CSE544
Query Execution
Thursday, February 2\textsuperscript{nd}, 2011
Outline

• Relational Algebra: Ch. 4.2
• Overview of query evaluation: Ch. 12
• Evaluating relational operators: Ch. 14

• Shapiro’s paper
The WHAT and the HOW

• In SQL we write WHAT we want to get form the data

• The database system needs to figure out HOW to get the data we want

• The passage from WHAT to HOW goes through the Relational Algebra
SQL = WHAT

Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
x.price > 100 and z.city = ‘Seattle’

It’s clear WHAT we want, unclear HOW to get it
Relational Algebra = HOW

\[
\begin{align*}
\text{Product}(\text{pid, name, price}) \\
\text{Purchase}(\text{pid, cid, store}) \\
\text{Customer}(\text{cid, name, city}) \\
\end{align*}
\]

**Temporary tables**

T1, T2, . . .

**Final answer**

T4(name, name)

\[
\begin{align*}
\text{Final answer} & \quad \text{price} > 100 \text{ and } \text{city} = \text{‘Seattle’} \\
\text{T1}(\text{pid, name, price, pid, cid, store}) & \quad \text{cid} = \text{cid} \\
\text{T2}(\ldots) & \quad \text{x.name, z.name} \\
\text{T3}(\ldots) & \quad \text{T4(name, name)} \\
\end{align*}
\]
The order is now clearly specified:

Iterate over PRODUCT…
…join with PURCHASE…
…join with CUSTOMER…
…select tuples with Price>100 and City='Seattle'…
…eliminate duplicates…
…and that’s the final answer!
Sets v.s. Bags

- Sets: \{a,b,c\}, \{a,d,e,f\}, \{\}\ldots
- Bags: \{a, a, b, c\}, \{b, b, b, b, b\}, \ldots

Relational Algebra has two semantics:
- Set semantics
- Bag semantics
Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference -
- Selection $\sigma$
- Projection $\Pi$
- Join $\Join$
- Rename $\rho$
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$
Relational Algebra (1/3)

The Basic Five operators:

• Union: \( U \)
• Difference: \( - \)
• Selection: \( \sigma \)
• Projection: \( \Pi \)
• Join: \( \Join \)
Relational Algebra (2/3)

Derived or auxiliary operators:

- Renaming: \( \rho \)
- Intersection, complement
- Variations of joins
  - natural, equi-join, theta join, semi-join, cartesian product
Relational Algebra (3/3)

Extensions for bags:

• Duplicate elimination: $\delta$
• Group by: $\gamma$
• Sorting: $\tau$
Union and Difference

\[ R_1 \cup R_2 \]
\[ R_1 \setminus R_2 \]

What do they mean over bags?
What about Intersection?

- Derived operator using minus
  \[ R_1 \cap R_2 = R_1 - (R_1 - R_2) \]

- Derived using join (will explain later)
  \[ R_1 \cap R_2 = R_1 \bowtie R_2 \]
Selection

• Returns all tuples which satisfy a condition

\[ \sigma_c(R) \]

• Examples
  - \( \sigma_{\text{Salary} > 40000} \) (Employee)
  - \( \sigma_{\text{name} = "Smith"} \) (Employee)

• The condition c can be \(=, <, \leq, >, \geq, <>\)
### Employee

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>200000</td>
</tr>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>500000</td>
</tr>
</tbody>
</table>

\( \sigma_{\text{Salary} > 40000} (\text{Employee}) \)

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>500000</td>
</tr>
</tbody>
</table>
Projection

• Eliminates columns

\[ \Pi_{A_1, \ldots, A_n}(R) \]

• Example: project social-security number and names:
  - \[ \Pi_{\text{SSN}, \text{Name}}(\text{Employee}) \]
  - \[ \text{Answer}(\text{SSN}, \text{Name}) \]

Semantics differs over set or over bags
### Employee

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>5423341</td>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

\[ \Pi_{\text{Name,Salary}} (\text{Employee}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

