Query Processing
Review

- Support for data retrieval at the physical level:
  - **Indices**: data structures to help with some query evaluation:
    - SELECTION queries (ssn = 123)
    - RANGE queries (100 <= ssn <=200)
  - **Index choices**: Primary vs secondary, dense vs sparse, ISAM vs B+-tree vs Extendible Hashing vs Linear Hashing

- Sometimes, indexes not useful, even for SELECTION queries.

- And what about join queries or other queries not directly supported by the indices? How do we evaluate these queries?

- What decides these implementation choices?

**Ans**: Query Processor (one of the most complex components of a database system)
QP & O

SQL Query

Query Processor

Data: result of the query
QP & O

SQL Query

Query Processor

Parser

Algebraic Expression

Evaluator

Query Optimizer

Execution plan

Data: result of the query
Parser / translator (1st step)

Input: SQL Query (or OQL, …)
Output: Algebraic representation of query (relational algebra expression)

Eg SELECT balance
    FROM account
    WHERE balance < 2500

\[
\Pi_{\text{balance}}(\sigma_{\text{balance} < 2500}(\text{account}))
\]

or

\[
\Pi_{\text{balance}}(\sigma_{\text{balance} < 2500}(\text{account}))
\]
Plan Generator produces:

Query execution plan

- Algorithms of operators that read from disk:
  - Sequential scan
  - Index scan
  - Merge-sort join
  - Nested loop join
  - ...

Plan Evaluator (last step)

Input: Query Execution Plan
Output: Data  (Query results)
Query Processing & Optimization

Query Rewriting

Input: Algebraic representation of query
Output: Algebraic representation of query

Idea: Apply heuristics to generate equivalent expression that is likely to lead to a better plan

E.g.: \( \sigma_{\text{amount} > 2500} (\text{borrower} \bowtie \text{loan}) \)

Why is 2\textsuperscript{nd} better than 1\textsuperscript{st}?
Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

\[ (E) = \sigma_1 \sigma_2 (E) \]

2. Selection operations are commutative.

\[ \sigma_1 (\sigma_2 (E)) = \sigma_2 (\sigma_1 (E)) \]

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

\[ \Pi_{L_1} (\Pi_{L_2} (\ldots (\Pi_{L_n} (E)) \ldots)) = \Pi_{L_1} (E) \]

4. Selections can be combined with Cartesian products and theta joins.

a. \( \sigma_\theta (E_1 \times E_2) = E_1 \bowtie_\theta E_2 \)

b. \( \sigma_{\theta_1} (E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \land \theta_2} E_2 \)
Query Processing & Optimization

Plan Generator

Input: Algebraic representation of query
Output: Query execution plan

Idea:

1) generate alternative plans for evaluating a query
   - $\sigma_{\text{amount} > 2500}$ Sequential scan
   - Index scan

2) Estimate cost for each plan

3) Choose the plan with the lowest cost

COST: approx., counts sources of latency
Goal: generate plan with minimum cost (i.e., fast as possible)

Cost factors:
1. CPU time (trivial compared to disk time)
2. Disk access time
   - main cost in most DBs
3. Network latency
   - Main concern in distributed DBs

Our metric: count disk accesses
How do we predict the cost of a plan?

Ans: Cost model

- For each plan operator and each algorithm we have a cost formula

- Inputs to formulas depend on relations, attributes, etc.

- Database maintains statistics about relations for this (Metadata)
Metadata

- Given a relation $r$, DBMS likely maintains the following metadata:

  1. **Size** (# tuples in $r$)  \( n_r \)
  2. **Size** (# blocks in $r$)  \( b_r \)
  3. **Block size** (# tuples per block)  \( f_r \)
      (typically \( b_r = \lceil n_r / f_r \rceil \))
  4. **Tuple size** (in bytes)  \( s_r \)
  5. **Attribute Values**  \( V(\text{att}, r) \)
      (for each attribute \( \text{att} \) in $r$, # of different values)
  6. **Selection Cardinality**  \( SC(\text{att}, r) \)
      (for each attribute \( \text{att} \) in $r$,
      \textit{expected} size of a selection: \( \sigma_{\text{att}} = K(r) \))
Example

