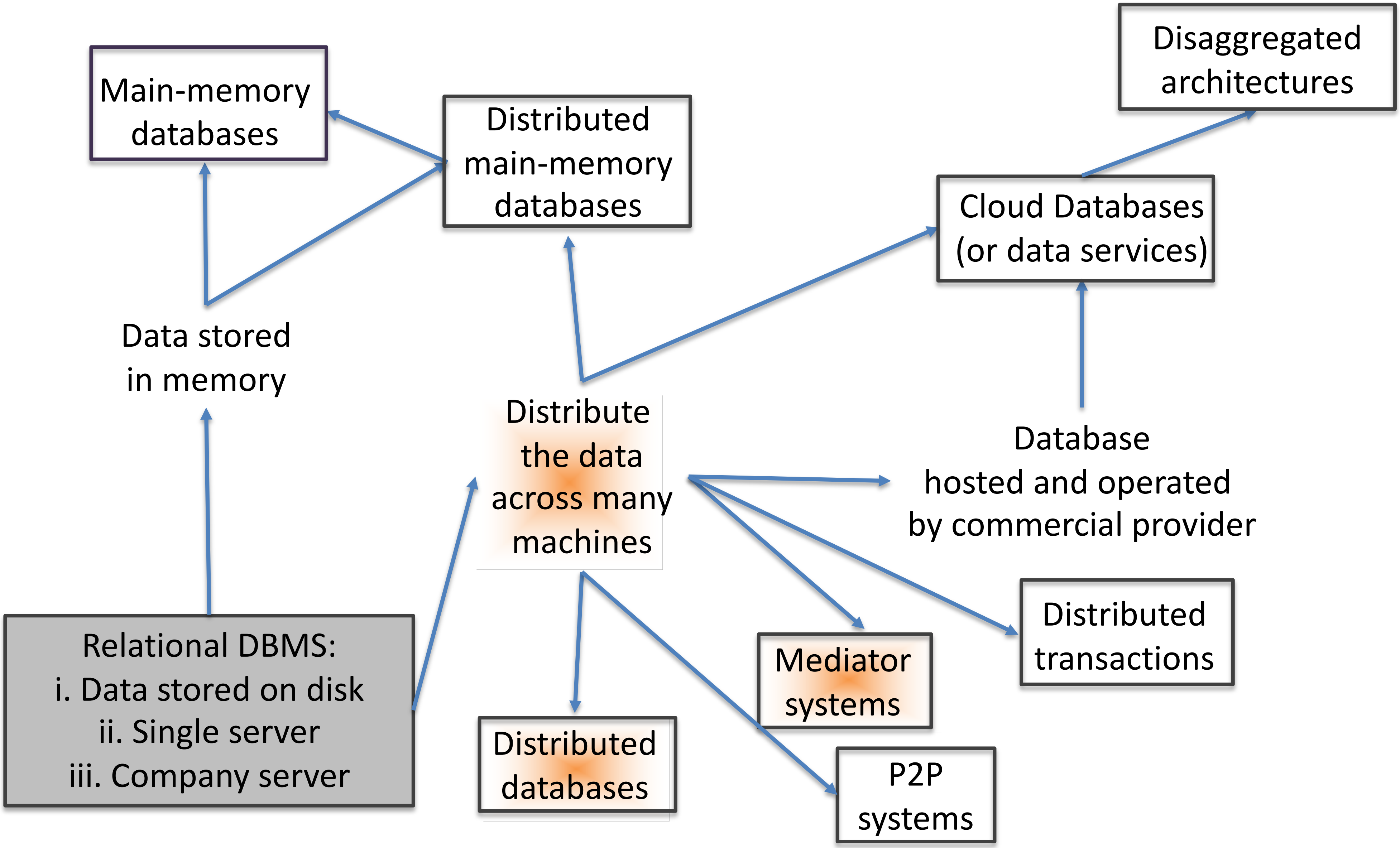
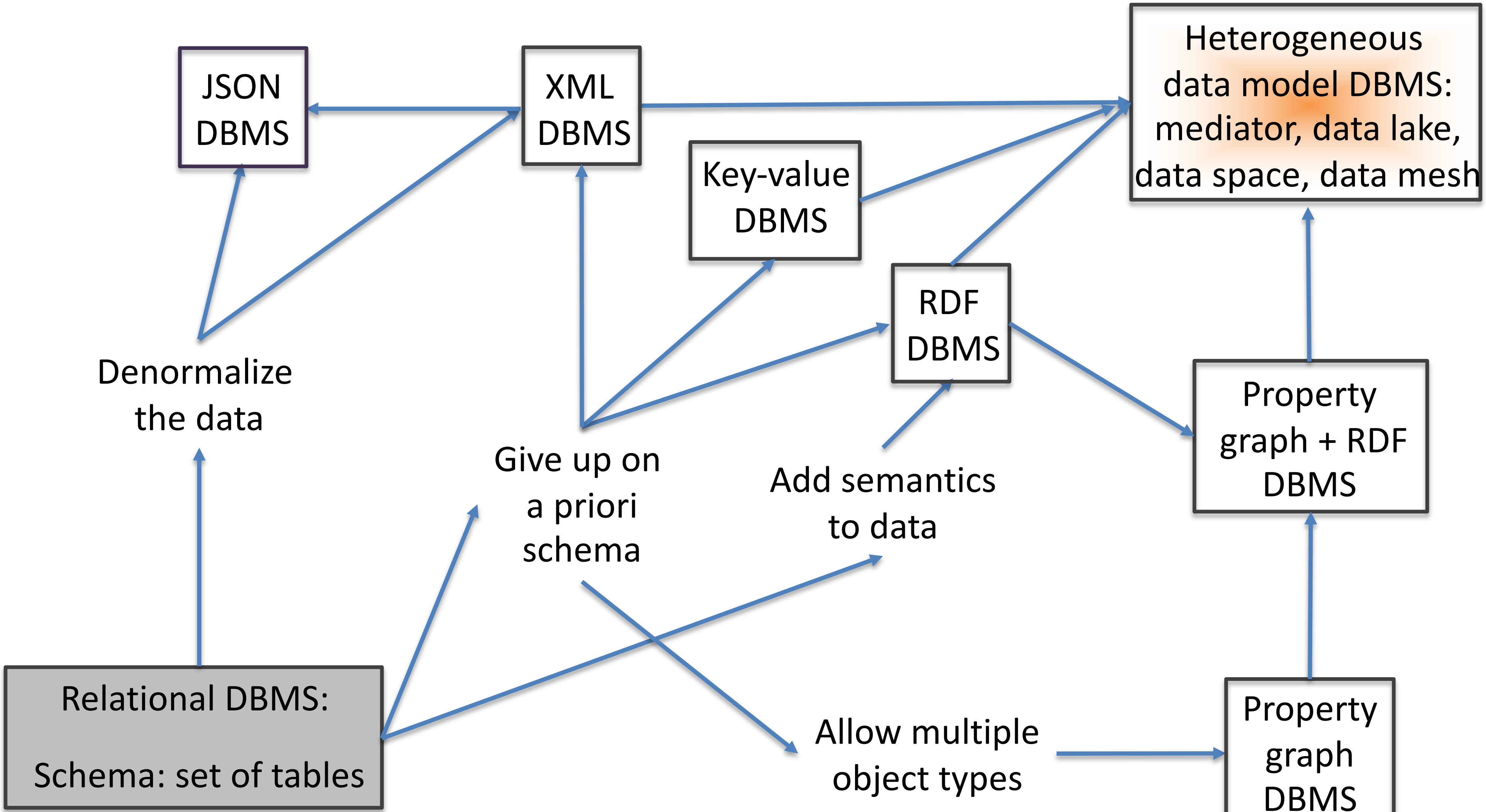
BIG DATA ARCHITECTURES:
- INTRODUCING DISTRIBUTION
- HANDLING HETEROGENEITY

From databases to Big Data



From databases to Big Data



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
- **Hardware**:
  - Hardware type: from disk to memory
  - Scale of distribution: small (~10-20 sites) or large (~10.000 sites)
- **ACID** properties
- **Interoperability and control**:
  - Who decides: data structure, data publication, data placement
  - What is the logical relation between datasets, how do they relate?
  - Who does what when processing queries or updates

DISTRIBUTED RELATIONAL DATABASES

Distributed relational databases

- Oldest distributed architecture ('70s): IBM System R*
- Illustrate/introduce the main priciples
- Data is relational (tables).
- Data is distributed among many *nodes* (*sites*, *peers*...)
  - **Data catalog**: information on which data is stored where
    - Catalog stored at a master/central server.
    - E.g., « Paris sales are stored in Paris », « Lyon sales are stored in Paris », « Client data is stored in London », etc.
- Queries are distributed (may come from any site)
  - First analyzed through catalog
- Query processing is distributed
  - Operators may run on different sites → network transfer

Traditional distributed relational databases (since 1970)

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

Query evaluation in distributed relational database: query unfolding

DB1: R1(a,b), S1(a,c)

DB2: R2(a,b), S2(a,c),

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

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

Query on DB3:

```
select a
from S3
where a = 3;
```

The query is formulated on S3, but there is no actual data there!