#### Bag semantics

#### Set semantics

Which is more efficient to implement?
Cartesian Product

- Each tuple in $R_1$ with each tuple in $R_2$

$$R_1 \times R_2$$

- Very rare in practice; mainly used to express joins
<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>John</td>
<td>9999999999</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>
Renaming

• Changes the schema, not the instance

\[ \rho_{B_1, \ldots, B_n}(R) \]

• Example:
  - \( \rho_{N, S}(\text{Employee}) \rightarrow \text{Answer}(N, S) \)
Natural Join

\[ R_1 \Join R_2 \]

- Meaning: \[ R_1 \Join R_2 = \Pi_A(\sigma(R_1 \times R_2)) \]

- Where:
  - The selection \( \sigma \) checks equality of all common attributes
  - The projection eliminates the duplicate common attributes
Natural Join

\[ R \Join S = \Pi_{ABC} (\sigma_{R.B=S.B} (R \times S)) \]
Natural Join

• Given the schemas R(A, B, C, D), S(A, C, E), what is the schema of R $\bowtie$ S?

• Given R(A, B, C), S(D, E), what is R $\bowtie$ S?

• Given R(A, B), S(A, B), what is R $\bowtie$ S?
Theta Join

- A join that involves a predicate

\[ R_1 \bowtie_{\theta} R_2 = \sigma_{\theta} (R_1 \times R_2) \]

- Here \( \theta \) can be any condition
Eq-join

- A theta join where $\theta$ is an equality

$$R1 \bowtie_{A=B} R2 = \sigma_{A=B} (R1 \times R2)$$

- This is by far the most used variant of join in practice
So Which Join Is It?

• When we write $R \bowtie S$ we usually mean an eq-join, but we often omit the equality predicate when it is clear from the context.
Semijoin

\[ R \bowtie_C S = \Pi_{A_1, \ldots, A_n} (R \bowtie_C S) \]

- Where \( A_1, \ldots, A_n \) are the attributes in \( R \)

Formally, \( R \bowtie_C S \) means this: retain from \( R \) only those tuples that have some matching tuple in \( S \)
- Duplicates in \( R \) are preserved
- Duplicates in \( S \) don’t matter
Semijoins in Distributed Databases

Task: compute the query with minimum amount of data transfer

Employee \( \bowtie_{\text{SSN}=\text{EmpSSN}} (\sigma_{\text{age}>71} (\text{Dependent})) \)

Assumptions: Very few Employees have dependents. Very few dependents have age > 71. “Stuff” is big.
Semijoins in Distributed Databases

Employee \Join_{SSN=EmpSSN} (\sigma_{age>71}(Dependent))

T(SSN) = \prod_{SSN} \sigma_{age>71}(Dependents)
Semijoins in Distributed Databases

\[
\begin{array}{c|c|c|c}
\text{SSN} & \text{Name} & \text{Stuff} \\
\hline
\ldots & \ldots & \ldots \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{EmpSSN} & \text{DepName} & \text{Age} & \text{Stuff} \\
\hline
\ldots & \ldots & \ldots & \ldots \\
\end{array}
\]

Employee \bowtie_{\text{SSN}=\text{EmpSSN}} (\sigma_{\text{age}>71} (\text{Dependent}))

T(\text{SSN}) = \Pi_{\text{SSN}} \sigma_{\text{age}>71} (\text{Dependents})

R = Employee \bowtie_{\text{SSN}=\text{EmpSSN}} = Employee \bowtie_{\text{SSN}=\text{EmpSSN}} (\sigma_{\text{age}>71} (\text{Dependents}))

Semijoins in Distributed Databases

**Employee**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Dependent**

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
<th>Age</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

network

\[ T(SSN) = \Pi_{SSN} \sigma_{\text{age}>71} (\text{Dependent}) \]

\[ R = \text{Employee} \bowtie_{SSN=\text{EmpSSN}} T \]

**Answer**

\[ R \bowtie_{SSN=\text{EmpSSN}} \text{Dependents} \]
Joins R US

• The join operation in all its variants (eq-join, natural join, semi-join, outer-join) is at the heart of relational database systems

• WHY?
Operators on Bags

• Duplicate elimination $\delta$
  $\delta(R) = \text{select distinct * from } R$

• Grouping $\gamma$
  $\gamma_{A,\text{sum}(B)}(R) = \text{select A,\text{sum}(B) from } R \text{ group by } A$

• Sorting $\tau$
Complex RA Expressions

\[ \gamma \sigma_{\text{name} = \text{fred}} \Pi \sigma_{\text{name} = \text{gizmo}} \]

\[ \Pi_{\text{ssn}} \sigma_{\text{name} = \text{fred}} \Pi_{\text{pid}} \sigma_{\text{name} = \text{gizmo}} \]

\[ \Pi_{\text{ssn}} \]
RA = Dataflow Program

• Several operations, plus strictly specified order

• In RDBMS the dataflow graph is always a tree

• Novel applications (s.a. PIG), dataflow graph may be a DAG
Limitations of RA

• Cannot compute “transitive closure”

<table>
<thead>
<tr>
<th>Name1</th>
<th>Name2</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>Mary</td>
<td>Father</td>
</tr>
<tr>
<td>Mary</td>
<td>Joe</td>
<td>Cousin</td>
</tr>
<tr>
<td>Mary</td>
<td>Bill</td>
<td>Spouse</td>
</tr>
<tr>
<td>Nancy</td>
<td>Lou</td>
<td>Sister</td>
</tr>
</tbody>
</table>

• Find all direct and indirect relatives of Fred
• Cannot express in RA !!! Need to write Java program
• Remember *the Bacon number* ? Needs TC too !
Steps of the Query Processor

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

SQL query

Query optimization

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

```sql
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:

```sql
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
FROM Supplies
WHERE pno = 2 )
```

Rewritten query:

```sql
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 $\bowtie$
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$
- Rename $\rho$
Logical Query Plan

\[ \Pi_{\text{sname}} \]
\[ \sigma_{\text{sscity}='Seattle' \land \text{sstate}='WA' \land \text{pno}=2} \]
\[ \text{sno} = \text{sno} \]

Suppliers\quad 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)

\[
\text{SELECT-PROJECT-JOIN Query}
\]
Typical Plan For Block (2/2)

\[ \sigma_{\text{having-ondition}} \]

\[ \gamma \text{ fields, sum/count/min/max(fields)} \]

\[ \sigma \text{ selection condition} \]

join condition

\[ \ldots \]

\[ \ldots \]
How about Subqueries?

