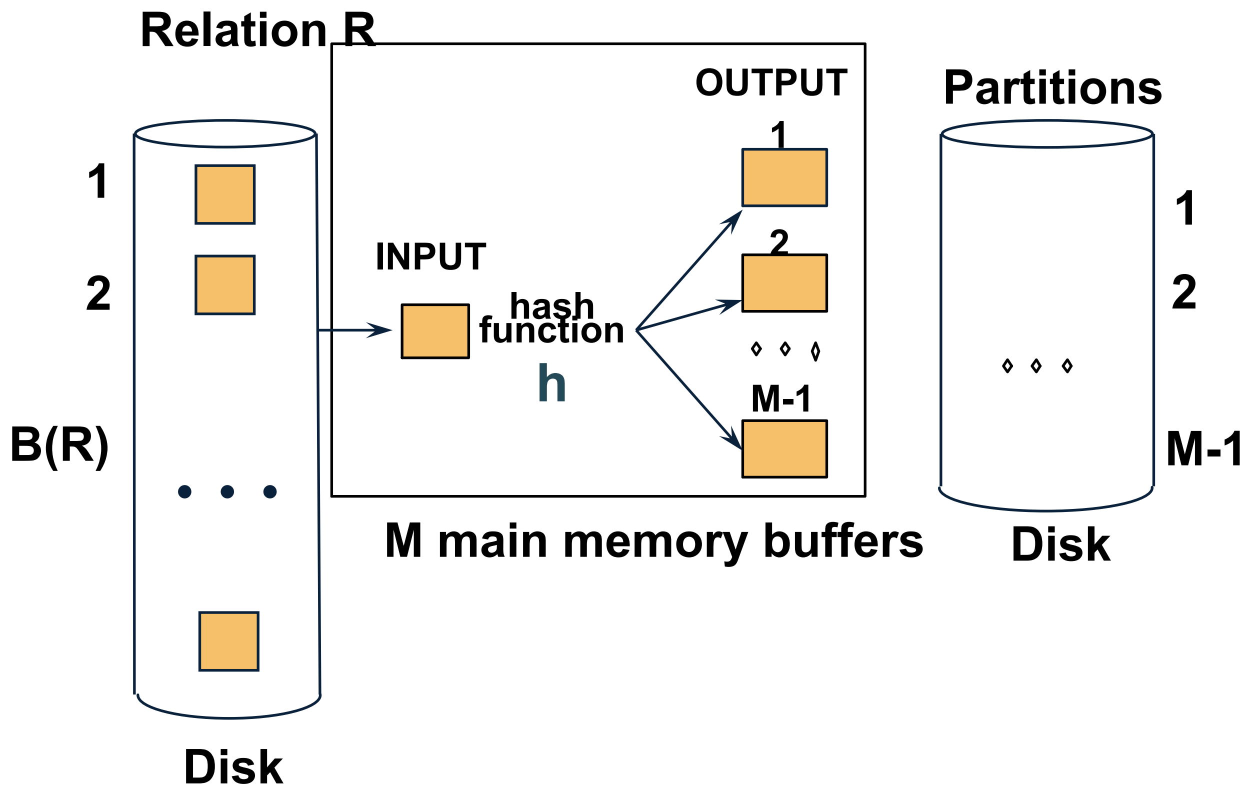
Step 1: Hash-partition

- Partition R into buckets, on disk



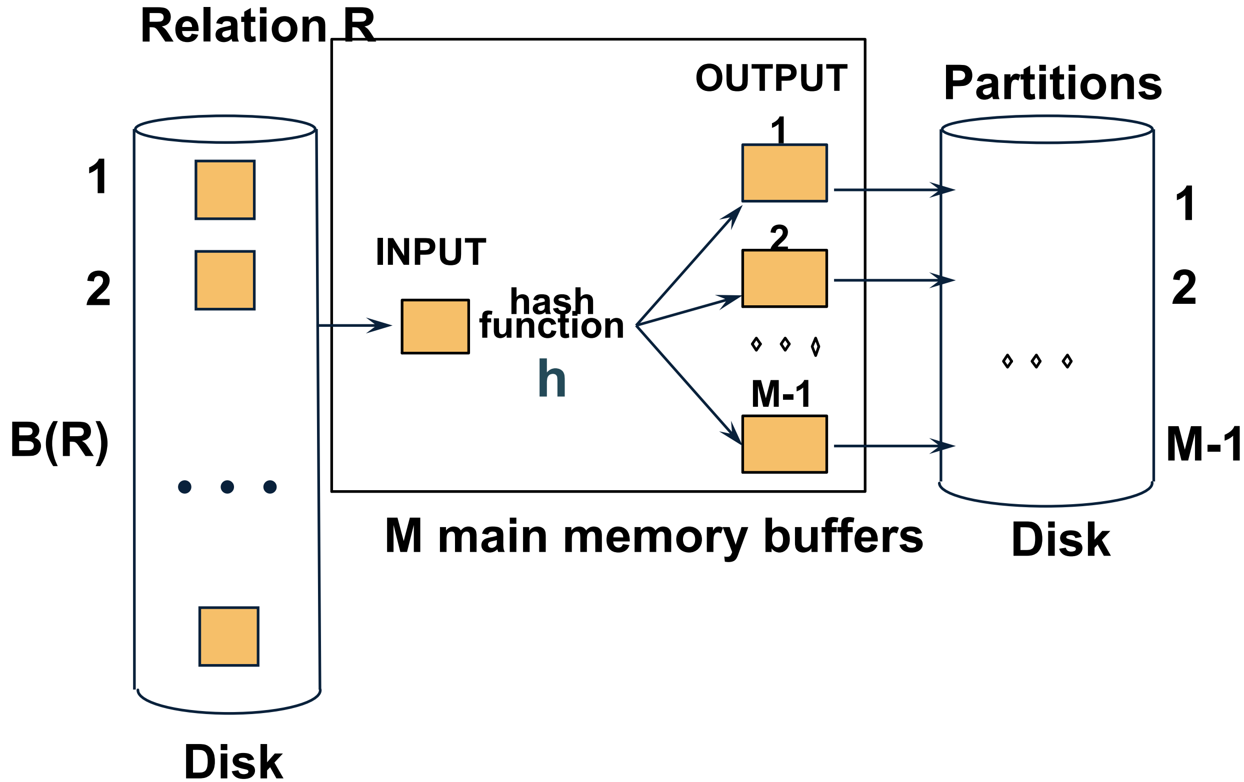
Step 1: Hash-partition

- Partition R into buckets, on disk



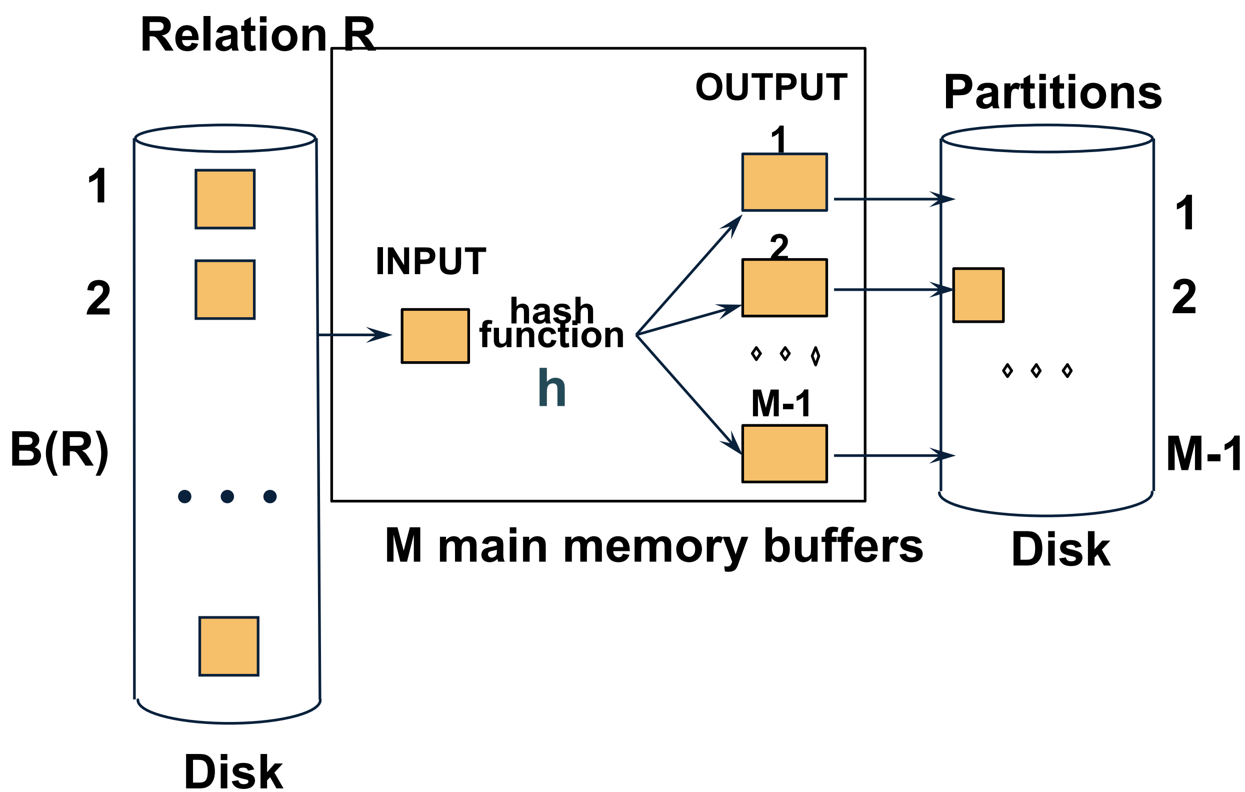
Step 1: Hash-partition

- Partition R into buckets, on disk



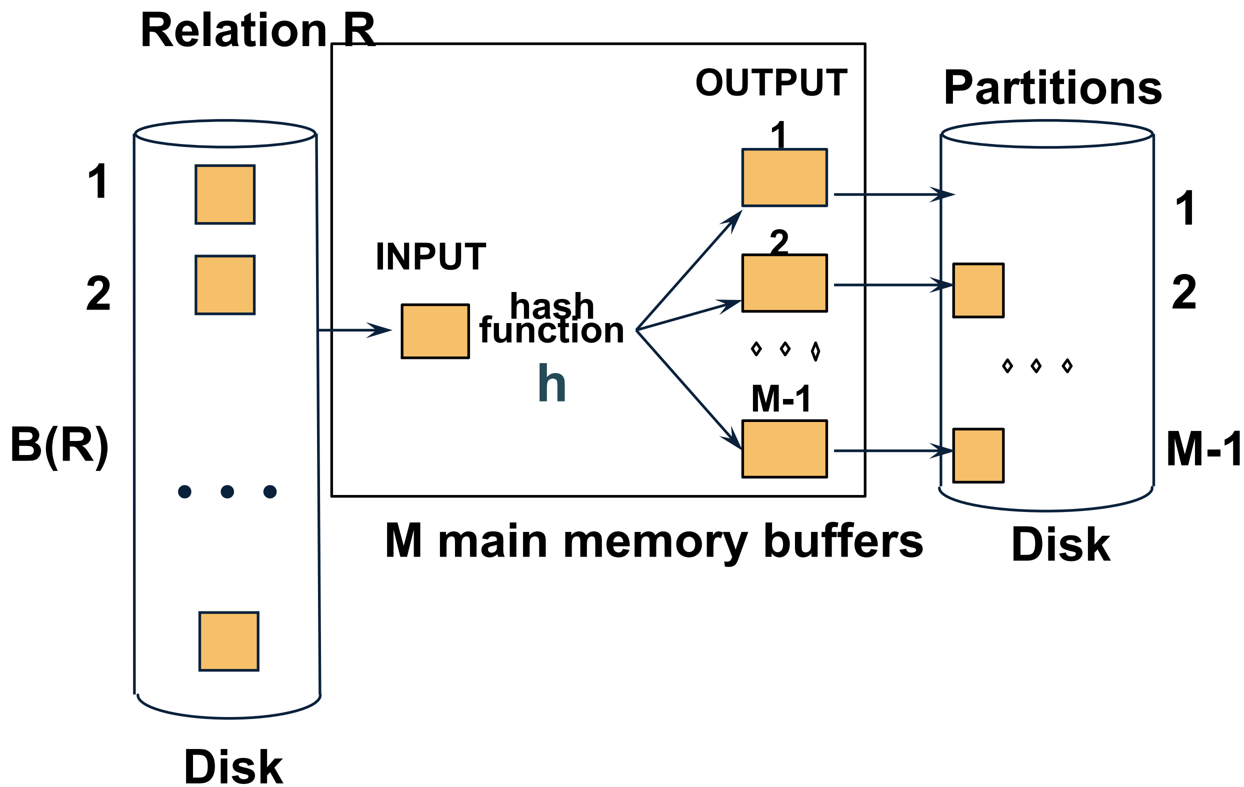
Step 1: Hash-partition

- Partition R into buckets, on disk



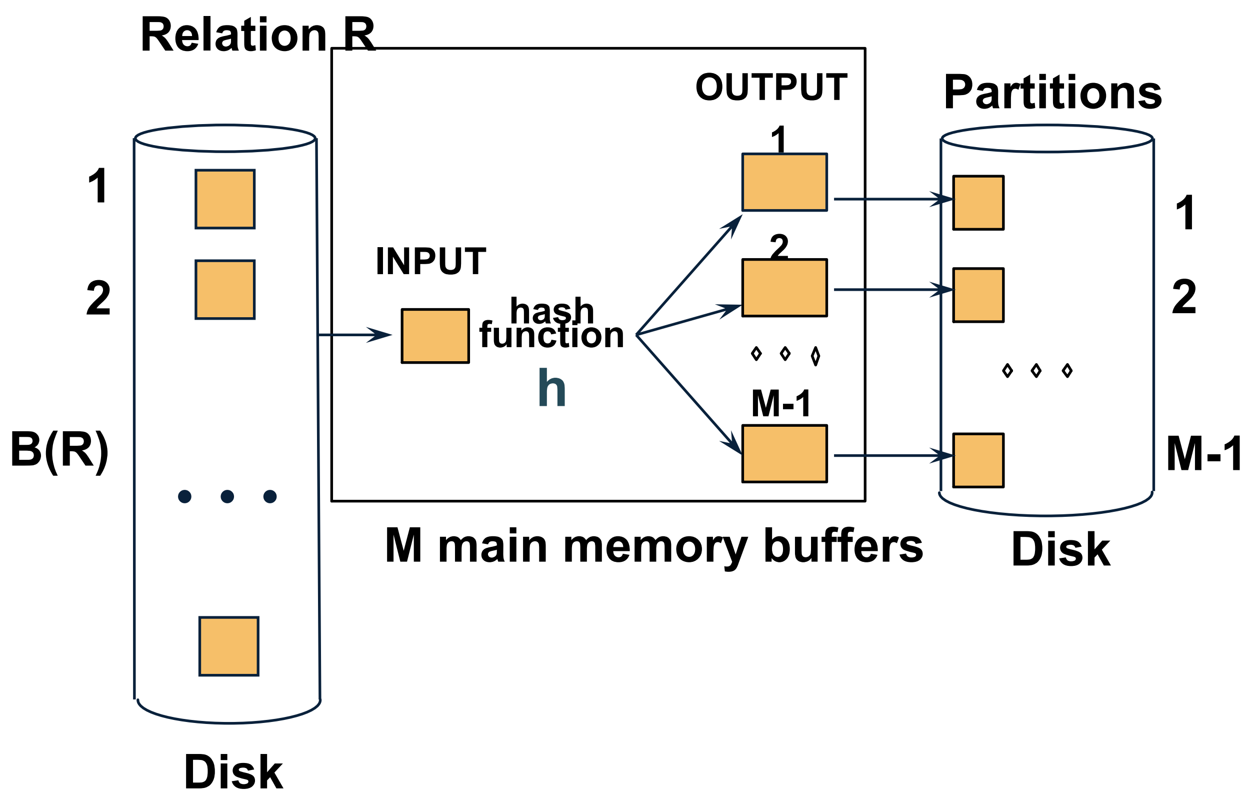
Step 1: Hash-partition

