Outline

- Relational Algebra: Ch. 4.2
- Overview of query evaluation: Ch. 12
- Evaluating relational operators: Ch. 14
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
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'
Relational Algebra = HOW

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

Temporary tables
T1, T2, ...

Product(pid, name, price, pid, cid, store)

Purchase(pid, cid, store)

Customer(cid, name, city)

price > 100 and city = 'Seattle'
cid = cid

Final answer

T4(name, name)

T3( . . . )

T2( . . . )
Relational Algebra = HOW

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\}, \{\}\, . . .
- Bags: \{a, a, b, c\}, \{b, b, b, b, b\}, . . .

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: ▲
- Difference: -
- Selection: σ
- Projection: Π
- 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

$R1 \cup R2$
$R1 - R2$

What do they mean over bags?
What about Intersection?

- Derived operator using minus

\[ R_1 \setminus R_2 = R_1 - (R_1 - R_2) \]

- Derived using join (will explain later)

\[ R_1 \setminus R_2 = R_1 \bowtie \overline{R}_2 \]
Selection

• Returns all tuples which satisfy a condition

\[ \sigma c(R) \]

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

• The condition \( c \) can be \( =, <, \geq, >, \leq, <> \)
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:
  - \( \Pi \text{SSN, Name (Employee)} \)
  - Answer(SSN, Name)

Semantics differs over set or over bags

Dan Suciu -- CSEP544 Fall 2010
<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>

Which is more efficient to implement?
Cartesian Product

- Each tuple in R1 with each tuple in R2

\[
\begin{array}{c}
R1 \\
\times \\
R2
\end{array}
\]

- Very rare in practice; mainly used to express joins
### Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
</tr>
</tbody>
</table>

### Dependent

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Employee X Dependent

<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:
  - \[ \rho \ N, \ S(\text{Employee}) \quad \rightarrow \quad \text{Answer}(N, S) \]
Natural Join

\[
R_1 \Join \Box R_2
\]

- **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

\[ R \bowtie S = \Pi ABC(\sigma R.B=S.B(R \times S)) \]

\[
\begin{array}{|c|c|}
\hline
A & B \\
\hline
X & Y \\
X & Z \\
Y & Z \\
Z & V \\
\hline
\end{array}
\] 

\[
\begin{array}{|c|c|}
\hline
B & C \\
\hline
Z & U \\
V & W \\
Z & V \\
\hline
\end{array}
\] 

\[
\begin{array}{|c|c|c|}
\hline
A & B & C \\
\hline
X & Z & U \\
X & Z & V \\
Y & Z & U \\
Y & Z & V \\
Z & V & W \\
\hline
\end{array}
\]
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

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

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

- A theta join where \( \theta \) is an equality

\[
R_1 \bowtie A=B R_2 = \sigma A=B (R_1 \bowtie R_2)
\]

- 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

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

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

Formally, \( R \Join_C S \) 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

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

Task: compute the query with minimum amount of data transfer

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

Employee ⋈ SSN=EmpSSN (σ age>71 (Dependent))
Semijoins in Distributed Databases

Employee ⨝ SSN=EmpSSN (σ age>71 (Dependents))

T(SSN) = Π SSN σ age>71 (Dependants)
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>

\[ R = \text{Employee} \Join_{\text{SSN}=\text{EmpSSN}} T \]
\[ T(\text{SSN}) = \Pi \text{SSN} \sigma \text{age}>71 \text{ (Dependent)} \]

\[ \text{R} = \text{Employee} \Join_{\text{SSN}=\text{EmpSSN}} T \]
\[ = \text{Employee} \Join_{\text{SSN}=\text{EmpSSN}} (\sigma \text{age}>71 \text{ (Dependent)}) \]
Semijoins in Distributed Databases

Employee network

Dependent

Employee $\bowtie SSN=\text{EmpSSN} \ (\sigma \text{age}>71 \ (\text{Dependents}))$

T(SSN) = \Pi SSN \ (\sigma \text{age}>71 \ (\text{Dependents}))

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

- Sorting $\tau$
Complex RA Expressions

\[ \gamma \text{name=fred} \sigma \text{name=gizmo} \Pi \text{pid} \Pi \text{ssn} \gamma \text{u.name, count(*)} \]

Person x        Purchase y             Person z           Product u

σ\text{name=fred} σ\text{name=gizmo} \Pi \text{pid} \Pi \text{ssn} \gamma \text{u.name, count(*)}
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
5. Disk

Query optimization

SQL query

Logical plan

Physical plan
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

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

 Suppliers

 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)

