4) Sort-Merge \((R \bowtie S)\) for Equi-Join

- **Sort** \(R\) and \(S\) on join column using external sorting.
- **Merge** \(R\) and \(S\) on join column, output result tuples.

Repeat until either \(R\) or \(S\) is finished:

- **Scanning**:
  - Advance scan of \(R\) until current \(R\)-tuple \(\geq\) current \(S\) tuple,
  - Advance scan of \(S\) until current \(S\)-tuple \(\geq\) current \(R\) tuple;
  - Do this until current \(R\) tuple = current \(S\) tuple.

- **Matching**:
  - Match all \(R\) tuples and \(S\) tuples with same value (called \(R\)-group and \(S\)-group of the current value).
  - Output \(<r, s>\) for all pairs of such tuples.
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>uppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- **Cost:** $\text{Sorting\_cost}(R) + \text{Sorting\_cost}(S) + \text{Merging\_cost}$
  - $\text{Merging\_cost} \in [M+N, M*N]$
  - $M+N$: *foreign key join* with the referenced reln. as inner.
  - $M*N$: uncommon but possible. When?
- What is the I/O pattern in the sort-merge join?
- How many buffers are needed in the merge phase?
Refinement of Sort-Merge Join

- Is there a **linear-time (2 pass)** algorithm for **Equi-Join**?

- Observed repeated merging phases:
  - *Sorting* of R and S has respective merging phases.
  - *Join* of R and S also has a merging phase.
  - Combine all these merging phases!
Merging in Two-Pass Sort-Merge

Relation R

Relation S

B memory buffer pages

Run1 of R
Run2 of R
RunK of R

Run1 of S
Run2 of S
RunK of S

OUTPUT

Join Results
Merging in Two-Pass Sort-Merge

Relation R

Relation S

B memory buffer pages

OUTPUT

Join Results
Two-Pass Sort-Merge Join

- **Pass 1 Sorting**: sort subfiles of R and S individually
- **Pass 2 Merging**: merge sorted runs of R and S
  - merge sorted runs of R,
  - merge sorted runs of S, and
  - compare R and S tuples using the join condition.

- There exists a linear (2-pass) algorithm for foreign key-primary key join
Memory Requirement and Cost

- Memory requirement for two-pass sort-merge:
  - Let $U$ be the size of the *larger* relation, $U = \max(M, N)$.
  - *Sorting* pass produces sorted runs of size up to $2B$. So, Number of runs per relation $\leq U/2B$.
  - *Merging* pass holds sorted runs of both relations and an output buffer. So, $2*(U/2B) + 1 \leq B \rightarrow B \geq \sqrt{U}$
  - A tighter bound on memory size exists: $B > \sqrt{(M+N)/2}$

- **Cost:** read & write each relation in sorting pass + read each relation in merging pass (+ writing result tuples, ignore here) $= 3(M+N)$!
5) Hash-Join for *Equi-Join*

- **Idea:** For an *Equi-Join*, partition both R and S using a hash function s.t. R tuples will only match S tuples in partition i.

- **Phase 1 Partitioning:** Partition both relations using hash function *h* (R_i tuples will only match with S_i tuples).
Hash-Join

- Phase 2 Probing:
  - Read in partition Ri, build hash table using h2 (<> h!).
  - Scan partition Si, one page at a time, search for matches.
Memory Requirement

- Partitioning: # partitions in memory ≤ B-1,
  Probing: to fit each Ri in memory, size of partition ≤ B-2.
  - A little more memory needed to build hash table, but ignored here.

- Assuming uniformly sized partitions, \( L = \min(M, N) \):
  - \( \frac{L}{B-1} \leq (B-2) \Rightarrow B > \sqrt{L} + 1 \)
  - Use the *smaller* relation as the building relation in probing phase.

- What if hash fn \( h \) does not partition uniformly?
  - One or more R partitions may not fit in memory.
  - Can apply hash-join recursively to this R-partition and the corresponding S-partition. Higher cost, of course…

- What is the I/O pattern in the hash join?
Cost of Hash-Join

- **Partitioning**: reads+writes both relns; 2(M+N).
  **Probing**: reads both relns; M+N I/Os.

  **Total cost = 3(M+N).**
  - In our running example, a total of 4500 I/Os using hash join (compared to 501,000 I/Os w. Page NLJ).

- **Sort-Merge Join vs. Hash Join:**
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os.
  - Hash Join superior on this count if relation sizes differ greatly.
  - Sort-Merge less sensitive to data skew; result is sorted.
Rough Comparisons of Join Methods

- **Block NLJ**: \( L + LU/(B-2) \)
- **Sort Merge**: \([2L \log_{B-1} L + 2U \log_{B-1} U + 3(L+U), L+U]\)
- **(Hybrid) Hash Join (no skew)**: \([3(L+U), L+U]\)
General Join Conditions

- Equalities over several attributes (e.g., $R.sid=S.sid \text{ AND } R.rname=S.sname$):
  - Block NL works fine.
  - For Index NL,
    - use index on <$sid$, $sname$> if available; or
    - use an index on $sid$ or $sname$, check the other predicate on the fly.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
General Join Conditions

- Inequality conditions (e.g., $R.rname < S.sname$):
  - For Index NL, need B+ tree index.
    - Range probes on inner; number of matches likely to be much higher than for equality joins.
    - Clustered index is much preferred.
  - Block NL often works well.
- Hash Join, Sort Merge Join not applicable.
Outline

- Selection
- Sorting routine
- Join
- Projection
- Set operators
- Group By aggregation
- Statistics for estimating output sizes
Cost Estimation

- For each query plan of a SQL query, must estimate its cost.