<table>
<thead>
<tr>
<th>account</th>
<th>bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dntn</td>
<td>A-101</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>Mianus</td>
<td>A-215</td>
<td>700</td>
<td></td>
</tr>
<tr>
<td>Perry</td>
<td>A-102</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>R.H.</td>
<td>A-305</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>Dntn</td>
<td>A-200</td>
<td>700</td>
<td></td>
</tr>
<tr>
<td>Perry</td>
<td>A-301</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

\[ n_{account} = 6 \]
\[ s_{account} = 33 \text{ bytes} \]
\[ f_{account} = \lfloor 4K/33 \rfloor \]

\[ V(balance, account) = 3 \]
\[ V(acct_no, account) = 6 \]

\[ SC (balance, account) = 2 \quad (n_r / V(att, r)) \]
Some typical plans and their costs

Query: $\sigma_{\text{att} = K}(r)$

- **A1** (linear search). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate (number of disk blocks scanned) = $b_r$
    - $b_r$ denotes number of blocks containing records from relation $r$
  - If selection is on a **key attribute**, cost = ($b_r/2$)
    - stop on finding record (on average in the middle of the file)
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

$$E_{A1} = \begin{cases} b_r & \text{(if attr is not a key)} \\ b_r/2 & \text{(if attr is a key)} \end{cases}$$
Selection Operation (Cont.)

Query: $\sigma_{\text{att} = K}(r)$

- **A2 (binary search).** Applicable if selection is an equality comparison on the attribute on which file is sorted.

- Requires that the blocks of a relation are sorted and stored contiguously.

- Cost estimate:
  - $\lceil \log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
  - *Plus* number of blocks containing records that satisfy selection condition

\[
E_{A2} = \lceil \log_2(b_r) \rceil + \left\lceil \text{SC (att, } r) / f_r \right\rceil - 1
\]

What is the cost if att is a key?

- $E_{A2} = \lceil \log_2(b_r) \rceil$

less the one you already found
Example

Account (bname, acct_no, balance)

Query:  \( \sigma_{\text{bname}=\text{"Perry"}}(\text{Account}) \)

\( n_{\text{account}} = 10,000 \)
\( f_{\text{account}} = 20 \text{ tuples/block} \)
\( b_{\text{account}} = 10,000 / 20 = 500 \)
\( V(\text{bname, Account}) = 50 \)

\[
\text{SC}(\text{bname, Account}) = \left\lceil \frac{10,000}{50} \right\rceil = 200
\]

Assume sorted on bname

Cost Estimates:

A1: \( E_{A1} = \left\lceil \frac{n_{\text{Account}}}{f_{\text{Account}}} \right\rceil = \left\lceil \frac{10,000}{20} \right\rceil = 500 \text{ I/O’ s} \)

A2: \( E_{A2} = \left\lceil \log_2(b_{\text{Account}}) \right\rceil + \left\lceil \frac{\text{SC}(\text{bname, Account})}{f_{\text{account}}} \right\rceil - 1 \)
\( = 9 + 9 = 18 \text{ I/O’ s} \)
More Plans for selection

- What if there’s an index (B+Tree) on att?

We need metadata on size of index (i). DBMS keeps track of:

1. Index height: \( H_{T_i} \)
2. Index “Fan Out”: \( f_i \)
   
   Average # of children per node (not same as order.)
3. Index leaf nodes: \( L_{B_i} \)

Note: \( H_{T_i} \sim \lceil \log_{f_i}(L_{B_i}) \rceil \)
B+-trees

REMINDER

B+-tree of order 3:

This is a primary index

root: internal node

leaf node

Data File

3 4

6 7

9 13

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………

(3, Joe, 23)

(3, Bob, 23)

(4, John, 23)

………

………

………
B+Tree, Primary Index

\[
\left\lfloor \frac{SC(\text{att}, r)}{f_r} \right\rfloor = \frac{14}{7}
\]

