CSE544 Data Management

Lectures 7
Query Execution – Part I
Announcements

• HW1 was due on Friday night.

• HW2 is posted, due on Tuesday, 4/27

• Review 4 is due on Wednesday
Query Optimization

- Local Client Protocols
- Remote Client Protocols

Client Communications Manager
- Query Parsing and Authorization
- Query Rewrite
- Query Optimizer
- Plan Executor

Relational Query Processor (Section 4)
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager

Transactional Storage Manager (Sections 5 & 6)

- Catalog Manager
- Memory Manager
- Administration, Monitoring & Utilities
- Replication and Loading Services
- Batch Utilities
- Shared Components and Utilities (Section 7)
Outline

• Steps involved in processing a query
  • Main Memory Operators
  • Query execution
  • External Memory Operators
Lifecycle of a Query

SQL query
- Parse & Rewrite Query
  - Select Logical Plan
  - Select Physical Plan
- Query Execution
- Disk

Query optimization
- Logical plan
- Physical plan
Simple Example
Simple Example

View: Suppliers in Seattle

CREATE VIEW NearbySupp AS
SELECT x.sno, x.sname
FROM Supplier x
WHERE x.scity='Seattle' AND x.sstate='WA'
Simple Example

View: Suppliers in Seattle

```
CREATE VIEW NearbySupp AS
    SELECT x.sno, x.sname
    FROM Supplier x
    WHERE x.scity='Seattle' AND x.sstate='WA'
```
Simple Example

• Find the names of all suppliers in Seattle who supply part number 2

```
SELECT x.sno, x.sname FROM NearbySupp x
WHERE x.sno IN (SELECT y.sno
    FROM Supply y
    WHERE y.pno = 2 )
```
Lifecycle of a Query (1)

• **Step 0: admission control**
  – User connects to the db with username, password
  – User sends query in text format

• **Step 1: Query parsing**
  – Parses query into an internal format
  – Performs various checks using catalog:
    Correctness, authorization, integrity constraints

• **Step 2: Query rewrite**
  – View rewriting, flattening, decorrelation, etc.
View Rewriting, Flattening

Original query:

```sql
SELECT x.sno, x.sname
FROM NearbySupp x
WHERE x.sno IN
  (SELECT y.sno
   FROM Supply y
   WHERE y.pno = 2 )
```

View rewriting
  = view inlining
  = view expansion
Flattening
  = unnesting
View Rewriting, Flattening

Original query:

```sql
SELECT x.sno, x.sname
FROM NearbySupp x
WHERE x.sno IN
  (SELECT y.sno
   FROM Supply y
   WHERE y.pno = 2 )
```

Rewritten query:

```sql
SELECT x.sno, x.sname
FROM Supplier x, Supply y
WHERE x.scity='Seattle' AND x.sstate='WA'
AND x.sno = y.sno
AND y.pno = 2;
```
Lifecycle of a Query (2)

• **Step 3: Query optimization**
  – Find an efficient query plan for the query
  – We will spend two lectures on this topic

• **A query plan is**
  – **Logical query plan**: a relational algebra tree
  – **Physical query plan**: add specific algorithms
Five Basic Relational Operators

- **Selection**: $\sigma_{\text{condition}}(S)$
- **Projection**: $\pi_{\text{list-of-attributes}}(S)$
- **Union** ($\cup$)
- **Set difference** ($-$),
- **Cross-product/cartesian product** ($\times$),

**Join**: $R \bowtie_\theta S = \sigma_\theta(R \times S)$

Other operators: semi-join, anti-semijoin
Extended Operators of Relational Algebra

• **Duplicate elimination** \((\delta)\)
  - Bag to set
  - Special case of \(\gamma\)

• **Group-by/aggregate** \((\gamma)\)
  - Example: \(\gamma_{\text{pcolor, max(psize)} \rightarrow m, \text{avg(psize)} \rightarrow s}(\text{Part})\)
  - Min, max, sum, average, count

• **Sort operator** \((\tau)\)
Logical Query Plan

SELECT x.sname 
FROM Supplier x, Supply y 
WHERE x.sno=y.sno 
    and x.scity='Seattle' 
    and x.sstate='WA' 
    and y.pno=2
Logical Query Plan

\[ \pi_{\text{sname}} \]

\[ \sigma_{\text{scity}='Seattle' \land \text{sstate}='WA' \land \text{pno}=2} \]

