Lecture 10:
HW3 Overview & DB Components

Guest lecturer: Kyle Deeds

Announcements

- HW 2 is due tonight at midnight
- HW 3 will be released today at noon
- My OH this week are pushed to Monday (2/12) at 11:30 in Gates 274

Agenda

- High Level Homework Structure
  - What is SimpleDB?
  - Code environment
- Implementation Pieces
  - Database Catalog
  - Buffer Pool Manager
  - Heap File/Page
  - The Tuple Iterator Model
    - Scan
    - Filter
    - Join

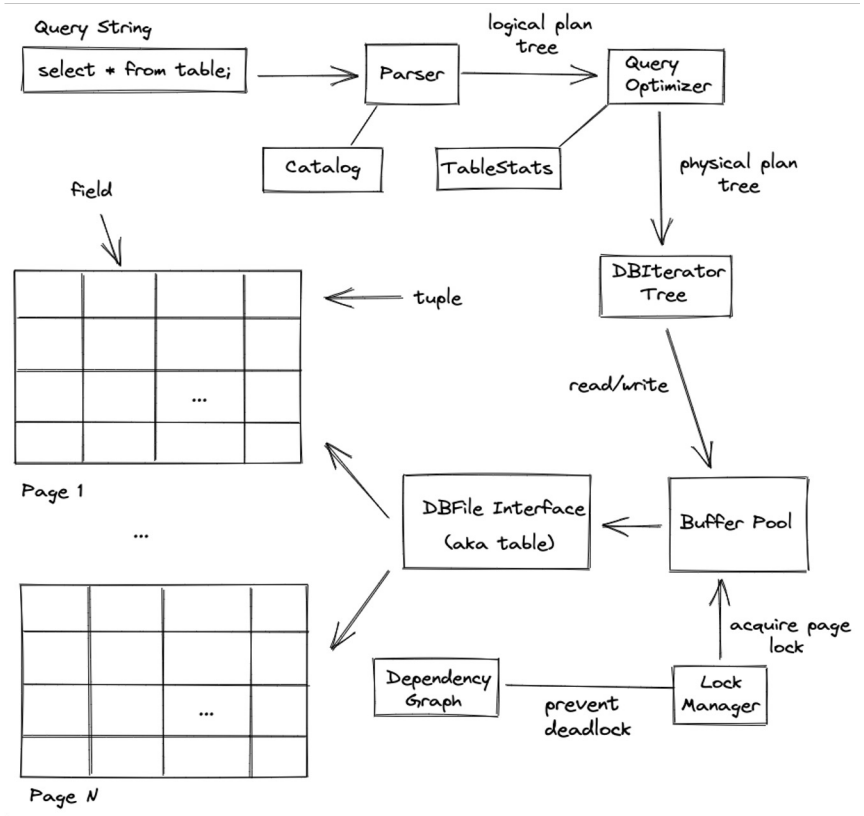
SimpleDB

SimpleDB is a simple database system:

- Originally intended for MIT's intro database course
- Written in Java
- Disk-based, row-oriented, single-threaded

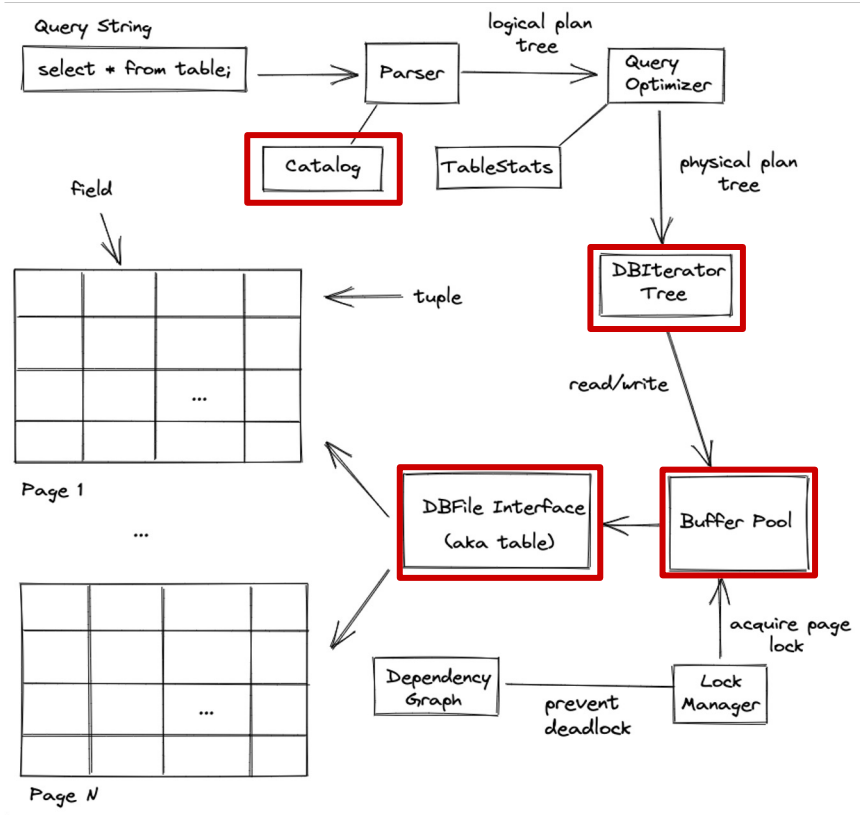
If you're not familiar with Java, you should to do a short online tutorial to get (re)acquainted! (One is linked in the HW)

SimpleDB System Overview



https://github.com/awelm/simpledb?tab=readme-ov-file

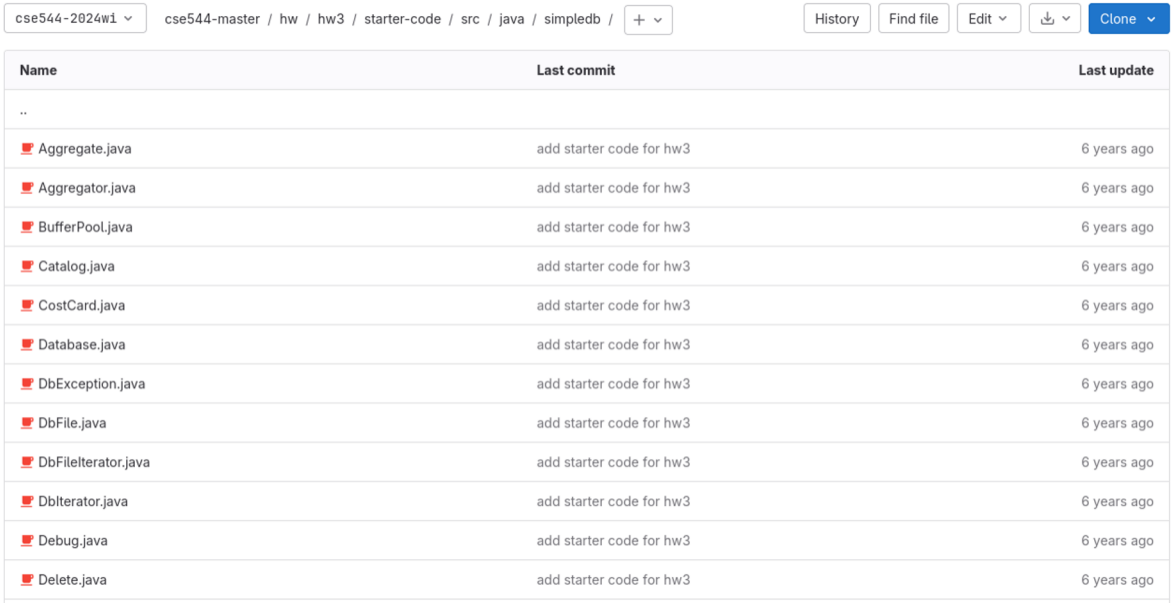
SimpleDB System Overview



https://github.com/awelm/simpledb?tab=readme-ov-file

Code Environment

The code is written as a series of java files in the “starter-code” folder of HW3.

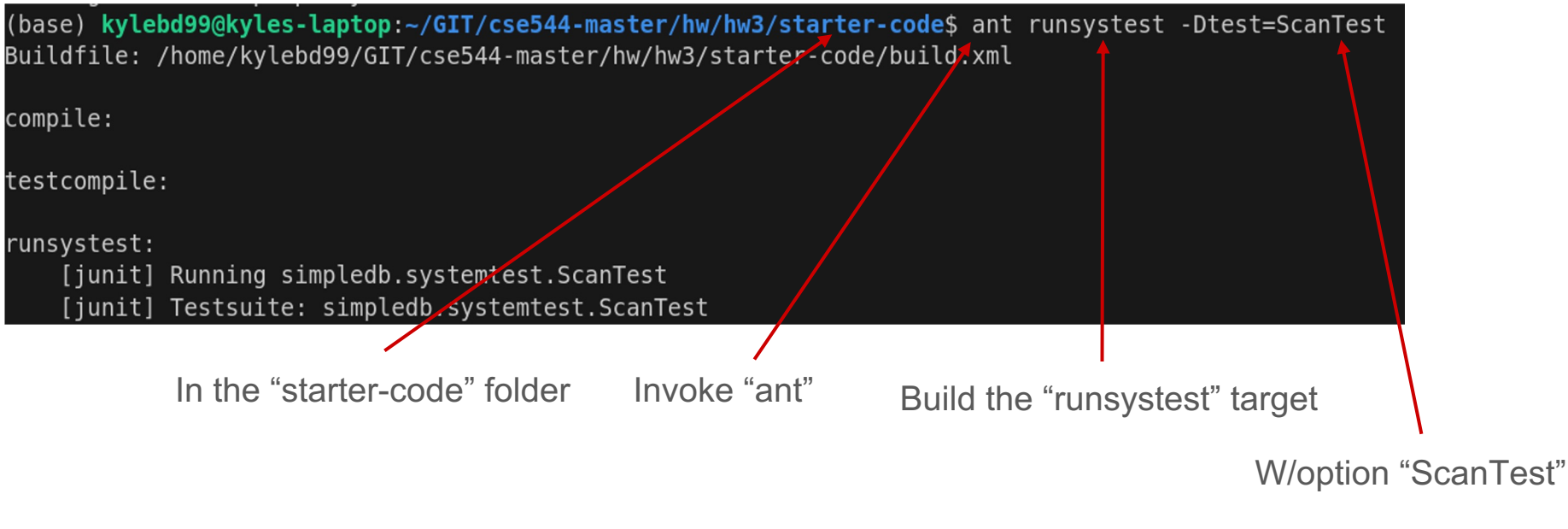


Code Environment

The code is written as a series of java files in the “starter-code” folder of HW3.

