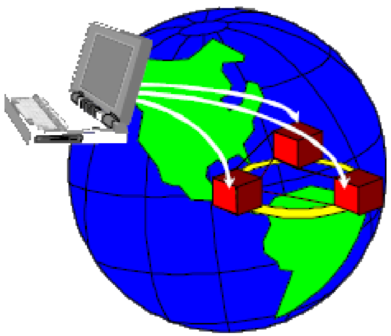
Query Processing and Optimization in Distributed Databases

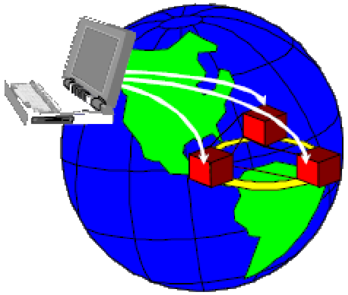


Erdem USLU

Nihal SARMAŞIK

CONTENT

- Introduction to Distributed DBMS
- Storing Data in Distributed DBMS
- Distributed Query Processing
- Cost-Based Query Optimization



Introduction to Distributed DBMS

Main Considerations:

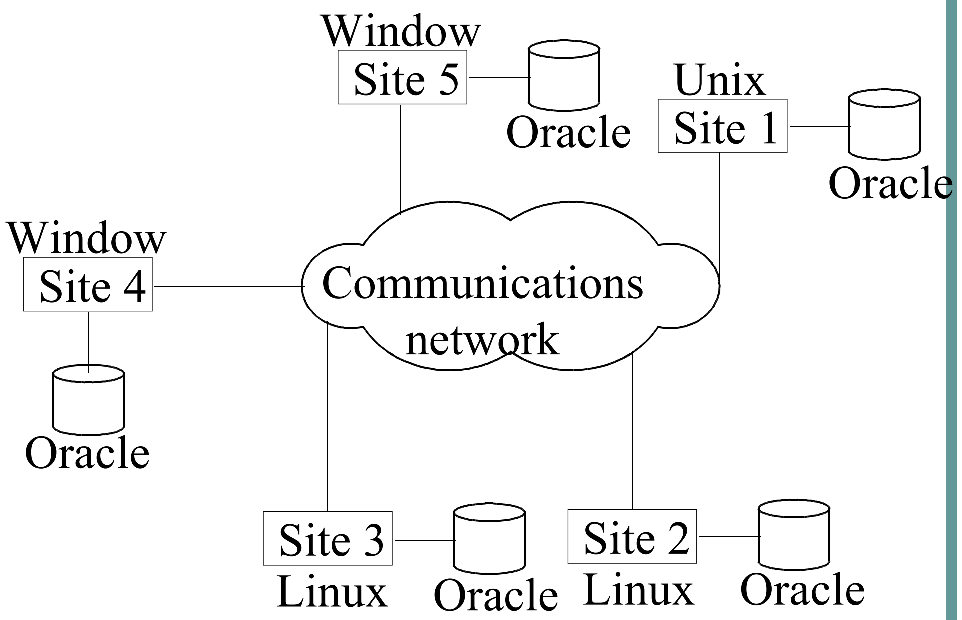
- Distributed Transaction Atomicity
  - users should be able to write transactions that access and update data at several sites just as they would write transactions over purely local data
- Location Transparency
  - user does not have to know the location of the data
  - data requests automatically forwarded to appropriate sites

Introduction to Distributed DBMS

Types of Distributed Database System

- Homogeneous Distributed Database System

  - All sites of the database system have identical setup, i.e., same database system software.
  - The underlying operating system may be different

