Query Processing

• The query processor turns user queries and data modification commands into a query plan - a sequence of operations (or algorithm) on the database
  – from high level queries to low level commands
• Decisions taken by the query processor
  – Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
  – For each algebraic operator what algorithm should we use to run the operator?
  – How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

Example

Select B,D
From R,S
Where R.A = "c" ∧ S.E = 2 ∧ R.C=S.C
How do we execute query eventually?

- Scan relations
- Do Cartesian product
- Select tuples
- Do projection

One idea

RxS

<table>
<thead>
<tr>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Bingo!
Got one...
**Relational Algebra** - can be enhanced to describe plans...

**Ex:** Plan I

\[
\begin{align*}
\pi_{B,D}^{FLY} & \\
\sigma_{R.A = 'c' \land S.E = 2 \land R.C = S.C}^{FLY} & \\
R^{SCAN} & \times S^{SCAN}
\end{align*}
\]

1. Scan R
2. For each tuple r of R scan S
3. For each (r,s), where s in S select and project on the fly

**OR:**

\[
\pi_{B,D}^{FLY} \left( \sigma_{R.A = 'c' \land S.E = 2 \land R.C = S.C}^{FLY} (R^{SCAN} \times S^{SCAN}) \right)
\]

"FLY" and "SCAN" are the defaults

**Ex:** Plan I

\[
\begin{align*}
\pi_{B,D} & \\
\sigma_{R.A = 'c' \land S.E = 2 \land R.C = S.C} & \\
R & \times S
\end{align*}
\]

Another idea:

**Plan II**

\[
\begin{align*}
\pi_{B,D} & \\
\sigma_{R.A = 'c' \land S.E = 2}^{HASH} & \\
\sigma_{S.E = 2} & \\
R & \bowtie S
\end{align*}
\]

Scan R and S, perform on the fly selections, do hash join, project
Plan III
Use R.A and S.C Indexes

1. Use R.A index to select R tuples with R.A = "c"
2. For each R.C value found, use S.C index to find matching join tuples
3. Eliminate join tuples S.E ≠ 2
4. Project B.D attributes

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>σ(R)</th>
<th>σ(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10 x 2</td>
</tr>
<tr>
<td>c</td>
<td>20 y 2</td>
</tr>
<tr>
<td>d</td>
<td>20 y 2</td>
</tr>
<tr>
<td>e</td>
<td>20 y 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

next tuple: <c,7,15>
Algebraic Form of Plan

From Query To Optimal Plan

• Complex process
• Algebra-based logical and physical plans
• Transformations
• Evaluation of multiple alternatives

Issues in Query Processing and Optimization

• Generate Plans
  – employ efficient execution primitives for computing relational algebra operations
  – systematically transform expressions to achieve more efficient combinations of operators
• Estimate Cost of Generated Plans
  – Statistics
• “Smart” Search of the Space of Possible Plans
  – always do the “good” transformations (relational algebra optimization)
  – prune the space (e.g., System R)
• Often the above steps are mixed
Example: The Journey of a Query

The Journey of a Query cont’d:
Summary of Logical Plan Generator

• 4 logical query plans created
• algebraic rewritings were used for producing the candidate logical query plans
• the last one is the winner (at least, cannot be a big loser)
• in general, multiple logical plans may "win" eventually
The Journey of a Query Continues at the Physical Plan Generator

Physical Plan Generators chooses execution primitives and data passing

Example: Nested SQL query

```
SELECT title
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

Example: Parse Tree

```
<Query>
  SELECT <SelList> FROM <FromList> WHERE <Condition>
    <Attribute> <RelName> <Tuple> IN <Query>
    title StarsIn <Attribute> ( <Query> )

