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? (e.g., 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?

One idea

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

RxA | R.B | R.C | S.C | S.D | S.E
- a 1 10 10 x 2
- a 1 10 20 y 2
- C 2 10 10 x 2

Answer: B 2 D x
Relational Algebra - can be enhanced to describe plans...

Ex: Plan I

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

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

“FLY” and “SCAN” are the defaults

Ex: Plan I

\[ \Pi_{B,D} \]

Another idea:

Plan II

\[ \Pi_{B,D}^{HASH} \sigma_{R.A = 'c'}^{HASH} \sigma_{S.E = 2}^{HASH} (R^{SCAN} \bowtie S^{SCAN}) \]

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
**Algebraic Form of Plan**

![Diagram of 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

SELECT Theater
FROM Movie M, Schedule S
WHERE M.Title = S.Title
AND M.Actor = 'Winger'

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

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

More than one plans may be generated by choosing different primitives.
Example: Generating Relational Algebra

\[ \Pi_{\text{title}} \big( \sigma_{\text{birthdate} \text{ LIKE} \text{ "1960"}}(\text{StarsIn}) \big) \]

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

Example: Logical Query Plan (Relational Algebra)

\[ \Pi_{\text{title}} \big( \delta_{\text{starName} = \text{name}} \big) \times \Pi_{\text{name}} \big( \sigma_{\text{birthdate} \text{ LIKE} \text{ "1960"}}(\text{StarsIn}) \big) \]

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

\[ \Pi_{\text{title}} \big( \sigma_{\text{starName} = \text{name}}(\text{StarsIn}) \big) \times \Pi_{\text{name}} \big( \sigma_{\text{birthdate} \text{ LIKE} \text{ "1960"}}(\text{MovieStar}) \big) \]

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

Example: One Physical Plan

Topics

- Bag 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
- **δ(R)** converts the bag R into a set
  - SQL’s `R UNION S` is really `δ(R ∪ 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 `π_A` as follows:
  - The attribute list may include `x→y` in the list A to indicate that the attribute x is renamed to y
  - Arithmetic or string operators on attributes are allowed. For example,
    - `a+b→x` means that the sum of a and b is renamed into x
    - `c||d→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
  - `MULT_{A,B→C}R`: for each tuple of R, multiply attribute A with attribute B and put the result in a new attribute named C.
  - `PLUS_{A,B→C}R`
  - `CONCAT_{A,B→C}R`
- **Exercise**: Write the above operators using extended projection. Assume the schema of R is `R(A,B,D,E).`
Product 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** $R \natural S = \pi_a \sigma_C(R \times S)$ 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 this above as a rewriting rule
  - **Theta Join**
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

- Operators that combine the GROUP-BY clause with the aggregation operator (AVG, SUM, MIN, MAX, …)

- $\text{SUM}_{\text{GroupbyList}, \text{GroupedAttribute} \rightarrow \text{ResultAttribute}} R$ corresponds to
  - $\text{SELECT GroupbyList,}$
  - $\text{SUM(\text{GroupedAttribute}) AS ResultAttribute}$
  - $\text{FROM R}$
  - $\text{GROUP BY GroupbyList}$

- Similar for AVG, MIN, MAX, COUNT…

- Note that $\delta(R)$ could be seen as a special case of grouping and aggregation

- **Example**

Relational algebra optimization

- Transformation rules
  (preserve equivalence)
- What are good transformations?
Algebraic Rewritings:
Commutativity and Associativity

Commutativity

Associativity

Cartesian Product

Natural Join

**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

Associativity

Union

Intersection

**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

Does it apply to bags?
Algebraic Rewritings for Selection: Decomposition of Negation

Question: \[ \sigma_{\text{cond1 AND NOT cond2}} R \]

Complete: \[ \sigma_{\text{cond1 OR NOT cond2}} R \]

Pushing the Selection Thru Binary Operators: Union and Difference

Exercise: Do the rule for intersection

Pushing Selection thru Cartesian Product and Join

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)

$$\pi_x[\sigma_p(R)] = \pi_x \{ \pi_{px} \}$$

Pushing Simple Projections Thru Binary Operators

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

$$\pi_{RA} \cup \pi_{SA} \rightarrow \pi_{RA} \cup \pi_{SA}$$

**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

$$\pi_{RA} \times \pi_{CB}$$

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

\[
\begin{array}{c}
\pi_X R \\
\rightarrow \\
\pi_X Y R
\end{array}
\]

Some Rewriting Rules Related to Aggregation: SUM

- \( \sigma_{\text{cond}} \) \( \text{SUM}_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} \) \( R \)
  \( \Leftrightarrow \) \( \text{SUM}_{\text{GroupbyList};\text{GroupedAttribute}} \rightarrow \text{ResultAttribute} \) \( \sigma_{\text{cond}} \) \( R \),
  if \( \text{cond} \) involves only the \( \text{GroupbyList} \)