- Partition R into buckets, on disk



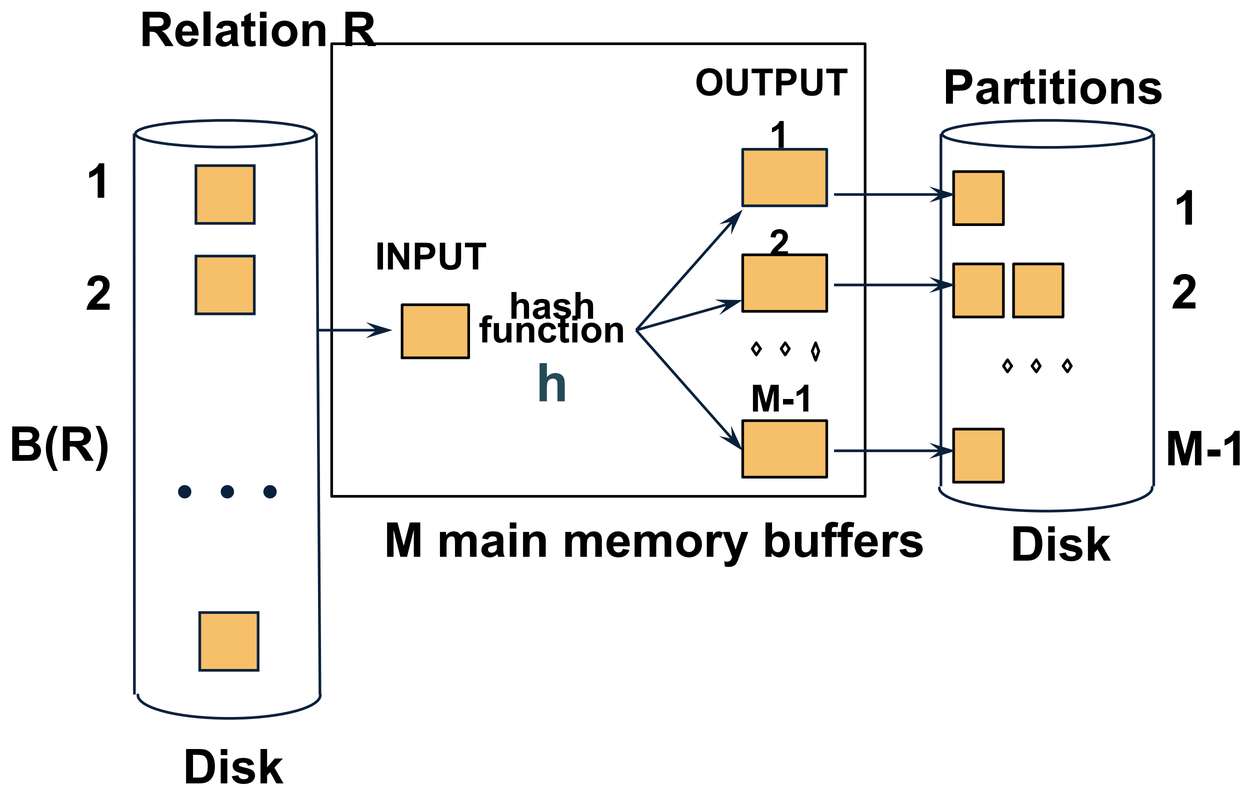
Step 1: Hash-partition

- Partition R into buckets, on disk



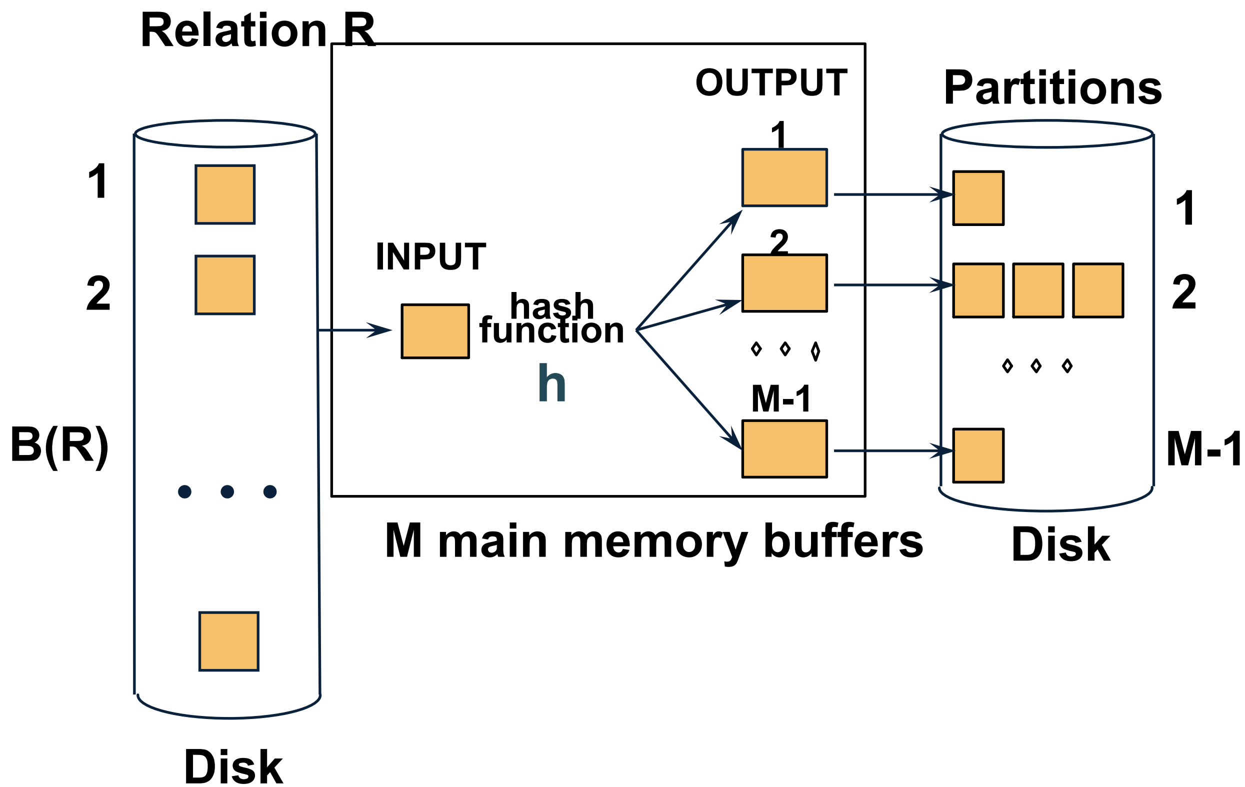
Step 1: Hash-partition

- Partition R into buckets, on disk