14 tuples / 7 tuples per block = 2 blocks

Leaf nodes

Data File
More Plans for selection

Query: $\sigma_{att = K} (r)$

- A3: Index scan, Primary ($B^+$-Tree) Index
  What: Follow primary index, searching for key $K$
  Prereq: Primary index on att, $i$

Cost:

\[
EA3 = HT_i + 1, \text{ if att is a candidate key} \\
EA3 = HT_i + 1 + \left \lceil \frac{SC(att, r)}{f_r} \right \rceil, \text{ if not}
\]

Remember for primary index, data file is sorted $\Rightarrow$ sparse index

Number of blocks containing att=$K$ in data file
secondary index: typically, with ‘postings lists’

### STUDENT

<table>
<thead>
<tr>
<th>Ssn</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>smith</td>
<td>main str</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>forbes ave</td>
</tr>
<tr>
<td>345</td>
<td>tomsen</td>
<td>main str</td>
</tr>
<tr>
<td>456</td>
<td>stevens</td>
<td>forbes ave</td>
</tr>
<tr>
<td>567</td>
<td>smith</td>
<td>forbes ave</td>
</tr>
</tbody>
</table>
B+Tree, Primary Index

HTi

Posting List

SC(att, r)

The number of records with value = 3 = # blocks read

The number of records with value = 3 = # blocks read
**A4: Index scan, Secondary Index**

- **Prereq:** Secondary index on att, \( i \)
- **What:** Follow index, searching for key \( K \)

**Cost:**

- If \( att \) not a key:
  
  \[
  E_{A4} = HT_i + 1 + SC(att, r) \]

- Else, if \( att \) is a candidate key: \( E_{A4} = HT_i + 1 \)
How Big is the Posting List

$BPL_r = \text{Blocks in Posting List}$

$P_r = \text{size of a pointer}$ → NEW

$f_r = \text{size of a block}$

$$BPL_r = \frac{SC(attr, r)}{f_r} \frac{f_r}{P_r}$$

Num occurrences

Num pointers in a block
Selections Involving Comparisons

Query: $\sigma_{\text{Att} \geq K}(r)$

- **A5** *(primary index, comparison)*. (Relation is sorted on Att)
  - For $\sigma_{\text{Att} \geq K}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{\text{Att} \leq K}(r)$ just scan relation sequentially until first tuple $> K$; do not use index

Cost of first: $E_{A5} = HT_i + \lceil c / f_r \rceil$ (where $c$ is the cardinality of result)
Query: $\sigma_{\text{Att} \geq K}(r)$

Cardinality: More metadata on $r$ is needed:

- $\text{min}(\text{att}, r)$: minimum value of att in $r$
- $\text{max}(\text{att}, r)$: maximum value of att in $r$

Then the selectivity of $\sigma_{\text{Att} \geq K}(r)$ is estimated as:

$$n_r \frac{\text{max}(\text{attr}, r) - K}{\text{max}(\text{attr}, r) - \text{min}(\text{attr}, r)} \quad \text{(or } \frac{n_r}{2} \text{ if min, max unknown)}$$

Intuition: assume uniform distribution of values between min and max
Plan generation: Range Queries

A6: *(secondary index, comparison)*.

Att is a candidate key

\[ \sigma_{\text{Att} \geq K}(r) \]

Cost: \( E_{A6} = HT_i - 1 + \text{# of leaf nodes to read} + \text{# of file blocks to read} \)

\[ = HT_i - 1 + \left\lceil LB_i \cdot \left( \frac{c}{n_r} \right) \right\rceil + c, \quad \text{if att is a candidate key} \]

- There will be c file pointers for a key.
Plan generation: Range Queries

A6: (secondary index, range query). Att is NOT a candidate key

Query: $\sigma_{K \leq \text{Att} \leq K+m}$ (r)

Leaf nodes

Posting Lists

File

...
Cost: \( E_{A6} = HT_i - 1 + \# \text{ leaf nodes to read} + \)

\[ \# \text{ file blocks to read} + \]

\[ \# \text{ posting list blocks to read} \]

\[ = HT_i - 1 + \left\lceil LB_i \times \left( \frac{c}{n_r} \right) \right\rceil + c + x \]

where \( x \) accounts for the posting lists computed like before.
Cardinalities

Cardinality: the number of tuples in the query result (i.e., size)

Why do we care?