- The query is reformulated (or unfolded) based on the definition of S3

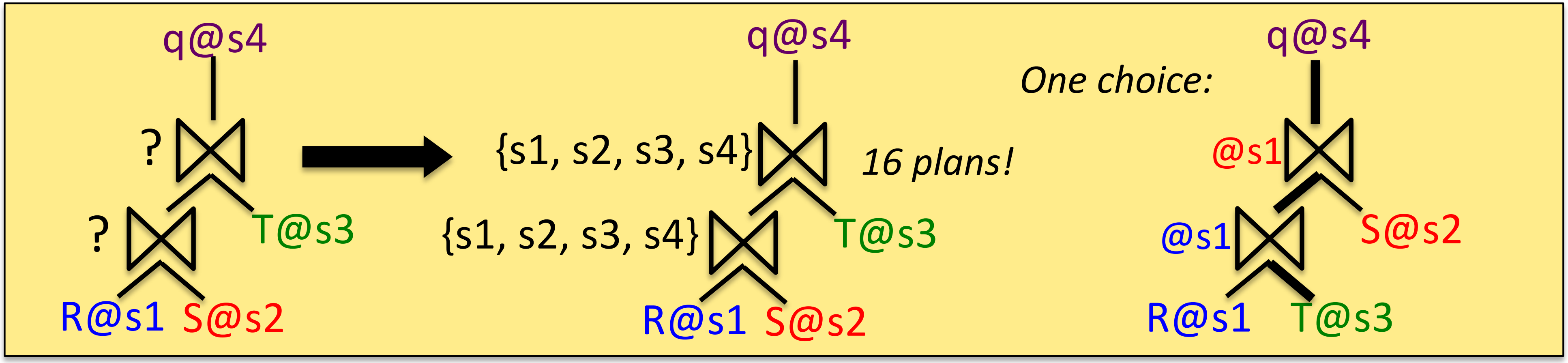
In classical DBMSs, a query over a view is also unfolded (demo)

How is a query unfolded?

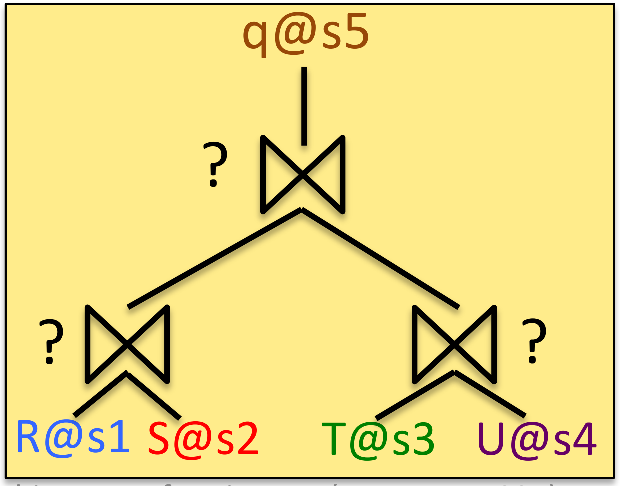
- Based on logical algebra

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

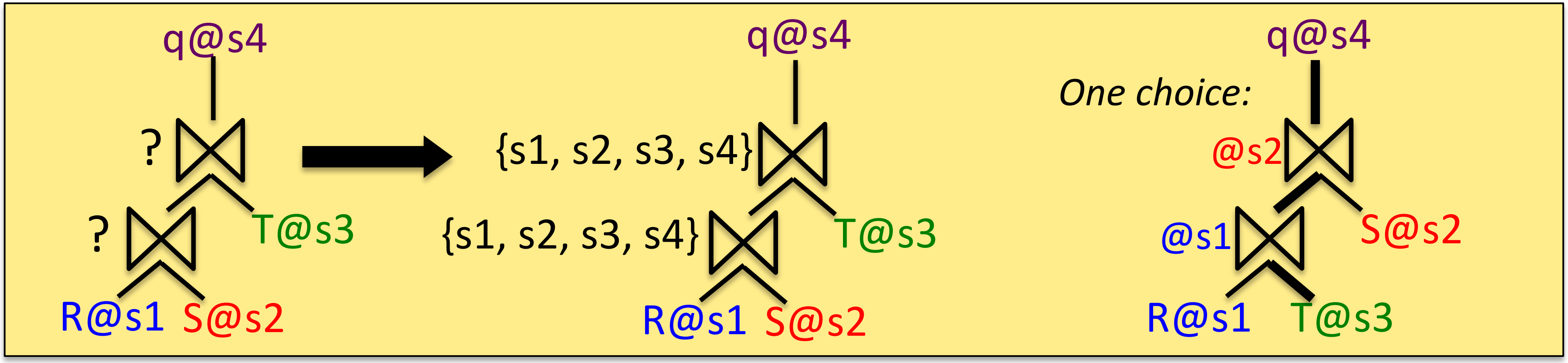


Plan pruning criteria if all the sites and network connections have equal performance:

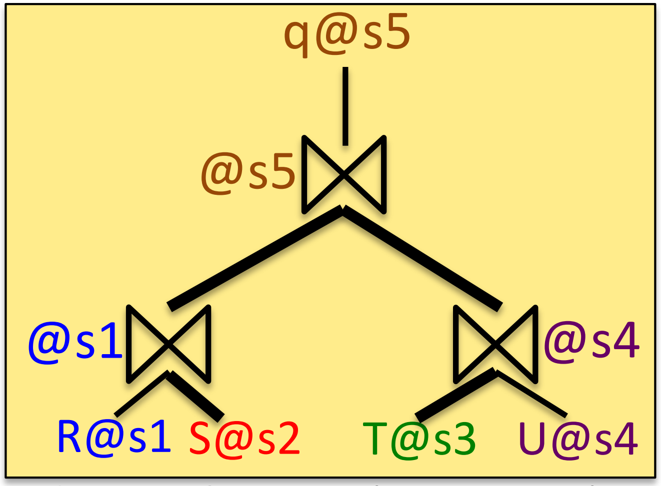
- *Ship the smaller collection*

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

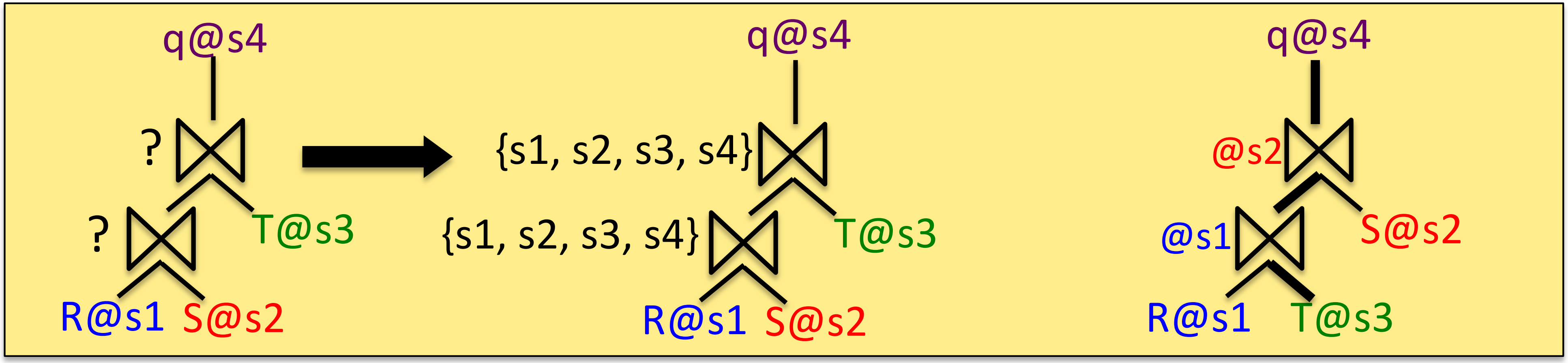


Plan pruning criteria if all the sites and network connections have equal performance:

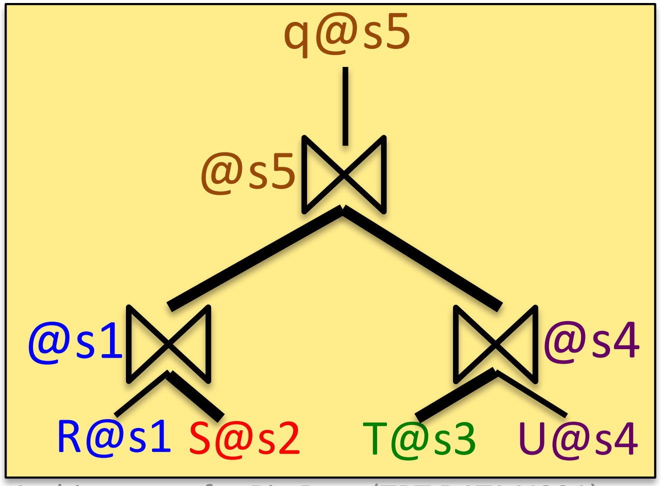
- *Ship the <u>smaller</u> collection*
- *Transfer to join partner or the query site*