SELECT <SelList> FROM <FromList> WHERE <Condition>
    <Attribute> <RelName> <Attribute> LIKE <Pattern>
    name MovieStar birthDate %1960'
```
Example: Generating Relational Algebra

$\Pi_{\text{title}}$ 

$\sigma_{\text{birthdate LIKE } '%1960'}$ 

$\sigma_{\text{starName = MovieStar}}$

An expression using a two-argument $\sigma$, midway between a parse tree and relational algebra

Example: Logical Query Plan (Relational Algebra)

$\Pi_{\text{title}}$ 

$\sigma_{\text{starName = name}}$

$\times$

$\sigma_{\text{birthdate LIKE } '%1960'}$

$\sigma_{\text{starName = MovieStar}}$

May consider "IN" elimination as a rewriting in the logical plan generator or may consider it a task of the converter

Example: Improved Logical Query Plan

$\Pi_{\text{title}}$

$\sigma_{\text{starName = name}}$

$\times$

$\sigma_{\text{birthdate LIKE } '%1960'}$

$\sigma_{\text{starName = MovieStar}}$

Question: Push project to StarsIn?
Example: Result sizes are important for selecting physical plans

Example: One Physical Plan

Topics

- Bag Algebra, List Algebra and other extensions
  - name & value conversions, functions, aggregation
Algebraic Operators: A Bag version

- **Union of R and S**: a tuple \( t \) is in the result as many times as the sum of the number of times it is in \( R \) plus the times it is in \( S \)
- **Intersection of R and S**: a tuple \( t \) is in the result the minimum of the number of times it is in \( R \) and \( S \)
- **Difference of R and S**: a tuple \( t \) is in the result the number of times it is in \( R \) minus the number of times it is in \( S \)
- \( \delta(R) \) converts the bag \( R \) into a set
  - SQL’s \( R \cup S \) is really \( \delta(R \cup S) \)
- **Example**: Let \( R=\{A,B,B\} \) and \( S=\{C,A,B,C\} \). Describe the union, intersection and difference...

Extended Projection

- We extend the relational project \( \pi_A \) as follows:
  - The attribute list may include \( x \rightarrow y \) in the list \( A \) to indicate that the attribute \( x \) is renamed to \( y \)
  - Arithmetic, string operators and scalar functions on attributes are allowed. For example,
    - \( a+b \rightarrow x \) means that the sum of \( a \) and \( b \) is renamed into \( x \)
    - \( c||d \rightarrow y \) concatenates the result of \( c \) and \( d \) into a new attribute named \( y \)
  - The result is computed by considering each tuple in turn and constructing a new tuple by picking the attributes names in \( A \) and applying renamings and arithmetic and string operators
- **Example**:...

An Alternative Approach to Arithmetic and Other 1-1 Computations

- Special purpose operators that for every input tuple they produce one output tuple
  - \( \text{MULT}_{A,B \rightarrow C} R \): for each tuple of \( R \), multiply attribute \( A \) with attribute \( B \) and put the result in a new attribute named \( C \).
  - \( \text{PLUS}_{A,B \rightarrow C} R \)
  - \( \text{CONCAT}_{A,B \rightarrow C} R \)
- **Exercise**: Write the above operators using extended projection. Assume the schema of \( R \) is \( R(A,B,D,E) \).
Products and Joins

- **Product of R and S (R×S):**
  - If an attribute named \(a\) is found in both schemas then rename one column into \(R.a\) and the other into \(S.a\)
  - If a tuple \(rs\) is found \(n\) times in \(R\) and a tuple \(s\) is found \(m\) times in \(S\) then the product contains \(nm\) instances of the tuple \(rs\)

- **Joins**
  - **Natural Join**
    \(R \bowtie S = \pi_c(\sigma_A(R \times S))\)
    - \(c\) is a condition that equates all common attributes
    - \(A\) is the concatenated list of attributes of \(R\) and \(S\) with no duplicates
    - you may view this above as a rewriting rule
  - **Theta Join**
