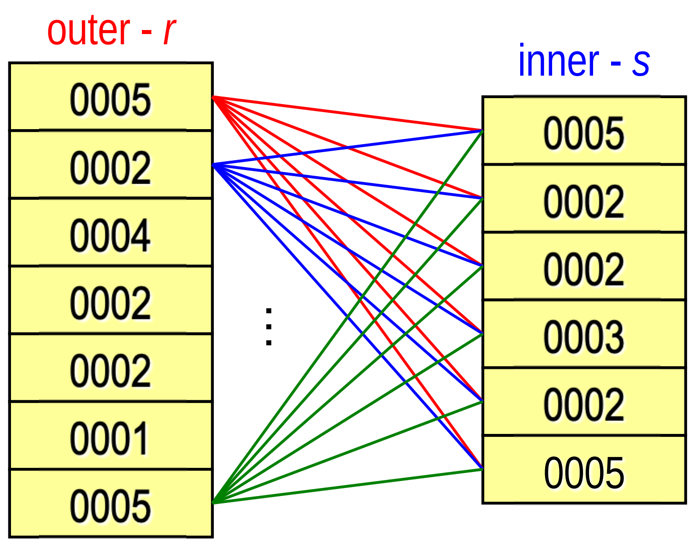
COMP 3311
DATABASE MANAGEMENT SYSTEMS

TUTORIAL 7
QUERY PROCESSING

REVIEW: BLOCK NESTED-LOOP JOIN

- Read in the outer relation r page-by-page.
- For each page of r, scan the entire inner relation s.
- **Best cost:** $b_r + b_s$
  - where b_r and b_s are the number of pages of r and s, respectively, *and* the inner relation, s, is small enough to fit in memory.
- Buffer needed: at least 3 pages (1 for r, 1 for s, 1 for the output).
  - If there are M pages of memory, read M-2 pages of r at a time and use the remaining two pages for s and the output.

Cost: $\lceil b_r / (M-2) \rceil * b_s + b_r$



REVIEW: INDEXED NESTED-LOOP JOIN

- An index lookup can replace a file scan if an index is available on the join attribute of the inner relation.

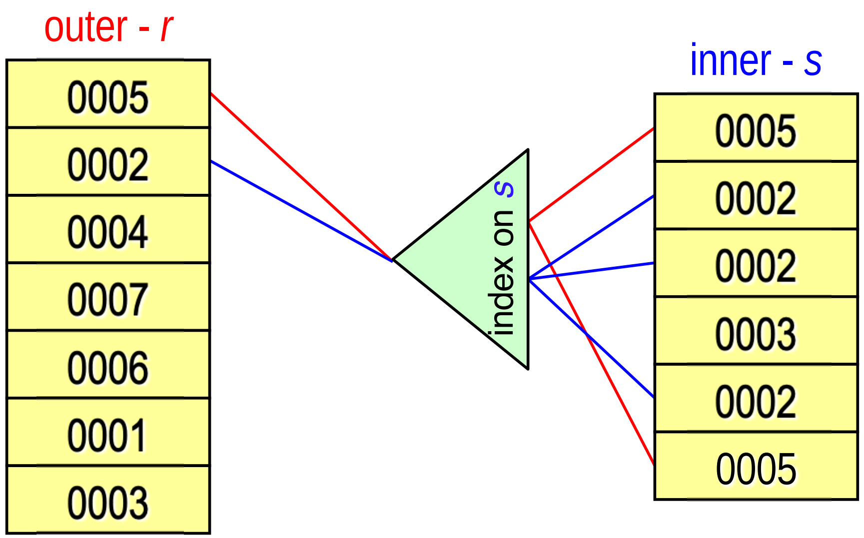
- For each tuple t_r in the outer relation r, use the index to look up tuples in the inner relation s that satisfy the join condition with tuple t_r.

Cost: $b_r + n_r * c$

– n_r is the number of tuples in r.

– c is the cost to traverse the index and fetch all matching s tuples for one tuple, t_r, of r.

– c can be estimated as the cost of a single selection on s using the join condition.



