Parallel Database Query Processing

Parallel Query Processing

Different queries/transactions can be run in parallel with each other.

Interquery parallelism

- There are queries Q1, Q2, Q3, …..Qn
- Queries are processed in parallel

Intraquery Parallelism

- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
- data can be partitioned and each processor can work independently on its own partition

- Queries are expressed in high level language (SQL, translated to relational algebra) makes parallelization easier.

Intraquery Parallelism

Intraquery parallelism: execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.

Two complementary forms of intraquery parallelism:

Intraoperation Parallelism

- parallelize the execution of each individual operation in the query
- Supports high degree of parallelism

Interoperation Parallelism

- execute the different operations in a query expression in parallel.
- Limited degree of parallelism

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture
  - n nodes, N_1, ..., N_n
    - *Each assumed to have disks and processors*.
  - Initial focus on parallelization to a shared-nothing node
    - Parallel processing within a shared memory/shared disk node discussed later
  - Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
    - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
    - However, some optimizations may be possible.

Intraquery Parallelism

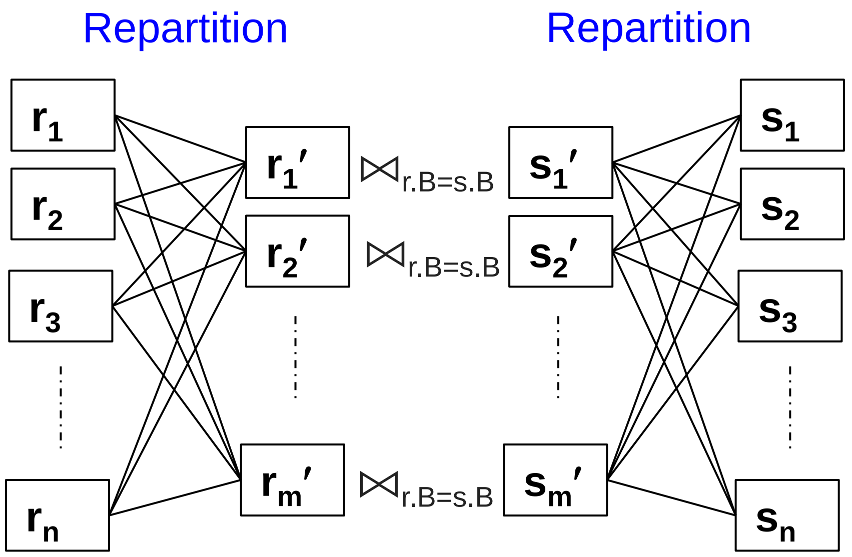
SELECT * FROM r, s where r.B = s.B

Given:
Relations r (A, D, E, B), s(A, C, B, G)
Shared nothing architecture
Nodes: $N_1, N_2, \ldots\ldots N_n$ where r and s are stored by horizontal partition vector on $A = A_1, A_2, \ldots.A_{n-1}$

r is horizontal partitioned into $r_1, r_2, \ldots. r_n$

s is horizontal partitioned into $s_1, s_2, \ldots. s_n$

Perform r $\bowtie_{r.B=s.B}$ s using m nodes.



Intraquery Parallelism

SELECT * FROM r, s where r.B = s.B

Shared nothing architecture
Nodes: $N_1, N_2, \ldots.. N_n$ where r and s are stored by horizontal partition vector on $A = A_1, A_2, \ldots.A_n$
r is horizontal partitioned into $r_1, r_2, \ldots. r_n$
s is horizontal partitioned into $s_1, s_2, \ldots. s_n$

Perform r r $\bowtie_{r.B=s.B}$ s s using **m** nodes.
Partition vector: $B_1, B_2, \ldots.. B_{m-1}$

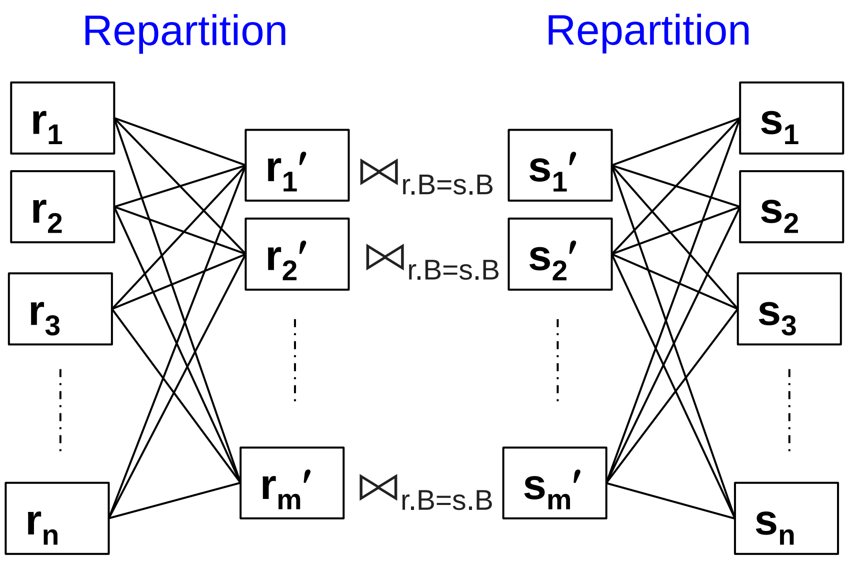
Steps to process the query

Step 1:
Repartition $r_1, r_2, \ldots. r_n$ into new partitions $r_1', r_2', \ldots. r_m'$

Repartition $s_1, s_2, \ldots. s_n$ into new partitions $s_1', s_2', \ldots. s_m'$

Step 2:
Assign $r_1', r_2', \ldots. r_m'$ into nodes $N_1, N_2, \ldots.. N_m$

Assign $s_1', s_2', \ldots. s_m'$ into nodes $N_1, N_2, \ldots.. N_m$



Intraquery Parallelism

SELECT * FROM r, s where r.B = s.B

Shared nothing architecture
Nodes: N_1, N_2, ….. N_n where r and s are stored by horizontal partition vector on $A = A_1, A_2, \ldots.A_n$
r is horizontal partitioned into r_1, r_2, …. r_n
s is horizontal partitioned into s_1, s_2, …. s_n

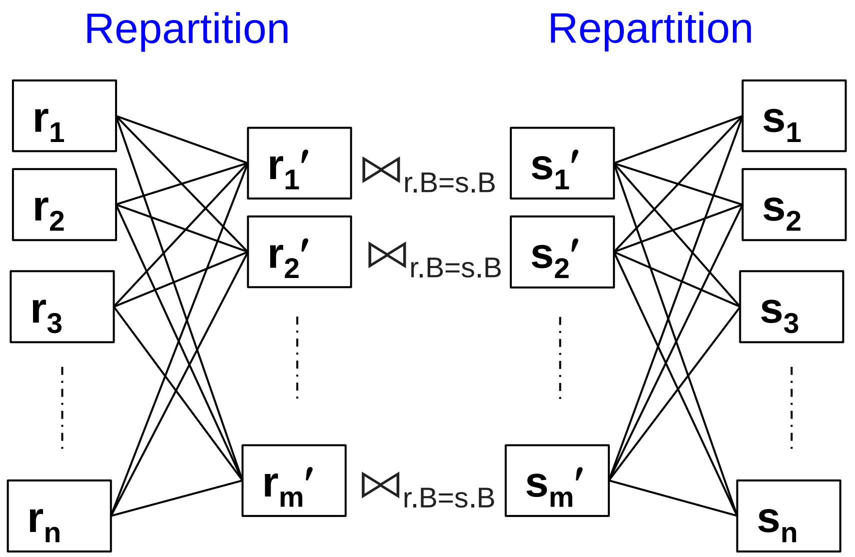
Perform r r $\bowtie_{r.B=s.B}$ s s using **m** nodes.