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

- **γ** GroupByList; \(\text{aggrFn1} \rightarrow \text{attr1}, \ldots, \text{aggrFnN} \rightarrow \text{attrN}\)
- Conceptually, grouping leads to nested tables and is immediately followed by functions that aggregate the nested table
- **Example:**
  - \(\gamma\) Dept; AVG(Salary) \(\rightarrow\) AvgSal, SUM(Salary) \(\rightarrow\) SalaryExp
  - Find the average salary for each department
    - SELECT Dept, AVG(Salary) AS AvgSal, SUM(Salary) AS SalaryExp
    - FROM Employee
    - GROUP BY Dept

Grouping and Aggregation: An Alternate approach

- Operators that combine the GROUP-BY clause with the aggregation operator (AVG, SUM, MIN, MAX, …)
- **SUM** GroupByList; GroupedAttribute \(\rightarrow\) ResultAttribute \(R\) corresponds to
  - SELECT GroupByList,
    - SUM(GroupedAttribute) AS ResultAttribute
  - FROM \(R\)
  - GROUP BY GroupByList
- Similar for AVG, MIN, MAX, COUNT…
- Note that \(\delta(R)\) could be seen as a special case of grouping and aggregation
- **Example**
Sorting and Lists

• SQL and algebra results are ordered
• Could be non-deterministic or dictated by SQL ORDER BY, algebra $\tau$
• $\tau_{\text{OrderByList}}$
• A result of an algebraic expression $o(\text{exp})$ is ordered if
  – If $o$ is a $\tau$
  – If $o$ retains ordering of exp and exp is ordered
    • Unfortunately this depends on implementation of $o$
  – If $o$ creates ordering
  – Consider that leaf of tree may be SCAN(R)

Relational algebra optimization

• Transformation rules
  (preserve equivalence)
• What are good transformations?

Algebraic Rewritings:
Commutativity and Associativity

<table>
<thead>
<tr>
<th>Commutativity</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartesian Product</td>
<td><img src="image" alt="Cartesian Product" /></td>
</tr>
<tr>
<td>Natural Join</td>
<td><img src="image" alt="Natural Join" /></td>
</tr>
</tbody>
</table>

**Question 1**: Do the above hold for both sets and bags?
**Question 2**: Do commutativity and associativity hold for arbitrary Theta Joins?
Algebraic Rewritings: Commutativity and Associativity (2)

**Question 1:** Do the above hold for both sets and bags?
**Question 2:** Is difference commutative and associative?

Algebraic Rewritings for Selection:
Decomposition of Logical Connectives

**Question**

- $\sigma_{\text{cond1 AND cond2}}$
- $\sigma_{\text{cond1 OR cond2}}$

Does it apply to bags?

Algebraic Rewritings for Selection:
Decomposition of Negation

**Question**

- $\sigma_{\text{cond1 AND NOT cond2}}$
- $\sigma_{\text{NOT cond2}}$
- $\sigma_{\text{cond1 OR NOT cond2}}$

Complete
Pushing the Selection Thru Binary Operators: Union and Difference

**Union**

\[ \sigma_{\text{cond}} R \cup \sigma_{\text{cond}} S = \sigma_{\text{cond}} (R \cup S) \]

**Difference**

\[ \sigma_{\text{cond}} R \setminus \sigma_{\text{cond}} S = \sigma_{\text{cond}} (R \setminus S) \]

**Exercise:** Do the rule for intersection

---

Pushing Selection thru Cartesian Product and Join

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only

The right direction requires that all the attributes used by \( \text{cond} \) appear in both \( R \) and \( S \)

**Exercise:** Do the rule for theta join

---

**Rules:** \( \pi, \sigma \) combined

Let \( x = \) subset of \( R \) attributes
\( z = \) attributes in predicate \( P \)
(subset of \( R \) attributes)

\[ \pi_x [ \sigma_p (R) ] = \pi_x \{ \pi_{xz} \sigma_p \pi_{xz} (R) \} \]
Pushing Simple Projections Thru Binary Operators

A projection is simple if it only consists of an attribute list

\[ \pi_p R \cup \pi_p S \]

Union

**Question 1**: Does the above hold for both bags and sets?

**Question 2**: Can projection be pushed below intersection and difference?
Answer for both bags and sets.

