Query Processing Notes

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

R x S

<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>

Answer: \[ \begin{array}{c} B \\ 2 \\ D \\ x \end{array} \]
Relational Algebra - can be enhanced to describe plans...

Ex: Plan I

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

OR:

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

"FLY" and "SCAN" are the defaults

Ex: Plan I

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

Another idea:

Plan II

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

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

<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>C</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
</tr>
<tr>
<td>20</td>
<td>y</td>
</tr>
<tr>
<td>30</td>
<td>z</td>
</tr>
<tr>
<td>40</td>
<td>x</td>
</tr>
<tr>
<td>50</td>
<td>y</td>
</tr>
</tbody>
</table>

**Output:** <2,x>

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

```
\[
\begin{array}{c}
\sigma_{\text{E}=2} \ 
\text{INDEX} \ 
\pi_{\text{R,B,S,D}} \ 
\sigma_{\text{a} \leq c} \ 
\text{INDEX} \ 
\sigma_{\text{R,I}} \ 
\text{R} \ 
\end{array}
\]
```

**Right Index Join**

**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>
  <SelSet>
    SELECT title
    FROM StarsIn
    WHERE starName IN (  
      SELECT name
      FROM MovieStar
      WHERE birthdate LIKE "%1960"
    )
  
  <FromSet>
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE "%1960"
  
  <TupleSet>
    SELECT title
    FROM StarsIn
    WHERE starName IN (  
      SELECT name
      FROM MovieStar
      WHERE birthdate LIKE "%1960"
    )

(Hint: Each query can be represented as a parse tree.)
Example: Generating Relational Algebra

$\Pi_{\text{title}}$

$\sigma$  

StarsIn  

$\langle \text{condition} \rangle$

$\langle \text{tuple} \rangle$

$\langle \text{IN} \rangle$

$\Pi_{\text{name}}$

$\langle \text{attribute} \rangle$

birthdate LIKE '%1960''

StarName  

MovieStar

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

Example: Logical Query Plan (Relational Algebra)

$\Pi_{\text{title}}$

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

$\times$

StarsIn  

$\Pi_{\text{name}}$

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

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

$\Pi_{\text{title}}$

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

StarsIn  

$\Pi_{\text{name}}$

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

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 UNION 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 \times 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 \( r \) 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**
    - \( R \bowtie S = \pi_A \sigma_C (R \times S) \) where
      - \( C \) 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 that above as a rewriting rule
  - **Theta Join**
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

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

<table>
<thead>
<tr>
<th>Dept</th>
<th>AvgSal</th>
<th>SalaryExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>PCs</td>
<td>45</td>
<td>90</td>
</tr>
</tbody>
</table>

Grouping and Aggregation: An Alternate approach

- Operators that combine the \( \text{GROUP-BY} \) clause with the aggregation operator (\( \text{AVG, SUM, MIN, MAX, …} \))
- \( \text{SUM} \) GroupByList GroupedAttribute \( \rightarrow \) ResultAttribute
  - \( R \) corresponds to \( \text{SELECT GroupByList, SUM(GroupedAttribute) AS ResultAttribute FROM R GROUP BY GroupByList} \)
- Similar for \( \text{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 \( \tau \)
• \( T_{\text{OrderByList}} \)
• A result of an algebraic expression \( o(\text{exp}) \) is ordered if
  – If \( o \) is a \( \tau \)
  – If \( o \) retains ordering of \( \text{exp} \) and \( \text{exp} \) 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></td>
</tr>
<tr>
<td>Natural Join</td>
<td></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)

**Commutativity**

- Union: $R \cup S = S \cup R$
- Intersection: $R \cap S = S \cap R$

**Associativity**

- Union: $(R \cup S) \cup T = R \cup (S \cup T)$
- Intersection: $(R \cap S) \cap T = R \cap (S \cap T)$

**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

- \( \sigma_{\text{cond1 AND cond2}} R \)
- \( \sigma_{\text{cond1 OR cond2}} R \)

Does it apply to bags?

