Lecture 7:
Query Execution and Optimization

Tuesday, February 20, 2007
Outline

• Chapters 4, 12-15
DBMS Architecture

How does a SQL engine work?

• SQL query → relational algebra plan
• Relational algebra plan → Optimized plan
• Execute each operator of the plan
Relational Algebra

• Formalism for creating new relations from existing ones
• Its place in the big picture:

Declarative query language  →  Algebra  →  Implementation

SQL, relational calculus  ❯  Relational algebra
Relational bag algebra
Relational Algebra

- Five operators:
  - Union: \( \cup \)
  - Difference: -
  - Selection: \( \sigma \)
  - Projection: \( \Pi \)
  - Cartesian Product: \( \times \)

- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming: \( \rho \)
1. Union and 2. Difference

- $R_1 \cup R_2$
- Example:
  - `ActiveEmployees \cup RetiredEmployees`

- $R_1 - R_2$
- Example:
  - `AllEmployees -- RetiredEmployees`
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)
- Example
  - UnionizedEmployees $\cap$ RetiredEmployees
3. Selection

• Returns all tuples which satisfy a condition
• Notation: $\sigma_c(R)$
• Examples
  – $\sigma_{\text{Salary} > 40000}(\text{Employee})$
  – $\sigma_{\text{name} = \text{“Smith”}}(\text{Employee})$
• The condition $c$ can be $=, <, \leq, >, \geq, <>$
\[
\sigma_{\text{Salary } > 40000} (\text{Employee})
\]

<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>
4. Projection

• Eliminates columns, then removes duplicates
• Notation: $\Pi_{A_1, \ldots, A_n}(R)$
• Example: project social-security number and names:
  – $\Pi_{\text{SSN}, \text{Name}}(\text{Employee})$
  – Output schema: Answer(SSN, Name)

Note that there are two parts:
1. Eliminate columns (easy)
2. Remove duplicates (hard)
In the “extended” algebra we will separate them.
\[ \Pi_{\text{Name}, \text{Salary}} (\text{Employee}) \]

<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>John</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>200000</td>
</tr>
</tbody>
</table>
5. Cartesian Product

- Each tuple in R1 with each tuple in R2
- Notation: R1 × R2
- Example:
  - Employee × Dependents
- Very rare in practice; mainly used to express joins
## Cartesian Product Example

### 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>

### Dependents

<table>
<thead>
<tr>
<th>EmployeeSSN</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Employee x Dependents

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmployeeSSN</th>
<th>Dname</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>
Relational Algebra

• Five operators:
  – Union: ∪
  – Difference: -
  – Selection: σ
  – Projection: Π
  – Cartesian Product: ×

• Derived or auxiliary operators:
  – Intersection, complement
  – Joins (natural, equi-join, theta join, semi-join)
  – Renaming: ρ
Renaming

• Changes the schema, not the instance
• Notation: $\rho_{B_1, \ldots, B_n} (R)$
• Example:
  – $\rho_{\text{LastName}, \text{SocSocNo}} (\text{Employee})$
  – Output schema:
    Answer(\text{LastName, SocSocNo})
## Renaming Example

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>SSN</td>
</tr>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

\[
\rho_{LastName, SocSocNo} (Employee)
\]

<table>
<thead>
<tr>
<th>LastName</th>
<th>SocSocNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>
Natural Join

• Notation: $R_1 \times R_2$

• Meaning: $R_1 \times R_2 = \Pi_A(\sigma_C(R_1 \times R_2))$

• Where:
  – The selection $\sigma_C$ checks equality of all common attributes
  – The projection eliminates the duplicate common attributes
Natural Join Example

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

<table>
<thead>
<tr>
<th><strong>Dependents</strong></th>
<th><strong>SSN</strong></th>
<th><strong>Dname</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>Emily</td>
<td></td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
<td></td>
</tr>
</tbody>
</table>

$\text{Employee} \bowtie \text{Dependents} =$

\[\Pi_{\text{Name}, \text{SSN}, \text{Dname}}(\sigma_{\text{SSN}=\text{SSN}_2}(\text{Employee} \times \rho_{\text{SSN}_2, \text{Dname}}(\text{Dependents})))\]

<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th><strong>SSN</strong></th>
<th><strong>Dname</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>
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 |×| 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 $R \times S$?