SELECT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
and x.scity='Seattle'
and x.sstate='WA'
and y.pno=2

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)
Part(pno, pname, psize, pcolor)
Logical Query Plan

\[ \pi_{\text{name}} \]

\[ \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA} \land pno=2} \]

\[ \text{sno} = \text{sno} \]

Supplier

Supply

Part
Physical Query Plan

(On the fly) $\pi_{\text{snname}}$

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

(Nested loop) $\text{sno} = \text{sno}$

Algorithm

Supplier (File scan)

Suply (Index lookup)

Physical plan = Logical plan + choice of algorithms + choice of access path

CSEP 544 - Spring 2021
Find all suppliers in ‘WA’ that supply *only* parts under $100
Decorrelation

Find all suppliers in ‘WA’ that supply *only* parts under $100

```sql
SELECT x.sno
FROM Supplier x
WHERE x.sstate = 'WA'
    and not exists
    (SELECT *
     FROM Supply y
     WHERE x.sno = y.sno
     and y.price > 100)
```
Decorrelation

SELECT x.sno
FROM Supplier x
WHERE x.sstate = 'WA'
    and not exists
    (SELECT *
      FROM Supply y
      WHERE x.sno = y.sno
           and y.price > 100)

Problem: RA consists of “set-at-at-time” operators.

Cannot express correlated subqueries
Decorrelation

```
SELECT x.sno
FROM Supplier x
WHERE x.sstate = 'WA'
    and not exists
    (SELECT *
     FROM Supply y
     WHERE x.sno = y.sno
     and y.price > 100)
```

```
SELECT x.sno
FROM Supplier x
WHERE x.sstate = 'WA'
    and x.sno not in
    (SELECT y.sno
     FROM Supply y
     WHERE y.price > 100)
```
Decorrelation

Un-nesting

(SELECT x.sno
 FROM Supplier x
 WHERE x.sstate = 'WA')
EXCEPT
(SELECT y.sno
 FROM Supply y
 WHERE y.price > 100)
EXCEPT = set difference

SELECT x.sno
 FROM Supplier x
 WHERE x.sstate = 'WA'
 and x.sno not in
(SELECT y.sno
 FROM Supply y
 WHERE y.price > 100)
(SELECT x.sno
FROM Supplier x
WHERE x.sstate = 'WA')
EXCEPT
(SELECT y.sno
FROM Supply y
WHERE y.price > 100)
Final Step in Query Processing

• **Step 4: Query execution**
  
  – Choice of algorithm
  
  – How to pass data between operators, e.g. materialized, or pipelined
Outline

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

• For each operator, several algorithms

• Main memory or external memory
Main Memory Algorithms

Logical operator:
\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

Three algorithms:
1. Nested Loops
2. Hash-join
3. Merge-join
1. Nested Loop Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid}=\text{sid}} \text{Supply} \]

for x in Supplier do
  for y in Supply do
    if x.sid = y.sid
      then output(x,y)
1. Nested Loop Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

for \( x \) in Supplier do
for \( y \) in Supply do
  if \( x.\text{sid} = y.\text{sid} \)
  then output\((x, y)\)

If \( |R| = |S| = n \), what is the runtime?
1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply}$$

for x in Supplier do
    for y in Supply do
        if x.sid = y.sid
            then output(x,y)

If $|R| = |S| = n$, what is the runtime?

$O(n^2)$
2. Hash Join

Logical operator:
\[ \text{Supply} \Join_{\text{sid} = \text{sid}} \text{Supplier} \]

```
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
2. Hash Join

Logical operator:
\[ \text{Supply} \bowtie_{\text{sid} = \text{sid}} \text{Supplier} \]

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

If \(|R| = |S| = n\), what is the runtime?
2. Hash Join

Logical operator:  
Supply \bowtie_{\text{sid}=\text{sid}} \text{Supplier}

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

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

If $|R|=|S|=n$, what is the runtime? 
$O(n)$
2. Hash Join

Logical operator: \( \bowtie_{\text{id} = \text{sid}} \)

Supplier(\(\text{sid}, \text{sname}, \text{scity}, \text{sstate}\))
Supply(\(\text{sid}, \text{pno}, \text{quantity}\))

for x in Supplier do
  insert(x.ssid, x)
for y in Supply do
  x = find(y.ssid);
  output(x, y);

If \(|R| = |S| = n\), what is the runtime?
\(O(n)\)
2. Hash Join

**Logical operator:**

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

for \( y \) in \text{Supply} do

\[ \text{insert}(y.\text{sid}, y) \]

for \( x \) in \text{Supplier} do

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

2. Hash Join

Logical operator: 
Supplier $\bowtie_{\text{sid} = \text{sid}}$ Supply

for y in Supply do
    insert(y.sid, y)

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

Duplicate sid’s
Change join order
2. Hash Join

Logical operator:
\[ \text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply} \]

for y in Supply do
  insert(y.sid, y)

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

If \(|R| = |S| = n\), what is the runtime?
2. Hash Join

Logical operator: 
\[ \text{Supplier} \Join_{\text{sid}=\text{sid}} \text{Supply} \]

For \( y \) in Supply do
insert(\( y.\text{sid}, y \))

For \( x \) in Supplier do
for \( y \) in find(\( x.\text{sid} \)) do
output(\( x,y \));

If \(|R|=|S|=n\), what is the runtime?
\[ O(n) \]
But can be \( O(n^2) \) why?
3. Merge Join

Logical operator:

\[
\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply}
\]

Sort(Supplier); Sort(Supply);
\[x = \text{Supplier}.\text{first}();\]
\[y = \text{Supply}.\text{first}();\]
3. Merge Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid}=\text{sid}} \text{Supply} \]

Sort(Supplier); Sort(Supply);
\( x = \text{Supplier}.\text{first}(); \)
\( y = \text{Supply}.\text{first}(); \)
while \( y \neq \text{NULL} \) do
  case:
    \( x.\text{sid} < y.\text{sid} \): ???
    \( x.\text{sid} = y.\text{sid} \): ???
    \( x.\text{sid} > y.\text{sid} \): ???
3. Merge Join

Logical operator:
\[ \text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply} \]

Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
  x.sid < y.sid: x = x.next()
x.sid = y.sid: ???
x.sid > y.sid: ???
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:
Supplier $\bowtie_{\text{sid} = \text{sid}}$ Supply

Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: output(x,y); y = y.next();
    x.sid > y.sid: ???
merge join

**Logical operator:**

$$\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply}$$

Sort(Supplier); Sort(Supply);

\[
x = \text{Supplier}.\text{first}();
\]

\[
y = \text{Supply}.\text{first}();
\]

\[
\text{while } y \neq \text{NULL do}
\]

\[
\text{case:}
\]

\[
x.\text{sid} < y.\text{sid}: x = x.\text{next}();
\]

\[
x.\text{sid} = y.\text{sid}: \text{output}(x, y); y = y.\text{next}();
\]

\[
x.\text{sid} > y.\text{sid}: y = y.\text{next}();
\]
3. Merge Join

Logical operator:

\[
\text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply}
\]

Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: output(x, y); y = y.next();
    x.sid > y.sid: y = y.next();

If |R| = |S| = n, what is the runtime?
3. Merge Join

Logical operator:

\[
\text{Supplier} \Join_{\text{sid}=\text{sid}} \text{Supply}
\]

Sort(Supplier); Sort(Supply);
\[x = \text{Supplier.first}();\]
\[y = \text{Supply.first}();\]
while \(y \neq \text{NULL} \) do
  case:
  \[x.\text{sid} < y.\text{sid}: \ x = x.\text{next}()\]
  \[x.\text{sid} = y.\text{sid}: \ \text{output}(x,y); \ y = y.\text{next}();\]
  \[x.\text{sid} > y.\text{sid}: \ y = y.\text{next}();\]

If \(|R| = |S| = n\), what is the runtime?
\[O(n \log(n))\]
Summary of Main Memory Algorithms

• Join \( \bowtie \):
  – Nested loop join
  – Hash join
  – Merge join

• Selection \( \sigma \)
  – “on-the-fly”
  – Index-based selection (next lecture)

• Group by \( \gamma \)
  – Hash-based
  – Merge-based

Briefly discuss in class
Outline

• Steps involved in processing a query
• Main Memory Operators
  • Query execution
• External Memory Operators
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: Iterator Interface
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
Supplier\((\text{sid, surname, scity, sstate})\) 
Supply\((\text{sid, pno, quantity})\) 

\begin{center}
\textbf{Pipelining}
\end{center}

\begin{itemize}
\item \textbf{(On the fly)}
\item \textbf{(On the fly)}
\item \textbf{(Nested loop)}
\end{itemize}

Discuss: open/next/close for nested loop join

\begin{align*}
\pi_{\text{ surname}} \\
\sigma_{\text{ scity} = 'Seattle' \text{ and sstate} = 'WA' \text{ and pno} = 2} \\
\sigma_{\text{ sid} = \text{ sid}} \\
\text{Supply (File scan)} & \quad \text{Supplier (File scan)}
\end{align*}
Supplier\((sid, \ sname, \ scity, \ sstate)\)
Supply\((sid, \ pno, \ quantity)\)

\[\text{Pipelining}\]

\(\text{(On the fly)}\)

\(\text{open(}\)°\text{)}\)

\(\pi_{\text{sname}}\)

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