Pushing Simple Projections Thru Binary Operators: Join and Cartesian Product

\[ \pi_p R \times \pi_p C \]

Where \( B \) is the list of \( R \) attributes that appear in \( A \).
Similar for \( C \).

**Question**: What is \( B \) and \( C \)?

**Exercise**: Write the rewriting rule that pushes projection below theta join.

Projection Decomposition

\[ \pi_p X \Rightarrow \pi_p Y \]

\[ \pi_p R \]

\[ \pi_p Y \]

\[ \pi_p R \]

\[ \pi_p Y \]
Some Rewriting Rules Related to Aggregation: SUM

- $\sigma_{\text{cond}} \sum_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} R$
  $\iff \sum_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} R$, if $\text{cond}$ involves only the GroupbyList

- $\sum_{\text{GL},\text{GA}} \rightarrow RA(R\cup S) \iff \text{PLUS}_{RA1,RA2:RA}(\text{SUM}_{\text{GL},\text{GA}} \rightarrow RA1 R) > < (\text{SUM}_{\text{GL},\text{GA}} \rightarrow RA2 S))$

- $\sum_{\text{GL2};RA1} \rightarrow RA2 \sum_{\text{GL1};\text{GA}} \rightarrow RA1 R \iff \sum_{\text{GL2},\text{GA}} \rightarrow RA2 R$
  - Question: does the above hold for both bags and sets?

Derived Rules: $\sigma + \bowtie$ combined

More Rules can be Derived:

- $\sigma_{p \land q} (R \bowtie S) =$
- $\sigma_{p \land q \land m} (R \bowtie S) =$
- $\sigma_{p \lor q} (R \bowtie S) =$
  - $p$ only at $R$, $q$ only at $S$, $m$ at both $R$ and $S$

--> Derivation for first one:

- $\sigma_{p \land q} (R \bowtie S) =$
- $\sigma_p [\sigma_q (R \bowtie S)] =$
- $\sigma_p [ R \bowtie \sigma_q (S)] =$
- $[\sigma_p (R)] \bowtie [\sigma_q (S)]$
Which are always “good” transformations?

- $\sigma_{p_1 \land p_2} (R) \rightarrow \sigma_{p_1} [\sigma_{p_2} (R)]$
- $\sigma_p (R \bowtie S) \rightarrow [\sigma_p (R)] \bowtie S$
- $R \bowtie S \rightarrow S \bowtie R$
- $\pi_x [\sigma_p (R)] \rightarrow \pi_x \{\sigma_p [\pi_{xz} (R)]\}$

In textbook: more transformations

- Eliminate common sub-expressions
- Other operations: duplicate elimination

Bottom line:

- No transformation is always good at the l.q.p level
- Usually good:
  - early selections
  - elimination of cartesian products
  - elimination of redundant subexpressions
- Many transformations lead to “promising” plans
  - Commuting/rearranging joins
  - In practice too "combinatorially explosive" to be handled as rewriting of l.q.p.
Algorithms for Relational Algebra Operators

• Three primary techniques
  – Sorting
  – Hashing
  – Indexing

• Three degrees of difficulty
  – data small enough to fit in memory
  – too large to fit in main memory but small enough to be handled by a “two-pass” algorithm
  – so large that “two-pass” methods have to be generalized to “multi-pass” methods (quite unlikely nowadays)

The dominant cost of operators running on disk:

• Count # of disk blocks that must be read (or written) to execute query plan

To estimate costs, we use additional parameters:

\[ B(R) = \text{# of blocks containing } R \text{ tuples} \]
\[ f(R) = \text{max # of tuples of } R \text{ per block} \]
\[ M = \text{# memory blocks available} \]

Sorting information

\[ HT(i) = \text{# levels in index } i \]

Caching information (eg, first levels of index always cached)

\[ LB(i) = \text{# of leaf blocks in index } i \]
Clustering index

Index that allows tuples to be read in an order that corresponds to a sort order

Clustering can radically change cost

- Clustered relation
  - \( R_1 \ R_2 \ R_3 \ R_4 \quad R_5 \ R_5 \ R_7 \ R_8 \quad \ldots \)
- Clustering index

Pipelining can radically change cost

- Interleaving of operations across multiple operators
- Smaller memory footprint, fewer object allocations
- Operators support:
  - \( \text{open()} \)
  - \( \text{getNext()} \)
  - \( \text{close()} \)
- Simple for unary