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

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

• A join that involves a predicate
• $R_1 \left| \times \right|_\theta R_2 = \sigma_\theta (R_1 \times R_2)$
• Here $\theta$ can be any condition
Eq-join

• A theta join where $\theta$ is an equality
• $R1 \mid \times \mid_{A=B} R2 = \sigma_{A=B} (R1 \times R2)$
• Example:
  – Employee $\mid \times \mid_{SSN=SSN}$ Dependents

• Most useful join in practice
Semijoin

- $R \times S = \Pi_{A_1, \ldots, A_n} (R \mid \times \mid S)$
- Where $A_1, \ldots, A_n$ are the attributes in $R$
- Example:
  - Employee $\mid \times$ Dependents
Semijoins in Distributed Databases

- Semijoins are used in distributed databases

Employee

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

Dependents

<table>
<thead>
<tr>
<th>SSN</th>
<th>Dname</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Employee \( \times \) \( \sigma_{ssn=ssn} \) \( \sigma_{age>71} \) (Dependents)

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

R = Employee \( \times \) T

Answer = R \( \times \) Dependents
Complex RA Expressions

\[
\Pi_{\text{name}} \left( \sigma_{\text{name}=\text{fred}} \left( \Pi_{\text{ssn}} \left( \sigma_{\text{name}=\text{gizmo}} \left( \Pi_{\text{pid}} \left( \sigma_{\text{buyer-ssn}=\text{ssn}} \left( \sigma_{\text{seller-ssn}=\text{ssn}} \left( \Pi_{\text{pid}} \left( \Pi_{\text{ssn}} \left( \text{Person} \right) \rightarrow \text{Purchase} \right) \rightarrow \text{Person} \rightarrow \text{Product} \right) \right) \right) \right) \right) \right) \right)
\]
Summary on the Relational Algebra

- A collection of 5 operators on relations
- Codd proved in 1970 that the relational algebra is equivalent to the relational calculus

Relational calculus/
First order logic/ SQL/
declarative language

= WHAT

Relational algebra/
procedural language

= HOW
Operations on Bags

A **bag** = a set with repeated elements

All operations need to be defined carefully on bags

- \{a,b,b,c\} ∪ \{a,b,b,b,e,f,f\} = \{a,a,b,b,b,b,b,c,e,f,f\}
- \{a,b,b,b,c,c\} – \{b,c,c,c,d\} = \{a,b,b,d\}
- \(\sigma_C(R)\): preserve the number of occurrences
- \(\Pi_A(R)\): no duplicate elimination
- \(\delta\) = explicit duplicate elimination
- Cartesian product, join: no duplicate elimination

Important ! Relational Engines work on bags, not sets !

**Reading assignment: 5.3 – 5.4**
Note: RA has Limitations!

- 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 C program
From SQL to RA

Purchase(buyer, product, city)
Person(name, age)

\[
\text{SELECT DISTINCT } P.\text{buyer} \\
\text{FROM } \text{Purchase } P, \text{ Person } Q \\
\text{WHERE } P.\text{buyer}=Q.\text{name AND} \\
P.\text{city}=\text{‘Seattle’ AND} \\
Q.\text{age} > 20
\]
Also...

Purchase(buyer, product, city)
Person(name, age)

```
SELECT DISTINCT P.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
    P.city='Seattle' AND
    Q.age > 20
```
Non-monontone Queries (in class)

Purchase(buyer, product, city)
Person(name, age)

```sql
SELECT DISTINCT P.product
FROM Purchase P
WHERE P.city='Seattle' AND
not exists (select *
from Purchase P2, Person Q
where P2.product = P.product
and P2.buyer = Q.name
and Q.age > 20)
```
Extended Logical Algebra Operators
(operate on Bags, not Sets)

• Union, intersection, difference
• Selection $\sigma$
• Projection $\Pi$
• Join $|x|$  
• Duplicate elimination $\delta$
• Grouping $\gamma$
• Sorting $\tau$
Logical Query Plan

```
SELECT city, count(*)
FROM sales
GROUP BY city
HAVING sum(price) > 100
```

$T_1,$ $T_2,$ $T_3 = \text{temporary tables}$
Logical v.s. Physical Algebra

• We have seen the logical algebra so far:
  – Five basic operators, plus group-by, plus sort

• The Physical algebra refines each operator into a concrete algorithm
Physical Plan

SELECT DISTINCT P.buyer FROM Purchase P, Person Q WHERE P.buyer=Q.name AND P.city='Seattle' AND Q.age > 20

SELECT DISTINCT P.buyer FROM Purchase P, Person Q WHERE P.buyer=Q.name AND P.city='Seattle' AND Q.age > 20

Parse tree: [Diagram]

- Index-join
- Sequential scan
- Projection
- Selection
- Hash-based dup. elim
Physical Plans Can Be Subtle

```sql
SELECT *
FROM Purchase P
WHERE P.city='Seattle'
```

Where did the join come from?
Architecture of a Database Engine

SQL query

Parse Query

Select Logical Plan

Select Physical Plan

Query Execution

Query optimization

Logical plan

Physical plan
Logical operator:

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

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

1. 
2. 
3.
Question in Class

Product(pname, cname) \( \times \) 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 time =
- Sort and merge merge-join time =
- Hash join time =
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

NOTE: Book uses $M$ for the number of blocks in $R$, and $B$ for the number of blocks in main memory
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, $V(R,a)$
Selection and Projection

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

- Both are *tuple-at-a-time* algorithms
- Cost: $B(R)$
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: \( h(x) = x \mod 4 \)

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

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

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

• Place in right bucket, if space
• E.g. $h(d)=2$
Insertion in Hash Table

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

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

- Cost: \( B(R) + B(S) \)
- Assumption: \( B(S) \leq M \)
Main Memory
Duplicate Elimination

Duplicate elimination $\delta(R)$

- Hash table in main memory

- Cost: $B(R)$
- Assumption: $B(\delta(R)) \leq M$
Main Memory 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 $R \bowtie S$

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
Nested Loop Joins

• We can be much more clever

• *Question:* how would you compute the join in the following cases? What is the cost?
  