Steps to process the query
Step 3:
Perform $r_1' \bowtie s_1'$ at node N_1

Perform $r_2' \bowtie s_2'$ at node N_2

…………………

Perform $r_m' \bowtie s_m'$ at node N_m



Intraquery Parallelism

Question 9-1

Student (id, name,cgpa, yearAdmit)
Takes(id, course-id, semester, year)

SELECT r.id, course-id, CGPA FROM student r, takes s where r.yearAdmit = s.year

Shared nothing architecture

Nodes: N_1, N_2, ….. N_{10} where r and s are stored by horizontal partition vector on id = 121000, 131000, 141000, 151000, 161000, 171000, 181000, 191000, 201000

Total 10 partitions into 10 nodes

r is horizontal partitioned into r_1, r_2, .. r_{10}
s is horizontal partitioned into s_1, s_2, . s_{10}
The range of yearAdmit is 2015 to 2021

a. Find partitions of student and takes using yearAdmit
b. Perform r r $\bowtie_{yearAdmit=year}$ s using **6** nodes.

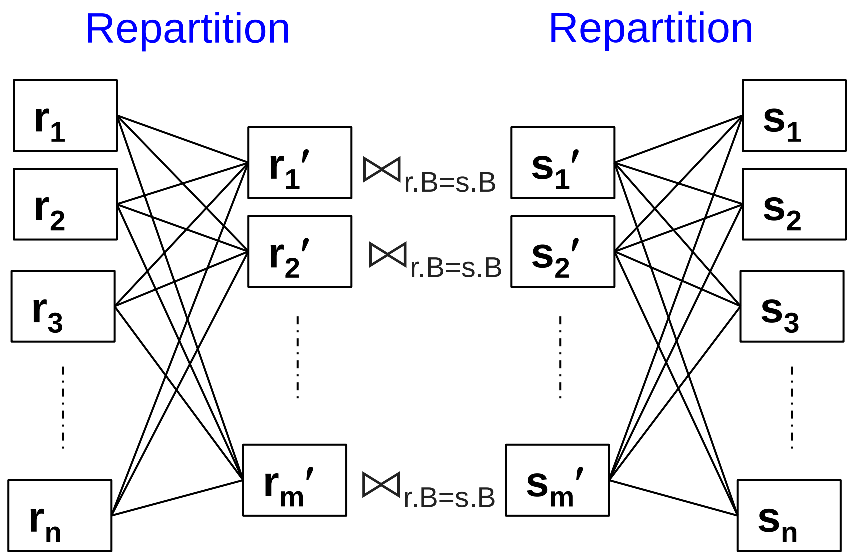
Partitions

r_1 = all *students* id < 121000
s_1 = all *takes* id < 121000

r_2 = all *students* 121000 <= id < 131000
s_2 = all *takes* 121000 <= id < 131000

………

r_{10} = all *students* id >= 201000
s_{10} = all *takes* id >= 201000



Intraquery Parallelism

SELECT * FROM r, s where r.A = s.A

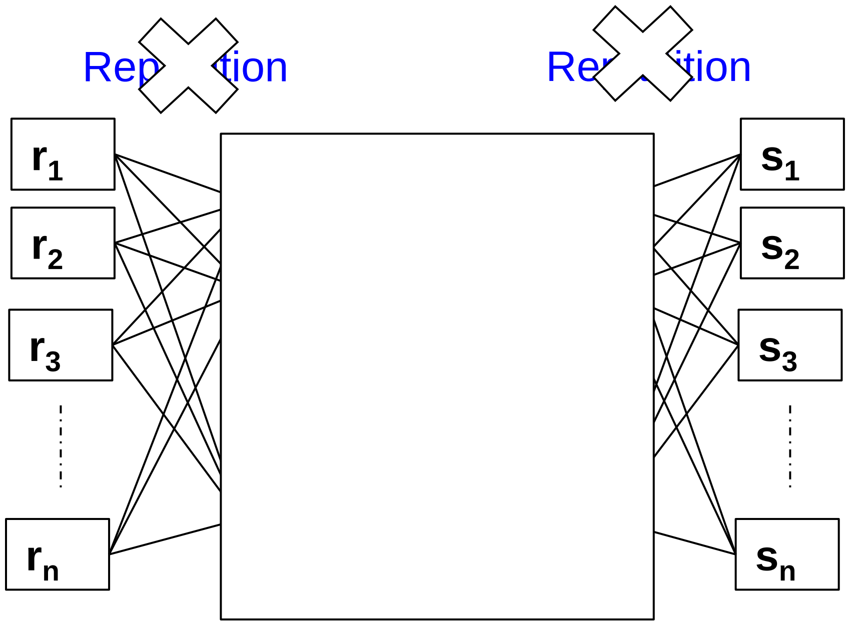
Given:
Relations r (<u>A</u>, D, E, B), s(<u>A, C</u>, B, G)
Shared nothing architecture
Nodes: N_1, N_2, ….. N_n where r and s are stored by horizontal partition vector on $A = A_1, A_2, \dots A_{n-1}$

r is horizontal partitioned into r_1, r_2, …. r_n

s is horizontal partitioned into s_1, s_2, …. s_n

Perform r $\bowtie_{r.A=s.A}$ s using n nodes.

No Need Repartition



Case 2:
No. of portioning node = number of query node
Partitioning attribute and query attribute is same.

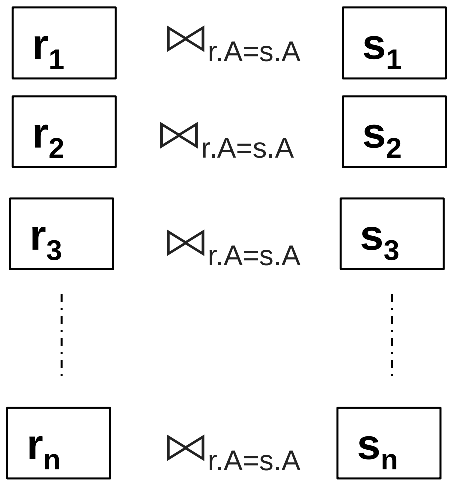
Intraquery Parallelism

SELECT * FROM r, s where r.A = s.A

Given:
Relations r (<u>A</u>, D, E, B), s(<u>A, C</u>, B, G)
Shared nothing architecture
Nodes: N_1, N_2, ….. N_n where r and s are stored by horizontal partition vector on $A = A_1, A_2, \dots A_{n-1}$

r is horizontal partitioned into $r_1, r_2, \dots r_n$

s is horizontal partitioned into $s_1, s_2, \dots s_n$



$r_1 \bowtie_{r.A=s.A} s_1$
$r_2 \bowtie_{r.A=s.A} s_2$
$r_3 \bowtie_{r.A=s.A} s_3$
$r_n \bowtie_{r.A=s.A} s_n$

Perform $r \bowtie_{r.A=s.A} s$ using n nodes.

Intraquery Parallelism

Example
Student (id, name,cgpa, yearAdmit)
Takes(id, course-id, semester, year)

SELECT r.id, course-id FROM student r, takes s where r.id = s.id
Shared nothing architecture
Nodes: N_1, N_2, ….. N_{10} where r and s are stored by horizontal partition vector on id = 121000, 131000, 141000, 151000, 161000, 171000, 181000, 191000, 201000
Total 10 partitions into 10 nodes
r is horizontal partitioned into $r_1, r_2, . . . r_{10}$
s is horizontal partitioned into $s_1, s_2,.. s_{10}$

- Find partitions of student and takes
- Perform r $\bowtie_{r.id = s.id}$ s using **10** nodes.