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



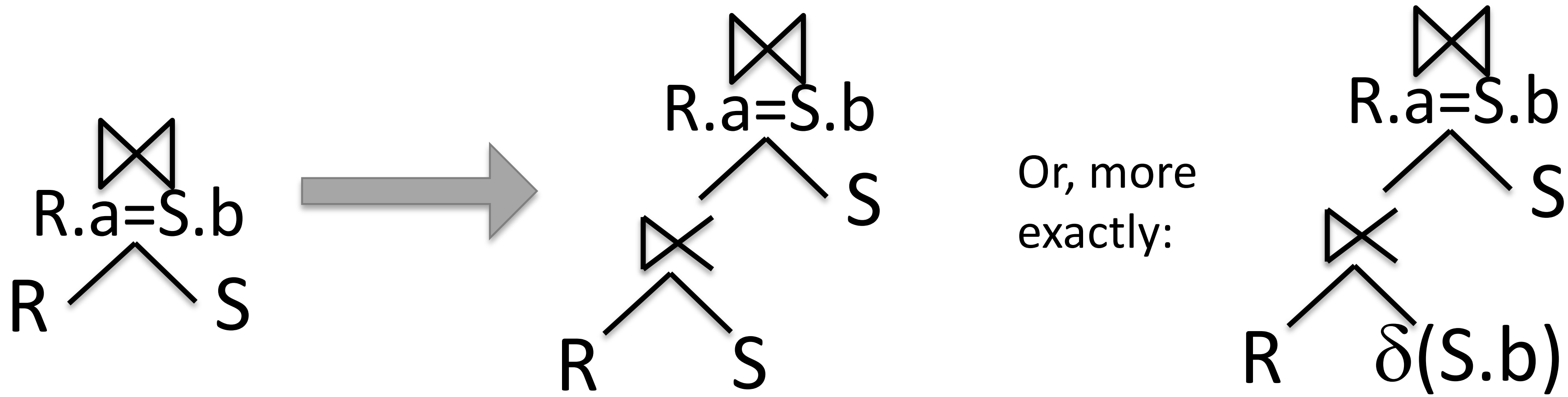
Plan pruning criteria if all the sites and network connections have equal performance:

- *Ship the <u>smaller</u> collection.*
- *Transfer to join partner or the query site*

This plan illustrates total effort != response time

Distributed query optimization technique: semijoin reducers

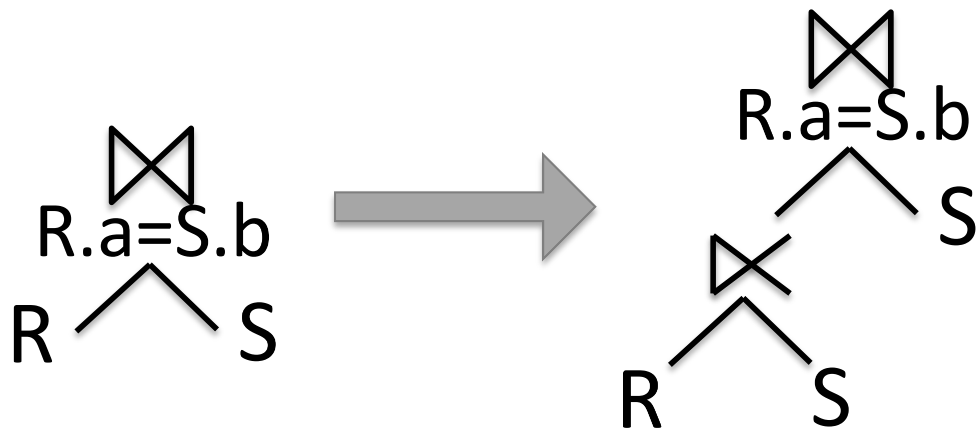
- R join S = (R semijoin S) join S



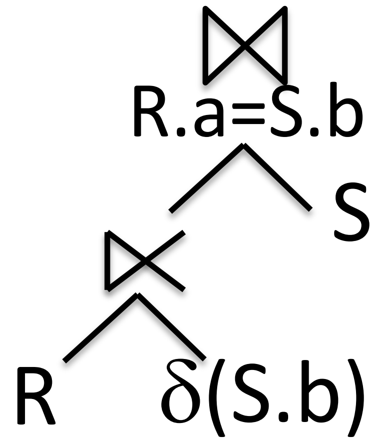
  - Useful in distributed settings to reduce transfers: *if the distinct S.b values* are smaller than *the non-joining R tuples*
  - Example: 1.000.000 tuples in R, 1.000.000 tuples in S, 900.000 distinct values of R.a, 10 distinct values of S.b

Distributed query optimization technique: semijoin reducers

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



Or, more exactly:



  - 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]

Data distribution vs. replication

- **Distribution**: splitting a dataset, e.g., a database, or a relation, among two or more distributed nodes

- **Replication**: copying a dataset, e.g., a database, or a relation on one or more sites.
  - To ensure *durability* even in the face of hardware (storage) destruction
  - To increase *availability* during a software crash at one site (until it is back again)
  - To increase *speed* for queries that run on a data replica which is close to the query

Modern distributed databases: H-Store (→ VoltDB)

- From the team of Michael Stonebraker (Turing Award, author of the PostgreSQL system)
  - H-Store: research prototype, 2008
  - VoltDB: commercial product issued from H-Store, 2010
- 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)

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** (or: evaluating a hash function) on tuple key

  - The table must have at least one key
  - Assume R.a is key of R. Then, for each tuple r from R:
    - Compute h(r.a) = k
    - Tuple r will be part of shard number k
  - Hashing ensures (with high probability) uniform distribution
- Key-based hashing is a very frequent data distribution mechanism!

Transactions in H-Store

- 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
  - E.g., *purchaseProduct(productID, clientID, cardNo)*
- 1 **transaction** = 1 call of a stored procedure
  - E.g., purchaseProduct(prod101, cl10, 12345678)
- Can be submitted to any node, together with parameters
- The node can run the procedure up to the query(ies) → detailed qistributed plan → transaction manager

HETEROGENEOUS DATA INTEGRATION

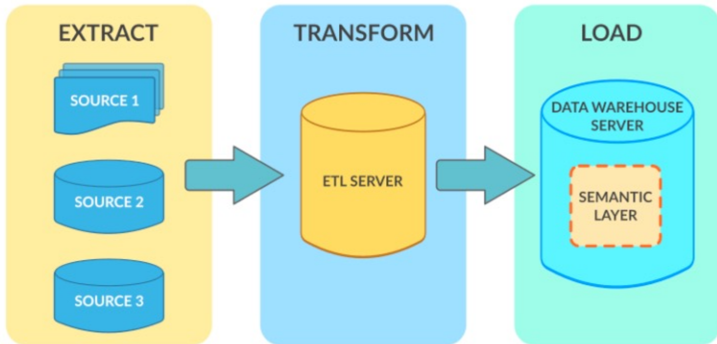
Heterogeneous data integration

- Heterogeneous data:
  - Tables with different schemas
  - Databases with different set of tables
  - Datasets in different data models (RDF, XML, relational…)
- Data integration:
  - Exploit heterogeneous data sources as if they were part of just one database
  - This typically means with a single consolidated schema
  - Thus, first task is: *understand* the sources and *how they relate to the global schema* we want

Heterogeneous data integration

- **Data integration**: treat several data sources as a single one, under a single schema
- Once the integration schema has been computed, either:
  - *Extract* the data from the sources, *transform* it into the global schema, *load* it into a **data warehouse (ETL)**

    or:
  - Devise a **mediator** which interacts with the sources and provides the illusion of a single database.

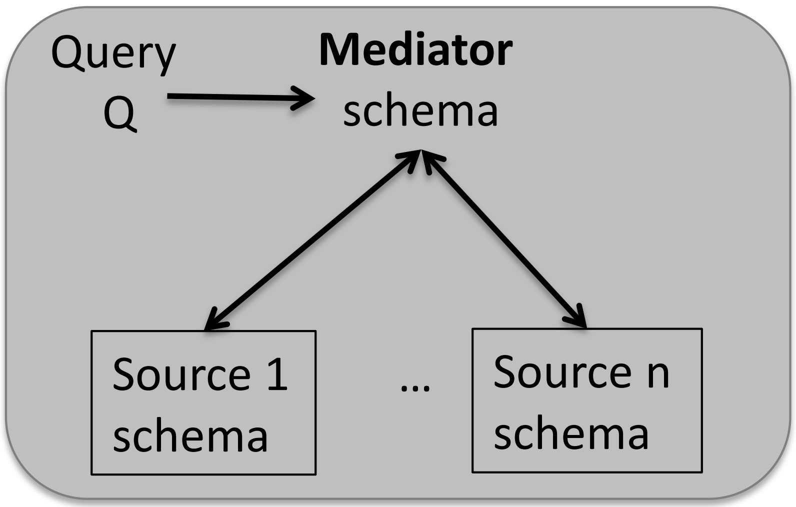


HETEROGENEOUS DATA INTEGRATION: MEDIATORS

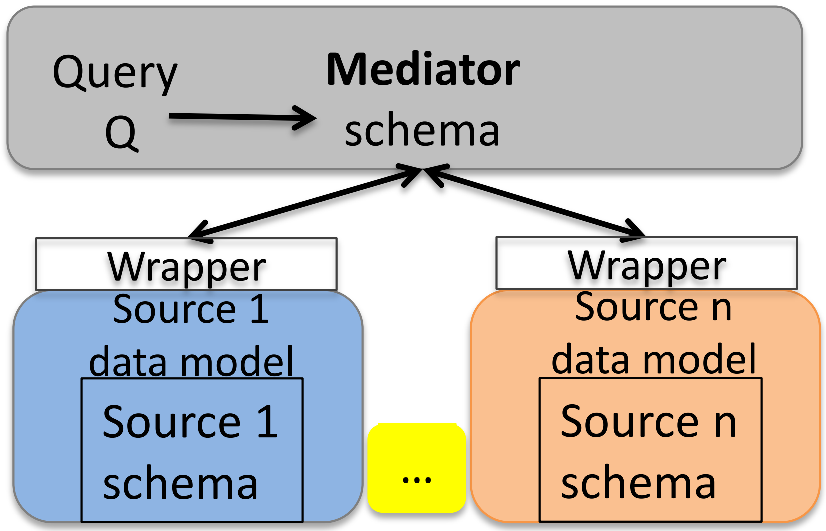
Mediator systems

- A set of **data sources**, each with: data model, query language, and schema (also called source schemas).
  - DM and QL may or may not differ across sources
- A **mediator** with its own DM, QL and mediator schema
  - Queries are asked against the mediator schema
- **Wrappers** interface the sources to the mediator's model