\(\text{(Nested loop)}\)

\(\text{sid} = \text{sid}\)

Supply \((\text{File scan})\)
Supplier \((\text{File scan})\)

Discuss: open/next/close for nested loop join
Supplier$(sid, sname, scity, sstate)$
Supply$(sid, pno, quantity)$

**Pipelining**

(On the fly)

$$\pi_{sname}$$

(On the fly)

$$\sigma_{\text{scity}='Seattle' \text{ and } sstate='WA' \text{ and } pno=2}$$

(Nested loop)

$$\sigma_{sid = sid}$$

Supply (File scan)  Supply (File scan)

Discuss: open/next/close for nested loop join
**Pipelining**

(On the fly)

\[ \pi_{\text{sname}} \]

(On the fly)

\[ \sigma_{\text{scity}=\text{Seattle} \text{ and } \text{sstate}=\text{WA} \text{ and } \text{pno}=2} \]

(Nested loop)

\[ \text{sid = sid} \]

**Suppliers (File scan)**

**Supplies (File scan)**

Discuss: open/next/close for nested loop join
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

(On the fly)

(On the fly)

(On the fly)

(Nested loop)

\( \text{Discuss: open/next/close for nested loop join} \)

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

\( \Pi_{\text{sname}} \)

\( \text{sid} = \text{sid} \)

Supply (File scan)

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

(On the fly)

(On the fly)

(On the fly)

(Nested loop)

Discuss: open/next/close for nested loop join

\[ \sigma_{\text{scity}=\text{'Seattle'} \text{ and } \text{sstate}=\text{'WA'} \text{ and } pno=2} \]

\[ \pi_{\text{sname}} \]

sid = sid

Supply (File scan)

Supplier (File scan)
Pipelining

(On the fly)

(On the fly)

(Nested loop)

\[ \text{next}() \]

\[ \pi_{\text{name}} \]

\( \sigma_{\text{city} = 'Seattle' \text{ and state} = 'WA' \text{ and pno=2}} \)

\( \bowtie \)

\( \text{sid} = \text{sid} \)

Supply (File scan)

Supplier (File scan)

Discuss: open/next/close for nested loop join

Supplier\((\text{sid, sname, scity, sstate})\)
Supply\((\text{sid, pno, quantity})\)
Nested loop

On the fly

On the fly

Nested loop

Discuss: open/next/close for nested loop join

\[
\sigma_{\text{scity} = 'Seattle' \text{ and } \text{sstate} = 'WA' \text{ and } pno=2}
\]

\[
\Pi_{\text{sname}}
\]

\[
\text{Supply}
\]

\[
\text{Supplier}
\]

(File scan)

(File scan)
Supplier($\text{sid, sname, scity, sstate}$)
Supply($\text{sid, pno, quantity}$)

(On the fly)

$\pi_{\text{sname}}$ (Next)

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

(Nested loop)

$\text{sid = sid}$

Supply (File scan)  Supplier (File scan)

Discuss: open/next/close for nested loop join
Supplier\((\text{sid, sname, scity, sstate})\)
Supply\((\text{sid, pno, quantity})\)

**(Nested loop)**

\(\text{sid} = \text{sid}\)

**Next()**

**File scan**

**On the fly**

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

\(\pi_{\text{sname}}\)

**Next()**

Discuss: open/next/close for nested loop join

**On the fly**

\(\pi_{\text{sname}}\)

**Next()**

**Next()**

**File scan**

CSEP 544 - Spring 2021
### Pipelining

(Pip pip pip)

(On the fly)

(On the fly)

(On the fly)

(Nested loop)

 Discuss: open/next/close for nested loop join

\[
\begin{align*}
\text{Supplier}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \\
\text{Supply}(\text{sid}, \text{pno}, \text{quantity})
\end{align*}
\]

\[
\pi_{\text{sname}}(\text{next}())
\]

\[
\sigma_{\text{scity} = 'Seattle' \text{and sstate} = 'WA' \text{and pno}=2}(\text{next}())
\]

\[
\text{sid} = \text{sid}
\]

\[
\text{next}()
\]

\[
\text{Supply (File scan)}
\]

\[
\text{Supplier (File scan)}
\]
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Pipelining

\( \text{On the fly} \)
\( \text{On the fly} \)
\( \text{Nested loop} \)

\( \text{Discuss: open/next/close for nested loop join} \)
Suppliers (sid, sname, scity, sstate)

Supplies (sid, pno, quantity)

Discuss hash-join in class

(On the fly) π

(On the fly) σ

(Hash Join)

(sid = sid)

Supply (File scan)

Supplier (File scan)
Pipelining

(On the fly) $\pi_{\text{sname}}$

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

(Hash Join) sid = sid

Supply (File scan)

Supplier (File scan)

Discuss hash-join in class

Supplier($\text{sid, sname, scity, sstate}$)
Supply($\text{sid, pno, quantity}$)

Tuples from here are “blocked”
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Pipelining

(On the fly)

(On the fly) π_{sname}

(On the fly) σ_{scity='Seattle' and sstate='WA' and pno=2}

(Hash Join)

π_{sname}

Tuples from here are pipelined

Tuples from here are “blocked”

Supply (File scan)

Suppliers (File scan)
Blocked Execution

(On the fly)

(On the fly)

(Merge Join)

Discuss merge-join in class

\[ \pi_{\text{sname}} (\sigma_{\text{scity}=\text{Seattle} \text{ and sstate}=\text{WA} \text{ and pno}=2} (\text{Supplier} \land \text{Supply})) \]

\[ \text{Supply (File scan)} \land \text{Supplier (File scan)} \]

\[ \text{sid} = \text{sid} \]
Blocked Execution

Suppliers$(\text{sid}, \text{sname}, \text{scity}, \text{sstate})$
Supply$(\text{sid}, \text{pno}, \text{quantity})$

(On the fly)

$\prod_{\text{sname}}$

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

(Merge Join)

Blocked

Supply
(File scan)

Blocked

Supply
(File scan)

Discuss merge-join in class
Pipeline v.s. Blocking

• Pipeline
  – A tuple moves 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
Iterator Model
A.k.a. Volcano-style execution

interface Operator {

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

interface Operator {

    // initializes operator state
    // and sets parameters
    void open (...);

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