---

Algebraic Rewritings for Selection:
Decomposition of Negation

- \( \sigma_{\text{cond1 AND NOT cond2}} R \)
- \( \sigma_{\text{NOT cond2}} R \)
- \( \sigma_{\text{cond1 OR NOT cond2}} R \)

Complete
Pushing the Selection Thru Binary Operators: Union and Difference

\[ \sigma_{cond}(R) \quad \leftarrow \quad \sigma_{cond}(S) \quad \text{Union} \]

\[ \sigma_{cond}(R) \quad \leftarrow \quad \sigma_{cond}(S) \quad \text{Difference} \]

Exercise: Do the rule for intersection

Pushing Selection thru Cartesian Product and Join

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

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

The right direction requires that all the attributes used by \( cond \) 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)

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

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

\[ \pi_{\mathbf{A}} \mathbf{R} \cup \mathbf{S} \]

**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

\[ \pi_{\mathbf{A}} \mathbf{R} \times \mathbf{S} \]

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

\[ \pi_{\mathbf{A}} \mathbf{R} \]
Some Rewriting Rules Related to Aggregation: SUM

- \( \sigma_{\text{cond}} \ SUM_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} \)  
  \( \Leftrightarrow \ SUM_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} \)  
  if \( \text{cond} \) involves only the \text{GroupbyList}

- \( SUM_{\text{GL};\text{GA}} \rightarrow RA \)  
  \( \Leftrightarrow \) PLUS \( RA_1, RA_2 \rightarrow RA \)  
  \( ((SUM_{\text{GL};\text{GA}} \rightarrow RA_1) \upharpoonright (SUM_{\text{GL};\text{GA}} \rightarrow RA_2)) \)

- \( SUM_{\text{GL2};RA_1} \rightarrow RA_2 \)  
  \( SUM_{\text{GL1};GA} \rightarrow RA_1 \)  
  \( \Rightarrow \)  
  \( SUM_{\text{GL2};GA} \rightarrow RA_2 \)

- **Question**: does the above hold for both bags and sets?

**Derived Rules**: \( \sigma + \bowtie \) combined

**More Rules can be Derived**:

- \( \sigma_{pq} (R \bowtie S) = \)
- \( \sigma_{pq,m} (R \bowtie S) = \)
- \( \sigma_{pq} (R \bowtie S) = \)

  \( p \) only at \( R \), \( q \) only at \( S \), \( m \) at both \( R \) and \( S \)

--> Derivation for first one:

- \( \sigma_{pq} (R \bowtie S) = \)
- \( \sigma_p [\sigma_q (R \bowtie S) ] = \)
- \( \sigma_p [ R \bowtie \sigma_q (S) ] = \)

  \( [\sigma_p (R)] \bowtie [\sigma_q (S)] \)
Which are always “good” transformations?

- \( \sigma_{p1 \land p2} (R) \rightarrow \sigma_{p1} [\sigma_{p2} (R)] \)
- \( \sigma_{p} (R \bowtie S) \rightarrow [\sigma_{p} (R)] \bowtie S \)
- \( R \bowtie S \rightarrow S \bowtie R \)
- \( \pi_x [\sigma_{p} (R)] \rightarrow \pi_x \{ \sigma_p \{ \pi_x (R) \} \} \)

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:

- \( B(R) = \) # of blocks containing \( R \) tuples
- \( f(R) = \) max # of tuples of \( R \) per block
- \( M = \) # memory blocks available
- Sorting information
  - \( HT(i) = \) # levels in index \( i \)
- Caching information (eg, first levels of index always cached)
  - \( LB(i) = \) # of leaf blocks in index \( i \)
Clustering index

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

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>37</td>
</tr>
</tbody>
</table>

Clustering can radically change cost

- Clustered relation
  - R1 R2 R3 R4
  - R5 R5 R7 R8
- Clustering index

Pipelining can radically change cost

- Interleaving of operations across multiple operators
- Smaller memory footprint, fewer object allocations
- Operators support:
  - open()
  - getNext()
  - 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
\[
\begin{align*}
jj & \leftarrow j; \\
\text{while} \ (R1 \{ i \}.C = R2 \{ jj \}.C) \land (jj \leq T(R2)) \ do \ \\
& \quad [\text{output pair } R1 \{ i \}, R2 \{ jj \}] \\
& \quad jj \leftarrow jj+1  \\
i & \leftarrow i+1
\end{align*}
\]

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></td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>7</td>
</tr>
</tbody>
</table>

- Join with index (Conceptually)

For each \( r \in R1 \) do

\[
[ \ X \leftarrow \text{index} (R2, C, r.C) \\
\quad \text{for each } s \in X \ do \ \\
\quad \text{output } r,s \text{ pair}]
\]