- \( \text{SUM}_{\text{GL};\text{GA}} \rightarrow \text{RA} \) \( (R \cup S) \)
  \( \Leftrightarrow \) \( \text{PLUS}_{\text{RA}_1, \text{RA}_2;\text{RA}} \)
  \( (\text{SUM}_{\text{GL};\text{GA}} \rightarrow \text{RA}_1 \text{R}) \) \( \geq \) \( (\text{SUM}_{\text{GL};\text{GA}} \rightarrow \text{RA}_2 \text{S}) \)\)

- \( \text{SUM}_{\text{GL}_2;\text{RA}_1} \rightarrow \text{RA}_2 \) \( \text{SUM}_{\text{GL}_1;\text{GA}} \rightarrow \text{RA}_1 \) \( \text{R} \)
  \( \Leftrightarrow \) \( \text{SUM}_{\text{GL}_2;\text{GA}} \rightarrow \text{RA}_2 \text{R} \)

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

Derived Rules: \( \sigma + \Join \) combined

More Rules can be Derived:

- \( \sigma_{pq} (R \Join S) = \)

- \( \sigma_{pqam} (R \Join S) = \)

- \( \sigma_{pvq} (R \Join S) = \)

- \( \text{p only at } R, \text{q only at } S, \text{m at both } R \text{ and } S \)
--> 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.

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)


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

Index "NName="Canada"
Nation → Customer → Order "LineItem → Product" Supplier
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

“Nation” is small because it has the equality selection \( \text{NName} = "Canada" \)

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

After a bunch of steps…

"Index" of relation "Nation" with conditions "NName = "Canada""
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.
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)

Estimating IOs:

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

To estimate costs, we may have additional parameters:

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

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

Index that allows tuples to be read in an order that corresponds to physical order

```
A
10
15
17
19
35
37
```

Clustering can radically change cost

- Clustered file organization
  ```
  R1 R2 S1 S2
  R3 R4 S3 S4
  ..... 
  ```
- Clustered relation
  ```
  R1 R2 R3 R4
  R5 R5 R7 R8
  ..... 
  ```
- Clustering index

Example  \( R1 \bowtie R2 \) over common attribute C

\[
\begin{align*}
T(R1) &= 10,000 \\
T(R2) &= 5,000 \\
S(R1) = S(R2) &= \frac{1}{10} \text{ block} \\
\text{Memory available} &= 101 \text{ blocks}
\end{align*}
\]

→ Metric: # of IOs
  (ignoring writing of result)
Caution!

This may not be the best way to compare
- ignoring CPU costs
- ignoring timing
- ignoring double buffering requirements

• 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 \( (i \le T(R_1)) \land (j \le T(R_2)) \) do
        if \( R_1(i).C = R_2(j).C \) then output tuples
        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) ∧ (i ≤ T(R1)) do

[j j ← j;

while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do

[output pair R1{ i }, R2{ jj };

jj ← jj+1 ]

i ← 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></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 ∈ R1 do Assume R2.C index

[ X ← index (R2, C, r.C)

for each s ∈ X do

output r,s pair]

Note: X ← index(rel, attr, value)

then X = 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 \( k \) do
    match tuples in \( G_i, H_i \) buckets

