# Query Processing 4

**Instructor: Beom Heyn Kim**

beomheynkim@hanyang.ac.kr

Department of Computer Science

# Overview

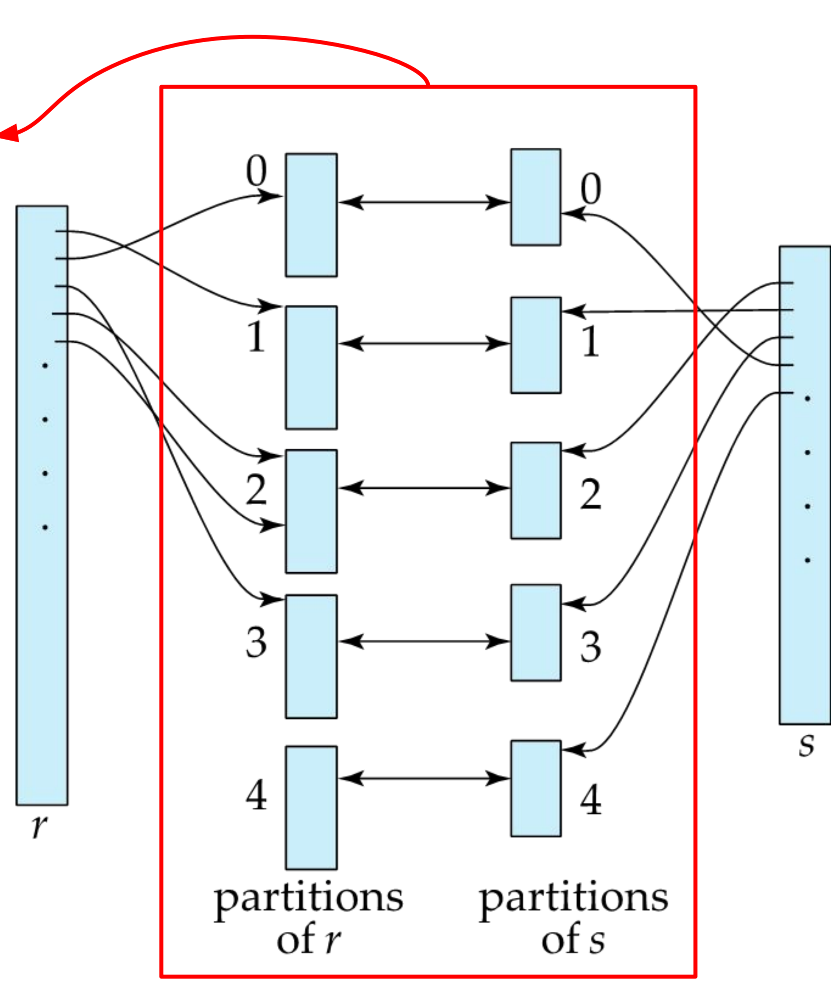
- Join Operation II
- Assignments

# Hash-Join

- Hash-Join is only applicable for equi-joins and natural joins
- Hash-Join uses a hash function $h$ to partition tuples of both relations
  - $h$(tuple's *JoinAttrs* value) = $i$
    - $i \in \{0, 1, ..., n\}$
    - *JoinAttrs* denotes the common attributes of $r$ and $s$ used in the natural join
  - $r_0, r_1, . . ., r_n$ denote partitions of $r$ tuples
    - Each tuple $t_r \in r$ is put in partition $r_i$ where $i = h(t_r[JoinAttrs])$.
  - $r_0,, r_1. . ., r_n$ 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 Figure 15.10 of the book, $r_i$ is denoted as $H_{r_i}$ , $s_i$ is denoted as $H_{s_i}$ and $n$ is denoted as $n_h$.

