Databases and Information Systems
CS303

Query Processing

26-10-2023

Recap

- Algorithms to evaluate operations
  - Selection of tuples
    - Linear Scan
    - Scans using index, hashing
  - Sorting
    - External MergeSort Algorithm
  - Join Operation
    - Nested Loop Join
    - Block Nested Loop Join
    - Index Nested Loop Join

Join Operation

- Nested loop join

for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \cdot t_s$ to the result;
    end
end

- Requires no index
- Can be used for any join-condition

- Number of tuples considered : $n_r * n_s$
- Number of block transfers : $n_r * b_s + b_r$
- Number of seeks : $n_r + b_r$

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Join Operation

- Nested loop join

for each tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**
    test pair (t_r, t_s) to see if they satisfy the join condition θ
    if they do, add $t_r \cdot t_s$ to the result;
  end
end

- If there is space to bring one full relation to memory:
  - Bring the inner loop relation

- Number of tuples considered : $n_r * n_s$
- Number of block transfers : $b_s + b_r$
- Number of seeks : 2

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Join Operation

- ## Block Nested-loop join

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result;
            end
        end
    end
end
```

- Each block of inner relation is read once for every block of outer relation
- Total block transfers : $b_r * b_s + b_r$
- Total number of seeks : $2\, b_r$
- Efficient to use smaller relation as outer (if neither fits into the memory)
- Exercise: Analyze for student ⋈ takes

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Join Operation

- Indexed Nested-Loop join:
  - If an index is available on the inner loop's join attribute, index lookups can replace file scans
  - For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r
  - Cost : $b_r (t_T + t_S) + n_r * c$
    - n_r is the number of records in r
    - c is the cost of a single selection on s using the join condition

for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \cdot t_s$ to the result;
    end
end

Join Operation

- Merge Join
  - Can be used for Natural Join and Equi-joins
  - Associates one pointer for each

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

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                        end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ⋈ t_r to result;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

Join Operation

- Merge Join
  - Works well if S_s fits into the memory
    - If not use block nested joins for such S_s
  - Both r and s should be sorted with respect to the join attributes.
  - Assume sorted:
    - Number of block transfers : $b_r + b_s$
    - If b_b blocks are allocated to each relation then we need $b_r/b_b + b_s/b_b$ many seeks
  - If not sorted, add sorting cost
  - If S_s does not fit in the memory, add the nested block join cost

```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                        end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ⋈ t_r to result;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

Join Operation

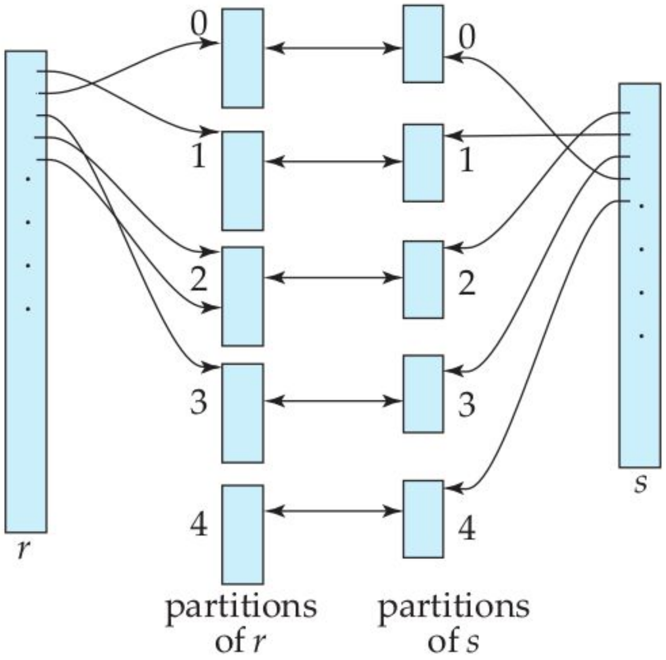
- Hybrid Merge Join
  - Merge-join over unsorted tuples using secondary index on the join attributes
    - Can be costly since records can be distributed across blocks

  - If one relation is sorted, other has secondary index
    - For every record of the sorted relation compute the pointers in the secondary relation
    - Sort this result with respect to the pointers and retrieve every block of the second relation at most once to compute the result
      - Sometimes this could be better

  - How to handle two unsorted index (but with secondary index) using above technique ?
    - Exercise

Join Operation

- Hash Join
  - For Natural join and Equi join
  - Partition the tuples of each of the relations into sets that have the same hash value on the join attributes
  - Assume:
    - h is a hash function mapping JoinAttrs values to $\{0, 1, \ldots, n_h\}$, where JoinAttrs denotes the common attributes of r and s used in the natural join
    - $r_0, r_1, \ldots, r_{nh}$ denote partitions of r tuples, each initially empty.
      Each tuple $t_r \in r$ is put in partition r_i
      where $i = h(t_r\,[\text{JoinAttrs}])$
    - $s_0, s_1, \ldots, s_{nh}$ denote partitions of s tuples, each initially empty.
      Each tuple $t_s \in s$ is put in partition s_i
      where $i = h(t_s\,[\text{JoinAttrs}])$



Join Operation

- Hash Join
  - Suppose that an r tuple and an s tuple satisfy the join condition
    - They 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
  - Values need to be compared only with same partitions
  - Algorithm builds partition on one relation (build relation) and probes the other
    - Here s is build relation and r is the probe relation

```
/* Partition s */
for each tuple t_s in s do begin
   i := h(t_s[JoinAttrs]);
   H_si := H_si ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
   i := h(t_r[JoinAttrs]);
   H_ri := H_ri ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
   read H_si and build an in-memory hash index on it;
   for each tuple t_r in H_ri do begin
      probe the hash index on H_si to locate all tuples t_s
         such that t_s[JoinAttrs] = t_r[JoinAttrs];
      for each matching tuple t_s in H_si do begin
         add t_r ⋈ t_s to the result;
      end
   end
end
```

Join Operation

- Hash Join
  - Value of n_h should be large enough
  - Best to use the smaller input relation as the build relation
  - If the size of the build relation is b_s blocks
    - Each of the n_h partitions have to be of size less than or equal to M
      - n_h must be at least b_s /M.
  - But distribution need not be uniform
    - So n_h should be correspondingly larger

```
/* Partition s */
for each tuple t_s in s do begin
   i := h(t_s[JoinAttrs]);
   H_{s_i} := H_{s_i} ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
   i := h(t_r[JoinAttrs]);
   H_{r_i} := H_{r_i} ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
   read H_{s_i} and build an in-memory hash index on it;
   for each tuple t_r in H_{r_i} do begin
      probe the hash index on H_{s_i} to locate all tuples t_s
         such that t_s[JoinAttrs] = t_r[JoinAttrs];
      for each matching tuple t_s in H_{s_i} do begin
         add t_r ⋈ t_s to the result;
      end
   end
end
```

Join Operation

- Hash Join
  - If n_h is larger than number of blocks in the memory then relation cannot be partitioned in one pass
  - So in first pass, split into at most as many partitions as there are blocks available for use as output buffers
  - Buckets of this pass is read by the subsequent passes (Recursive partitioning)
  - Recursive partitioning is not needed if $M > n_h + 1$
- Cost : Exercise

Join Operation

- ## Hash Join : Overflows
    - Partitions can have overflows

    - Use fudge factor (allocate more space for all partitions than required)
        - Even then overflows can happen

    - Overflow resolution is performed during the build phase, if a hash-index overflow is detected.
        - If some s_i is found to be too large, it is further partitioned into smaller partitions by using a different hash function.
        - Hash for r_i is updated accordingly

    - Overflow avoidance performs the partitioning carefully, so that overflows never occur during the build phase.
        - The build relation s is initially partitioned into many small partitions, and then some partitions are combined in such a way that each combined partition fits in memory.
        - The probe relation r is partitioned in the same way as the combined partitions on s
            - Tthe sizes of r_i do not matter.

    - If a large number of tuples in s have the same value for the join attributes, the resolution and avoidance techniques may fail.
        - In that case, creating an in-memory hash index does not work
            - Use block nested-loop join, on those partitions.