**Simple example**

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

**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 8 14</td>
<td>5 3 13 11</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 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 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:

  ```
  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 $R'$
      *Pr++
    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.

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 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></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></td>
<td></td>
<td>1</td>
<td></td>
<td>a</td>
</tr>
<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:
  $T(W) = T(R)/2$

- Solution #2:
  $T(W) = T(R)/3$

- 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>

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

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

Equivalently:

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

$T(W) = [f \times V(Z,R)] \times T(R) = f \times T(R)$
Size estimate for \( W = R_1 \bowtie R_2 \)

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** \[ W = R_1 \bowtie R_2 \quad X \cap Y = A \]

<table>
<thead>
<tr>
<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>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

\( \Pi_A \mathbf{R}_1 \subseteq \Pi_A \mathbf{R}_2 \)

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

Take 1 tuple

1 tuple matches with \( \frac{T(\mathbf{R}_2)}{V(\mathbf{R}_2, A)} \) tuples...

so \( T(W) = \frac{T(\mathbf{R}_2) \times T(\mathbf{R}_1)}{V(\mathbf{R}_2, A)} \)

• \( V(\mathbf{R}_1, A) \leq V(\mathbf{R}_2, A) \) \( T(W) = \frac{T(\mathbf{R}_2) T(\mathbf{R}_1)}{V(\mathbf{R}_2, A)} \)

• \( V(\mathbf{R}_2, A) \leq V(\mathbf{R}_1, A) \) \( T(W) = \frac{T(\mathbf{R}_2) T(\mathbf{R}_1)}{V(\mathbf{R}_1, A)} \)

[A is common attribute]

In general \( W = \mathbf{R}_1 \bowtie \mathbf{R}_2 \)

\[ T(W) = \frac{T(\mathbf{R}_2) T(\mathbf{R}_1)}{\max\{ V(\mathbf{R}_1, A), V(\mathbf{R}_2, A) \}} \]
Example 1(a) Iteration Join R1 $\bowtie$ R2

- Relations not contiguous
- Recall $T(R1) = 10,000$ $T(R2) = 5,000$
  $S(R1) = S(R2) = 1/10$ block
  MEM=101 blocks

Cost: for each R1 tuple:
  [Read tuple + Read R2]
Total = $10,000 \times [1+5000]=50,010,000$ IOs

Can we do better?

Use our memory
1. Read 100 blocks of R1
2. Read all of R2 (using 1 block) + join
3. Repeat until done

Cost: for each R1 chunk:
  Read chunk: 1000 IOs
  Read R2 5000 IOs
  6000
Total = $10,000 \times 6000 = 60,000$ IOs
• Can we do better?

  □ Reverse join order: \( R_2 \bowtie R_1 \)

  Total = \( \frac{5000 \times (1000 + 10,000)}{1000} \) = 5 \( \times 11,000 = 55,000 \) IOs

  \[
  \begin{align*}
  &\text{Example 1(b) Iteration Join } R_2 \bowtie R_1 \\
  &\text{• Relations contiguous} \\
  &\text{Cost} \\
  &\text{For each } R_2 \text{ chunk:} \\
  &\quad \text{Read chunk: 100 IOs} \\
  &\quad \text{Read } R_1: \quad 1000 \text{ IOs} \\
  &\text{Total} = 5 \text{ chunks } \times 1,100 = 5,500 \text{ IOs}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{Example 1(c) Merge Join} \\
  &\text{• Both } R_1, R_2 \text{ ordered by } C; \text{ relations contiguous} \\
  &\text{Memory} \\
  &\begin{array}{c|c}
  R_1 & \ldots R_1 \\
  \hline
  R_2 & \ldots R_2 \\
  \end{array} \\
  &\text{Total cost: Read } R_1 \text{ cost } + \text{ read } R_2 \text{ cost} \\
  &\quad = 1000 + 500 = 1,500 \text{ IOs}
  \end{align*}
  \]
Example 1(d)  Merge Join

• R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first…. HOW?

One way to sort:  Merge Sort

(i) For each 100 blk chunk of R:
  - Read chunk
  - Sort in memory
  - Write to disk

(ii) Read all chunks + merge + write out
Example 1(d) Merge Join (continued)

R1, R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500 = 7,500 IOs

But: Iteration cost = 5,500
so merge joint does not pay off!

But say
R1 = 10,000 blocks contiguous
R2 = 5,000 blocks not ordered

Iterate: 5000 x (100+10,000) = 50 x 10,100
100
= 505,000 IOs

Merge join: 5(10,000+5,000) = 75,000 IOs

Merge Join (with sort) WINS!
How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks

... 100 chunks ⇒ to merge, need 100 blocks!

In general:
Say $k$ blocks in memory
$x$ blocks for relation sort

$\#$ chunks = $\frac{x}{k}$  size of chunk = $k$

$\#$ chunks $\leq$ buffers available for merge

so... $\frac{x}{k} \leq k$

or $k^2 \geq x$  or  $k \geq \sqrt{x}$

In our example
R1 is 1000 blocks,  $k \geq 31.62$
R2 is 500 blocks,  $k \geq 22.36$

Need at least 32 buffers
Can we improve on merge join?