We will compile & run the code using “ant”, a java build tool:

```
(base) kylebd99@kyles-laptop:~/GIT/cse544-master/hw/hw3/starter-code$ ant runsystest -Dtest=ScanTest
Buildfile: /home/kylebd99/GIT/cse544-master/hw/hw3/starter-code/build.xml

compile:

testcompile:

runsystest:
    [junit] Running simpledb.systemtest.ScanTest
    [junit] Testsuite: simpledb.systemtest.ScanTest
```



How to Edit the Code

In the Readme.md for the HW, you'll be pointed to files in the codebase with sections which are incomplete,

```
public void addTable(DbFile file, String name, String pkeyField) {
    // some code goes here
}
```

The assignment consists of filling these sections in with the appropriate logic. To do this you may want to:

- Define new functions
- Add attributes to classes
- Import data structures from the standard java library (e.g. a hash table)

DB Catalog

What is a Catalog?

Catalogs are a general term for data structures which store metadata in a DBMS,

- The list of databases
- Database to table mapping
- Table to column mapping
- Table to index mapping
- Table to statistics mapping
- User security privileges

In most systems, these catalogs are themselves stored as tables!

SimpleDB Catalog

In SimpleDB, the catalog only stores a list of tables,

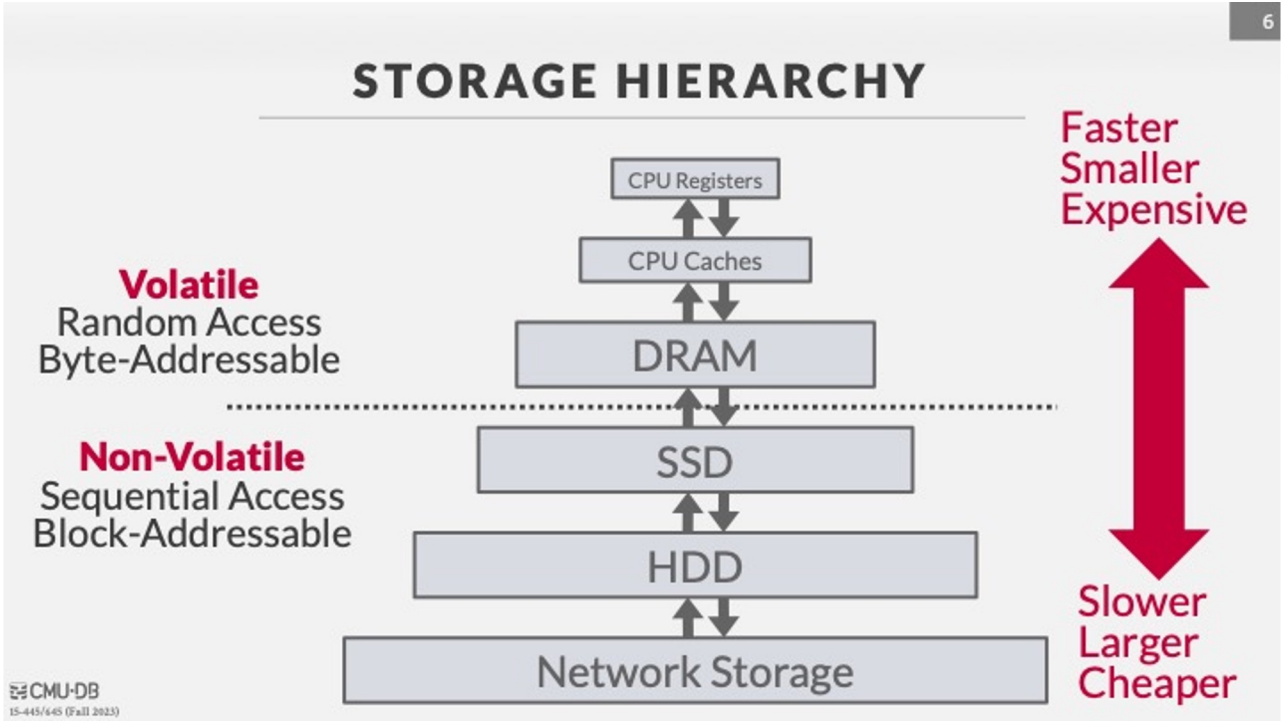
```
public class Catalog {

    /**
     * Constructor.
     * Creates a new, empty catalog.
     */
    public Catalog() {
        // some code goes here
    }
```

You'll implement:

- The data structure for storing table information
- Adding/removing a table
- Getting information about a table
- Iterating through the tables in the database

Buffer Pools

Recap: The Problem of Storage



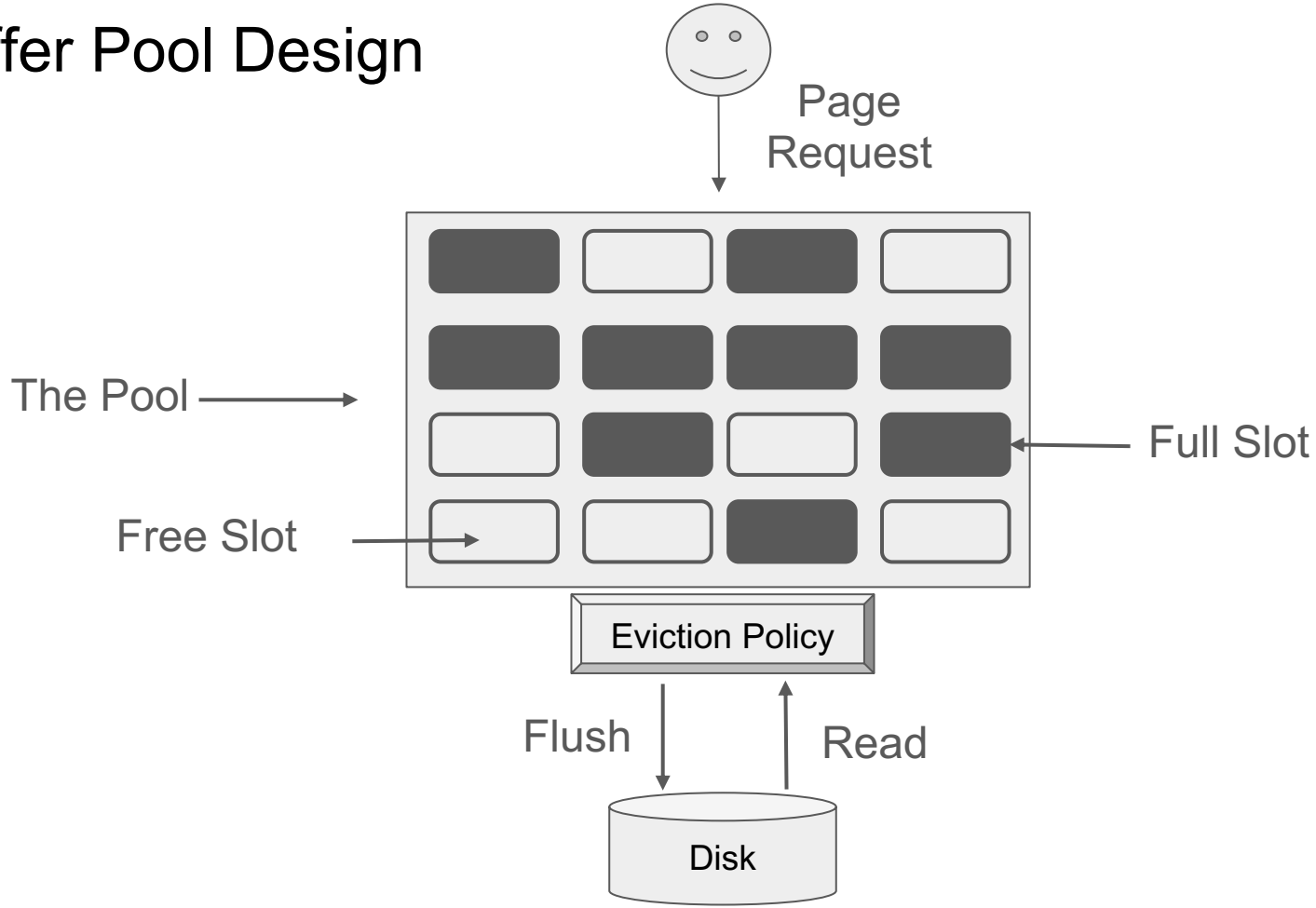
The Buffer Pool

A buffer pool is a space in main memory which caches pages that have been retrieved from disk.

When a disk page is requested, the buffer pool

1. Checks whether the page is already in the pool
    a. If so, immediately returns the page
2. Checks whether the pool is full
    a. If so, consults the eviction policy and flushes a page to the disk
3. Loads the page from disk

Buffer Pool Design



Eviction Policies

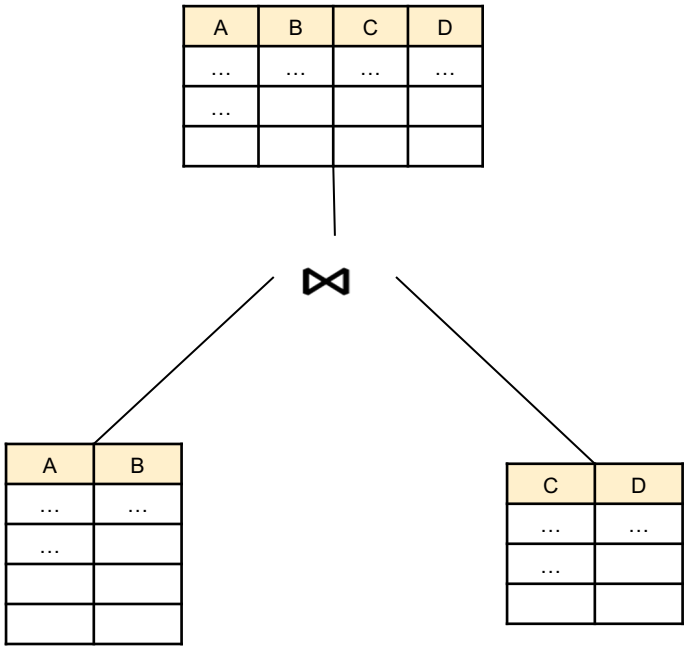
The eviction policy determines which page needs to be flushed to disk when the buffer pool fills up. Potential algorithms for this:

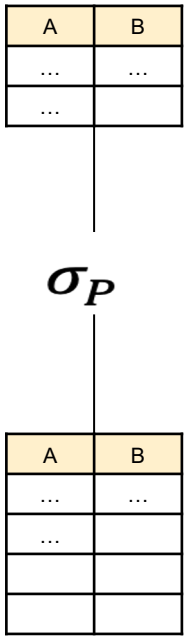
- LRU: Least Recently Used
- FIFO: First In First Out
- Clock: Circular Eviction
- MRU: Most Recently Used

5 Minute Discussion: How would you design a better eviction policy for a DBMS?

The Iterator Model
(i.e. “The Pull Model”)

Set-at-a-time Operators

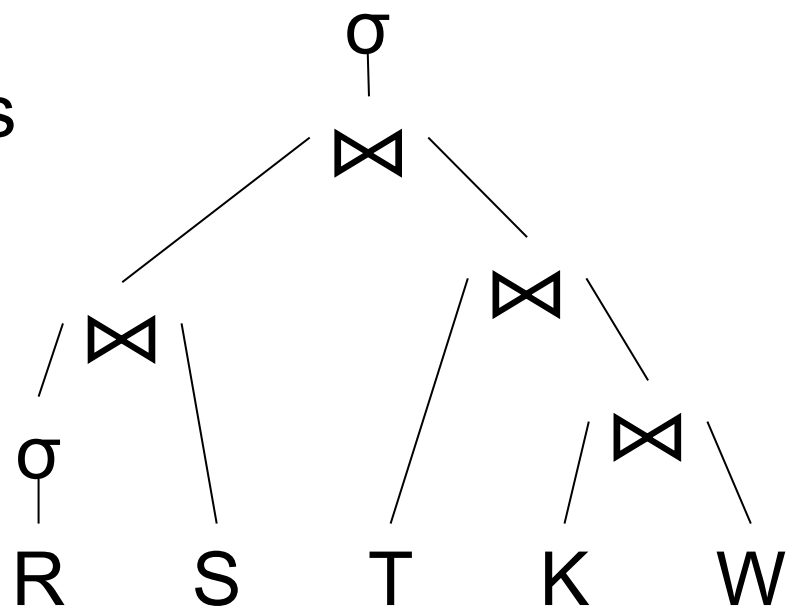




How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
treat operators as iterators



Operator Interface

Volcano model:

- open(), next(), close()
- Pull model
- Supported by most DBMS today
- Will discuss next

Newer alternative: push model (won’t discuss)

Operator Interface

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

Iterator Model
A.k.a. Volcano-style execution

```
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 ();
}
```

Iterator Model
A.k.a. Volcano-style execution

```
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();
    }
```

Iterator Model
A.k.a. Volcano-style execution

```
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;
  }
```

Iterator Model
A.k.a. Volcano-style execution

```
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(); }
}
```

Iterator Model
A.k.a. Volcano-style execution

Query plan execution

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

Iterator Model
A.k.a. Volcano-style execution

Query plan execution

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

q.open();
```

Iterator Model
A.k.a. Volcano-style execution

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
}
```

Iterator Model
A.k.a. Volcano-style execution

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();
```

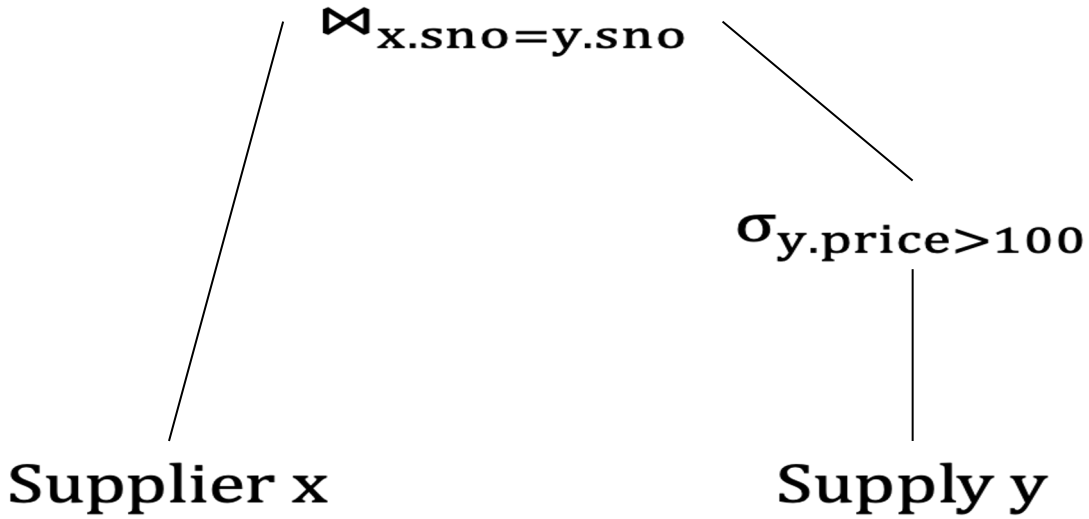
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



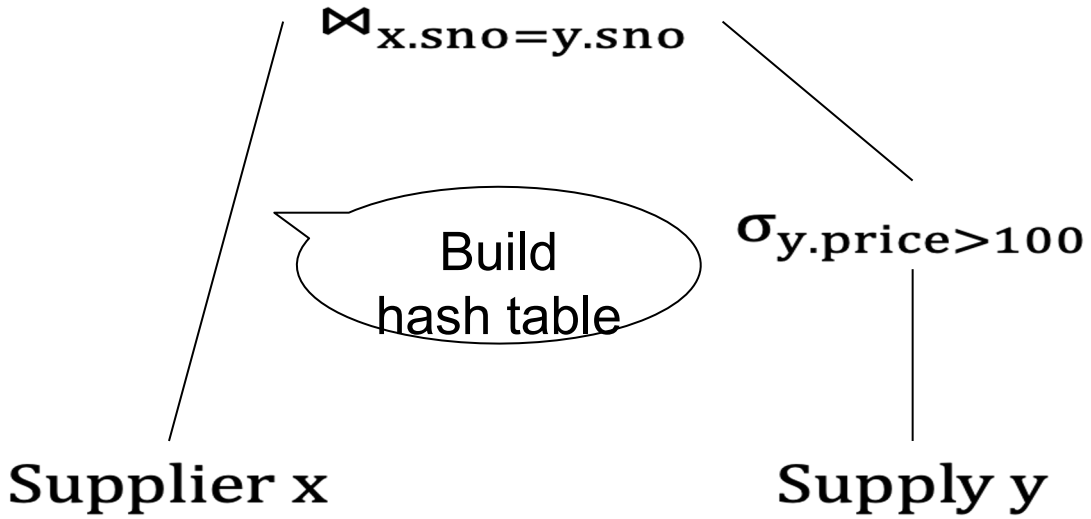
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



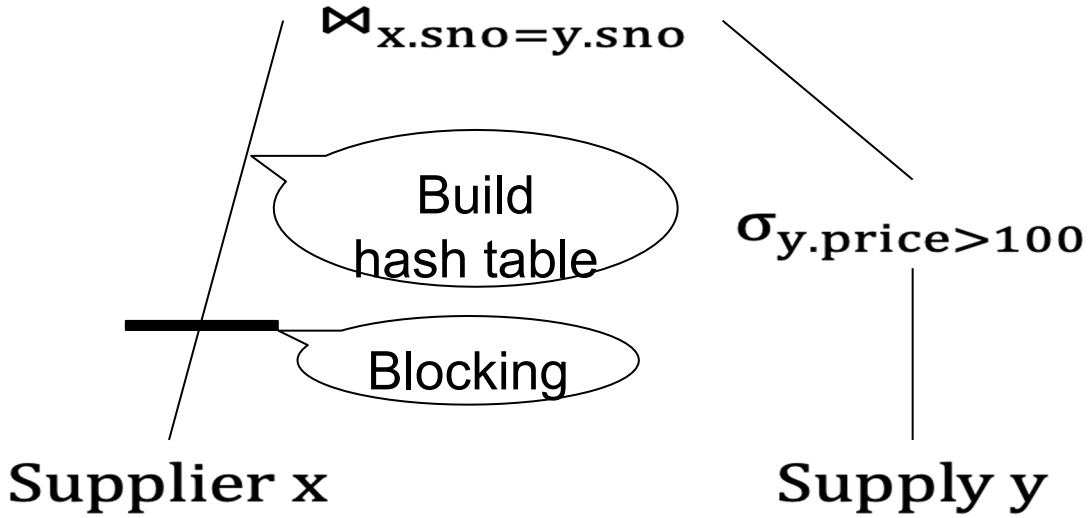
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



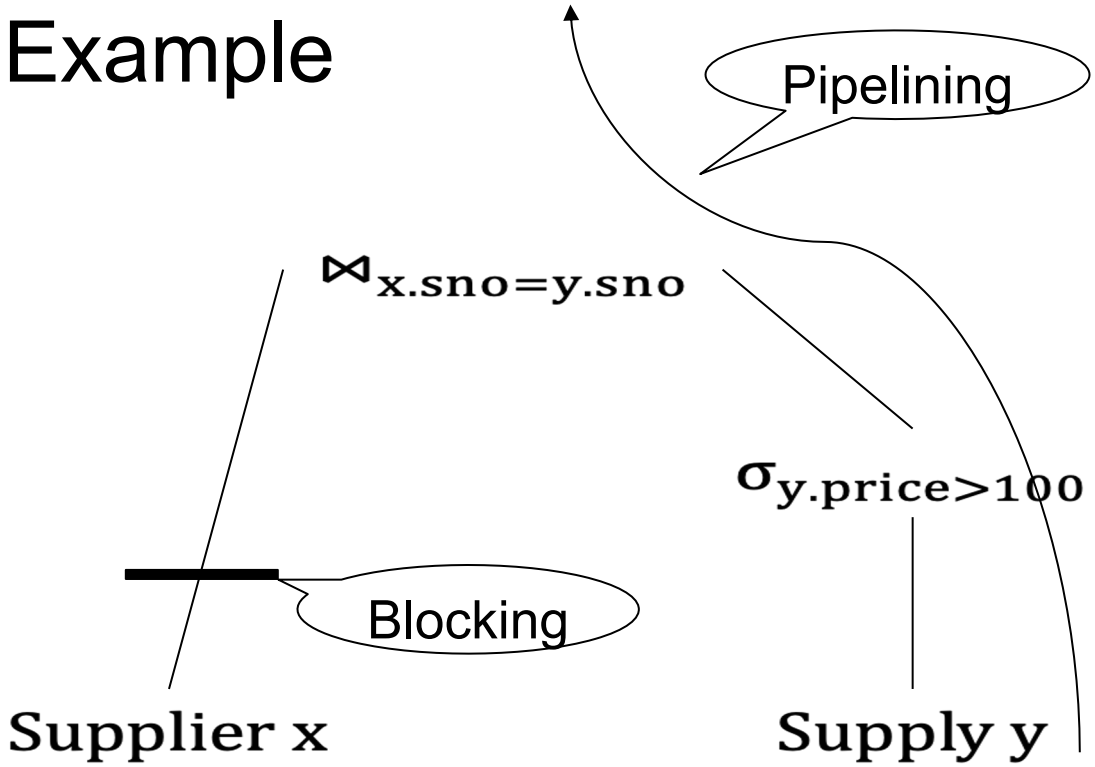
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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);
```