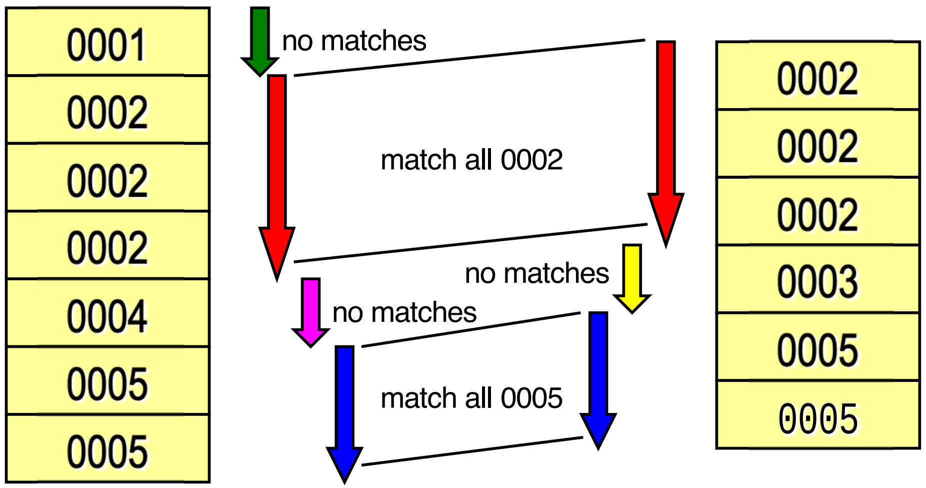
If indexes are available on the join attribute of both r and s, use the relation with *fewer tuples* as the outer relation as this will result in fewer index lookups.

REVIEW: SORT-MERGE JOIN

☞ **Applicable for equi-joins and natural joins.**

- Sort both relations on the join attribute (if not already sorted).
- Merge the sorted relations to join them.
  - ➢ The join step is similar to the merge phase of the merge-sort algorithm.
  - ➢ The main difference is the handling of duplicate values in the join attribute — every pair with the same value on the join attribute must be matched.
- Each block needs to be read only once.

Cost: b_r + b_s + cost of sorting if relations are unsorted



HASH-JOIN

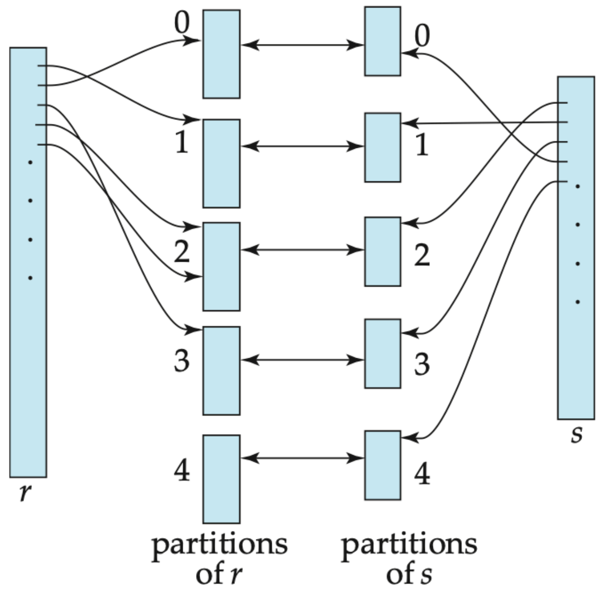
☞ **Applicable for equi-joins and natural joins.**

- A hash function *h* is used to place tuples of both relations into *n* partitions (buckets) (i.e., a hash file organization).

  ☞ **Partitions the tuples of each of the relations into sets that have the same hash value on the join attributes.**

- Only need to compare *r* tuples in partition r_i with *s* tuples in partition s_i.

- **Do not** need to compare *r* tuples in partition r_i 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.
  - Hence, they will hash to the same value $i \Longrightarrow$ the *r* tuple has to be in partition r_i and the *s* tuple in partition s_i!



