Query Processing

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

Problems w/ b-trees

- \( n_R = 10,000, b_R = 1000 \), primary (clustered), index on non-candidate key A
- \( n_S = 1,000, b_S = 500 \), secondary (non-clustered) index on non-candidate key A.
- \( t_T = 0.1 \text{ msec}, t_S = 4 \text{ msec} \)
- Fanout, \( D \), on both is 50.

- Let’s first assume that \( b = N= 50 \); i.e. we are looking for max capacity
- What is the height of \( R’s \ tree ? \)
  - root has \( b \) ptrs
  - \( h=1 \) has \( b \) leaves, each of which have \( b \) pointers = \( b^2 = 2500 \)
  - \( h=2 \) means \( b^3 = 125,000 \) pointers, so:
    \[ h_R = 2 \]
  - More generally, tree of fanout \( b \) and height \( h \) has capacity of \( b^{h+1} \)
    - …except each leaf devotes one ptr to point to next leaf!
    - so really:
      \[ b^{h+1} - b^h \]

Fanout is always between half of \( N \) and \( N \). \( b \) is average # ptrs used at each level, usually less than \( N \).
Problems w/ b-trees

- \( n_R = 10,000, b_R = 1000 \), primary (clustered), index on non-candidate key A
- \( n_S = 1,000, b_S = 500 \), secondary (non-clustered) index on non-candidate key A.
- \( t_T = 0.1 \) msec, \( t_S = 4 \) msec
- Fanout, \( N \), on both is 50.

• Cost to return first tuple for \( R.A = 42 \)?
  - primary, not a candidate key
  - \((h_R + 1) \times (t_T + t_S) = 3\times(0.1 + 4.0) = 12.3 \) msec

• Cost to return the rest, assuming blocking factor is 10, and 100 total matches?
  - \((b-1) \times t_T = 9 \times 0.1 = 0.9 \) msecs

\( b \) is average # ptrs used at each level, usually less than \( N \)

Problems w/ b-trees

- \( n_R = 10,000, b_R = 1000 \), primary (clustered), index on non-candidate key A
- \( n_S = 1,000, b_S = 500 \), secondary (non-clustered) index on non-candidate key A.
- \( t_T = 0.1 \) msec, \( t_S = 4 \) msec
- Fanout, \( N \), on both is 50.

• What is the height of \( S's \) tree?
  - \( h_S = 1 \) gives room for 2500, \( n_s \) is only 1000, so: \( h_S = 1 \)

• Cost to return first tuple for \( S.A = 42 \)?
  - secondary, not a candidate key
  - \((h_S + 1) \times (t_T + t_S) = 2\times(0.1 + 4.0) = 8.2 \) msec

• Cost to return the rest, assuming blocking factor is 10, and 100 total matches?
  - \((b-1) \times t_T = 99 \times (0.1 + 4.0) = 200.9 \) msecs

Blocking factor is irrelevant because matches are randomly scattered.
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
- $b_r, b_s \rightarrow$ Number of blocks of R and S
- $n_r, n_s \rightarrow$ Number of tuples of R 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: $b_r$ blocks
    - For each R tuple, must scan S: $n_r \cdot b_s$
    - $b_r + n_r \cdot b_s$
  - Seeks?
    - $n_r + b_r$
Nested-loops Join

- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: \( \pi_r \ast b_s + b_r \)
  - Seeks: \( \pi_r + b_r \)

- **Example**:
  - Number of records -- \( R: n_r = 10,000, S: n_s = 5000 \)
  - Number of blocks -- \( R: b_r = 400, S: b_s = 100 \)

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

- **S outer relation**:
  - \( 5000 \ast 400 + 100 = 2,000,100 \) 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
  - \( = 500 t_r + 2 t_s = 0.058\text{ sec} \)

*Four orders of magnitude difference*
Block Nested-loops Join

- Simple modification to “nested-loops join”
  
  \[
  \text{for each block } B_r \text{ in } R \\
  \text{for each block } B_s \text{ in } S \\
  \text{for each tuple } r \text{ in } B_r \\
  \quad \text{for each tuple } s \text{ in } B_s \\
  \quad \text{check if } r.a = s.a \text{ (or whether } |r.a - s.a| < 0.5) \\
  \]

- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: \( b_r \ast b_s + b_r \)
  - Seeks: \( 2 \ast b_r \)
  
  **For the example:**
  - blocks: 40400, seeks: 800 = 4.04 + 3.2 = 7.24 sec

- **Case 2: S fits in memory**
  - Blocks transferred: \( b_s + b_r \)
  - Seeks: 2

- What about in between?
  - Say there are 50 blocks, but \( S \) is 100 blocks
  - Why not use all the memory that we can…
Block Nested-loops Join

- **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 ceiling($b_r / 48$) times (instead of $b_r$ times)
  - Blocks transferred:
    - $r \lfloor b_r / 48 \rfloor b_s + b_r = \left\lfloor \frac{400}{48} \right\rfloor *100 + 400 = 1300$
  - Seeks:
    - $2 * \left\lfloor \frac{b_r}{48} \right\rfloor = 18$
  - cost is 1300(0.1) + 18(4) = 206 msecs

- **Use S as the outer relation:**
  - Blocks transferred:
    - $r \lfloor b_r / 48 \rfloor b_s + b_r = \left\lfloor \frac{100}{48} \right\rfloor *400 + 100 = 1300$
  - Seeks:
    - $2 * \left\lfloor \frac{b_r}{48} \right\rfloor = 6$
  - cost is 1300(0.1) + 6(4) = 154 msecs

- **48 blocks for R**
- **1 block for S**
- **1 block for output**