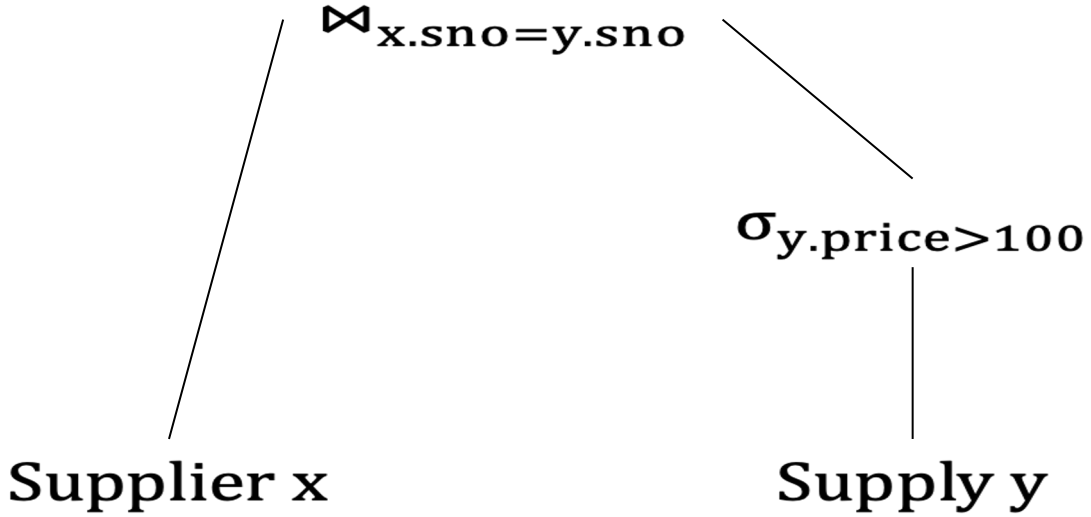
Pipelining changes the order significantly



Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Example

Details



”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 order significantly

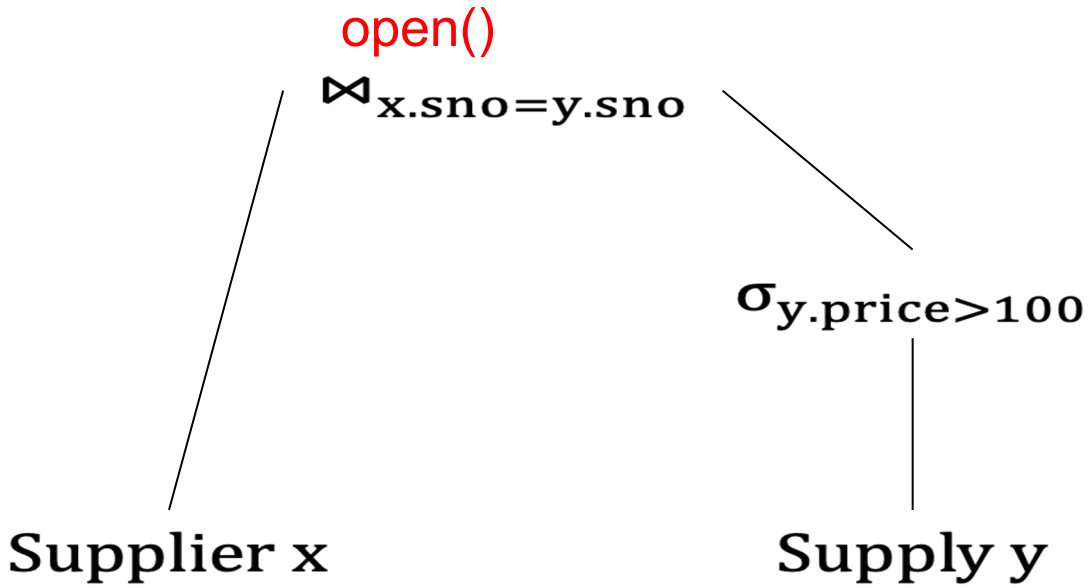
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



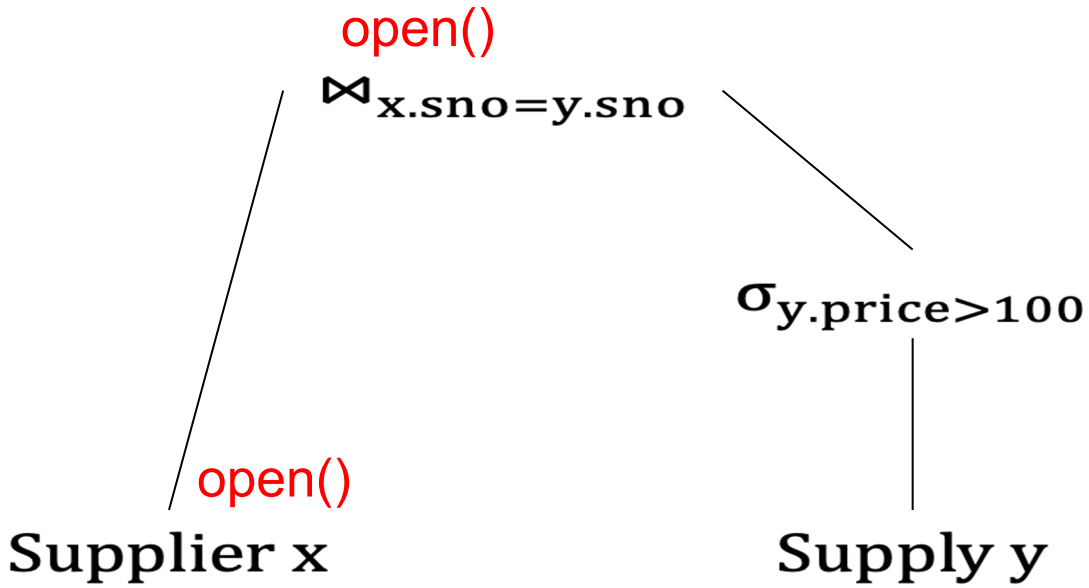
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



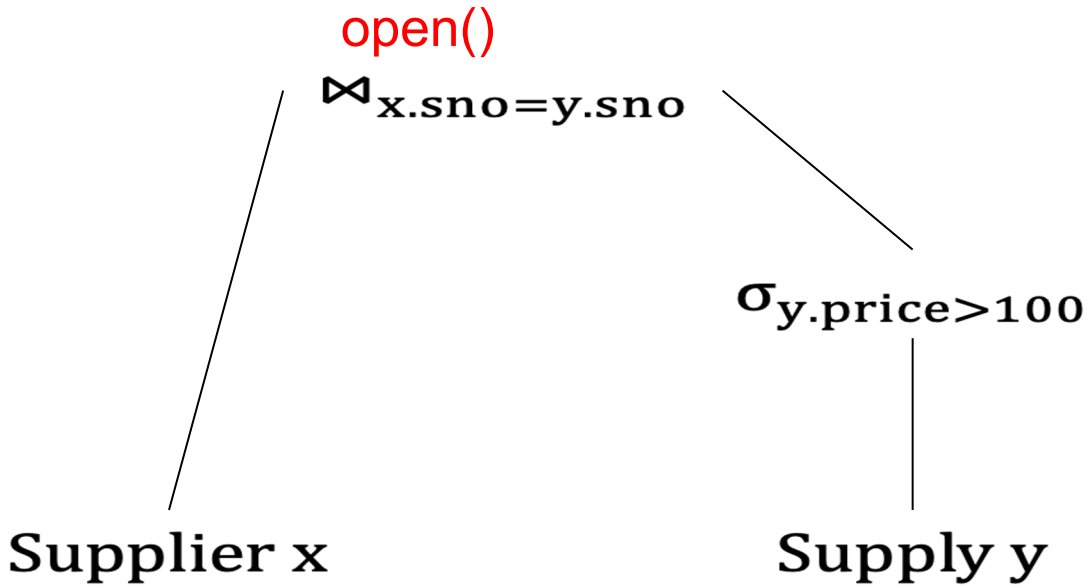
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



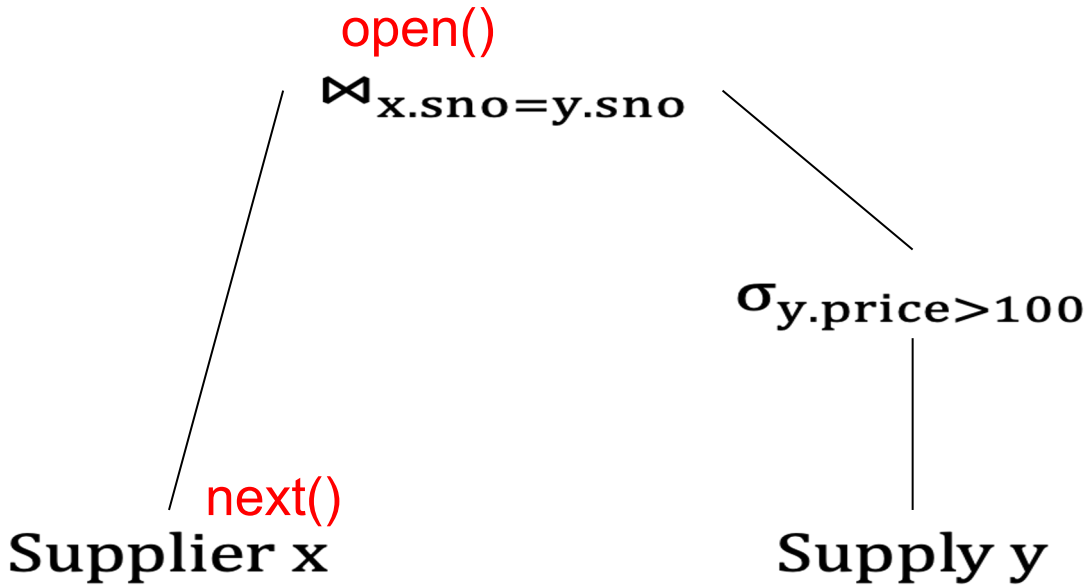
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



