ANALYSIS OF BIG DATA ARCHITECTURES

Dimensions of Big Data architectures

- **Data model(s)**:
  - Relations, trees (XML, JSON), graphs (RDF, others…), nested relations
  - Query language
- **Heterogeneity** (DM, QL): none, some, a lot
- **Scale**: small (~10-20 sites) or large (~10.000 sites)
- **ACID** properties
- **Control**:
  - Single master w/complete control over N slaves (Hadoop/HDFS)
  - Sites publish independently and process queries as directed by single master/*mediator*
  - Many-mediator systems, or peer-to-peer (P2P) with *super-peers*
  - Sites completely independent (P2P)

Architectures we will cover

- Distributed databases
- Mediator (data integration) systems
- Peer-to-peer data management systems
- Structured data management on top of MapReduce
- Dataspaces, polystores, datalakes

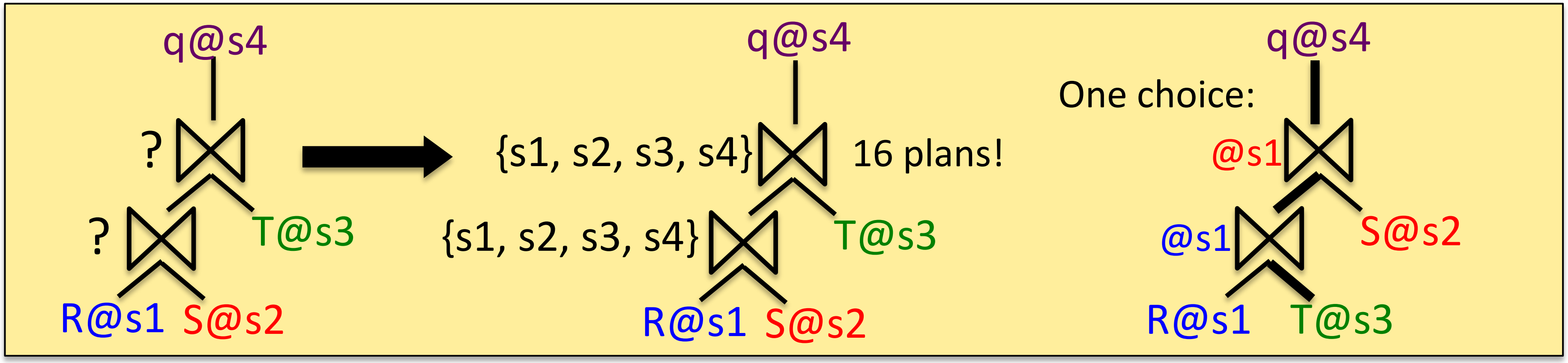
DISTRIBUTED RELATIONAL DATABASES

Distributed relational databases

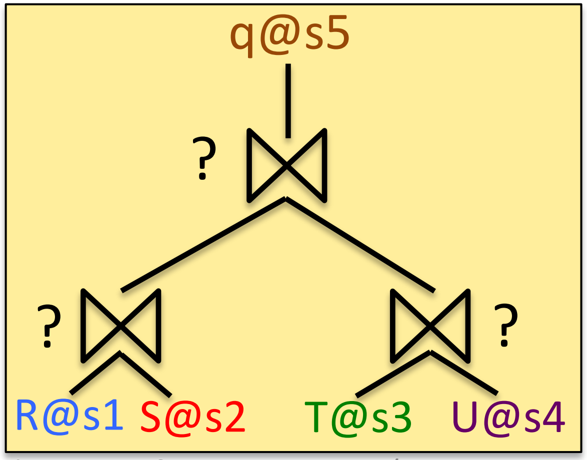
- Oldest distributed architecture ('70s): IBM System R*
- Illustrate/introduce the main priciples
- Data is distributed among many *nodes* (*sites*, *peers*...)
  - **Data catalog**: information on which data is stored where
    - Explicit : « All Paris sales are stored in Paris ».
      Horizontal/vertical table fragmentation
      Catalog stored at a master/central server.
    - Implicit: « Data is distributed by the value of the city » (« somewhere »)
- Queries are distributed (may come from any site)
- Query processing is distributed
  - Operators may run on different sites → network transfer
  - Another layer of complexity to the optimization process

Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4



Example 2: R@s1, S@s2, T@s3, U@s4, q@s5

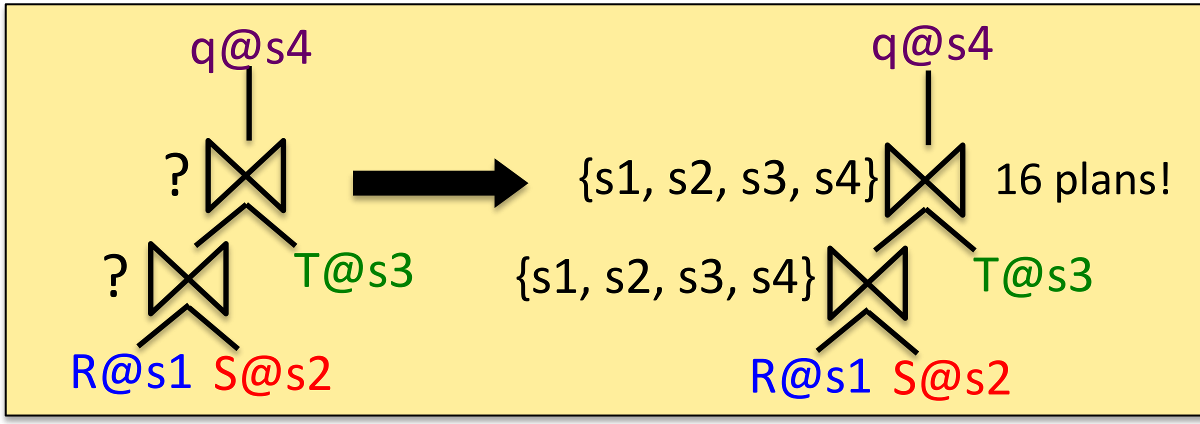


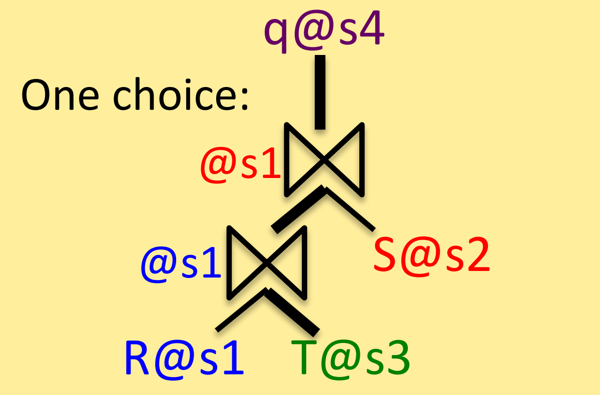
Plan pruning criteria if all the sites and network connections have equal performance:

- *Ship the <u>smaller</u> collection*

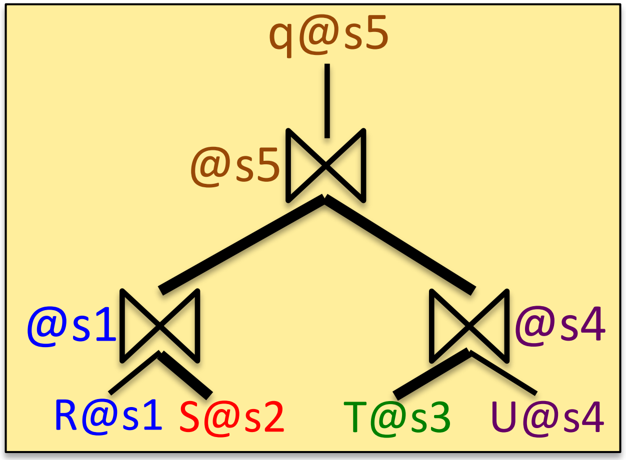
Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4





Example 2: R@s1, S@s2, T@s3, U@s4, q@s5

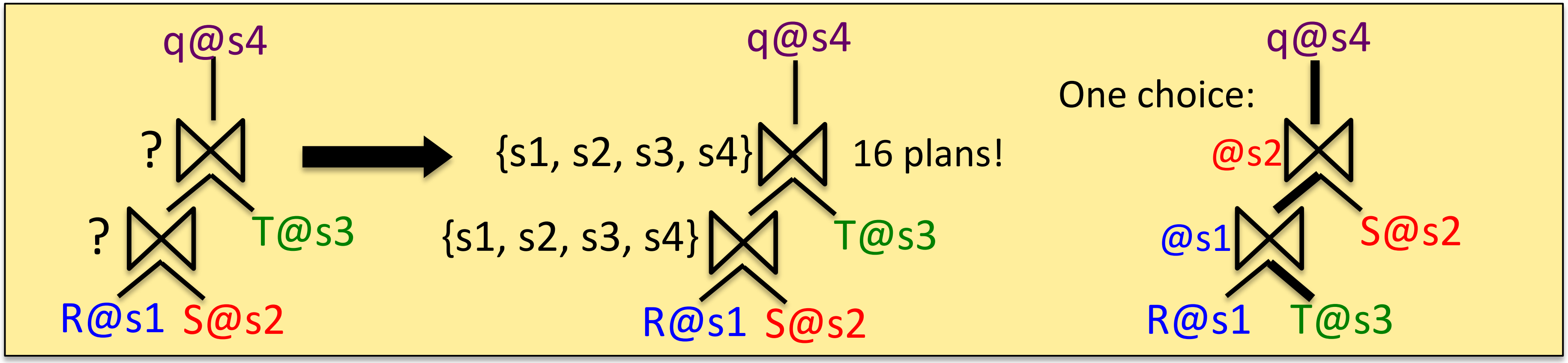


Plan pruning criteria if all the sites and network connections have equal performance:

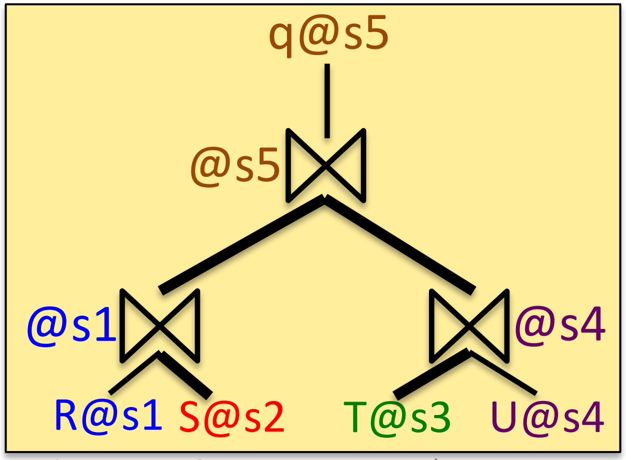
- *Ship the smaller collection*
- *Transfer to join partner or the query site*

Distributed query optimization

Example 1: R@s1, S@s2, T@s3, q@s4



Example 2: R@s1, S@s2, T@s3, U@s4, q@s5



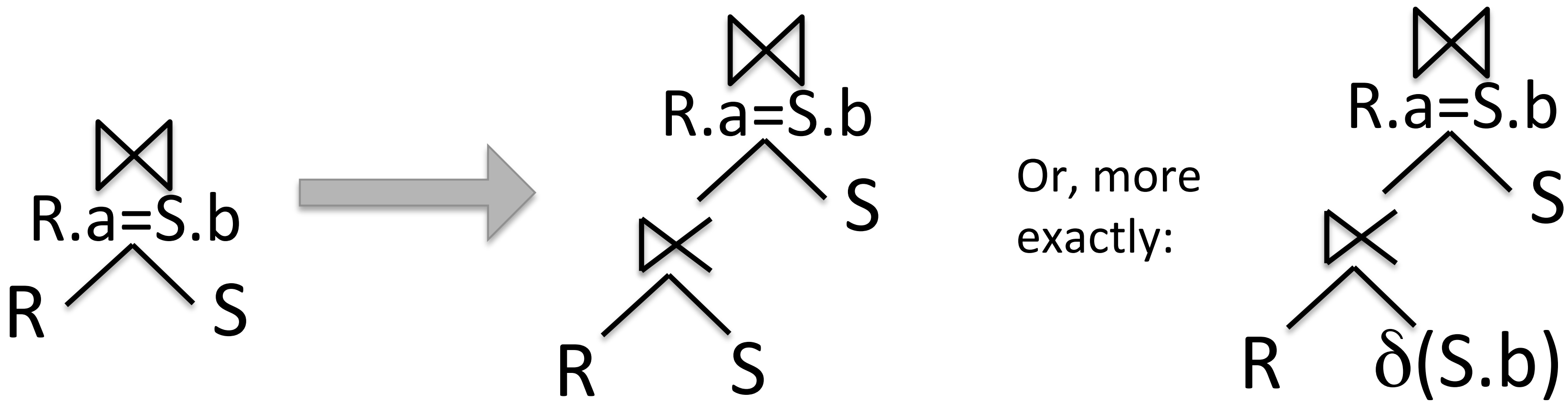
Plan pruning criteria if all the sites and network connections have equal performance:

- *Ship the smaller collection.*
- *Transfer to join partner or the query site*

This plan illustrates total effort != response time

Distributed query optimization technique: semijoin reducers

- R join S = (R semijoin S) join S



  - Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-matching R tuples*
  - Symetrical alternative: R join S = R join (S semijoin R)
  - This gives one more alternative in every join → search space explosion
  - Heuristics [Stocker, Kossmann et al., ICDE 2001]

Distribution of control in distributed relational DBs ('70s)

Servers DB1@site1: R1(a,b), S1(a,c)

Server DB2@site2: R2(a,b), S2(a,c),

Server DB3@site3: R3(a,b),
S3(a,c) defined as:

```
select * from DB1.S1 union all
select * from DB2.S2 union all
select R1.a as a, R2.b as c from DB1.R1 r1, DB2.R2 r2
where r1.a=r2.a
```

DB3@site3 decides what to import from site1, site2 (« hard links »)

Site1, site2 are independent servers

Also: replication policies, distribution etc. (usually with one or a few masters)

Modern distributed databases: H-Store (subsequently VoltDB)

- From the team of Michael Stonebraker (Turing Award, author of the Postgres system)
- Main goal: quick OLTP (**o**nline **t**ransaction **p**rocessing), e.g., sales, likes, posts...
- Built to run on **cluster** for horizontal scalability

- **Share-nothing architectur**e: each node stores tables **shards** (+ k replication for durability)

H-Store transactions

- Applications call **stored procedures** = code which also contains SQL queries
  - Each contained SQL query is partially unknown (depends on parameters specified at runtime); H-Store "pre-optimizes" it
- 1 **transaction** = 1 call of a stored procedure
- Can be submitted to any node (together with parameters)
- The node can run the procedure up to the query(ies) → updated, completely known plan → transaction manager

Frequent concept in Big Data architectures: shards

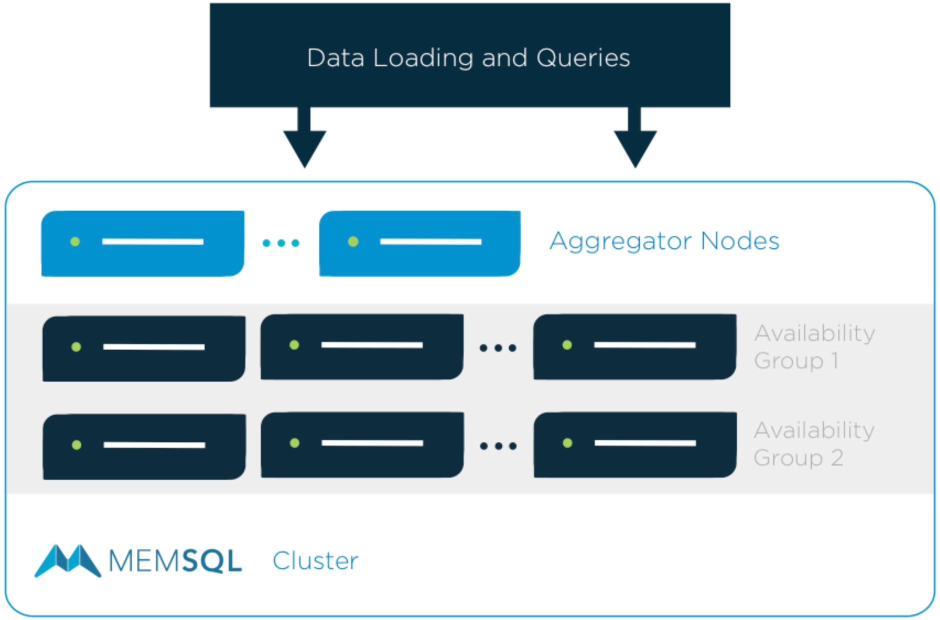
- **Shard** = small fragment of a data collection (e.g., a table)
- The assignment of data items (e.g. tuples) into shards is often done by **hashing** on tuple key

  - The table must have at least one key
  - Hashing ensures (with high probability) uniform distribution
- Key-based hashing is used as a mechanism for implementing distributed data catalogs. We will encounter it often.

Modern distributed RDB: MemSQL

MemSQL runs with

- a **master aggregator**, responsible of the metadata (catalog)
- possibly more aggregators
- at least one **leaf**, each of which stores part(s) of some table(s)
- In each leaf, there are **partitions** (by default: 1 per CPU core)



Availability group: a set of machines + a set of replica machines (one-to-one)

Query processing in MemSQL

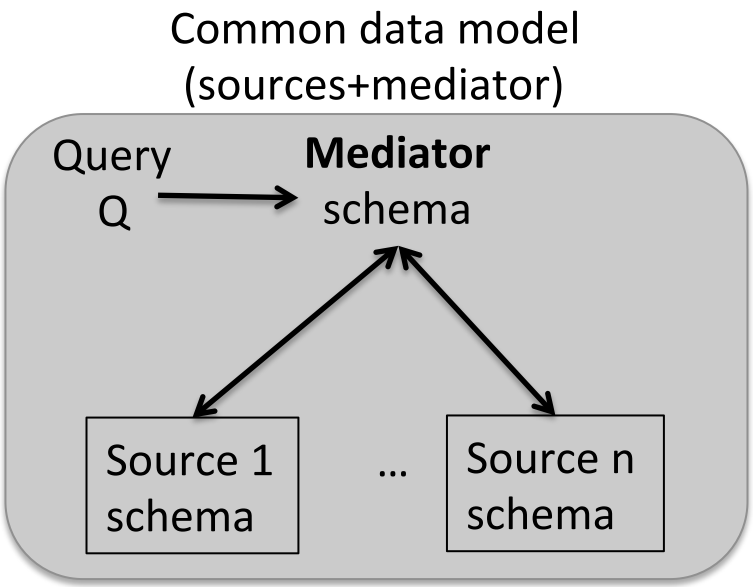
- **Indexes** managed within each partition
- In general, every query is run with a level of parallelism equal to the number of partitions

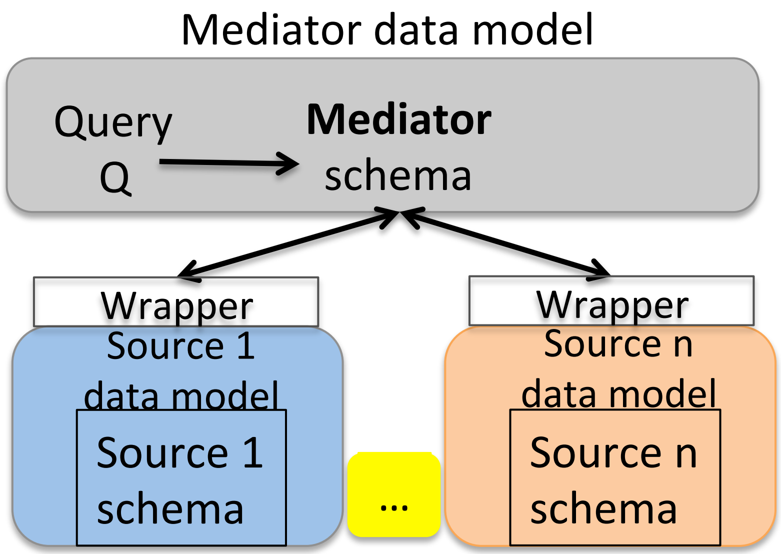
- **Select** queries are executed by the leaves which hold some partition(s) with data matching the query
- **Aggregation** queries run at the leaves involved and at the aggregator(s)
- **Join** queries
  - Easy if one input is a *reference* (small) table: one that is replicated fully to every machine in the cluster
  - Otherwise, they recommend sharing the shard key across tables to be joined
  - Otherwise, joins will incur data transfer within the cluster.

MEDIATOR SYSTEMS

Mediator systems

- A set of **data sources**, of the same or different data model, query language; source schemas
- A **mediator** data model and mediator schema
- Queries are asked against the mediator schema

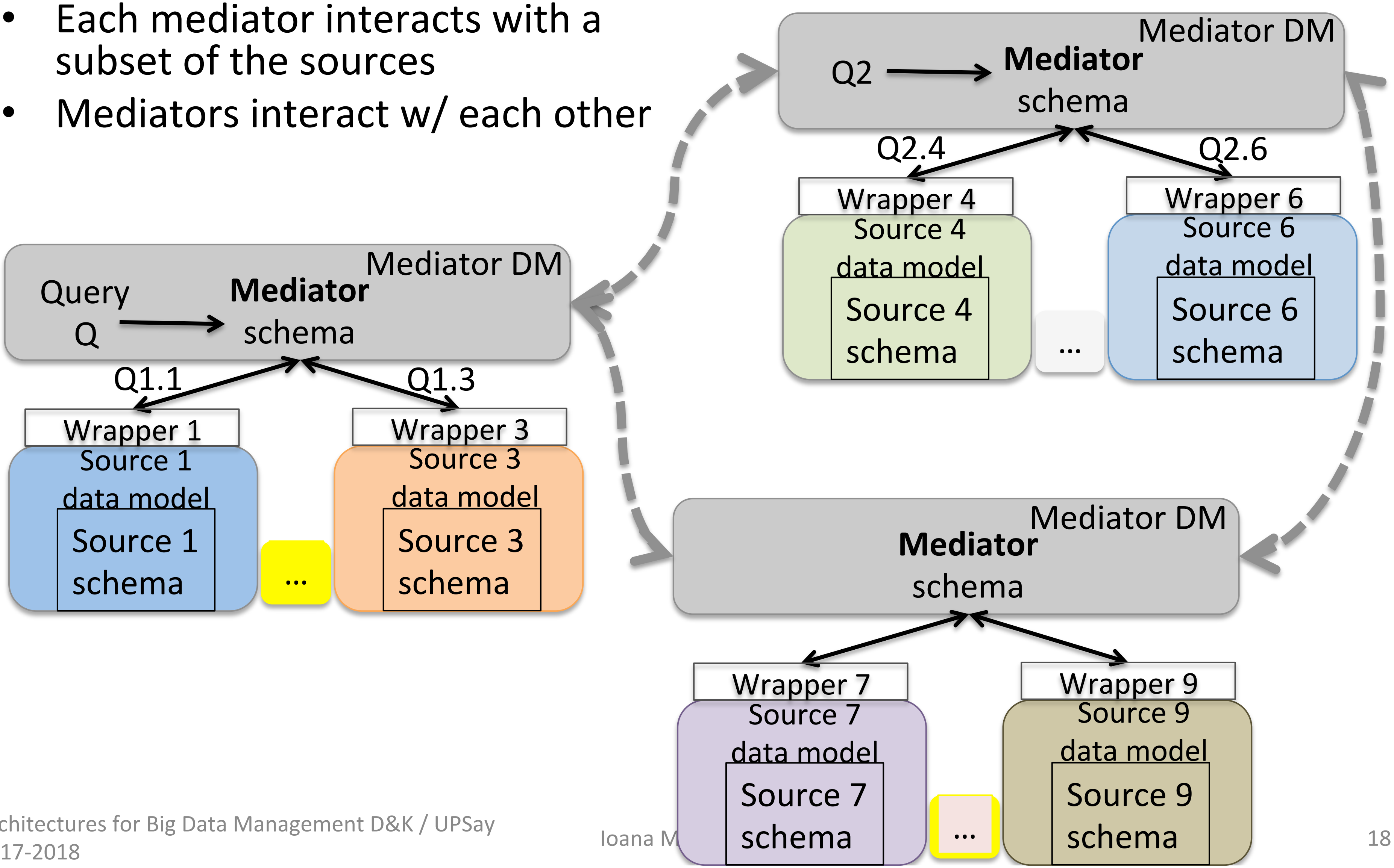




- **ACID**: mostly read-only; **size**: small
- **Control**: Independent publishing; mediator-driven integration

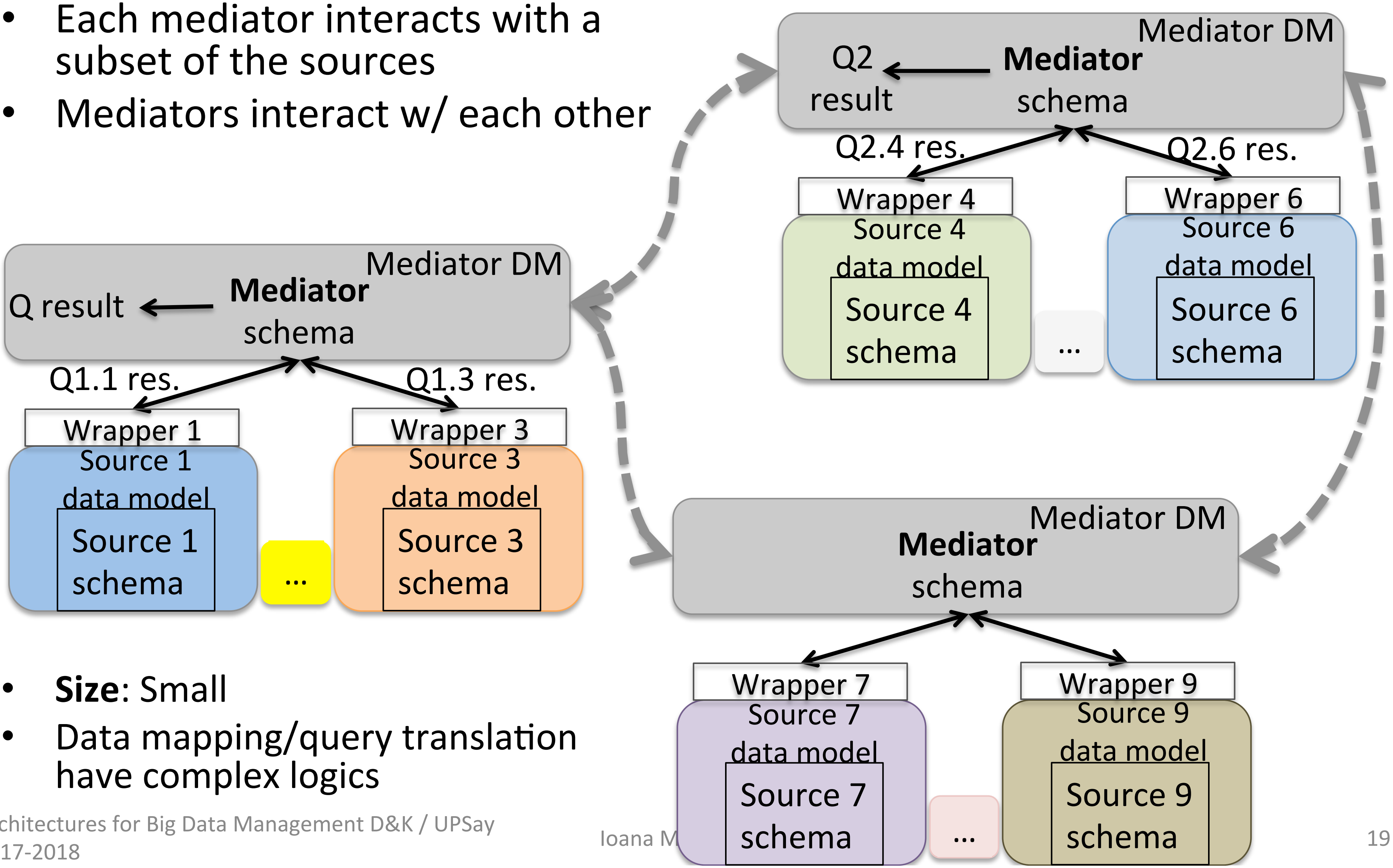
Many-mediator systems

- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other



Many-mediator systems

- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other



- **Size**: Small
- Data mapping/query translation have complex logics

Connecting the source schemas to the global schema

Example scenario:

Source **s1** has the schema:
    ParisHotels(street, name, roomPrice)
Source **s2** has the schema:
    LyonHotel(street, name, roomDesc, roomPrice)
Source **s3** has the schema:
    Restaurants(city, street, name, rating)
The **global schema** is:
    Hotel(city, street, name, descr, price)
    Restaurants, street, name, rating)

Connecting the source schemas to the global schema: Global-as-view (GAV)

s1:ParisHotels(street, name, roomPrice)

s2:LyonHotel(street, name, roomDesc, roomPrice)

s3:Restaurant(city, street, name, rating)

Global: Hotel(city, street, name, descr, price),
Restaurant(city, street, name, rating)

<u>Defining Hotel as a view over the source schemas:</u>

```
define view Hotel as
select 'Paris' as city, street, name, null as roomDesc, roomPrice as price
from s1:ParisHotels
union all
select 'Lyon' as city, street, name, descr as roomDesc, price
from s2:LyonHotel
```

<u>Defining Restaurant as a view over the source schemas:</u>

```
define view Restaurant as select * from s3:Restaurant
```

Query processing in global-as-view (GAV)

define view Hotel as
select 'Paris' as city, street, name, null as roomDesc, roomPrice as price
from s1:ParisHotels

union all

select 'Lyon' as city, street, name, descr as roomDesc, price
from s2:LyonHotel

Query:

select * from Hotel where city='Paris' and price<200 →

select * from (select 'Paris' as city... union... select 'Lyon' as city...) where city='Paris' and price < 200 →
select * from (select 'Paris' as city...) where city='Paris' and price < 200 →

select * from s1:ParisHotels where price < 200

Query processing in global-as-view (GAV)

define view Hotel as
select 'Paris' as city, street, name, null as roomDesc, roomPrice as price
from s1:ParisHotels
union all
select 'Lyon' as city, street, name, descr as roomDesc, price from s2:LyonHotel
define view Restaurant as select * from s3:Restaurant

Query:

select h.street, r.rating from Hotel h, Restaurant r where h.city=r.city and r.city='Lyon' and and h.street=r.street and h.price<200 →

select h.street, r.rating from (select 'Paris' as city... from s1:ParisHotels union all select 'Lyon' as city... from s2:LyonHotel) h, (select * from s3:Restaurant) r where h.city=r.city and r.city='Lyon' and h.street=r.street and h.price<200 →

select h.street,r.rating from (select ... from s2:LyonHotel) h, s3:Restaurant r where r.city='Lyon' and h.street=r.street and h.price<200 →

select h.street, r.rating from s2:LyonHotel h, s3.Restaurant r where r.city='Lyon' and h.price<200 and h.street=r.street

Concluding remarks on global-as-view (GAV)

- Query processing = view unfolding: replacing the view name with its definition and working out simple equivalences from there
  - Allows to push to each data source as much as it can do (trusted heuristic)
- Weakness: changes in the data sources require changes of the global schema and, in the worst case, of all applications written based on this global schema
  - E.g., if s4:GrenobleHotel joins or s2:LyonHotel leaves the system
  - Worst case: the global schema comprises a table T12 that joins S1:T1 with S2:T2, then S2:T2 leaves the system → T12 is empty, and S2:T2 data is inaccessible to users

Connecting the source schemas to the global schema: Local-as-view (LAV)

s1:ParisHotels(street, name, roomPrice)

s2:LyonHotel(street, name, roomDesc, roomPrice)

s3:Restaurant(city, street, name, rating)

Global**:** Hotel(city, street, name, descr, price), Restaurant(city, street, name, rating)

<u>Defining s1:ParisHotels as a view over the global schema:</u>

define view s1:ParisHotels as
select street, name, price as roomPrice
from Hotel where city='Paris'

<u>Defining s2:LyonHotel as a view over the global schema:</u>

define view s2:LyonHotel as
select street, name, descr as roomDesc, price as roomPrice
from Hotel where city='Lyon'

<u>Defining s3:Restaurant as a view over the global schema:</u>

define view s3:Restaurant as
select * from Restaurant

Query processing in Local-as-View (LAV)

define view s1:ParisHotels as
select street, name, price as roomPrice
from Hotel where city='Paris'
define view s2:LyonHotel as
select street, name, descr as roomDesc, price as roomPrice from Hotel where city='Lyon'
define view s3:Restaurant as
select * from Restaurant

Query:
select h.street, h.price, r.rating from Hotel h, Restaurant r where r.city=h.city and h.street=r.street

Query processing in Local-as-View (LAV)

define view **s1:ParisHotels** as
select street, name, price as roomPrice
from Hotel where city='Paris'
define view **s2:LyonHotel** as
select street, name, descr as roomDesc, price as roomPrice from Hotel where city='Lyon'
define view **s3:Restaurant** as
select * from Restaurant

Step 1: identify potentially useful views

Query:
select h.street, h.price, r.rating from Hotel h, Restaurant r where r.city=h.city and h.street=r.street

Query processing in Local-as-View (LAV)

s1:ParisHotels
or s2:LyonHotel
s3:Restaurant

- **Query:**
select h.street, h.price, r.rating from Hotel h, Restaurant r where r.city=h.city and h.street=r.street
- **Step 2**: generate view combinations that may be used to answer the query
  - s1:ParisHotels and s3:Restaurant
  - s2:LyonHotels and s3:Restaurant
- **Step 3**: for each view combination and each view, check:
  - If the view returns enough attributes: we need the attributes returned by the query and those on which possible query joins are based
  - If the view selections (if any) are compatible with those of the query
- **Step 4**: for each view combination, add the necessary joins among the views, possibly selections and projections → rewriting
- **Step 5**: return the union of the rewritings thus obtained

Query processing in Local-as-View (LAV)

define view s1:ParisHotels as... from Hotel where city='Paris'
define view s2:LyonHotel as... from Hotel where city='Lyon'
define view s3:Restaurant as select * from Restaurant

Query:
select h.street, h.price, r.rating from Hotel h, Restaurant r where
r.city=h.city and h.street=r.street

Rewriting of the query using the views:

select h1.street, h1.price, r3.rating
from s1:ParisHotels h1, s3:Restaurant r3
where h1.city=r3.city and h1.street=r3.street
union all
select h2.street, h2.price, r3.rating
from s2:LyonHotels h2, s3:Restaurant r3
where h2.city=r3.city and h2.street=r3.street

Concluding remarks on Local-as-View (LAV)

Query processing

- The problem of finding all rewritings given the source and global schemas and the view definitions = view-based query rewriting, NP-hard in the size of the (schema+view definitions). These are often much smaller than the data
- Fundamental concept: *containment mappings* between a view and the query [Chandra and Merlin, 1978]

The schema definition is more robust:

- One can independently add/remove sources from the system without the global schema being affected at all
- Thus, no application needs to be aware of the changes in the schema

Containment mappings

variable in the query head=projection

variable

variable reuse=join

constant= selection

Datalog query notation: q(x) :- R(x), S(x), T(x,5)

- This is exactly the same as: q(m) :-R(m), S(m), T(y,5)
- Variable names do not matter; the position in the atom matters

Containement mapping from Q1 into Q2

- function ϕ that assigns to each variable of Q1 one variable or constant in Q2, and is the identity for constants, such that the image of the atoms of Q1 through ϕ is in Q2

Examples

View v1(x) :- R(x), S(x); mapping of v1 into q: x→x. R(x), S(x) in q.

View v2(x) :- R(x), T(x,y); mapping into q: x→x, y→5. R(x), T(x,5) in q.

View v3(x) :- R(x), T(x,7); no mapping, as 7 cannot be mapped to 5

Equivalent view-based rewriting

Given a query q and a set of views V={v1, ... vn}, an **equivalent rewriting of q using V** is a query over v1, ..., vn which gives the same result as q over any database instance

Example (1)

q(x) :- R(x), S(x), T(x,5)

v1(x,y) :- R(x), T(x,y)
v2(x) :- R(x), S(x)

Equivalent rewriting: rew(x) :- v1(x,5), v2(x)

Equivalent view-based rewriting

Example (2)

q(x) :- R(x,y), S(y,z), R(z,t), S(t,u)

v(x,y,z) :- R(x,y), S(y,z)

Two containment mappings:

x→x, y→y, z→z and x→z, y→t, z→u

Equivalent rewriting: rew(x) :- v(x,y,z), v(z,t,u)

- "join v with itself on third attribute=first attribute, then return the first attribute from the first occurrence of v"
- Uses the views twice, once for each containment mapping
- rew(x) and q(x) have exactly the same results for any instance (data content) of the relations R and S

View-based rewriting vs. containment mappings

- Theorem (Chandra and Merlin):

A view can be used in an equivalent rewriting of a query only if there is a containment mapping from the the view into the query

- Searching for containment mappings is an NP-hard problem
- Once containment mappings are found, we still have to check how and if views can be combined to rewrite the query

View-based query rewriting

v1(x) :- R(x,y), S(y,z)   v2(u) :- S(u,w),T(w,p)

Can we rewrite q1(x,m) :- R(x,y), S(y,z), T(z,m) using v1 and v2?

Can we rewrite q2(x) :- R(x,y), S(y,z), T(z,m) using v1 and v2?

Now assume v3(u, p) :- S(u,w),T(w,p).

Can we rewrite q1(x,m) using v1 and v3?

Can we rewrite q2(x) using v1 and v3?