Common data model (sources+mediator)

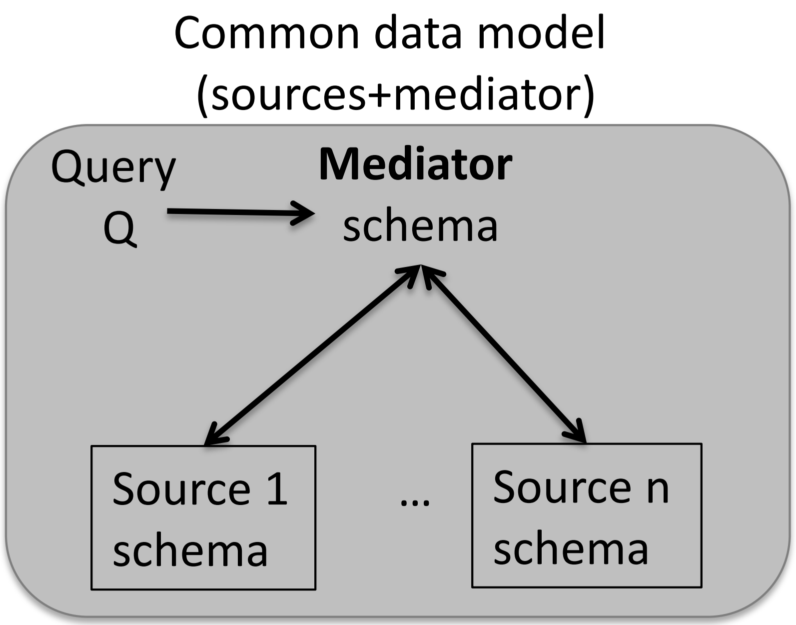


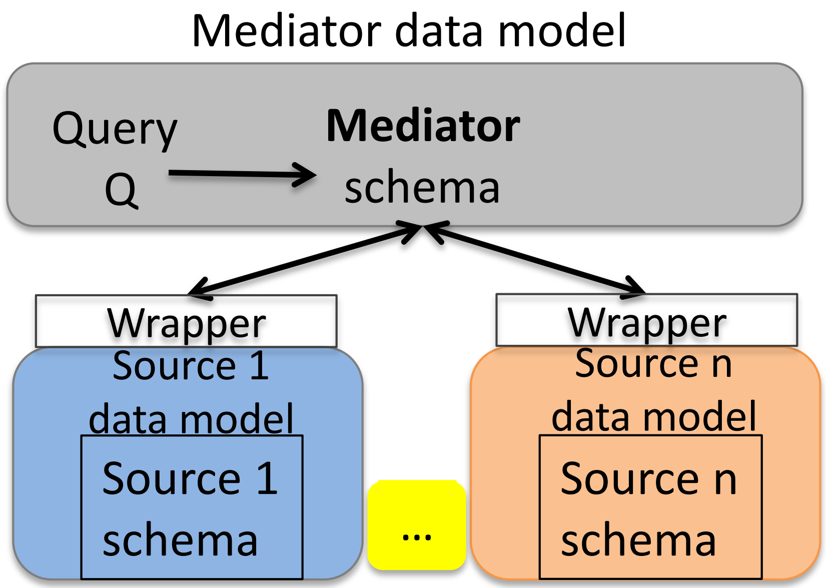
Mediator data model



Mediator systems

- A set of **data sources**, each with: data model, query language, and schema (also called source schemas).
  - DM and QL may differ across sources
- A **mediator** with its own DM, QL and mediator schema

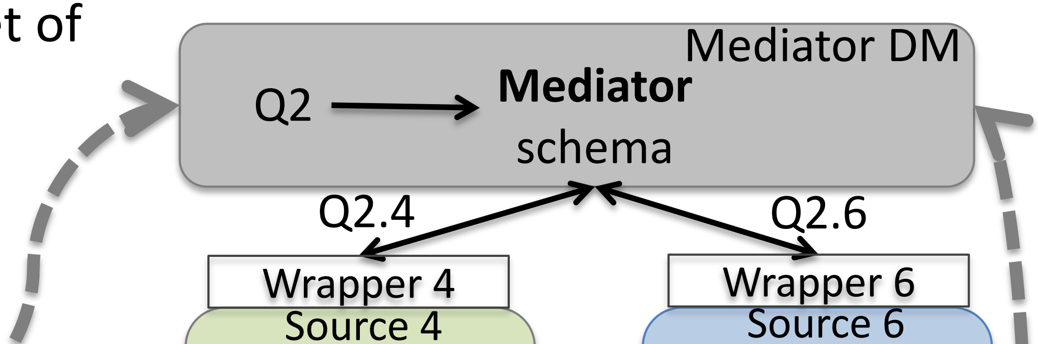


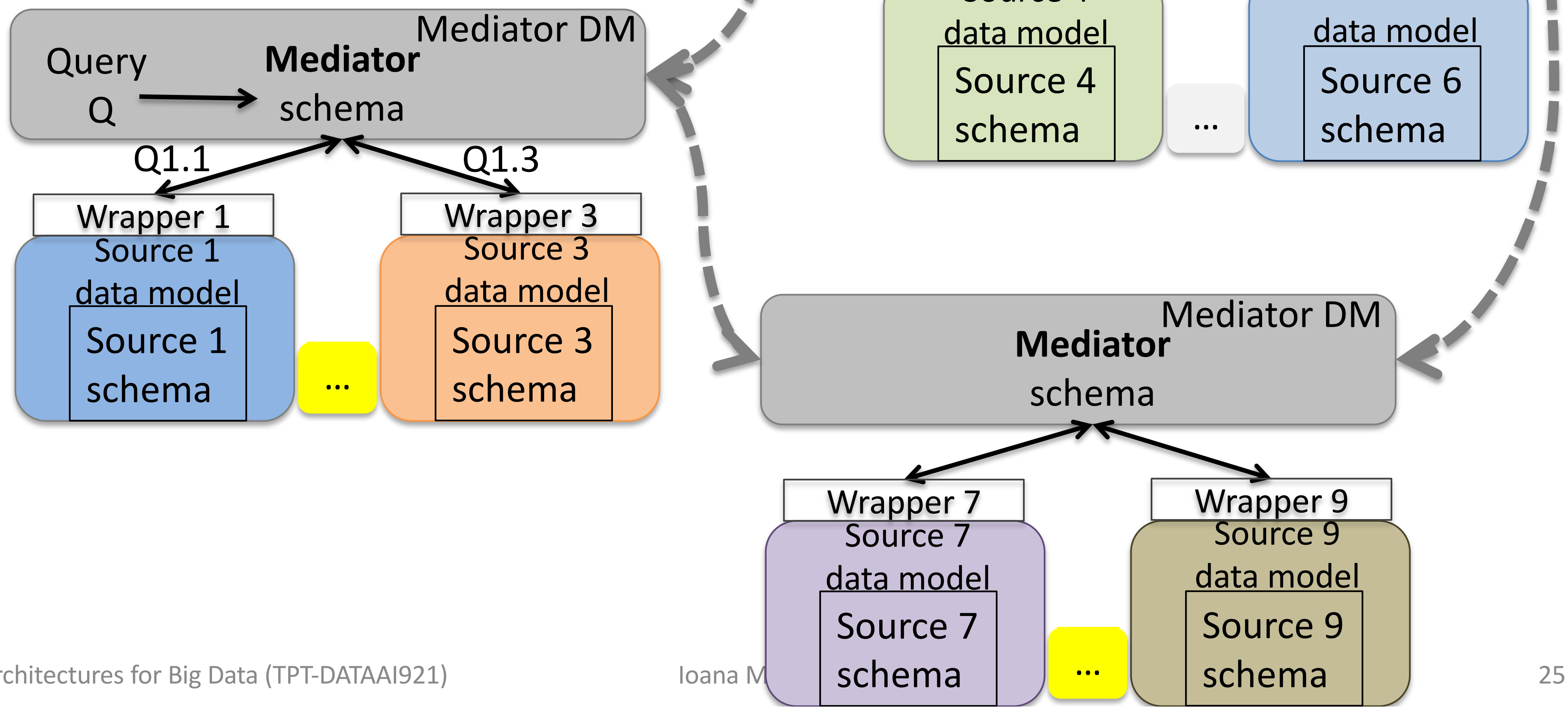


- **ACID**: mostly read-only; **size**: small
- **Control**: Independent publishing; mediator-driven integration

Many-mediator systems

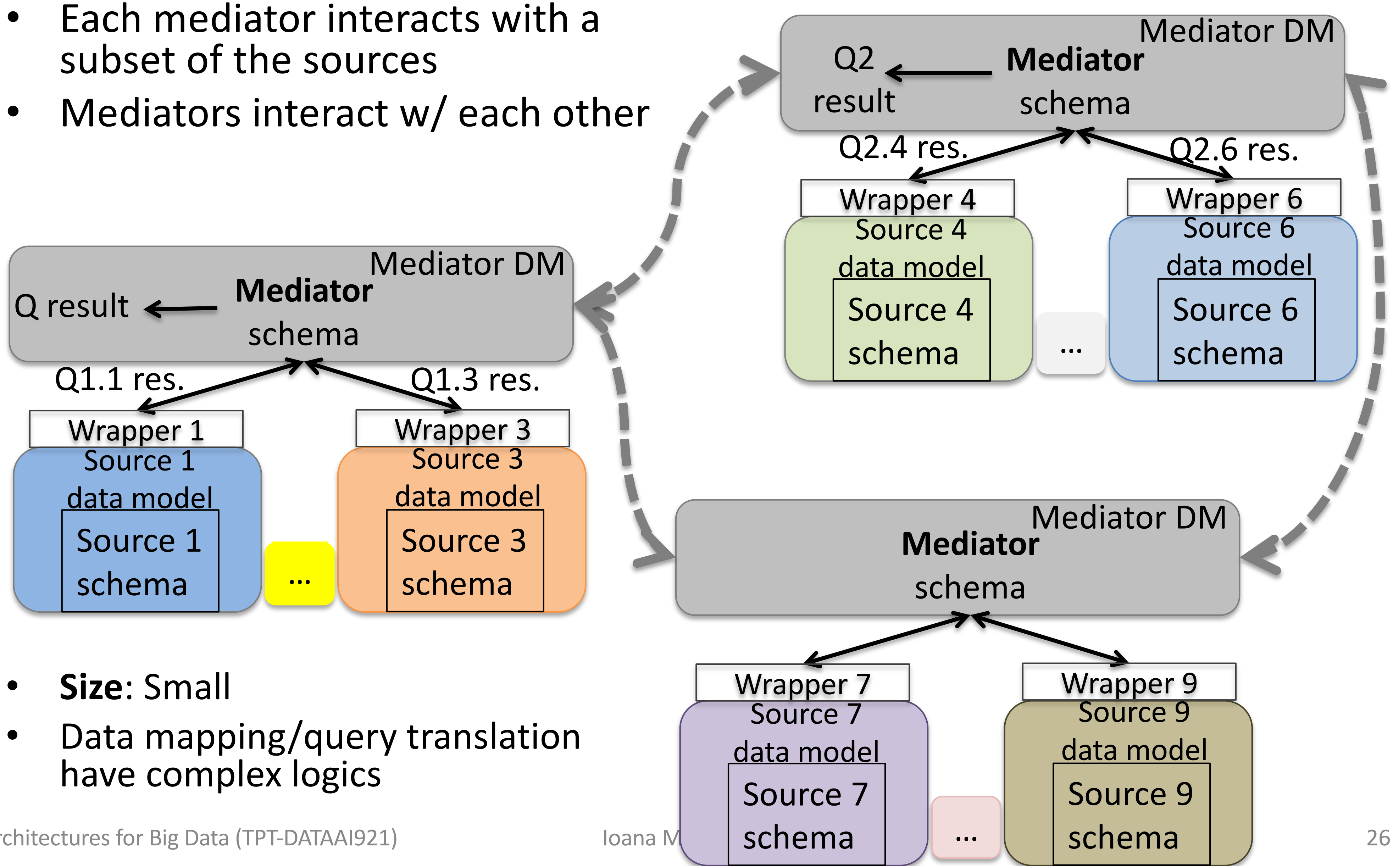
- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other
  - A mediator can play the role of a source for processing a given query





Many-mediator systems

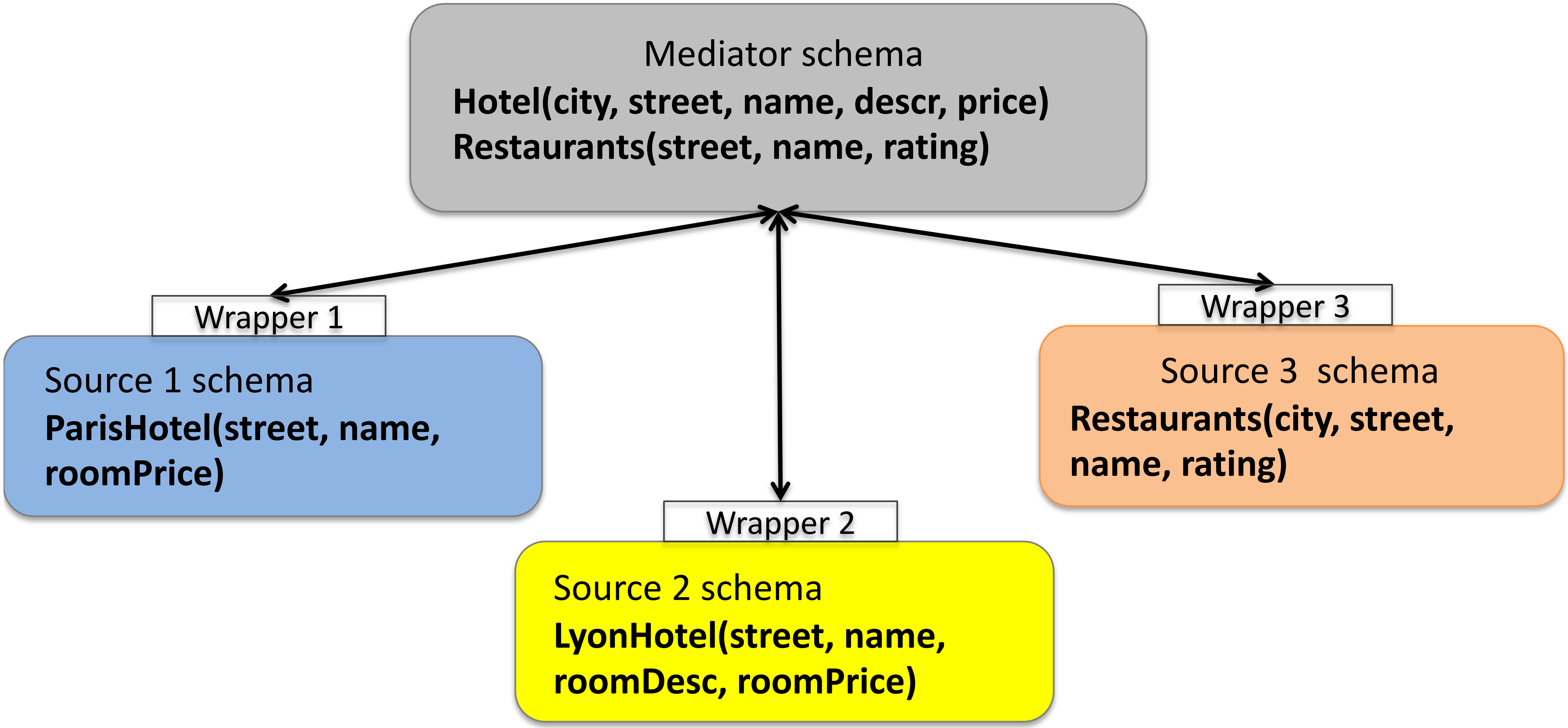
- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other



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

Connecting the source schemas to the global schema

- Sample scenario:



Connecting the source schemas to the global schema

- Data only exists in the sources.
- Applications only have access to, and only query, the mediator schema.

- How to **express the relation** between
  - the **mediator schema** acccessible to applications, and
  - the **source schemas** reflecting the real data
  - so that a query over the mediator schema can be **automatically translated** into a query over the source schemas ***?***
- Three approaches exist (see next)

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)

