Introduction to Data Management
CSE 414

Unit 4: RDBMS Internals
Logical and Physical Plans
Query Execution
Query Optimization

(3 lectures)
Introduction to Data Management
CSE 414

Lecture 15: Introduction to Query Evaluation
Announcements

• WQ5 (datalog) due tomorrow

• HW4 (datalog) due tomorrow

• Midterm review session this evening
  – 5:30pm, CSE 2nd Floor Breakout
Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDBMS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)
From Logical RA Plans to Physical Plans
Query Evaluation Steps Review

- Parse & Rewrite Query
- Select Logical Plan
- Select Physical Plan
- Query Execution

SQL query

Query optimization

Logical plan (RA)

Physical plan

Disk
Logical vs Physical Plans

• Logical plans:
  – Created by the parser from the input SQL text
  – Expressed as a relational algebra tree
  – Each SQL query has many possible logical plans

• Physical plans:
  – Goal is to choose an efficient implementation for each operator in the RA tree
  – Each logical plan has many possible physical plans
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

**Relational Algebra**

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

Relational algebra expression is also called the “logical query plan”
Logical Plan v.s. Physical Plan

- Logical Plan = a Relational Algebra tree

- Physical Plan = a Logical Plan plus annotation of each operator with an algorithm
Query Optimization and Execution

- Query optimizer:
  - Choose a good logical plan
  - Refine it to a good physical plan
  - Sometimes these steps are intertwined

- Query execution
  - Execute the physical plan
Query Execution
Physical Operators

Relational algebra operators:
• Selection, projection, join, union, difference
• Group-by, distinct, sort

Physical operators:
• For each operators above, several possible algorithms
• Main memory algorithms, or disk-based algorithms
Main Memory Algorithms

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

Propose three physical operators for the join, assuming the tables are in main memory:
1.
2.
3.
Main Memory Algorithms

Logical operator:

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

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \(O(??)\)
2. Merge join \(O(??)\)
3. Hash join \(O(??)\)
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Main Memory Algorithms

Logical operator:

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

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \( O(n^2) \)
2. Merge join \( O(n \log n) \)
3. Hash join \( O(n) \ldots O(n^2) \)
BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

\[ h(x) = x \mod 10 \]

Operations:

- find(103) = ??
- insert(488) = ??

Duplicates OK

WHY ??
BRIEF Review of Hash Tables

• insert(k, v) = inserts a key k with value v

• Many values for one key
  – Hence, duplicate k’s are OK

• find(k) = returns the list of all values v associated to the key k
Query Execution

• Join $R \bowtie S$: e.g. using hash-join:
  – Nested-loop: forall $x$ in $R$ forall $y$ in $S$ do …
  – Hash-join: build a hash table on $S$, probe $R$
• Selection: $\sigma(R)$: e.g. “on-the-fly”

• But what about a larger plan?
  – Each operator implements the Iterator Interface
Implementing Query Operators with the Iterator Interface

Each operator implements three methods:

- open()
- next()
- close()
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```java
interface Operator {

    void open (...);

    Tuple next ();

    void close ();

}
```

```java
class Select implements Operator {

    void open (Predicate p, Iterator child) {
        this.p = p;
        this.child = child;
    }

    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 ();
    }

}
```
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

interface Operator {

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

}
Implementing Query Operators with the Iterator Interface

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 ();

    void close ();

}
```

CSE 414 - Autumn 2018 22
Implementing Query Operators with the Iterator Interface

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 ();
}
```
Implementing Query Operators with the Iterator Interface

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 child) {
        this.p = p; this.child = child;
    }

    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
Implementing Query Operators with the Iterator Interface

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 child) {
        this.p = p; this.child = child;
    }
    Tuple next () {
    }
}
```
Implementing Query Operators with the Iterator Interface

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 child) {
        this.p = p; this.child = child;
    }

    Tuple next () {
        boolean found = false;
        Tuple r = null;
        while (!found) {
            r = child.next();
            if (r == null) break;
            found = p(in);
        }
    }
}
```

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 child) {
        this.p = p; this.child = child;
    }

    Tuple next () {
        boolean found = false;
        Tuple r = null;
        while (!found) {
            r = child.next();
            if (r == null) break;
            found = p(in);
        }
    }
}
```
Implementing Query Operators with the Iterator Interface

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 child) {
        this.p = p; this.child = child;
    }

    Tuple next () {
        boolean found = false;
        Tuple r = null;
        while (!found) {
            r = child.next();
            if (r == null) break;
            found = p(in);
        }
        return r;
    }
}
```

Example “on the fly” selection operator
Implementing Query Operators with the Iterator Interface

```
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 child) {
        this.p = p; this.child = child;
    }

    Tuple next () {
        boolean found = false;
        Tuple r = null;
        while (!found) {
            r = child.next();
            if (r == null) break;
            found = p(in);
        }
        return r;
    }

    void close () {
        child.close();
    }
}
```

Example “on the fly” selection operator
Implementing Query Operators with the Iterator Interface

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

Pipelining

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

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

(Nested loop)  \[\text{sno} = \text{sno}\]

Suppliers  (File scan)
Supplies  (File scan)

Discuss: open/next/close for nested loop join
Suppliers

Supplies

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

\( \text{Supply}(\text{sid, pno, quantity}) \)

**Pipelining**

(On the fly)

(On the fly)

(Nested loop)

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

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

\( \text{sno} = \text{sno} \)

Suppliers

Supplies

(File scan)

(File scan)

Discuss: open/next/close for nested loop join
Suppliers

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

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

Supplies

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

Discuss: open/next/close for nested loop join
Suppliers

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

Supplies

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

(On the fly)

(On the fly)

(Nested loop)

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

**Pipelining**

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

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

(Nested loop) \(sno = sno\)

Discuss: open/next/close for nested loop join
Suppliers

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

Supplies

\( \text{Supply}(sid, pno, \text{quantity}) \)

**Pipelining**

(On the fly)

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

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

Discuss: open/next/close for nested loop join

\[
\begin{align*}
\text{Suppliers} & \quad \text{Supplies} \\
\text{Open (File scan)} & \quad \text{Open (File scan)} \\
\text{Open} & \quad \text{Open}
\end{align*}
\]
Suppliers
\( \sigma \text{scity} = \text{'Seattle'} \) and \( \text{sstate} = \text{'WA'} \) and \( \text{pno} = 2 \)

Supplies
(On the fly)

(On the fly)

(Nested loop)

Suppliers
(File scan)

Supplies
(File scan)

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

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

\[
\Pi_{\text{sname}} \left( \sigma_{\text{scity}=\text{Seattle} \text{ and } sstate=\text{WA'} \text{ and } pno=2} \right)
\]

\[
\text{nested loop join}
\]

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

Pipelining

(On the fly)

(On the fly)

(Nested loop)

Discuss: open/next/close for nested loop join

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

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

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

Suppliers
(File scan)

Supplies
(File scan)
Supplier\((s_{id},\ s_{name},\ sc_{ity},\ s_{state})\)
Supply\((s_{id},\ p_{no},\ quantity)\)

Pipelining

\(\mathit{Suppliers}\)
\(\mathit{Supplies}\)

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

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

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

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

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

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

\(\Pi_{s_{name}}\)

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

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

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

(On the fly)

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

(Nested loop) \( \sigma_{sno = sno} \)

Discuss: open/next/close for nested loop join

Suppliers (File scan)

Supplies (File scan)
Suppliers
(Sid, sname, scity, sstate)
Supply(Sid, pno, quantity)

Next

\[ \Pi_{sname} \]

Next

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

Next

Sno = Sno

Next

 Suppliers
(File scan)

Next

 Supplies
(File scan)

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

Pipelining

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

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

\(\text{sno} = \text{sno}\)

\(\text{Suppliers (File scan)}\)

\(\text{Supplies (File scan)}\)

Discuss hash-join in class
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Pipelining

(On the fly) π_{sname}

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

(Hash Join) sno = sno

Suppliers (File scan)

Tuples from here are pipelined

Supplies (File scan)

Discuss hash-join in class
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Pipelining

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

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

(Hash Join) \( \sigma_{\text{sno} = \text{sno}} \)

Suppliers (File scan)
Supplies (File scan)

Discuss hash-join in class

Tuples from here are "blocked"

Tuples from here are pipelined
Pipeline v.s. Blocking

• Pipeline
  – A tuple moves all the way through up the query plan
  – Advantages: speed
  – Disadvantage: need all hash at the same time in memory

• Blocking
  – The entire result of the subplan is computed (and stored to disk) before the first tuple is sent up the plan
  – Advantage: saves memory
  – Disadvantage: slower
Discussion on Physical Plan

More components of a physical plan:

• **Access path selection** for each relation
  – Scan the relation or use an index (next lecture)

• **Implementation choice** for each operator
  – Nested loop join, hash join, etc.

• **Scheduling decisions** for operators
  – Pipelined execution or intermediate materialization