Selection Operation

- Selections involving ranges
  - `select * from accounts where balance > 100000`
  - `select * from matches where matchdate between '10/20/06' and '10/30/06'`
- Option 1: Sequential scan
- Option 2: Using an appropriate index
  - Can’t use hash indexes for this purpose

Selection w/ B+-Tree Indexes

<table>
<thead>
<tr>
<th>why?</th>
<th>cost of finding the first leaf</th>
<th>cost of retrieving the tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary index, candidate key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$1 \times (t_T + t_S)$</td>
</tr>
<tr>
<td>primary index, not a key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$t_S + b \times t_T$ Note: primary == sorted $b = number of pages that contain the matches$</td>
</tr>
<tr>
<td>secondary index, candidate key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$1 \times (t_T + t_S)$</td>
</tr>
<tr>
<td>secondary index, not a key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$n \times (t_T + t_S)$ $n = number of records that match$ This can be bad</td>
</tr>
</tbody>
</table>

$h_i = height of the index$ root assumed in-memory
Selection Operation

- Complex selections
  - **Conjunctive:** `select * from accounts where balance > 100000 and SSN = “123”`
  - **Disjunctive:** `select * from accounts where balance > 100000 or SSN = “123”`

- Option 1: Sequential scan

- Option 2 (Conjunctive only): Using an appropriate index on one of the conditions
  - E.g. Use SSN index to evaluate SSN = “123”. Apply the second condition to the tuples that match
  - Or do the other way around (if index on balance exists)
  - Which is better?

- Option 3 (Conjunctive only): Choose a multi-key index
  - Not commonly available

- Option 4: Conjunction or disjunction of record identifiers
  - Use indexes to find all RIDs that match each of the conditions
  - Do an *intersection* (for conjunction) or a *union* (for disjunction)
  - Sort the records and fetch them in one shot
  - Called “Index-ANDing” or “Index-ORing”
  - Heavily used in commercial systems
Sorting

- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of B+-tree
    - May be enough in many cases
  - But if relation not sorted, too many random accesses
- If relation small enough…
  - Read in memory, use quicksort (qsort() in C)
- What if relation too large to fit in memory?
  - External sort-merge

External sort-merge

- Divide and Conquer !!
- Let $M$ denote the memory size (in blocks)
- Phase 1:
  - Read first $M$ blocks of relation, sort, and write it to disk
  - Read the next $M$ blocks, sort, and write to disk …
  - Say we have to do this “$N$” times
  - Result: $N$ sorted runs of size $M$ blocks each
- Phase 2:
  - Merge the $N$ runs ($N$-way merge)
  - Can do it in one shot if $N < M$
External sort-merge

- **Phase 1:**
  - Create sorted runs of size $M$ each
  - Result: $N$ sorted runs of size $M$ blocks each

- **Phase 2:**
  - Merge the $N$ runs ($N$-way merge)
  - Can do it in one shot if $N < M$

- **What if $N > M$?**
  - Do it recursively
  - If $M = 1000$, can compare 1000 runs
    - (4KB blocks): can sort: 1000 runs, each of 1000 blocks, each of 4k bytes = 4GB of data

---

**Example: External Sorting Using Sort-Merge ($N >= M$)**

```
 initial relation
 a 14
 b 14
 c 33
 d 31
d 7
 m 3
 p 2
 r 16

 runs

 a 14
 b 14
 c 33
d 31
d 7
 e 16
 m 3
 p 2
 r 16

c 33
 d 31
d 21
d 21
m 3
 p 2
 r 16

c 33
d 31
d 21
e 16
g 24

merge pass-1

 sorted output
 a 19
d 31
b 14
c 33
d 31
e 16
g 24

merge pass-2

 M = 3
 N = 12

blocksize == tuplesize
for this example, not in general
```
External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required: \(\lceil \log_{M-1}(b_r/M) \rceil\).
  - Disk for initial run creation as well as in each pass is \(2b_r\).
    - for final pass, we don’t count write cost because:
      - \textit{output may be pipelined} (sent via memory to parent operation).