  – \( B(R) = 1000, B(S) = 2, M = 4 \)
  
  – \( B(R) = 1000, B(S) = 3, M = 4 \)
  
  – \( B(R) = 1000, B(S) = 6, M = 4 \)
Nested Loop Joins

- Block-based Nested Loop Join

```plaintext
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)
```
Nested Loop Joins

R & S

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

Input buffer for R
Output buffer

Join Result
Nested Loop Joins

- **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 |x| 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 on $a$: cost $T(R)/V(R,a)$
  - We have seen that this is like a join
Index Based Selection

- **Example:**
  - Table scan (assuming R is clustered):
    - \( B(R) = 2,000 \) 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!

\[
B(R) = 2000 \\
T(R) = 100,000 \\
V(R, a) = 20
\]

\[\text{cost of } \sigma_{a=v}(R) = ?\]
Index Based Join

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

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

Cost (Assuming R is clustered):

- If index is clustered: $B(R) + T(R)B(S)/V(S,a)$
- If index is 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) ≤ M²
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 \mid x \mid 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.

- **Probe**: Read in a partition of R, hash it using $h_2 (\neq h)$. Scan matching partition of S, search for matches.
Partitioned Hash Join

- Cost: $3B(R) + 3B(S)$
- Assumption: $\min(B(R), B(S)) \leq M^2$
Hybrid Hash Join Algorithm

• Partition S into k buckets
  t buckets $S_1, \ldots, S_t$ stay in memory
  k-t buckets $S_{t+1}, \ldots, S_k$ to disk

• Partition R into k buckets
  – First t buckets join immediately with S
  – Rest k-t buckets go to disk

• Finally, join k-t pairs of buckets:
  $(R_{t+1}, S_{t+1}), (R_{t+2}, S_{t+2}), \ldots, (R_k, S_k)$
Hybrid Join Algorithm

• How to choose k and t ?
  – Choose k large but s.t. \( k \leq M \)
  – Choose \( t/k \) large but s.t. \( t/k \cdot B(S) \leq M \)
  – Moreover: \( t/k \cdot B(S) + k-t \leq M \)

• Assuming \( t/k \cdot B(S) \gg k-t \): \( t/k = M/B(S) \)
Hybrid Join Algorithm

• How many I/Os?
• Cost of partitioned hash join: $3B(R) + 3B(S)$
• Hybrid join saves $2$ I/Os for a $t/k$ fraction of buckets
• Hybrid join saves $2t/k(B(R) + B(S))$ I/Os
• Cost: $(3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))$
Hybrid Join Algorithm

- Question in class: what is the real advantage of the hybrid algorithm?
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
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

• Read+write+read = 3B(R)

• Assumption: B(R) <= M^2
Extensions, Discussions

- **Blocked I/O**
  - Group b blocks and process them together
  - Same effect as increasing the block size by a factor b

- **Double buffering**:
  - Keep two buffers for each input or output stream
  - During regular merge on one set of buffers, perform the I/O on the other set of buffers
  - Decreases M to M/2
Extensions, Discussions

- Initial run formation (level 0-runs)
  - Main memory sort (usually Quicksort): results in initial runs of length $M$
  - Replacement selection: start by reading a chunk of file of size $M$, organize as heap, start to output the smallest elements in increasing order; as the buffer empties, read more data; *the new elements are added to the heap as long as they are > the last element output*. Expected run lengths turns out to be approx $2M$
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: $\gamma_{a, \sum(b)} (R)$

- 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 \( |x| \) 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 \( |x| S \)

- 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: $3B(R)+3B(S)$
  - $B(R)+B(S) \leq M^2$
Example

Product\( (\text{pname}, \text{maker}) \), Company\( (\text{cname}, \text{city}) \)

Select Product.pname
From Product, Company
Where Product.maker = Company.cname
and Company.city = "Seattle"

• How do we execute this query?
Example

Product(pname, maker), Company(cname, city)

Assume:

Clustered index:   Product.pname, Company.cname
Unclustered index: Product.maker, Company.city
Logical Plan:

```
 Company
 (cname,city)  \σ_{city="Seattle"}  Product
 (pname,maker)          \bowtie_{maker=cname}
```


Physical plan 1:

\[ \sigma_{\text{city} = \text{"Seattle"}} \]

\(\text{Company}\) (\(\text{cname,city}\)) \hspace{1cm} \text{Product} (\(\text{pname,maker}\))

Index-based selection

Index-based join

\(\text{cname=}\text{maker}\)
Physical plans 2a and 2b:

Which one is better??

