Query Processing

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

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: $b_r \cdot b_s + b_r$
  
  Seeks: $2 \cdot b_r$

- **For the example:**
  
  blocks: 40400, seeks: 800
  
  7.24 seconds

Number of records -- $R$: $n_r = 10,000$, $S$: $n_s = 5000$

Number of blocks -- $R$: $b_r = 400$, $S$: $b_s = 100$
Block Nested-loops Join

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

- **Case 2: S fits in memory**
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
  - 0.058 seconds

- **What about in between?**
  - Say there are 50 blocks mem avail, 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: $2 \times b_r/48$
  - 0.190 seconds
Index Nested-loops Join

- \textit{select * from }R, S\textit{ where }R.a = S.a
  - “equi-join”
- Nested-loops
  \begin{align*}
  \text{for each tuple } r & \text{ in } R \\
  \text{for each tuple } s & \text{ in } S \\
  \text{check if } r.a & = s.a \text{ (or whether } |r.a - s.a| < 0.5) \\
  \end{align*}
- Suppose there is an index on \( S.a \)
- \textit{Why not use the index instead of the inner loop?}
  \begin{align*}
  \text{for each tuple } r & \text{ in } R \\
  \text{use the index to find } S \text{ tuples with } S.a = r.a \\
  \end{align*}

Index Nested-loops Join

- \textit{select * from }R, S\textit{ where }R.a = S.a
  - Called an “equi-join”
- \textit{Why not use the index instead of the inner loop?}
  \begin{align*}
  \text{for each tuple } r & \text{ in } R \\
  \text{use the index to find } S \text{ tuples with } S.a = r.a \\
  \end{align*}
- \textbf{Cost of the join:}
  - \( b_r (t_r + t_s) + n_r \ast c \)
  - \( c = \text{ the cost of index access} \)
    - \textit{Computed using the formulas discussed earlier}
Index Nested-loops Join

- With 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 $S$ outer, $B^+$-tree on $R$ is height 4
  - Blocks transferred: $100 + 5000 \times (4 + 1) = 25,100$, each w/ seek and transfer
  - 102.9 seconds
- Assume $R$ outer, $B^+$-tree on $S$ is height 3
  - Blocks transferred: $400 + 10000 \times (3 + 1) = 40,400$, each w/ seek and transfer
  - 165.6 seconds

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about $|R.a - S.a| < 5$?
- Great for queries with joins and selections
  
  ```sql
  SELECT *
  FROM accounts, customers
  WHERE accounts.customer-SSN = customers.customer-SSN AND
  accounts.acct-number = "A-101"
  ```
  - Use `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:
    - \( b_i + b_s \)
    - 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`

---

**Query Processing**

- **Overview**
- **Selection operation**
- **Join operators**
- **Sorting**
- **Other operators**
- **Putting it all together…**
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
  - Not expected to happen
  - 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)**

- **M = 3**
- **N = 12**

Initial relation
- g 24
d 31
c 33
b 14
e 16
r 16
d 21
m 3
p 2
da 14

Runs
- a 19
d 31
g 24
b 14
c 33
e 16
d 31
c 33
e 16
g 24

Merge pass-1
- a 19
b 14
c 33
d 31
e 16

Sorted output
- d 21
m 3
r 16
da 14
d 21
m 3
p 2

Merge pass-2
- d 21
m 3
r 16
da 14
d 21
m 3
p 2
r 16
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
      - output may be pipelined (sent via memory to parent operation)

Thus total number of disk transfers for external sorting:

$$2b_r \lceil \log_{M-1}(b_r/M) \rceil - b_r + 2b_r = b_r \left( 2 \lceil \log_{M-1}(b_r/M) \rceil + 1 \right)$$

$$= 12 \times (2^2 + 1) = 60$$

Seeks:

$$2 \lceil b_r/M \rceil + 2 b_r \lceil \log_{M-1}(b_r/M) \rceil - b_r = 2 \lceil b_r/M \rceil +$$

$$b_r \left( 2 \lceil \log_{M-1}(b_r/M) \rceil - 1 \right)$$

$$= 8 + 12(2^2-1) = 44$$

#blocks read at a time, $b_{br}$ ignored here

External Merge Sort (Cont.)

- What if $M = 4$?
  - merge in one round (3 runs, 1 buffer for output).
  - disk transfers: $= b_r \left( 2 \lceil \log_{M-1}(b_r/M) \rceil + 1 \right) = 12 \times (2^1 + 1) = 36$
  - seeks: $= 2 \lceil b_r/M \rceil + b_r \left( 2 \lceil \log_{M-1}(b_r/M) \rceil - 1 \right) = 6 + 12(2-1) = 18$
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:
  
  ```plaintext
  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
  - Inner loop not cheap (for each tuple in R go through all S)

- Hash join:
  
  ```plaintext
  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, plus 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, build hash index ($S_i$ fits in memory)
  - Using 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 \(S\) into memory, and build a hash index on it (\(S\) fits in memory)
    - Read \(R\) 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)
    - \(b_b\) is size of each input buffer (assume \(b_b\) is 1 by default, but p 560)
    - Much better than Nested-loops join under the same conditions

Hash Join: Issues

- **How to guarantee that each partition of \(S\) fits in memory?**
  - Say \(S = 10,000\) blocks, Memory = \(M = 100\) blocks
  - Use a hash function that hashes to 100 different values?
    - Eg. \(h_n(a) = a \mod 100\)?
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to \(100^f\) different values
    - \(f\) is called fudge factor, typically around 1.2
    - So we may consider \(h_1(a) = a \mod 120\).
    - This is okay IF \(a\) is uniformly distributed

- **Why can’t we just set \(h_n\) to 200?**
  - need to have a per-value output block in mem during build phase
Hash Join: Issues

- Memory required?
  - Say $S = 10000$ blocks, $Memory = M = 100$ blocks
  - So 120 different partitions
  - During phase 1:
    - Need 1 block for reading $R$
    - Need 120 blocks for storing each partition of $R$
  - So must have at least 121 blocks of memory
  - We only have 100 blocks
- Typically need $\text{SORT}(|S| \times f)$ blocks of memory
  - So if $S$ is 10000 blocks, and $f = 1.2$, need 110 blocks of memory
  - Need:
    - $M > n_h + 1$
    - each partition of $S$ to fit in $M-1$ (why not $R$?)
    - space for hash build on $h_2()$ (usually ignored)
  - Example:
    - $h_2 = 109$, average size $= 10,000 / 109 = 91.7$

Hash Join: If $S_i$ Too Large

- Avoidance
  - Fudge factor

- Resolution
  - partition w/ a third hash $h_3()$
  - also partition $R_i$
  - go through each sub-partition
  - this approach could be used for every partition