Lecture 7:
Query Execution and Optimization
Tuesday, February 20, 2007
Outline

• Relational Algebra: Chapter 4
• Query evaluation: Chapters 12, 13, 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
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

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

Temporary tables T1, T2, . . .

Final answer

T4(name,name)

T3( . . . )

price>100 and city='Seattle'

Customer

Product

Purchase

T1(pid,name,price,pid,cid,store)

T2( . . . )

σ

π

δ

x.name,z.name

Final answer

π

σ

cid=cid

T2( . . . )

T1(pid,name,price,pid,cid,store)
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!
Plan for Today

• Relational Algebra

• Implementation of physical operators

Next lecture:
• Optimizations
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 flavors:

• Over sets: theoretically elegant but limited

• Over bags: needed to expresses SQL queries

We discuss set semantics, and mention bag semantics only where needed
Relational Algebra (1/3)

The Basic Five operators:

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

Derived or auxiliary operators:

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

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

\[ R_1 \cup R_2 \]
\[ R_1 - R_2 \]

What do they mean over bags?
What about Intersection?

• It is a derived operator

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

• Also expressed as a join (will see later)
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 \( =, <, \leq, >, \geq, <> \)
$\sigma_{\text{Salary} > 40000}$ (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>
Projection

• Eliminates columns

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

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

Semantics differs over set or over bags
<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>John</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>200000</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>200000</td>
</tr>
<tr>
<td>John</td>
<td>600000</td>
</tr>
</tbody>
</table>

Set semantics: duplicate elimination automatic
Bag semantics: no duplicate elimination; need explicit $\delta$

### Employee Table

<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>John</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>200000</td>
</tr>
</tbody>
</table>

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

- Each tuple in R1 with each tuple in R2

$$R1 \times R2$$

- 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>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

### Dependent

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Employee $\times$ Dependent

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
<td>777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>John</td>
<td>999999999</td>
<td>777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
<td>999999999</td>
<td>Emily</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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

S

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

R \bowtie S =

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
<td>W</td>
</tr>
</tbody>
</table>
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

\[ R_1 \Join_\theta R_2 = \sigma_\theta (R_1 \times R_2) \]

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

• A theta join where $\theta$ is an equality.

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

• 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

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

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

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

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

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

Answer = \( R \bowtie_{SSN=EmpSSN} \text{Dependents} \)
Operators on Bags

- Duplicate elimination $\delta$
- Grouping $\gamma$
- Sorting $\tau$
Complex RA Expressions

\[ \gamma_{u\text{.name, count(*)}} \]

\[ x\text{.ssn}=y\text{.buyer-ssn} \]

\[ y\text{.pid}=u\text{.pid} \]

\[ y\text{.seller-ssn}=z\text{.ssn} \]

\[ \Pi_{\text{ssn}} \]

\[ \sigma_{\text{name}=\text{fred}} \]

\[ \sigma_{\text{name}=\text{gizmo}} \]

\[ \Pi_{\text{pid}} \]

Person x

Purchase y

Person z

Product u
RA Expressions v.s. Programs

• An Algebra Expression is like a program
  – Several operations
  – Strictly specified order

• But Algebra expressions have limitations
RA and Transitive Closure

- 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!
Query Evaluation Steps

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

SQL query

Query optimization

Disk

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

\[ \pi_{\text{sname}} \sigma_{\text{sscity}='Seattle' \land \text{sstate}='WA' \land \text{pno}=2}(\text{Suppliers} \bowtie_{\text{sno} = \text{sno}} \text{Supplies}) \]
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)

\[
\begin{align*}
\pi & \text{ fields} \\
\sigma & \text{ selection condition} \\
\text{join condition} & \\
\text{join condition} & \\
R & \quad S
\end{align*}
\]

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

\[ \sigma_{\text{having-condition}} \]
\[ \gamma \text{ fields, sum/count/min/max(fields)} \]
\[ \Pi \text{ fields} \]
\[ \sigma \text{ selection condition} \]
\[ \text{join condition} \]

... ...
How about Subqueries?

```sql
SELECT Q.name
FROM Person Q
WHERE Q.age > 25
and not exists
    SELECT *
    FROM Purchase P
WHERE P.buyer = Q.name
    and P.price > 100
```
How about Subqueries?

```sql
SELECT Q.name
FROM Person Q
WHERE Q.age > 25
and not exists
  SELECT *
  FROM Purchase P
  WHERE P.buyer = Q.name
  and P.price > 100
```
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)  \( \sigma \text{ sscity='Seattle' \land sstate='WA' \land pno=2} \)

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

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

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

Suppliers (File scan) Suppliers (File scan)
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

(On the fly)

(Sort-merge join)

(Scan: write to T1)

(Scan: write to T2)

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

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

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

\(sno = sno\)

Suppliers
(File scan)

Supplies
(File scan)
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:

\[ \text{Product}(\text{pname}, \text{cname}) \bowtie \text{Company}(\text{cname}, \text{city}) \]

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

1.
2.
3.
Question in Class

Product(pname, cname) \(\Join\) Company(cname, city)

- 1000000 products
- 1000 companies

How much time do the following physical operators take if the data is in main memory?

- Nested loop join \(\text{time} = \)
- Sort and merge (=“merge-join”) \(\text{time} = \)
- Hash join \(\text{time} = \)
The Iterator Model of Execution

Each operator implements three methods:
• Open()
• GetNext()
• Close()
Cost Parameters

The cost of an operation = total number of I/Os
result assumed to be delivered in main memory

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
Cost Parameters

• *Clustered* table R:
  – Blocks consists only of records from this table
  – $B(R) \ll T(R)$

• *Unclustered* table R:
  – Its records are placed on blocks with other tables
  – $B(R) \approx T(R)$

• When a is a key, $V(R,a) = T(R)$
• When a is not a key, $V(R,a) \ll T(R)$
Selection and Projection

Selection $\sigma(R)$, projection $\Pi(R)$

- Both are *tuple-at-a-time* algorithms
- Cost: $B(R)$

What are $\text{Open}(\ )$, $\text{GetNext}(\ )$, $\text{Close}(\ )$ here?
Hash Tables

• Key data structure used in many operators
• May also be used for indexes, as alternative to B+ trees
• Recall basics:
  – There are n buckets
  – A hash function f(k) maps a key k to \{0, 1, \ldots, n-1\}
  – Store in bucket f(k) a pointer to record with key k
• Secondary storage: bucket = block, use overflow blocks when needed
Hash Table Example

• Assume 1 bucket (block) stores 2 keys + pointers
  • h(e)=0
  • h(b)=h(f)=1
  • h(g)=2
  • h(a)=h(c)=3

Here: h(x) = x mod 4
Searching in a Hash Table

- Search for a:
- Compute \( h(a)=3 \)
- Read bucket 3
- 1 disk access

```
<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
```
Insertion in Hash Table

- Place in right bucket, if space
- E.g. \( h(d) = 2 \)

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>
```
Insertion in Hash Table

• Create overflow block, if no space
• E.g. \( h(k) = 1 \)

![Hash Table Diagram]

• More overflow blocks may be needed
Hash Table Performance

• Excellent, if no overflow blocks
• Degrades considerably when number of keys exceeds the number of buckets (i.e. many overflow blocks).
Main Memory Hash Join

Hash join: \( R \bowtie S \)
- Scan S, build buckets in main memory
- Then scan R and join

- Cost: \( B(R) + B(S) \)
- Assumption: \( B(S) \leq M \)
Main Memory Hash Join

• What are Open( ), GetNext( ), Close( ) ?
Main Memory Hash Join