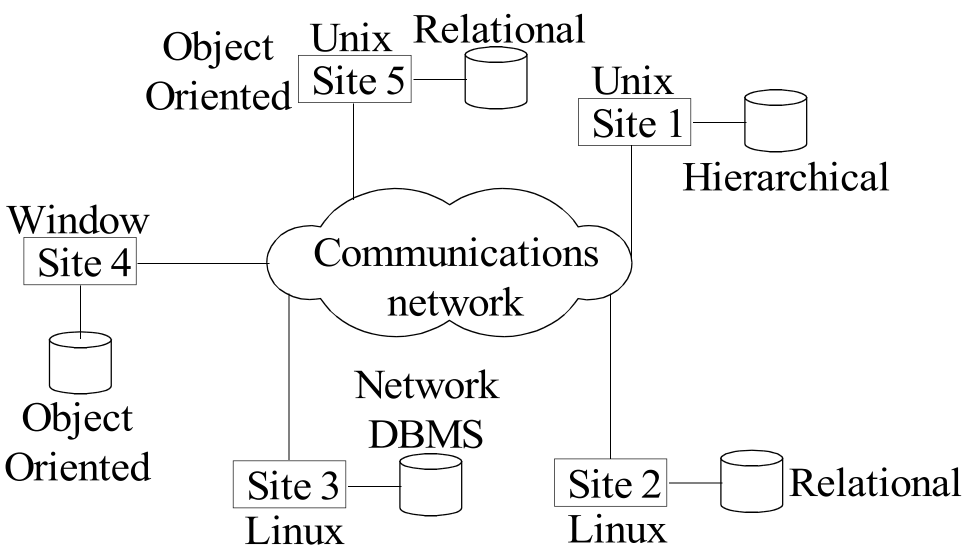


Introduction to Distributed DBMS

Types of Distributed Database System

- Heterogeneous Distributed Database System (Multidatabase System)