- Estimate cost of each operation in a plan tree:
  - Depends on input cardinalities.
  - Depends on the method (sequential scan, index scan, join method...)

- Estimate size of result for each operation in tree:
  - Use statistics about input relations.
Statistics in System Catalog

- Statistics about each relation (R) and index (I):
  - Relation cardinality: # tuples (NTuples) in R
  - Relation size: # pages (NPages) in R
  - Index cardinality: # distinct values (NKeys) in I
  - Index size: # pages (INPages) in I
  - Index height: # nonleaf levels (IHeight) of I
  - Index range: low/high key values (Low/High) in I
  - Number of distinct values in an attribute (NKeys)
  - Histogram for an attribute
Size Estimation & Reduction Factors

- **Reduction factor (RF) or Selectivity** of each term reflects the impact of the term in reducing result size.
  - **Assumption 1**: uniform distribution of the values.
  - Term $col=value$: $RF = 1/NKeys(I)$, if there is an index $I$ on $col$.
  - Term $col>value$: $RF = (High(I)-value)/(High(I)-Low(I))$
  - Term $col1=col2$: $RF = 1/\text{MAX}(NKeys(I1), NKeys(I2))$
    - Each value from $R$ with the smaller index $I1$ has a matching value in $S$ with the larger index $I2$; \{distinct val. In $R$\} is subset of \{distinct val. In $S$\}.
    - Values in $S$ are evenly distributed.
    - So each $R$ tuple has $NTuples(S)/NKeys(I2)$ matches, a RF of $1/NKeys(I2)$. 

```
SELECT attribute list
FROM relation list
WHERE term_1 AND ... AND term_k
```
Size Estimation & Reduction Factors

\[\text{SELECT attribute list} \]
\[\text{FROM relation list} \]
\[\text{WHERE term1 AND ... AND termk} \]

- **Reduction factor (RF) or Selectivity of each term:**
  - **Assumption 1:** uniform distribution of the values.
  - Term \(col=value\): \(RF = \frac{1}{N\text{Keys}(I)}\)
  - Term \(col>value\): \(RF = \frac{\text{High}(I)-value}{\text{High}(I)-\text{Low}(I)}\)
  - Term \(col1=col2\): \(RF = \frac{1}{\text{MAX}(N\text{Keys}(I1), N\text{Keys}(I2))}\)

- **Result cardinality** = Max \# tuples \* product of all RF’s.
  - Max \# tuples = the product of the cardinalities of relations in the FROM clause.
  - **Assumption 2:** terms are independent.
Issue with Assumption 1

- “Uniform distribution of values”: often causes highly inaccurate estimates
  - E.g., distribution of gender: male (15), female (4)
  - E.g., distribution of age:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Nkeys = 15, count = 45.
Reduction factor of ‘age=14’: 1/15? 9/45=1/5!

- Histogram: approximates a data distribution
### Equiwidth Histograms

**Equiwidth Histogram:** buckets of equal size.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>8/3</th>
<th>4/3</th>
<th>15/3</th>
<th>3/3</th>
<th>15/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
<td>8</td>
<td>4</td>
<td>15</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Buckets</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Still not accurate for value 14 with true frequency 9
# Equidepth Histograms

**Equidepth**: roughly equal counts of buckets.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>9/4</th>
<th>10/4</th>
<th>10/2</th>
<th>7/4</th>
<th>9/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

- Small errors for infrequent items: tolerable.
- Now accurate for value 14: 9

- Favors *frequent* values.
- What if we have 100 distinct values and 20 frequent values?
  - Keep a list of *most common values* (MCV’s) with their frequencies.
  - The histogram excludes MCV’s.
## Equidepth Histograms

**Equidepth**: roughly equal counts of buckets.

<table>
<thead>
<tr>
<th>Buckets</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Frequency</td>
<td>9/4</td>
<td>10/4</td>
<td>10/2</td>
<td>7/4</td>
<td>9/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Small errors for infrequent items: tolerable.

Now accurate for value 14: 9

- **Implementation is often sampling-based:**
  - Boundaries of 5 buckets \{0, 4, 8, 10, 14, 14\}
  - Count of tuples for each bucket (optional)
  - Number of distinct values for each bucket (optional)

- Reduction factor for “age = 12”?
- Reduction factor for “age < 12”?
Issue with Assumption 2

- “Independence of predicates”: often causes inaccurate estimates
  - E.g., Car DB: 10 makes, 100 models.
  - RF of make=‘honda’ and model=‘civic’
  - If independent, 1/10 * 1/100. In practice, much higher!

- Multi-dimensional histograms [PI’97, MVW’98, GKT’00]
  - Maintain counts and frequency in multi-attribute space.

- Dependency-based histograms [DGR’01]
  - Learn dependency between attributes and compute conditional probability \( P(\text{model}=\text{‘civic’} \mid \text{make}=\text{‘honda’}) \)
  - Can use graphical models…