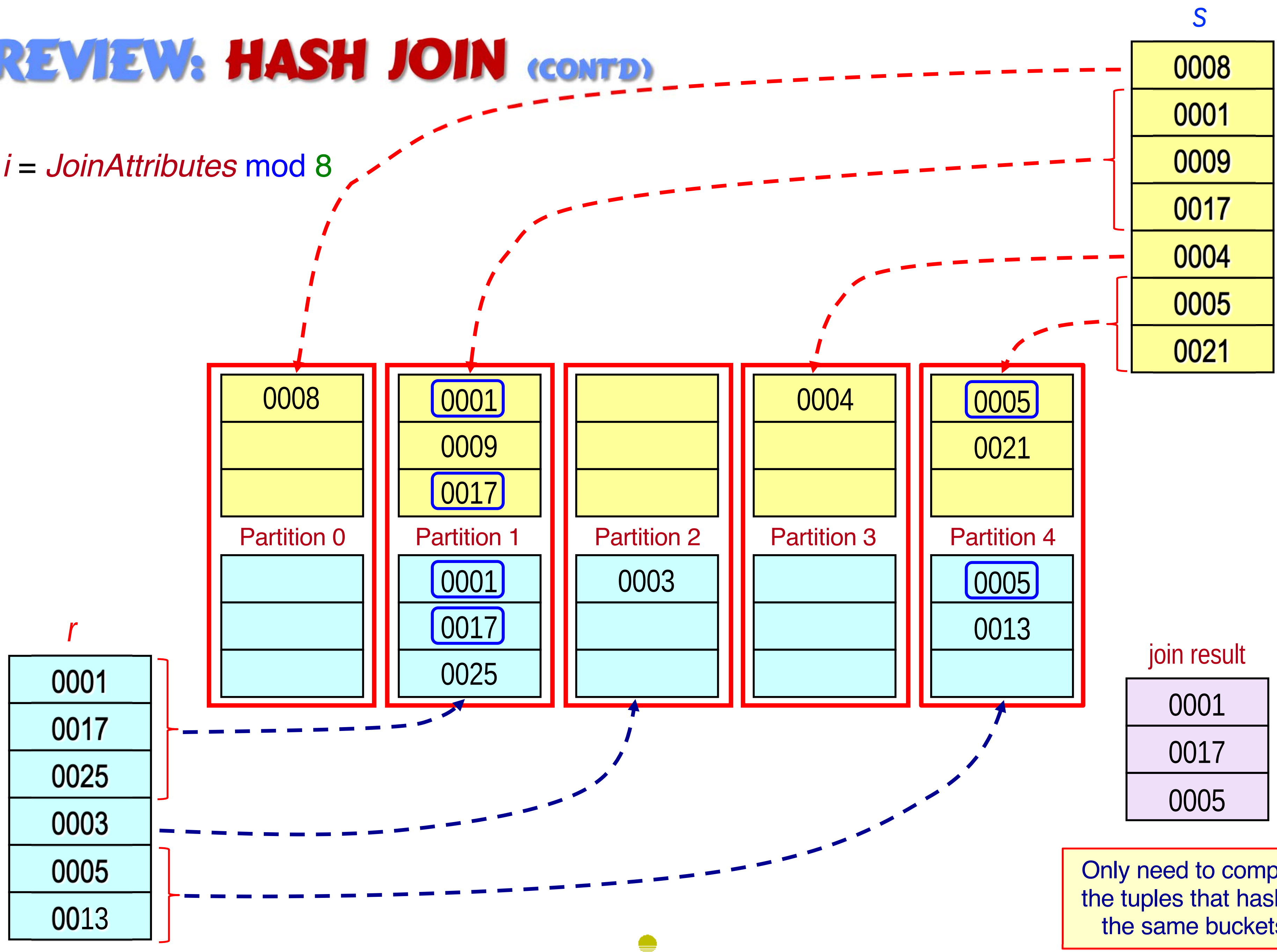
Cost: $3 * (B_r + B_s)$

$\Longrightarrow$ 1 read and 1 write to create the partitions;

1 read to compute the join.

REVIEW: HASH JOIN (CONT'D)

i = *JoinAttributes* mod 8



Only need to compare the tuples that hash to the same buckets.

EXERCISE 1

Relations: R_1(A, B, C) and R_2(<u>C</u>, D, E)

R_1 20,000 tuples	bf_{R1} = 25 tuples/page	R_1 pages: 800
R_2 45,000 tuples	bf_{R2} = 30 tuples/page	R_2 pages: 1500

Assume:

- 100 main memory pages.
- R_2 has a B^+-tree index with 3 levels on the join attribute C, the primary key of R_2.
- R_1 and R_2 are not initially sorted on the join attribute.