# Hash-Join (Cont.)

- *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 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 partitions of the probe relation need not fit in memory
- If number of partitions *n* is greater than number of pages *M* of memory, **Recursive partitioning** is required.
  - 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

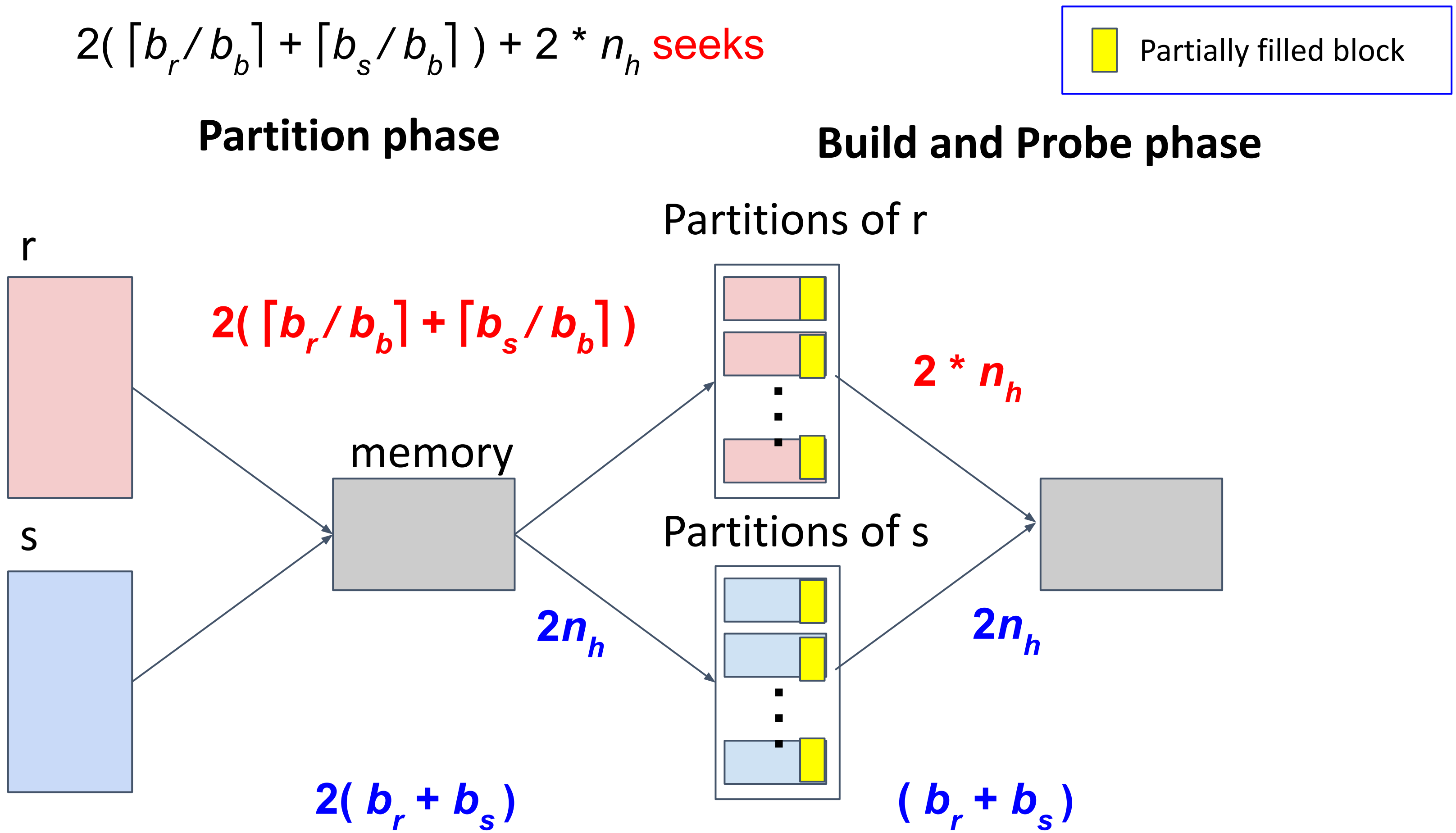
- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition $s_i$ if $s_i$ does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow 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-loop 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 and

  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks



# Overview

- Join Operation II
- Assignments

# Assignments

- Reading: Ch15.5.5
- Practice Excercises: 15.3, 15.5

Solutions to the Practice Excercises:
https://www.db-book.com/Practice-Exercises/index-solu.html

For the practice exercise 15.3, you need to note the followings:

- Just consider block transfers (no seeks needed). The term ‘disk accesses’ in the solution is equivalent to ‘block transfers’
- For b, assume you use the second method of improving block nested-loop join on page 707 in book
- For c, assume you use external merge-sort and you also need to write the final output to the disk for merge join
- For d, just consider that there is no recursive partitioning required. Also, assume that there is no partially filled blocks

# The End