Partitions

r_1 = all *students* id < 121000
s_1 = all *takes* id < 121000

r_2 = all *students* 121000 <= id < 131000
s_2 = all *takes* 121000 <= id < 131000

………
r_{10} = all *students* id >= 201000
s_{10} = all *takes* id >= 201000

Steps to process the query
Only one Step:

Perform $r_1 \bowtie s_1$ at node N_1

Perform $r_2 \bowtie s_2$ at node N_2

Perform $r_3 \bowtie s_3$ at node N_3

Perform $r_4 \bowtie s_4$ at node N_4

………

Perform $r_{10} \bowtie s_{10}$ at node N_{10}

Intraquery Parallelism

Question 9-2

customer (id, name,type, country)

purchase(id, product-id, p-country, date)

SELECT r.id, product-id FROM customer r, purchase s where r.id = s.id

Shared nothing architecture

Nodes: N_1, N_2, ….. N_5 where r and s are stored by horizontal partition vector on id = 1000, 2000, 3000, 4000

Total 5 partitions into 5 nodes

r is horizontal partitioned into r_1, r_2, …. r_5

s is horizontal partitioned into s_1, s_2, …. s_5

a. Perform $r \bowtie_{r.id = s.id} s$ using **5** nodes.
b. Why is repartition not needed?

Sorting

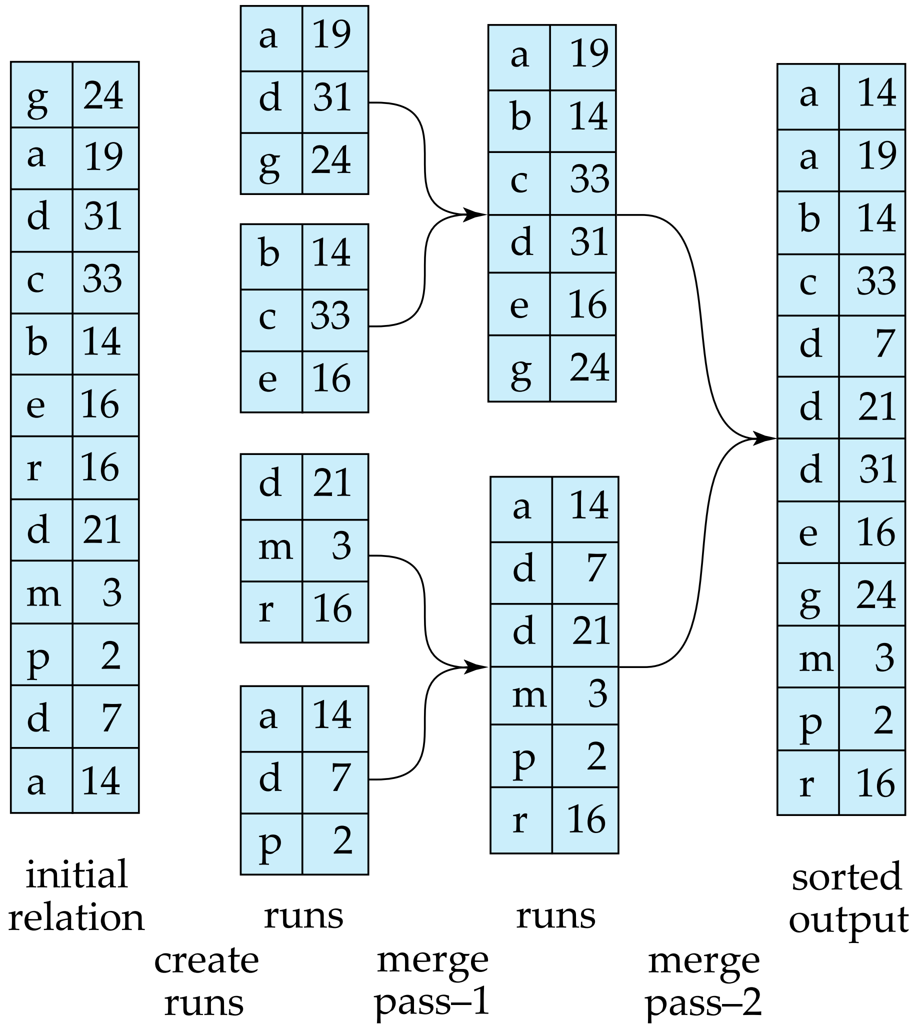
- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.

Discussion: The case when it May lead to one disk block access for each tuple.

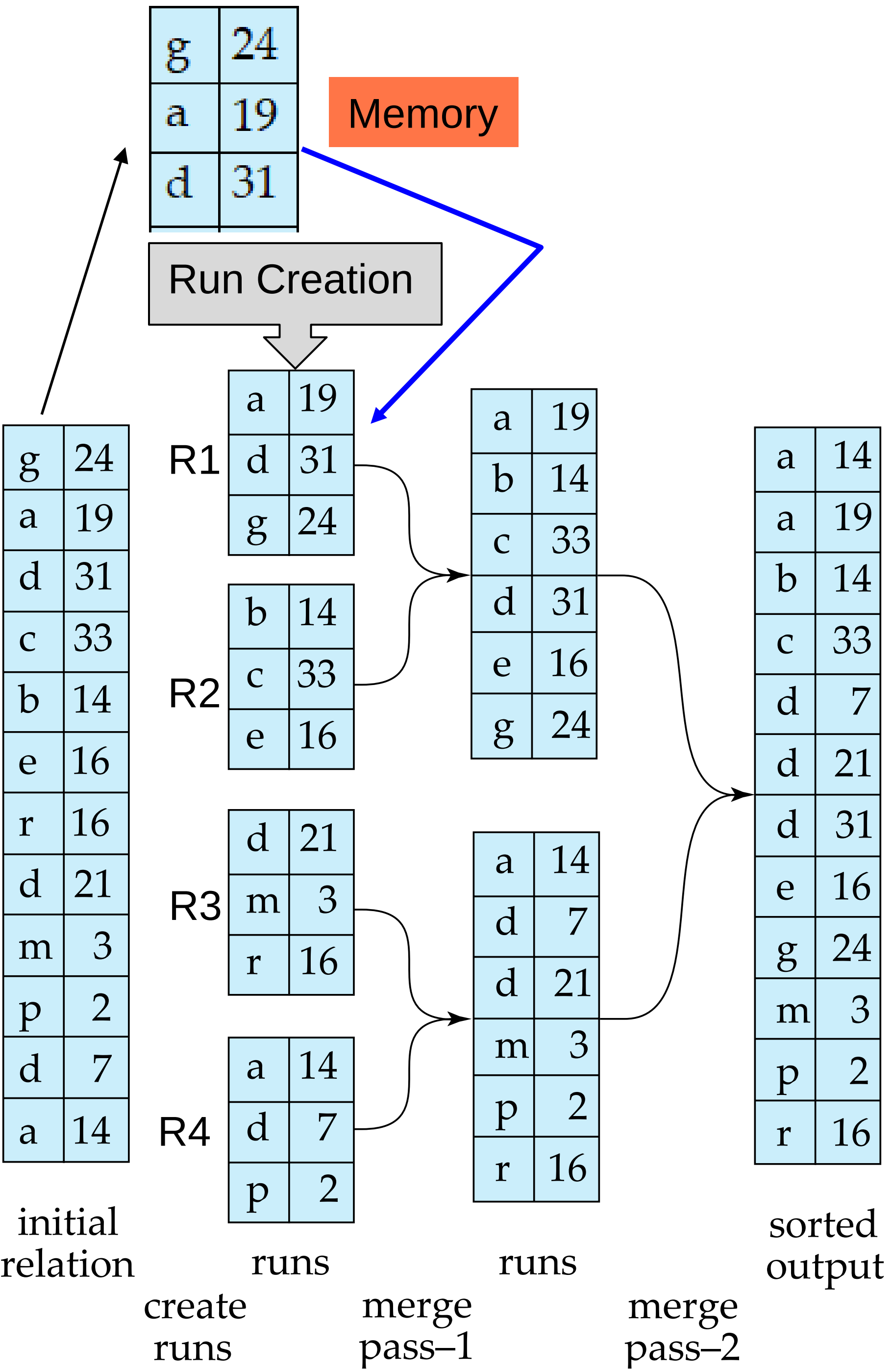
- For relations that fit in memory, techniques like quicksort can be used.

