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()
Recap: Hash Join

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

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

for $y$ in Supply do
  $x = \text{find}(y$.sid$)$;
  output($x, y$);

Probe phase

Build phase
Volcano Model

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

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

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

Supply (File scan)  Supplier (File scan)

for \( x \) in Supplier do
  insert(\( x.\text{sid} \), \( x \))
for \( y \) in Supply do
  \( x = \text{find}(y.\text{sid}) \);
  output(\( x,y \));
Volcano Model

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

(On the fly)

(On the fly)

(On the fly)

(Hash Join)

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

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

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

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

\( \text{sid = sid} \)

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

\[ \text{Supplier (File scan)} \]
Volcano Model

(On the fly)

(On the fly)

(Hash Join)

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);

for (x, y) in Supplier x Supply do
  if scity='Seattle' and sstate='WA' and pno=2 then
    open()
  else
    open()
  endif
  output(x, y);

(On the fly)

(On the fly)

(Hash Join)
Volcano Model

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

(On the fly)

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

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

(Hash Join)

for \(x\) in Supplier do
insert\((x.\text{sid}, x)\)
for \(y\) in Supply do
\(x = \text{find}(y.\text{sid});\)
output\((x,y)\);
Volcano Model

(On the fly)

$\Pi_{\text{sname}}$

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

(Hash Join)

sid = sid

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

next()
Volcano Model

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

(On the fly) $(\text{Hash Join})$

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

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

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

for $y$ in Supply do
$x = \text{find}(y$.sid$);$
output($x,y$);
Volcano Model

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

(On the fly) Π_{sname} (On the fly) σ_{scity='Seattle' and sstate='WA' and pno=2} (Hash Join) (File scan) (File scan)

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

(On the fly) $\text{Supplier}(\text{sid, sname, scity, sstate})$

(On the fly) $\text{Supply}(\text{sid, pno, quantity})$

(Hash Join) $\sigma_{\text{scity}=\text{\textquoteleft}Seattle\textquoteleft \text{ and } \text{sstate}=\text{\textquoteleft}WA\textquoteleft \text{ and } \text{pno}=2}$

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

(open) $\text{for } x \text{ in } \text{Supplier} \text{ do}$
\hspace{1cm} insert($x$.sid, $x$)

(open) $\text{for } y \text{ in } \text{Supply} \text{ do}$
\hspace{1cm} $x = \text{find}(y$.sid$);$
\hspace{1cm} output($x,y$);

(open) $\text{close}()$

(open) $\text{for } x \text{ in } \text{Supplier} \text{ do}$
\hspace{1cm} insert($x$.sid, $x$)

(open) $\text{for } y \text{ in } \text{Supply} \text{ do}$
\hspace{1cm} $x = \text{find}(y$.sid$);$
\hspace{1cm} output($x,y$);

(open) $\text{close}()$
Volcano Model

(On the fly)  \( \text{Supplier}(sid, \text{sname}, \text{scity}, \text{sstate}) \)

(On the fly)  \( \text{Supply}(sid, \text{pno}, \text{quantity}) \)

(Hash Join)  

\[ \text{for } x \text{ in Supplier do} \]
\[ \quad \text{insert}(x.\text{sid}, x) \]
\[ \text{for } y \text{ in Supply do} \]
\[ \quad x = \text{find}(y.\text{sid}); \]
\[ \quad \text{output}(x, y); \]

CSEP 544 - Spring 2021
Volcano Model

(On the fly)

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

(Hash Join)

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

\[
\begin{align*}
\text{next() } & \quad \pi_{\text{sname}} \\
\text{for } x \text{ in Supplier do} & \quad \text{insert}(x.\text{sid}, x) \\
\text{for } y \text{ in Supply do} & \quad x = \text{find}(y.\text{sid}); \quad \text{output}(x,y); \\
\end{align*}
\]
Volcano Model

(On the fly)  \( \Pi_{sname} \)

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

(Hash Join)

Supply  (File scan)  Supplier  (File scan)

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

(On the fly)

\( \Pi_{sname} \) (On the fly)

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

(Hash Join)

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

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

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

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

(Hash Join)

Suppliers

(Hash Join)

Supply (File scan)

next() (On the fly)

next() (On the fly)

next() (On the fly)

next() (Hash Join)

Supply (File scan)

next() (Hash Join)

Supplier (File scan)

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

(On the fly)

(On the fly)

(Hash Join)

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

\[ \text{for } x \text{ in Supplier do} \]
\[ \quad \text{insert}(x.\text{sid}, x) \]

\[ \text{for } y \text{ in Supply do} \]
\[ \quad x = \text{find}(y.\text{sid}); \]
\[ \quad \text{output}(x,y); \]

\[ \text{next()} \]
\[ \quad \pi_{\text{sname}} \]
\[ \quad \text{next()} \]
\[ \quad \sigma_{\text{scity}=\text{Seattle} \text{ and } \text{sstate}=\text{WA} \text{ and } \text{pno}=2} \]
\[ \quad \text{next()} \]
\[ \quad \text{sid} = \text{sid} \]
\[ \text{next()} \]
\[ \text{next()} \]

next()
next()
next()
next()
Volcano Model

(On the fly)