Each site may run different database system/software

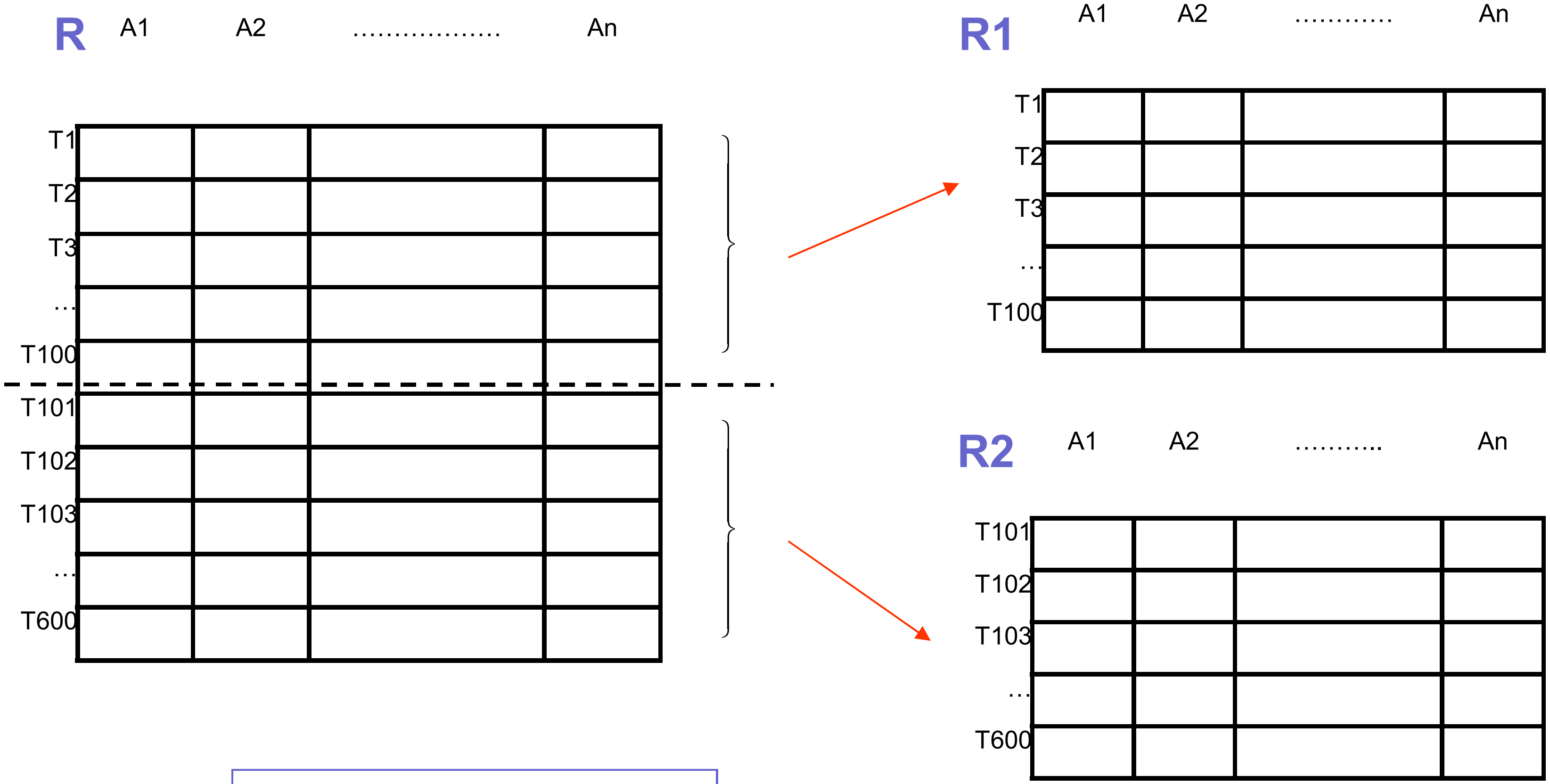


Storing Data in Distributed DBMS

- Fragmentation
  - Division of relation r into fragments *r*1, *r*2, …, *rn* which contain sufficient information to reconstruct relation r
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Fragmentation types
  - Horizontal Fragmentation
  - Vertical Fragmentation
  - Mixed Fragmentation

Storing Data in Distributed DBMS

Horizontal Fragmentation (Row-wise)