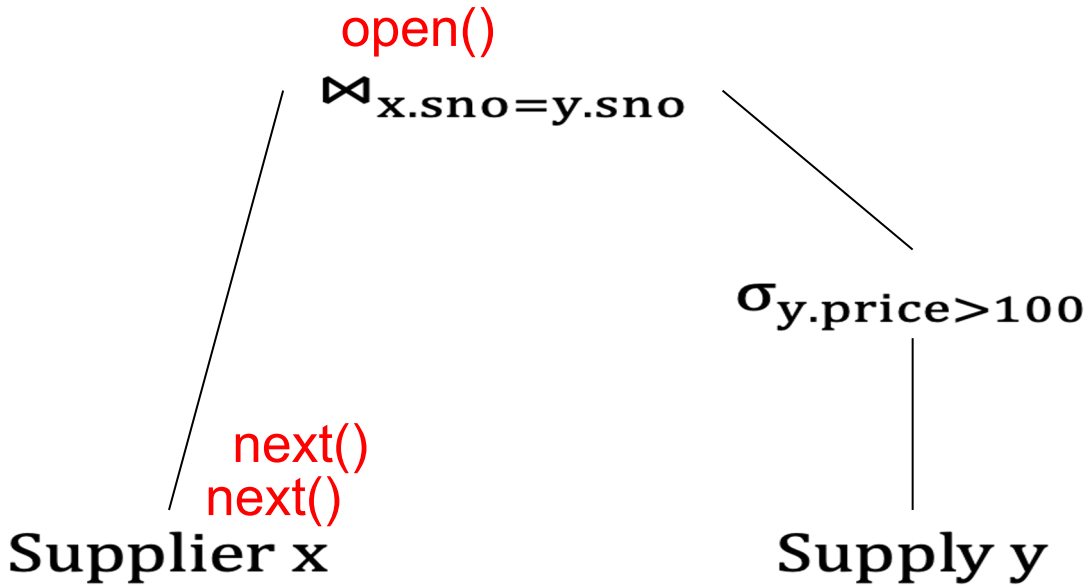
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



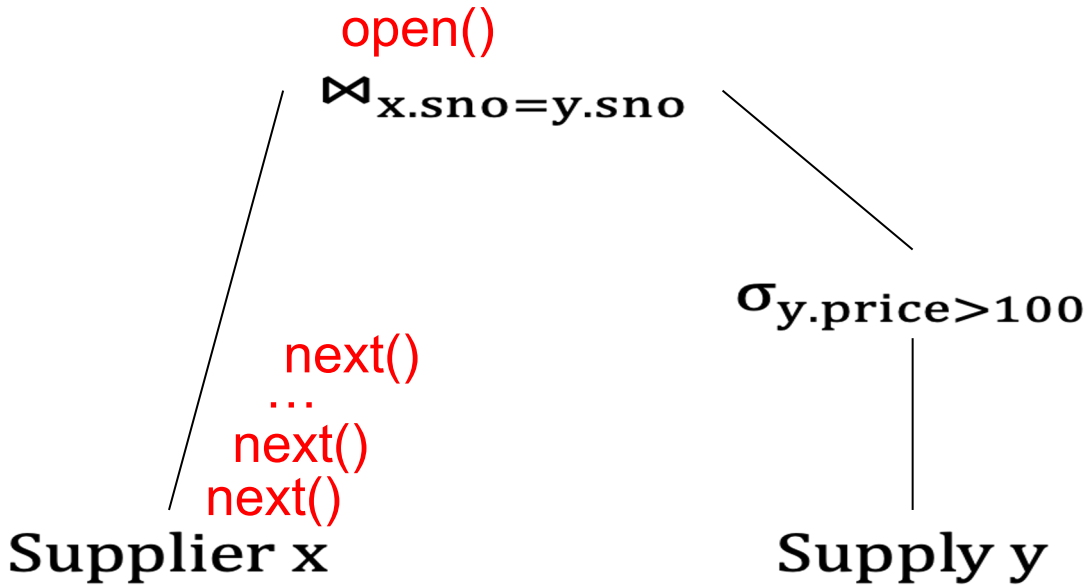
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



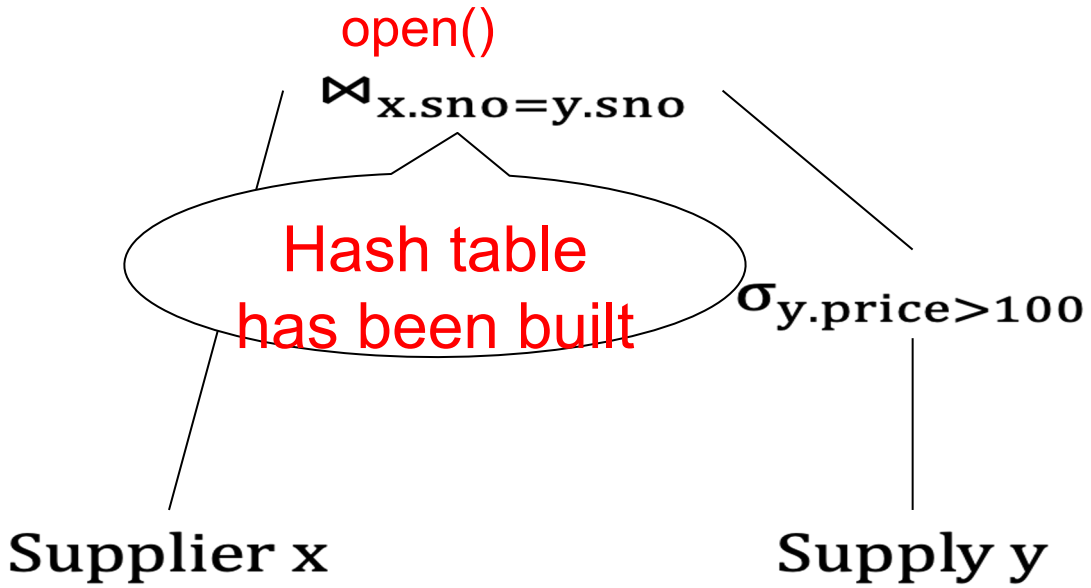
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



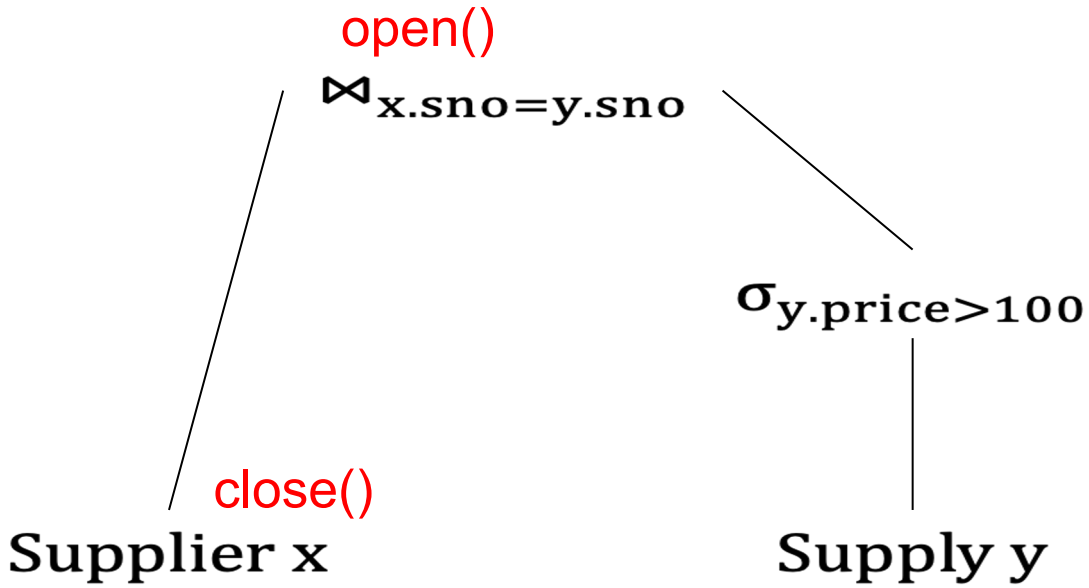
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



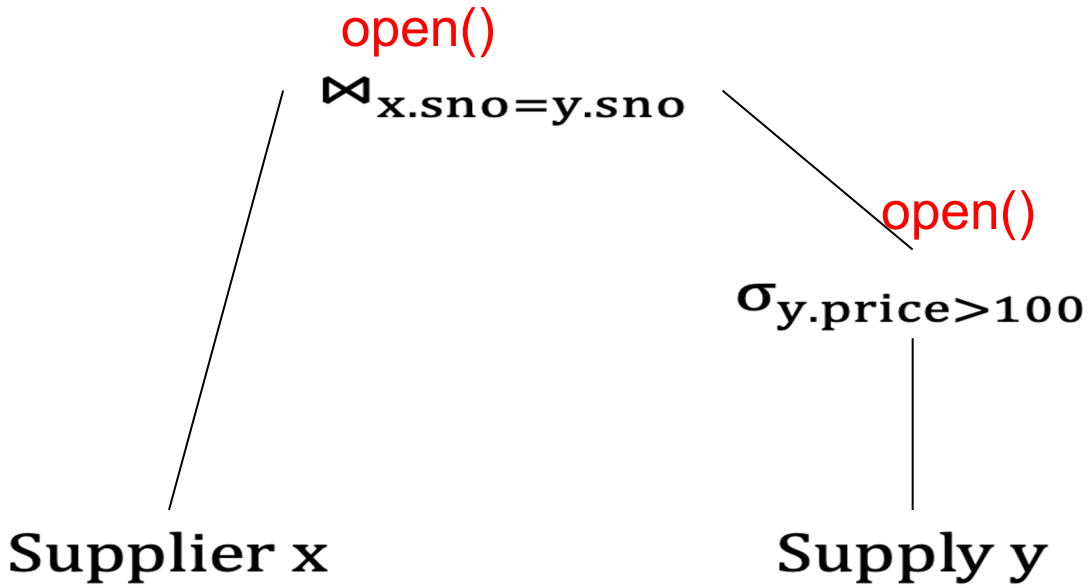
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