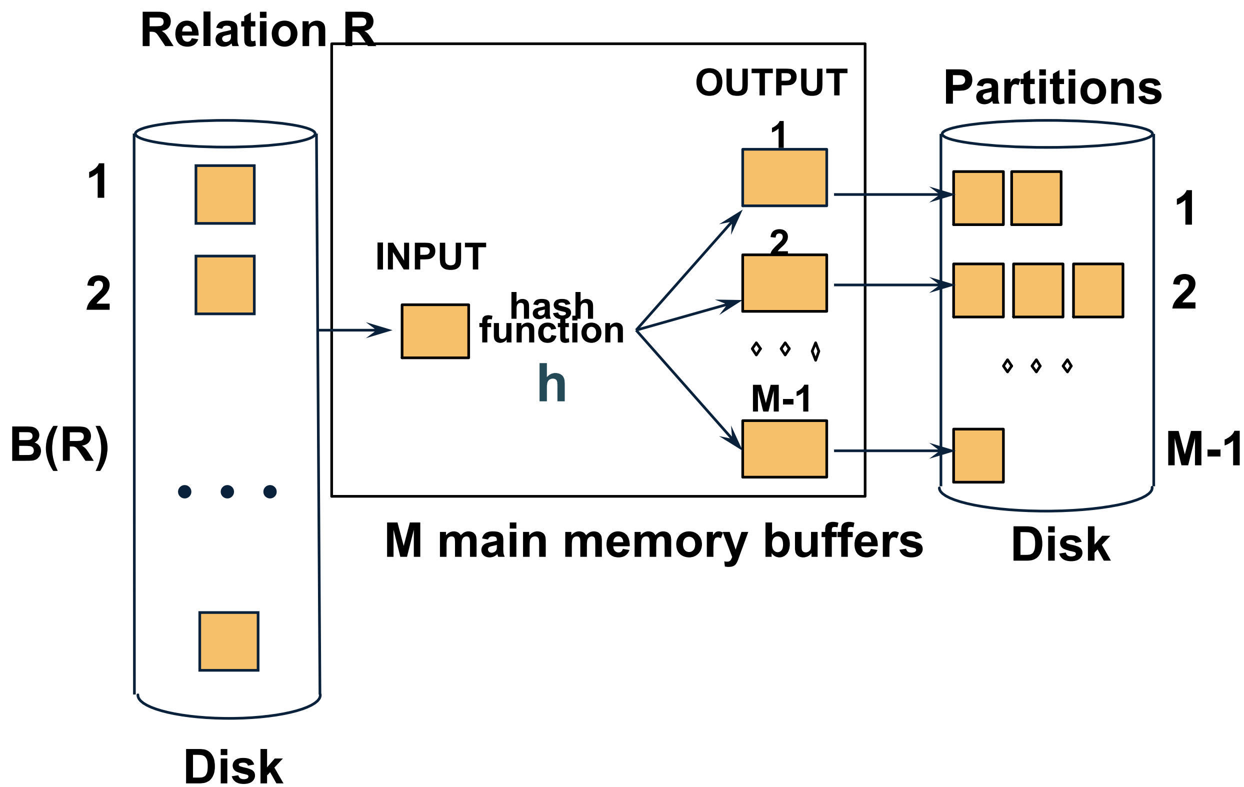
Step 1: Hash-partition

- Partition R into buckets, on disk



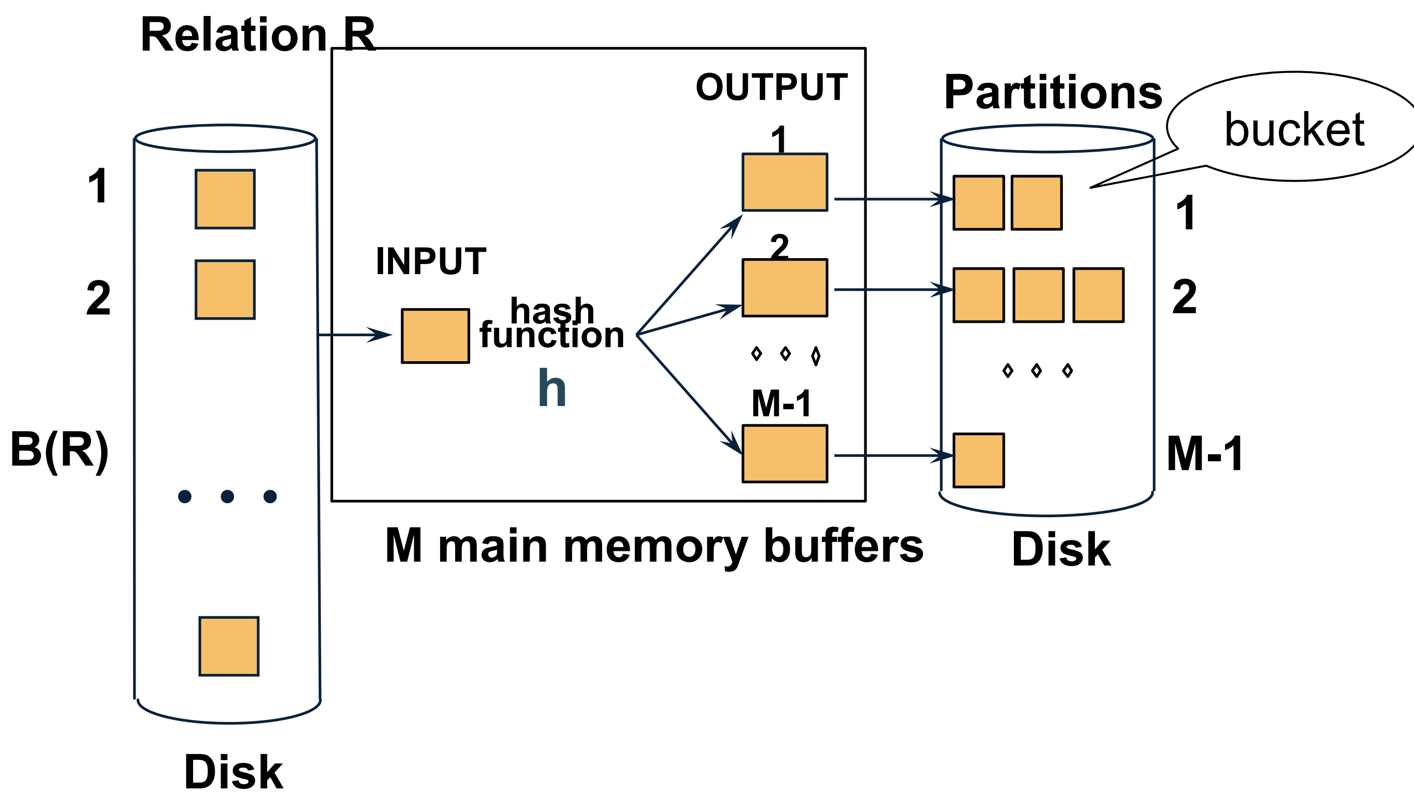
Step 1: Hash-partition

- Partition R into buckets, on disk



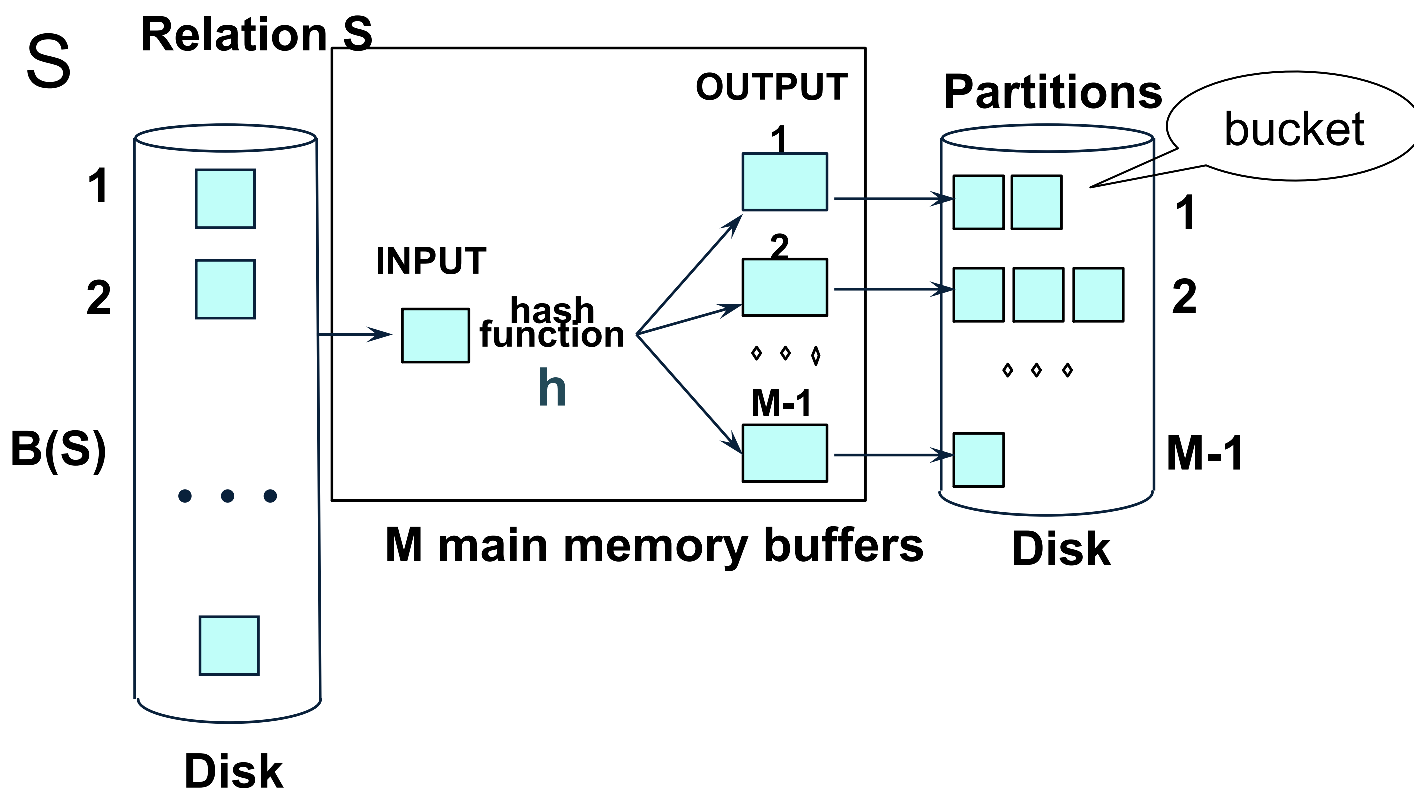