Ans: Cost of every plan depends on $n_r$

  e.g. Linear scan: $b_r = \lceil n_r / f_r \rceil$

But, what if $r$ is the result of another query?

  Must know the size of query results as well as cost

  Size of $\sigma_{\text{att} = K}(r)$?

  ans: $\text{SC(} \text{att, } r \text{)}$
Join Operation

- Size and cost of plans for join operation

- Running example: depositor \( \Join \) customer

Metadata:

- \( n_{\text{customer}} = 10,000 \)
- \( n_{\text{depositor}} = 5000 \)
- \( f_{\text{customer}} = 25 \)
- \( f_{\text{depositor}} = 50 \)
- \( b_{\text{customer}} = 400 \)
- \( b_{\text{depositor}} = 100 \)

\( V(\text{cname, depositor}) = 2500 \)

(each depositor has on average 2 accts)

\textit{cname} in \textit{depositor} is a foreign key (from \textit{customer})
Cardinality of Join Queries

- What is the cardinality (number of tuples) of the join?

**E1**: Cartesian product: \( n_{\text{customer}} \times n_{\text{depositor}} = 50,000,000 \)

**E2**: Attribute **cname** common in both relations,

2500 different cnames in depositor

Size: \( n_{\text{customer}} \times (\text{avg # tuples in depositor with same cname}) \)

= \( n_{\text{customer}} \times (n_{\text{depositor}} / V(\text{cname, depositor})) \)

= \( 10,000 \times (5000 / 2500) \)

= 20,000
Cardinality of Join Queries

E3: cname is a key in customer
    cname is a foreign key (exhaustive) in depositor
    (Star schema case)

Size: \( n_{\text{depositor}} \times (\text{avg # of tuples in customer with same cname}) \)
    \[ = n_{\text{depositor}} \times 1 \quad \text{key} \]
    \[ = 5000 \]

Note: If \textit{cname} is a key for \textit{Customer} but
    NOT an exhaustive foreign key for \textit{Depositor},
    then 5000 is an \textit{UPPER BOUND}

Some customer names may not match w/ any customers in customer
Cardinality of Joins in general

Assume join: $R \bowtie S$ (common attributes are not keys)

1. If $R$, $S$ have no common attributes: $n_r \cdot n_s$
2. If $R, S$ have attribute $A$ in common:

$$\min\left(\frac{n_r}{V(A,s)} \cdot \frac{n_s}{V(A,r)}, \frac{n_s}{V(A,s)} \cdot \frac{n_r}{V(A,r)}\right)$$

- These are not the same when $V(A,s) \neq V(A,r)$.
- When this is true, there are likely to be dangling tuples.
- Thus, the smaller is likely to be more accurate.
Nested-Loop Join

Query: \( R \bowtie S \)

Algorithm 1: Nested Loop Join

**Idea:**

<table>
<thead>
<tr>
<th>Blocks of...</th>
<th>R</th>
<th>S</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t1</td>
<td>u1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t2</td>
<td>u2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t3</td>
<td>u3</td>
<td></td>
</tr>
</tbody>
</table>

Compare: \((t1, u1), (t1, u2), (t1, u3)\) ....

Then: GET NEXT BLOCK OF S

Repeat: for EVERY tuple of R
Nested-Loop Join

Query: $R \bowtie S$

**Algorithm 1**: Nested Loop Join

for each tuple $t_r$ in $R$ do
  for each tuple $u_s$ in $S$ do
    test pair $(t_r, u_s)$ to see if they satisfy the join condition
    if they do (a “match”), add $t_r \cdot u_s$ to the result.

$R$ is called the **outer relation** and $S$ the **inner relation** of the join.
Nested-Loop Join (Cont.)