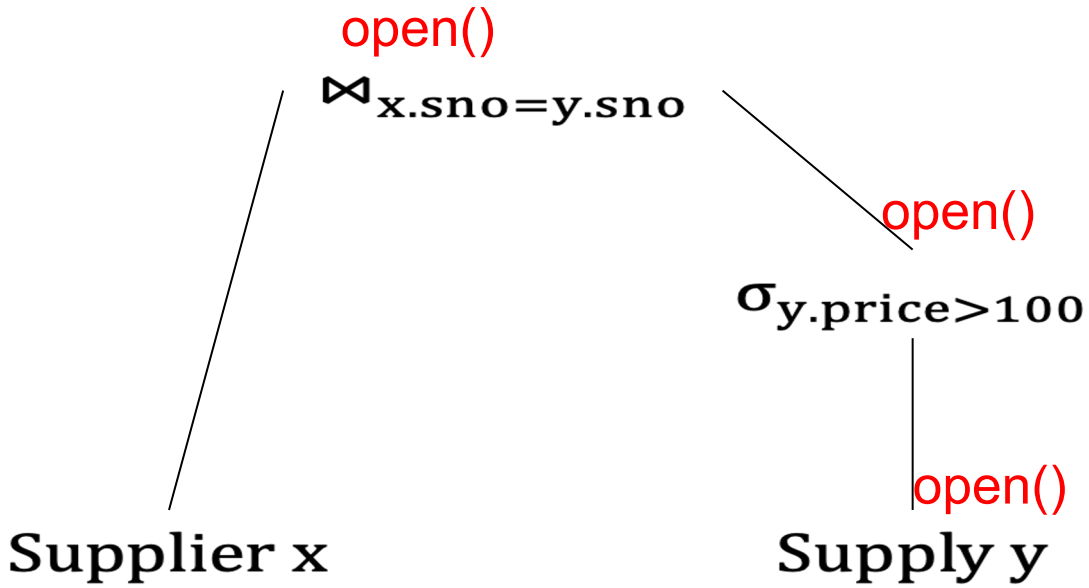
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



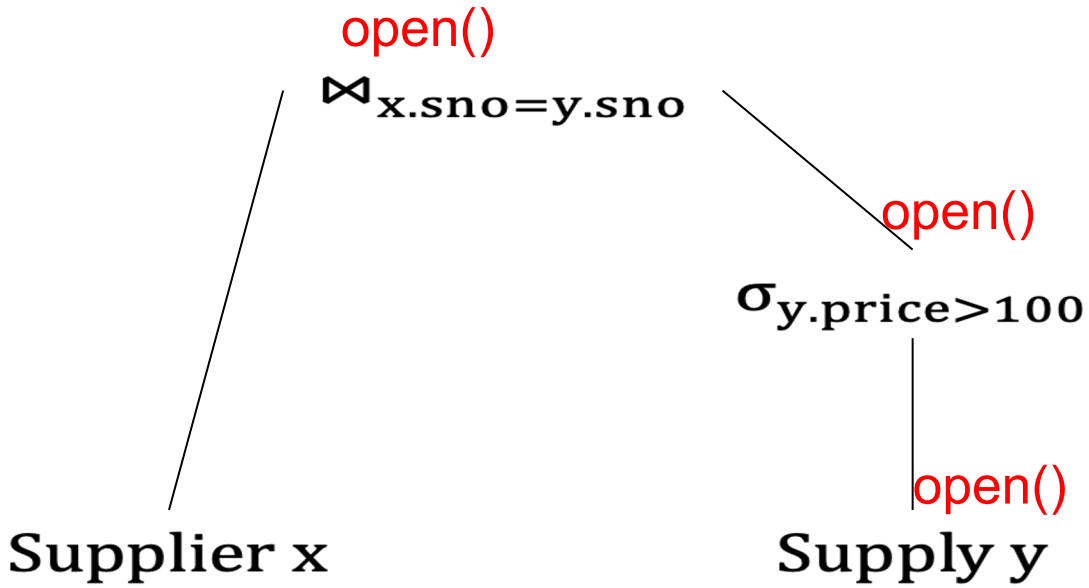
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



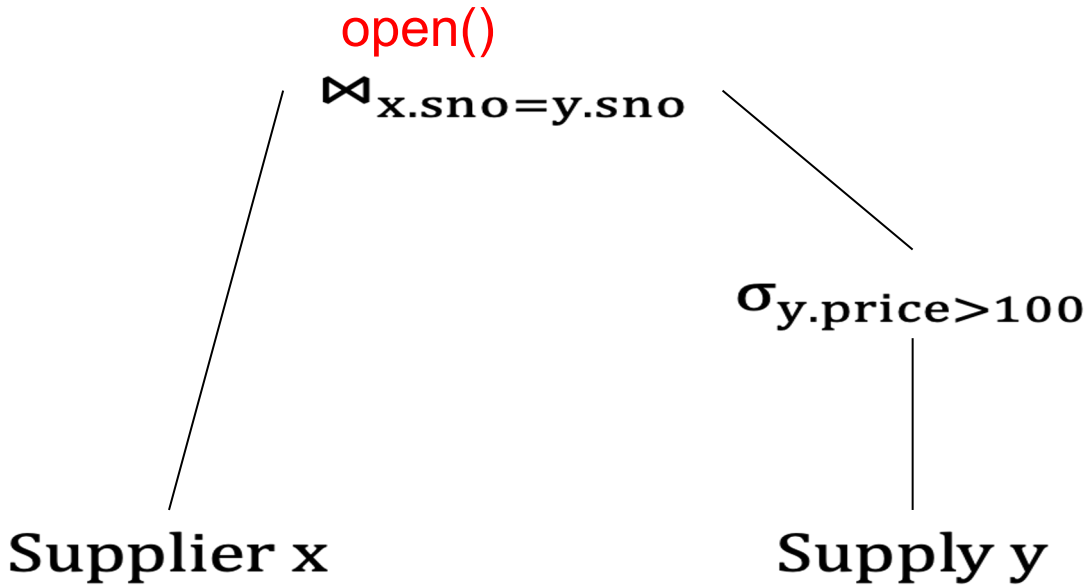
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



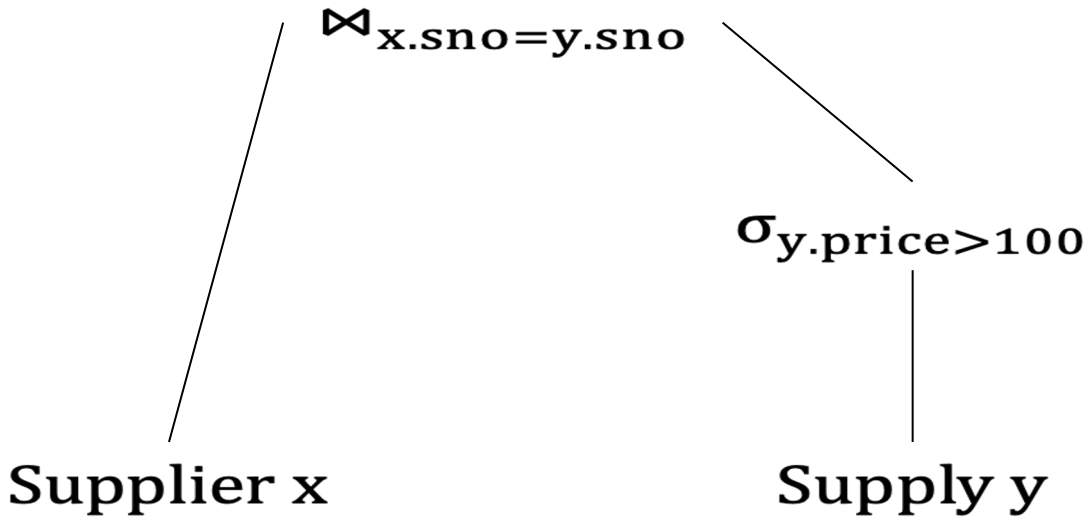
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



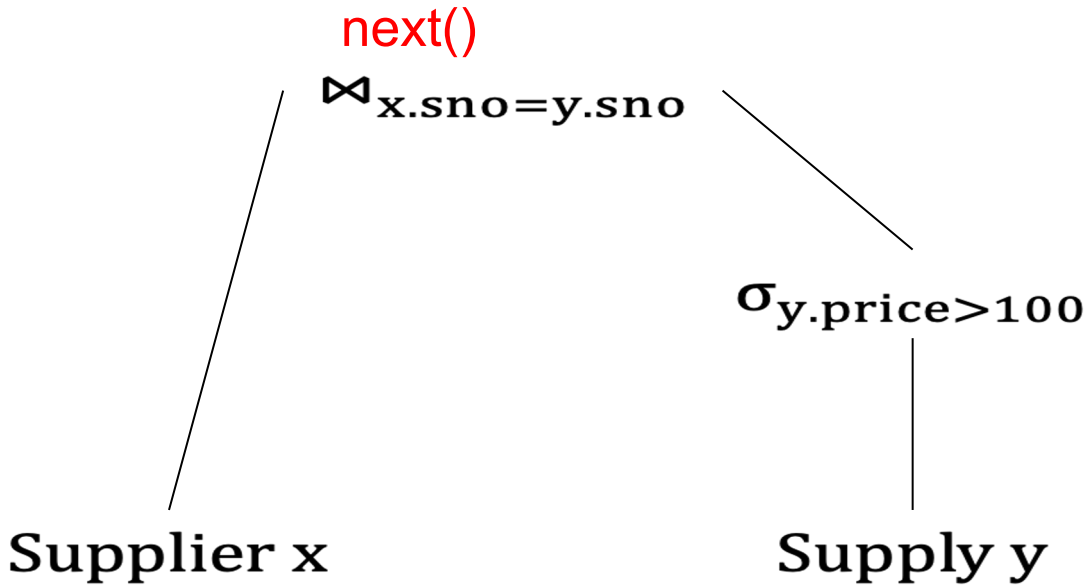
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



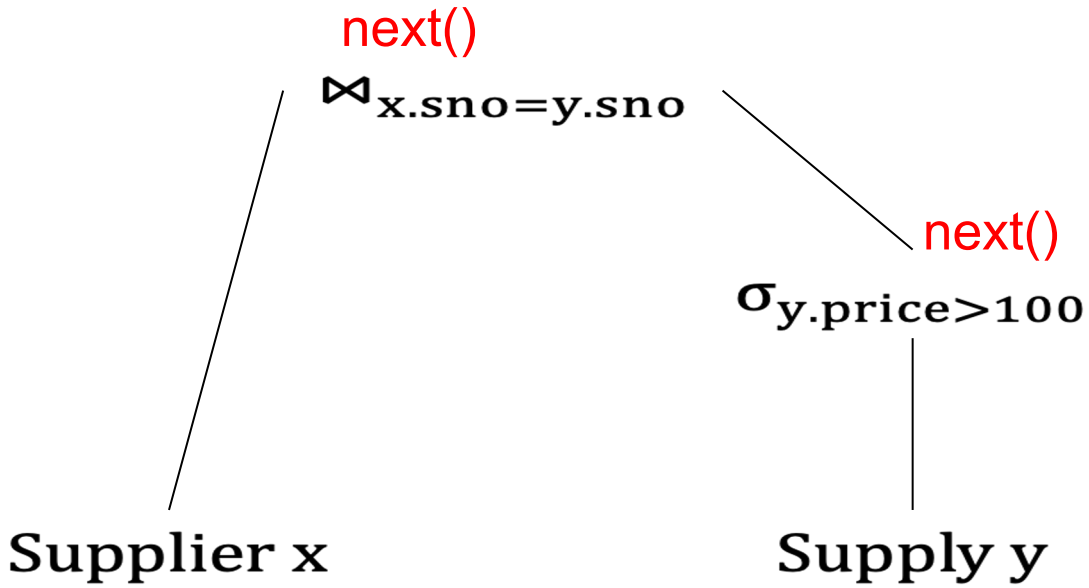
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