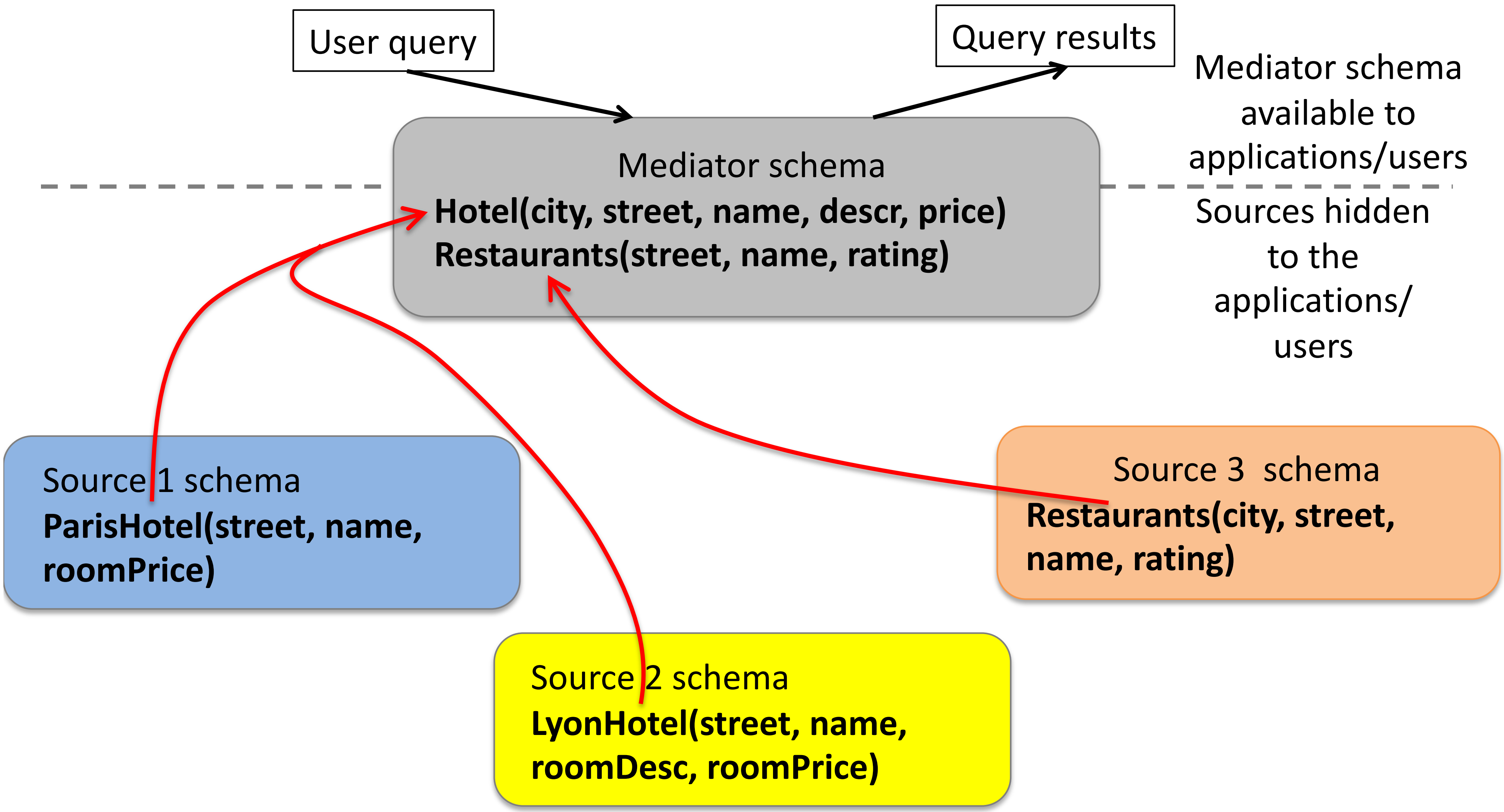
<u>Defining **Hotel** as a view over the source schemas:</u>

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

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

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

Connecting the source schemas to the global schema: Global-as-View



Query processing in global-as-view (GAV)

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

Query:

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

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

select * from (select 'Paris' as city...)
where city='Paris' and price < 200 *which becomes:*

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 Hotels h, Restaurant r where h.city=r.city and r.city='Lyon' and and h.street=r.street and h.price<200 *becomes:*

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

which becomes:

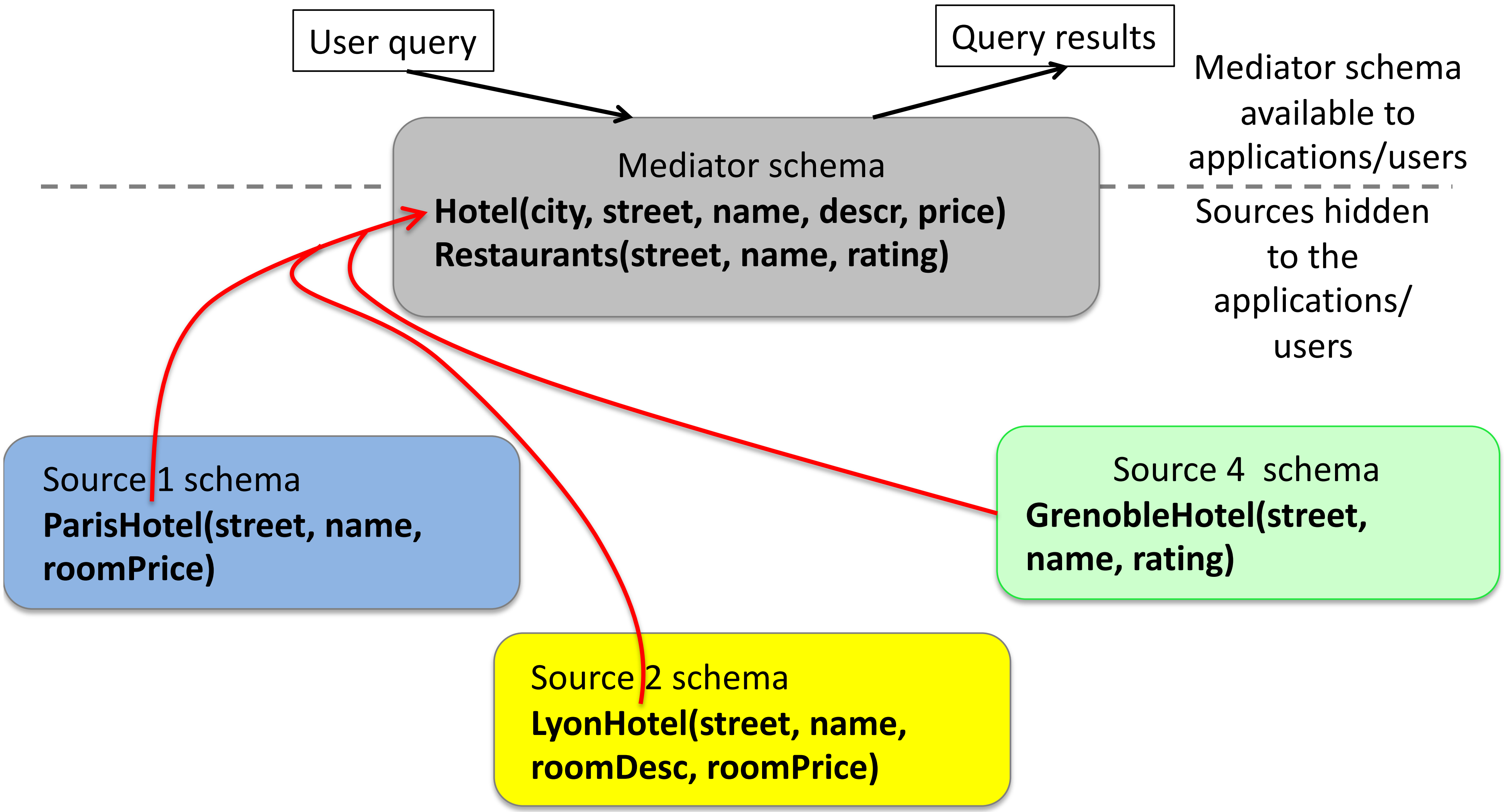
select h.street,r.rating from (select ... from s2:LyonHotel) h, s3:Restaurant r where r.city='Lyon' and h.street=r.street and h.price<200 *which becomes:*

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

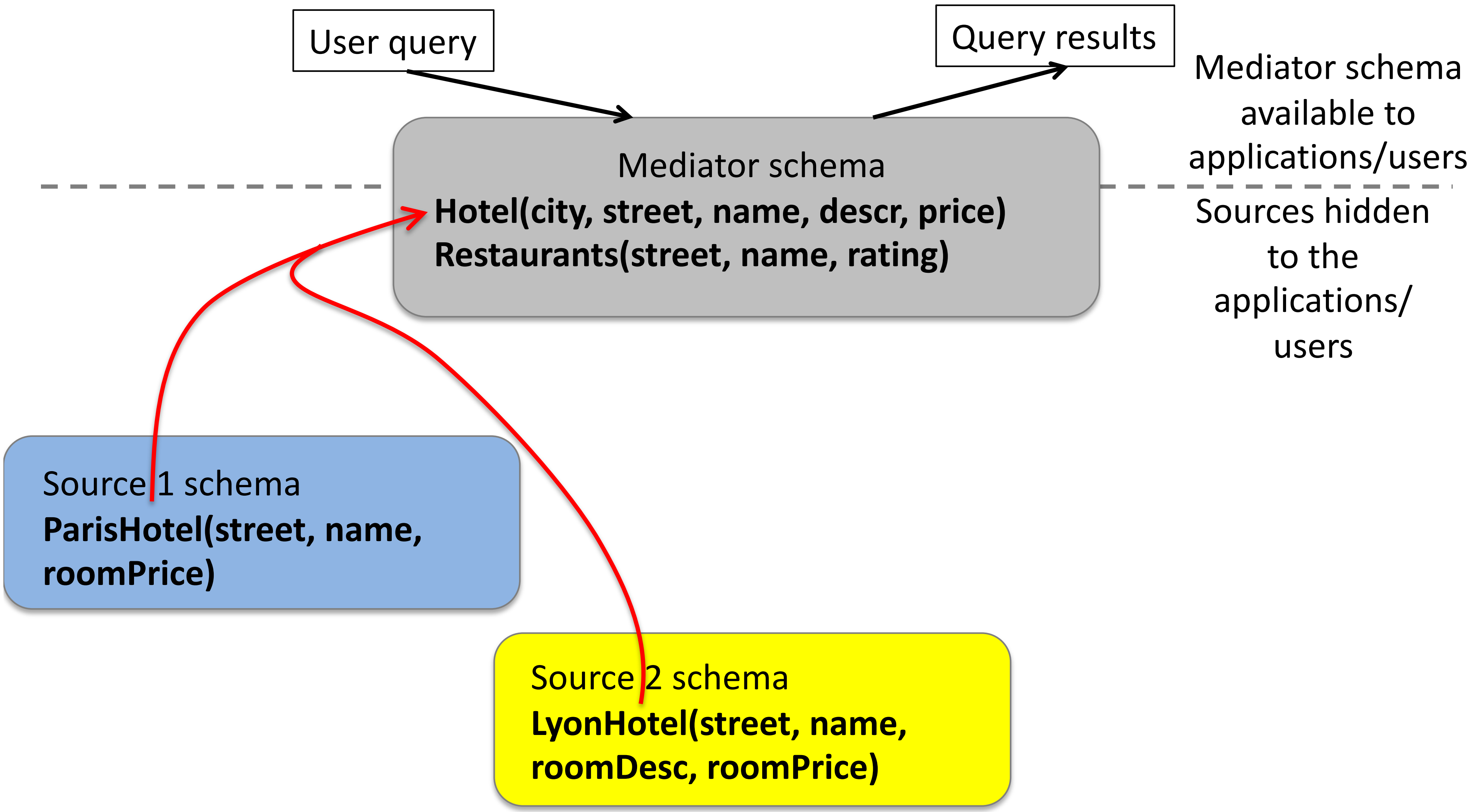
Concluding remarks on global-as-view (GAV)