\( \sigma _{\text{city} = \text{"Seattle"}} \)

Product
\((\text{pname}, \text{maker})\)

Company
\((\text{cname}, \text{city})\)

Merge-join

\(\text{maker} = \text{cname}\)

Scan and sort (2a)

Index scan (2b)
Physical plan 1:

\[ \sigma_{\text{city} = \text{"Seattle"}}(\text{Company}) \times \sigma_{\text{cname} = \text{maker}}(\text{Product}) \]

Total cost:
\[ T(\text{Company}) / V(\text{Company}, \text{city}) \times T(\text{Product}) / V(\text{Product}, \text{maker}) \]
Total cost:
(2a): $3B(\text{Product}) + B(\text{Company})$
(2b): $T(\text{Product}) + B(\text{Company})$

Physical plans 2a and 2b:

- **Merge-join**
- $\sigma_{\text{city}=\text{"Seattle"}}$
- $\bowtie$ maker=cname

- **Scan and sort (2a)**
- **index scan (2b)**

- **Table-scan**

- **No extra cost (why?)**
- $3B(\text{Product})$
- $T(\text{Product})$
- $B(\text{Company})$
Plan 1: \( T(\text{Company})/V(\text{Company,city}) \times T(\text{Product})/V(\text{Product,maker}) \)

Plan 2a: \( B(\text{Company}) + 3B(\text{Product}) \)

Plan 2b: \( B(\text{Company}) + T(\text{Product}) \)

Which one is better ??

It depends on the data !!
Example

<table>
<thead>
<tr>
<th>T(Company)</th>
<th>B(Company)</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>T(Product)</td>
<td>B(Product)</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>1,000</td>
<td></td>
</tr>
</tbody>
</table>

We may assume $V(Product, maker) \approx T(Company)$ (why ?)

- Case 1: $V(Company, city) \approx T(Company)$

  $V(Company, city) = 2,000$

- Case 2: $V(Company, city) << T(Company)$

  $V(Company, city) = 20$
Which Plan is Best?

Plan 1: $T(\text{Company})/V(\text{Company}, \text{city}) \times T(\text{Product})/V(\text{Product}, \text{maker})$
Plan 2a: $B(\text{Company}) + 3B(\text{Product})$
Plan 2b: $B(\text{Company}) + T(\text{Product})$

Case 1:

Case 2:
Lessons

• Need to consider several physical plan
  – even for one, simple logical plan
• No magic “best” plan: depends on the data
• In order to make the right choice
  – need to have statistics over the data
  – the B’s, the T’s, the V’s
Query Optimization

• Have a SQL query Q

• Create a plan P

• Find equivalent plans $P = P' = P'' = \ldots$

• Choose the “cheapest”.
Logical Query Plan

\[
\text{SELECT } \text{P.buyer} \\
\text{FROM } \text{Purchase P, Person Q} \\
\text{WHERE } \text{P.buyer}=\text{Q.name AND} \\
\quad \text{P.city}=\text{'seattle' AND} \\
\quad \text{Q.phone}>\text{‘5430000’}
\]

\[\sigma \text{City='seattle' \land phone>’5430000’} \quad \text{σ}
\]

\[\text{P} = \prod_{\text{buyer}} \quad \text{Π}_{\text{buyer}}
\]

\[
\begin{align*}
\text{In class:} \\
\text{find a “better” plan } P'
\end{align*}
\]
Logical Query Plan

Q =

SELECT city, sum(quantity)
FROM sales
GROUP BY city
HAVING sum(quantity) < 100

P =

T2(city,p)
\[\sigma_p < 100\]
T1(city,p)
\[\gamma_{city, sum(quantity)} \rightarrow_p\]
sales(product, city, quantity)

In class:
find a “better” plan P’
Optimization

• Main idea: rewrite a logical query plan into an equivalent “more efficient” logical plan
The three components of an optimizer

We need three things in an optimizer:

- Algebraic laws
- An optimization algorithm
- A cost estimator
Algebraic Laws

• Commutative and Associative Laws
  \[ R \cup S = S \cup R, \quad R \cup (S \cup T) = (R \cup S) \cup T \]
  \[ R \times S = S \times R, \quad R \times (S \times T) = (R \times S) \times T \]
  \[ R \times S = S \times R, \quad R \times (S \times T) = (R \times S) \times T \]

• Distributive Laws
  \[ R \times (S \cup T) = (R \times S) \cup (R \times T) \]
Algebraic Laws