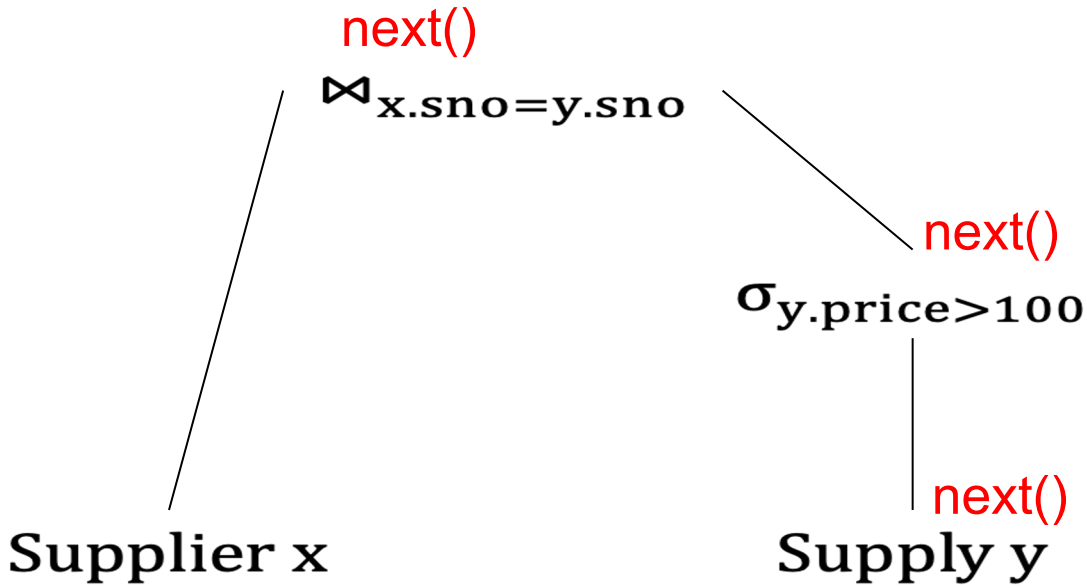
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



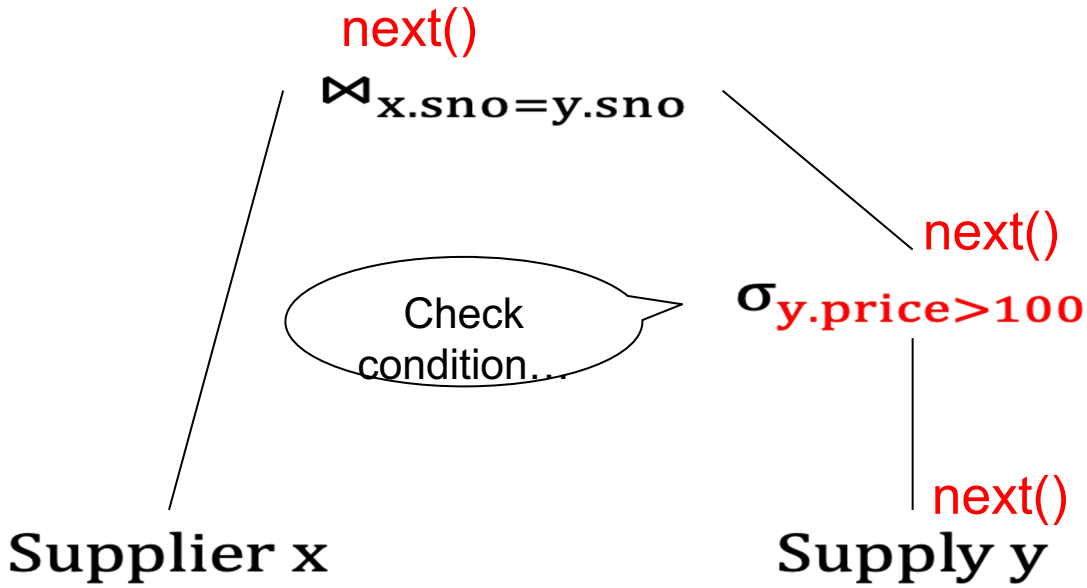
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



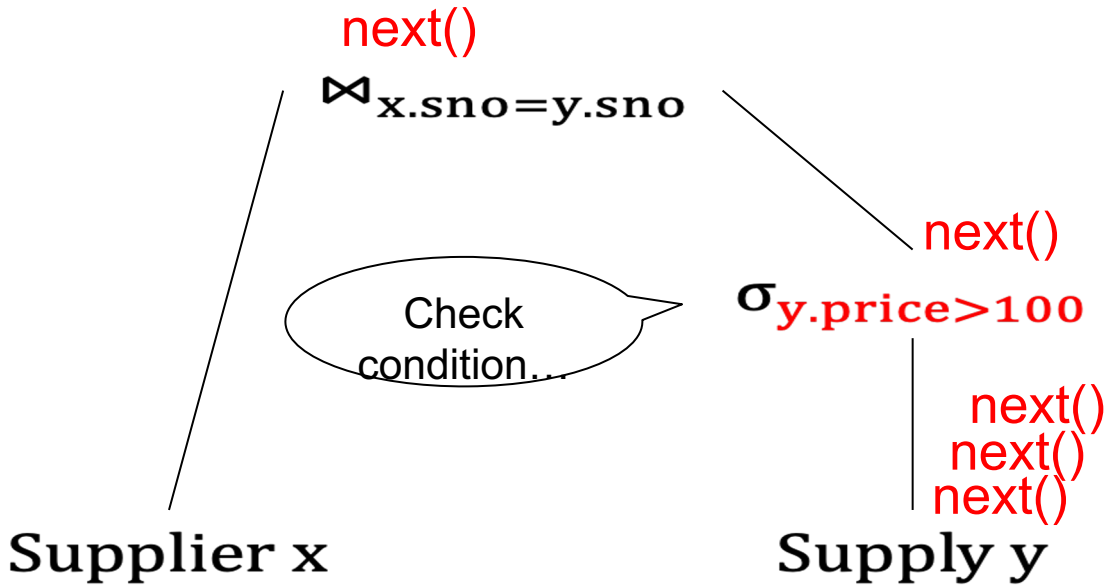
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



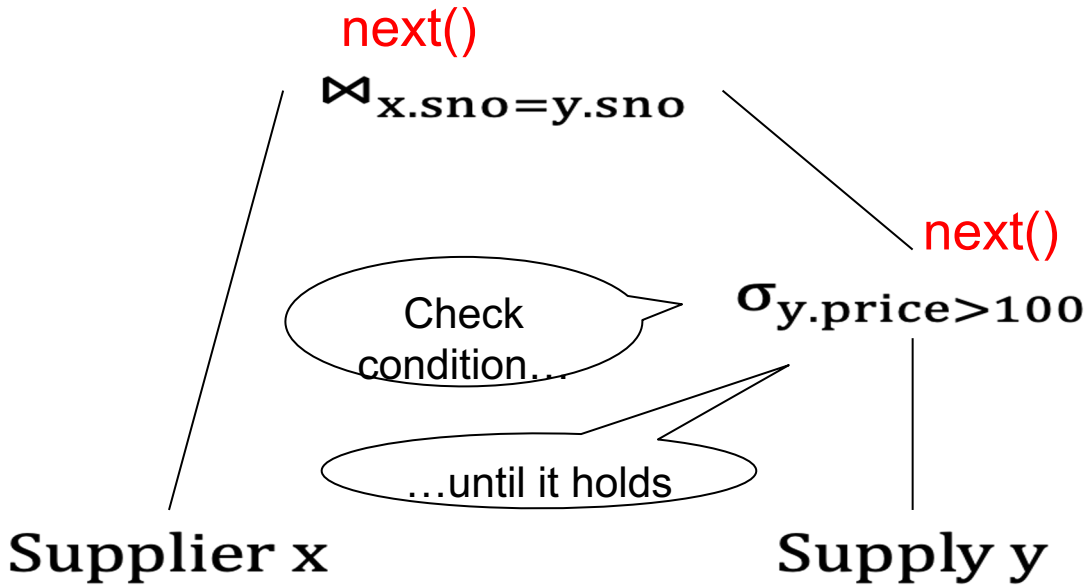
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



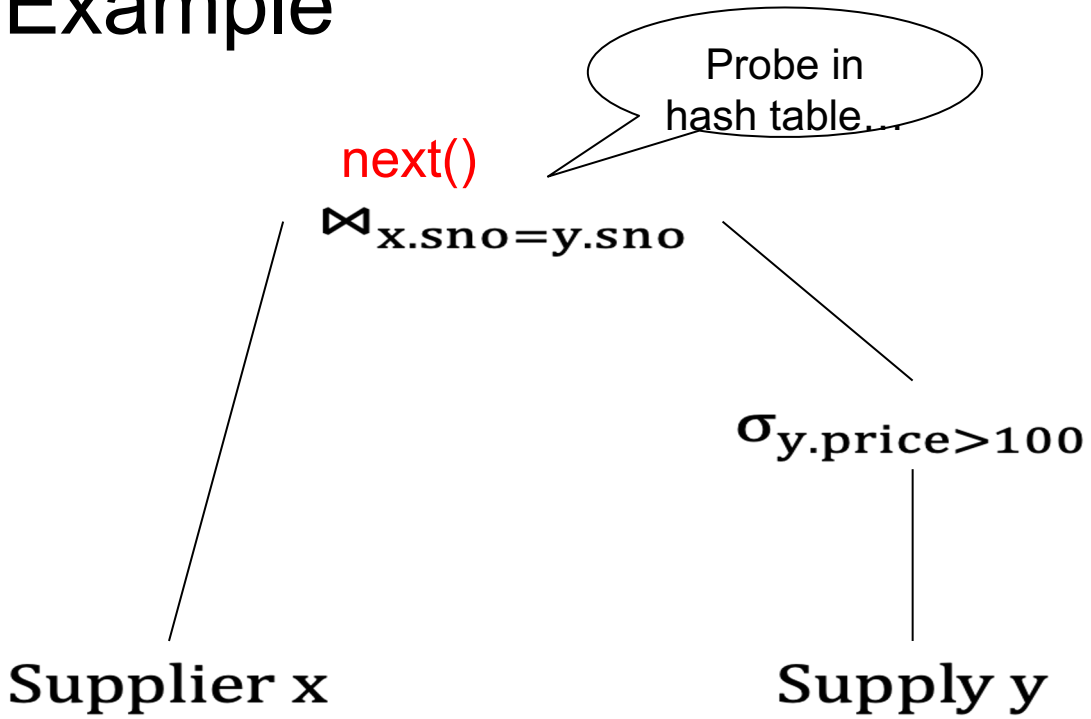
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



