COMP 3311 DATABASE MANAGEMENT SYSTEMS

LECTURE 16 QUERY PROCESSING: JOIN OPERATION

JOIN OPERATION

- Terminology:

  r, s the relations to be joined.

  n_r, n_s the number of tuples (records) in r and s, respectively.

  B_r, B_s the number of pages in r and s, respectively

  M the available pages of memory.

- All the examples *assume equi-join* on the following relations.

Relation	Customer	Depositor
Number of tuples	10,000	5,000
Number of pages	400	100

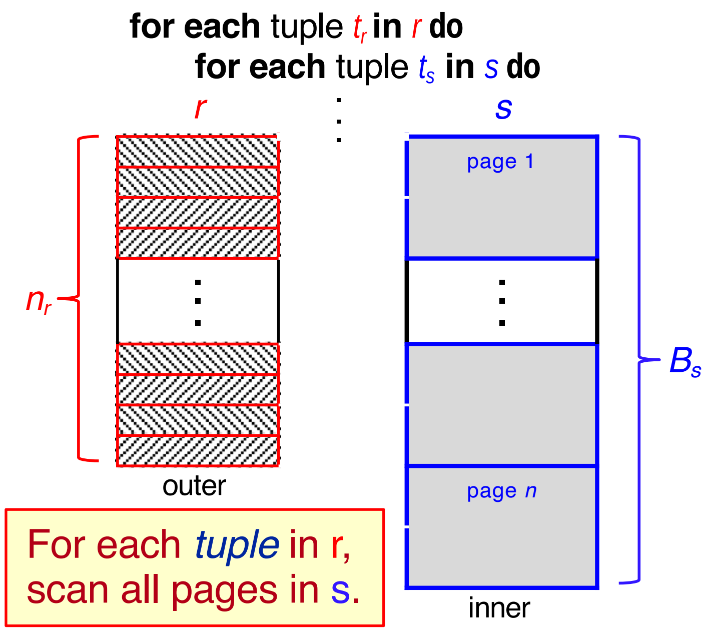
The join attribute is customerName, which is the key of Customer.

NESTED-LOOP JOIN

- Nested-loop join requires no indexes and can be used with any kind of join condition.

- *r* – outer relation; *s* – inner relation

```
for each tuple tr in r do /* r outer relation */
    for each tuple ts in s do /* s inner relation */
        if (tr, ts) satisfies the join condition then
            add (tr, ts) to the result;
```



Cost

Worst case: $n_r * B_s + B_r$ only 1 memory page available for each relation

- ➢ For each tuple t_r in *r*, a complete scan of *s* is performed.

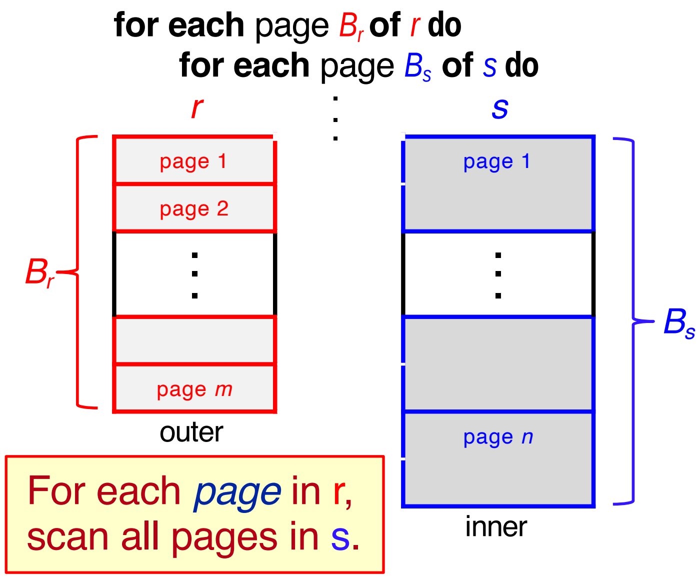
Best case: $B_r + B_s$ when both relations fit into memory

> If a relation fits in memory, use it as the inner relation *s* (to reduce page I/Os since *s* is then read only once).

