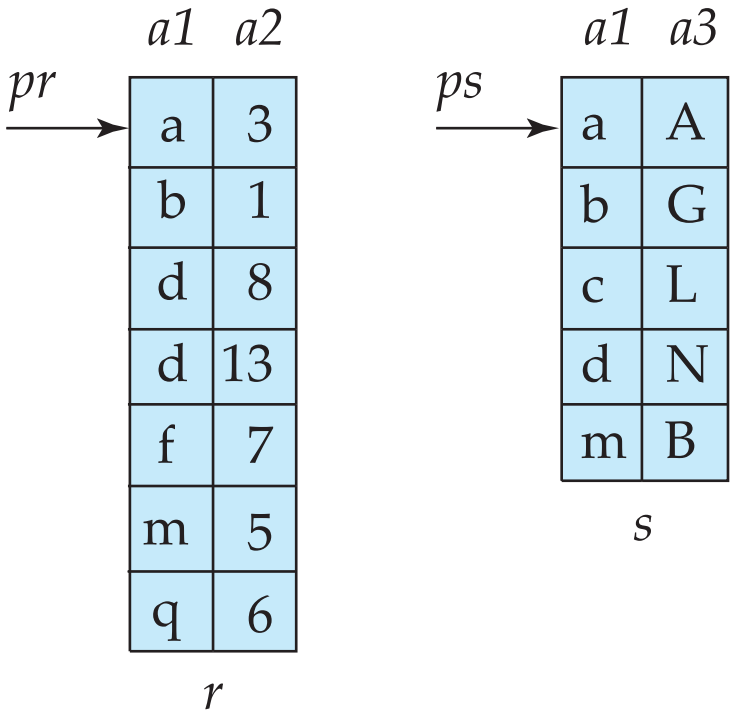
Merge-Join

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 similar to 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
    3. Detailed algorithm in book



Merge-Join (Cont.)

- 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:
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ 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