- For relations that don't fit in memory, quicksort is not applicable. Why? **external sort-merge** is a good choice in this case.

Example: External Sorting Using Sort-Merge



External Sort-Merge (Run Creation)



Let *M* denote memory size (in pages).
Here M = 3

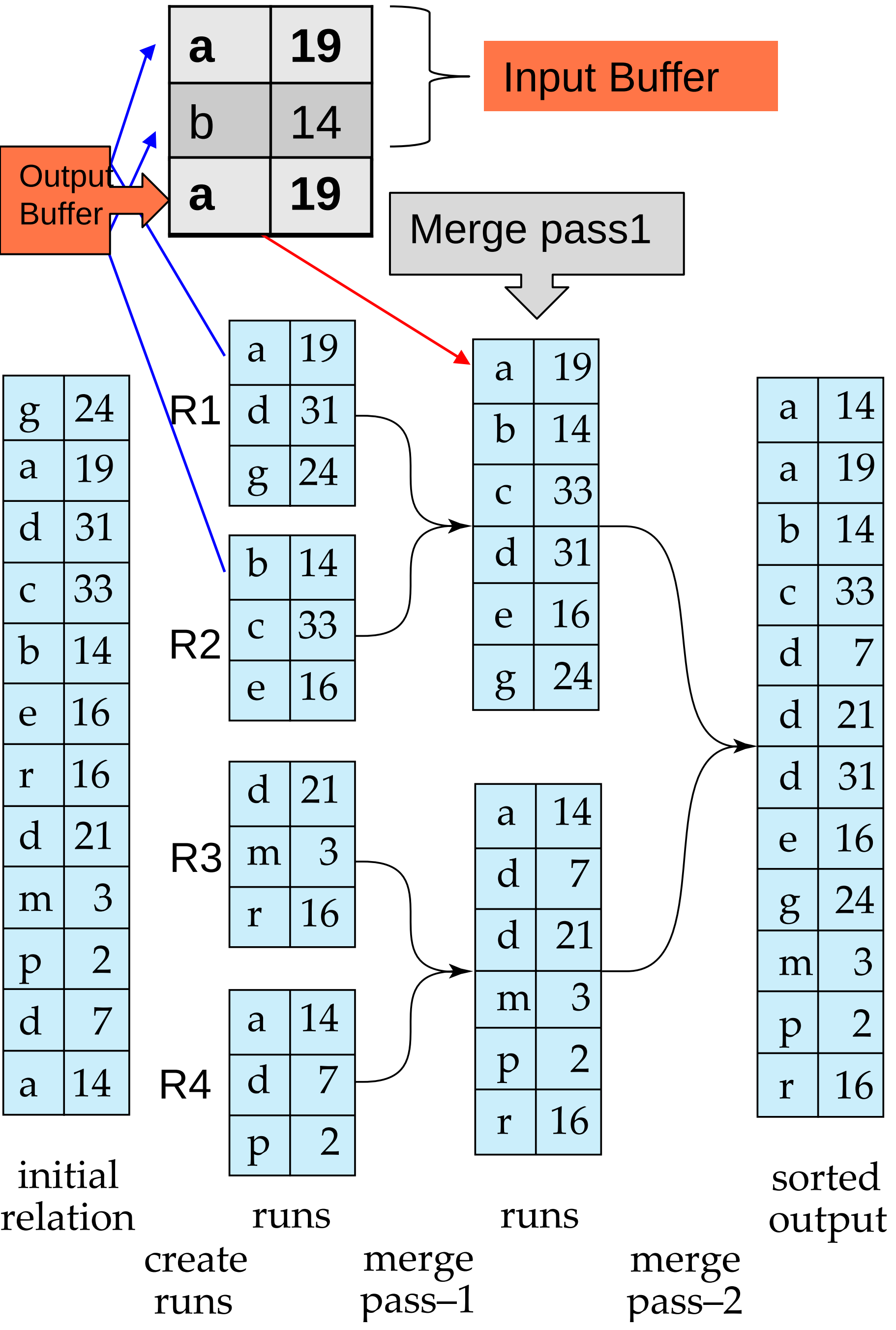
1. **Create sorted runs**. Let *i* be 1 initially.
   Repeatedly do the following till the end of the relation:
   (a) Read *M* blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment *i.*

Let the final value of *i* be *N* ***(Here N = ?)***

Here N = 4

2. *Merge the runs (next slide)…..*

External Sort-Merge (Sort-Merge)



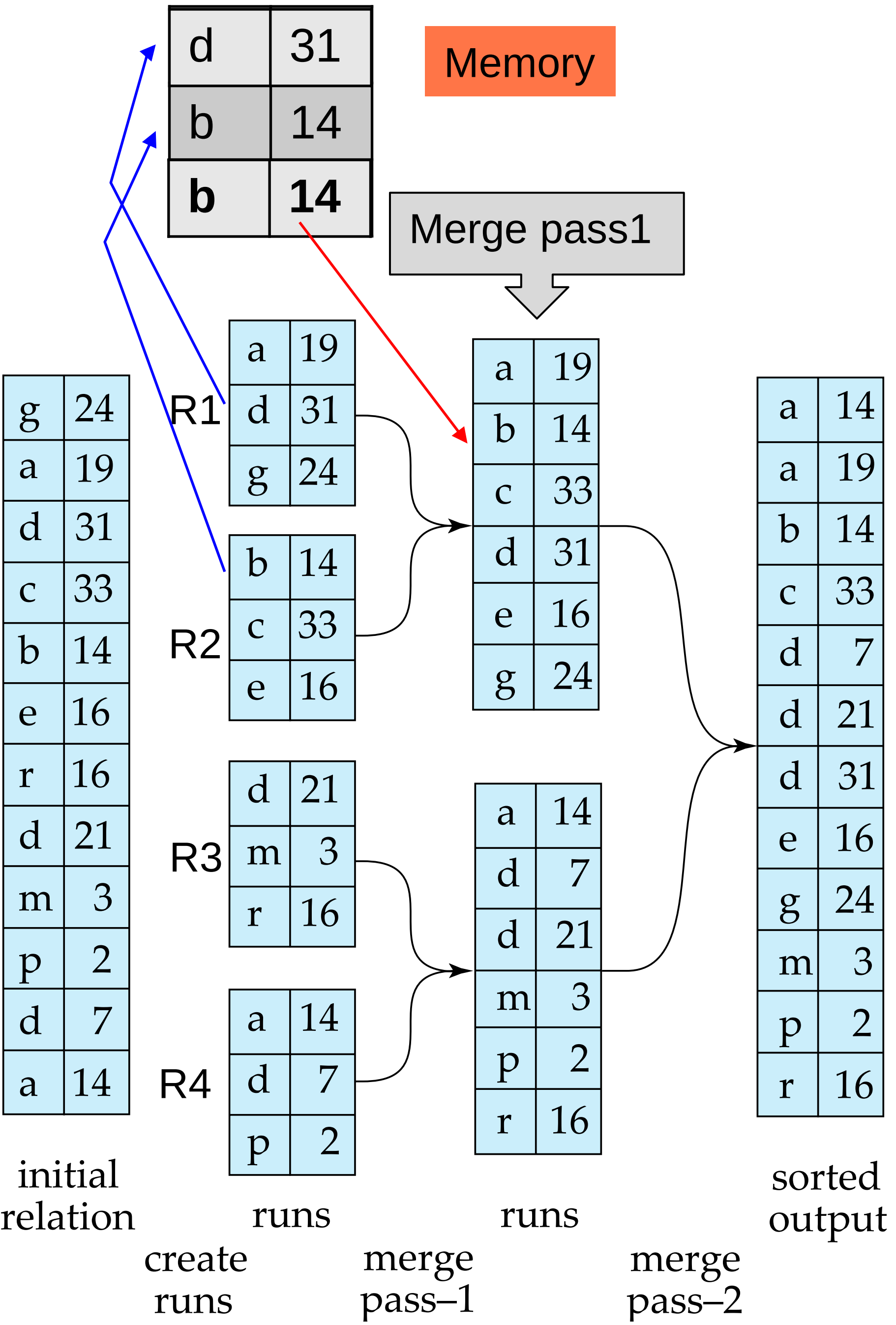
a 19
b 14
a 19
Input Buffer
Output Buffer
Merge pass1
R1
R2
R3
R4
g 24 | a 19 | d 31 | c 33 | b 14 | e 16 | r 16 | d 21 | m 3 | p 2 | d 7 | a 14
a 19 | d 31 | g 24
b 14 | c 33 | e 16
d 21 | m 3 | r 16
a 14 | d 7 | p 2
a 19 | b 14 | c 33 | d 31 | e 16 | g 24
a 14 | d 7 | d 21 | m 3 | p 2 | r 16
a 14 | a 19 | b 14 | c 33 | d 7 | d 21 | d 31 | e 16 | g 24 | m 3 | p 2 | r 16
initial relation
runs
runs
sorted output
create runs
merge pass–1
merge pass–2

