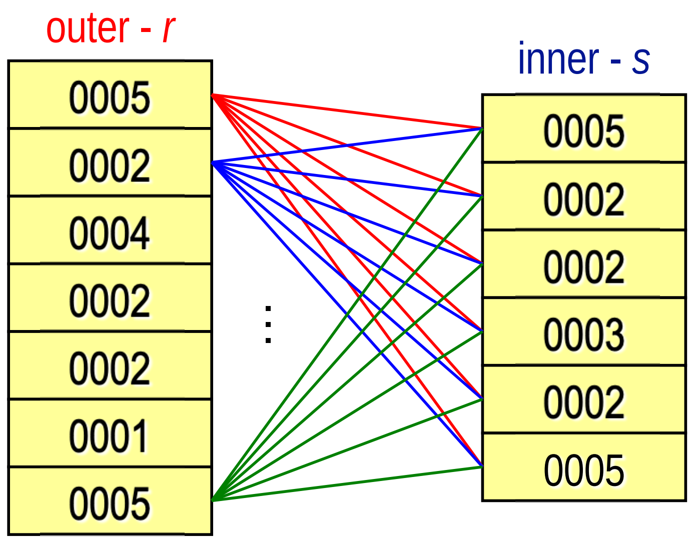
COMP 3311 DATABASE MANAGEMENT SYSTEMS

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 join page I/O 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 the buffer.
- Buffer pages needed: at least 3 (1 for *r*, 1 for *s*, 1 for the output).
  - If there are *M* buffer pages, read *M*-2 pages of *r* at a time and use the remaining two buffer pages for *s* and the output.

Join page I/O cost: $\lceil b_r / (M\text{-}2) \rceil * b_s + b_r$

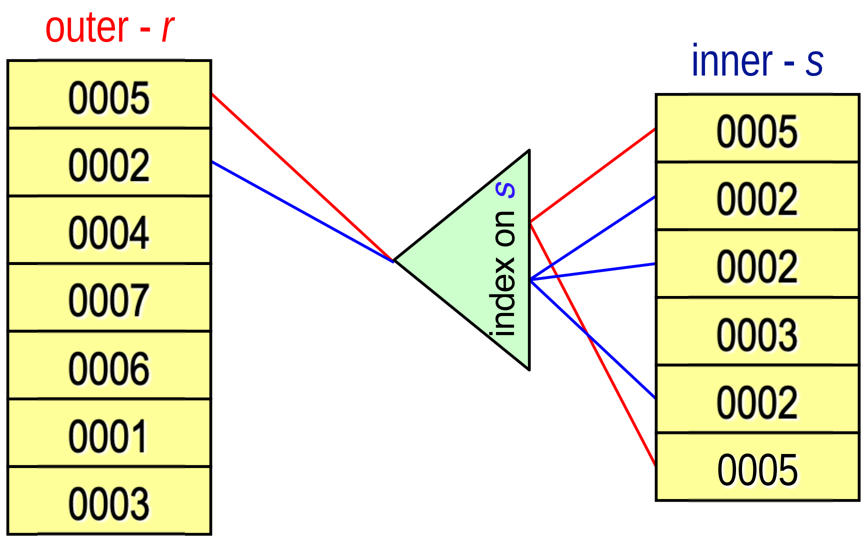


REVIEW: INDEXED NESTED-LOOP JOIN

- 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.

Join page I/O 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.

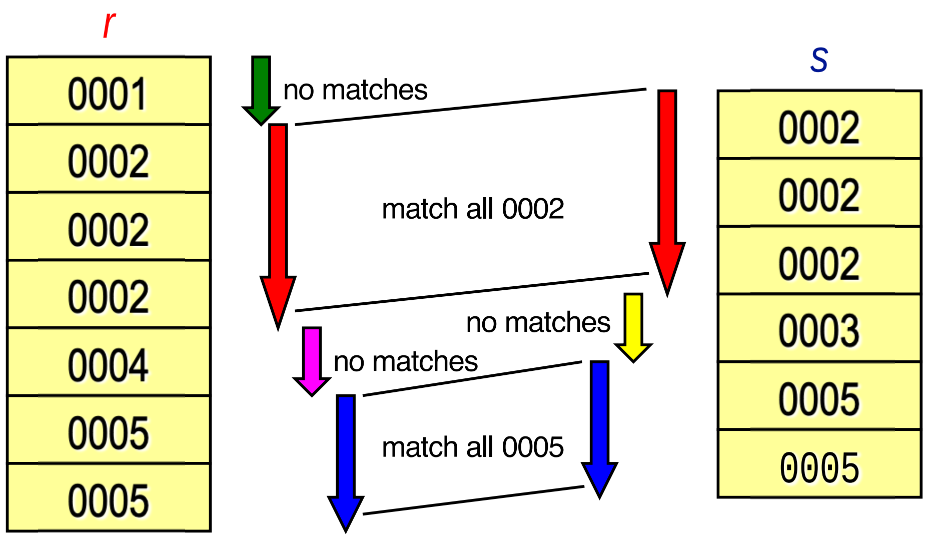
REVIEW: SORT-MERGE JOIN

☞ **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 like 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.