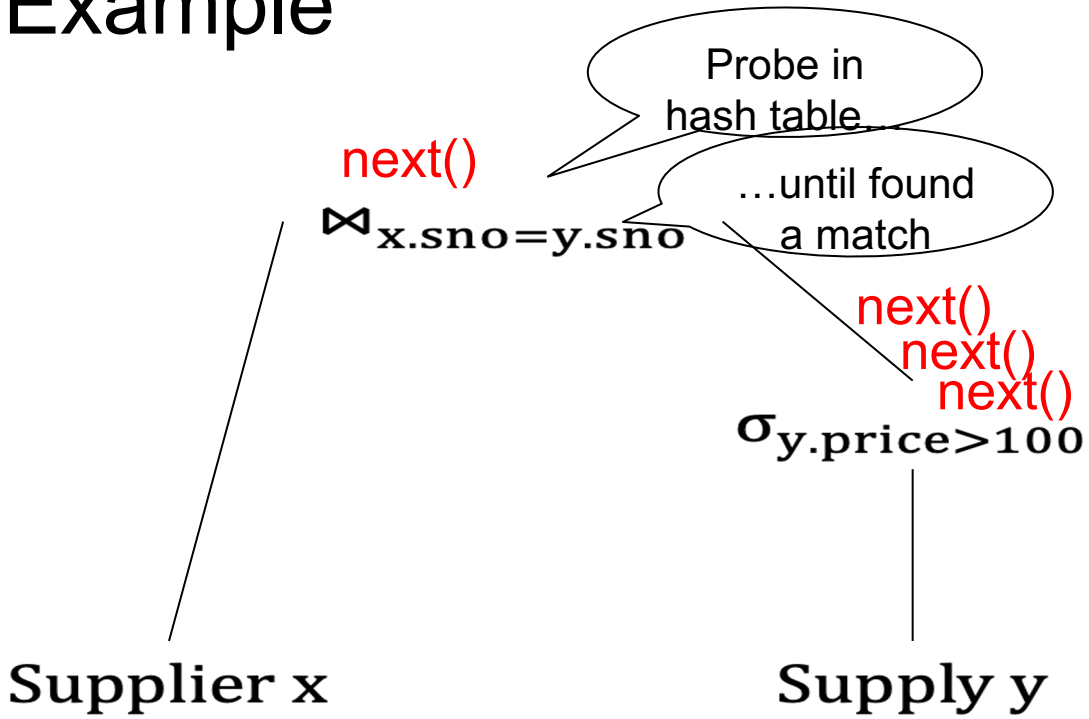
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



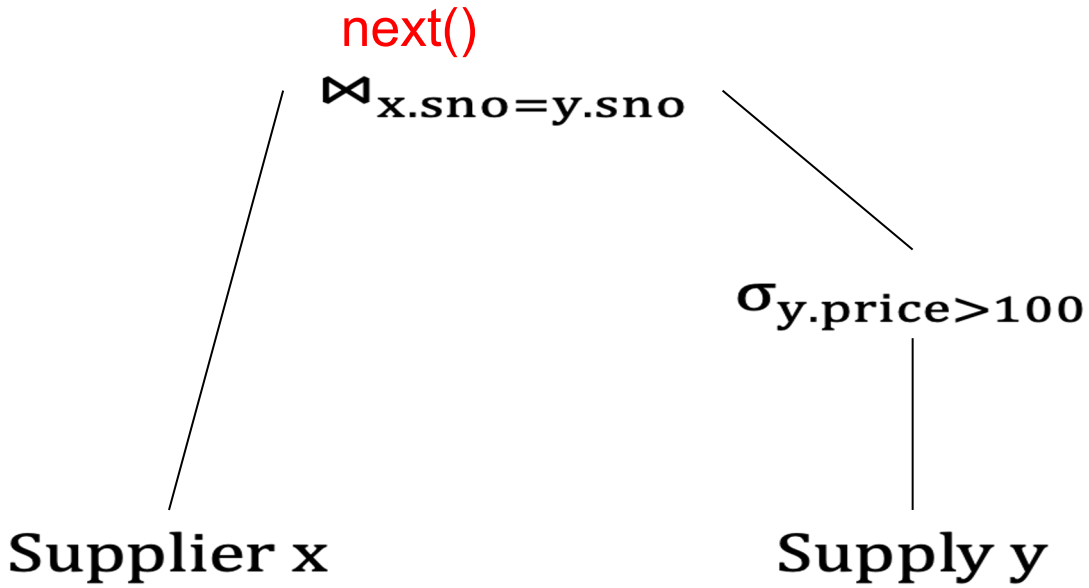
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



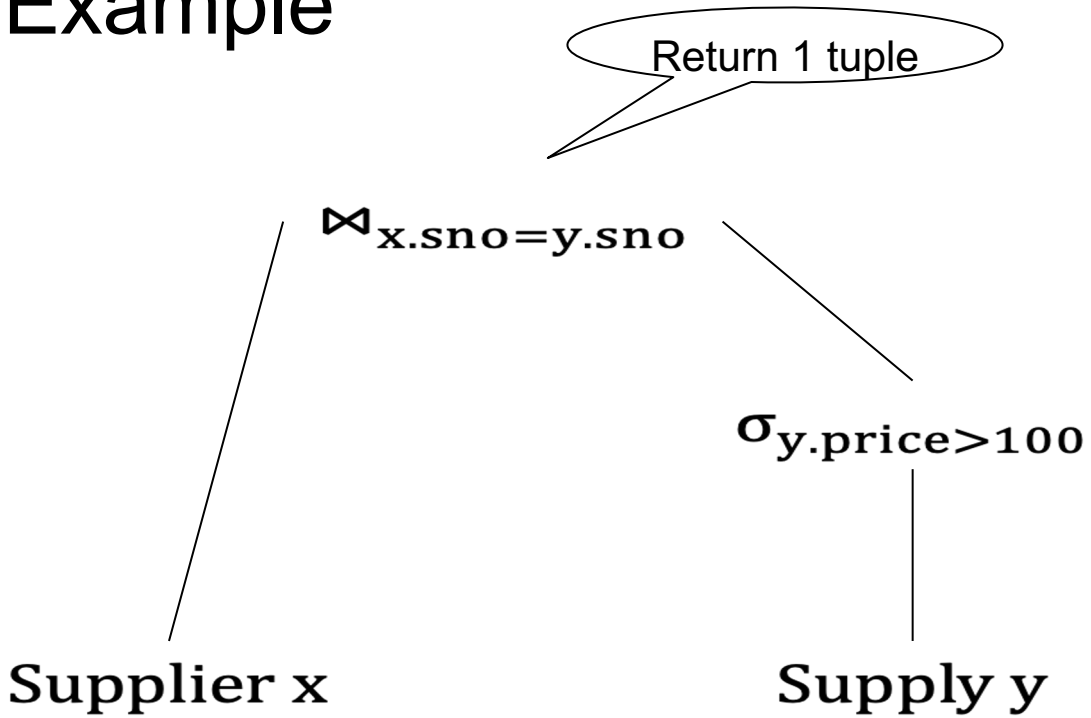
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



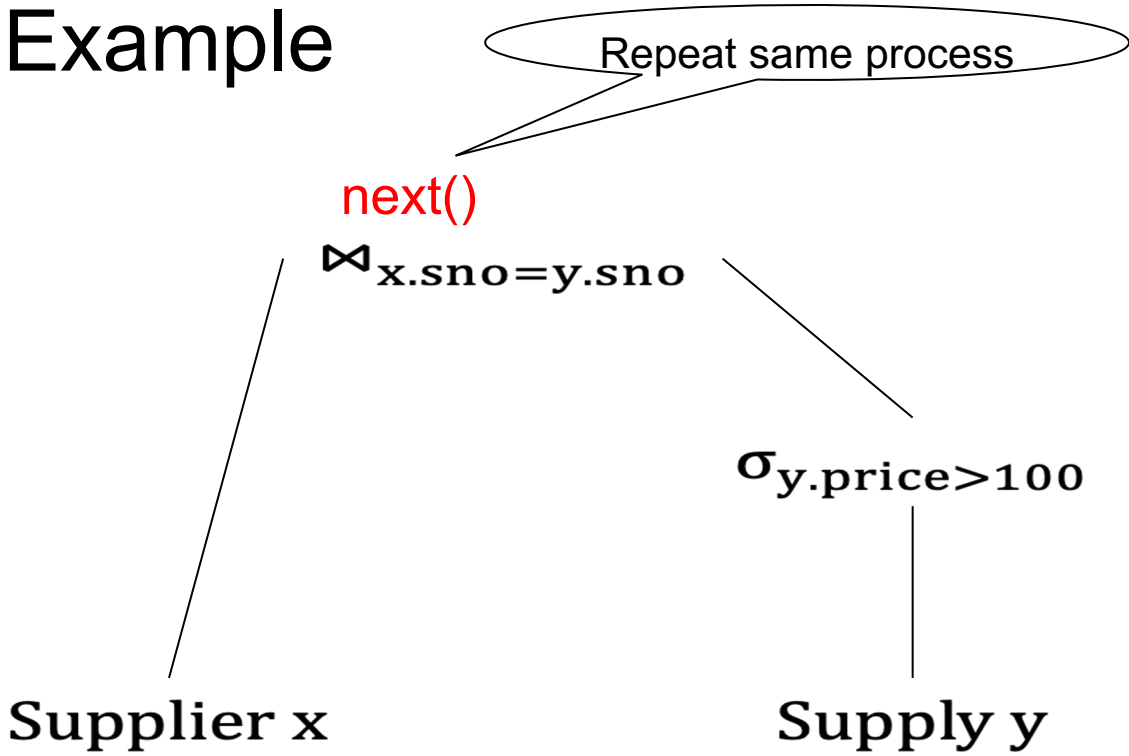
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

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 order significantly



Iterate v.s. Materialize

- Iterate
  - Disk IOs: O(Input)
  - Memory Footprint: O(~Input) <- keeps hash tables in memory
- Materialize
  - Disk IOs: O(Intermediates)
  - Memory Footprint: O(1)

Summary

- Start SimpleDB Early! (Due Feb 23rd)
- The catalog stores metadata
- The buffer pool caches disk pages
- The iterator model reduces disk I/O by using memory
    - Flexible model for implementation