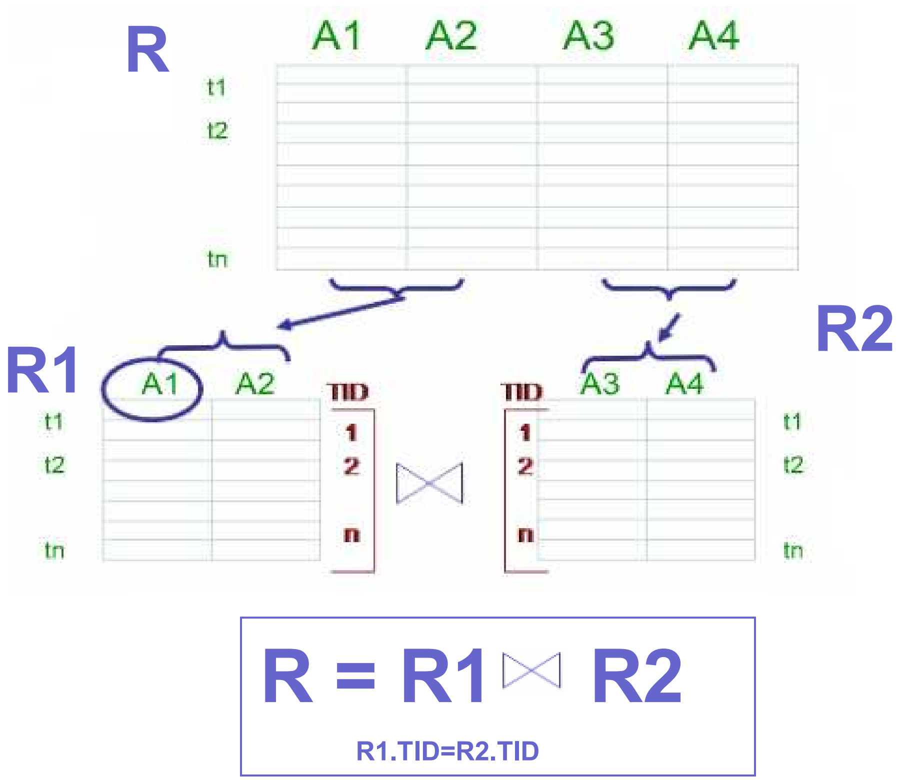


R = R1 U R2

Storing Data in Distributed DBMS

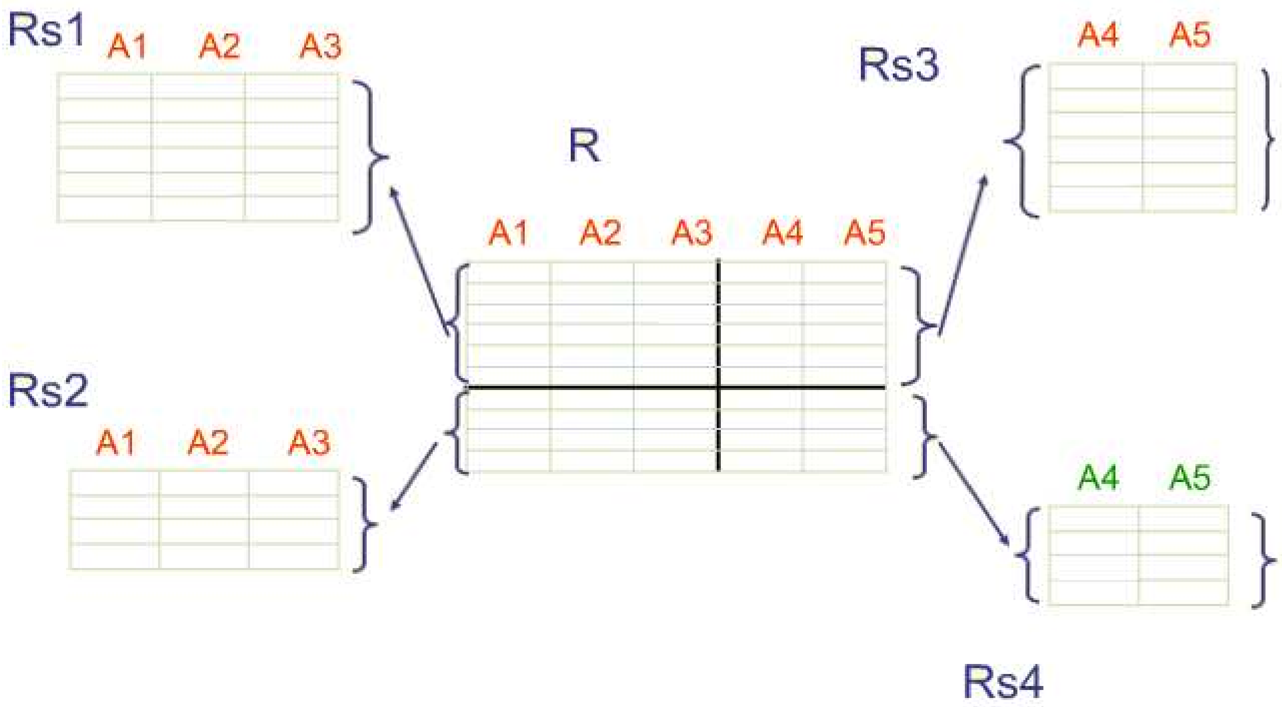
Vertical Fragmentation (Coloumn-wise)

- The schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key



Storing Data in Distributed DBMS

Mixed fragmentation



$$R = (Rs1 \cup Rs2) \bowtie (Rs3 \cup Rs4)$$

Storing Data in Distributed DBMS

- Replication
  - Multiple copies of data, stored in different sites

- Adventages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* is replicas exist
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel.
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*

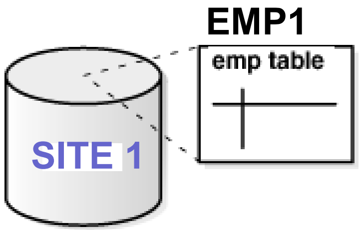
Distributed Query Processing

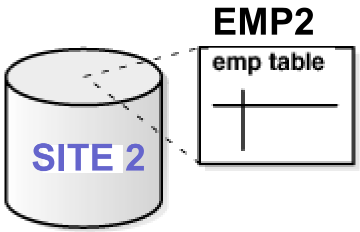
- Critera for measuring the cost of a query evalution strategy
  - For centralized DBMSs number of disk accesses (# blocks read / written)
  - For distributed databases, additionally
    - The cost of data transmission over the network
    - Potential gain in performance from having several sites processing parts of the query in parallel

Distributed Query Processing
Nonjoin Queries in Distributed DBMS

Example:

EMP = EMP1 U EMP2 (Horizontal Fragmentation)





EMP1= $\sigma_{emp\_id<5000}$(EMP)

EMP2= $\sigma_{emp\_id>=5000}$(EMP)

Distributed Query Processing

Nonjoin Queries in Distributed DBMS

QUERY: Select all employees whose ages between 20 and 30.

```
SELECT emp_id, name
FROM EMP1
WHERE age > 20 AND age < 30
UNION
SELECT emp_id, name
FROM EMP2
WHERE age > 20 AND age < 30
```

Distributed Query Processing

Nonjoin Queries in Distributed DBMS

QUERY: What is average age of all employee?