Cost:

- Worst case, if buffer size is 3 blocks
  \[ b_r + n_r \times b_s \] disk accesses.
- Best case: buffer big enough for entire INNER relation + 2
  \[ b_r + b_s \] disk accesses.
- Assuming worst case memory availability cost estimate is
  ★ 5000 \times 400 + 100 = 2,000,100 disk accesses with *depositor* as outer relation, and
  ★ 10000 \times 100 + 400 = 1,000,400 disk accesses with *customer* as the outer relation.
- If smaller relation (*depositor*) fits entirely in memory (+ 2 more blocks), the cost estimate will be 500 disk accesses.
Join Algorithms

Query: \( R \bowtie S \)

**Algorithm 2: Block Nested Loop Join**

**Idea:**

<table>
<thead>
<tr>
<th>Blocks of...</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>u1</th>
<th>u2</th>
<th>u3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compare:  
- \((t1, u1)\), \((t1, u2)\), \((t1, u3)\)  
- \((t2, u1)\), \((t2, u2)\), \((t2, u3)\)  
- \((t3, u1)\), \((t3, u2)\), \((t3, u3)\)

Then: GET NEXT BLOCK OF S

Repeat: for EVERY BLOCK of R
Block Nested-Loop Join

- Block Nested Loop Join

  for each block $B_R$ of $R$ do

  for each block $B_S$ of $S$ do

  for each tuple $t_r$ in $B_R$ do

  for each tuple $u_s$ in $B_S$ do begin

  Check if $(t_r, u_s)$ satisfy the join condition

  if they do (“match”), add $t_r \cdot u_s$ to the result.
Block Nested-Loop Join (Cont.)

Cost:

- Worst case estimate (3 blocks): \( b_r \times b_s + b_r \) block accesses.
- Best case: \( b_r + b_s \) block accesses. Same as nested loop.

- Improvements to nested loop and block nested loop algorithms for a buffer with \( M \) blocks:
  - In block nested-loop, use \( M - 2 \) disk blocks as blocking unit for outer relation, where \( M \) = memory size in blocks; use remaining two blocks to buffer inner relation and output
    \[
    \text{Cost} = \left\lceil \frac{b_r}{(M-2)} \right\rceil \times b_s + b_r
    \]
  - If equi-join attribute forms a key on inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
Join Algorithms

Query: \( R \bowtie S \)

**Algorithm 3: Indexed Nested Loop Join**

**Idea:**

<table>
<thead>
<tr>
<th>Blocks of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
</tr>
<tr>
<td>t3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
</table>

Assume A is the attribute R, S have in common

For each tuple \( t_i \) of R

If \( t_i.A = K \) then use the index to compute \( \sigma_{att = K}(S) \)

Demands: index on A for S
Indexed Nested-Loop Join

Indexed Nested Loop Join

- For each tuple $t_R$ in the outer relation $R$, use the index to look up tuples in $S$ that satisfy the join condition with tuple $t_R$.

- Worst case: buffer has space for only one page of $R$, and, for each tuple in $R$, we perform an index lookup on $S$.

- Cost of the join: $b_r + n_r \times c$
  - Where $c$ is the cost of traversing index and fetching all matching $s$ tuples for one tuple of $R$.
  - $c$ can be estimated as cost of a single selection on $S$ using the join condition.

- If indices are available on join attributes of both $R$ and $S$, use the relation with fewer tuples as the outer relation.
Example of Nested-Loop Join Costs

Query: \textit{depositor} \Join \textit{customer}

\[(\text{cname, acct\_no}) \quad (\text{cname, ccity, cstreet})\]

Metadata:

\[
\begin{align*}
\text{customer:} & \quad n_{\text{customer}} &= 10,000 \\
& \quad f_{\text{customer}} &= 25 \\
& \quad b_{\text{customer}} &= 400 \\
\text{depositor:} & \quad n_{\text{depositor}} &= 5000 \\
& \quad f_{\text{depositor}} &= 50 \\
& \quad b_{\text{depositor}} &= 100 \\
\text{V (cname, depositor)} &= 2500
\end{align*}
\]