• Laws involving selection:
  \[ \sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R) \]
  \[ \sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R) \]
  \[ \sigma_C(R \mid\times\mid S) = \sigma_C(R) \mid\times\mid S \]

• When \( C \) involves only attributes of \( R \)
  \[ \sigma_C(R - S) = \sigma_C(R) - S \]
  \[ \sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S) \]
  \[ \sigma_C(R \mid\times\mid S) = \sigma_C(R) \mid\times\mid S \]
Algebraic Laws

• Example: $R(A, B, C, D), S(E, F, G)$

\[ \sigma_{F=3} (R \mid \times \mid_{D=E} S) = \] ?

\[ \sigma_{A=5 \text{ AND } G=9} (R \mid \times \mid_{D=E} S) = \] ?
Algebraic Laws

• Laws involving projections
  \( \Pi_M(R \times | S) = \Pi_M(\Pi_P(R) \times | \Pi_Q(S)) \)
  \( \Pi_M(\Pi_N(R)) = \Pi_{M,N}(R) \)

• Example \( R(A,B,C,D), S(E, F, G) \)
  \( \Pi_{A,B,G}(R \times |_{D=E} S) = \Pi ?(\Pi ?(R) \times |_{D=E} \Pi ?(S)) \)
Algebraic Laws

• Laws involving grouping and aggregation:
  \[ \delta(\gamma_{A, \text{agg}(B)}(R)) = \gamma_{A, \text{agg}(B)}(R) \]
  \[ \gamma_{A, \text{agg}(B)}(\delta(R)) = \gamma_{A, \text{agg}(B)}(R) \text{ if agg is “duplicate insensitive”} \]

• Which of the following are “duplicate insensitive”? sum, count, avg, min, max

\[ \gamma_{A, \text{agg}(D)}(R(A,B) \mid \times |_{B=C} S(C,D)) = \gamma_{A, \text{agg}(D)}(R(A,B) \mid \times |_{B=C} (\gamma_{C, \text{agg}(D)}S(C,D))) \]
Optimizations Based on Semijoins

THIS IS ADVANCED STUFF; NOT ON THE FINAL

• $R \bowtie S = \Pi_{A_1,\ldots,A_n} (R \bowtie S)$

• Where the schemas are:
  – Input: $R(A_1,\ldots,A_n)$, $S(B_1,\ldots,B_m)$
  – Output: $T(A_1,\ldots,A_n)$
Optimizations Based on Semijoins

Semijoins: a bit of theory (see [AHV])

- Given a query:
  \[ Q \leftarrow \Pi \left( \sigma (R_1 \mid x \mid R_2 \mid x \mid \ldots \mid x \mid R_n ) \right) \]

- A full reducer for Q is a program:
  \[
  \begin{align*}
  R_{i1} &:= R_{i1} \bowtie R_{j1} \\
  R_{i2} &:= R_{i2} \bowtie R_{j2} \\
  &\ldots \\
  R_{ip} &:= R_{ip} \bowtie R_{jp}
  \end{align*}
  \]

- Such that no dangling tuples remain in any relation
Optimizations Based on Semijoins

- Example:

\[
\text{Q}(A,E) \gets R1(A,B) \, \land \, R2(B,C) \, \land \, R3(C,D,E)
\]

- A full reducer is:

\[
\begin{align*}
R2(B,C) & := R2(B,C) \, \land \, R1(A,B) \\
R3(C,D,E) & := R3(C,D,E) \, \land \, R2(B,C) \\
R2(B,C) & := R2(B,C) \, \land \, R3(C,D,E) \\
R1(A,B) & := R1(A,B) \, \land \, R2(B,C)
\end{align*}
\]

The new tables have only the tuples necessary to compute \(Q(E)_{07}\).
Optimizations Based on Semijoins

- Example:

\[
Q(E) \ :- \ R1(A, B) \mid x \mid R2(B, C) \mid x \mid R3(A, C, E)
\]

- Doesn’t have a full reducer (we can reduce forever)

**Theorem** a query has a full reducer iff it is “acyclic”
Optimizations Based on Semijoins

• Semijoins in [Chaudhuri’98]

```sql
CREATE VIEW DepAvgSal As (  
    SELECT E.did, Avg(E.Sal) AS avgsal  
    FROM Emp E  
    GROUP BY E.did)

SELECT E.eid, E.sal  
FROM Emp E, Dept D, DepAvgSal V  
WHERE E.did = D.did AND E.did = V.did  
    AND E.age < 30 AND D.budget > 100k  
    AND E.sal > V.avgsal
```
Optimizations Based on Semijoins

• First idea:

```sql
CREATE VIEW LimitedAvgSal As (
    SELECT E.did, Avg(E.Sal) AS avgsal
    FROM Emp E, Dept D
    WHERE E.did = D.did AND D.budget > 100k
    GROUP BY E.did)

SELECT E.eid, E.sal
FROM Emp E, Dept D, LimitedAvgSal V
WHERE E.did = D.did AND E.did = V.did
    AND E.age < 30 AND D.budget > 100k
    AND E.sal > V.avgsal
```
Optimizations Based on Semijoins