- Pipelined operation for binary discussed along with physical operators
Example \( R_1 \bowtie R_2 \) over common attribute \( C \)

First we will see main memory-based implementations

- **Iteration join** (conceptually – without taking into account disk block issues)
  
  for each \( r \in R_1 \) do
  
  for each \( s \in R_2 \) do
  
  if \( r.C = s.C \) then output \( r,s \) pair

- **Merge join** (conceptually)
  
  (1) if \( R_1 \) and \( R_2 \) not sorted, sort them
  (2) \( i \leftarrow 1; j \leftarrow 1; \)
  
  While \( (i \leq T(R_1)) \land (j \leq T(R_2)) \) do
  
  if \( R_1(i).C = R_2(j).C \) then outputTuples
  else if \( R_1(i).C > R_2(j).C \) then \( j \leftarrow j+1 \)
  else if \( R_1(i).C < R_2(j).C \) then \( i \leftarrow i+1 \)
Procedure Output-Tuples

While \((R1\{ i \}.C = R2\{ j \}.C) \land (i \leq T(R1))\) do

\(jj \leftarrow j;\)

while \((R1\{ i \}.C = R2\{ jj \}.C) \land (jj \leq T(R2))\) do

[output pair \(R1\{ i \}, R2\{ jj \};\)

\(jj \leftarrow jj+1\)

\(i \leftarrow i+1\)

Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{ i }.C</th>
<th>R2{ j }.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

• Join with index (Conceptually)

For each \(r \in R1\) do

\[ X \leftarrow \text{index}(R2, C, r.C) \]

for each \(s \in X\) do

output \(r, s\) pair

Assume \(R2.C\) index

Note: \(X \leftarrow \text{index}(\text{rel}, \text{attr}, \text{value})\)

then \(X = \text{set of rel tuples with attr = value}\)
• Hash join (conceptual)
  – Hash function \( h \), range \( 0 \rightarrow k \)
  – Buckets for \( R_1 \): \( G_0, G_1, \ldots, G_k \)
  – Buckets for \( R_2 \): \( H_0, H_1, \ldots, H_k \)

**Algorithm**
1. Hash \( R_1 \) tuples into \( G \) buckets
2. Hash \( R_2 \) tuples into \( H \) buckets
3. For \( i = 0 \) to \( k \)
   a. match tuples in \( G_i, H_i \) buckets

**Simple example**

<table>
<thead>
<tr>
<th></th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>Even Buckets</th>
<th>Odd Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 4</td>
<td>5 4</td>
<td>2 4 8</td>
<td>4 12 8 14</td>
</tr>
<tr>
<td>1</td>
<td>3 5</td>
<td>3 3</td>
<td>3 5 9</td>
<td>5 3 13 11</td>
</tr>
<tr>
<td>8</td>
<td>9 12</td>
<td>8 8</td>
<td>8 8 11 14</td>
<td></td>
</tr>
</tbody>
</table>

**Factors that affect performance**

1. Tuples of relation stored physically together?
2. Relations sorted by join attribute?
3. Indexes exist?
Disk-oriented Computation Model

• There are $M$ main memory buffers.
  – Each buffer has the size of a disk block
• The input relation is read one block at a time.
• The cost is the number of blocks read.
• If $B$ consecutive blocks are read the cost is $B/d$.
• The output buffers are not part of the $M$ buffers mentioned above.
  – Pipelining allows the output buffers of an operator to be the input of the next one.
  – We do not count the cost of writing the output.

Notation

• $B(R) =$ number of blocks that $R$ occupies
• $T(R) =$ number of tuples of $R$
• $V(R, [a_1, a_2, \ldots, a_n]) =$ number of distinct tuples in the projection of $R$ on $a_1, a_2, \ldots, a_n$

One-Pass Main Memory Algorithms for Unary Operators

• Assumption: Enough memory to keep the relation
• Projection and selection:
  – Scan the input relation $R$ and apply operator one tuple at a time
  – Incremental cost of "on the fly" operators is 0
• Duplicate elimination and aggregation
  – create one entry for each group and compute the aggregated value of the group
  – it becomes hard to assume that CPU cost is negligible
    • main memory data structures are needed
One-Pass Nested Loop Join

- Assume $B(R)$ is less than $M$
- Tuples of $R$ should be stored in an efficient lookup structure
- **Exercise:** Find the cost of the algorithm below