\[
\begin{align*}
\text{SELECT} & \quad Q.sno \\
\text{FROM} & \quad \text{Supplier } Q \\
\text{WHERE} & \quad Q.sstate = 'WA' \\
\text{and} & \quad \text{not exists} \\
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Supply } P \\
\text{WHERE} & \quad P.sno = Q.sno \\
\text{and} & \quad P.price > 100
\end{align*}
\]
How about Subqueries?

```sql
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?

\[
\begin{align*}
\text{SELECT} & \quad Q.sno \\
\text{FROM} & \quad \text{Supplier} \ Q \\
\text{WHERE} & \quad Q.sstate = 'WA' \\
& \quad \text{and not exists} \\
& \quad \text{SELECT} * \\
& \quad \text{FROM} \ P \\
& \quad \text{WHERE} \ P.sno = Q.sno \\
& \quad \text{and} \ P.\text{price} > 100
\end{align*}
\]

De-Correlation

\[
\begin{align*}
\text{SELECT} & \quad Q.sno \\
\text{FROM} & \quad \text{Supplier} \ Q \\
\text{WHERE} & \quad Q.sstate = 'WA' \\
& \quad \text{and} \ Q.sno \ \text{not in} \\
& \quad \text{SELECT} P.sno \\
& \quad \text{FROM} \ P \\
& \quad \text{WHERE} \ P.\text{price} > 100
\end{align*}
\]
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?

\[
\text{(SELECT } Q.sno \\
\text{FROM Supplier } Q \\
\text{WHERE } Q.sstate = 'WA') \\
\text{EXCEPT} \\
\text{(SELECT } P.sno \\
\text{FROM Supply } P \\
\text{WHERE } P.price > 100) \\
\text{EXCEPT}
\]
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

\( \pi_{\text{sname}} \)
\( \sigma_{\text{sscity}='Seattle' \land \text{sstate}='WA' \land \text{pno}=2} \)
\( \text{(Nested loop)} \)
\( \text{sno} = \text{sno} \)
\( \text{Suppliers} \)
\( \text{(File scan)} \)
\( \text{Supplies} \)
\( \text{(File scan)} \)

\text{Supplier(sno,sname,scity,sstate)}
\text{Part(pno,pname,psize,pcolor)}
\text{Supply(sno,pno,price)}
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 query
  – 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

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

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

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

Suppliers (File scan)

Supplies (File scan)
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 utilisations on single processor

• This approach is used whenever possible
Intermediate Tuple Materialization

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

\(\sigma_{\text{scity}= \text{‘Seattle’} \land \text{sstate}= \text{‘WA’}}\)

\(\sigma_{\text{pno}=2}\)

File scan

\(\sigma_{\text{pno}=2}\)

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

\(\sigma_{\text{scity}= \text{‘Seattle’} \land \text{sstate}= \text{‘WA’}}\)

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)

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

File scan

\(\sigma_{\text{pno}=2}\)
Intermediate Tuple Materialization

- Writes the results of an operator to an intermediate table on disk
- No direct benefit but
  - Necessary data is larger than main memory
  - Necessary when operator needs to examine the same tuples multiple times
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:
\[ \text{Supply}(sno,pno,price) \Join_{pno=pno} \text{Part}(pno,pname,psize,pcolor) \]

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

1.
2.
3.
Logical operator:
Supply(sno,pno,price) \Join_{pno=pno} 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()
Main Memory Nested Loop
Join Revisited

open ( ) {
    Supply.open( );
    Part.open( );
    S = Supply.get_next( );
}

close ( ) {
    Supply.close ( );
    Part.close ( );
}

get_next( ) {
    repeat {
        P= Part.get_next( );
        if (P== NULL) {
            Part.close();
            S= Supply.get_next( );
            if (S== NULL) return NULL;
            Part.open( );
            P= Part.get_next( );
        }
    } until (S.pno == P.pno);
    return (S, P)
}

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

• 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
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

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
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)
Main Memory Group By

Grouping:

Product(name, department, quantity)

\( \gamma_{\text{department, sum(quantity)}} \) (Product) \rightarrow 
Answer(department, sum)

Main memory hash table

Question: How?
Duplicate Elimination IS
Group By

Duplicate elimination $\delta(R)$ is \textit{the same} as group by $\gamma(R)$ WHY ???

- Hash table in main memory

- Cost: $B(R)$
- Assumption: $B(\delta(R)) \leq M$
Selections, Projections

• Selection = easy, check condition on each tuple at a time

• Projection = easy (assuming no duplicate elimination), remove extraneous attributes from each tuple
Review (1/2)

• Each **operator implements this interface**
  
  • **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
Review (2/2)

• Three algorithms for main memory join:
  – Nested loop join
  – Hash join
  – Merge join

• Algorithms for selection, projection, group-by

If $|R| = m$ and $|S| = n$, what is the asymptotic complexity for computing $R \bowtie S$?
External Memory Algorithms

• Data is too large to fit in main memory

• Issue: disk access is 3-4 orders of magnitude slower than memory access

• Assumption: runtime dominated by # of disk I/O’s; will ignore the main memory part of the runtime
Cost Parameters

The cost of an operation = total number of I/Os

Cost parameters:

- $B(R)$ = number of blocks for relation $R$
- $T(R)$ = number of tuples in relation $R$
- $V(R, a)$ = number of distinct values of attribute $a$
- $M$ = size of main memory buffer pool, in blocks

Facts: (1) $B(R) << T(R)$:
(2) When $a$ is a key, $V(R,a) = T(R)$
When $a$ is not a key, $V(R,a) << T(R)$
Ad-hoc Convention

• We assume that the operator *reads* the data from disk
• We assume that the operator *does not write* the data back to disk (e.g.: pipelining)
• Thus:

Any main memory join algorithms for $R \bowtie S$: Cost = $B(R) + B(S)$

Any main memory grouping $\gamma(R)$: Cost = $B(R)$
Sequential Scan of a Table R

- When R is *clustered*
  - Blocks consists only of records from this table
  - $B(R) << T(R)$
  - Cost = $B(R)$

- When R is *unclustered*
  - Its records are placed on blocks with other tables
  - $B(R) \approx T(R)$
  - Cost = $T(R)$
Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$

