Hybrid Merge-join

- If one relation is sorted, and the other has a secondary B*-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B*-tree
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
  - Sequential scan more efficient than random lookup

![Diagram of Hybrid Merge-join](image)
Exercise

- Compute depositor \( \bowtie \) customer, with depositor as the outer relation.
- Customer has a secondary B+-tree index on customer-name
  - Blocking factor 20 keys
  - \( \#\text{customer} = 400b/10,000t \) \( \#\text{depositor} = 100b/5,000t \)
- Merge join
  - \( \log(400) + \log(100) + 400 + 100 = 516 \)

Hash join

- 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 hash: even/odd

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>2 4 8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4 12 8 14</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Even

- 2 4 8
- R1

Odd:

- 3 5 9
- 13 11
- R2
Example: Hash Join

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

\[ R1 \rightarrow \]

Same for R2

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

Relation R1 is called the build input and R2 is called the probe input.

Then repeat for all buckets

Cost:

“Bucketize:” Read R1 + write
Read R2 + write
Join: Read R1, R2

Total cost = \(3 \times (b_1 + b_2)\)

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 = \( \frac{x}{k} \)
- \( k \) = number of memory buffers
- \( x \) = number of R1 blocks
So...
\( \frac{x}{k} < k \)
\( k > \sqrt{x} \)
Which relation should be the build relation?

Example of Cost of Hash-Join

- Assume that memory size is 20 blocks
- \( b_{depositor} = 100 \) and \( b_{customer} = 400 \).
- \( depositor \) is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition \( customer \) into five partitions, each of size 80. This is also done in one pass.
- Therefore total cost: \( 3(100 + 400) = 1500 \) block transfers
  - ignores cost of writing partially filled blocks

Complex Joins

- Join with a conjunctive condition:
  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins \( r \times_i s \)
    - Final result comprises those tuples in the intermediate result that satisfy the remaining conditions
      \( \theta_1 \wedge \ldots \wedge \theta_i \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n \)
- Join with a disjunctive condition
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins \( r \times_i s \):
    \( (r \times_i s) \cup (r \times_{i+1} s) \cup \ldots \cup (r \times_n s) \)
Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
- **Projection** is implemented by performing projection on each tuple followed by duplicate elimination.

Other Operations: Aggregation

- **Aggregation**
  - Sorting
  - Hashing

Other Operations: Set Operations

- **Set operations** (∪, ∩ and ⎯)
  - can either use variant of merge-join after sorting
  - or variant of hash-join.
Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - Materialization: generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk. Repeat.
  - Pipelining: pass on tuples to parent operations even as an operation is being executed

Query Processing

\[ Q \rightarrow \text{Query Plan} \]

Focus: Relational System

- Others?

Example

Select B,D
From R,S
Where R.A = "c" \∧ S.E = 2 \∧ R.C=S.C
Relational Algebra can be used to describe plans...

Ex: Plan I
\[ \Pi_{B,D} (\sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \bowtie S)) \]

OR: \[ \Pi_{B,D} (\sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \bowtie S)) \]

Another idea:

Plan II
\[ \Pi_{B,D} (\sigma_{R.A = "c"} \bowtie \sigma_{S.E = 2} (R \bowtie S)) \]

Example: Estimate costs

\[ \text{L.Q.P} \]
\[ \text{R1} \quad \text{P1} \quad \text{C1} \]
\[ \text{P2} \quad \text{C2} \]
\[ \vdots \quad \vdots \quad \vdots \]
\[ \text{Pn} \quad \text{Cn} \]

Pick best!
Relational algebra optimization

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

Rules: Natural joins & cross products & union

\[ R \bowtie S = S \bowtie R \]
\[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]

Note:
- Carry attribute names in results, so order is not important
- Can also write as trees, e.g.:
Rules: Natural joins & cross products & union

\[ R \bowtie S = S \bowtie R \]
\[(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]

\[ R \times S = S \times R \]
\[(R \times S) \times T = R \times (S \times T) \]

\[ R \cup S = S \cup R \]
\[ R \cup (S \cup T) = (R \cup S) \cup T \]