Step 1: Hash-partition

- Partition R into buckets, on disk



Step 1: Hash-partition

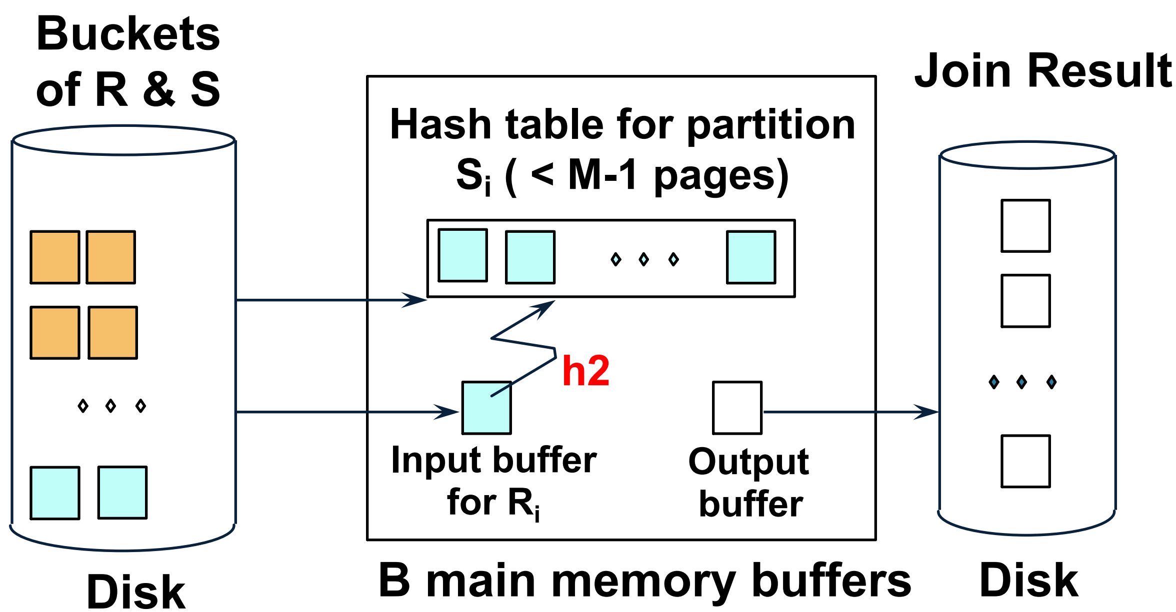
- Partition R into buckets, on disk
- Partition S



Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)