BLOCK NESTED-LOOP JOIN

- Block nested-loop join requires no indexes and can be used with any kind of join condition.

```
for each page Br of r do /* r outer relation */
    for each page Bs of s do /* s inner relation */
        for each tuple tr in Br do
            for each tuple ts in Bs do
                if (tr, ts) satisfies the join condition then
                    add (tr, ts) to the result;
```



Cost

Worst case: $B_r * B_s + B_r$ only 1 memory page available for each relation

- Each page in s (inner relation) is read only once for each page in r (outer relation) instead of once for each tuple in r.

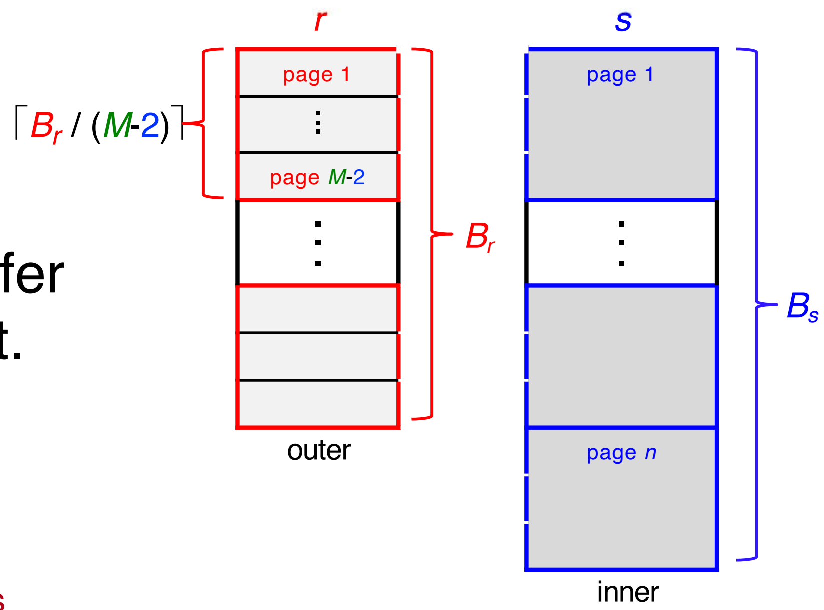
Best case: $B_r + B_s$ when inner relation, s, fits in memory

If a relation fits in memory, use it as the inner relation s.
If neither relation fits in memory, use the smaller relation as the outer relation r.

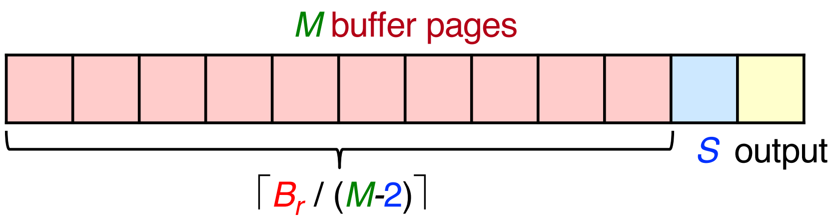
BLOCK NESTED-LOOP JOIN: OPTIMIZATIONS

1. Use M-2 pages as the blocking unit for the outer relation r.
   (M is the memory size in pages.)

   Use the remaining 2 pages to buffer the inner relation s and the output.



Cost: $\lceil B_r / (M\text{-}2) \rceil * B_s + B_r$



2. If the equi/natural-join attributes form a key for the inner relation, we can stop the inner loop on the first match.

EXAMPLE BLOCK NESTED-LOOP JOIN COST

Compute: Depositor JOIN Customer (Depositor is the outer relation)

Number of pages of $B_{Depositor} = 100$, $B_{Customer} = 400$

If neither relation fits in memory, use the smaller relation as the outer relation r.

Worst case cost ($B_r * B_s + B_r$)

100 * 400 + 100 = 40,100 page I/Os

Minimum main memory pages needed to achieve this cost?

Best case cost ($B_r + B_s$)

100 + 400 = 500 page I/Os

How many main memory pages are needed to achieve this cost?

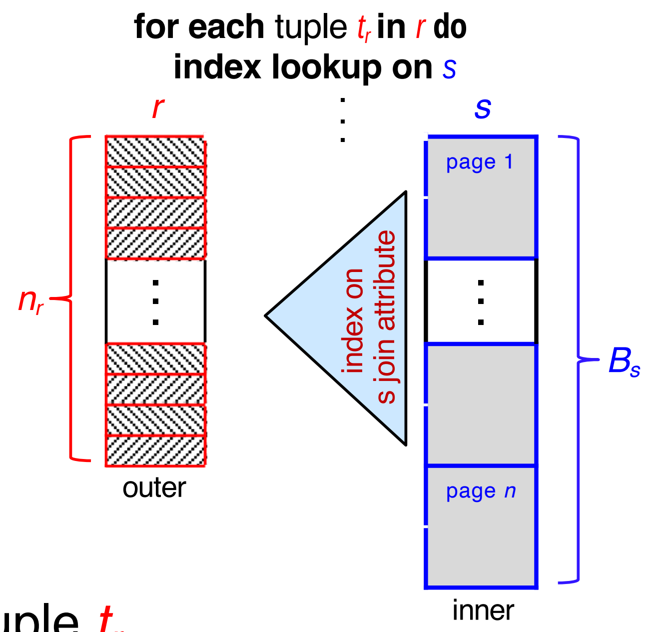
Optimization case ($\lceil B_r / (M\text{-}2) \rceil * B_s + B_r$) (with 52 main memory pages)

$\lceil 100 / (52\text{-}2) \rceil * 400 + 100 = 2 * 400 + 100 = 900$ page I/Os

INDEXED NESTED-LOOP JOIN

- Index lookups can replace file scans if the join is an equi-join or a natural join *and* an index is available on the inner relation's join attribute.

  ☞ **If cost effective, can construct an in memory index just to compute a join.**



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

  Cost: $B_r + n_r * c$

  - Where c is the cost of traversing the index *and* fetching all matching s tuples for one tuple 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 attributes of both relations, use the smaller relation as the outer relation r.

  Why?

EXAMPLE INDEXED NESTED-LOOP JOIN COST

Compute: Depositor JOIN Customer (Depositor is the outer relation)

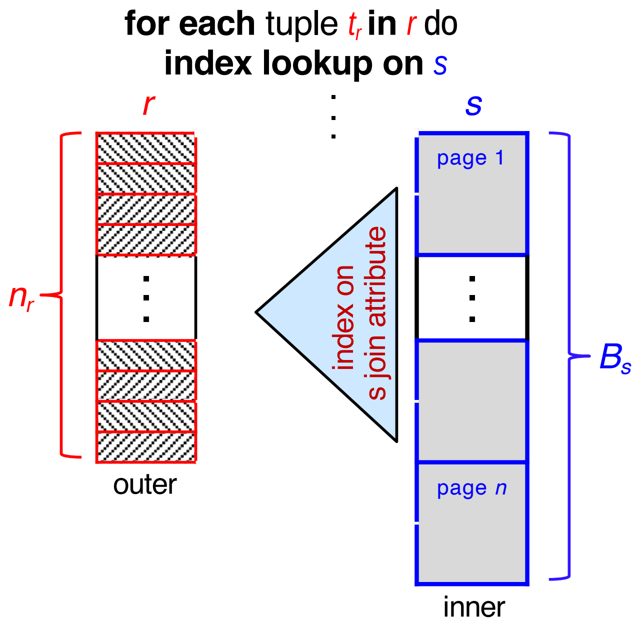
Number of pages of $B_{Depositor} = 100$, Number of tuples $n_{Depositor} = 5000$

- Suppose Customer has a primary B+-tree index with 4 levels on the join attribute customerName, the primary key of Customer.

