Lecture 17: Query execution

Wednesday, May 12, 2010
Outline of Next Few Lectures

• Query execution

• Query optimization
Steps of the Query Processor

1. Parse & Rewrite Query
2. Select Logical Plan
3. Select Physical Plan
4. Query Execution

Query optimization

SQL query

Logical plan

Physical plan

Disk
Example Database Schema

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

View: Suppliers in Seattle
CREATE VIEW NearbySupp AS
SELECT sno, sname
FROM Supplier
WHERE scity='Seattle' AND sstate='WA'
Example Query

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

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
  FROM Supplies
  WHERE pno = 2 )
```
Steps in Query Evaluation

• **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, etc.
Rewritten Version of Our Query

Original query:
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2 )

Rewritten query:
SELECT S.sname
FROM Supplier S, Supplies U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno
AND U.pno = 2;
Continue with Query Evaluation

• **Step 3: Query optimization**
  – Find an efficient query plan for executing the query

• **A query plan is**
  – *Logical query plan*: an extended relational algebra tree
  – *Physical query plan*: with additional annotations at each node
    • Access method to use for each relation
    • Implementation to use for each relational operator
Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Join $\Join$
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$
- Rename $\rho$
Logical Query Plan

\[ \Pi_{sname} \]

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

\[ \sigma_{sccity='Seattle'} \]

\[ \text{Suppliers} \]

\[ \text{Supplies} \]
Query Block

- Most optimizers operate on individual query blocks
- A query block is an SQL query with no nesting
  - Exactly one
    - SELECT clause
    - FROM clause
  - At most one
    - WHERE clause
    - GROUP BY clause
    - HAVING clause
Typical Plan for Block (1/2)

\[ \text{SELECT-PROJECT-JOIN Query} \]

Diagram:
- \( R \) and \( S \) are joined by a join condition.
- The result is filtered by a selection condition \( \sigma \).
- The selected fields are projected by \( \pi \).
Typical Plan For Block (2/2)

$\sigma$\text{having-condition}

$\gamma$ fields, sum/count/min/max(fields)

$\sigma$ selection condition

join condition

\ldots \ldots
How about Subqueries?

```
SELECT  Q.sno  
FROM    Supplier Q  
WHERE   Q.sstate = 'WA'  
    and not exists
      SELECT *
      FROM Supply P  
WHERE   P.sno = Q.sno  
    and P.price > 100
```
How about Subqueries?

```
SELECT  Q.sno
FROM    Supplier Q
WHERE   Q.sstate = 'WA'
        and not exists
        SELECT *
            FROM Supply P
            WHERE P.sno = Q.sno
            and P.price > 100
```

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
How about Subqueries?

```
SELECT Q.sno
FROM Supplier Q
WHERE Q.sstate = 'WA'
    and not exists
    SELECT *
    FROM Supply P
    WHERE P.sno = Q.sno
    and P.price > 100
```
How about Subqueries?

```
(SELECT Q.sno
 FROM Supplier Q
 WHERE Q.sstate = 'WA')
 EXCEPT
(SELECT P.sno
 FROM Supply P
 WHERE P.price > 100)
```