```plaintext
for each block $Br$ of $R$ do
    store tuples of $Br$ in main memory
for each each block $Bs$ of $S$ do
    for each tuple $s$ of $Bs$
        join tuples of $s$ with matching tuples of $R$
```

Generalization of Nested-Loops

```plaintext
for each chunk of $M$-1 blocks $Br$ of $R$ do
    store tuples of $Br$ in main memory
for each each block $Bs$ of $S$ do
    for each tuple $s$ of $Bs$
        join tuples of $s$ with matching tuples of $R$
```

**Exercise:** Compute cost

Simple Sort-Merge Join

- Assume natural join on $C$
- Sort $R$ on $C$ using the two-phase multiway merge sort
  - if not already sorted
- Sort $S$ on $C$
- Merge (opposite side)
  - assume two pointers $Pr$, $Ps$ to tuples on disk, initially pointing at the start
  - sets $R'$, $S'$ in memory
- **Remarks:**
  - Very low average memory requirement during merging (but no guarantee on how much is needed)
  - Cost:

```plaintext
while $Pr$!=$EOF$ and $Ps$!=$EOF$
    if $Pr[C] == Ps[C]$
        do_cart_prod($Pr$, $Ps$)
    else if $Pr[C] > Ps[C]$
        $Ps++$
    else if $Ps[C] > Pr[C]$
        $Pr++$
function do_cart_prod($Pr$, $Ps$)
    val=$Pr[C]$
    while $Pr[C]$=val
        store tuple *$Pr$ in set $R'$
    while $Ps[C]$=val
        store tuple *$Ps$ in set $S'$
    output cartesian product of $R'$ and $S'$
```
Efficient Sort-Merge Join

• Idea: Save two disk I/O’s per block by combining the second pass of sorting with the "merge".
• Step 1: Create sorted sublists of size $M$ for $R$ and $S$.
• Step 2: Bring the first block of each sublist to a buffer
  – assume no more than $M$ sublists in all
• Step 3: Repeatedly find the least $C$ value $c$ among the first tuples of each sublist. Identify all tuples with join value $c$ and join them.
  – When a buffer has no more tuple that has not already been considered load another block into this buffer.

**Example**

<table>
<thead>
<tr>
<th>$R$</th>
<th>$C$</th>
<th>$RA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r₁</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r₂</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r₃</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>r₂₀</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>$C$</th>
<th>$SA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s₁</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s₅</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>s₁₆</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>s₂₀</td>
<td></td>
</tr>
</tbody>
</table>

Assume that after first phase of multiway sort we get 4 sublists, 2 for $R$ and 2 for $S$.
Also assume that each block contains two tuples.

Two-Pass Hash-Based Algorithms

• General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value.
  – If there are $M$ buffers pick $M-1$ as the number of hash buckets
• Example: Duplicate Elimination
  – Phase 1: Hash each tuple of each input block into one of the $M$-1 bucket/buffers. When a buffer fills save to disk.
  – Phase 2: For each bucket:
    • load the bucket in main memory.
    • treat the bucket as a small relation and eliminate duplicates
    • save the bucket back to disk.
  – Catch: Each bucket has to be less than $M$.
  – Cost:
Hash-Join Algorithms

- Assuming natural join, use a hash function that
  - is the same for both input arguments $R$ and $S$
  - uses only the join attributes
- Phase 1: Hash each tuple of $R$ into one of the $M-1$
buckets $R_i$ and similar each tuple of $S$ into one of
  $S_i$
- Phase 2: For $i=1...M-1$
  - load $R_i$ and $S_i$ in memory
  - join them and save result to disk
- **Question:** What is the maximum size of buckets?
- **Question:** Does hashing maintain sorting?

Index-Based Join: The Simplest
Version

Assume that we do natural join of $R(A,B)$ and $S(B,C)$
and there’s an index on $S$

for each $Br$ in $R$ do
  for each tuple $r$ of $Br$ with $B$ value $b$
    use index of $S$ to find
tuples $\{s_1, s_2, ..., s_n\}$ of $S$ with $B=b$
    output $\{rs_1, rs_2, ..., rs_n\}$

**Cost:** Assuming $R$ is clustered and non-sorted and the
index on $S$ is clustered on $B$ then
$B(R) + T(R)B(S)/V(S,B) +$ some more for reading index

**Question:** What is the cost if $R$ is sorted?

Opportunities in Joins Using