Cost: $B_r + n_r * c$

$100 + 5000 * 5 =$ 25,100 page I/Os

What is *c*?



Indexed nested-loop is the best join algorithm if there are selective conditions on the outer relation (i.e., few tuples in the outer relation satisfy the condition).

MERGE-JOIN

- Sort both relations on their join attribute (if not already sorted on the join attributes).
- Merge the sorted relations to join them.
  - The join step is similar to the merge stage of the sort-merge algorithm.
  - Need to handle duplicate join attribute values ⟹ need to *match every pair* with the same join-attribute value.

pr →

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

ps →

a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

Cost: B_r + B_s + cost of sorting if relations are unsorted

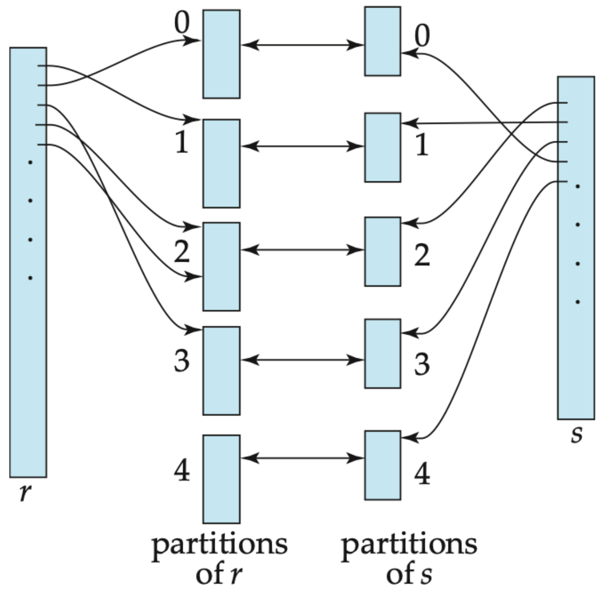
☞ **Can be used only for equi-joins and natural joins.**

- Each page needs to be read only once (assuming all tuples for any given value of the join attributes fit into memory).

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.

Cost: 3 * (B_r + B_s)

⟹ 1 read and 1 write to create the partitions;

1 read to compute the join.

- **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* ⟹ the *r* tuple has to be in partition r_i and the *s* tuple in partition s_i!

HASH-JOIN ALGORITHM

/* Create partitions */

1. Partition the relation *r* using the hash function ***h***.
   - ***h*** maps *JoinAttributes* values to {0, 1, ..., *n*-1} (i.e., *n* partitions), where *JoinAttributes* denotes the common attributes of the natural join of *r* and *s*.
   - r_0, r_1, …, r_{n-1} denote *n* partitions (buckets) of *r* tuples.
     - Each tuple $t_r \in r$ is in partition r_i, where i = ***h***(t_r [*JoinAttributes*]).
   - s_0, s_1, …, s_{n-1} denote *n* partitions (buckets) of *s* tuples.
     - Each tuple $t_s \in s$ is in partition s_i, where i = ***h***(t_s [*JoinAttributes*]).
   - One page of memory is reserved as the output buffer for each partition.

2. Partition *s* similarly.

   ☞ **Need to read and write *r* and *s* to create the partitions.**

/* Perform indexed nested-loop join using hash index */

3. For each *i:*
   - Load partition r_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***. Relation *r* is called the build input.
   - Read the tuples in partition s_i from the disk page-by-page. For each tuple t_s locate each matching tuple t_r in r_i using the in-memory hash index. Relation *s* is called the probe input.

HASH-JOIN ALGORITHM (CONT'D)

- The number of partitions n is such that each partition of the build input r should fit in the available main memory pages M. Assuming each partition has the same size: $M \geq \lceil B_r/n \rceil$.