Estimate the number of page I/Os required, *in the worst case*, using each of the following join algorithms for $R_1 \bowtie R_2$:

a) Block nested-loop join

b) Indexed nested-loop join

c) Sort-merge join

d) Hash join using 10 buckets

Exercise I (cont'd)

R_1 tuples: 20,000; bf_{R1}: 25
pages R_1: 800
R_2 tuples: 45,000; bf_{R2}: 30
pages R_2: 1500
R_2.C: 3 level B+-tree index
M: 100

a) **Block Nested-Loop Join**

– The worst case cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$

i. When R_1 is the outer relation

Cost: $\lceil 800 / (100-2) \rceil * 1500 + 800 =$ 14300 page I/Os

ii. When R_2 is the outer relation

Cost: $\lceil 1500 / (100-2) \rceil * 800 + 1500 =$ 14300 page I/Os

b) **Indexed Nested-Loop Join** (B+-tree index on R_2.C with 3 levels)

– The worst case cost = $b_r + n_r * c$

Cost: 800 + (3 + 1) * 20,000 = 80800 page I/Os

EXERCISE I (CONT'D)

R_1 tuples: 20,000; bf_{R1}: 25
pages R_1: 800
R_2 tuples: 45,000; bf_{R2}: 30
pages R_2: 1500
R_2.C: 3 level B$^+$-tree index
M: 100

c) **Sort-Merge Join**

The worst case cost = sorting cost + b_r + b_s

i. **Sorting cost of** R_1

Cost: $800 * 2 * (\lceil \log_{100-1}(800/100) \rceil + 1) = \underline{3200}$ page I/Os

ii. **Sorting cost of** R_2

Cost: $1500 * 2 * (\lceil \log_{100-1}(1500/100) \rceil + 1) = \underline{6000}$ page I/Os

iii. **Join cost** R_1 JOIN R_2

Cost: $1500 + 800 = \underline{2300}$ page I/Os

Total cost: $3200 + 6000 + 2300 = \underline{11500}$ page I/Os

Can you improve on this cost?

EXERCISE 1 (CONT'D)

R_1 tuples: 20,000; bf_{R1}: 25
pages R_1: 800
R_2 tuples: 45,000; bf_{R2}: 30
pages R_2: 1500
R_2.C: 3 level B$^+$-tree index
M: 100

Improved Merge-Sort Join

Previously we first sorted and merged R_1 and R_2 into 1 sorted run each before doing the join. Instead, we first only sort R_1 and R_2 into runs, but do not merge the sorted runs. Then, since we have 100 memory pages, we can do a 8 + 15 = 23-way merge and join during the merge.

- Create 8 sorted runs of R_1 and write them out.

  Cost: 800 + 800 = 1600 page I/Os

- Create 15 sorted runs of R_2 and write them out

  Cost: 1500 + 1500 = 3000 page I/Os

- Read the 8 sorted runs of R_1 and the 15 sorted runs of R_2 using 1 memory buffer page *per run* (i.e., a total of 23 memory pages) and join them during the merge.

  Cost: 800 + 1500 = 2300 page I/Os

Total cost: 3 * (800 + 1500) = 6900 page I/Os

R_1 tuples: 20,000; bf_{R1}: 25
pages R_1: 800
R_2 tuples: 45,000; bf_{R2}: 30
pages R_2: 1500
R_2.C: 3 level B^+-tree index
M: 100

EXERCISE 1 (CONT'D)

d) **Hash Join**

i. **Use R_1 is the build input** (since it is smaller)

Partition R_1 into 10 partitions, each of size 80 pages. This partitioning can be done in one pass since we use 10 memory pages for the 10 partitions and 1 memory page to read R_1 page-by-page. When a partition page becomes full, we write it to disk and continue doing the partitioning until all of R_1 is read and partitioned.

ii. **Use R_2 is the probe input**

Partition R_2 into 10 partitions, each of size 150 pages. This is also done in one pass similar to the way we partition R_1.

iii. Since we have 100 memory pages, we read each partition (i.e., 80 pages) of the build input R_1, in turn, into memory. For each R_1 partition, read the corresponding probe partition R_2 into memory page-by-page (requires only 1 page) and probe the build partition for matches.