Join Operation

- Complex join conditions
  - Nested loop join and Block nested loop joins work for all join conditions
    - Other than Natural Join and Equi-Joins
  - Consider $r \bowtie_{\theta 1 \wedge \theta 2 \wedge .. \theta n} s$
    - Different join technique might be applicable for each $r \bowtie_{\theta i} s$
    - Compute the simplest $r \bowtie_{\theta i} s$ and final result contains those combinations that satisfy $\theta_1 \wedge \theta_2 \wedge ,..\theta_{i-1} \wedge \theta_{i+1} \wedge ,..\theta_n$
  - Consider $r \bowtie_{\theta 1 \vee \theta 2 \vee .. \theta n} s$
    - Compute each each $r \bowtie_{\theta i} s$
    - Final result will $(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup ..... (r \bowtie_{\theta n} s)$

Duplicate Elimination

- Can be done via sorting
  - Duplicates can be eliminated at the merge step
- Can also be done using Hashing
  - Partition the relation based on hashing
  - When constructing the hash index, a tuple is added to the table only if it is not already present
- Duplicate elimination is costly
  - So SQL needs explicit instruction from the user

Projection

- Implemented by performing projection on each tuple
    - Duplicates eliminated if needed
- If Key is part of projection then duplicates do not occur

Set Operations

- First sort both the relations
  - Then perform union, intersection, set difference
  - Sorting can be done on any attribute as long as both relations use the same attribute for sorting
  - Cost : Exercise
- Hashing can also be used for set operations:
  - Partition r and s into $r_1 \ldots r_{nh}$ and $s_1 \ldots s_{nh}$ respectively
  - $r \cup s$
    - Build in memory hash index for r_i
    - Add tuples in s_i to the hash index only if they are not already present
    - Add tuples in the hash index to the result
  - $r \cap s$
    - Build in memory hash index for r_i
    - For each tuple in s_i output the tuple in the result only if it is already present in the hash index
  - $r - s$ Exercise

Outer Join

- First method :
  - Compute Inner Join and then add remaining tuples as required
  - Use set operations to achieve this

- Second method : Modify the Join algorithms
  - Easy to extend nested loop join to compute outer joins
  - Hard to extend Block nested Loop join
  - Natural outer joins and equi-outer joins can be done by extending merge-join and hash joins
    - For tuples without a match, output them with padded nulls

Aggregation

- Similar to duplicate elimination
  - Use sorting / hashing based on the GROUP BY attribute
  - Sum / min / max can be computed on the fly as the groups are being constructed
  - Count can keep track of count for each group
  - For average, compute sum and count for each group, finally divide appropriately

Evaluation of Expressions

- Evaluate one operation at a time in the appropriate order

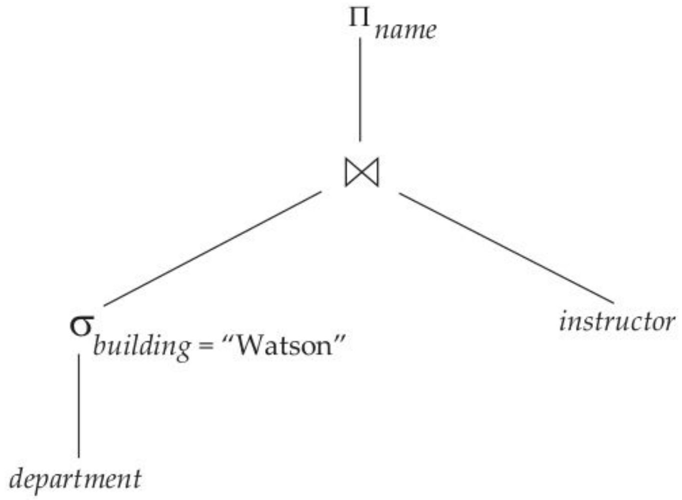
- Store the result in a temporary relation (materialization)
  - Disadvantage : needs to write temporary relations to disk if they are large

- Pipelining
  - Evaluate several operations simultaneously and results of one operation passed on to the next, without the need to store a temporary relation