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

next()

(On the fly)

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

(Hash Join)

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

Supply

(File scan)

Supplier

(File scan)

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)

(On the fly)

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

(Hash Join)

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

next()
Volcano Model

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

(On the fly)
for x in Supplier do
  insert(x.sid, x)
next()

(On the fly)
for y in Supply do
  x = find(y.sid);
  output(x, y);
next()
next()
next()
next()

(Hash Join)
π_{sname}

σ_{scity='Seattle' and sstate='WA' and pno=2}

next()
next()
sid = sid

Supply (File scan)

Supplier (File scan)
Volcano Model

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

(On the fly)

(On the fly)

(Hash Join)

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

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

\[ \text{next}() \]

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

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

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

for x in Supplier do
insert(x.sid, x)
for y in Supply do
x = find(y.sid);
output(x, y);
Each operator exports four methods:

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

called *once* by parent

called *repeatedly* by children
Data-Driven

(On the fly)

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

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

(Hash Join)

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

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

for x in Supplier do
  insert(x.sid, x)
for y in Supply do
  x = find(y.sid);
  output(x, y);
(On the fly) data-driven

```
(CS544 - Spring 2021)

```

(On the fly) data-driven

```
(CS544 - Spring 2021)

```

(Hash Join) data-driven

```
(CS544 - Spring 2021)

```
Data-Driven

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

(On the fly)

produce()

\( \pi_{sname} \)

produce()

produce()

(On the fly)

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

(Hash Join)

sid = sid

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

for x in Supplier do
produce()
for y in Supply do
produce()

produce()

produce()
Data-Driven

(On the fly)

(Hash Join)

(On the fly)

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

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

\[
\text{for } x \text{ in Supplier do }
\]
\[
\quad \text{insert}(x.\text{sid}, x)
\]

\[
\text{for } y \text{ in Supply do }
\]
\[
\quad x = \text{find}(y.\text{sid});
\]
\[
\quad \text{output}(x, y);
\]

\[
\text{produce()}
\]
\[
\text{produce()}
\]
\[
\text{produce()}
\]

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

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

\[
\text{produce()}
\]

\[
\text{for } x \text{ in Supplier do }
\]
\[
\quad \text{insert}(x.\text{sid}, x)
\]

\[
\text{for } y \text{ in Supply do }
\]
\[
\quad x = \text{find}(y.\text{sid});
\]
\[
\quad \text{output}(x, y);
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

\[
\text{produce()}
\]

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

(On the fly) produce() π_{sname}
(On the fly) σ_{scity='Seattle' and sstate='WA' and pno=2}
(Hash Join)