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

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

\[ \pi \text{ fields} \]

\[ \text{JOIN condition} \]

\[ R \quad S \]

Dan Suciu -- CSEP544 Fall 2010
Typical Plan For Block (2/2)

$\sigma$ having-ondition

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

$\sigma$ selection condition

join condition

... ...
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 LIKE '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

De-Correlation

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?

\[
\begin{align*}
&\text{(SELECT Q.sno} \\
&\text{FROM Supplier Q} \\
&\text{WHERE Q.sstate = 'WA')} \\
&\text{EXCEPT} \\
&\text{(SELECT P.sno} \\
&\text{FROM Supply P} \\
&\text{WHERE P.price > 100)}
\end{align*}
\]
How about Subqueries?

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

\( \pi \text{sname} \)

\( \sigma \text{sscity='Seattle'} \land \text{sstate='WA'} \land \text{pno=2} \)

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

Suppliers (File scan)

Supplies (File scan)

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

\[ \pi \text{sname} \]

\[ \sigma \text{sscity='Seattle' } \land \text{sstate='WA' } \land \text{pno=2} \]

\[ \text{sno = sno} \]

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 utilizations on single processor
- This approach is used whenever possible
Intermediate Tuple Materialization

\( \pi \) sname

\( \sigma \text{ sscity='Seattle' sstate='WA'} \)

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

Suppliers (File scan)

Supplies (File scan)

(On the fly)

(Sort-merge join)

(Scan: write to T1)

(Scan: write to T2)

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 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:

\[
\text{Supply}(sno,pno,price) \quad pno=pno \\
\text{Part}(pno,pname,psize,pcolor)
\]

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

1. 
2. 
3.
Logical operator:

$$\text{Supply}(\text{sno}, \text{pno}, \text{price}) \quad \text{pno} = \text{pno}$$

$$\text{Part}(\text{pno}, \text{pname}, \text{psize}, \text{pcolor})$$

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);
    }
}

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

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)
}
BRIEF Review of Hash Tables

A (naïve) hash function:

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

Operations:

- \( \text{find}(103) = ?? \)
- \( \text{insert}(488) = ?? \)

Separate chaining:
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
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();
    }
}

Why
Main Memory Group By

Grouping:
Product(name, department, quantity)
\( \gamma \)department, sum(quantity) (Product) \( \Rightarrow \)
Answer(department, sum)

Main memory hash table
Question: How ?
Duplicate Elimination IS
Group By

Duplicate elimination $\delta(R)$ is *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
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
Three algorithms for main memory join:
- Nested loop join
- Hash join
- Merge join

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:

\[
\text{Any main memory join algorithms for } R \Join S: \text{Cost} = B(R) + B(S)
\]

\[
\text{Any main memory grouping } \gamma(R): \text{Cost} = B(R)
\]
Sequential Scan of a Table $R$

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

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

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

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

- Cost: $T(R) \cdot B(S)$ when $S$ is clustered
- Cost: $T(R) \cdot T(S)$ 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 = ?

Can you do better?
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\textit{ inner} relation

Dan Suciu -- CSEP544 Fall 2010
Block Nested-loop Join

Input buffer for R

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

Output buffer

Join Result

R & S

...
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

\[
\text{SELET } * \\
\text{FROM Movie} \\
\text{WHERE id = '12345'}
\]

\[
\text{SELET } * \\
\text{FROM Movie} \\
\text{WHERE year = '1995'}
\]

B(Movie) = 10k
T(Movie) = 1M

What is your estimate of the I/O cost?
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):
    - $B(R) = 10k$ I/Os
  - 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

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

- $\gamma(R) =$ 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: $B(R) \leq M2$
Partitioned Hash Join

\[ R \Join S \]

- **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
Hash-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.
Partitioned Hash Join

- Cost: $3B(R) + 3B(S)$
- Assumption: $\min(B(R), B(S)) \leq M2$
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 $M (M - 1)^\geq M^2$

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

• Read+write+read = 3B(R)

• Assumption: B(R) <= M2
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: $B(\delta(R)) \leq M2$
Merge-Join

Join $R \Join S$

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

\[ M1 = \frac{B(R)}{M} \text{ runs for } R \]
\[ M2 = \frac{B(S)}{M} \text{ runs for } S \]

Merge-join \( M1 + M2 \) runs;

need \( M1 + M2 \leq M \)
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: $B(R) + B(S) \leq M2$
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: \( 3B(R) + 3B(S); \)
  - \( \text{min}(B(R),B(S)) \leq M^2 \)

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