CptS 415 Big Data

Query Language

Srini Badri

Acknowledgement: Tinghui Wang

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimation
- Examples use the following information
  - Number of records: student (5,000), takes (10,000)
  - Number of blocks of students: student (100), takes (400)
  - Student (ID, First Name, Last Name, Degree)
  - Takes (Course ID, Student ID)

Nested Loop Join

- To compute the theta join: $R \bowtie_\theta S$
- Intuition: $R \bowtie_\theta S = \sigma_\theta(R \times S)$

  for each tuple t_R in R
    for each tuple t_S in S
      test if pair (t_R, t_S) satisfy the join condition θ
    end for
  end for

- R is called the outer relation and S the inner relation of the join
- Requires no indices and can be used with any kind of join condition
- Expensive, since it examines every pair of tuples in the two relation $O(mn)$

Nested-Loop Join (Cont'd)

```
for each tuple t_R in R
  for each tuple t_S in S
    test if pair (t_R, t_S) satisfy the join condition θ
  end for
end for
```

- Assuming we are "reading" one tuple at a time, even though the tuples are organized in the blocks:
  - Number of disk IO operations: n_R * n_S
  - n_R - number of tuples in R, n_S - number of tuples in S
- This is the worst case scenario and does not take into account that multiple tuples may be present in each block of storage

Block Nested Loop Join

- Variant of nested loop join in which every block of inner relation is paired with every block of outer relation

```
for each block $B_R$ of $R$
  for each block $B_S$ of $S$
    for each tuple $t_R$ in $R$
      for each tuple $t_S$ in $S$
        test if pair $(t_R,\ t_S)$ satisfy the join condition $\theta$
      end for
    end for
  end for
end for
```

Block Nested Loop Join (cont.)

- Outer loop start:
  - seek 1 block of R, and transfer the block of R to memory <- repeats b_R times
    - Inner loop start:
      - seek 1 block of S, and transfer the block of S to memory <- repeats b_S times
      - read 1 tuple t_R from the block of R, and 1 tuple t_S from the block of S
      - perform join operation on tuple t_R and tuple t_S
    - Inner loop end
- Outer loop end

Worst Case:
- Total transfers: $(b_R + b_R * b_S)$
- Total seeks: $b_R + b_R*1 = 2b_R$

Best Case:
- Total transfers: $b_R + b_S$
- Total seeks: $1 + 1 = 2$

Merge Join (Sort Merge Join)

- Sort both relations on their join attribute (if not already sorted on the join attributes).
- Merge the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm.
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched

pr →

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

ps →

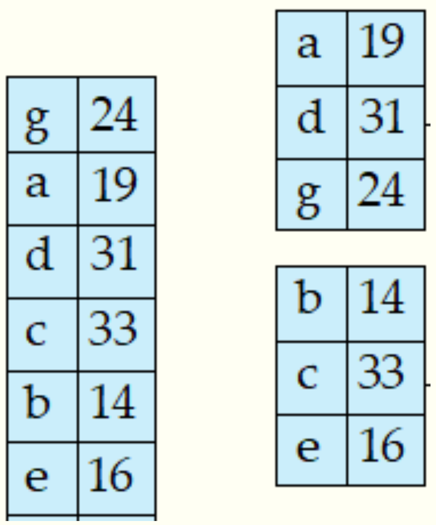
a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

External Merge Sort

Let M denote available memory size (in blocks).

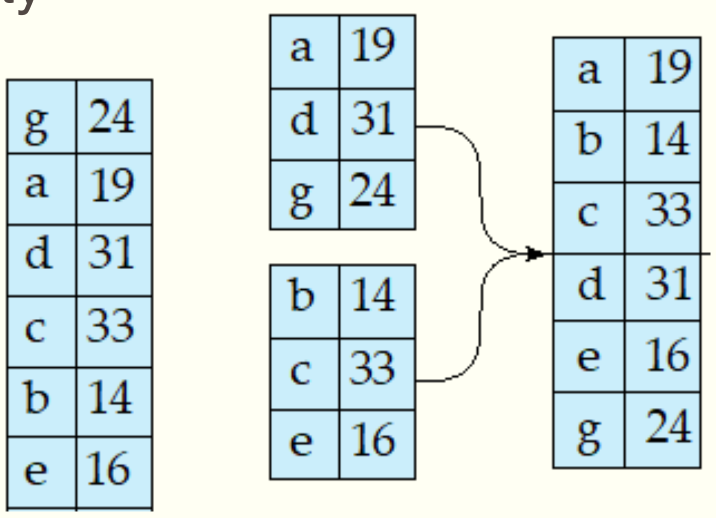
1. Create sorted sublists.
   - Repeatedly do the following till the end of the relation:
     (a) Read *M* blocks of relation into memory
     (b) Sort the in-memory blocks as sorted sublist
     (c) Write sorted sublist to disk
   - Let there be N sorted sublists of size *M*
   - Number of transfers: $b_R + b_R = 2b_R$
   - Number of seeks: $b_R/M + b_R/M = 2b_R/M$