\(i\) is a primary index on \text{cname} (dense) for \text{customer}

\text{Fanout} for \(i\), \(f_i = 20\)
Plan generation for Joins

Alternative 1: Block Nested Loop

1a: customer  = OUTER relation
   depositor = INNER relation
   cost: $b_{\text{customer}} + b_{\text{customer}} * b_{\text{depositor}} = 400 + (100 * 400 ) = 40,400$

1b: customer  = INNER relation
   depositor = OUTER relation
   cost: $b_{\text{depositor}} + b_{\text{depositor}} * b_{\text{customer}} = 100 + (400 *100) = 40,100$
Plan generation for Joins

Alternative 2: Indexed Nested Loop

We have an **index on **c**name for customer.**
Depositor is the outer relation

Cost:

\[ b_{\text{depositor}} + n_{\text{depositor}} \cdot c = 100 + (5000 \cdot c) \]

\( c \) is the cost of evaluating a selection \( c\text{name} = K \) using the index.

Primary index on \( c\text{name}, \) \( c\text{name} \) a key for customer

\[ c = HT_i + 1 \]
What is $HT_i$?

cname a key for customer. $V(\text{cname, customer}) = 10,000$

$f_i = 20$, $i$ is dense

$LB_i = \lceil \frac{10,000}{20} \rceil = 500$

$$HT_i \sim \lceil \log_{f_i}(LB_i) \rceil + 1 = \lceil \log_{20} 500 \rceil + 1 = 4$$

Cost of index nested loop is:

$= 100 + (5000 \times (4)) = 20,100$ Block accesses (cheaper than NLJ)
Another Join Strategy

Query: \( R \bowtie S \)

Algorithm 4: Merge Join

Idea: suppose \( R, S \) are both sorted on \( A \) (\( A \) is the common attribute)

```latex
\begin{array}{c|c|c|c}
 & A & & A \\
1 & \multicolumn{1}{l}{p_R} & & \multicolumn{1}{l|}{p_S} \\
2 & & & 2 \\
3 & & & 2 \\
4 & & & 3 \\
\ldots & & & \ldots \\
\end{array}
```

Compare:
- (1, 2) \( \rightarrow \) advance \( p_R \)
- (2, 2) \( \rightarrow \) match, advance \( p_S \) \( \rightarrow \) add to result
- (2, 2) \( \rightarrow \) match, advance \( p_S \) \( \rightarrow \) add to result
- (2, 3) \( \rightarrow \) advance \( p_R \)
- (3, 3) \( \rightarrow \) match, advance \( p_S \) \( \rightarrow \) add to result
- (3, 5) \( \rightarrow \) advance \( p_R \)
- (4, 5) \( \rightarrow \) read next block of \( R \)
Merge-Join

GIVEN R, S both sorted on A

1. Initialization
   - Reserve blocks of R, S in buffer reserving one block for result
   - Pr = 1, Ps = 1

2. Join (assuming no duplicate values on A in R)
   WHILE !EOF(R) && !EOF(S) DO
      if $B_R[Pr].A == B_S[Ps].A$ then
         output to result; Ps++
      else if $B_R[Pr].A < B_S[Ps].A$ then
         Pr++
      else (same for Ps)
      if Pr or Ps point past end of block,
         read next block and set Pr(Ps) to 1
Cost of Merge-Join

- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus number of block accesses for merge-join is 
  \[ b_R + b_S \]
- But....
  What if one/both of R,S not sorted on A?

Ans: May be worth sorting first and then perform merge join
  (called Sort-Merge Join)

Cost: \[ b_R + b_S + \text{sort}_R + \text{sort}_S \]
External Sorting

Not the same as internal sorting

Internal sorting:

- minimize CPU (count comparisons)
- best: quicksort, mergesort, ....

External sorting:

- minimize disk accesses (what we’re sorting doesn’t fit in memory!)
- best: external merge sort

WHEN used?