Thus total number of disk transfers for external sorting:
\[b_r \cdot (2 \lceil \log_{M-1}(b_r/M) \rceil + 1)\]

Seeks:
\[2 \lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2 \lceil \log_{M-1}(b_r/M) \rceil - 1)\]

\(b_b\) is:
- #blocks read at a time, and how many output blocks needed
- Assumed to be “1” unless otherwise specified

Example: External Sorting Using Sort-Merge (\(N \geq M\))

\[
\begin{align*}
\text{block transfers:} & & b_r \cdot (2 \lceil \log_{M-1}(b_r/M) \rceil + 1) = 60 \\
\text{seeks:} & & 2 \lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2 \lceil \log_{M-1}(b_r/M) \rceil - 1) = 44 \\
\end{align*}
\]

\(b_b\) for reading blocks at a time
Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Putting it all together…

Join

- `select * from R, S where R.a = S.a`
  - Called an “equi-join”
- `select * from R, S where |R.a – S.a| < 0.5`
  - Not an “equi-join”

- Option 1: Nested-loops
  
  for each tuple \( r \) in \( R \)
  
  for each tuple \( s \) in \( S \)
  
  check if \( r.a = s.a \) (or whether \( |r.a - s.a| < 0.5 \))

- Can be used for any join condition
  - As opposed to some algorithms we will see later
- \( R \) called outer relation
- \( S \) called inner relation
Nested-loops Join

- **Cost**? Depends on the actual values of parameters, especially memory
- \( b_r, b_s \rightarrow \text{Number of blocks of } R \text{ and } S \)
- \( n_r, n_s \rightarrow \text{Number of tuples of } R \text{ and } S \)
- **Case 1:** Minimum memory required = 3 blocks
  - One to hold the current \( R \) block, one for current \( S \) block, one for the result being produced
  - Blocks transferred:
    - Must scan \( R \) tuples once: \( b_r \)
    - For each \( R \) tuple, must scan \( S \): \( n_r \times b_s \)
  - Seeks ?
    - \( n_r + b_r \)

Example:
- Number of records -- \( R: n_r = 10,000, S: n_s = 5000 \)
  \[ t_R = 0.1\text{ms} \]
- Number of blocks -- \( R: b_r = 400, S: b_s = 100 \)
  \[ t_S = 4\text{ms} \]

**R** "outer relation":
- blocks transferred: \( n_r \times b_s + b_r = 10000 \times 100 + 400 = 1,000,400 \)
- seeks: 10400
- time: \( 1000400 t_r + 10400 t_s = 1000400(0.1\text{ms}) + 10400(4\text{ms}) = 141.64 \text{ sec} \)

**S** outer relation?
- \( 5000 \times 400 + 100 = 2,000,100 \) block transfers,
- 5100 seeks
- \( = 2000100 t_r + 5100 t_s = 220.41 \text{ sec} \)

*Order matters!*
Nested-loops Join

- **Case 2: S fits in memory**
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
- **Example**:
  - Number of records -- $R$: $n_r = 10,000$, $S$: $n_s = 5000$
  - Number of blocks -- $R$: $b_r = 400$, $S$: $b_s = 100$
- **Then**:
  - blocks transferred: $400 + 100 = 500$
  - seeks: 2
  - $= 500t_r + 2t_s = 0.058$ sec

*Orders of magnitude difference*

---

Block Nested-loops Join

- **Simple modification to “nested-loops join”** (block at a time)
  - for each block $B_r$ in $R$
    - for each block $B_s$ in $S$
      - for each tuple $r$ in $B_r$
        - for each tuple $s$ in $B_s$
          - check if $r.a = s.a$ (or whether $|r.a - s.a| < 0.5$)
- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: $b_r \times b_s + b_r$
  - Seeks: $2 \times b_r$
- **For the example**:
  - blocks: 40400, seeks: $800 = 4.04 + 3.2 = 7.24$ sec
  - blocks: 40100, seeks: $200 = 4.01 + 0.8 = 4.81$ sec (S outer)
Block Nested-loops Join

- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: $b_r \times b_s + b_r$
  - Seeks: $2 \times b_r$

- **Case 2: S fits in memory**
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
  - Cost: $(500)t_r + 2t_s = 58$ msec

- **What about in between?**
  - Say there are 50 blocks, but $S$ is 100 blocks
  - Why not use all the memory that we can…

- **Case 3: 50 blocks ($S = 100$ blocks)**
  - for each group of 48 blocks in $R$
    - for each block $B_s$ in $S$
      - for each tuple $r$ in the group of 48 blocks
        - for each tuple $s$ in $B_s$
          - check if $r.a = s.a$ (or whether $|r.a - s.a| < 0.5$)

- **Why is this good?**
  - We only have to read $S$ a total of $b_r/48$ times (instead of $b_r$ times)
  - Blocks transferred: $b_s \times b_r/48 + b_r = 100 \times 400/48 + 400 = 1233$
    - Or $b_s \times b_r/48 + b_r = 400 \times 100/48 + 100 = 933$ (but more seeks)
  - Seeks: $2 \times b_r/48$
  - Cost: $1233t_r + \left[800/48\right] t_s = 123 + 17 \times 4 = 0.191$ sec
**Index Nested-loops Join**

- **select** * from R, S where R.a = S.a
  - “equi-join”
  - Nested-loops
    
    for each tuple r in R
    for each tuple s in S
    check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Suppose there is an index on S.a
- Why not use the index instead of the inner loop?
  
  for each tuple r in R
  use the index to find S tuples with S.a = r.a

**Index Nested-loops Join**

- **select** * from R, S where R.a = S.a
  - Called an “equi-join”
  - Why not use the index instead of the inner loop?
    
    for each tuple r in R
    use the index to find S tuples with S.a = r.a

- **Cost of the join:**
  - \( b_r + n_r \cdot c_s \) (seeks and block transfers)
  - \( c_s \) is cost of index access for S
    
    - Computed using the formulas discussed earlier
Index Nested-loops Join

- W/ indexes for both \( R, S \), use one w/ fewer tuples as outer.
- Recall example:
  - Number of records -- \( R: n_r = 10,000, S: n_s = 5000 \)
  - Number of blocks -- \( R: b_r = 400, S: b_s = 100 \)
- Assume \( B^+ \)-tree for \( R \), avg fanout of 20, implies height \( R \) is 4
  - Cost is 100 + 5000 * (4 + 1) = 25,100, each w/ seek and transfer (102.9 sec)
- Assume \( B^+ \)-tree is on \( S \): height = 3
  - Cost is 400 + 10000 * (3+1) = 40,400, each w/ seek and transfer (165.6 sec)

\[ b_{out} + n_{out} \times c_{in} \]

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about \( |R.a - S.a| < 5 \) \( \text{nope} \)
- Great for queries with joins and selections
  
  \[
  \text{SELECT *}
  \]
  
  \[
  \text{FROM accounts, customers}
  \]
  \[
  \text{WHERE accounts.customer-SSN = customers.customer-SSN AND}
  \]
  \[
  \text{accounts.acct-number = "A-101"}
  \]
  
  - Use \textit{accounts} as outer, use select to prune reads of customers
So far…

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - \( b_r + b_s \)
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
  - Typically used when the smaller relation is really small (few tuples) and index nested-loops can’t be used

- **Index Nested-loops join**
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return *small number of tuples*
    - `select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN`