External Merge Sort (cont.)

2. If N < M, merge the sublists (N-way merge)
   1. Use b_b blocks per sublist to buffer input, and 1 block to buffer output.
   2. repeat
      1. Select the first tuple (in sort order) among all N buffers (N blocks)
      2. Write the tuple to the output buffer (1 block). If the output buffer is full write it to disk.
      3. Delete the tuple from its input buffer.
         If the input buffer becomes empty then
           read the next block (if any) of the run into the buffer.
   3. until all input buffers and output buffers are empty

- Number of transfers: $b_R + b_R = 2b_R$
- Number of seeks: $b_R/b_b + b_R/b_b = 2b_R/b_b$



External Merge Sort (Cont.)

- If $N >= M$, several merge passes are required.
  - If we only read 1 block per sublist, we can merge a group of $M - 1$ sublists per pass.
  - A pass reduces the number of sublists by a factor of $M - 1$, and creates sublists longer by the same factor.
    - E.g. If M=11, and there are 90 sublists, one pass reduces the number of sublists to 9, each 10 times the size of the initial sublists
  - Repeated passes are performed till all runs have been merged into one.

- Number of transfers: $2b_R * (\log_{(M-1)}(b_R/M))$
- Number of seeks: $2b_R * (\log_{(M-1)}(b_R/M))$

External Merge Sort (Cont.)

- If $N >= M$, several merge passes are required.
  - Alternatively, we read b_b block per sublist, so we can merge a group of *$(M/b_b -1)$* sublists per pass.
  - A pass reduces the number of sublists by a factor of *$(M/b_b -1)$,* and creates sublists longer by the same factor.
  - Repeated passes are performed till all runs have been merged into one.
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_R/M) \rceil$.
  - b_b determines the trade-off between number of passes, and disk I/O operation time per pass

- Number of transfers: *$2b_R * (\log_{(M/bb-1)}(b_R/M))$*
- Number of seeks: *$2b_R/b_b * (\log_{(M/bb-1)}(b_R/M))$*

Total Cost for Merge Sort

- If (N < M):
  - Transfer cost: $2b_R + 2b_R$
  - Seek Cost: $2b_R/M + 2b_R/b_b$

- If (N >= M):
  - Transfer cost: $2b_R + 2b_R * (\log_{(M/bb-1)}(b_R/M))$
  - Seek cost: $2b_R/M + 2b_R/b_b * (\log_{(M/bb-1)}(b_R/M))$