Sorted Indexes

- Do a conventional Sort-Join avoiding the
  sorting of one or both of the input
  operands
• Estimating cost of query plan

(1) Estimating size of results
(2) Estimating # of IOs

Estimating result size

• Keep statistics for relation R
  – \( T(R) \) : # tuples in R
  – \( S(R) \) : # of bytes in each R tuple
  – \( B(R) \) : # of blocks to hold all R tuples
  – \( V(R, A) \) : # distinct values in R for attribute A

Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>30</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>40</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>1</td>
<td>50</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

A: 20 byte string  B: 4 byte integer  C: 8 byte date  D: 5 byte string

\( T(R) = 5 \)  \( S(R) = 37 \)
\( V(R,A) = 3 \)  \( V(R,C) = 5 \)
\( V(R,B) = 1 \)  \( V(R,D) = 4 \)
Size estimates for $W = R_1 \times R_2$

$T(W) = T(R_1) \times T(R_2)$

$S(W) = S(R_1) + S(R_2)$

Size estimate for $W = \sigma_{z-val} (R)$

$S(W) = S(R)$

$T(W) = ?$

Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>30</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>40</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>1</td>
<td>50</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

$V(R,A)=3$  $V(R,B)=1$  $V(R,C)=5$  $V(R,D)=4$

$W = \sigma_{z-val}(R) \quad T(W) = \frac{T(R)}{V(R,Z)}$
What about $W = \sigma_{z \geq \text{val}(R)}$?

$T(W) = ?$

- **Solution #1:**
  $T(W) = \frac{T(R)}{2}$

- **Solution #2:**
  $T(W) = \frac{T(R)}{3}$

- **Solution #3:** Estimate values in range

**Example:**

<table>
<thead>
<tr>
<th>R</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min=1</td>
<td>V(R,Z)=10</td>
</tr>
<tr>
<td>Max=20</td>
<td>W= \sigma_{z \geq 15} (R)</td>
</tr>
</tbody>
</table>

$f = \frac{20-15+1}{20-1+1} = \frac{6}{20} = \frac{3}{10}$ (fraction of range)

$T(W) = f \times T(R)$

Equivalently:

$f \times V(R,Z) = \text{fraction of distinct values}$

$T(W) = \frac{f \times V(Z,R)}{V(Z,R)} \times T(R) = f \times T(R)$
Size estimate for \( W = R_1 \Join R_2 \)

Let \( x = \) attributes of \( R_1 \)
\( y = \) attributes of \( R_2 \)

**Case 1**
\[ X \cap Y = \emptyset \]

Same as \( R_1 \times R_2 \)

**Case 2**
\[ W = R_1 \Join R_2 \quad X \cap Y = A \]

| \( R_1 \) | \( A \) | \( B \) | \( C \) |
| \( R_2 \) | \( A \) | \( D \) |

Assumption:
\[ \Pi_A R_1 \subseteq \Pi_A R_2 \Rightarrow \text{Every A value in } R_1 \text{ is in } R_2 \]
(typically \( A \) of \( R_1 \) is foreign key of the primary key of \( A \) of \( R_2 \))

\[ \Pi_A R_2 \subseteq \Pi_A R_1 \Rightarrow \text{Every A value in } R_2 \text{ is in } R_1 \]
“containment of value sets” (justified by primary key – foreign key relationship)

Computing \( T(W) \) when \( A \) of \( R_1 \) is the foreign key \( \Pi_A R_1 \subseteq \Pi_A R_2 \)

| \( R_1 \) | \( A \) | \( B \) | \( C \) |
| \( R_2 \) | \( A \) | \( D \) |

1 tuple of \( R_1 \) matches with exactly 1 tuple of \( R_2 \)
so \( T(W) = T(R_1) \)
Another way to approach when

\[ \Pi_{A} R_1 \subseteq \Pi_{A} R_2 \]

\[
\begin{array}{ccc|c|c}
R_1 & A & B & C & R_2 & A & D \\
\hline
\end{array}
\]

Take 1 tuple

1 tuple matches with \( \frac{T(R_2)}{V(R_2,A)} \) tuples...

so \( T(W) = \frac{T(R_2)}{V(R_2,A)} \times T(R_1) \)

\( V(R_1,A) \leq V(R_2,A) \)

\( T(W) = \frac{T(R_2) \times T(R_1)}{V(R_2,A)} \)

In general \( W = R_1 \bowtie R_2 \)

\[
T(W) = \frac{T(R_2) \times T(R_1)}{\max\{V(R_1,A), V(R_2,A)\}}
\]

[A is common attribute]
Combining estimates on subexpressions:

Value preservation

Value preservation may have to be pushed to a weird assumption (but there’s logic behind it!)

Value preservation of join attribute
If in doubt, think in terms of probabilities and matching records

- A SID of Student appears in CSEEnroll with probability 1000/20000
- i.e., 5% of students are enrolled in CSE
- A SID of Student appears in Honors with probability 500/20000
- i.e., 2.5% of students are honors students
- An SID of Student appears in the join result with probability 5% x 2.5%
- i.e., each CSE-enrolled student has 10 enrollments
- On the average, each SID of Honors appears in 5000/500 tuples
- i.e., each honors' student has 10 honors