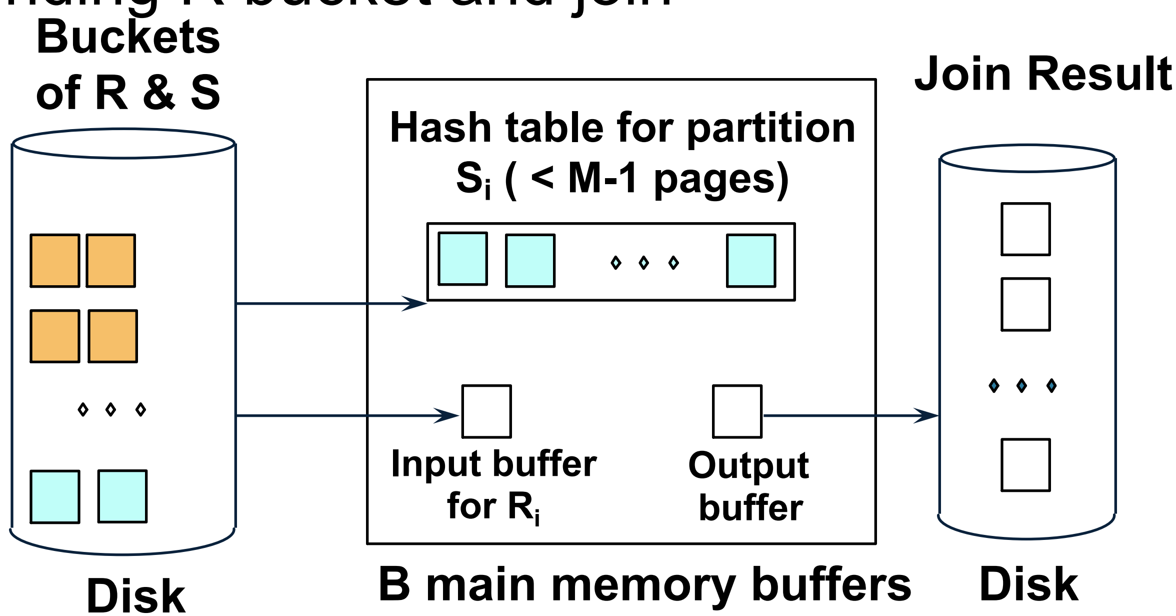


One entire S-bucket fits in M if B(S) / M ≤ M,
or B(S) ≤ M^2. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join

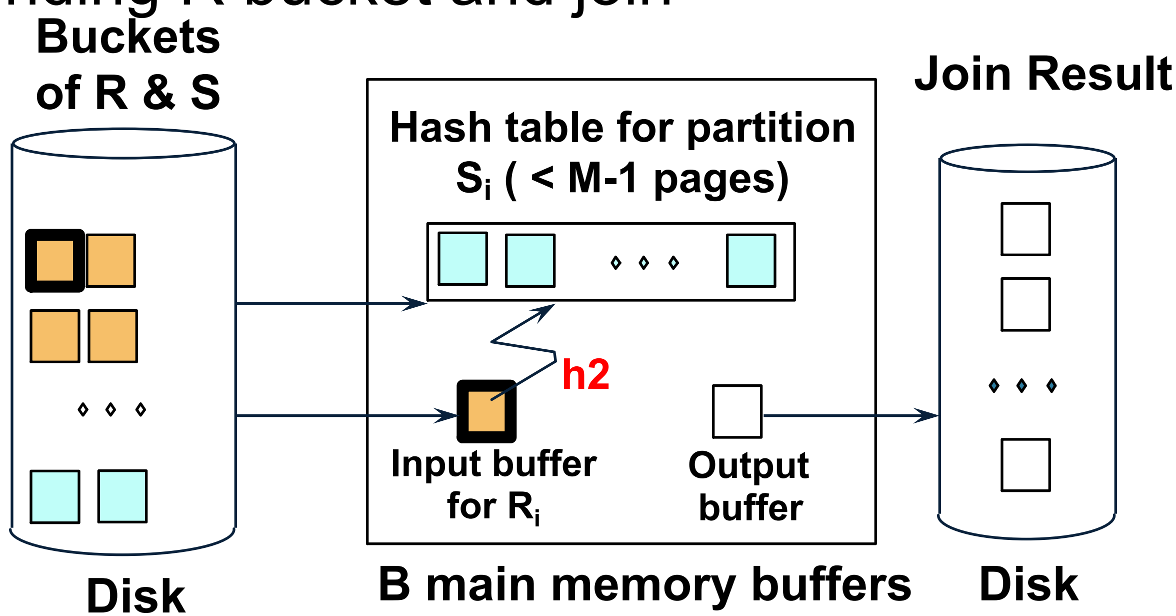


One entire S-bucket fits in M if $B(S) / M \leq M$, or $B(S) \leq M^2$. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join

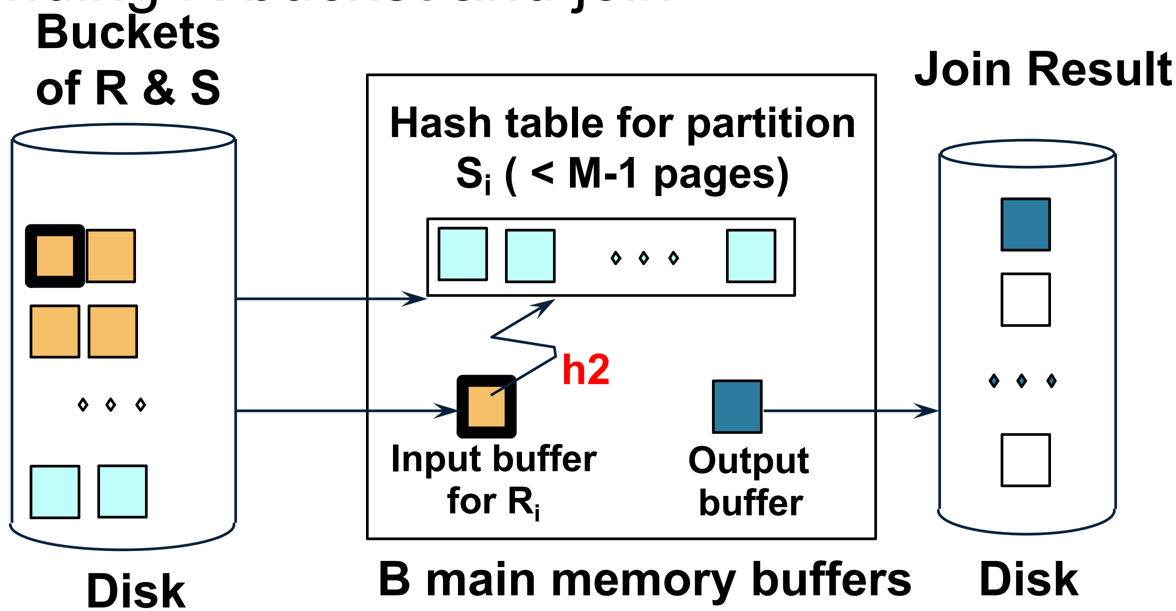


One entire S-bucket fits in M if B(S) / M ≤ M, or $B(S) \le M^2$. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join