```plaintext
for each tuple $r$ in $R$ do
  for each tuple $s$ in $S$ do
    if $r$ and $s$ join then output $(r,s)$
```

- Cost: $T(R) B(S)$ when $S$ is clustered
- Cost: $T(R) T(S)$ when $S$ is unclustered
Examples

M = 4; R, S are clustered

• Example 1:
  – B(R) = 1000, T(R) = 10000
  – B(S) = 2, T(S) = 20
  – Cost = ?

• Example 2:
  – B(R) = 1000, T(R) = 10000
  – B(S) = 4, T(S) = 40
  – Cost = ?
Block-Based Nested-loop Join

for each (M-2) blocks bs of S do
  for each block br of R do
    for each tuple s in bs
      for each tuple r in br do
        if “r and s join” then output(r,s)

Terminology alert: book calls S the *inner* relation
Block Nested-loop Join

R & S

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

Input buffer for R

Output buffer

Join Result
Examples

M = 4; R, S are clustered

• Example 1:
  – B(R) = 1000, T(R) = 10000
  – B(S) = 2, T(S) = 20
  – Cost = B(S) + B(R) = 1002

• Example 2:
  – B(R) = 1000, T(R) = 10000
  – B(S) = 4, T(S) = 40
  – Cost = B(S) + 2B(R) = 2004

Note: T(R) and T(S) are irrelevant here.
Cost of Block Nested-loop Join

- Read S once: cost $B(S)$
- Outer loop runs $B(S)/(M-2)$ times, and each time need to read R: costs $B(S)B(R)/(M-2)$

Cost = $B(S) + B(S)B(R)/(M-2)$
Index Based Selection

Recall IMDB; assume indexes on Movie.id, Movie.year

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Movie} \\
\text{WHERE} & \quad \text{id} = '12345' \\
\end{align*}
\]

\[
\begin{align*}
\text{B(Movie)} & = 10k \\
\text{T(Movie)} & = 1M \\
\end{align*}
\]

What is your estimate of the I/O cost?

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Movie} \\
\text{WHERE} & \quad \text{year} = '1995' \\
\end{align*}
\]
Index Based Selection

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

- Clustered index on a: cost $B(R)/V(R,a)$
- Unclustered index: cost $T(R)/V(R,a)$
Index Based Selection

• Example:
  - Table scan (assuming R is clustered):
    - \( B(R) = 10k \)
    - \( T(R) = 1M \)
    - \( V(R, a) = 100 \)
  - Index based selection:
    - If index is clustered: \( B(R)/V(R, a) = 100 \) I/Os
    - If index is unclustered: \( T(R)/V(R, a) = 10000 \) I/Os

Rule of thumb:
don’t build unclustered indexes when \( V(R, a) \) is small!
Index Based Join

- $R \bowtie S$
- Assume $S$ has an index on the join attribute

\[
\text{for each tuple } r \text{ in } R \text{ do}
\]
\[
\text{lookup the tuple(s) } s \text{ in } S \text{ using the index}
\]
\[
\text{output } (r,s)
\]
Index Based Join

Cost (Assuming R is clustered):

- If index is clustered: $B(R) + T(R)B(S)/V(S,a)$
- If unclustered: $B(R) + T(R)T(S)/V(S,a)$
Operations on Very Large Tables

• Compute \( R \bowtie S \) when each is larger than main memory

• Two methods:
  – Partitioned hash join (many variants)
  – Merge-join

• Similar for grouping