Merge the runs (2-way merge). Here *N* > *M*. M=3, N=4

Use *2* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page

```
repeat
    Select the first record (in sort order) among all buffer pages
    Write the record to the output buffer.  If the output buffer is full write it to disk.
    Delete the record from its input buffer page.
    If the buffer page becomes empty then
        read the next block (if any) of the run into the buffer.
until all input buffer pages are empty:
```

External Sort-Merge (Sort Merge)

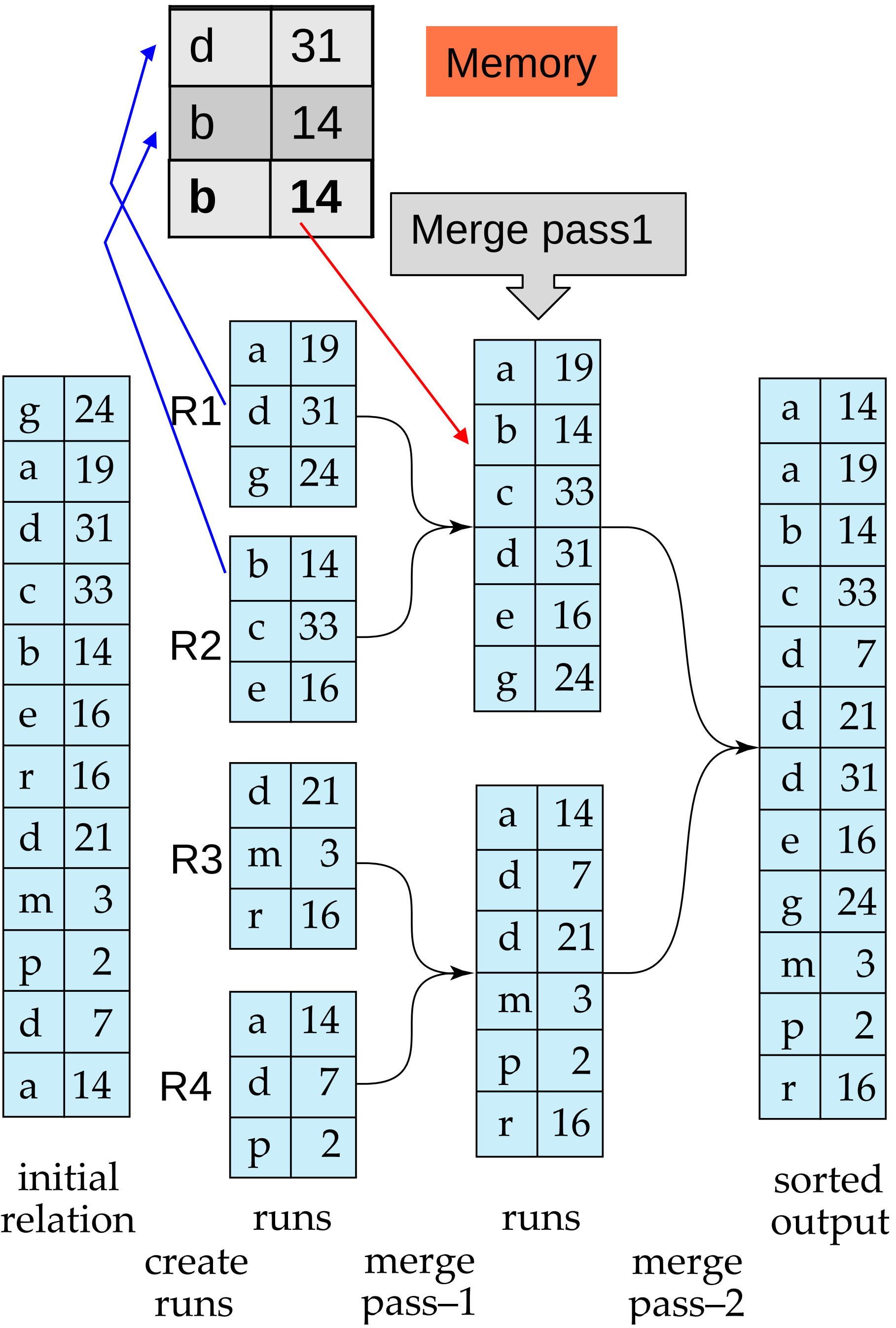

d 31
b 14
b 14
Memory
Merge pass1
g 24, a 19, d 31, c 33, b 14, e 16, r 16, d 21, m 3, p 2, d 7, a 14
R1: a 19, d 31, g 24
R2: b 14, c 33, e 16
R3: d 21, m 3, r 16
R4: a 14, d 7, p 2
a 19, b 14, c 33, d 31, e 16, g 24
a 14, d 7, d 21, m 3, p 2, r 16
a 14, a 19, b 14, c 33, d 7, d 21, d 31, e 16, g 24, m 3, p 2, r 16
initial relation
create runs
runs
merge pass–1
runs
merge pass–2
sorted output

Merge the runs (2-way merge). Here *N* > *M*. M=3, N=4

Use *2* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page

```
repeat
    Select the first record (in sort order) among all buffer pages
    Write the record to the output buffer.  If the output buffer is full write it to disk.
    Delete the record from its input buffer page.
    If the buffer page becomes empty then
        read the next block (if any) of the run into the buffer.
until all input buffer pages are empty:
```

External Sort-Merge (Sort Merge)



Memory
d 31
b 14
b 14
Merge pass1
R1
R2
R3
R4
g 24, a 19, d 31, c 33, b 14, e 16, r 16, d 21, m 3, p 2, d 7, a 14
a 19, d 31, g 24
b 14, c 33, e 16
d 21, m 3, r 16
a 14, d 7, p 2
a 19, b 14, c 33, d 31, e 16, g 24
a 14, d 7, d 21, m 3, p 2, r 16
a 14, a 19, b 14, c 33, d 7, d 21, d 31, e 16, g 24, m 3, p 2, r 16
initial relation
create runs
runs
merge pass–1
runs
merge pass–2
sorted output

Question 10-1: In the given example, N =4 and M = 2.

a. Find the size of the memory when the external sort can be done in one pass.

b. Find the size of the memory when the quicksort can be used.

c. Explain the impact of memory size in database sort performance.

Parallel Sort

- Suppose that we wish to sort a relation r that resides on n nodes N1, N2, … , Nn.
- If the relation has been range-partitioned on the attributes on which it is to be sorted, we can sort each partition separately and concatenate the results to get the full sorted relation.
- Since the tuples are partitioned on n nodes, the time required for reading the entire relation is reduced by a factor of **n** by the parallel access.

Question 10-2:

Given relation

Person(NID, name, street, city)

The relation has 160 million tuples and is range partitioned into 160 nodes using NID. You have to process the following queries:

a. Find the list of persons sorted by NID in ascending order
b. Find the list of persons sorted by district in alphabetic order

Explain how the above two queries will be processed.

