COL 362 & COL 632

Joins, Pipelining

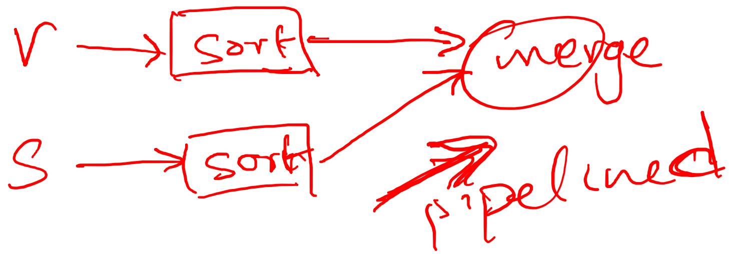
22 Mar 2023

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available (or construct) on the inner relation's join attribute
- 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: $b_r (t_T + t_S) + n_r * \boxed{c}$
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple or 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.

Merge-Join

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



Merge-Join (Cont.)

b_b

- 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:

  $b_r + bs$ block transfers and $\lceil b_r / bb \rceil + \lceil bs / bb \rceil$ seeks

  \+ the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

b_b

Hash-Join

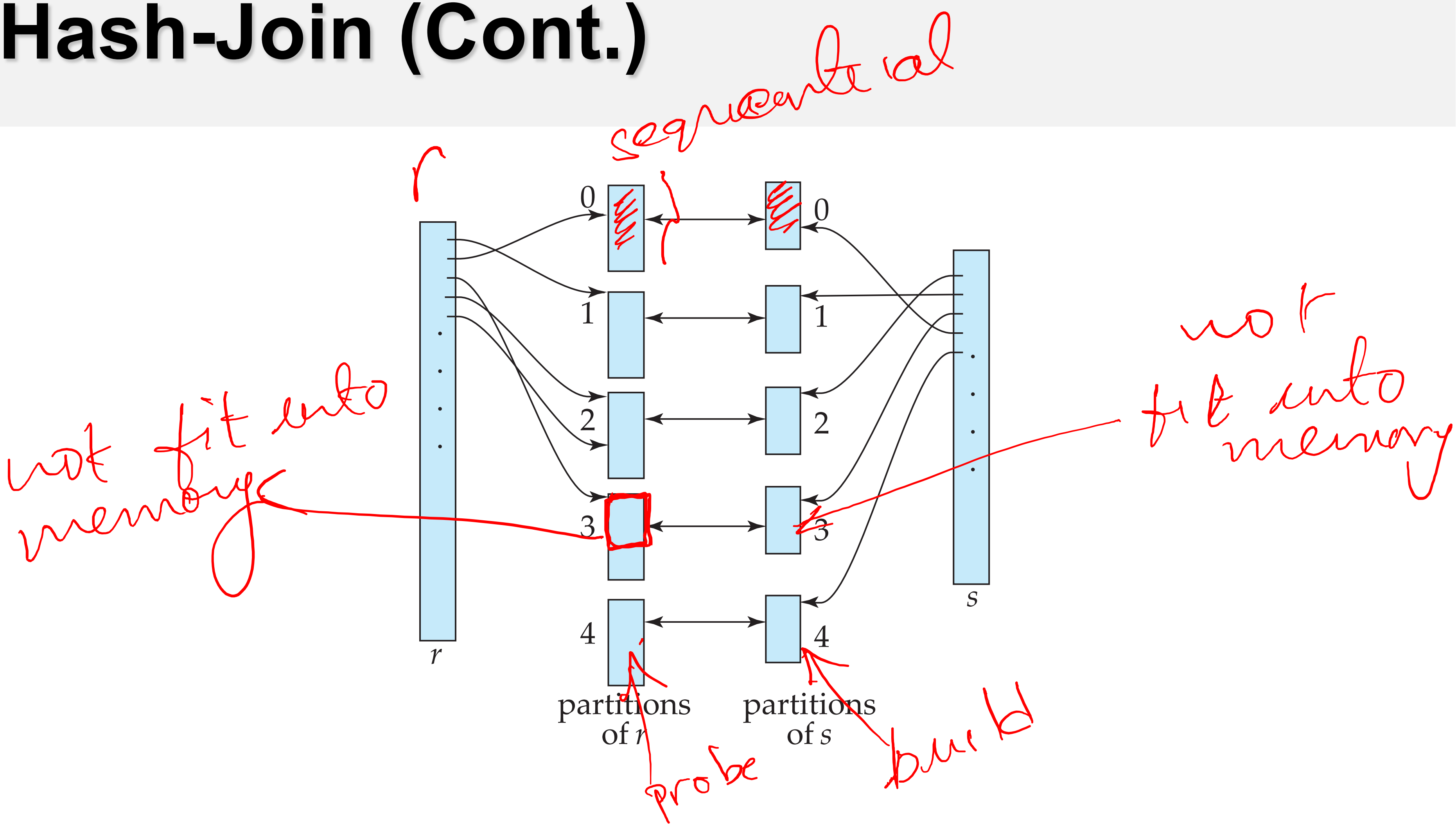
(1) data resides on disk

(2) read the entire record

- Applicable for equi-joins 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 $i = h(t_r [JoinAttrs])$.
  - $r_0, r_1, \ldots, r_n$ denotes partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.

$$\left\lceil \frac{b_r}{b_b} \right\rceil$$

Hash-Join (Cont.)



Hash-Join Algorithm

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.

second phase

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
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g., partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

$b_r + b_r + b_s + b_s + 4n_h$

$b_r + b_s$

n_h = # of partitions due to h.

Cost of Hash-Join

$2n_h$ $2n_h$

$n_h + n_h$ n_h n_h

- If recursive partitioning is not required: cost of hash join is $3(b_r + bs) + 4 n_h$ block transfers + $2(\lceil b_r / bb \rceil + \lceil bs / bb \rceil)$ seeks
- 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

$\left\lceil \frac{b_r}{n} \right\rceil$

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

Hybrid Hash–Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

  Keep the first partition of the build relation in memory.
- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
  - Division of memory:
    - The first partition occupies 20 blocks of memory
    - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
  - the first is used right away for probing, instead of being written out
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M >> \sqrt{b_s}$

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \dots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

      $\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$

- Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

    $(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s)$

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail