Distributed data management

Lecture 8
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

22 May 2015

Admin

- project update (project part 3) due by the end of Monday
- team meetings with instructor next week
  - sign up your team in Doodle (choose exactly one of the unclaimed time slots)

Agenda

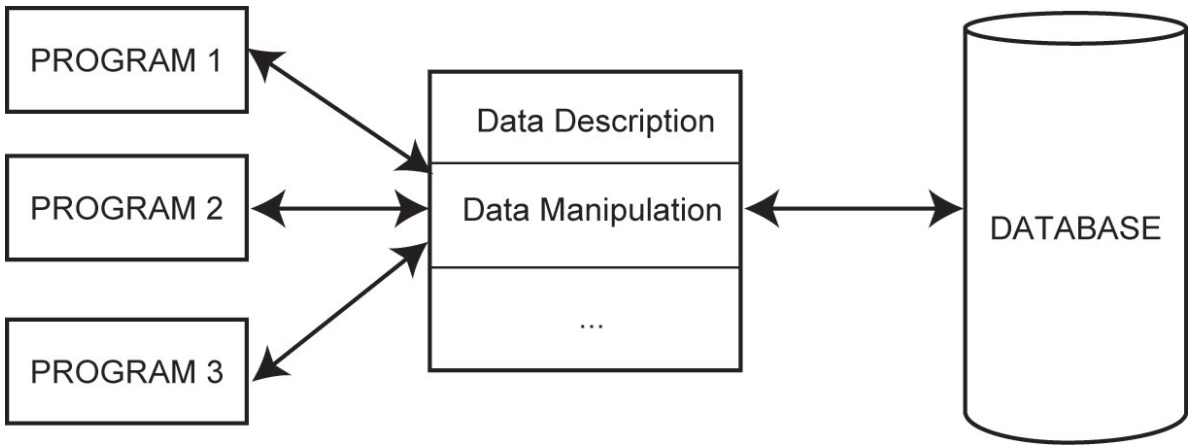
Today:

- overview of distributed DBMSs
- distributed query processing

The story so far ...

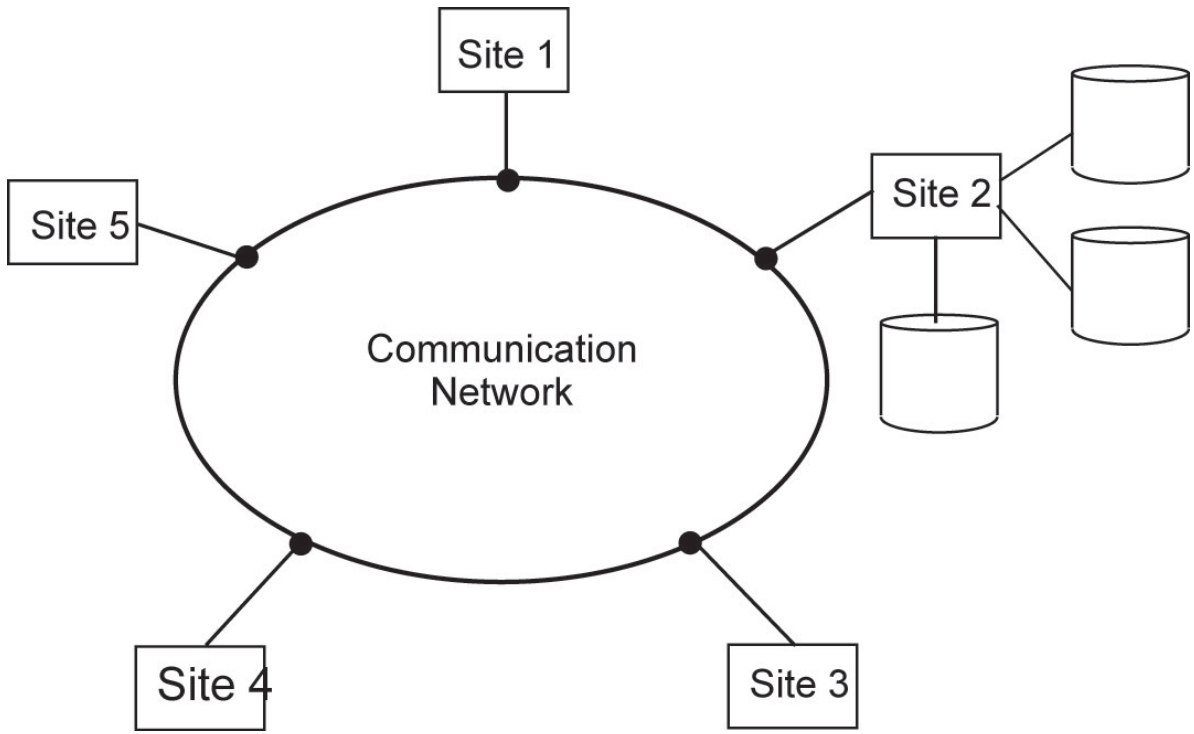
Up to this point, we've positioned a DBMS as:

1. protector of a precious commodity
2. centralized
3. directly interacting with clients



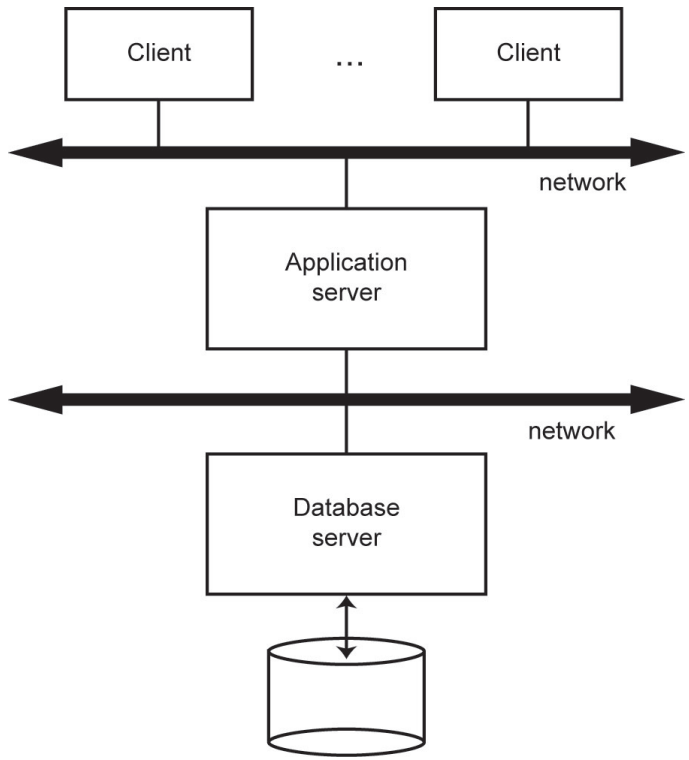
The story so far ...

- of course, we no longer live in the age of the mainframe
- in particular, a centralized DB is now typically served to many clients over a network (i.e., client-server/multi-tier architecture)



The story so far ...

- of course, we no longer live in the age of the mainframe
- in particular, a centralized DB is now typically served to many clients over a network (i.e., client-server/multi-tier architecture)

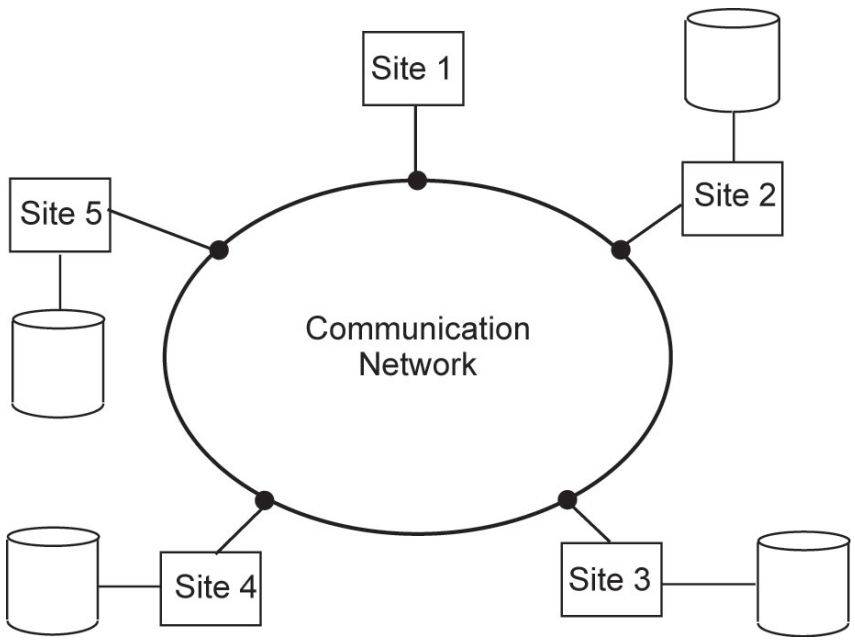


Distribution of resources

In fact, distribution is now often the norm

- data/storage
- CPUs
- clients

and it is desirable to both deal with and take advantage of this distribution



Distributed DBMSs

Today, we survey the broader landscape introduced by distribution

Distributed database

- a collection of multiple, logically interrelated databases distributed over a computer network

Distributed DBMSs

Today, we survey the broader landscape introduced by distribution

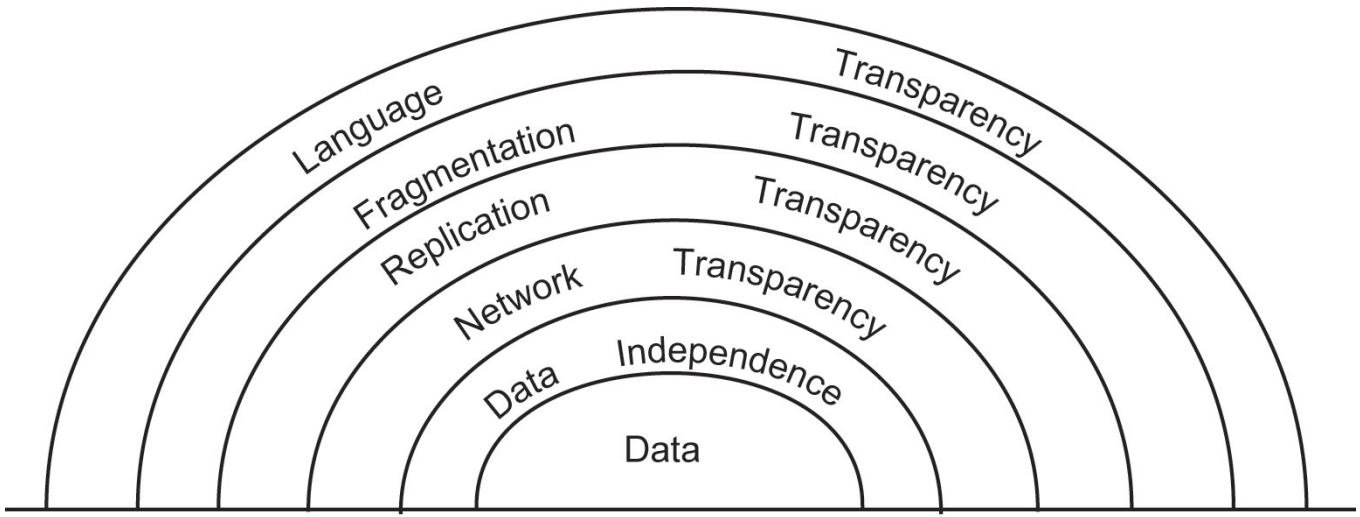
Distributed database

- a collection of multiple, logically interrelated databases distributed over a computer network

Distributed DBMS

- a software system that permits the management of a distributed database which makes the distribution transparent to its clients

Distributed DBMSs: transparency



Distributed DBMSs: hope & reality

Promises of distributed DBMSs

- transparent management of distributed and replicated data
- increased reliability through replication
- improved performance through data localization (fragmentation) and parallelism in query processing
- easier system expansion and language transparency

Distributed DBMSs: hope & reality

Promises of distributed DBMSs

- transparent management of distributed and replicated data
- increased reliability through replication
- improved performance through data localization (fragmentation) and parallelism in query processing
- easier system expansion and language transparency

Complications introduced by distributed DBMSs

- maintenance of replicated data
- dealing with communication and site failures
- synchronization of distributed transactions

Distributed DBMSs: architectures

centralized systems

- client-server/multi-tier

Distributed DBMSs: architectures

centralized systems

- client-server/multi-tier

parallel systems: multiple CPUs & disks in parallel, centrally administered

- (shared memory or/and disk)
- shared nothing

Distributed DBMSs: architectures

centralized systems

- client-server/multi-tier

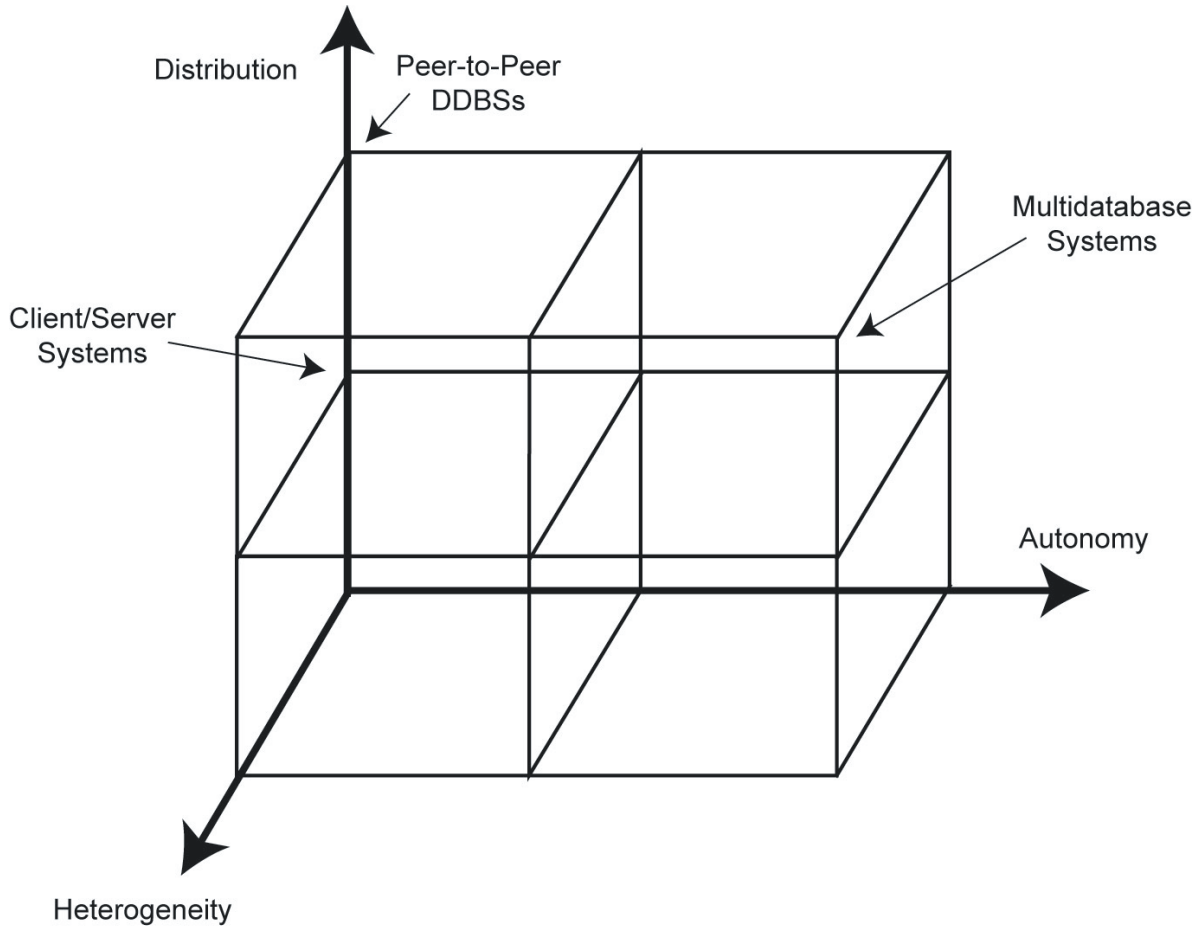
parallel systems: multiple CPUs & disks in parallel, centrally administered

- (shared memory or/and disk)
- shared nothing

distributed systems: greater separation in administration and communication

- homogeneous sites: peer-to-peer
- heterogeneous sites: multidatabase, federated, mediator systems

Distributed DBMSs: architectures



Parallel DBMSs

Parallel DBMSs

Suppose we have a terabyte of data, which we'd like to scan

- with one "pipe" of 10 MB/s, this will take a little over one day

Parallel DBMSs

Suppose we have a terabyte of data, which we'd like to scan

- with one "pipe" of 10 MB/s, this will take a little over one day
- with 1,000 such pipes scanning separate partitions (i.e., disjoint subsets) in parallel, this will take about 90s

Parallel DBMSs

Parallelism is very natural for relational data processing

- bulk processing over many partitions in parallel
- natural pipelining in parallel
- inexpensive hardware is often sufficient
- clients are not forced to think in parallel (just SQL)

Parallel DBMSs

Parallelism is very natural for relational data processing

- bulk processing over many partitions in parallel
- natural pipelining in parallel
- inexpensive hardware is often sufficient
- clients are not forced to think in parallel (just SQL)

Extremely successful in practice: every major vendor has a parallel server product (e.g., OLTP, OLAP)

Parallel DBMSs: query processing

Forms of query parallelism

- intra-operator parallelism
  - multiple sites working to compute a single operator (e.g., selection, sort, join)

Parallel DBMSs: query processing

Forms of query parallelism

- intra-operator parallelism
  - multiple sites working to compute a single operator (e.g., selection, sort, join)
- inter-operator parallelism
  - each operator running concurrently on different sites, with pipelining of results

Parallel DBMSs: query processing

Forms of query parallelism

- intra-operator parallelism
  - multiple sites working to compute a single operator (e.g., selection, sort, join)
- inter-operator parallelism
  - each operator running concurrently on different sites, with pipelining of results
- inter-query parallelism
  - different queries running concurrently on different sites

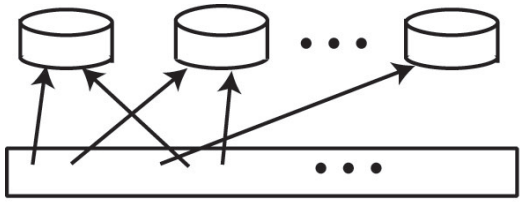
Parallel DBMSs: query processing

Forms of query parallelism

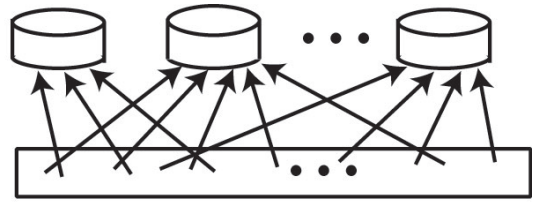
- intra-operator parallelism
  - multiple sites working to compute a single operator (e.g., selection, sort, join)
- inter-operator parallelism
  - each operator running concurrently on different sites, with pipelining of results
- inter-query parallelism
  - different queries running concurrently on different sites

We illustrate intra-operator parallelism in selection, sort, and join processing

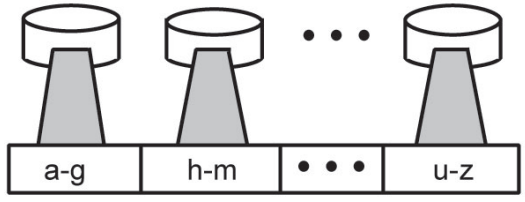
Parallel DBMSs: data placement



(a) Round-Robin



(b) Hashing

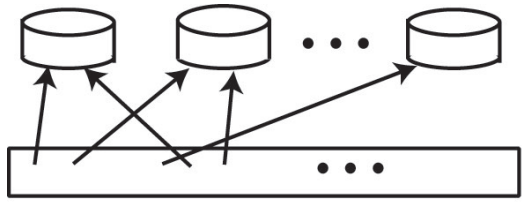


(c) Range

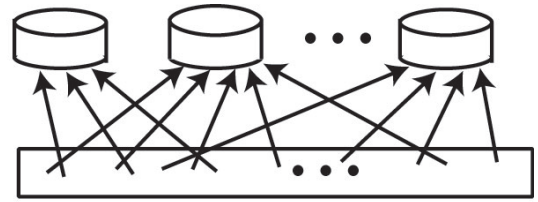
(a) round-robin: given n partition blocks, ith tuple is assigned to block ($i \mod n$)

- load-balanced
- selections and range queries require full scan

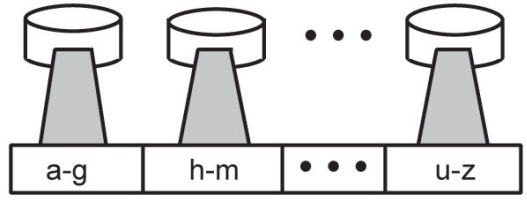
Parallel DBMSs: data placement



(a) Round-Robin



(b) Hashing

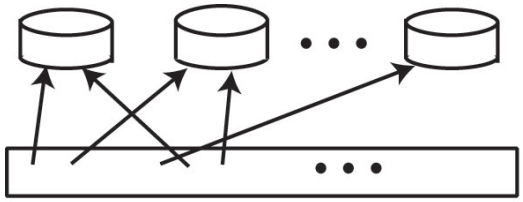


(c) Range

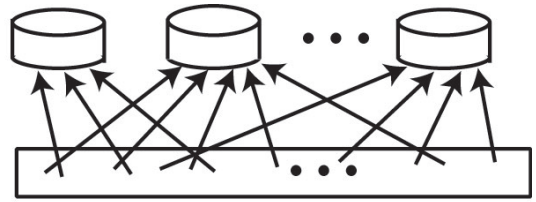
(b) hash: tuples assigned to partition blocks based on hash value

- problems with data skew
- range queries require full scan
- excellent for point-selections

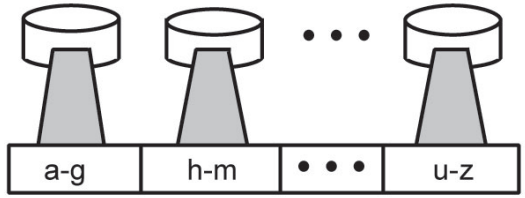
Parallel DBMSs: data placement



(a) Round-Robin



(b) Hashing



(c) Range

(c) range: tuples assigned to partition blocks in intervals

- problems with data skew
- excellent for point-selections and range queries

Parallel DBMSs: selections

parallel point-selections

- round-robin partitioning ☹
- hash partitioning ☺
- range partitioning ☺

Parallel DBMSs: selections

parallel point-selections

- round-robin partitioning ☹
- hash partitioning ☺
- range partitioning ☺

parallel range-selections

- round-robin partitioning ☹
- hash partitioning ☹
- range partitioning ☺

Parallel DBMSs: selections

parallel point-selections

- round-robin partitioning ☹
- hash partitioning ☺
- range partitioning ☺

parallel range-selections

- round-robin partitioning ☹
- hash partitioning ☹
- range partitioning ☺

... but good behavior is often offset by the costs introduced by data skew.

Parallel DBMSs: sorting

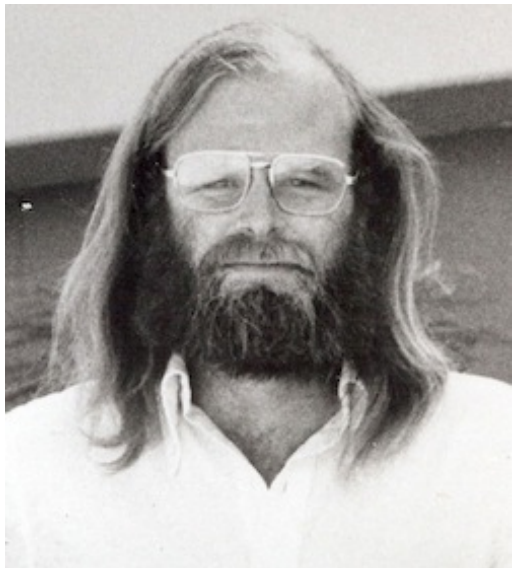
parallel sorting: sorting phases are intrinsically parallel

- scan in parallel, range-partition as you go
- use standard algorithm for local sorting
- resulting data is sorted and range-partitioned

Parallel DBMSs: sorting

parallel sorting: sorting phases are intrinsically parallel

- scan in parallel, range-partition as you go
- use standard algorithm for local sorting
- resulting data is sorted and range-partitioned



Jim Gray

Parallel DBMSs: joins

$R \bowtie S$: nested loop join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S

Parallel DBMSs: joins

$R \bowtie S$: nested loop join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, all m sites containing fragments of R ship their fragment to all n sites containing a fragment of S
- In the 2nd phase, joins are performed locally at each S site

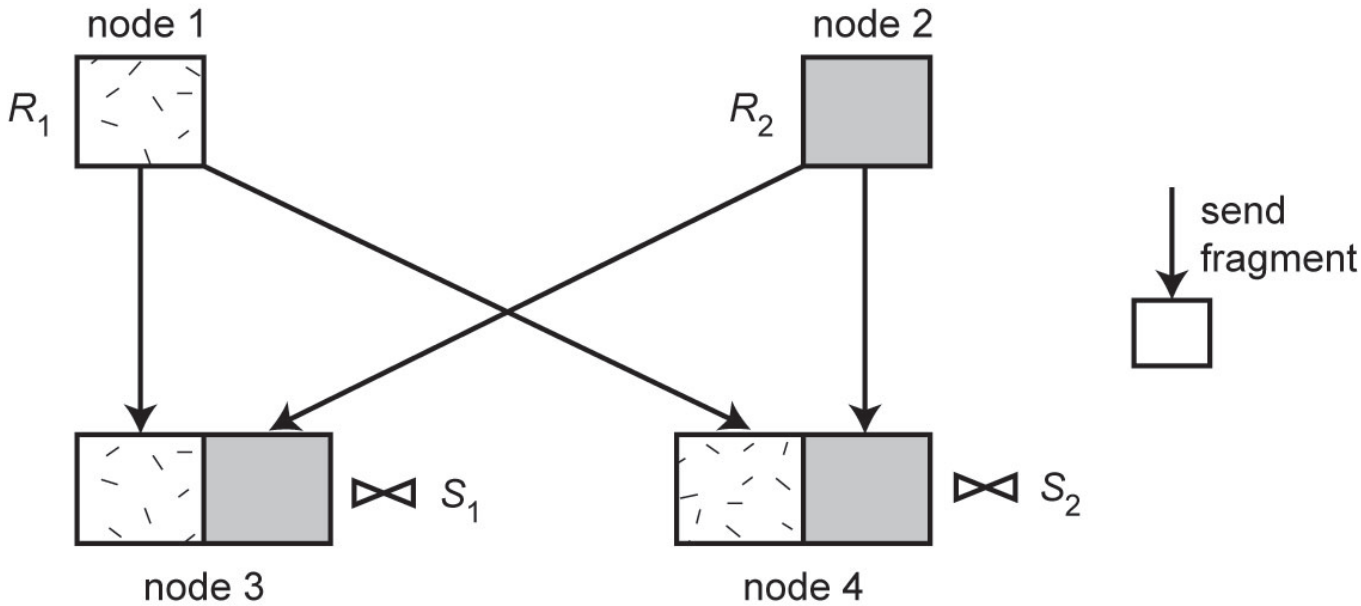
Parallel DBMSs: joins

$R \bowtie S$: nested loop join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, all m sites containing fragments of R ship their fragment to all n sites containing a fragment of S
- In the 2nd phase, joins are performed locally at each S site
- **output.** $T_1, \ldots, T_n$: result fragments

So, $R \bowtie S$ is computed as $\bigcup_{1 \leq i \leq n} (R \bowtie S_i)$

Parallel DBMSs: joins



Parallel nested loop with $m = n = 2$

Parallel DBMSs: joins

$R \bowtie S$: (sort-) merge join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S

Parallel DBMSs: joins

$R \bowtie S$: (sort-) merge join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, range partition R and S across p sites
- The 2nd sort and merge phase is performed locally at each site

Parallel DBMSs: joins

$R \bowtie S$: (sort-) merge join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, range partition R and S across p sites
- The 2nd sort and merge phase is performed locally at each site
- **output.** $T_1, \ldots, T_p$: result fragments

Parallel DBMSs: joins

$R \bowtie S$: hash join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S

Parallel DBMSs: joins

$R \bowtie S$: hash join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, each of the m sites containing a fragment of R hashes and ship its tuples to p sites.
- In the 2nd phase, the S fragments are also hashed and shipped, and joins are performed locally at each of the p sites

Parallel DBMSs: joins

$R \bowtie S$: hash join

- **input.** $R_1, \ldots, R_m$: fragments of R
- **input.** $S_1, \ldots, S_n$: fragments of S
- In the 1st phase, each of the m sites containing a fragment of R hashes and ship its tuples to p sites.
- In the 2nd phase, the S fragments are also hashed and shipped, and joins are performed locally at each of the p sites
- **output.** $T_1, \ldots, T_p$: result fragments

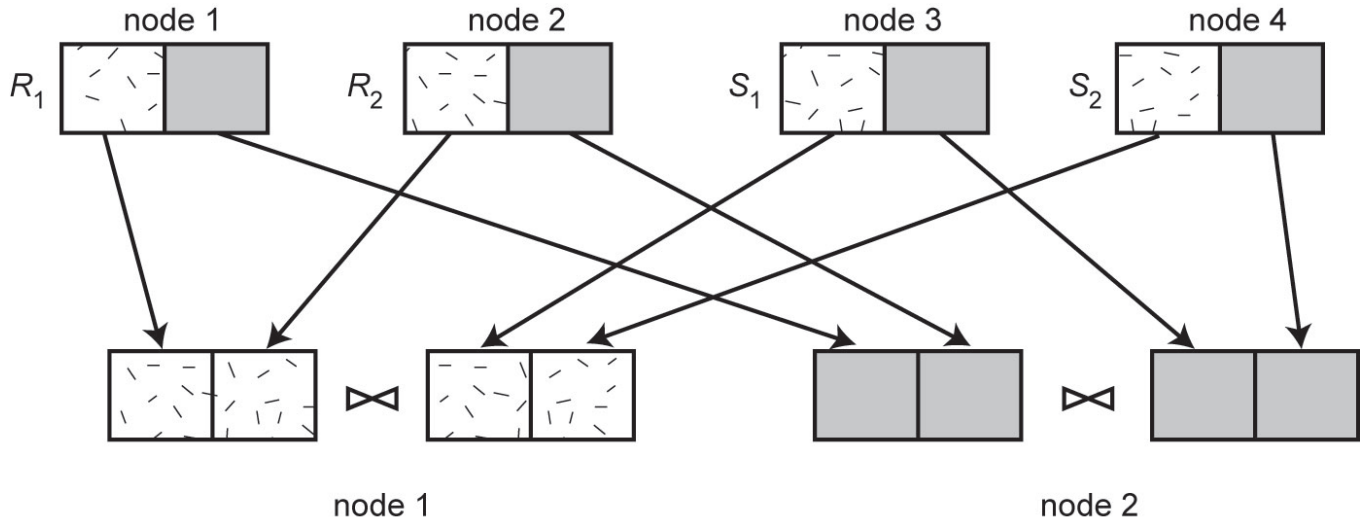
Parallel DBMSs: joins

$R \bowtie S$: hash join

So, $R \bowtie S$ is computed as

$$\bigcup_{1 \le i \le p} \left[\left(\bigcup_{1 \le j \le m} R_{ji} \right) \bowtie \left(\bigcup_{1 \le j \le n} S_{ji} \right) \right]$$

Parallel DBMSs: joins



Parallel hash join with $m = n = p = 2$, where results are produced at sites 1 and 2 (i.e., some transfers are local).

Parallel DBMSs

some other major issues we haven't touched on

- query optimization
- transaction management
- load balancing

Parallel DBMSs

some other major issues we haven't touched on

- query optimization
- transaction management
- load balancing

to wrap up this topic

- parallel processing is very natural for relational query processing
- many new challenges and opportunities for data management
- a major success story in practice, with new models/applications regularly emerging

Distributed DBMSs

Distributed DBMSs

two extremes of distributed DB systems

- homogeneous systems: same schema, tight cooperation, essentially identical sites
  - peer-to-peer

Distributed DBMSs

two extremes of distributed DB systems

- homogeneous systems: same schema, tight cooperation, essentially identical sites
  - peer-to-peer
- heterogeneous systems: varied schema and capabilities, loose cooperation
  - multi-database
  - federated databases
  - mediator systems

Distributed DBMSs: data storage

often have *replication and fragmentation* of data at multiple sites

- advantage: availability (localization)
- advantage: increased parallelism
- disadvantage: increased update cost

Distributed DBMSs: data storage

vertical fragmentation of a relation R

- $R = R_1 \bowtie \cdots \bowtie R_n$
- i.e., the R_i's are subsets of the attributes of R (including a key)

Distributed DBMSs: data storage

vertical fragmentation of a relation R

- $R = R_1 \bowtie \cdots \bowtie R_n$
- i.e., the R_i's are subsets of the attributes of R (including a key)

horizontal fragmentation of a relation R

- $R = R_1 \cup \cdots \cup R_n$
- i.e., the R_i's are subsets of tuples of R
- each fragment R_i has a guard condition g_i which defines it

$$R_i = \sigma_{g_i}(R)$$

Distributed DBMSs: data storage

Hence, at a high level, query processing involves *unfolding* the logical view definition, and then *optimizing*

Distributed DBMSs: data storage

example. suppose we have the following database schema

accounts(AID, balance, branch)
loans(LID, amount, branch)
holds(CID, AID, branch)
customers(CID, name, address)

where *holds* and *customers* are stored at the main bank office site, and (horizontal fragments of) *holds*, *accounts*, and *loans* are stored at the corresponding branch office sites (i.e., the guard at each branch b is defined as "$branch = b$")

Distributed DBMSs: data storage

if we wish to find the balances of all accounts, then we must unfold the logical *accounts* schema in terms of its fragmentation, i.e.,

$$\pi_{balance}(accounts) \;=\; \bigcup_{b \in B} \pi_{balance}(accounts_b)$$

for the set of branch names B and

$$accounts_b \;=\; \sigma_{branch=b}(accounts).$$

Distributed DBMSs: data storage

and, *updating* also faces the issues we saw for views

...

to update R with insertion of tuple t

- if $R = R_1 \bowtie \cdots \bowtie R_n$, then insert $t[R_i]$ into fragment i

Distributed DBMSs: data storage

and, *updating* also faces the issues we saw for views

...

to update R with insertion of tuple t

- if $R = R_1 \bowtie \cdots \bowtie R_n$, then insert $t[R_i]$ into fragment i
- if $R = R_1 \cup \cdots \cup R_n$, then find one i such that $g_i(t)$ is true, and insert t into fragment i
  - if we have a choice, prefer a local fragment

Distributed DBMSs: data storage

to update R with deletion of tuple t

- if $R = R_1 \bowtie \cdots \bowtie R_n$, then we can't simply delete $t[R_i]$ at each site i, since this impacts all tuples t' where $t'[R_i] = t[R_i]$
  - use TIDs to reconstruct and delete all and only those tuples t' where $t' = t$

Distributed DBMSs: data storage

to update R with deletion of tuple t

- if $R = R_1 \bowtie \cdots \bowtie R_n$, then we can't simply delete $t[R_i]$ at each site i, since this impacts all tuples t' where $t'[R_i] = t[R_i]$
  - use TIDs to reconstruct and delete all and only those tuples t' where $t' = t$
- if $R = R_1 \cup \cdots \cup R_n$, then delete t from each i where $g_i(t)$ is true

Distributed DBMSs: homogeneous case

assumptions

- tight site cooperation
  - we have a globally known directory
- site independence
- communication cost is significant
  - the cost of transmitting n bytes is modeled as

$$cost(n) = c_0 + c_1 n$$

for some setup cost c_0 and transfer cost c_1

Distributed DBMSs: homogeneous case

assumptions

- tight site cooperation
  - we have a globally known directory
- site independence
- communication cost is significant
  - the cost of transmitting n bytes is modeled as

$$cost(n) = c_0 + c_1 n$$

for some setup cost c_0 and transfer cost c_1

for example, suppose we have relations R_1 and R_2 distributed over sites s_1 and s_2, with directory

	s_1	s_2
R_1	F_{11}	F_{12}
R_2	F_{21}	F_{22}

Distributed DBMSs: homogeneous case

let's consider query processing under these assumptions

Distributed DBMSs: homogeneous case

let's consider query processing under these assumptions

- $\sigma_c(R)$ and $\pi_{\overline{A}}(R)$ are straightforward

Distributed DBMSs: homogeneous case

let's consider query processing under these assumptions

- $\sigma_c(R)$ and $\pi_{\overline{A}}(R)$ are straightforward
- for $R_1 \bowtie R_2$, it would be nice if we could compute this as

$$R_1 \bowtie R_2 = (F_{11} \bowtie F_{21}) \cup (F_{12} \bowtie F_{22})$$

since this would mean no network communication

Distributed DBMSs: homogeneous case

of course, if we have $t_1 \in F_{11}$ and $t_2 \in F_{22}$ where $t_1 \bowtie t_2$, then this tuple won't appear in the result

however, it isn't unusual to have horizontal fragmentation where this strategy does work, for example

$$accounts_b \bowtie holds_b$$

and

$$employee_l \bowtie dependents_l$$

Distributed DBMSs: homogeneous case

of course, if we have $t_1 \in F_{11}$ and $t_2 \in F_{22}$ where $t_1 \bowtie t_2$, then this tuple won't appear in the result

however, it isn't unusual to have horizontal fragmentation where this strategy does work, for example

$$accounts_b \bowtie holds_b$$

and

$$employee_l \bowtie dependents_l$$

it would be nice to have the tools to determine when it is indeed possible to avoid communication ...

Distributed DBMSs: homogeneous case

definition. Two fragmented relations R_i and R_j have a placement dependency on attribute A if and only if $F_{is} \bowtie_A F_{jt} = \emptyset$ for distinct sites s and t.
In other words,

$$R_i \bowtie_A R_j = \bigcup_{s \in S} \left(F_{is} \bowtie_A F_{js} \right)$$

for sites S containing fragments of R_i and R_j.

Distributed DBMSs: homogeneous case

observation 1. If R_i and R_j have a placement dependency (pd) on attribute A, then they have a pd on any set of attributes B containing A

- since, if $F_{is} \bowtie_A F_{jt} = \emptyset$ (for $s \neq t$), then these fragments have no common value on A, and hence also on B.

Distributed DBMSs: homogeneous case

observation 1. If R_i and R_j have a placement dependency (pd) on attribute A, then they have a pd on any set of attributes B containing A

- since, if $F_{is} \bowtie_A F_{jt} = \emptyset$ (for $s \neq t$), then these fragments have no common value on A, and hence also on B.

Similarly, if $C \rightarrow A$ (i.e., C functionally determines A), then R_i and R_j have a pd on C

- since, if there is no pd on C, then there exist distinct fragments with tuples sharing a value on C, and hence on A, a contradiction.

Distributed DBMSs: homogeneous case

observation 2. If R_i and R_j have a pd on A and R_j and R_k have a pd on B, then

$$R_i \bowtie_A R_j \bowtie_B R_k = \bigcup_{s \in S} \left(F_{is} \bowtie_A F_{js} \bowtie_B F_{ks} \right)$$

for sites S containing fragments of R_i, R_j, and R_k (i.e., can be processed without data transfer).

Distributed DBMSs: homogeneous case

observation 2. If R_i and R_j have a pd on A and R_j and R_k have a pd on B, then

$$R_i \bowtie_A R_j \bowtie_B R_k = \bigcup_{s \in S} \left(F_{is} \bowtie_A F_{js} \bowtie_B F_{ks} \right)$$

for sites S containing fragments of R_i, R_j, and R_k (i.e., can be processed without data transfer).

This can be generalized to queries $Q = R_i \bowtie_B Q'$ where

1. the relation R_j to which R_i joins in Q' has a pd on (a subset of) B with R_i, and
2. Q' can be processed without data transfer.

Then, Q can be processed without data transfer.

Distributed DBMSs: homogeneous case

These observations give us the following simple algorithm to determine if a query Q can be processed without data transfer.

Distributed DBMSs: homogeneous case

let $R = \{R_1, \ldots, R_n\}$ be the set of relations referenced in input query Q

1. $S \leftarrow \emptyset$
2. if a pair of relations R_i and R_j in R can be found such that they have a pd on some attribute A, and $R_i \bowtie_C R_j$ is in Q, $A \subseteq C$, place R_i and R_j in S
3. else return FALSE
4. while there is a $R_k \in R - S$ such that $\varphi(R_k)$
   4.1 place R_k in S
5. if $S = R$ return TRUE, else return FALSE

where $\varphi(R_k)$ holds if there is a $R_j \in S$ that has a pd with R_k on some attribute B such that $R_j \bowtie_B R_k$ is in Q (or follows from Q)

Distributed DBMSs: homogeneous case

example. Let $Q = R_1 \bowtie_A R_2 \bowtie_B R_3 \bowtie_C R_4$.
Suppose there is a pd between R_1 and R_2 on A, between R_2 and R_3 on B, and between R_3 and R_4 on C.

Distributed DBMSs: homogeneous case

example. Let $Q = R_1 \bowtie_A R_2 \bowtie_B R_3 \bowtie_C R_4$.
Suppose there is a pd between R_1 and R_2 on A, between R_2 and R_3 on B, and between R_3 and R_4 on C.

Initially, S is empty. In the first "if" (line 2), R_1 and R_2 can be placed in S.

Distributed DBMSs: homogeneous case

example. Let $Q = R_1 \bowtie_A R_2 \bowtie_B R_3 \bowtie_C R_4$.
Suppose there is a pd between R_1 and R_2 on A, between R_2 and R_3 on B, and between R_3 and R_4 on C.

Initially, S is empty. In the first "if" (line 2), R_1 and R_2 can be placed in S.

Following this, the while-loop (line 4) will first place R_3 in S, and then R_4 in S.

Since the loop terminates with $S = R$, we know that Q can be processed without data transfer (line 5).

Distributed DBMSs: homogeneous case

Now, of course this algorithm won't always be successful. When data transfer is necessary, the challenge becomes that of minimizing communication cost.

Distributed DBMSs: homogeneous case

Now, of course this algorithm won't always be successful. When data transfer is necessary, the challenge becomes that of minimizing communication cost.

suppose for R_1 and R_2 that

	s_1	s_2
R_1	F_{11}	
R_2		F_{22}

and we need to compute $R_1 \bowtie_A R_2$ at s_2

Distributed DBMSs: homogeneous case

solution a. ship F_{11} to s_2 and compute the join.

Distributed DBMSs: homogeneous case

solution a. ship F_{11} to s_2 and compute the join.
Then

$$cost_a = c_0 + c_1|R_1|.$$

Suppose R_1 has n tuples of two attributes, each of some fixed size (say, one byte).
Then

$$cost_a = c_0 + 2nc_1.$$

Distributed DBMSs: homogeneous case

solution b.

1. at s_2 compute $\pi_A(F_{22})$ and ship it to s_1
2. at s_1 compute $R_1 \ltimes R_2 = F_{11} \bowtie \pi_A(F_{22})$ and ship it to s_2
3. compute the final result, using the fact that $R_1 \bowtie R_2 = (R_1 \ltimes R_2) \bowtie R_2$

Distributed DBMSs: homogeneous case

solution b.

1. at s_2 compute $\pi_A(F_{22})$ and ship it to s_1
2. at s_1 compute $R_1 \ltimes R_2 = F_{11} \bowtie \pi_A(F_{22})$ and ship it to s_2
3. compute the final result, using the fact that $R_1 \bowtie R_2 = (R_1 \ltimes R_2) \bowtie R_2$

This requires two joins and two data transfers!

Distributed DBMSs: homogeneous case

Suppose $|\pi_A(R_1)| = n$ and $|\pi_A(R_2)| = 2$. Then

$$\begin{aligned} cost_{s_2 \to s_1} &= c_0 + 2c_1 \\ cost_{s_1 \to s_2} &= c_0 + 4c_1 \\ cost_b &= 2c_0 + 6c_1 \end{aligned}$$

Distributed DBMSs: homogeneous case

Suppose $|\pi_A(R_1)| = n$ and $|\pi_A(R_2)| = 2$. Then

$$\begin{aligned} cost_{s_2 \to s_1} &= c_0 + 2c_1 \\ cost_{s_1 \to s_2} &= c_0 + 4c_1 \\ cost_b &= 2c_0 + 6c_1 \end{aligned}$$

so solution (b) is cheaper than solution (a) when $c_0 + 6c_1 < 2nc_1$

- i.e., when $\pi_{R_1 \cap R_2}(R_2)$ or $R_1 \ltimes R_2$ is small

Distributed DBMSs: homogeneous case

efficient implementation of this semi-join strategy for processing $R_1 \bowtie R_2$ using Bloom filters

1. initialize a bit vector to all 0's
2. each value of $\pi_{R_1 \cap R_2}(R_2)$ is hashed into the bit vector (i.e., bit is set to 1)
3. bit vector is shipped to s_1

Distributed DBMSs: homogeneous case

efficient implementation of this semi-join strategy for processing $R_1 \bowtie R_2$ using Bloom filters

1. initialize a bit vector to all 0's
2. each value of $\pi_{R_1 \cap R_2}(R_2)$ is hashed into the bit vector (i.e., bit is set to 1)
3. bit vector is shipped to s_1
4. for each tuple $t \in R_1$, hash $\pi_{R_1 \cap R_2}(t)$ into the filter
5. if t hashes to 1, then it is shipped to s_2, and otherwise not (note that this is just a scan of R_1, instead of a full (semi)-join)
6. at s_2, compute $R_1 \bowtie R_2$

Distributed DBMSs: homogeneous case

efficient implementation of this semi-join strategy for processing $R_1 \bowtie R_2$ using Bloom filters

1. initialize a bit vector to all 0's
2. each value of $\pi_{R_1 \cap R_2}(R_2)$ is hashed into the bit vector (i.e., bit is set to 1)
3. bit vector is shipped to s_1
4. for each tuple $t \in R_1$, hash $\pi_{R_1 \cap R_2}(t)$ into the filter
5. if t hashes to 1, then it is shipped to s_2, and otherwise not (note that this is just a scan of R_1, instead of a full (semi)-join)
6. at s_2, compute $R_1 \bowtie R_2$

only false positives in step (5)

Distributed DBMSs: homogeneous case

In our example above, the semi-join eliminated all unnecessary tuples from R_1.

This reduction is the advantage of the *semi-join* strategy. Unfortunately, a full reduction isn't always possible.

Distributed DBMSs: homogeneous case

In our example above, the semi-join eliminated all unnecessary tuples from R_1.

This reduction is the advantage of the *semi-join* strategy. Unfortunately, a full reduction isn't always possible.

Example.

R

A	B
1	1
3	4
5	5

S

B	C
1	1
3	3
5	6

T

C	A
1	2
3	3
5	5

Note that $R \bowtie S \bowtie T = \emptyset$.

Distributed DBMSs: homogeneous case

In our example above, the semi-join eliminated all unnecessary tuples from R_1.

This reduction is the advantage of the *semi-join* strategy. Unfortunately, a full reduction isn't always possible.

Example.

R

A	B
1	1
3	4
5	5

S

B	C
1	1
3	3
5	6

T

C	A
1	2
3	3
5	5

Note that $R \bowtie S \bowtie T = \emptyset$.

When is a full reduction possible?

Distributed DBMSs: homogeneous case

definition. A hypergraph $\mathcal{H}$ is a finite set U and a set E of hyper-edges (i.e., subsets of U).

Distributed DBMSs: homogeneous case

definition. A hypergraph $\mathcal{H}$ is a finite set U and a set E of hyper-edges (i.e., subsets of U).
A tree decomposition of $\mathcal{H}$ is a tree T together with a set $B_t \subseteq U$, for each node t of T, such that the following two conditions hold:

1. for every $a \in U$, the set $\{t \mid a \in B_t\}$ is a subtree of T; and,
2. every hyper-edge of $\mathcal{H}$ is contained in one of the B_t's.

Distributed DBMSs: homogeneous case

definition. A hypergraph $\mathcal{H}$ is a finite set U and a set E of hyper-edges (i.e., subsets of U).
A tree decomposition of $\mathcal{H}$ is a tree T together with a set $B_t \subseteq U$, for each node t of T, such that the following two conditions hold:

1. for every $a \in U$, the set $\{t \mid a \in B_t\}$ is a subtree of T; and,
2. every hyper-edge of $\mathcal{H}$ is contained in one of the B_t's.

$\mathcal{H}$ is acyclic if there exists a tree decomposition T of $\mathcal{H}$ such that $\forall t \in T$, B_t is a hyper-edge of $\mathcal{H}$.

Distributed DBMSs: homogeneous case

definition. Given a conjunctive query

$$Q(\overline{A}) \leftarrow \alpha_1(\overline{A_1}), \ldots, \alpha_n(\overline{A_n})$$

its hypergraph $\mathcal{H}_Q$ is defined as follows. Its node set U is the set of all variables in Q and its hyperedges E are precisely $\overline{A_1}, \ldots, \overline{A_n}$.
Q is acyclic if $\mathcal{H}_Q$ is acyclic.

Distributed DBMSs: homogeneous case

example.

$$Q(X) \leftarrow R(X, Y), R(Y, Z)$$

Distributed DBMSs: homogeneous case

example.

$$Q(X) \leftarrow R(X, Y), R(Y, Z)$$

is acyclic.

example.

$$Q(X) \leftarrow R(X, Y), R(Y, Z), R(Z, X)$$

Distributed DBMSs: homogeneous case

example.

$$Q(X) \leftarrow R(X, Y), R(Y, Z)$$

is acyclic.

example.

$$Q(X) \leftarrow R(X, Y), R(Y, Z), R(Z, X)$$

is not acyclic (i.e., is cyclic).

Distributed DBMSs: homogeneous case

exercise.

$$Q_1(X) \leftarrow R(X, Y, Z), R(Z, U, V), S(U, Z), S(X, Y), S(V, W)$$

Distributed DBMSs: homogeneous case

exercise.

$$Q_1(X) \leftarrow R(X,Y,Z), R(Z,U,V), S(U,Z), S(X,Y), S(V,W)$$

is acyclic.

exercise.

$$Q_2(X) \leftarrow R(X,Y,Z), R(Z,U,V), R(X,V,W)$$

Distributed DBMSs: homogeneous case

exercise.

$$Q_1(X) \leftarrow R(X,Y,Z), R(Z,U,V), S(U,Z), S(X,Y), S(V,W)$$

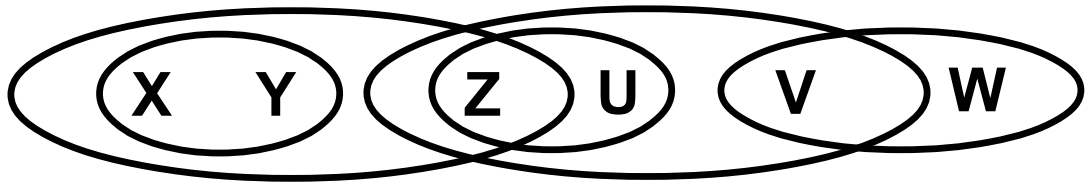
is acyclic.

exercise.

$$Q_2(X) \leftarrow R(X,Y,Z), R(Z,U,V), R(X,V,W)$$

is cyclic.

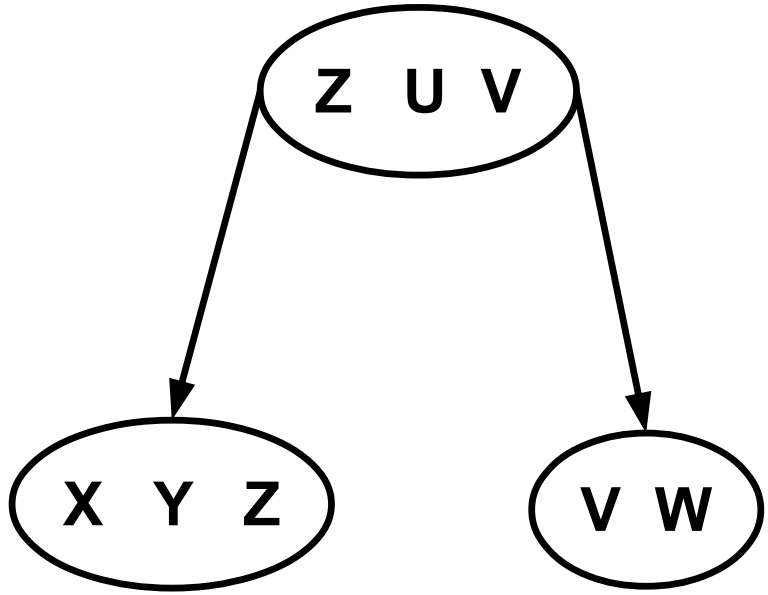
Distributed DBMSs: homogeneous case



The hypergraph $\mathcal{H}_{Q_1}$ of

$$Q_1(X) \leftarrow R(X, Y, Z), R(Z, U, V), S(U, Z), S(X, Y), S(V, W)$$

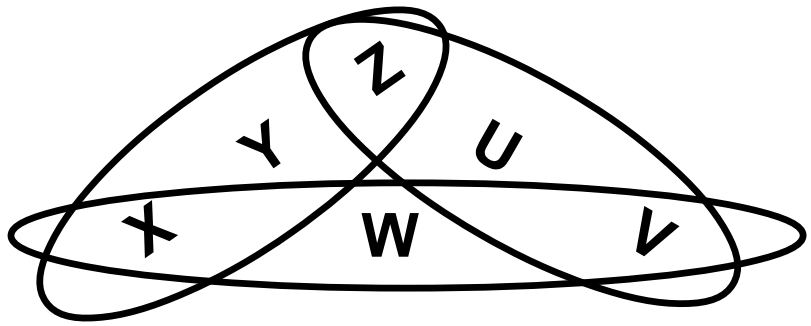
Distributed DBMSs: homogeneous case



The tree decomposition (i.e., join tree) of the hypergraph $\mathcal{H}_{Q_1}$ of

$$Q_1(X) \leftarrow R(X,Y,Z), R(Z,U,V), S(U,Z), S(X,Y), S(V,W)$$

Distributed DBMSs: homogeneous case



The hypergraph $\mathcal{H}_{Q_2}$ of

$$Q_2(X) \leftarrow R(X, Y, Z), R(Z, U, V), R(X, V, W)$$

Distributed DBMSs: homogeneous case

Recall from Lecture 2: Let

- *FO* be the full TRC
  - i.e., any of: the RA, SQL (w/o aggregation), non-recursive safe Datalog, TRC
- *Conj* be the TRC using only $\exists$ and $\wedge$
  - corresponds to: the $\{\sigma, \pi, \times\}$ fragment of RA
  - corresponds to: single SELECT-FROM-WHERE blocks
  - corresponds to: single positive safe Datalog rules
  - aka: conjunctive or SPJ queries
- *AConj* be the "acyclic" *Conj* queries
  - queries with join trees
  - aka: acyclic conjunctive queries

Distributed DBMSs: homogeneous case

Recall from Lecture 2: Let

- *FO* be the full TRC
  - i.e., any of: the RA, SQL (w/o aggregation), non-recursive safe Datalog, TRC
- *Conj* be the TRC using only $\exists$ and $\wedge$
  - corresponds to: the $\{\sigma, \pi, \times\}$ fragment of RA
  - corresponds to: single SELECT-FROM-WHERE blocks
  - corresponds to: single positive safe Datalog rules
  - aka: conjunctive or SPJ queries
- *AConj* be the "acyclic" *Conj* queries
  - queries with join trees
  - aka: acyclic conjunctive queries

full reduction is only possible for *AConj* queries!

Complexity of query evaluation

Then

	FO	*Conj*	*AConj*
combined	PSPACE-complete	NP-complete	LOGCFL-complete
data	Logspace	Logspace	Linear time

where *combined* means in the size of the query and the database; and *data* means in the size of the database (i.e., for some fixed query)

Complexity of query evaluation

Then

	FO	*Conj*	*AConj*
combined	PSPACE-complete	NP-complete	LOGCFL-complete
data	Logspace	Logspace	Linear time

where *combined* means in the size of the query and the database; and *data* means in the size of the database (i.e., for some fixed query)

furthermore, determining acyclicity and then, if so, computing a "plan" for full reduction (i.e., a join tree), are both computable in polynomial time

Query containment

Checking query containment for conjunctive queries is NP-complete, as we established in Lecture 6. For FO queries, checking containment is undecidable!

However, checking query containment for acyclic conjunctive queries is tractable (i.e., computable in polynomial time)

Query containment

Checking query containment for conjunctive queries is NP-complete, as we established in Lecture 6. For FO queries, checking containment is undecidable!

However, checking query containment for acyclic conjunctive queries is tractable (i.e., computable in polynomial time)

So, an optimization strategy which can be used alongside the semi-join strategy is to first *minimize* the original query

- until not possible: remove a clause from the query, checking if the reduced query is equivalent to the original query. If so, eliminate it from the query.

Query containment

example. minimize

$$\begin{aligned} \mathit{result}(T) \leftarrow\ & \mathit{sales}(P_1, S_1, C), \mathit{cust}(C, A_1), \mathit{part}(P_1, T), \\ & \mathit{sales}(P_2, S_2, C), \mathit{cust}(C, A_2) \end{aligned}$$

Query containment

example. minimize

$$\begin{aligned} \mathit{result}(T) \leftarrow\ & \mathit{sales}(P_1, S_1, C), \mathit{cust}(C, A_1), \mathit{part}(P_1, T), \\ & \mathit{sales}(P_2, S_2, C), \mathit{cust}(C, A_2) \end{aligned}$$

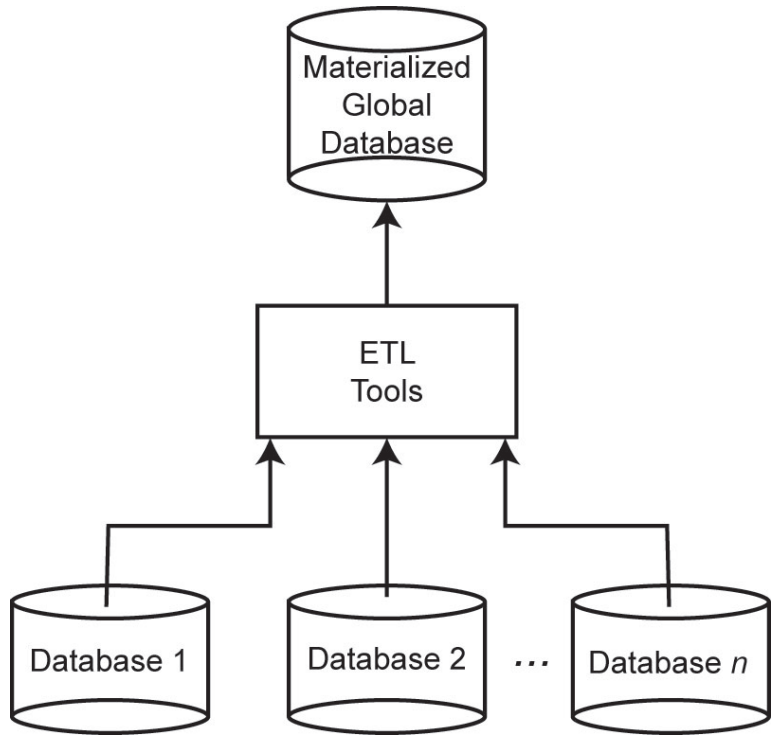
solution.

$$\mathit{result}(T) \leftarrow \mathit{sales}(P, S, C), \mathit{cust}(C, A), \mathit{part}(P, T)$$

Distributed DBMSs: heterogeneous case

Also known as federated or multi- databases, e.g.,

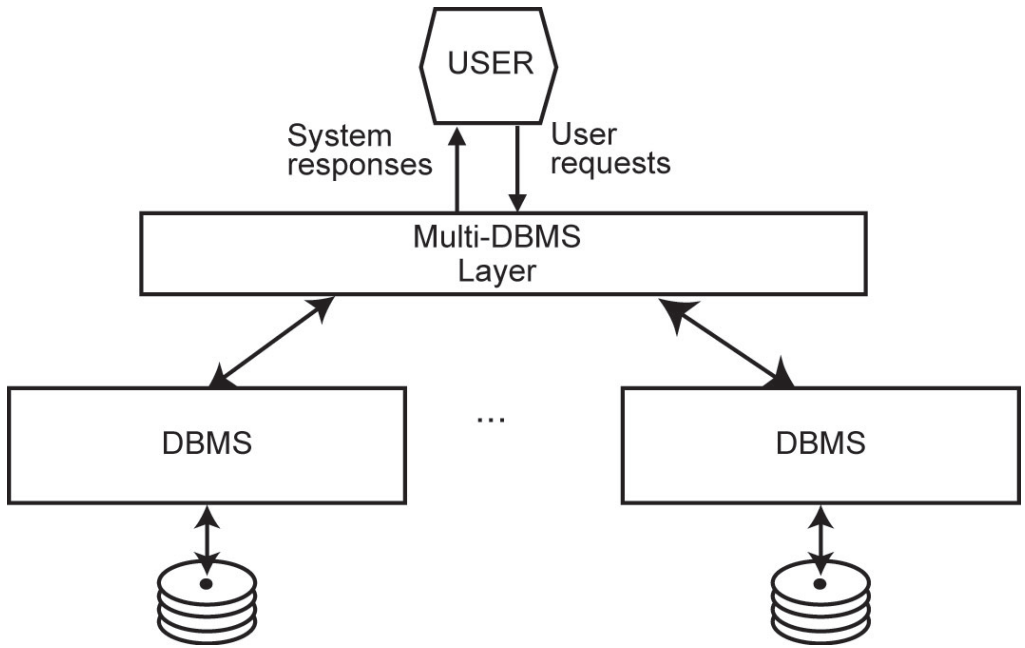
- data warehouses (materialized federation)
- mediator systems (logical federation)



Distributed DBMSs: heterogeneous case

Also known as federated or multi- databases, e.g.,

- data warehouses (materialized federation)
- mediator systems (logical federation)



Distributed DBMSs: heterogeneous case

data mappings are the basic "glue" for building heterogeneous distributed DBMSs

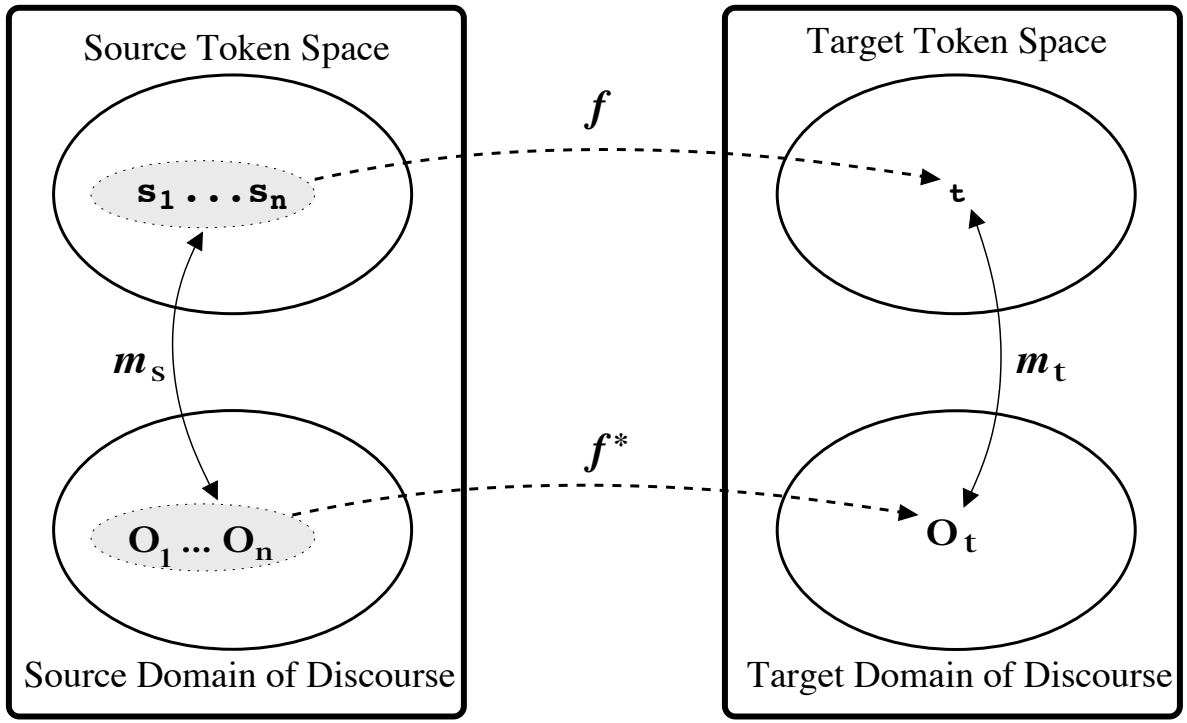
Distributed DBMSs: heterogeneous case

data mappings are the basic "glue" for building heterogeneous distributed DBMSs

two fundamental issues in data mapping

- dealing with semantic heterogeneity
  - schema matching
  - semantic mapping
- dealing with structural heterogeneity
  - query discovery

Distributed DBMSs: heterogeneous case



semantic mappings

Distributed DBMSs: heterogeneous case

FlightsA

Flights:

Carrier	Fee	ATL29	ORD17
AirEast	15	100	110
JetWest	16	200	220

FlightsB

Prices:

Carrier	Route	Cost	AgentFee
AirEast	ATL29	100	15
JetWest	ATL29	200	16
AirEast	ORD17	110	15
JetWest	ORD17	220	16

FlightsC

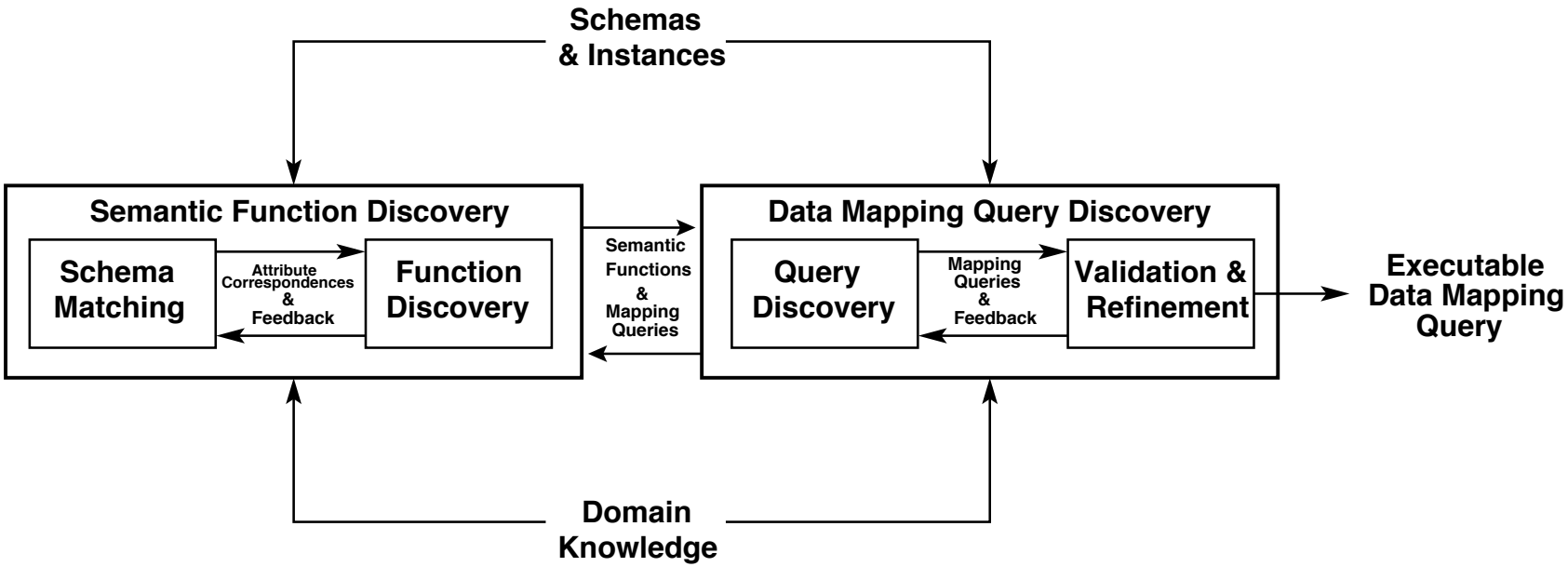
AirEast:

Route	BaseCost	TotalCost
ATL29	100	115
ORD17	110	125

JetWest:

Route	BaseCost	TotalCost
ATL29	200	216
ORD17	220	236

Distributed DBMSs: heterogeneous case



data mapping discovery (i.e., *data integration*) is the process of overcoming structural and semantic heterogeneity

Distributed DBMSs: heterogeneous case

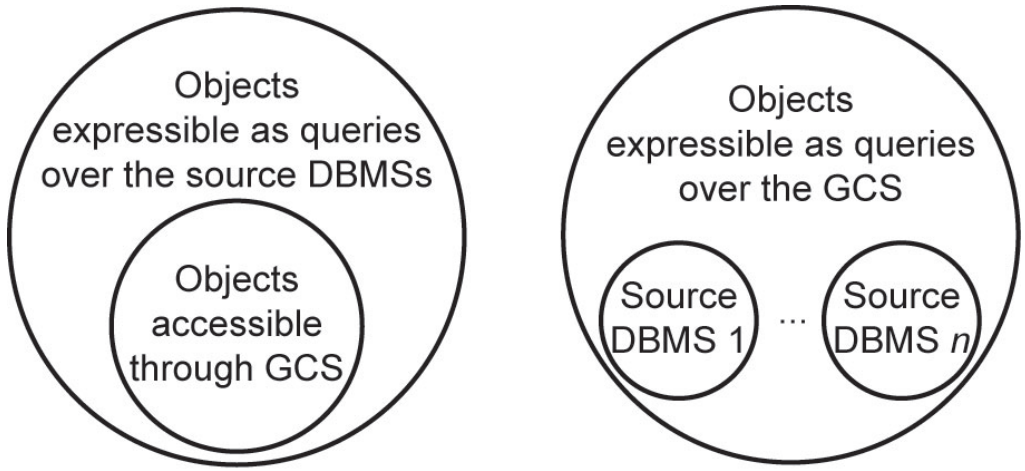
In the wrapper/mediator scenario, there are two fundamental ways of reasoning about the global coordinated schema (GCS), namely

- global-as-view (GAV), where global schema is a view over the local sources; and,
- local-as-view (LAV), where local schemas are views over the global schema.

Distributed DBMSs: heterogeneous case

global-as-view (GAV), where global schema is a view over the local sources

- query processing is just view unfolding, as before
- difficult to extend/update the mediated schema

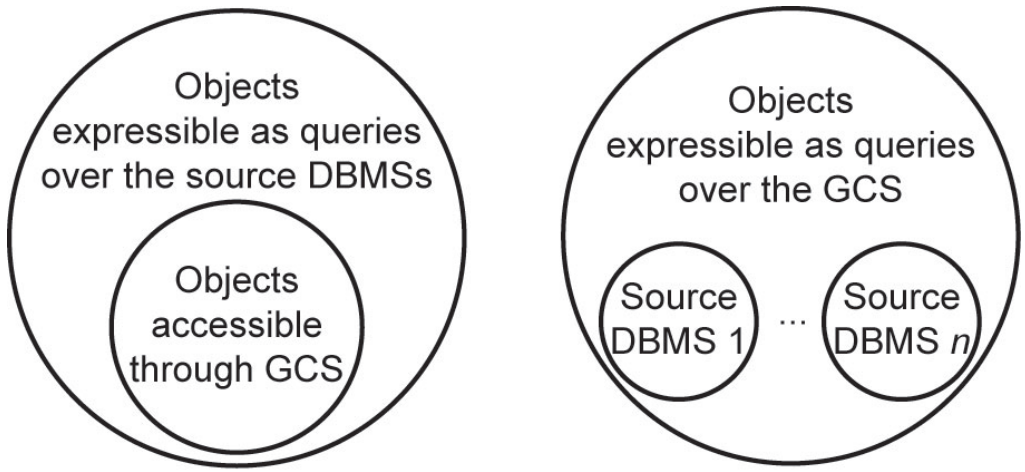


(a) GAV (b) LAV

Distributed DBMSs: heterogeneous case

local-as-view (LAV), where local schemas are views over the global schema

- easy to extend/update the global schema
- how to process queries on the global schema???



(a) GAV (b) LAV

Distributed DBMSs: heterogeneous case

LAV query processing is difficult

- finding correspondences between attributes of the global schema and local schemas requires comparison with each view
- this may be very costly, especially if there are many local sources (i.e., local views) – actually, this is NP-complete
- may not be possible to find an equivalent rewriting of the original query over the local views

Distributed DBMSs: heterogeneous case

Bucket algorithm for LAV query rewriting of query

$$Q(\overline{A}) \leftarrow \alpha_1(\overline{A_1}), \ldots, \alpha_n(\overline{A_n})$$

over the global schema

1. for each α_i build a bucket b_i, and insert the head of each local view v into b_i if α_i unifies with some subgoal of v
2. for each view V of the cartesian product of all non-empty buckets, check if $V \subseteq Q$. Such a query is one way of contributing to Q.
3. return the union of all such views

Distributed DBMSs: heterogeneous case

consider the query

$$Q(B) \leftarrow accounts(A, B, R)$$

where *accounts* is now in a mediated schema, over the *accounts* table distributed locally by branch

$$\begin{aligned} Daccounts(AID, BAL) &\leftarrow accounts(AID, BAL, R), \\ &\quad R = \text{downtown} \\ Uaccounts(AID, BAL) &\leftarrow accounts(AID, BAL, R), \\ &\quad R = \text{uptown} \end{aligned}$$

Distributed DBMSs: heterogeneous case

then Q is rewritten by the bucket algorithm as

$$\begin{aligned} Q(BAL) &\leftarrow Daccounts(AID, BAL) \\ Q(BAL) &\leftarrow Uaccounts(AID, BAL) \end{aligned}$$

Distributed DBMSs

other major issues we haven't touched on

- query optimization
- transaction management
- load balancing
- data integration solutions
- ...

Distributed DBMSs

other major issues we haven't touched on

- query optimization
- transaction management
- load balancing
- data integration solutions
- ...

to wrap up

- semi-join strategy for join processing
- data mapping problem
- query processing over mediated schemas

Recap

- Overview of distributed DBMSs
- Distributed query processing
  - parallel DBMSs
  - homogeneous distributed DBMSs (e.g., P2P)
  - heterogeneous distributed DBMSs (e.g., mediator systems)

Recap

- Overview of distributed DBMSs
- Distributed query processing
  - parallel DBMSs
  - homogeneous distributed DBMSs (e.g., P2P)
  - heterogeneous distributed DBMSs (e.g., mediator systems)

Looking ahead

- Next week: project meetings (sign up!)
- Following lecture (3 June): transaction management (data consistency, recovering from failures)

Credits

- our textbook
- Özsu & Valduriez, 2011
- Viglas, 2010
- Libkin, 2004