Parallel Sort

- If relation *r* has been partitioned in any other way, we can sort it in one of two ways:
  a. We can range-partition *r* on the sort attributes, and then sort each partition separately.
  b. We can use a parallel version of the external sort-merge algorithm.

Question 10-2:

Given relation

Person(NID, name, street, city)

The relation has 160 million tuples and is range partitioned into 160 nodes using NID. You have to process the following queries:

a. Find the list of persons sorted by NID in ascending order
b. Find the list of persons sorted by district in alphabetic order

Explain how the above two queries will be processed.

Range-Partitioning Sort

The picture can't be displayed.

- Choose nodes N_1, ..., N_m, where $m \leq n\text{-}1$ to do sorting.
- Create range-partition vector with m-1 entries, on the sorting attributes
- Redistribute the relation using range partitioning
- Each node N_i sorts its partition of the relation locally.
  - Example of **data parallelism: e**ach node executes same operation in parallel with other nodes, without any interaction with the others.
- Final merge operation is trivial: range-partitioning ensures that, if i < *j*, all key values in node N_i are all less than all key values in N_j.

Parallel External Sort-Merge

- Assume the relation has already been partitioned among nodes N_1, ..., N_n (in whatever manner).
- Each node N_i locally sorts the data (using local disk as required)
- The sorted runs on each node are then merged in parallel:
  - The sorted partitions at each node Ni are range-partitioned across the processors N_1, ..., N_m.
  - Each node Ni performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on nodes N_1, ..., N_m are concatenated to get the final result.

Parallel External Sort-Merge (Cont..)

The picture can't be displayed.

- Algorithm as described vulnerable to execution skew
  - all nodes send to node 1, then all nodes send data to node 2, …
  - Can be modified so each node sends data to all other nodes in parallel (block at a time)

Interoperator Parallelism

Pipelined parallelism

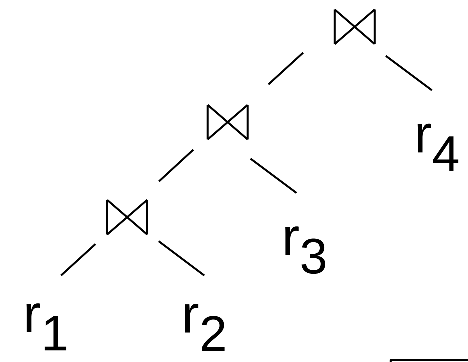
Consider a join of four relations

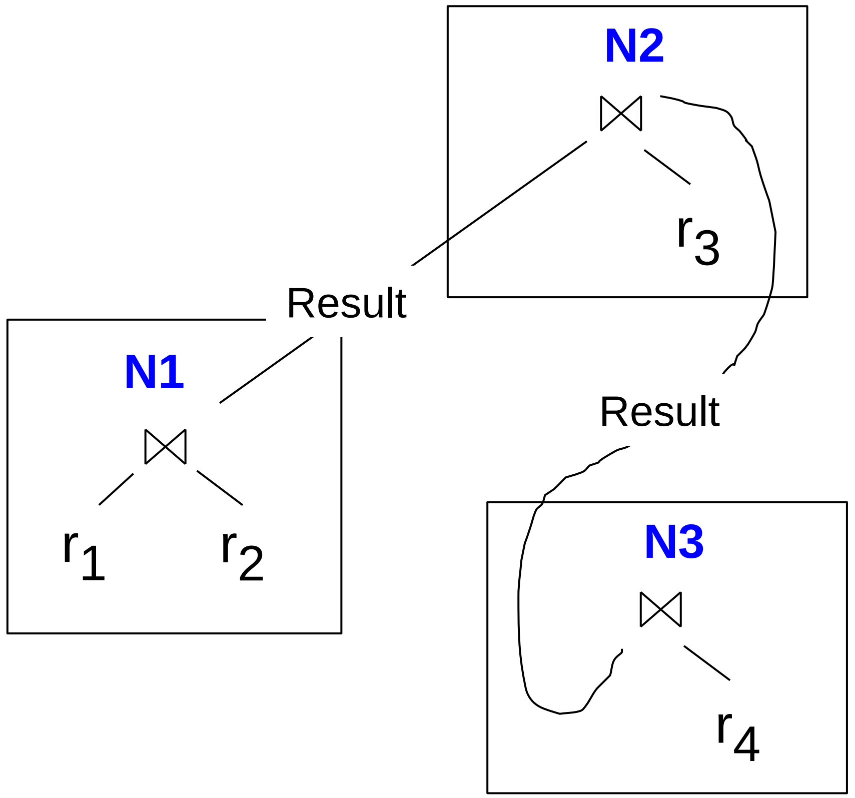
$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

Set up a pipeline that computes the three joins in parallel

Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results

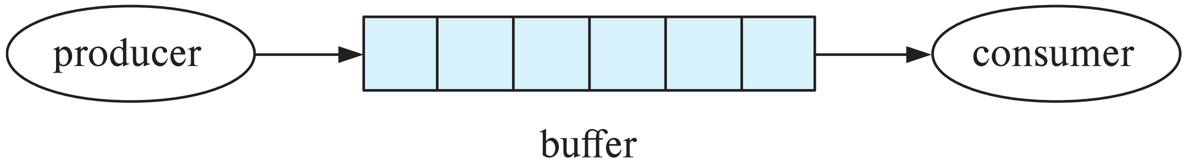
Provided a pipelineable join evaluation algorithm (e.g. indexed nested loops join) is used



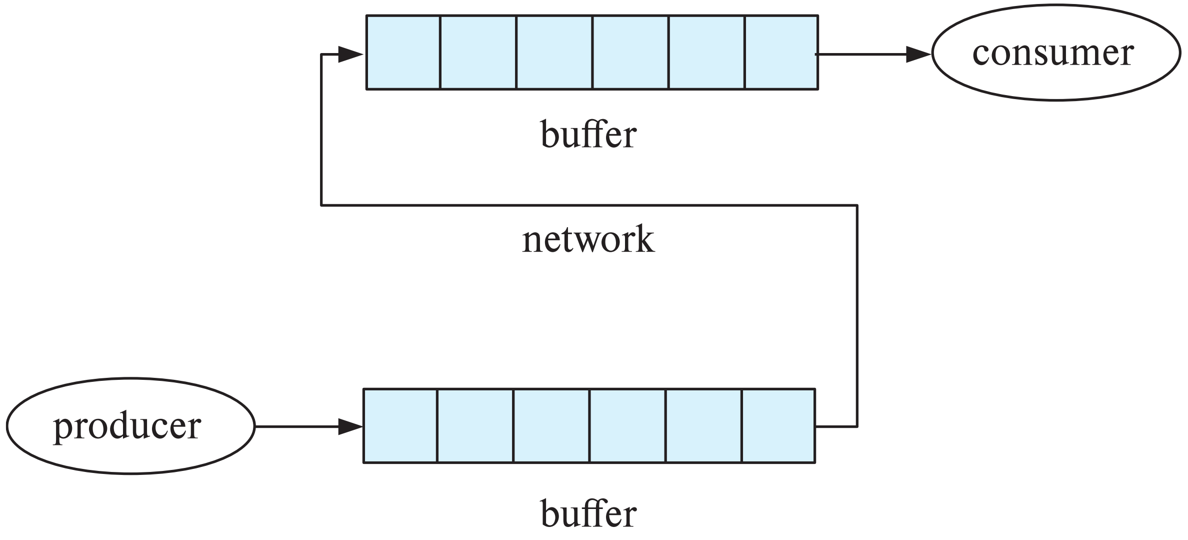


Pipelined Parallelism

- Push model of computation appropriate for pipelining in parallel databases
- Buffer between consumer and producer
- Can batch tuples before sending to next operator
  - Reduce number of messages,
  - reduce contention on shared buffers