produce() sid = sid consume()
produce() produce() produce()
```

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

(On the fly)
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

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

(Hash Join)
π_{sname}

produce()

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

produce()

consume()

consume()

produce()

produce()

produce()

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

produce()

consume()

consume()

produce()

produce()
Data-Driven

(On the fly)

(Hash Join)

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

data = Supplier(points)
data = Supply(points)

g = (scity = 'Seattle' and sstate = 'WA' and pno = 2)

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

produce() for x in Supplier do
  produce(x)

produce() for y in Supply do
  consume(y)

produce() for x in Supplier do
  consume(x)

produce() for y in Supply do
  produce(y)

produce() for x in Supplier do
  consume(x)

produce() for y in Supply do
  consume(y)

produce() for x in Supplier do
  consume(x)

produce() for y in Supply do
  consume(y)
Data-Driven

(On the fly)

(On the fly)

(Hash Join)

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);

Build phase done

produce()

\[ \pi_{sname} \]

produce()

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

produce()

sid = sid

Supply (File scan)  Supplier (File scan)
Data-Driven

(On the fly)

(On the fly)

(Hash Join)

produce()

π_{sname}

produce()

σ_{scity='Seattle' and sstate='WA' and pno=2}

produce()

produce()

produce()

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)

produce()

produce()

produce()

produce()

produce()

File scan

File scan

CSEP 544 - Spring 2021
Data-Driven

(On the fly)

(On the fly)

(Hash Join)

-produce()

-produce()

-produce()

-produce()

-produce()

-supply(
  (File scan)
)

-supplier(
  (File scan)
)

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

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

π_{sname}

σ_{scity='Seattle' and sstate='WA' and pno=2}

consume()

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

(On the fly) π_{sname} produce()

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

(Hash Join) consume()

produce() Supply (File scan)
produce() Supplier (File scan)

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

(On the fly)

(Hash Join)

Producer() 
π\text{sname} 
Producer() 
σ\text{scity}=‘Seattle’\text{ and sstate}=‘WA’\text{ and pno}=2 
Producer() 
consume() 
consume() 
consume() 
consume() 
Producer() 
Producer() 
Producer() 
Producer()

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

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

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

π_{sname}

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

consume() produce() produce() produce() consume() consume() consume() generate() generate() generate() Supplier (File scan) Supply (File scan)
Supplier($sid, sname, scity, sstate$)
Supply($sid, pno, quantity$)

(On the fly)

produce()
consume($\pi_{sname}$)
produce()
consume()
produce()

(On the fly)

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

(Hash Join)

consume()
consume()
sid = sid
consume()
produce($\text{Supply (File scan)}$)
produce($\text{Supplier (File scan)}$)

for $x$ in Supplier do
insert($x.sid, x$)
for $y$ in Supply do
  $x = \text{find}(y.sid)$;
  output($x, y$);
Data-Driven

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

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

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

(Hash Join)
consume()
consume()
consume()
produce()
produce()
consume()

Supply (File scan)
Supplier (File scan)

CSEP 544 - Spring 2021

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)

(On the fly) 
produce() 
π_{sname}
produce() 
produce() 
consume() 
consume() 
consume() 
produce() 
Supply (File scan)

(On the fly) 
σ_{scity='Seattle' and sstate='WA' and pno=2}
consume() 
sid = sid
consume() 
consume() 
consume() 
produce() 
Supplier (File scan)

(Hash Join) 

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

produce()
produce()
consume()
Data-Driven

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

(On the fly)
produce()
consume()
produce()
consume()
produce()
consume()
produce()
consume()
produce()
consume()
produce()
consume()
consume()
produce()
consume()
produce()
consume()
consume()
produce()
consume()
consume()
produce()
consume()

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

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

(Hash Join)
produce()
consume()
consume()
consume()
consume()
consume()
produce()
Supplier (File scan)

CSEP 544 - Spring 2021
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
class HashJoin(left: Op, right: Op) {
    (lkey: KeyFun)(rkey: KeyFun) extends Op {
        val hm = new HashMap()
        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))
        }
    }
}

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

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)                                                  (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 HashMap()
    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))
    }
  }
}

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 HashMap()
      left.exec { rec => // Step 1
        hm += (lkey(rec), rec)
      }
      right.exec { rec => // Step 2
        for (lr <- hm(rkey(rec)))
          cb(merge(lr, rec))
      }
    }
  }
}

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))
      }
    }
  }
}

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))
      }
    }
  }
}

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

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

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

Supply($sid$, $pno$, $quantity$)

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 = $
Cost Parameters

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

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 =
Cost Parameters

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

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, surname) = 40,000,000 meaning?
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =
Cost Parameters

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

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 =
Cost Parameters

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

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 =
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 = 10,000,000 = 80GB why so little?
Selection

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

$V(R, a) = \# \text{ of distinct values of attribute } a$

- **Sequential scan:**
  - $\text{cost} = B(R)$

CSEP 544 - Spring 2021

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

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

\[ V(R, a) = \# \text{ of distinct values of attribute } a \]

- **Sequential scan:**
  - cost = \( B(R) \)

- **Index-based selection:**
  - Unclustered index on \( a \): cost = \( T(R) / V(R,a) \)
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 \( a \): \( \text{cost} = \frac{T(R)}{V(R,a)} \)
  – Clustered index on \( a \): \( \text{cost} = \frac{B(R)}{V(R,a)} \)

```sql
SELECT *
FROM Supplier
WHERE city = 'Seattle'
```
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 = \( \frac{T(R)}{V(R,a)} \)
  - Clustered index on \( a \):
    - cost = \( \frac{B(R)}{V(R,a)} \)

- **Assumptions:**
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

```sql
SELECT * 
FROM Supplier 
WHERE city = 'Seattle'
```
Selection on equality: \( \sigma_{a=v}(R) \)

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

- **Sequential scan:**
  - \( \text{cost} = B(R) \)

- **Index-based selection:**
  - Unclustered index on \( a \):
    - \( \text{cost} = T(R) / V(R,a) \)
  - Clustered index on \( a \):
    - \( \text{cost} = B(R) / V(R,a) \)

- **Assumptions:**
  - Values are uniformly distributed
  - Ignore the cost of reading the index (why?)