Evaluation of Expressions : Materialization

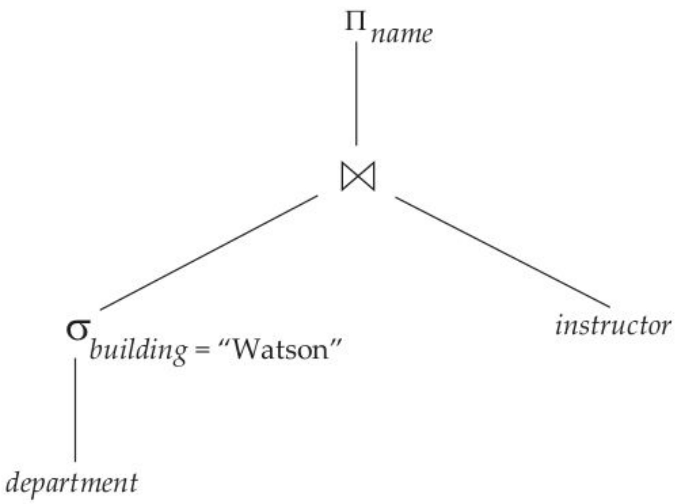
- Consider the operation tree of the given expression

$$\Pi_{name}(\sigma_{building = \text{"Watson"}}(department) \bowtie instructor)$$



Evaluation of Expressions : Materialization

- Start from the leaf and evaluate each expression
- Store the result at every step
- Cost of evaluation needs to add the cost of writing relation back to the disk



Evaluation of Expressions : Pipelining

- Combining several relational operations into a pipeline of operations
  - results of one operation are passed along to the next operation in the pipeline

- Example $\Pi_{a1\ a2}\ (r \bowtie s)$
  - When the join operation generates a tuple of its result,
  - Passes that tuple immediately to the project operation for processing.
  - Avoids creating the intermediate result, and instead create the final result directly

- Advantages
  - Eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation
  - Start generating query results quickly, if the root operator of a query-evaluation plan is combined in a pipeline with its inputs.
    - Useful if the results are displayed to a user as they are generated
    - Otherwise there may be a long delay before the user sees any query results

Evaluation of Expressions : Pipelining

- Demand-driven pipelining (Top down)
  - System makes repeated requests for tuples from the operation at the top of the pipeline
  - Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns it
  - If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations
    - System keeps track of what has been returned so far
  - If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs

Evaluation of Expressions : Pipelining

- Producer-driven pipelining (Bottom up)
  - Operations do not wait for requests to produce tuples, but instead generate the tuples
  - Each operation in a producer-driven pipeline is modeled as a separate process or thread
    - Takes a stream of tuples from its pipelined inputs and generates a stream of tuples for its output

Evaluation of Expressions : Pipelining

- Demand driven pipelining
  - Each operation is called an iterator
  - Provides the functions:
    - open() asks the input to start
    - next() asks the input to give the next tuple (or tuples)
    - close() says no more input is required
  - Iterator maintains the state of its execution in between calls, so that successive next() requests receive successive result tuples.
  - Example:
    - For linear scan : open() starts a file scan next() scan continues from the previous point
    - For merge-join :
      - open() opens the inputs, if they are not sorted then it will sort it
      - next() returns the next pair of matching tuples

Evaluation of Expressions : Pipelining

- Producer driven pipelining
    - For each pair of adjacent operations the system creates a buffer to hold tuples being passed from one operation to the next
    - The processes or threads corresponding to different operations execute concurrently
    - Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full
    - An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full
    - Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer
    - The operation repeats this process until all the output tuples have been generated.

Evaluation of Expressions : Evaluation Algorithms

- Some operations are inherently blocking operations like sorting
  - May not be able to output any results until all tuples from their inputs have been examined
- Some operations are inherently non-blocking like joins
  - But specific algorithms can make it blocking like hash-join
    - Requires both relations to be fully partitioned before producing outputs
- It is common to have input relations which are not sorted
  - Use double pipelined join (where input relations are pipelined)

$done_r := false;$
$done_s := false;$
$r := \emptyset;$
$s := \emptyset;$
$result := \emptyset;$
while not $done_r$ **or not** $done_s$ **do**
  begin
    if queue is empty, **then** wait until queue is not empty;
    $t :=$ top entry in queue;
    if $t = End_r$ **then** $done_r := true$
      else if $t = End_s$ **then** $done_s := true$
        else if t is from input r
          then
            begin
              $r := r \cup \{t\};$
              $result := result \cup (\{t\} \bowtie s);$
            end
          else /* t is from input s */
            begin
              $s := s \cup \{t\};$
              $result := result \cup (r \bowtie \{t\});$
            end
  end

Reference:

Database System Concepts by Silberschatz, Korth and Sudarshan (6th edition)

Chapter 12