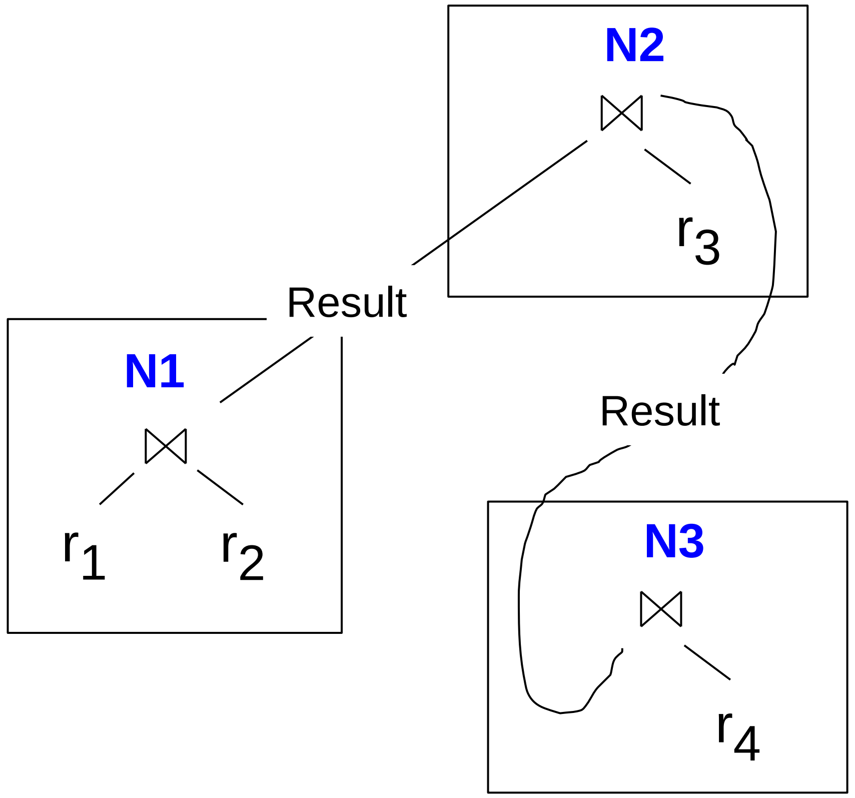


(a) Producer-consumer in shared memory

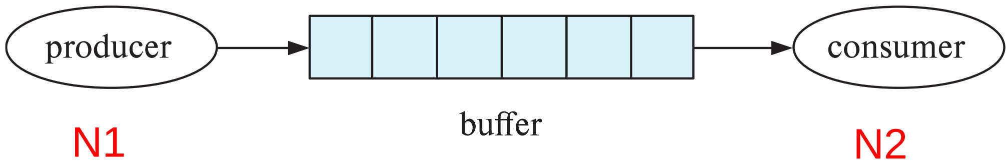


(b) Producer-consumer across a network

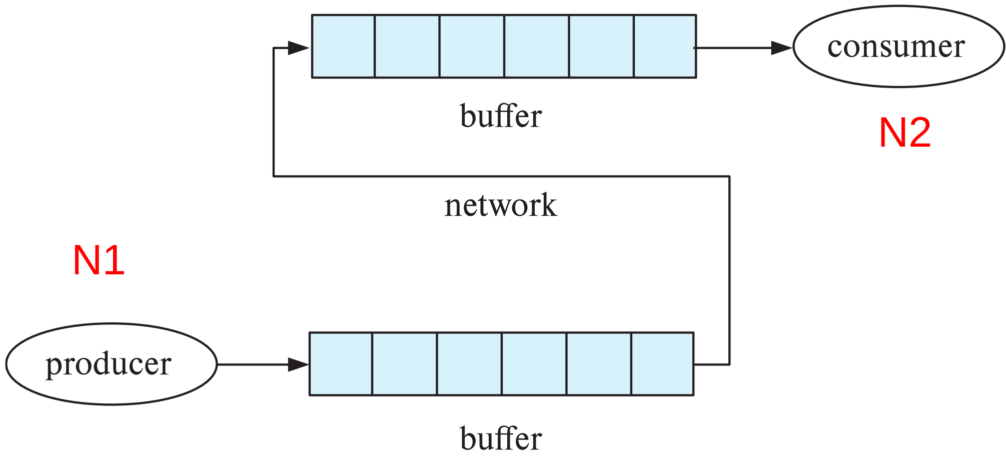
Pipelined Parallelism



Producer-consumer for N1 to N2

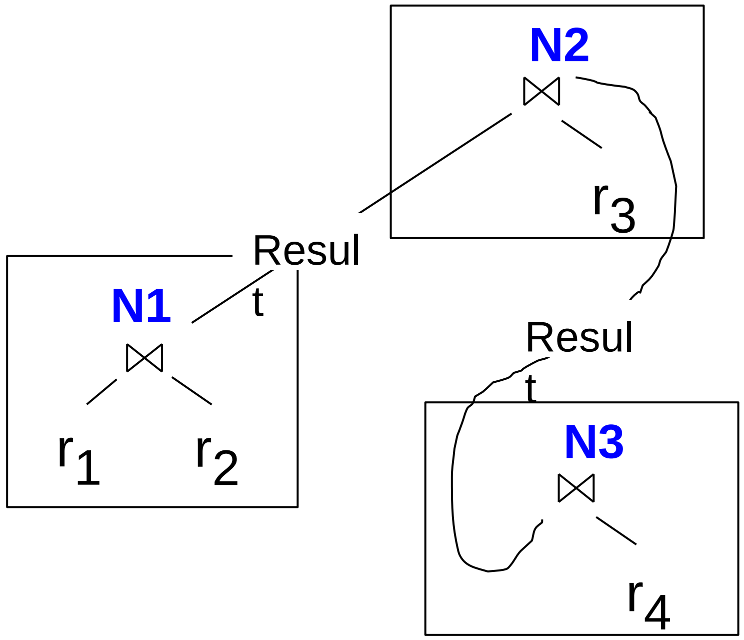


(a) Producer-consumer in shared memory

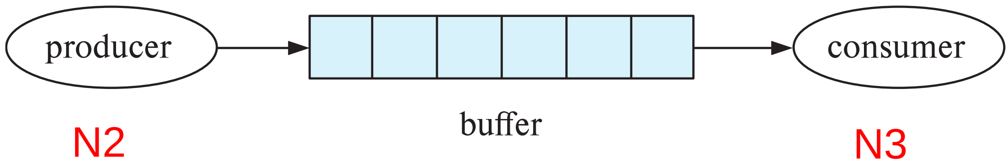


(b) Producer-consumer across a network

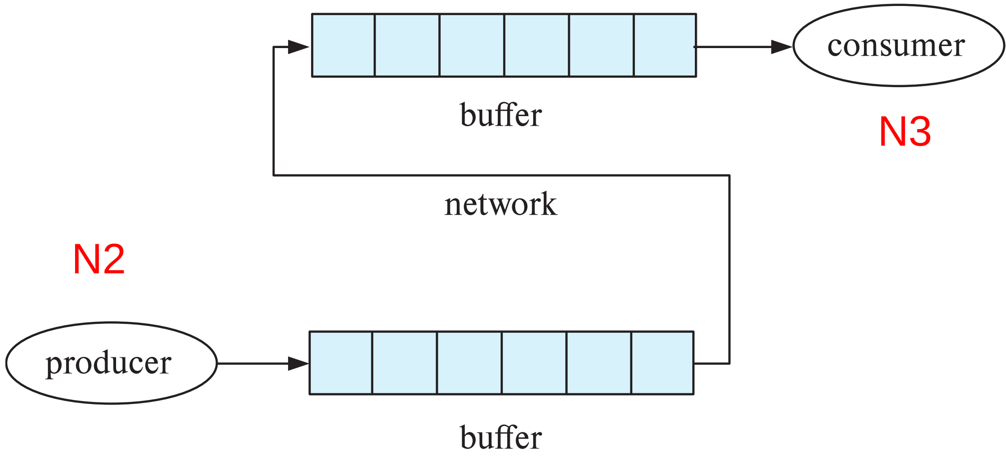
Pipelined Parallelism



Producer-consumer for N2 to N3



(a) Producer-consumer in shared memory

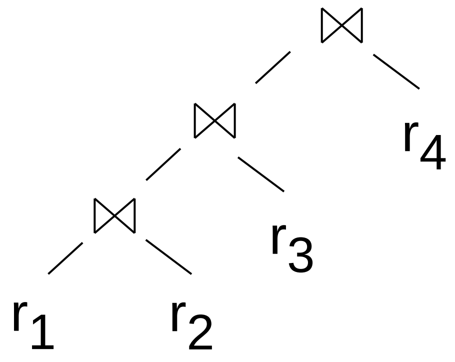


(b) Producer-consumer across a network

Question 11-1:

a. The relations r1, r2, r3 and r4 are partitioned into 30 nodes N1, N2, …. N30. Propose a query execution plan using itra-operation parallelism for individual joins and inter-operation pipeline parallelism.
b. Show the diagram.

Utility of Pipeline Parallelism



Limitations

1. Does not provide a high degree of parallelism since pipeline chains are not very long (**Explain?**)

2. Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort) (**Explain?**)

3. Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others **(Explain?)**.

- But pipeline parallelism is still very useful since it avoids writing intermediate results to disk

How many pipelines?
Answer = 3

Explanation 2:
Query:
Select r1.id, max(r4.salary)
From r1, r2, r3, r4
Where r1.id = r2.id and r2.id = r3.id and r3.id = r4.id

Can we process max(r4.id) in pipeline? Why?

Utility of Pipeline Parallelism

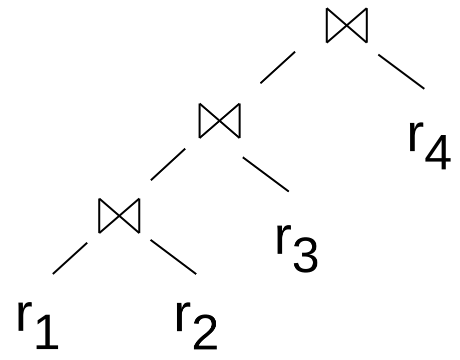
Limitations

1. Does not provide a high degree of parallelism since pipeline chains are not very long (**Explain?**)

2. Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort) (**Explain?**)

3. Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others **(Explain?)**.

- But pipeline parallelism is still very useful since it avoids writing intermediate results to disk



Explanation 3

Query:

Select r1.id, max(r4.salary)

From r1, r2, r3, r4

Where r1.id = r2.id and r2.id = r3.id and r3.id = r4.id

r1, r2 and r3 has indices and join is faster.

r4 has no index

N1 and N2 will complete the operation

N3 will be skew.

Independent Parallelism

Independent parallelism

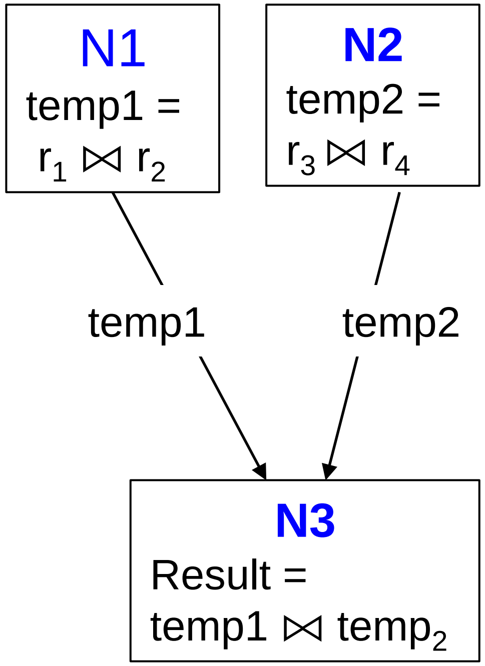
- Consider a join of four relations

  $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

- Let N_1 be assigned the computation of

  temp1 = $r_1 \bowtie r_2$

- And N_2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
- And N_3 be assigned the computation of temp1 $\bowtie$ temp_2
- N_1 and N_2 can work **independently in parallel**
- N_3 has to wait for input from N_1 and N_2
  - Can pipeline output of N_1 and N_2 to N_3, combining independent parallelism and pipelined parallelism
- Does not provide a high degree of parallelism
  - useful with a lower degree of parallelism.
  - less useful in a highly parallel system,



N1
temp1 = $r_1 \bowtie r_2$
N2
temp2 = $r_3 \bowtie r_4$
temp1
temp2
N3
Result = temp1 $\bowtie$ temp_2

Question 11-2:

a. Prepare query plan combining independent parallelism and pipeline parallelism for processing $s_1 \bowtie s_2 \bowtie s_3 \bowtie s_4 \bowtie s_5 \bowtie s_6$ where s_1, s_2, s_3, s_4, s_5 and s_6 are relations. The nodes N1, N2, N3, N4 and N5 are connected in a network and used to process the query.

b. Compare independent parallelism with pipeline parallelism.

Independent parallelism

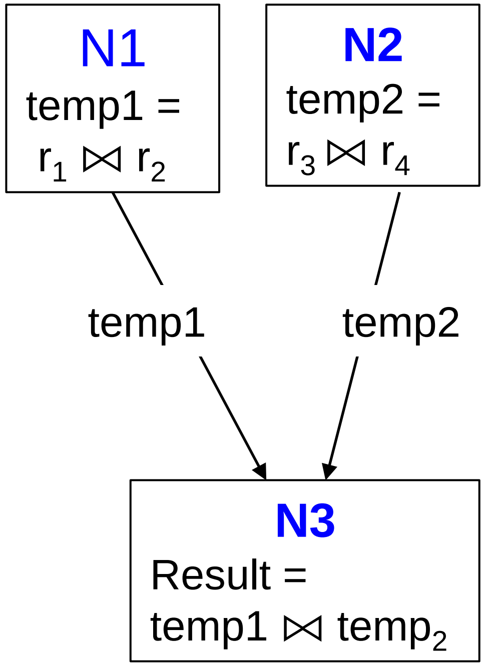
- Consider a join of four relations

  $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

- Let N_1 be assigned the computation of

  temp1 = $r_1 \bowtie r_2$

- And N_2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
- And N_3 be assigned the computation of temp1 $\bowtie$ temp_2
- N_1 and N_2 can work **independently in parallel**
- N_3 has to wait for input from N_1 and N_2
  - Can pipeline output of N_1 and N_2 to N_3, combining independent parallelism and pipelined parallelism
- Does not provide a high degree of parallelism
  - useful with a lower degree of parallelism.
  - less useful in a highly parallel system,


N1
temp1 = $r_1 \bowtie r_2$
N2
temp2 = $r_3 \bowtie r_4$
temp1
temp2
N3
Result = temp1 $\bowtie$ temp_2

Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single node.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each node whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the nodes.

Other Relational Operations (Cont.)

- **Duplicate elimination**
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash- partitioning) and perform duplicate elimination locally at each node.
- **Projection**
  - Projection without duplicate elimination can be performed as tuples are read from disk, in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- **Step 1**: Partition the relation on the grouping attributes
- **Step 2**: Compute the aggregate values locally at each node.
- **Optimization:** Can reduce cost of transferring tuples during partitioning by **partial aggregation** before partitioning
  - For distributive aggregate
  - Can be done as part of run generation
  - Consider the **sum** aggregation operation:
    - Perform aggregation operation at each node N_i on those tuples stored its local disk
      - results in tuples with partial sums at each node.
    - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each node N_i to get the final result.
  - Fewer tuples need to be sent to other nodes during partitioning.