```java
Open() {
    H = newHashTable();
    S.Open();
x = S.GetNext();
while (x != null) { H.insert(x); x = S.GetNext();}
    S.Close();
    R.Open();
    buffer = [ ];
}
```
Main Memory Hash Join

GetNext( ) {
    while (buffer == [ ]) {
        x = R.GetNext( );
        if (x==Null) return NULL;
        buffer = H.find(x);
    }
    z = buffer.first( );
    buffer = buffer.rest( );
    return z;
}
Main Memory Hash Join

Close() {
    release memory (H, buffer, etc.);
    R.Close()
}

CSEP 544 - Spring, 2009
Duplicate Elimination

Duplicate elimination $\delta(R)$

- Hash table in main memory

- Cost: $B(R)$

- Assumption: $B(\delta(R)) \leq M$
Grouping

Grouping:

Product(name, department, quantity)

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

Main memory hash table

Question: How ?
Nested Loop Joins

- Tuple-based nested loop \( R \bowtie S \)

\[
\text{for each tuple } r \text{ in } R \text{ do } \\
\quad \text{for each tuple } s \text{ in } S \text{ do } \\
\quad \quad \text{if } r \text{ and } s \text{ join then output } (r, s)
\]

- Cost: \( T(R) \times B(S) \) when \( S \) is clustered
- Cost: \( T(R) \times T(S) \) when \( S \) is unclustered

What are Open( ), GetNext( ), Close( ) here?
Nested Loop Joins

• We can be much more clever

• Question: how would you compute the join in the following cases? What is the cost?
  – B(R) = 1000, B(S) = 2, M = 4
  – B(R) = 1000, B(S) = 3, M = 4
  – B(R) = 1000, B(S) = 6, M = 4
Block-Based Nested-loop Join

\[
\text{for each (M-2) blocks } bs \text{ of } S \text{ do} \\
\quad \text{for each block } br \text{ of } R \text{ do} \\
\quad \quad \text{for each tuple } s \text{ in } bs \\
\quad \quad \quad \text{for each tuple } r \text{ in } br \text{ do} \\
\quad \quad \quad \quad \text{if "r and s join" then output}(r,s)
\]
Block-Based Nested-loop Join

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

Input buffer for R  Output buffer  Join Result

R & S
Block-Based Nested-loop Join

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

• Notice: it is better to iterate over the smaller relation first

• $R \Join S$: \( R=\)outer relation, \( S=\)inner relation
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):
    - \( B(R) = 2000 \) I/Os
  - Index based selection:
    - If index is clustered: \( \frac{B(R)}{V(R,a)} = 100 \) I/Os
    - If index is unclustered: \( \frac{T(R)}{V(R,a)} = 5,000 \) I/Os

- Lesson: don’t build unclustered indexes when \( V(R,a) \) is small!

\[
\begin{align*}
B(R) &= 2000 \\
T(R) &= 100,000 \\
V(R, a) &= 20
\end{align*}
\]

Cost of \( \sigma_{a=v}(R) = ? \)
Index Based Join

• $R \bowtie S$

• Assume $S$ has an index on the join attribute

```
for each tuple $r$ in $R$ do
    lookup the tuple(s) $s$ in $S$ 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

- Partitioned hash algorithms

- Merge-sort algorithms
Partitioned Hash Algorithms

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

- Does each bucket fit in main memory?
  - Yes if B(R)/M <= M, i.e. B(R) <= M^2
Duplicate Elimination

• Recall: $\delta(R) = \text{duplicate elimination}$
• Step 1. Partition $R$ into buckets
• Step 2. Apply $\delta$ to each bucket (may read in main memory)

• Cost: $3B(R)$
• Assumption: $B(R) \leq M^2$
Grouping

• Recall: $\gamma(R) = \text{grouping and aggregation}$
• 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

\( R \bowtie S \)

- **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
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 $h2$ ($\langle\rangle h$). Scan matching partition of $S$, search for matches.
Partitioned Hash 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, for when $B < M^2$
External Merge-Sort: Step 1

- Phase one: load M bytes in memory, sort

![Diagram showing the process]
External Merge-Sort: Step 2

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

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

• Read+write+read = 3B(R)

• Assumption: B(R) <= M^2
Duplicate Elimination

Duplicate elimination $\delta(R)$

- 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: $B(\delta(R)) \leq M^2$
Grouping

Grouping: $\gamma_a, \text{sum}(b) \ (R)$

- Same as before: sort, then compute the sum(b) for each group of a’s
- Total cost: $3B(R)$
- Assumption: $B(R) \leq M^2$
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

\[ M_1 = \frac{B(R)}{M} \text{ runs for } R \]
\[ M_2 = \frac{B(S)}{M} \text{ runs for } S \]

If \( B \leq M^2 \) then we are done
Two-Pass Algorithms Based on Sorting

Join $R \Join S$

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

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