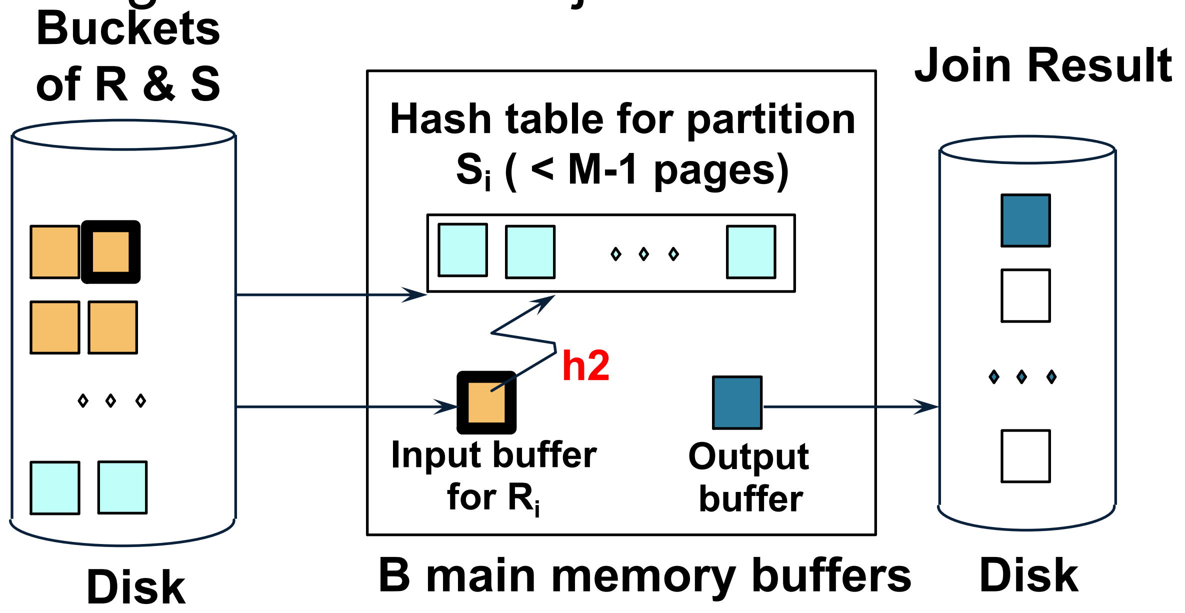


One entire S-bucket fits in M if B(S) / M ≤ M, or B(S) ≤ M^2. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join

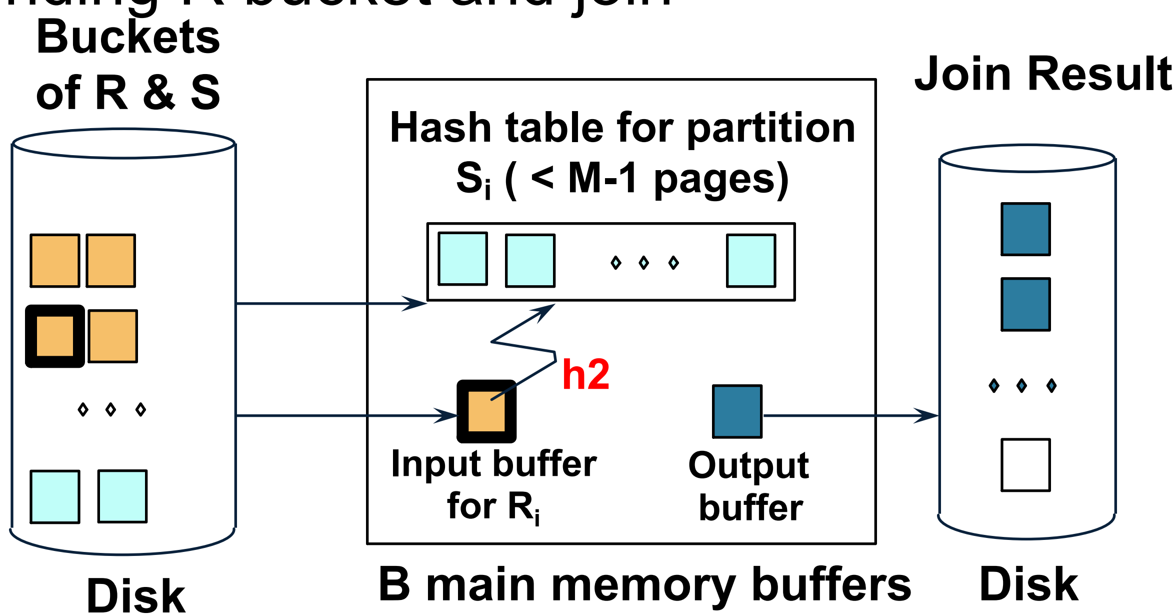


One entire S-bucket fits in M if $B(S) / M \leq M$, or $B(S) \leq M^2$. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join

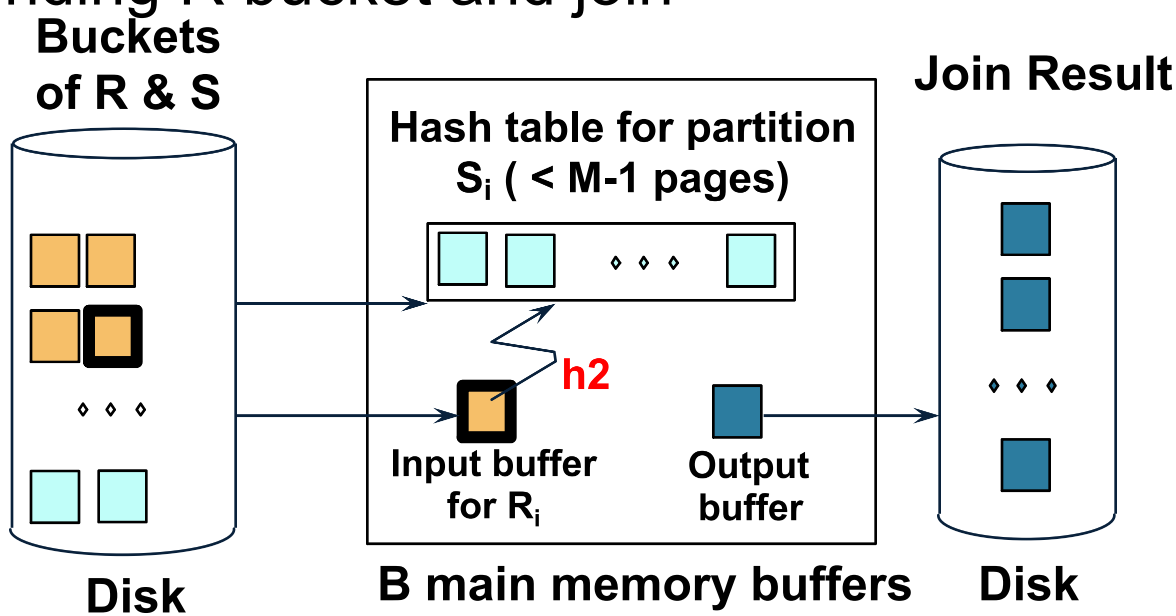


One entire S-bucket fits in M if B(S) / M ≤ M, or B(S) ≤ M^2. WHY?

Step 2: Join Buckets

R ⋈ S

- Read one S-backed; hash-partition it using h2 (≠ h)
- Scan corresponding R bucket and join



One entire S-bucket fits in M if B(S) / M ≤ M, or B(S) ≤ M^2. WHY?

Partitioned Hash Join

- Cost: 3B(R) + 3B(S)
- Assumption: $\min(B(R), B(S)) \leq M^2$

Hybrid Hash Join Algorithm

- Assume we have extra memory available

- Partition S into k buckets

  t buckets S_1 , …, S_t stay in memory

  k-t buckets S_{t+1}, …, S_k to disk

- Partition R into k buckets
  - First t buckets join immediately with S
  - Rest k-t buckets go to disk

- Finally, join k-t pairs of buckets:

  (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$
- Thus: $t/k * B(S) + k-t \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$
- Thus: $t/k * B(S) + k-t \leq M$

Assuming $t/k * B(S) \gg k-t$: $t/k = M/B(S)$

Hybrid Hash Join Algorithm

- How many I/Os ?