```java
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 ();
}
```
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
Iterator Model
A.k.a. Volcano-style execution

Example “on the fly” selection operator

```java
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

Example “on the fly” selection operator

interface Operator {
  void open (...);

  // 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 ();
}

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

Example "on the fly" selection operator

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 ();
}

class Select implements Operator {
    void open (Predicate p, Operator c) {
        this.p = p; this.c = c; c.open();
    }
    Tuple next () {
    }
}
Iterator Model
A.k.a. Volcano-style execution
Example “on the fly” selection operator

interface Operator {
    void open (...);
    Tuple next ();
    void close ();
}

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);
        }
    }
}
Iterator Model
A.k.a. Volcano-style execution

Example “on the fly” selection operator

```java
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 ();
}

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

Example “on the fly” selection operator

```java
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 ();
}

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

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 ();
}

Query plan execution

Operator q = parse("SELECT ...");
q = optimize(q);
q.open();

while (true) {
    Tuple t = q.next();
    if (t == null) break;
    else printOnScreen(t);
}
q.close();
Summary

• Three join operators: loop, hash, merge
• Many variations:
  – ”double pipelined hash join” – what is this?
• Tuple flow: materialize, pipeline
• Interface:
  – Volcano (pull): `next()`
  – Data-driven (push): `produce()`, `consume()`