1) SORT-MERGE join
2) ORDER BY queries
3) SELECT DISTINCT (duplicate elimination)
**External Sorting**

Idea:

1. Sort fragments of file in memory using internal sort (runs). Store runs on disk. (run size = 3; =block size)

2. Merge runs. E.g.:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
External Sorting (cont.)

Algorithm
Let M = size of buffer (in blocks)

1. Sort runs of size M blocks each (except for last) and store.
   Use internal sort on each run.

2. Merge M-1 runs at a time into 1 and store as a new run. Merge for all runs. (1 block per run + 1 block for output)

3. If step 2 results in more than 1 run, go to step 2.
External Sorting (cont.)

Cost: \(2 b_R \times (\lceil \log_{M-1}(b_R / M) \rceil + 1)\)

Step 1: Create runs
- every block read and written once
- cost = 2 \(b_R\) I/Os

Step 2: Merge
- every merge iteration requires reading and writing entire file (\(= 2 b_R\) I/Os)
- every iteration reduces the number of runs by factor of M-1

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Runs Left to Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\frac{b_R}{M})</td>
</tr>
<tr>
<td>2</td>
<td>(\frac{b_R}{M} \times \frac{1}{M - 1})</td>
</tr>
<tr>
<td>3</td>
<td>(\frac{b_R}{M} \times \frac{1}{M - 1} \times \frac{1}{M - 1})</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Number of iterations

\(# \text{merge passes} = \left\lceil \log_{M-1}(b_R / M) \right\rceil\)

Initial number of runs
What if we need to sort?

Query: \textit{depositor} \bowtie \textit{customer}

\begin{align*}
\text{Sort } \textit{depositor} &= 2 \times 100 \times \left(\lceil \log_2(100 / 3) \rceil \right) + 1 \\
&= 1400 \\
\text{Same for } \textit{customer}. &= 7200 \\
\textbf{Total: }&= 100 + 400 + 1400 + 7200 = 9100 \text{ I/O’s!}
\end{align*}

Still beats BNLJ (40K), INLJ (20K) 
Why not use SMJ always?

Ans: 1) Sometimes inner relation can fit in memory 
2) Sometimes index is small 
3) SMJ only work for natural joins, “equijoins”
Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function $h$ is used to partition tuples of both relations
- $h$ maps $JoinAttr$ values to $\{0, 1, \ldots, n\}$, where $JoinAttr$ denotes the common attributes of $r$ and $s$ used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of tuples of $r$.
    - Each tuple $t_r \in r$ is put in partition $r_i$ where $i = h(t_r[JoinAttr])$.
  - $r_0, r_1, \ldots, r_n$ denote partitions of tuples of $s$.
    - Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[JoinAttr])$. 
Hash-Join (Cont.)

partitions of $r$  partitions of $s$
Hash-Join (Cont.)

- $r$ tuples in $r_i$ need only to be compared with $s$ tuples in $s_i$.
  Need not be compared with $s$ tuples in any other partition, since:

  ★ an $r$ tuple and an $s$ tuple that satisfy the join condition
  will have the same value for the join attributes.

  ★ If that value is hashed to some value $i$, the $r$ tuple has to be in $r_i$ and the $s$ tuple in $s_i$. 
Hash-Join Algorithm

The hash-join of \( r \) and \( s \) is computed as follows.

1. Partition the relation \( s \) using hashing function \( h \). When partitioning a relation, one block of memory is reserved as the output buffer for each partition.

2. Partition \( r \) similarly.

3. For each \( i \):
   
   (a) Load \( s_i \) into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one \( h \).

   (b) Read the tuples in \( r_i \) from the disk one by one. For each tuple \( t_r \), locate each matching tuple \( t_s \) in \( s_i \) using the in-memory hash index. Output the concatenation of their attributes.

Relation \( s \) is called the **build input** and \( r \) is called the **probe input**.
Hash-Join algorithm (Cont.)

- The value $n$ and the hash function $h$ is chosen such that each $s_i$ should fit in memory.
  - Typically $n$ is chosen as $\lceil b_s / M \rceil * f$ where $f$ is a “fudge factor”, typically around 1.2
  - The probe relation partitions $s_i$ need not fit in memory

- Cost of hash join is
  \[
  3(b_r + b_s) + 4 * n_h \text{ block transfers}
  \]

- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$. 