- Each Student SID that is in both Honors and CSEEnroll is in 10x10 result tuples
- T(result) = 20,000 x 5% x 2.5% x 10 = 2,500 tuples

Plan Enumeration

- A smart exhaustive algorithm
  - According to textbook’s Section 16.6
  - no ppt notes
- The INGRES heuristic for plan enumeration

Arranging the Join Order: the Wong-Youssefi algorithm (INGRES)

Sample TPC-H Schema

Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
Lineitem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, SName)

SELECT SName
FROM Nation, Customer, Order, Lineitem, Product, Supplier
WHERE Nation.NationKey = Customer.NationKey
AND Customer.CustKey = Order.CustKey
AND Order.OrderKey = Lineitem.OrderKey
AND Lineitem.PartKey = Product.PartKey
AND Product.Suppkey = Supplier.SuppKey
AND NName = “Canada”

Find the names of suppliers that sell a product that appears in a line item of an order made by a customer who is in Canada
Challenges with Large Natural Join Expressions

For simplicity, assume that in the query
1. All joins are natural
2. whenever two tables of the FROM clause have common attributes we join on them
3. Consider Right-Index only

\[ \sigma_{\text{NName}=\text{Canada}} \pi_{\text{SName}} \]

One possible order

Multiple Possible Orders

Wong-Yussefi algorithm
assumptions and objectives

- Assumption 1 (weak): Indexes on all join attributes (keys and foreign keys)
- Assumption 2 (strong): At least one selection creates a small relation
  - A join with a small relation results in a small relation
- Objective: Create sequence of index-based joins such that all intermediate results are small
Hypergraphs

- relation hyperedges
- two hyperedges for same relation are possible
- each node is an attribute
- can extend for non-natural equality joins by merging nodes

Small Relations/Hypergraph Reduction

- Pick a small relation (and its conditions) to start the plan
- Remove small relation (hypergraph reduction) and color as "small" any relation that joins with the removed "small" relation
- Pick a small relation (and its conditions if any) and join it with the small relation that has been reduced
After a bunch of steps...

Multiple Instances of Each Relation

SELECT S.SName
FROM Nation, Customer, Order, LineItem L, Product P, Supplier S,
LineItem LE, Product PE, Supplier Enron
WHERE Nation.NationKey = Customer.NationKey
AND Customer.CustKey = Order.CustKey
AND Order.OrderKey=L.OrderKey
AND L.PartKey= P.Partkey
AND P.Suppkey = S.SuppKey
AND Order.OrderKey=LE.OrderKey
AND LE.PartKey= PE.Partkey
AND PE.Suppkey = Enron.SuppKey
AND Enron.Sname = "Enron"
AND NName = "Cayman"

Find the names of suppliers whose products appear in an order made by a customer who is in Cayman Islands and an Enron product appears in the same order

Multiple Instances of Each Relation
Multiple choices are possible
The basic dynamic programming approach to enumerating plans

for each sub-expression
\( op(e_1, e_2 \ldots e_n) \) of the logical plan
- (recursively) compute the best plan and cost for each subexpression \( e_i \)
- for each physical operator \( op^p \) implementing \( op \)
  - evaluate the cost of computing \( op \) using \( op^p \) and the best plan for each subexpression \( e_i \)
  - (for faster search) memo the best \( op^p \)

Local suboptimality of basic approach and the Selinger improvement

- Basic dynamic programming may lead to (globally) suboptimal solutions
- Reason: A suboptimal plan for \( e_i \) may lead to the optimal plan for \( op(e_1, e_2 \ldots e_n) \)
  - Eg. consider \( e_1 \) as \( e_2 \) and assume that the optimal computation of \( e_1 \) produces unsorted result
  - Optimal \( \sqcup \sqcap \) is via sort-merge join on A
  - It could have paid off to consider the suboptimal computation of \( e_1 \) that produces result sorted on A
- Selinger improvement: memo also any plan (that computes a subexpression) and produces an order that may be of use to ancestor operators
Using dynamic programming to optimize a join expression

• Goal: Decide the join order and join methods
• Initiate with n-ary join $\Join_{C}(e_1, e_2, \ldots, e_n)$, where $c$ involves only join conditions
• Bottom up: consider 2-way non-trivial joins, then 3-way non-trivial joins etc
  – “non trivial” -> no cartesian product