Hint: do we really need the fully sorted files?

![Diagram of R1 and R2 files being joined](image)

Cost of improved merge join:

\[ C = \text{Read R1} + \text{write R1 into runs} + \text{read R2} + \text{write R2 into runs} + \text{join} \]
\[ = 2000 + 1000 + 1500 = 4500 \]

---> Memory requirement?

Example 1(e)  Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered
- Assume R1.C index fits in memory
Cost: Reads: 500 IOs
for each R2 tuple:
- probe index - free
- if match, read R1 tuple: 1 IO

What is expected # of matching tuples?

(a) say R1.C is key, R2.C is foreign key
then expect = 1

(b) say V(R1,C) = 5000, T(R1) = 10,000
with uniform assumption
expect = 10,000/5,000 = 2

What is expected # of matching tuples?

(c) Say DOM(R1, C)=1,000,000
T(R1) = 10,000
with alternate assumption
Expect = $\frac{10,000}{1,000,000} = 0.01$
Total cost with index join

(a) Total cost = 500+5000(1) \cdot 1 = 5,500

(b) Total cost = 500+5000(2) \cdot 1 = 10,500

(c) Total cost = 500+5000(1/100) \cdot 1 = 550

What if index does not fit in memory?

Example: say R1.C index is 201 blocks

- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is

\[ E = \frac{(0)99 + (1)101}{200} \approx 0.5 \]

Total cost (including probes)

\[ = 500+5000 \text{ [Probe + get records]} \]
\[ = 500+5000 \times [0.5+2] \text{ uniform assumption} \]
\[ = 500+12,500 = 13,000 \text{ (case b)} \]

For case (c):
\[ = 500+5000[0.5 \times 1 + (1/100) \times 1] \]
\[ = 500+2500+50 = 3050 \text{ IOs} \]
So far

<table>
<thead>
<tr>
<th>Not contiguous</th>
<th>Iterate R2 △ R1</th>
<th>Merge Join</th>
<th>Sort+ Merge Join</th>
<th>R1.C Index</th>
<th>R2.C Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>55,000 (best)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contiguous</th>
<th>Iterate R2 △ R1</th>
<th>Merge join</th>
<th>Sort+Merge Join</th>
<th>R1.C Index</th>
<th>R2.C Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5500</td>
<td>1500</td>
<td>7500 → 4500</td>
<td>5500 → 3050 → 550</td>
<td></td>
</tr>
</tbody>
</table>


Example 1(f) Hash Join

- R1, R2 contiguous (un-ordered)
  - Use 100 buckets
  - Read R1, hash, + write buckets

R1 →

- Same for R2
- Read one R1 bucket; build memory hash table
- Read corresponding R2 bucket + hash probe

Then repeat for all buckets
Cost:

"Bucketize:" Read R1 + write
Read R2 + write
Join: Read R1, R2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum memory requirements:

Size of R1 bucket = (x/k)
  k = number of memory buffers
  x = number of R1 blocks

So... \( (x/k) < k \)

\( k > \sqrt{x} \) need: k+1 total memory buffers

Trick: keep some buckets in memory

E.g., \( k' = 33 \) R1 buckets = 31 blocks
  keep 2 in memory

called hybrid hash-join
Next: Bucketize R2
- R2 buckets = $\frac{500}{33} = 16$ blocks
- Two of the R2 buckets joined immediately with G0, G1

Finally: Join remaining buckets
- for each bucket pair:
  * read one of the buckets into memory
  * join with second bucket

Cost
- Bucketize R1 = $1000 + 31 \times 31 = 1961$
- To bucketize R2, only write 31 buckets:
  so, cost = $500 + 31 \times 16 = 996$
- To compare join (2 buckets already done)
  read $31 \times 31 + 31 \times 16 = 1457$

Total cost = $1961 + 996 + 1457 = 4414$
• How many buckets in memory?

OR...

See textbook for answer...

Another hash join trick:

• Only write into buckets <val,ptr> pairs
• When we get a match in join phase, must fetch tuples

To illustrate cost computation, assume:
– 100 <val,ptr> pairs/block
– expected number of result tuples is 100

• Build hash table for R2 in memory
  5000 tuples \rightarrow 5000/100 = 50 blocks
• Read R1 and match
• Read \sim 100 R2 tuples

Total cost = \begin{align*}
\text{Read R2:} & \quad 500 \\
\text{Read R1:} & \quad 1000 \\
\text{Get tuples:} & \quad \frac{100}{1600}
\end{align*}
So far:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate</td>
<td>5500</td>
</tr>
<tr>
<td>Merge join</td>
<td>1500</td>
</tr>
<tr>
<td>Sort+merge joint</td>
<td>7500</td>
</tr>
<tr>
<td>R1.C index</td>
<td>5500</td>
</tr>
<tr>
<td>R2.C index</td>
<td>550</td>
</tr>
<tr>
<td>Build R.C index</td>
<td></td>
</tr>
<tr>
<td>Build S.C index</td>
<td></td>
</tr>
<tr>
<td>Hash join</td>
<td>4500+</td>
</tr>
<tr>
<td>with trick, R1 first</td>
<td>4414</td>
</tr>
<tr>
<td>with trick, R2 first</td>
<td></td>
</tr>
<tr>
<td>Hash join, pointers</td>
<td>1600</td>
</tr>
</tbody>
</table>

Summary

- Iteration ok for “small” relations (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, hash join usually best
- Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use merge join
- If index exists, it could be useful (depends on expected result size)