- Cost of partitioned hash join: 3B(R) + 3B(S)

- Hybrid join saves 2 I/Os for a t/k fraction of buckets
- Hybrid join saves   2t/k(B(R) + B(S))   I/Os

Cost: (3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

External Memory Algorithms

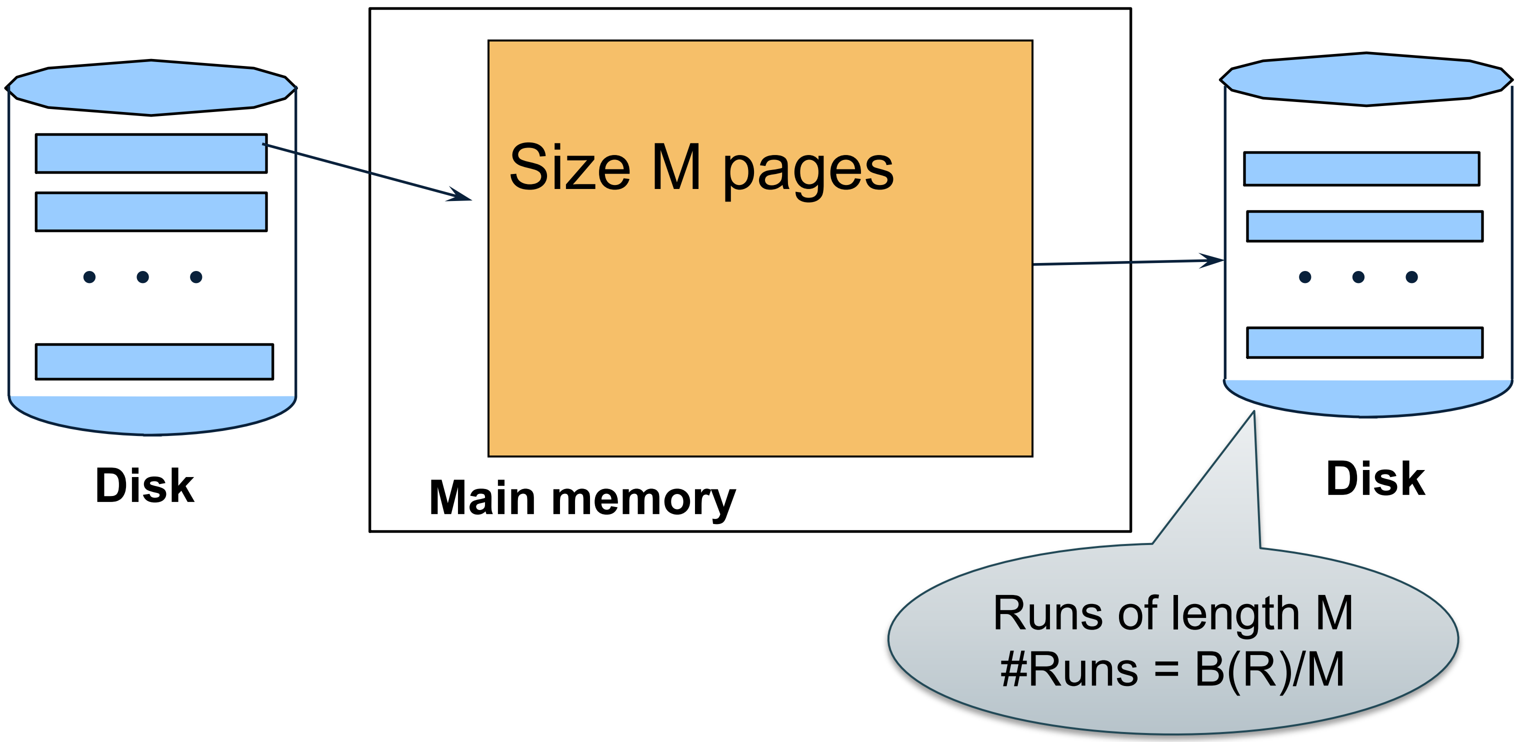
- Selection and index-join
- Nested loop join
- Partitioned hash-join, a.k.a. grace join
- Merge-join

Merge-Sort

- Problem: Sort a file of size B with memory M
- Will discuss only 2-pass sorting, for when $B \leq M^2$

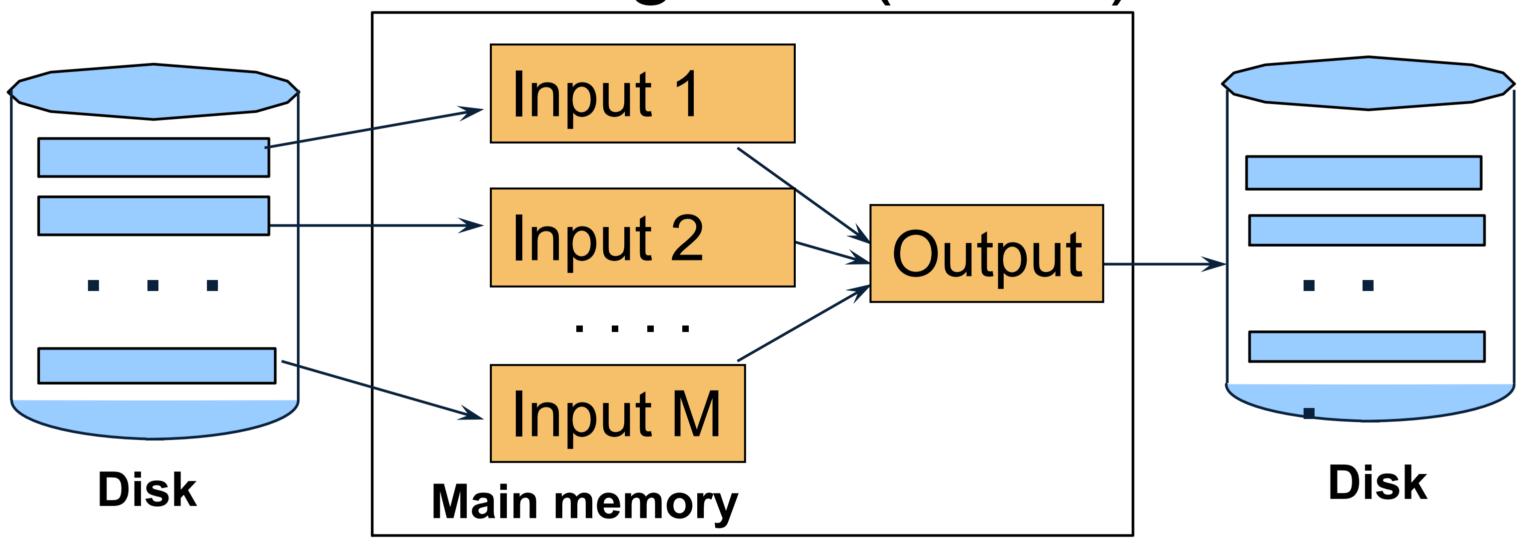
Merge-Sort: Step 1

- Phase one: load M pages in memory, sort



Merge-Sort: Step 2

- Merge M – 1 runs into a new run
- Result: runs of length M (M – 1) $\approx M^2$



Assuming $B \leq M^2$, we are done

Merge-Sort

- Cost:
  - Read+write+read = 3B(R)
  - Assumption: B(R) <= M^2

- Other considerations
  - In general, a lot of optimizations are possible

Summary

- Three EM join algorithms:
  - Nested loop join
  - Hash-partitioned aka Grace Join
  - Merge join
- Easy adaptation to other operators:
  - Group-by, union, difference
- 2 pass can be extended to N pass