Partitioned Hash-based Algorithms

Idea:

• If $B(R) > M$, then partition it into smaller files: $R_1, R_2, R_3, \ldots, R_k$

• Assuming $B(R_1) = B(R_2) = \ldots = B(R_k)$, we have $B(R_i) = B(R)/k$

• Goal: each $R_i$ should fit in main memory: $B(R_i) \leq M$

How big can $k$ be?
Partitioned Hash Algorithms

- Idea: partition a relation R into M-1 buckets, on disk
- Each bucket has size approx. $B(R)/(M-1) \approx B(R)/M$

Assumption: $B(R)/M \leq M$, i.e. $B(R) \leq M^2$
Grouping

• $\gamma(R) = \text{grouping and aggregation}$
• Step 1. Partition $R$ into buckets
• Step 2. Apply $\gamma$ to each bucket (may read in main memory)

• Cost: $3B(R)$
• Assumption: $B(R) \leq M^2$
Partitioned Hash Join

GRACE Join

$R \bowtie S$

- Step 1:
  - Hash $S$ into $M$ buckets
  - send all buckets to disk

- Step 2
  - Hash $R$ into $M$ buckets
  - Send all buckets to disk

- Step 3
  - Join every pair of buckets
**Grace-Join**

- Partition both relations using hash fn $h$: R tuples in partition $i$ will only match S tuples in partition $i$.

- Read in a partition of R, hash it using $h2 (<> h!)$. Scan matching partition of S, search for matches.
Grace Join

• Cost: $3B(R) + 3B(S)$
• Assumption: $\min(B(R), B(S)) \leq M^2$
External Sorting

• Problem:
• Sort a file of size B with memory M
• Where we need this:
  – ORDER BY in SQL queries
  – Several physical operators
  – Bulk loading of B+-tree indexes.
• Will discuss only 2-pass sorting, when $B < M^2$
External Merge-Sort: Step 1

• Phase one: load M bytes in memory, sort
External Merge-Sort: Step 2

• Merge $M - 1$ runs into a new run
• Result: runs of length $M (M - 1) \approx M^2$

If $B \leq M^2$ then we are done
Cost of External Merge Sort

• $\text{Read+write+read} = 3B(R)$

• Assumption: $B(R) \leq M^2$
Grouping

Grouping: $\gamma_a, \sum(b) (R)$

• Idea: do a two step merge sort, but change one of the steps

• Question in class: which step needs to be changed and how?

Cost = $3B(R)$
Assumption: $B(\delta(R)) \leq M^2$
Merge-Join

Join $R \bowtie S$

- Step 1a: initial runs for $R$
- Step 1b: initial runs for $S$
- Step 2: merge and join
Merge-Join

\[ M_1 = \frac{B(R)}{M} \text{ runs for } R \]
\[ M_2 = \frac{B(S)}{M} \text{ runs for } S \]
Merge-join \( M_1 + M_2 \) runs; need \( M_1 + M_2 \leq M \)
Two-Pass Algorithms Based on Sorting

Join $R \bowtie S$

- If the number of tuples in $R$ matching those in $S$ is small (or vice versa) we can compute the join during the merge phase
- Total cost: $3B(R) + 3B(S)$
- Assumption: $B(R) + B(S) \leq M^2$
Summary of External Join Algorithms

• Block Nested Loop: \( B(S) + B(R) \times B(S)/M \)

• Index Join: \( B(R) + T(R)B(S)/V(S,a) \)

• Partitioned Hash: \( 3B(R)+3B(S); \)
  - \( \min(B(R),B(S)) \leq M^2 \)

• Merge Join: \( 3B(R)+3B(S) \)
  - \( B(R)+B(S) \leq M^2 \)