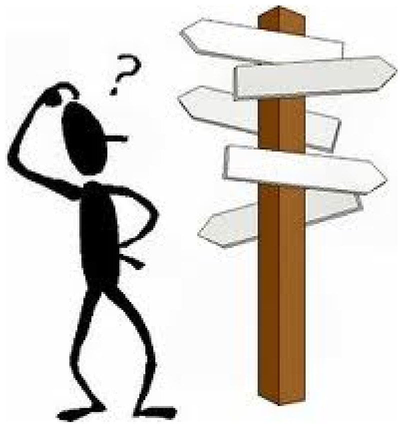
```
SELECT    (T1.age+T2.age)/
(T1.emp_number+T2.emp_number)

FROM

(SELECT SUM(age) as age, count(*) as
emp_number FROM EMP1) as T1 ,

(SELECT SUM(age) as age, count(*) as
emp_number FROM EMP2) as T2
```

Distributed Query Processing

Join Queries in Distributed DBMS

- JOIN STRATEGY
  - Ship whole
  - Fetch as needed
  - Semijoins
  - Bloomjoins

Which strategy is better for me?



Distributed Query Processing

Join Queries in Distributed DBMS (Ship Whole)

Ship Whole: Transferring the complete relation

Example:

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

$R \bowtie S$

A	B	C	D
1	1	5	1
4	5	7	8

QUERY: The query asks for $R \bowtie S$

Distributed Query Processing

Join Queries in Distributed DBMS (Ship whole)

COST ANALYSIS

- When execution at nodeR
  - nodeR: send data request message (relation S) to nodeS
  - nodeS: send requested data (relation S) to nodeR

Total costs: 2 messages, 18 attribute value

- When execution at nodeS
  - nodeS: send data request message (relation R) to nodeR
  - nodeR: send requested data (relation R) to nodeS

Total costs: 2 messages, 14 attribute value

Distributed Query Processing

Join Queries in Distributed DBMS (Fetch As Needed)

Fetch As Needed: Transferring the relation piecewise

Example:

R

A	B
3	7
1	1
4	6
7	7
4	5
6	2
5	7

S

B	C	D
9	8	8
1	5	1
9	4	2
4	3	3
4	2	6
5	7	8

$R \bowtie S$

A	B	C	D
1	1	5	1
4	5	7	8

QUERY: The query asks for $R \bowtie S$

Distributed Query Processing

Join Queries in Distributed DBMS (Fetch As Needed)

COST ANALYSIS

- When execution at nodeR
  - **nodeR:** send data request message (tuples of relation S with B = ‘7’) to nodeS
  - **nodeS:** send requested data (0 tuples of relation S with B = ‘7’) to nodeR
  - **nodeR:** send data request message (tuples of relation S with B = ‘1’) to nodeS
  - **nodeS:** send requested data (1 tuple of relation S with B = ‘1’) to nodeR

  . . .

  Total costs: 7 * 2 = 14 messages, 7 + 2 * 3 = 13 attribute value

- When execution at nodeS
  - **nodeS:** send data request message (tuples of relation S with B = ‘9’) to nodeR
  - **nodeR:** send requested data (0 tuples of relation S with B = ‘9’) to nodeS
  - **nodeS:** send data request message (tuples of relation S with B = ‘1’) to nodeR
  - **nodeR:** send requested data (1 tuple of relation S with B = ‘1’) to nodeS

  . . .

  Total costs: 6 * 2 = 12 messages, 6 + 2 * 2 = 10 attribute value

Distributed Query Processing

Join Queries in Distributed DBMS

Ship Whole vs Fetch as needed:

- Fetch as needed results in

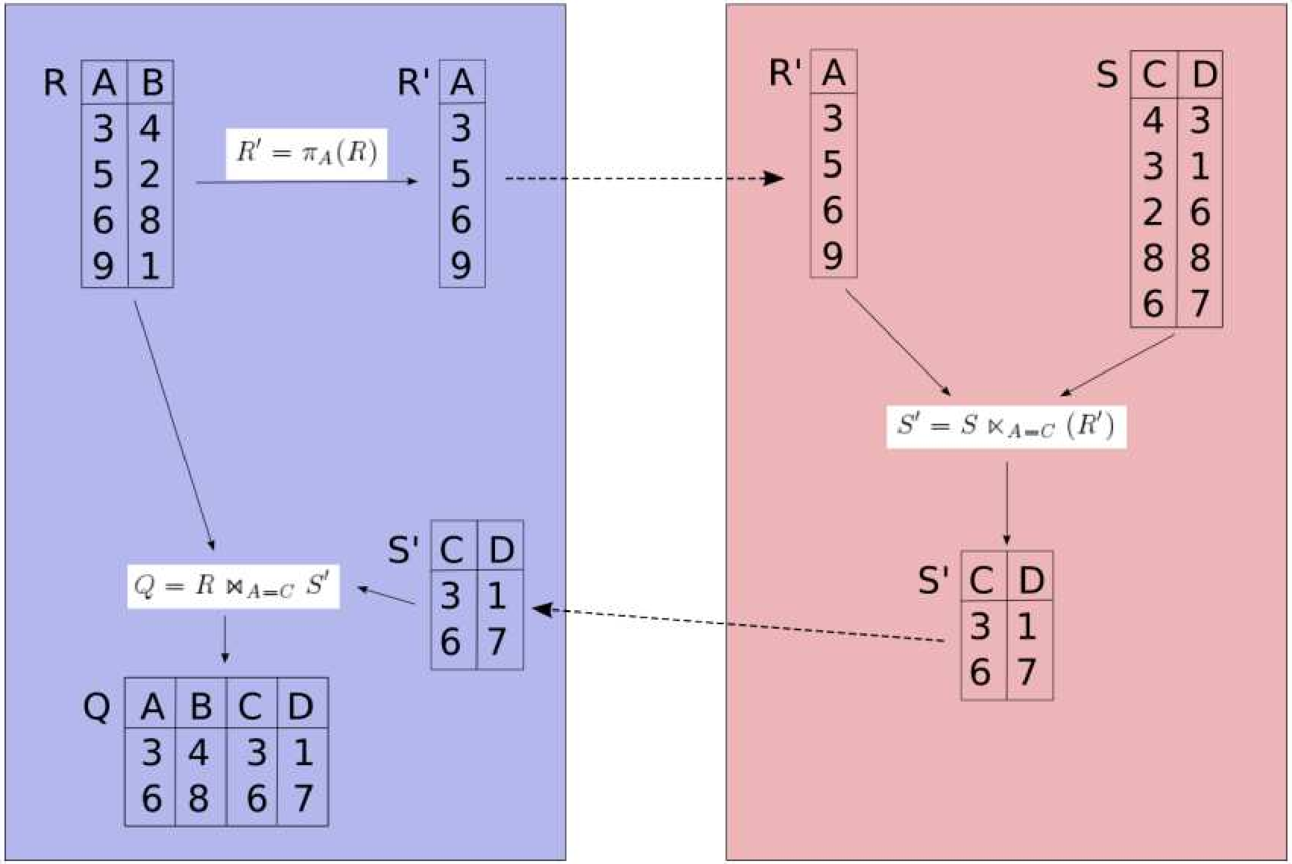
a high number of messages

- Ship whole results in

high amounts of transferred data

Distributed Query Processing
Join Queries in Distributed DBMS (Semijoin)

Semijoin: Requesting all join partners in just one step



Distributed Query Processing
Join Queries in Distributed DBMS (Bloom join)