- Also $M \geq n+1$ because for each partition we should have one buffer page (plus one page for the input buffer).

- In order to satisfy these conditions: $M > \sqrt{B_r}$
  - The probe relation partitions need not fit in memory.

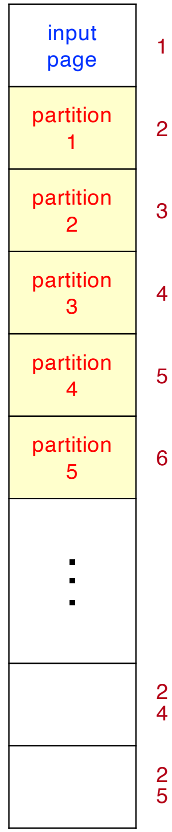
- **Recursive partitioning** is required if the number of partitions n is greater than the number of pages M of memory.
  - This is rarely necessary: recursive partitioning is not needed for relations of 1GB or less with memory size of 2MB and page size of 4KB.

EXAMPLE HASH-JOIN

Compute: Depositor JOIN Customer
M = 25 pages $B_{Depositor}$ = 100 $B_{Customer}$ = 400

- Depositor is the build input (since it is the smaller relation).
  - Partition Depositor into 5 partitions, each of size 20 pages. This can be done in one pass (need to read and write Depositor relation).
  - Requires 6 pages of memory; one to read Depositor and 5 for the hash partitions.
  - Creates 5 partitions of 20 pages each.

- Customer is the probe input.
  - Partition Customer into 5 partitions, each of size 80 pages. This is also done in one pass (need to read and write Customer relation).
  - Requires 6 pages of memory; one to read Depositor and 5 for the hash partitions.
  - Creates 5 partitions of 80 pages each.



☞ **The number of partitions for the two relations must be the same. Why?**

EXAMPLE HASH-JOIN (CONT'D)

Compute: Depositor JOIN Customer

M = 25 pages $B_{Depositor}$ = 100 $B_{Customer}$ = 400

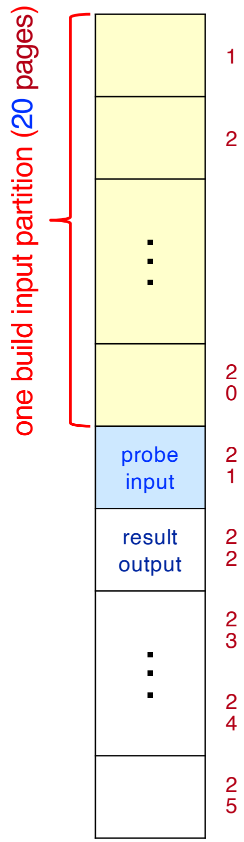
- In turn, read each build input partition into memory.
  - The build input is 5 partitions of size 20 pages each.

☞ **Can hold one partition of build input in memory.**

- Read the *corresponding* probe input partitions page-by-page and probe against the build input partition.
  - The probe input is 5 partitions of size 80 pages each.
- Need to read both Depositor and Customer relations once to compute the join result.

Total cost: 3 * (100 + 400) = 1500 page I/Os.

➢ This ignores the cost of writing partially filled pages.



COMPLEX JOINS

- Join with a conjunctive condition: $r \textbf{ JOIN}_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}\ s$
  1. Use either nested-loop/block nested-loop join, ***or***
  2. Compute the result of one of the *simpler joins* $r \textbf{ JOIN}_{\theta i}\ s$
     - The final result contains those tuples in the intermediate result that also 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: $r \textbf{ JOIN}_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}\ s$
  1. Use either nested-loop/block nested-loop join, ***or***
  2. Compute the result as the *union of the records in the individual joins*

     $(r \textbf{ JOIN}_{\theta 1}\ s) \cup (r \textbf{ JOIN}_{\theta 2}\ s) \cup \ldots \cup (r \textbf{ JOIN}_{\theta n}\ s)$

     ☞ **Useful only if all conditions are restrictive (selective).**