Note: \( X \leftarrow \text{index(rel, attr, value)} \)
then \( X = \text{set of rel tuples with attr = value} \)
• Hash join (conceptual)
  – Hash function h, range 0 \rightarrow k
  – Buckets for R1: G0, G1, ... Gk
  – Buckets for R2: H0, H1, ... Hk

Algorithm
(1) Hash R1 tuples into G buckets
(2) Hash R2 tuples into H buckets
(3) For i = 0 to k do
    match tuples in Gi, Hi buckets

Simple example
hash: even/odd

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>Even</th>
<th>Odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>2 4 8</td>
<td>3 5 9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4 12 14</td>
<td>5 3 13 11</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></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

for each block $Br$ of $R$ do
  store tuples of $Br$ in main memory
for 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

for each chunk of $M$-1 blocks $Br$ of $R$ do
  store tuples of $Br$ in main memory
for 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 $R'$
  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.

Efficient Sort-Merge Join

Example

<table>
<thead>
<tr>
<th>$R$</th>
<th>$C$</th>
<th>$RA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$r_1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$r_2$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$r_3$</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>$r_{20}$</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>$s_1$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$s_5$</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>$s_{16}$</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>$s_{20}$</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 into 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 $R$
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></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>20</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>30</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>40</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>50</td>
<td>d</td>
<td></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 \( W = R_1 \times R_2 \)

\[
\begin{align*}
T(W) &= T(R_1) \times T(R_2) \\
S(W) &= S(R_1) + S(R_2)
\end{align*}
\]

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

\[
\begin{align*}
S(W) &= S(R) \\
T(W) &= ?
\end{align*}
\]

**Example**

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>( V(R,A) = 3 )</th>
<th>( V(R,B) = 1 )</th>
<th>( V(R,C) = 5 )</th>
<th>( V(R,D) = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>20</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>30</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>40</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>1</td>
<td>50</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
W = \sigma_{z-val}(R) \quad T(W) = \frac{T(R)}{V(R,Z)}
\]
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

\[ \begin{array}{c|c|c}
\text{Example} & R & Z \\
\hline
\text{Min} = 1 & V(R, Z) = 10 & W = \sigma_{z \geq 15} (R) \\
\text{Max} = 20 & & \\
\end{array} \]

\[ f = \frac{20 - 15 + 1}{20 - 1 + 1} = \frac{6}{20} = \frac{3}{10} \]

(fraction of range)

\[ T(W) = f \times T(R) \]

Equivalently:

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

\[ T(W) = \left[ f \times V(Z, R) \right] \times T(R) = f \times T(R) \]

\[ V(Z, R) \]
**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**

\( W = R_1 \Join R_2 \quad X \cap Y = A \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>R2</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
</table>

**Assumption:**

\( \Pi_A R_1 \subseteq \Pi_A R_2 \Rightarrow \) Every \( A \) value in \( R_1 \) is in \( R_2 \)  
(typically \( A \) of \( R_1 \) is foreign key of the primary key of \( A \) of \( R_2 \))

\( \Pi_A R_2 \subseteq \Pi_A R_1 \Rightarrow \) Every \( A \) value in \( R_2 \) 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** \( \Pi_A R_1 \subseteq \Pi_A R_2 \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>R2</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
</table>

1 tuple of \( R_1 \) matches with exactly 1 tuple of \( R_2 \)

so \( T(W) = T(R_1) \)
Another way to approach when \( \Pi_A R_1 \subseteq \Pi_A R_2 \)

<table>
<thead>
<tr>
<th>R1</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>A</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Take 1 tuple

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

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

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

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

[A is common attribute]

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

\[ T(W) = \frac{T(R_2) \times T(R_1)}{\max\{ V(R_1, A), V(R_2, A) \}} \]
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
1. Consider Right-Index only

One possible order
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

“Nation” is small because it has the equality selection NName = “Canada”

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…

\[ \text{Index} \]
\[ NName = \text{“Canada”} \]
\[ \text{Nation} \]

\[ \text{Customer} \]
\[ \text{Nation} \]
\[ \text{Order} \]
\[ \text{LineItem} \]
\[ \text{Product} \]
\[ \text{Supplier} \]
\[ \sigma \text{NName}=\text{"Cayman"} \]

\[ \sigma \text{SName}=\text{"Enron"} \]