- Query processing = **view unfolding**: replacing the view name with its definition
  - Just like queries over views in a centralized database
  - Heuristic: push as many operators (select, project, join; navigate…) on the sources as possible
- **Weakness**: changes in the data sources require changes of the global schema
  - In the worst case, all applications written based on this global schema need to be updated
  - Hard to maintain

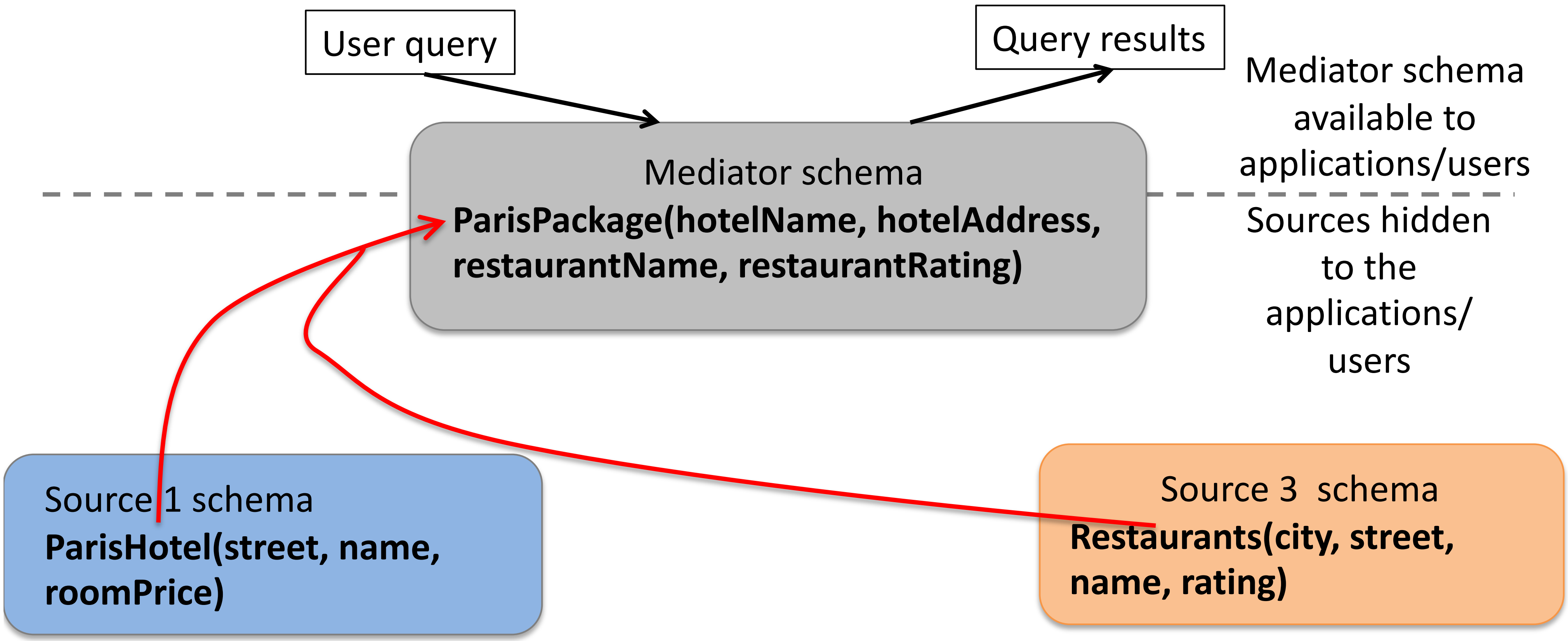
Global-as-View: Adding a new source



Global-as-View: Removing a source (1)



Global-as-View: Removing a source (2)



If **Source3.Restaurant** withdraws, the **ParisPackage** relation in the global schema becomes empty; applications cannot even access **Source1.ParisHotels**, even though they are still available.

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

s1:ParisHotel(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)

Defining **s1:ParisHotels** as a view over the global schema:

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

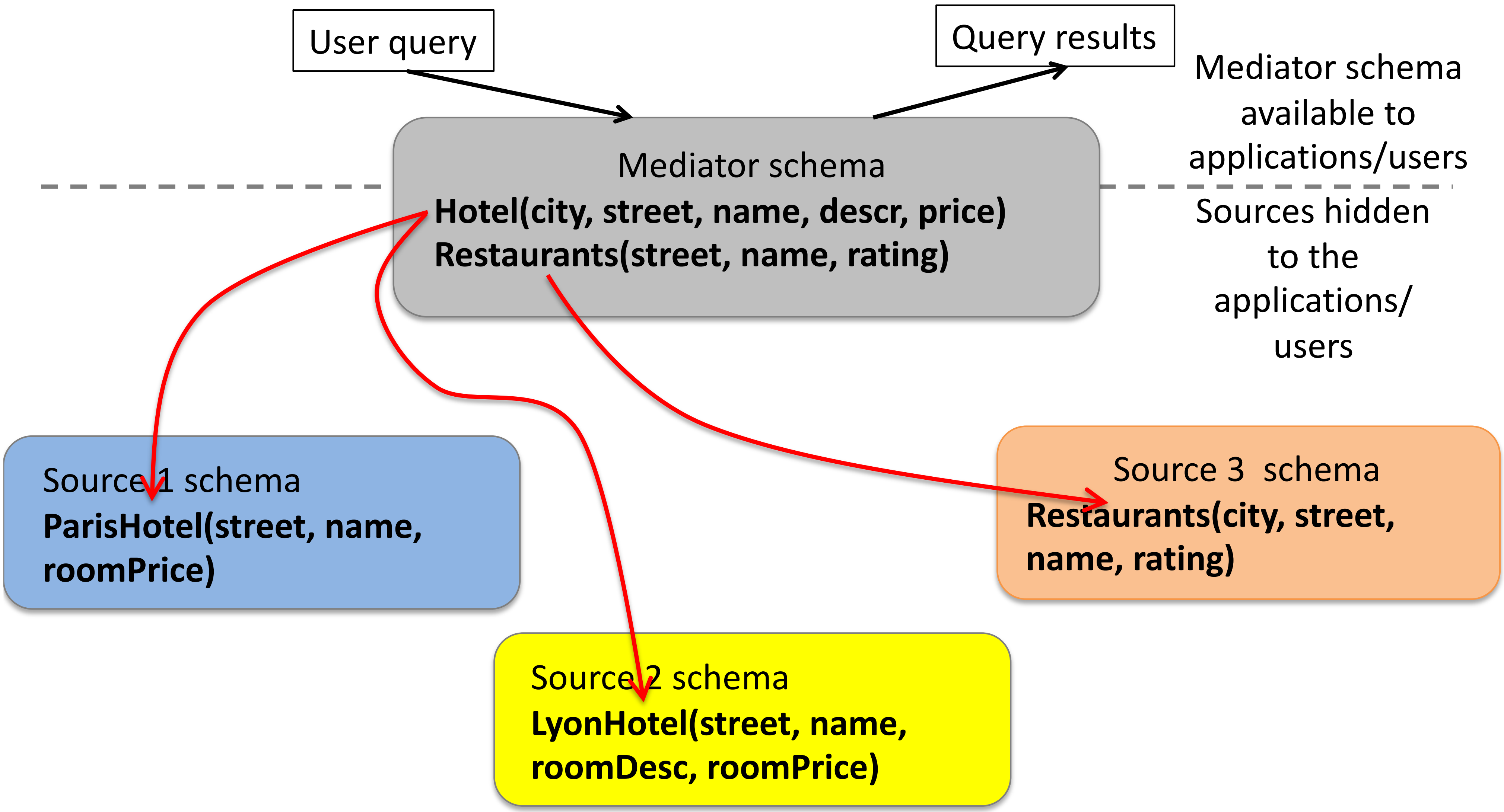
Defining **s2:LyonHotel** as a view over the global schema:

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

Defining **s3:Restaurant** as a view over the global schema:

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

Connecting the source schemas to the global schema: Local-as-View



GAV and LAV have different expressive power

- **Some GAV scenarios cannot be expressed in LAV**
- Example:

```
create view ParisPackage as
select ph.name as hotelName, ph.street as hotelAddress,
       r.name as restaurantName, r.rating as restaurantRating
from s1:ParisHotel ph, s3:Restaurants r
where r.city='Paris' and r.street=ph.street
```

- The view only contains (hotel, restaurant) pairs that are *on the same street* in Paris
- Not possible to express this with LAV mappings
  - LAV describes each source *individually* w.r.t. the global schema
  - Not in correlation with data available in *other sources* !