```
SELECT *
FROM Supplier
WHERE city = '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?)

\[ \begin{align*}
  \text{B(R)} &= 2000 \\
  \text{T(R)} &= 100,000 \\
  \text{V(R, a)} &= 20
\end{align*} \]
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 city = 'Seattle'

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

2000
5000
100
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<?
SELECT * 
FROM R 
WHERE R.K>? and R.K<?
\[
\text{SELECT} \quad * \\
\text{FROM} \quad R \\
\text{WHERE} \quad R.K > ? \quad \text{and} \quad R.K < ?
\]
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: $T(R) + T(R) \times B(S)$
  - WHY?
- Switch order: $B(S) + B(R) \times T(S)$

- 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”

\[
\begin{align*}
&\text{for each (M-2) pages } ps \text{ of } S \text{ do} \\
&\quad \text{for each page } pr \text{ of } R \text{ do} \\
&\quad \quad \text{for each tuple } s \text{ in } ps \\
&\quad \quad \quad \text{for each tuple } r \text{ in } pr \text{ do} \\
&\quad \quad \quad \quad \text{if } r \text{ and } s \text{ join then output}(r,s)
\end{align*}
\]
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)
```

$B(S) + B(S)B(R)/(M-2)$ disk I/Os. WHY?
Block Nested Loop Join

Hash table for block of S (M-2 pages)

Input buffer for R

Output buffer

Join Result

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

• $R \bowtie S$, 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

Relation R

Disk

B(R)

1

2

\ldots

M main memory buffers

INPUT

\text{hash function } h

OUTPUT

1

2

\ldots

M-1

Partitions

Disk

1

2

\ldots

M-1
Step 1: Hash-partition

- Partition R into buckets, on disk

```
Relation R

INPUT

hash function h

M main memory buffers

OUTPUT

Partitions

Disk

1

2

B(R)

... ...

Disk

1

2

M-1
```
Step 1: Hash-partition

- Partition R into buckets, on disk

```
Relation R

INPUT

hash function h

OUTPUT

Partitions

Disk

B(R)

1

2

M main memory buffers

1

2

M-1

Disk
```
Step 1: Hash-partition

- Partition R into buckets, on disk

Relation R

Disk

B(R)

\[ h : M \rightarrow 1, 2, \ldots, M-1 \]

M main memory buffers

INPUT

\[ h \]

hash function

OUTPUT

Partitions

Disk

1

2

M-1
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk

Relation R

INPUT

hash function $h$

OUTPUT

M main memory buffers

Disk

B(R)

1

2

...
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk

$$\text{Relation } R \xrightarrow{\text{hash function } h} M \text{ main memory buffers} \rightarrow \text{Partitions}$$

$$\text{Disk} \rightarrow B(R) \rightarrow \text{Disk}$$
Step 1: Hash-partition

- Partition R into buckets, on disk.
Step 1: Hash-partition

- Partition R into buckets, on disk
- Partition S

![Diagram of hash partitioning process]

Relation S

Disk

M main memory buffers

Disk

Partitions

Bucket
Step 2: Join Buckets

\[ R \bowtie S \]

- Read one S-backed; hash-partition it using \( h_2 \) (\( \neq h \))

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 \bowtie S

- Read one S-backed; hash-partition it using $h2$ ($\neq h$)
- 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

$R \bowtie S$

- Read one $S$-backed; hash-partition it using $h_2 \neq h$
- Scan corresponding $R$ bucket and join

One entire $S$-bucket fits in $M$ if $B(S) / M \leq M$, or $B(S) \leq M^2$. \textbf{WHY?}
Step 2: Join Buckets

R \bowtie 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 \( \frac{B(S)}{M} \leq M \), or \( B(S) \leq M^2 \). **WHY?**
Step 2: Join Buckets

\[ R \bowtie S \]

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

$R \bowtie S$

- Read one $S$-backed; hash-partition it using $h2$ ($\neq h$)
- 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

\( R \bowtie S \)

- Read one S-backed; hash-partition it using \( h_2 \) (\( \neq h \))
- 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?
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 fit in $M$:
  \[ \frac{t}{k} \times B(S) \leq M \]
How to choose k and t?

- The first t buckets must fit in M: $\frac{t}{k} \times B(S) \leq M$
- 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: \( \frac{t}{k} \times B(S) \leq M \)
• Need room for k-t additional pages: \( k-t \leq M \)
• Thus: \( \frac{t}{k} \times B(S) + k-t \leq M \)
Hybrid Hash Join Algorithm

How to choose \( k \) and \( t \) ?

- The first \( t \) buckets must fit in \( M \): \( \frac{t}{k} \cdot B(S) \leq M \)
- Need room for \( k-t \) additional pages: \( k-t \leq M \)
- Thus:
  \[
  \frac{t}{k} \cdot B(S) + k-t \leq M
  \]

Assuming \( \frac{t}{k} \cdot B(S) \gg k-t \):
\[
\frac{t}{k} = \frac{M}{B(S)}
\]
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

![Diagram showing loading of M pages into main memory from disk and number of runs calculation]

Size M pages

Main memory

Runs of length M

#Runs = B(R)/M

Disk

Disk
- 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