Total cost: 3 * (800 + 1500) = 6900 page I/Os

EXERCISE 2

Given relation R(<u>A</u>, B, C, D, E), organized as a sequential file on search key A, and the information below, answer the questions.

Tuple size: 200 bytes
Attribute A: 16 bytes
Page size: 2400 bytes
Number of tuples: 500,000
Pointer size: 4 bytes

a) How many pages are required to store R?

b) How many index pages are required if the search key A is organized using a static, multi-level index?

c) Consider the query: select * from R where A=xxx. For each of the query evaluation strategies given below, determine the cost in page I/Os of each strategy.
   i. linear scan
   ii. binary search
   iii. index search

d) Consider the query: select * from R where A>700000. What is the cost in page I/Os to answer this query using the index assuming that A is uniformly distributed on the interval [200,000; 800,000] and the leaf index pages are chained?

EXERCISE 2 (CONT'D)

tuple size: 200 bytes
\# tuples: 500,000
attribute A: 16 bytes
pointer size: 4 bytes
page size: 2400 bytes

a) How many pages are required to store R?

$bf_R = \lfloor \text{page size} / \text{tuple size} \rfloor = \lfloor 2400 / 200 \rfloor = \underline{12}$ tuples/page

$\#pages = \lceil \#tuples / bf_R \rceil = \lceil 500000 / 12 \rceil = \underline{41{,}667}$ pages

b) How many index pages are required if the search key A is organized using a static, multi-level index?

Since the file is organized as a sequential file on search key A, the tuples are stored in search-key order. Therefore, the index is sparse.

The number of search-key values in the leaf nodes is equal to the number of pages of the file.

$bf_{indexA} = \lfloor \text{page size} / \text{index entry size} \rfloor = \lfloor 2400 / (16+4) \rfloor$
$= \underline{120}$ index entries/page

$\#pages_{level1} = \lceil \#index\ pages / bf_{indexA} \rceil = \lceil 41667 / 120 \rceil = \underline{348}$ pages

$\#pages_{level2} = \lceil \#level1\ pages / bf_{indexA} \rceil = \lceil 348 / 120 \rceil = \underline{3}$ pages

$\#pages_{level3} = \lceil \#level2\ pages / bf_{indexA} \rceil = \lceil 3 / 120 \rceil = \underline{1}$ page

Total # index pages: 348 + 3 + 1 = 352 pages

tuple size: 200 bytes
tuples: 500,000
attribute A: 16 bytes
pointer size: 4 bytes
page size: 2400 bytes

EXERCISE 2 (CONT'D)

c) Consider the query: select * from R where A=xxx.

i. linear scan

Cost: $\lceil \#pages / 2 \rceil = \lceil 41667 / 2 \rceil$ = 20,834 page I/Os

ii. binary search

Cost: $\lceil \log_2(\#pages) \rceil = \lceil \log_2(41667) \rceil$ = 16 page I/Os

iii. index search

Cost: height of the index + 1 = 3 + 1 = 4 page I/Os

tuple size: 200 bytes
tuples: 500,000
attribute A: 16 bytes
pointer size: 4 bytes
page size: 2400 bytes

EXERCISE 2 (CONT'D)

d) Consider the query: select * from R where A>700000. What is the cost in page I/Os to answer this query using the index assuming that A is uniformly distributed on the interval [200,000; 800,000] and that the leaf index pages are chained?

Since A is uniformly distributed on the interval [200,000; 800,000], we can estimate the proportion of the pages that will be retrieved as

$\lceil(800000 - 700000) / (800000 - 200000)\rceil = \lceil(100000) / (600000)\rceil = 1/6$

Thus, we expect to retrieve 1/6th of the relation's pages.

Cost: $\lceil \#\text{index levels} + 1/6 * \#\text{pages} \rceil = \lceil 3 + 1/6 * 41667 \rceil$

$= \lceil 3 + 6944.5 \rceil$

$=$ 6948 page I/Os