Rules: Selects

\[ \sigma_{p_1 \land p_2}(R) = \sigma_{p_1}(\sigma_{p_2}(R)) \]
\[ \sigma_{p_1 \lor p_2}(R) = \sigma_{p_1}(R) \cup \sigma_{p_2}(R) \]

Rules: Project

Let: \( X = \) set of attributes
\( Y = \) set of attributes
\( XY = X \cup Y \)

\[ \pi_{xy}(R) = \pi_{x}(\pi_{y}(R)) \]

\[ \pi_{xy}(R) = \pi_{y}(\pi_{x}(R)) \]
Let $p$ = predicate with only $R$ attribs
$q$ = predicate with only $S$ attribs
$m$ = predicate with only $R,S$ attribs

$$\sigma_p (R \bowtie S) = \{\sigma_p(R) \bowtie |S$$
$$\sigma_q (R \bowtie S) = \sigma_q(S)$$

Some Rules can be Derived:

$$\sigma_{p\cap q} (R \bowtie S) =$$
$$\sigma_{p\cap q\cap m} (R \bowtie S) =$$
$$\sigma_{pq} (R \bowtie S) =$$

$$\sigma_{p\cap q} (R \bowtie S) = \{\sigma_p(R) \bowtie |S$$
$$\sigma_{p\cap q\cap m} (R \bowtie S) = \sigma_m \left[ (\sigma_p(R) \bowtie (\sigma_q(S)) \right]$$
$$\sigma_{pq} (R \bowtie S) = \left[ (\sigma_p(R) \bowtie |S$$
Rules: $\pi, \sigma$ combined

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

$$\pi_x(\sigma_z(R)) = \pi_x\{\sigma_z(\pi_x(R))\}$$

Rules: $\pi$, $\times$ combined

Let $x$ = subset of $R$ attributes
$y$ = subset of $S$ attributes
$z$ = intersection of $R, S$ attributes

$$\pi_{xy}(R \times S) = \pi_{xy}\{\pi_{xz}(R) \times \pi_{yz}(S)\}$$

$$\pi_{xy}\{\sigma_p(R \times S)\} = \pi_{xy}\{\sigma_p[\pi_{xz}(R) \times \pi_{yz}(S)]\}$$

$z' = z \cup \{\text{attributes used in } P\}$
Rules for $\sigma$, $\pi$ combined with $X$

similar...

e.g., $\sigma_{P_1}(R \times S) = ?$

\[\begin{align*}
\sigma_P(R \cup S) &= \sigma_P(R) \cup \sigma_P(S) \\
\sigma_P(R - S) &= \sigma_P(R) - S = \sigma_P(R) - \sigma_P(S)
\end{align*}\]

Rules $\sigma, \cup$ combined:

\[\begin{align*}
\sigma_{P_1}(R \cup S) &= \sigma_{P_1}(R) \cup \sigma_{P_1}(S) \\
\sigma_{P_1}(R - S) &= \sigma_{P_1}(R) - S = \sigma_{P_1}(R) - \sigma_{P_1}(S)
\end{align*}\]

Which are “good” transformations?

- $\sigma_{P_1 \times P_2}(R) \rightarrow \sigma_{P_1}[\sigma_{P_2}(R)]$
- $\sigma_P(R \bowtie S) \rightarrow [\sigma_P(R)] \bowtie S$
- $R \bowtie S \rightarrow S \bowtie R$
- $\pi_{x_1} [\sigma_P(R)] \rightarrow \pi_{x_1} \{\sigma_P[\pi_{x_2}(R)]\}$
Conventional wisdom: do projects early

Example: \( R(A,B,C,D,E) \) \( x=E \)  
\( P: (A=3) \land (B="cat") \)

\[ \pi_x \{ \sigma_p(R) \} \text{ vs. } \pi_E \{ \sigma_p(\pi_{ABE}(R)) \} \]

But What if we have A, B indexes?

\( B = "cat" \) \( A=3 \)

Intersect pointers to get pointers to matching tuples

Bottom line:
- No transformation is always good
- Usually good: early selections