Join page I/O cost:

b_r + b_s + cost of sorting if the relations are not sorted



REVIEW: HASH JOIN

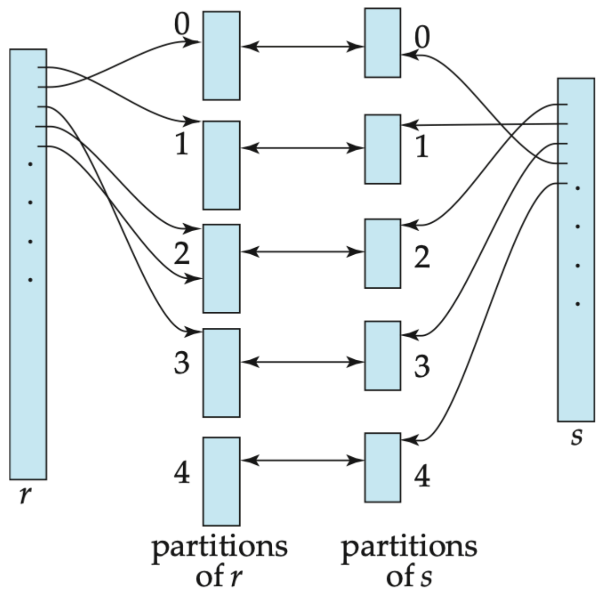
☞ **Applicable for equi-joins and natural joins.**

- 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 join attribute value.
  - 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!



Join page I/O cost:

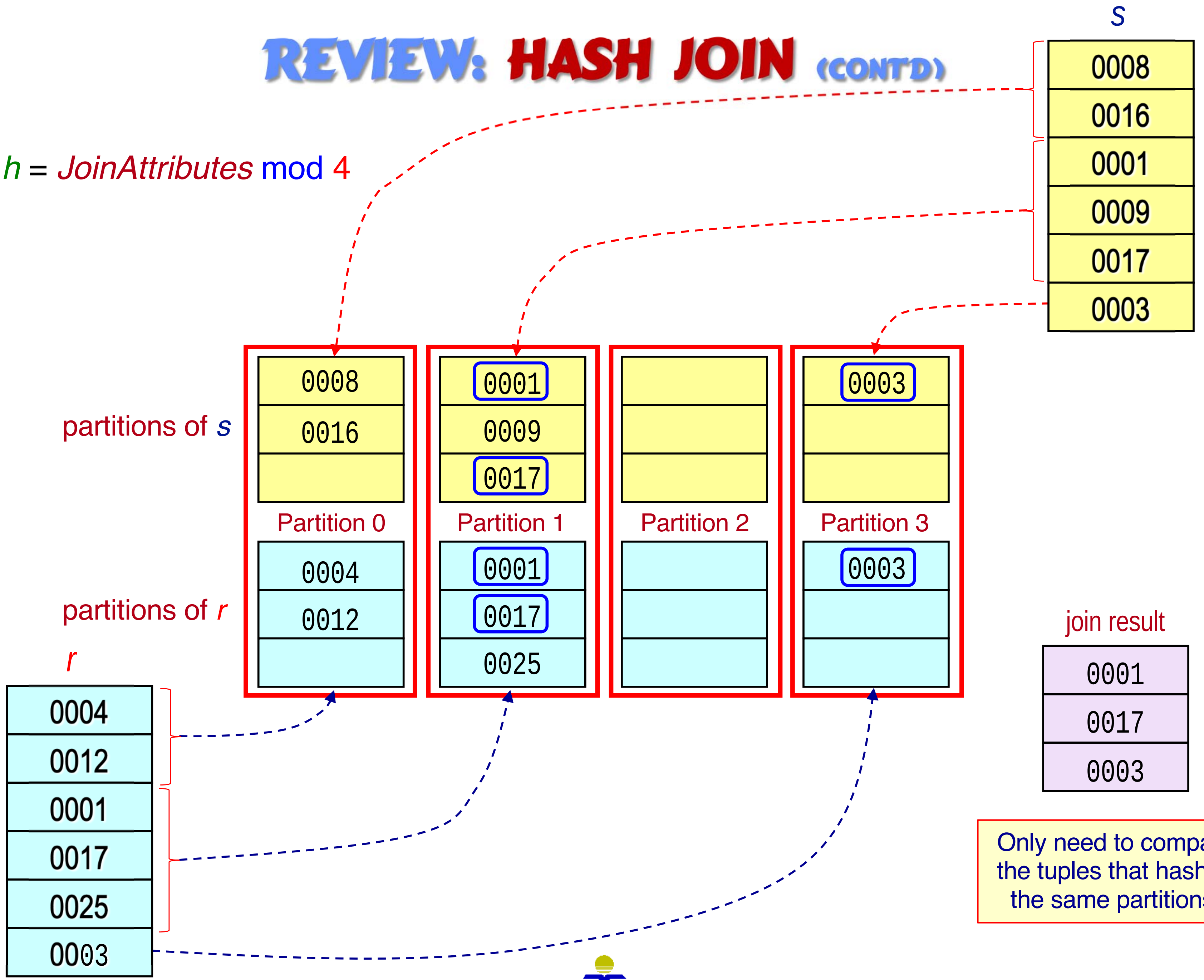
$3 * (B_r + B_s)$

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

1 read to compute the join.