```
SELECT Q.sno
 FROM Supplier Q
 WHERE Q.sstate = 'WA'
 and Q.sno not in
 SELECT P.sno
 FROM Supply P
 WHERE P.price > 100
```
How about Subqueries?

\[
\begin{align*}
\text{(SELECT } & \text{ Q.sno} \\
\text{FROM } & \text{ Supplier Q} \\
\text{WHERE } & \text{ Q.sstate = \textquote{WA}}) \\
\text{EXCEPT} & \text{ (SELECT P.sno} \\
\text{FROM } & \text{ Supply P} \\
\text{WHERE } & \text{ P.price > 100})
\end{align*}
\]
Physical Query Plan

• Logical query plan with extra annotations

• Access path selection for each relation
  – Use a file scan or use an index

• Implementation choice for each operator

• Scheduling decisions for operators
Physical Query Plan

(On the fly)

\( \pi \text{ sname} \)

(On the fly)

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

(Nested loop)

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

Suppliers (File scan)

Supplies (File scan)
Final Step in Query Processing

- **Step 4: Query execution**
  - How to synchronize operators?
  - How to pass data between operators?

- What techniques are possible?
  - One thread per process
  - Iterator interface
  - Pipelined execution
  - Intermediate result materialization
Iterator Interface

• Each operator implements this interface
• Interface has only three methods
  • `open()`
    – Initializes operator state
    – Sets parameters such as selection condition
  • `get_next()`
    – Operator invokes `get_next()` recursively on its inputs
    – Performs processing and produces an output tuple
  • `close()`: cleans-up state
Pipelined Execution

• Applies parent operator to tuples directly as they are produced by child operators

• Benefits
  – No operator synchronization issues
  – Saves cost of writing intermediate data to disk
  – Saves cost of reading intermediate data from disk
  – Good resource utilizations on single processor

• This approach is used whenever possible
Pipelined Execution

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

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

(Nested loop) $sno = sno$

Suppliers (File scan)

Supplies (File scan)

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
Intermediate Tuple Materialization

- Writes the results of an operator to an intermediate table on disk

- No direct benefit but
- Necessary for some operator implementations
- When operator needs to examine the same tuples multiple times
Intermediate Tuple Materialization

\[
\text{Suppliers} \quad \pi_{\text{sname}} \quad \text{Supplies}
\]

\[
\sigma_{\text{sscity}='Seattle' \land \text{sstate}='WA'}
\]

\[
\sigma_{\text{pno}=2}
\]

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

\[
\text{(Sort-merge join)}
\]

\[
\text{(Scan: write to T1)}
\]

\[
\text{(Scan: write to T2)}
\]
Physical Operators

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join
Question in Class

Logical operator:
\[ \text{Supply}(sno,pno,price) \bowtie_{pno=pno} \text{Part}(pno,pname,psize,pcolor) \]

Propose three physical operators for the join, assuming the tables are in main memory:
1. 
2. 
3. 
Logical operator:
\( \text{Supply}(sno,pno,price) \bowtie_{pno=pno} \text{Part}(pno,pname,psize,pcolor) \)

Propose three physical operators for the join, assuming the tables are in main memory:
1. Nested Loop Join
2. Merge join
3. Hash join
1. Nested Loop Join

for S in Supply do {
    for P in Part do {
        if (S.pno == P.pno) output(S,P);
    }
}

Supply = outer relation
Part = inner relation
Note: sometimes terminology is switched

Would it be more efficient to choose Part=inner, Supply=outer? What if we had an index on Part.pno?
It’s more complicated…

• Each **operator** implements this interface
  • `open()`
  • `get_next()`
  • `close()`
Nested Loop Join Revisited

\[
\begin{align*}
\text{open}( ) \{ \\
\quad & \text{Supply.open( );} \\
\quad & \text{Part.open( );} \\
\quad & \text{S = Supply.get_next( );} \\
\}\n\end{align*}
\]

\[
\begin{align*}
\text{close}( ) \{ \\
\quad & \text{Supply.close ( );} \\
\quad & \text{Part.close ( );} \\
\}\n\end{align*}
\]

\[
\begin{align*}
\text{get\_next}( ) \{ \\
\quad & \text{repeat} \{ \\
\quad & \qquad \text{P= Part.get\_next( );} \\
\quad & \qquad \text{if (P== NULL)} \\
\quad & \qquad \qquad \{ \text{Part.close();} \\
\quad & \qquad \qquad \quad \text{S= Supply.get\_next( );} \\
\quad & \qquad \qquad \quad \text{if (S== NULL) return NULL;} \\
\quad & \qquad \qquad \quad \text{Part.open( );} \\
\quad & \qquad \qquad \quad \text{P= Part.get\_next( );} \\
\quad & \text{\} until (S.pno == P.pno);} \\
\quad & \text{return (S, P)}
\}\n\end{align*}
\]

ALL operators need to be implemented this way!
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

- \text{insert}(k, v) = \text{inserts a key } k \text{ with value } v

- Many values for one key
  - Hence, duplicate \textit{k’s} are OK

- \text{find}(k) = \text{return} the \textit{list} of all values \textit{v} associated to the key \textit{k}
2. Hash Join (main memory)

for S in Supply do
    insert(S.pno, S);

for P in Part do {
    LS = find(P.pno);
    for S in LS do {
        output(S, P);
    }
}

Recall: need to rewrite as open, get_next, close
3. Merge Join (main memory)

Part1 = sort(Part, pno);
Supply1 = sort(Supply, pno);
P = Part1.get_next(); S = Supply1.get_next();

While (P!=NULL and S!=NULL) {
    case:
        P.pno > S.pno:  P = Part1.get_next();
        P.pno < S.pno:  S = Supply1.get_next();
        P.pno == S.pno { output(P,S);
                         S = Supply1.get_next();
                         }
    }

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

Why ???