---

**Recall:** External Sorting Using Sort-Merge (\( N \geq M \))

\[
\begin{align*}
b_r(2 \lfloor \log_{b_r}(M/b_s) \rfloor + 1) \text{ blocks} \\
\text{seeks:} 2 \lfloor b_r/M \rfloor + \lfloor b_r/b_s \rfloor(2 \lfloor \log_{b_r}(M/b_s) \rfloor - 1) \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Initial relation</th>
<th>Runs</th>
<th>Merge pass-1</th>
<th>Merge pass-2</th>
<th>Sorted output</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial relation</td>
<td>runs</td>
<td>merge pass-1</td>
<td>merge pass-2</td>
<td>sorted output</td>
</tr>
<tr>
<td>g 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c 33</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = 3  
N = 12
Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called “sort-merge join” sometimes

```
SELECT *
FROM r, s
WHERE r.a1 = s.a1
```

Step:
1. Compare the tuples at pr and ps
2. Move pointers down the list
   - Depending on the join condition
3. Repeat

<table>
<thead>
<tr>
<th>pr</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ps</th>
<th>a1</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on $a1$
  - This might make further operations easier to do
    - E.g. duplicate elimination
So far…

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition

- **Index Nested-loops join**
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - `select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN`

- **Merge joins**
  - Join algorithm of choice when the relations are large
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

Hash Join

- **Case 1: Smaller relation (S) fits in memory**

- Nested-loops join:
  
  `for each tuple r in R`
  
  `for each tuple s in S`
  
  `check if r.a = s.a`

  - Cost: $b_r + b_s$ transfers, 2 seeks
  - The inner loop is not exactly cheap (high CPU cost)

- Hash join:
  
  `read S in memory and build a hash index on it`
  
  `for each tuple r in R`
  
  `use the hash index on S to find tuples such that S.a = r.a`
Hash Join

- **Case 1:** Smaller relation \((S)\) fits in memory
  - Hash join:
    - *read S in memory and build a hash index on it*
    - *for each tuple r in R*
    - *use the hash index on S to find tuples such that S.a = r.a*
  - **Cost:** \(b_r + b_s\) transfers, 2 seeks (unchanged)
  - **Why good?**
    - CPU cost is much better (even though we don’t care about it too much)
    - Much better than nested-loops join when \(S\) doesn’t fit in memory (next)

Hash Join

- **Case 2:** Smaller relation \((S)\) doesn’t fit in memory
  - **Basic idea:**
    - partition tuples of each relation into sets that have same value on join attributes
    - must be equi-/natural join
  - **Phase 1:**
    - Read \(R\) block by block and partition it using a hash function: \(h1(a)\)
      - Create one partition for each possible value of \(h1(a)\) \((n_h\) partitions\)
    - Write the partitions to disk
      - \(R\) gets partitioned into \(R_1, R_2, \ldots, R_k\)
    - Similarly, read and partition \(S\), and write partitions \(S_1, S_2, \ldots, S_k\) to disk
    - **Only requirements:**
      - Room for a single input block and one output block for each hash value
      - Each \(S\) partition fits in memory
Hash Join

- Case 2: Smaller relation \((S)\) doesn’t fit in memory
- Two “phases”
- Phase 2:
  - Read \(S_i\) into memory, and build a hash index on it \((S_i\) fits in memory\)
    - Use a different hash function from the partition hash: \(h_2(a)\)
  - Read \(R_i\) block by block, and use the hash index to find matches.
  - Repeat for all \(i\).
Hash Join

- **Case 2: Smaller relation (S) doesn’t fit in memory**
- Two “phases”:
  - Phase 1:
    - Partition the relations using one hash function, $h_1(a)$
  - Phase 2:
    - Read $S_i$ into memory, and build a hash index on it ($S_i$ fits in memory)
    - Read $R_i$ block by block, and use the hash index to find matches.
- **Cost**?
  - $3(b_r + b_s)$ block transfers
    - $R$ or $S$ might have partially full block to be read and written (ignored)
  - $+ 2\left(\left\lfloor \frac{b_r}{b_b} \right\rfloor + \left\lfloor \frac{b_s}{b_b} \right\rfloor\right)$ seeks (seek count unclear)
    - Where $b_b$ is the size of each input buffer (p 560)
  - Much better than Nested-loops join under the same conditions