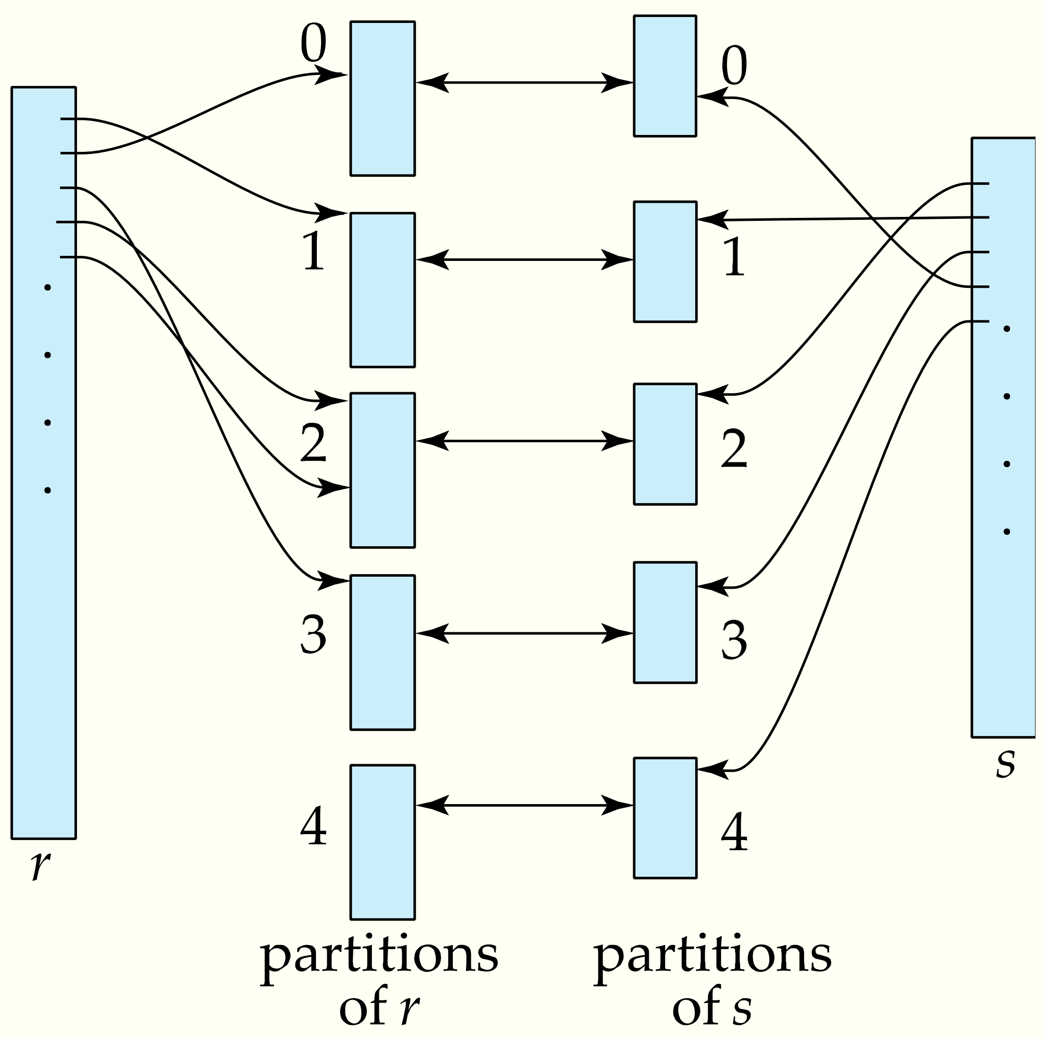
Merge-Join (Cont'd)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus, the cost of merge join is:
  - the cost of sorting, if relations are unsorted, plus
  - $b_r + b_s$ block transfers; $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

Hash Join

- Applicable for equijoins and natural joins
- A hash function h is used to partition tuples of both relations
- H maps JoinAttrs values to {0, 1, ..., n} where JoinAttrs denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h\left(t_r\left[JoinAttrs\right]\right)$.
  - $s_0, s_1, \ldots, s_n$ denote partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i where $i = h\left(t_s\left[JoinAttrs\right]\right)$.
- Tuples in r_i need only to be compared with tuples in s_i

Hash Join (Cont'd)



Hash Join Algorithm

- Partition the relation s and r using hashing function h.
  - When partitioning a relation, one block of memory is reserved as the output buffer for each partition
- For each i:
  - Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index using a different hash function h' than the earlier h.
  - Read the tuples in r_i from the disk one by one, for each tuple locate each matching tuple 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.

Cost of Hash-Join

- If recursive partitioning is not required, cost of hash join is:
  - $3(b_r + b_s) + 4n_h$ block transfers. = approx. $3(b_r + b_s)$
  - $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2n_h$ seeks = approx. $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$
- If recursive partitioning required
  - Number of passes required for partitioning build relation s to less than M blocks per partition is $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - Best to choose the smaller relation as the build relation.
  - Total cost estimate is:

    $2(b_r + b_s) \lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +

    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Best case: $b_r + b_s$ transfers, 2 seeks

Example: Hash Join

- Compute $Student \bowtie Takes$, with student as the build relation
- Assume that memory size is 20 blocks
- Perfect hash function that divide
  - students into 5 partitions, each of size 20.
  - Takes into 5 partitions, each of size 80
- Total cost, ignoring cost of writing partially filled blocks
  - Block Transfer: 3 * (100 + 400)
  - Seeks: 2 (100/3 + 400/3) = 336, assuming b_b=3

Number of records:
student (5,000), takes (10,000)

Number of blocks of students:
student (100), takes (400)