REVIEW: HASH JOIN (CONT'D)

h = *JoinAttributes* mod 4



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

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 buffer 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) Optimized block nested-loop join

b) Indexed nested-loop join

c) Sort-merge join

d) Hash join using 10 buckets

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)

a) Optimized 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

Join page I/O cost: $\lceil 800 / (100-2) \rceil * 1500 + 800 = \underline{14{,}300}$

ii. When R_2 is the outer relation

Join page I/O cost: $\lceil 1500 / (100-2) \rceil * 800 + 1500 = \underline{14{,}300}$

b) Indexed nested-loop join (B^+-tree index on R_2.C with 3 levels)

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

Join page I/O cost: $800 + (3 + 1) * 20{,}000 = \underline{80{,}800}$

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

c) Sort-merge join

The worst-case cost = sorting cost + b_r + b_s

i. Sorting cost of R_1

Sort page I/O cost: $800 * 2 * (\lceil \log_{100-1}(800/100) \rceil + 1) = \underline{3{,}200}$

ii. Sorting cost of R_2

Sort page I/O cost: $1500 * 2 * (\lceil \log_{100-1}(1500/100) \rceil + 1) = \underline{6{,}000}$

iii. Merge cost for R_1 JOIN R_2

Merge page I/O cost: 1500 + 800 = 2,300

Join page I/O cost: 3200 + 6000 + 2300 = 11,500

Can you improve on this cost?

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)

Improved Sort-Merge 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 buffer pages, we can do an 8 + 15 = 23-way merge and join during the merge.

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

  Sort page I/O cost: 800 + 800 = 1,600

– Create 15 sorted runs of R_2 and write them out

  Sort page I/O cost: 1500 + 1500 = 3,000

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

  Merge page I/O cost: 800 + 1500 = 2,300

Join page I/O cost: 1,600 + 3,000 + 2,300 = 6,900

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 (using 10 partitions)

i. Use R_1 as 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 buffer pages for the 10 partitions and 1 buffer page to read R_1 page-by-page. Write a partition page to disk when it becomes full and continue partitioning.

ii. Use R_2 as the probe input

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

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

Join page I/O cost: 3 * (800 + 1500) = 6,900

EXERCISE 2

Given relation R(<u>A</u>, B, C, D, E), organized as an ordered 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 (i.e., not a B^+-tree index)?

c) For the query select * from R where A=xxx, determine the page I/O cost for each of the query evaluation strategies given below.

   i. linear search   ii. binary search   iii. index search

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

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

EXERCISE 2 (CONT'D)

a) How many pages are required to store R?

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

Number of pages needed: $\lceil \#\text{tuples} / bf_R \rceil = \lceil 500000 / 12 \rceil = \underline{41{,}667}$

b) How many index pages are required if the search key A is organized using a static, multi-level index (i.e., not a B$^+$-tree 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 index entries in the leaf nodes is equal to the number of pages of the file (one index entry points to each page of the file).

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

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

$\#\text{pages}_{\text{level2}} = \lceil \#\text{level1 index entries} / bf_{indexA} \rceil = \lceil 348 / 120 \rceil = \underline{3}$

$\#\text{pages}_{\text{level3}} = \lceil \#\text{level2 index entries} / bf_{indexA} \rceil = \lceil 3 / 120 \rceil = \underline{1}$

Total number of index pages: $348 + 3 + 1 = \underline{352}$

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

EXERCISE 2 (CONT'D)

c) For the query select * from R where A=xxx, determine the page I/O cost for each of the query evaluation strategies given below.

   i. linear search

   Query page I/O cost: $\lceil \#pages / 2 \rceil = \lceil 41667 / 2 \rceil = \underline{20,834}$

   ii. binary search

   Query page I/O cost: $\lceil \log_2(\#pages) \rceil = \lceil \log_2(41667) \rceil = \underline{16}$

   iii. index search using the index from b)

   Query page I/O cost: height of the index + 1 = 3 + 1 = $\underline{4}$

EXERCISE 2 (CONT'D)

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

d) For the query select * from R where A>700000, what is the page I/O cost to answer this query using the index from b) assuming that A is uniformly distributed on the interval [200,000; 800,000] and 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

(800000 - 700000) / (800000 - 200000)⌉= (100000) / (600000) = 1/6

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

Query page I/O cost: $\lceil$#index levels + 1/6 * #pages$\rceil$

$= \lceil 3 + 1/6 * 41667 \rceil$

$= \lceil 3 + 6944.5 \rceil$

= 6,948