• Better: full reducer

CREATE VIEW PartialResult AS
  (SELECT E.id, E.sal, E.did
   FROM Emp E, Dept D
   WHERE E.did=D.did AND E.age < 30
   AND D.budget > 100k)

CREATE VIEW Filter AS
  (SELECT DISTINCT P.did FROM PartialResult P)

CREATE VIEW LimitedAvgSal AS
  (SELECT E.did, Avg(E.Sal) AS avgsal
   FROM Emp E, Filter F
   WHERE E.did = F.did GROUP BY E.did)
Optimizations Based on Semijoins

```
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```
Cost-based Optimizations

• Main idea: apply algebraic laws, until estimated cost is minimal

• Practically: start from partial plans, introduce operators one by one
  – Will see in a few slides

• Problem: there are too many ways to apply the laws, hence too many (partial) plans
Cost-based Optimizations

Approaches:

- **Top-down**: the partial plan is a top fragment of the logical plan

- **Bottom up**: the partial plan is a bottom fragment of the logical plan
Dynamic Programming

Originally proposed in System R

- Only handles single block queries:
  
  \[
  \text{SELECT list FROM list WHERE cond}_1 \text{ AND cond}_2 \text{ AND } \ldots \text{ AND cond}_k
  \]

- Heuristics: selections down, projections up
- Dynamic programming: \textit{join reordering}
Join Trees

- $R_1 \times R_2 \times \ldots \times R_n$
- Join tree:

A plan = a join tree
A partial plan = a subtree of a join tree
Types of Join Trees

• Left deep:
Types of Join Trees

• Bushy:

```
            ┌───┐
           │   │
          └───┘
            ┌───┐
           │   │
          └───┘
            ┌───┐
           │   │
          └───┘
            ┌───┐
           │   │
          └───┘
```
Types of Join Trees

• Right deep:

```
        ▲
       /\  
      ▼  ▼  
     /    /  
    ▲    ▲  
   /      /  
  ▼      ▼  
  R3  R1  R5
       /    /  
      ▼    ▼  
     /      /  
    ▲      ▲  
   /        /  
  ▼        ▼  
  R2  R4  
```
Dynamic Programming

• Given: a query $R_1 \times R_2 \times \ldots \times R_n$
• Assume we have a function cost() that gives us the cost of every join tree
• Find the best join tree for the query
Dynamic Programming

• Idea: for each subset of \{R1, …, Rn\}, compute the best plan for that subset
• In increasing order of set cardinality:
  – Step 1: for \{R1\}, \{R2\}, …, \{Rn\}
  – Step 2: for \{R1,R2\}, \{R1,R3\}, …, \{Rn-1, Rn\}
  – …
  – Step n: for \{R1, …, Rn\}
• It is a bottom-up strategy
• A subset of \{R1, …, Rn\} is also called a subquery
Dynamic Programming

• For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ compute the following:
  – Size($Q$)
  – A best plan for $Q$: Plan($Q$)
  – The cost of that plan: Cost($Q$)
Dynamic Programming

• **Step 1:** For each \( \{R_i\} \) do:
  
  – Size(\( \{R_i\} \)) = B(\( R_i \))
  
  – Plan(\( \{R_i\} \)) = R_i
  
  – Cost(\( \{R_i\} \)) = (\text{cost of scanning } R_i)
Dynamic Programming

• **Step i:** For each $Q \subseteq \{R_1, \ldots, R_n\}$ of cardinality $i$ do:
  
  – Compute $\text{Size}(Q)$ (later…)
  
  – For every pair of subqueries $Q'$, $Q''$ such that $Q = Q' \cup Q''$
    compute $\text{cost}(\text{Plan}(Q') \times \text{Plan}(Q''))$
  
  – $\text{Cost}(Q) = \text{the smallest such cost}$
  
  – $\text{Plan}(Q) = \text{the corresponding plan}$
Dynamic Programming

• Return Plan(\{R_1, \ldots, R_n\})
Dynamic Programming

To illustrate, we will make the following simplifications:

• \( \text{Cost}(P_1 \times P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) + \text{size(Intermediate result(s))} \)

• Intermediate results:
  – If \( P_1 \) = a join, then the size of the intermediate result is \( \text{size}(P_1) \), otherwise the size is 0
  – Similarly for \( P_2 \)

• Cost of a scan = 0
Dynamic Programming

- Example:
- \( \text{Cost}(R5 \mid \times \mid R7) = 0 \) (no intermediate results)
- \( \text{Cost}((R2 \mid \times \mid R1) \mid \times \mid R7) = \text{Cost}(R2 \mid \times \mid R1) + \text{Cost}(R7) + \text{size}(R2 \mid \times \mid R1) = \text{size}(R2 \mid \times \mid R1) \)
Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: $T(A \times B) = 0.01 \times T(A) \times T(B)$
<table>
<thead>
<tr>
<th>Subquery</th>
<th>Size</th>
<th>Cost</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSTU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subquery</td>
<td>Size</td>
<td>Cost</td>
<td>Plan</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>RS</td>
<td>100k</td>
<td>0</td>
<td>RS</td>
</tr>
<tr>
<td>RT</td>
<td>60k</td>
<td>0</td>
<td>RT</td>
</tr>
<tr>
<td>RU</td>
<td>20k</td>
<td>0</td>
<td>RU</td>
</tr>
<tr>
<td>ST</td>
<td>150k</td>
<td>0</td>
<td>ST</td>
</tr>
<tr>
<td>SU</td>
<td>50k</td>
<td>0</td>
<td>SU</td>
</tr>
<tr>
<td>TU</td>
<td>30k</td>
<td>0</td>
<td>TU</td>
</tr>
<tr>
<td>RST</td>
<td>3M</td>
<td>60k</td>
<td>(RT)S</td>
</tr>
<tr>
<td>RSU</td>
<td>1M</td>
<td>20k</td>
<td>(RU)S</td>
</tr>
<tr>
<td>RTU</td>
<td>0.6M</td>
<td>20k</td>
<td>(RU)T</td>
</tr>
<tr>
<td>STU</td>
<td>1.5M</td>
<td>30k</td>
<td>(TU)S</td>
</tr>
<tr>
<td>RSTU</td>
<td>30M</td>
<td>60k+50k=110k</td>
<td>(RT)(SU)</td>
</tr>
</tbody>
</table>
Reducing the Search Space

- Left-linear trees v.s. Bushy trees
- Trees without cartesian product

Example: \( R(A,B) \times S(B,C) \times T(C,D) \)

Plan: \( (R(A,B) \times T(C,D)) \times S(B,C) \) has a cartesian product – most query optimizers will not consider it.
Dynamic Programming: Summary

• Handles only join queries:
  – Selections are pushed down (i.e. early)
  – Projections are pulled up (i.e. late)

• Takes exponential time in general, BUT:
  – Left linear joins may reduce time
  – Non-cartesian products may reduce time further
Rule-Based Optimizers

- **Extensible** collection of rules
  
  Rule = Algebraic law with a direction

- Algorithm for firing these rules
  
  Generate many alternative plans, in some order
  
  Prune by cost

- Volcano (later SQL Sever)
- Starburst (later DB2)
Completing the Physical Query Plan

• Choose algorithm to implement each operator
  – Need to account for more than cost:
    • How much memory do we have ?
    • Are the input operand(s) sorted ?

• Decide for each intermediate result:
  – To materialize
  – To pipeline
Materialize Intermediate Results Between Operators

HashTable \leftarrow S
repeat  
  read(R, x)
  y \leftarrow join(HashTable, x)
  write(V1, y)

HashTable \leftarrow T
repeat  
  read(V1, y)
  z \leftarrow join(HashTable, y)
  write(V2, z)

HashTable \leftarrow U
repeat  
  read(V2, z)
  u \leftarrow join(HashTable, z)
  write(Answer, u)
Materialize Intermediate Results Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan?
  - Cost =
- How much main memory do we need?
  - M =
Pipeline Between Operators

HashTable1 $\leftarrow$ S
HashTable2 $\leftarrow$ T
HashTable3 $\leftarrow$ U
repeat
  read(R, x)
  y $\leftarrow$ join(HashTable1, x)
  z $\leftarrow$ join(HashTable2, y)
  u $\leftarrow$ join(HashTable3, z)
write(Answer, u)

pipeline
Pipeline Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan?
  - Cost =

- How much main memory do we need?
  - M =
Pipeline in Bushy Trees
Example

• Logical plan is:

```
  k blocks
  /      \
R(w,x)    S(x,y)
  \
  5,000 blocks  10,000 blocks
```

• Main memory $M = 101$ buffers
Example

\[ M = 101 \]

Naïve evaluation:
- 2 partitioned hash-joins
- Cost \(3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k\)
Example

\[ M = 101 \]

Smarter:
- Step 1: hash \( R \) on \( x \) into 100 buckets, each of 50 blocks; to disk
- Step 2: hash \( S \) on \( x \) into 100 buckets; to disk
- Step 3: read each \( R_i \) in memory (50 buffer) join with \( S_i \) (1 buffer); hash result on \( y \) into 50 buckets (50 buffers) \( \text{-- here we } \textit{pipeline} \)
- Cost so far: \( 3B(R) + 3B(S) \)
Example

$M = 101$

Continuing:
- How large are the 50 buckets on $y$? Answer: $k/50$.
- If $k \leq 50$ then keep all 50 buckets in Step 3 in memory, then:
  - Step 4: read $U$ from disk, hash on $y$ and join with memory
  - Total cost: $3B(R) + 3B(S) + B(U) = 55,000$
Example

\[ M = 101 \]

Continuing:
- If \( 50 < k \leq 5000 \) then send the 50 buckets in Step 3 to disk
  - Each bucket has size \( k/50 \leq 100 \)
- Step 4: partition \( U \) into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: \( 3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k \)
Example

M = 101

Continuing: 5,000 blocks 10,000 blocks
• If k > 5000 then materialize instead of pipeline
• 2 partitioned hash-joins
• Cost $3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k$
Example

Summary:

• If $k \leq 50$, $\text{cost} = 55,000$
• If $50 < k \leq 5000$, $\text{cost} = 75,000 + 2k$
• If $k > 5000$, $\text{cost} = 75,000 + 4k$
Size Estimation

The problem: Given an expression $E$, compute $T(E)$ and $V(E, A)$

- This is hard without computing $E$
- Will ‘estimate’ them instead
Size Estimation

Estimating the size of a projection

- Easy: $T(\Pi_L(R)) = T(R)$
- This is because a projection doesn’t eliminate duplicates
Size Estimation

Estimating the size of a selection

- $S = \sigma_{A=c}(R)$
  - $T(S)$ can be anything from 0 to $T(R) - V(R,A) + 1$
  - Estimate: $T(S) = T(R)/V(R,A)$
  - When $V(R,A)$ is not available, estimate $T(S) = T(R)/10$

- $S = \sigma_{A<c}(R)$
  - $T(S)$ can be anything from 0 to $T(R)$
  - Estimate: $T(S) = (c - \text{Low}(R,A))/(\text{High}(R,A) - \text{Low}(R,A))T(R)$
  - When Low, High unavailable, estimate $T(S) = T(R)/3$
Size Estimation

Estimating the size of a natural join, \( R \mid \times \mid_A S \)

- When the set of \( A \) values are disjoint, then 
  \[ T(R \mid \times \mid_A S) = 0 \]

- When \( A \) is a key in \( S \) and a foreign key in \( R \), then 
  \[ T(R \mid \times \mid_A S) = T(R) \]

- When \( A \) has a unique value, the same in \( R \) and \( S \), then 
  \[ T(R \mid \times \mid_A S) = T(R) \cdot T(S) \]
Size Estimation

Assumptions:

- **Containment of values**: if $V(R,A) \leq V(S,A)$, then the set of $A$ values of $R$ is included in the set of $A$ values of $S$
  
  - Note: this indeed holds when $A$ is a foreign key in $R$, and a key in $S$

- **Preservation of values**: for any other attribute $B$, $V(R \mid \times \mid_A S, B) = V(R, B)$ (or $V(S, B)$)
Size Estimation

Assume \( V(R,A) \leq V(S,A) \)

- Then each tuple \( t \) in \( R \) joins \emph{some} tuple(s) in \( S \)
  - How many?
  - On average \( T(S)/V(S,A) \)
  - \( t \) will contribute \( T(S)/V(S,A) \) tuples in \( R \times_A S \)

- Hence \( T(R \times_A S) = T(R) \cdot T(S) / V(S,A) \)

In general: \( T(R \times_A S) = T(R) \cdot T(S) / \max(V(R,A), V(S,A)) \)
Size Estimation

Example:

- \( T(R) = 10000, \; T(S) = 20000 \)
- \( V(R, A) = 100, \; V(S, A) = 200 \)
- How large is \( R \times_A S \)?

Answer: \( T(R \times_A S) = 10000 \times \frac{20000}{200} = 1M \)
Size Estimation

Joins on more than one attribute:

\[ T(R \mid \times \mid_{A, B} S) = \frac{T(R) \cdot T(S)}{(\max(V(R, A), V(S, A)) \cdot \max(V(R, B), V(S, B)))} \]
Histograms

• Statistics on data maintained by the RDBMS
• Makes size estimation much more accurate (hence, cost estimations are more accurate)
Histograms

Employee(ssn, name, salary, phone)

- Maintain a histogram on salary:

<table>
<thead>
<tr>
<th>Salary:</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

- \( T(\text{Employee}) = 25000 \), but now we know the distribution
Histories

**Ranks(rankName, salary)**

- Estimate the size of Employee $| \times |$ Salary Ranks

<table>
<thead>
<tr>
<th>Employee</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ranks</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>
Histograms

- **Eqwidth**
  
<table>
<thead>
<tr>
<th>0..20</th>
<th>20..40</th>
<th>40..60</th>
<th>60..80</th>
<th>80..100</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>104</td>
<td>9739</td>
<td>152</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Eqdepth**
  
<table>
<thead>
<tr>
<th>0..44</th>
<th>44..48</th>
<th>48..50</th>
<th>50..56</th>
<th>55..100</th>
</tr>
</thead>
</table>