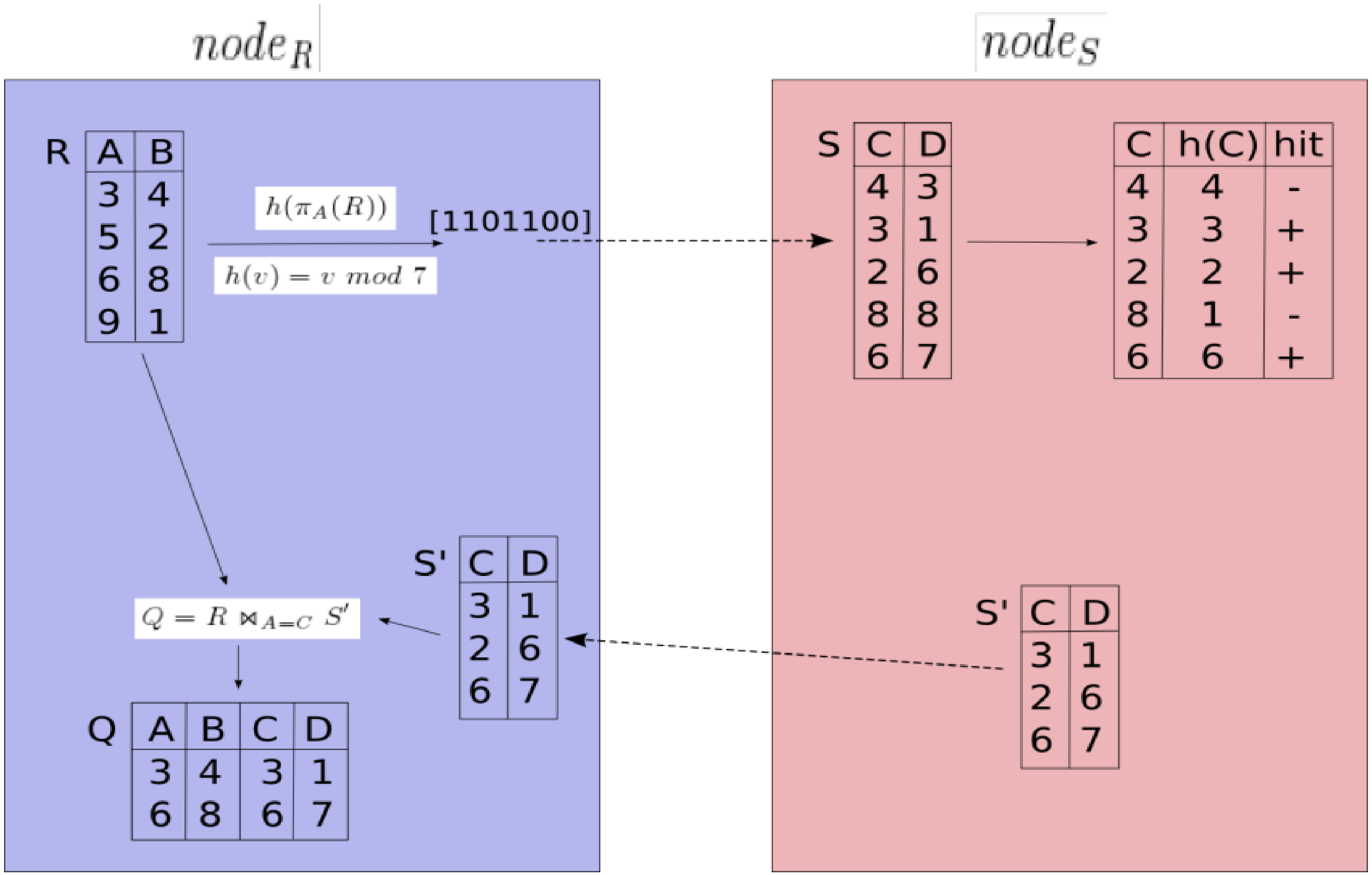
Bloom join:

- Also known as bit-vector join
- Avoiding to transfer all join attribute values to the other node
- Instead transferring a bitvector B[1 : : : n]

- Transformation
  - Choose an appropriate hash function *h*
  - Apply *h* to transform attribute values to the range [1 : : : n]
  - Set the corresponding bits in the bitvector B[1 : : : n] to 1

Distributed Query Processing

Join Queries in Distributed DBMS (Bloom join)



Distributed Query Processing

Join Queries in Distributed DBMS (Bloom join)

Conclusion:

- Transferring the bit-vector reduces network load
- Bit-vector only indicates potential join partners because multiple attribute values might map to the same hash value *Might result in transferring unnecessary tuples*
- Requirements: an appropriate hash function *h* and *n* needs to be large enough to avoid a high number of collision

Distributed Query Processing

Join Queries in Distributed DBMS

Semijoin vs bloom-join

* The cost of shipping is less in bloom-join since bit-vector is used rather than projection

* In Bloom join the size of the reductioned part which transferred back is likely to be larger than in Semijoin, so the costs of shipping are likely to be higher

Cost-Based Query Optimization

Main Consideration for Query Optimizition

- Communication cost
- If there is several copies of a relation, decide which copy to use
- Amount of data being shipped
- Relative processing speed at each site
- Site selection

Cost-Based Query Optimization

- Global plan:
  - Includes several local plans (subqueries)
  - If response time is critical, subqueries can be carried out in parallel
  - Local plans constructed by optimizer of each site

THANK YOU!