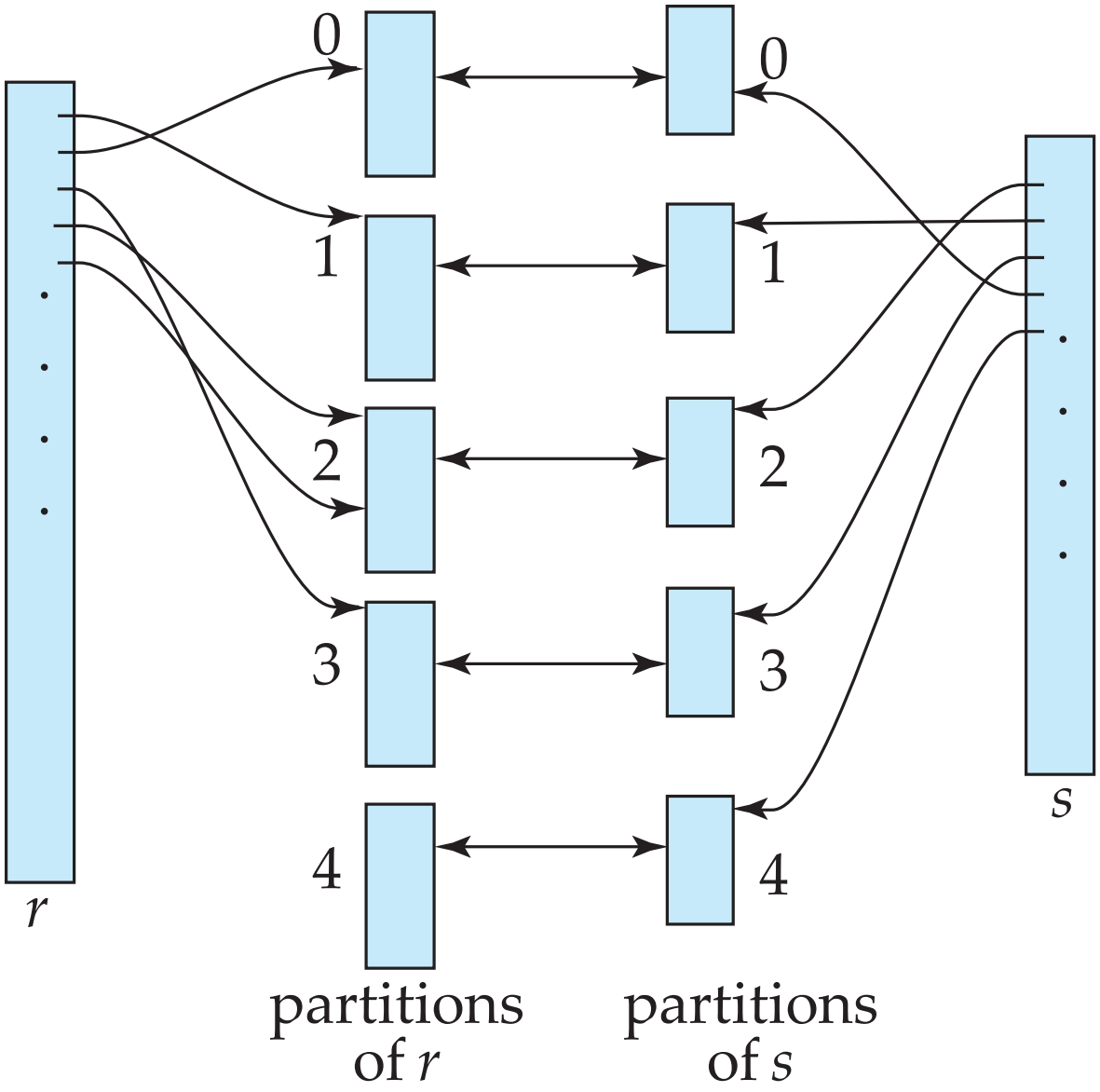
Hash-Join

- 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 $i = h(t_s\,[JoinAttrs])$.
- *Note:* In book, r_i is denoted as H_{ri}, s_i is denoted as H_{si} and n is denoted as n_h.

Hash-Join (Cont.)

- General idea:
  - Partition the relations according to the previous method
  - Perform the join of each partition r_i and s_i
- Note that r tuples in r_i need only to be compared with s tuples in s_i . Need not be compared with s tuples in any other partition, since:
  - an r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s_i.

Hash-Join (Cont.)



Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

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.

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

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers +
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ 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
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

student ⋈ *takes*

- Assume that memory size is 20 blocks
- $b_{student}$= 100 and b_{takes} = 400.
- *Student* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *takes* into five partitions,each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) = 1500 block transfers +
    2(⌈100/3⌉ + ⌈400/3⌉) = 336 seeks (1.5 secs)
  - Block nested loop: 40100 block transfers plus 200 seeks
    Index nested loop: 25100 block transfers and seeks
    Opt. block nested-loop: 2700 transfers plus 46 seeks (0.454 secs)

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, *student* 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.
- *Takes* 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:

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

  - 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

$$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

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

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimization:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

- **Set operations** (∪, ∩ and —): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i add the tuples in the hash index to the result.

Other Operations : Set Operations

- E.g., Set operations using hashing:
  1. as before partition *r* and *s,*
  2. as before, process each partition *i* as follows
     1. build a hash index on r_i
     2. Process s_i as follows
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.
- Modifying merge join to compute $r \mathbin{⟕} s$
  - In $r \mathbin{⟕} s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r \mathbin{⟕} s$:
    - During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.

Other Operations : Outer Join

- Modifying hash join to compute $r \overline{\bowtie} s$
  - If r is probe relation, output non-matching r tuples padded with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

Evaluation of Expressions

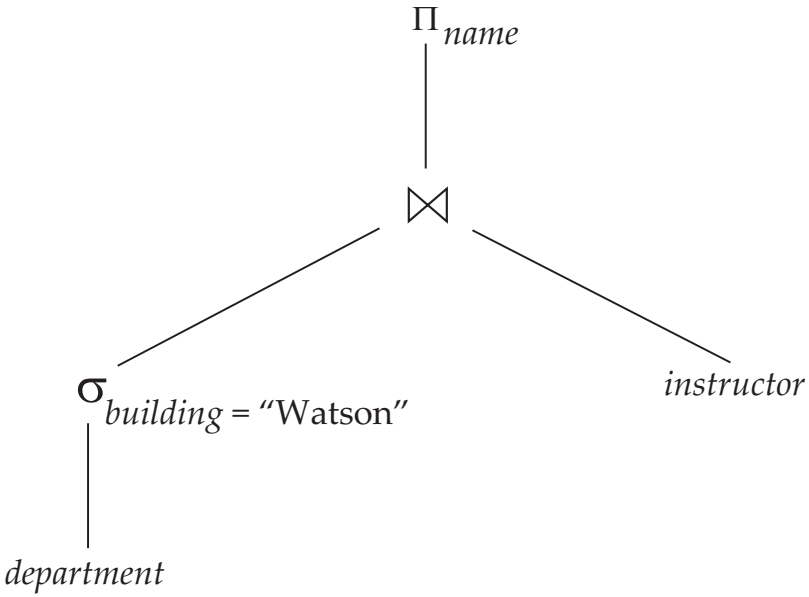
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples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

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building = \text{"Watson"}}(department)$$

then compute the store its join with *instructor,* and finally compute the projection on *name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building = \text{"Watson"}}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

Pipelining (Cont.)

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g. file scan: initialize file scan
        - » state: pointer to beginning of file
      - E.g.merge join: sort relations;
        - » state: pointers to beginning of sorted relations
    - **next()**
      - E.g. for file scan: Output next tuple, and advance and store file pointer
      - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**

Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - intermediate results written to disk and then read back
- Algorithm variants to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
  - **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
    - Symmetrically for s_0 tuples

End of Chapter