GAV and LAV have different expressive power

- There exist **LAV scenarios that cannot be expressed in GAV**
- Example: s3:MHotels(city, street, name, price) only has data about Marseille hotels, s4:WHotels(city, street, name price) has only data about Wien hotels
  - Assume Hotels is defined as:

    select * from Mhotels union all select * from WHotels
  - A query about hotels in Rome will also be sent to s3 and s4, although it will bring no results
  - LAV query processing avoids this (see next)

GAV and LAV have different expressive power

- There exist **GAV scenarios that cannot be expressed in LAV**
- Example:

  create view **ParisPackage** as
  select ph.name as hotelName, ph.street as hotelAddress, r.name as restaurantName, r.rating as restaurantRating
  from s1:ParisHotel ph, s3:Restaurants r
  where r.city='Paris' and r.street=ph.street

- The closest we can do is define s1.ParisHotel and s3.Restaurants *each* as a projection over ParisPackage
- But this changes the semantics of ParisPackage:
  - It does not express that *only Paris restaurants* are in ParisPackage
  - Not possible to express that only (hotel, restaurants) *on the same street* are available through the integration system
  - ParisPackage becomes the cartesian product of ParisHotel with all restaurants...

Query processing in Local-as-View (LAV) Simple scenario 1

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

Query:
select street, name from Hotel

No *equivalent* rewriting exists.

- We should not use a rewriting that has some wrong answers (not contained in those of the query)!
- We should aim for *contained rewritings*
- Better: *Maximally contained rewritings, such as:*

  Select street, name from s1:ParisHotels

Query processing in Local-as-View (LAV) Simple scenario 2

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

define view **s7:CheapHotels** as
select street, name
from Hotel where price<80

Query:
select street, name from Hotel

S1:ParisHotels has some useful answers. So does s7:CheapHotels.
They may overlap (or not); no way of knowing.

Maximally contained rewriting:

select street, name from s1:ParisHotels union
select street, name from s7:CheapHotels

Query processing in Local-as-View (LAV) Simple scenario 3

define view **s7:CheapHotels** as
select street, name
from Hotel where price<80

Query:
select street, name from Hotel where city='Paris'

Maximally contained rewriting:

None exists! ❌

Query processing in Local-as-View (LAV)
Simple scenario 4

define view **s7:CheapHotels** as
select street, name, city
from Hotel where price<80

Query:
select street, name from Hotel where city='Paris'

Maximally contained rewriting:

select street, name from s7:CheapHotels where city='Paris'

Query processing in Local-as-View (LAV) Simple scenario 5

define view **s7:CheapHotels** as
select street, name
from Hotel where price<80

Query:
select street, name, price from Hotel where city='Paris'

Maximally contained rewriting:

None exists! ❌

Query processing in Local-as-View (LAV) Observation

A view may fail to be usable to answer a query if the view:

- The query applies a restriction the view does not apply and the rewriting cannot apply it either
- And/or, the view fails to project (store) an attribute that the query needs

define view **s7:CheapHotels** as
select street, name
from Hotel where price<80
Query:
select street, name, price
from Hotel where city='Paris'

define view **s7:CheapHotels** as
select street, name
from Hotel where price<80

Query:
select street, name
from Hotel where city='Paris'

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'

Step 1: identify potentially useful views

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)

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 (one view per table):

s1:ParisHotels and s3:Restaurant

s2:LyonHotels and s3:Restaurant

Step 3: for each view combination and each view, check:

- If the view returns the attributes we need:
  - Those 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

If one condition is not met, discard the view combination.

define view s**1:ParisHotels** as
select street, name, price as roomPrice
from Hotel where city='Paris'

The query needs:

- street, price, rating (returned): the view provides them
- city and street for the join: street is provided, city is not (but it is a constant, thus known)

The view has a selection on the city which the query does not have → The view provides *part* of the data needed by the query. The view selection is compatible with the query.

The view s1:ParisHotels is OK.

define view s3:Restaurant as select * from Restaurant

The view s3:Restaurants is OK.

The view combination s1:ParisHotels, s3:Restaurants is OK provided that Restaurant.city is set to Paris.

Query processing in Local-as-View (LAV)

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 (one view per query table):

s1:ParisHotels and s3:Restaurant

s2:LyonHotels and s3:Restaurant

Step 3: for each view combination and each view, check:

[...]

If one condition is not met, discard the view combination.

Step 4: for each view combination, add the necessary joins among the views, possibly selections and projections → rewriting

Query rewriting using <u>s1:ParisHotels and s3:Restaurant</u>:

select h.street, h.price, r.rating
from s1:ParisHotels h and s3:Restaurant r
where r.city='Paris' and h.street=r.street

This is a *partial* rewriting, and so is:

Query rewriting using <u>s2:LyonHotel and s3:Restaurant</u>:

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

Query processing in Local-as-View (LAV)

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 (one view per query table):

s1:ParisHotels and s3:Restaurant

s2:LyonHotels and s3:Restaurant

Step 3: for each view combination and each view, check:

[...]

If one condition is not met, discard the view combination.

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

Full query rewriting:

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

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)

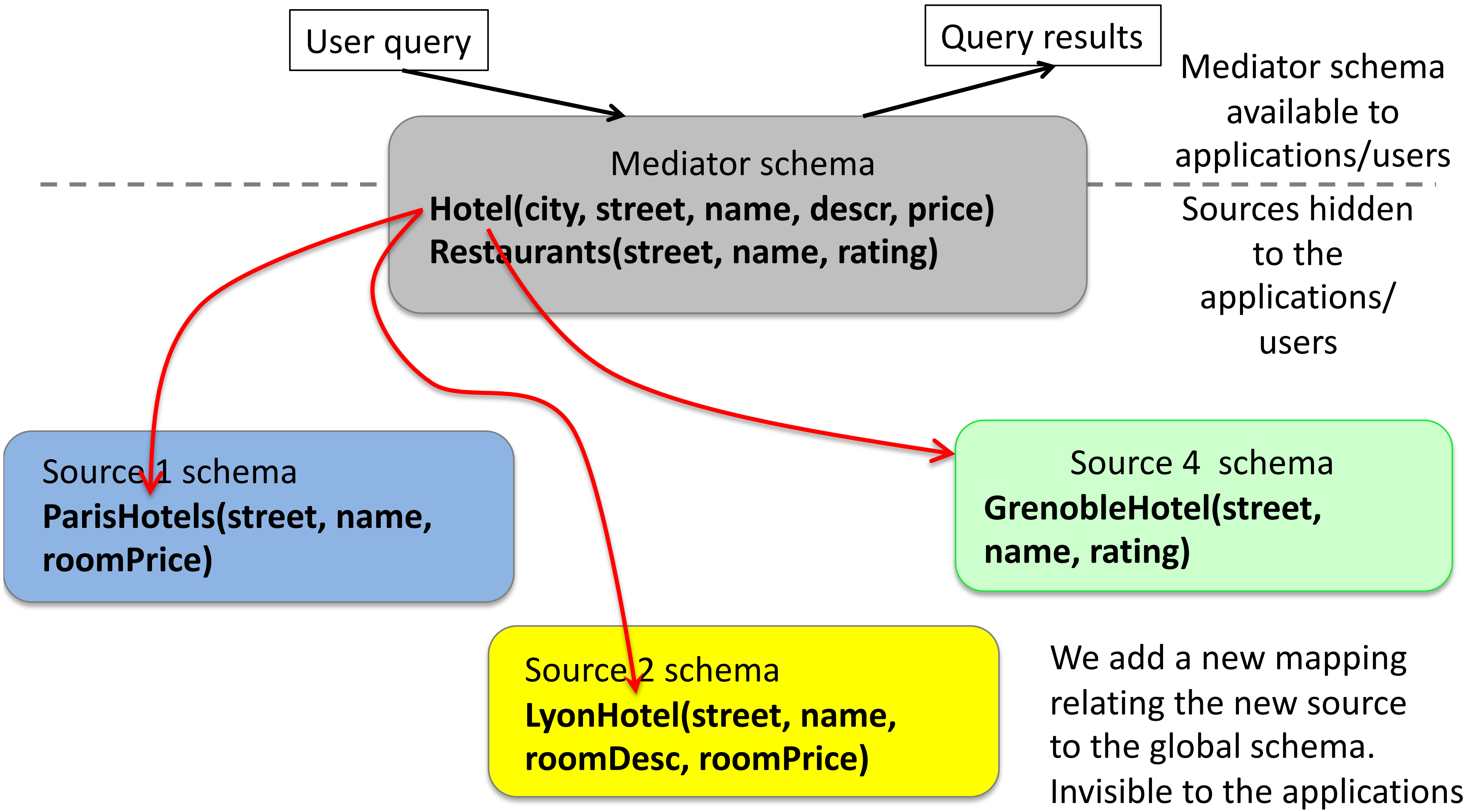
Query processing

- 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

The schema definition is **more robust**:

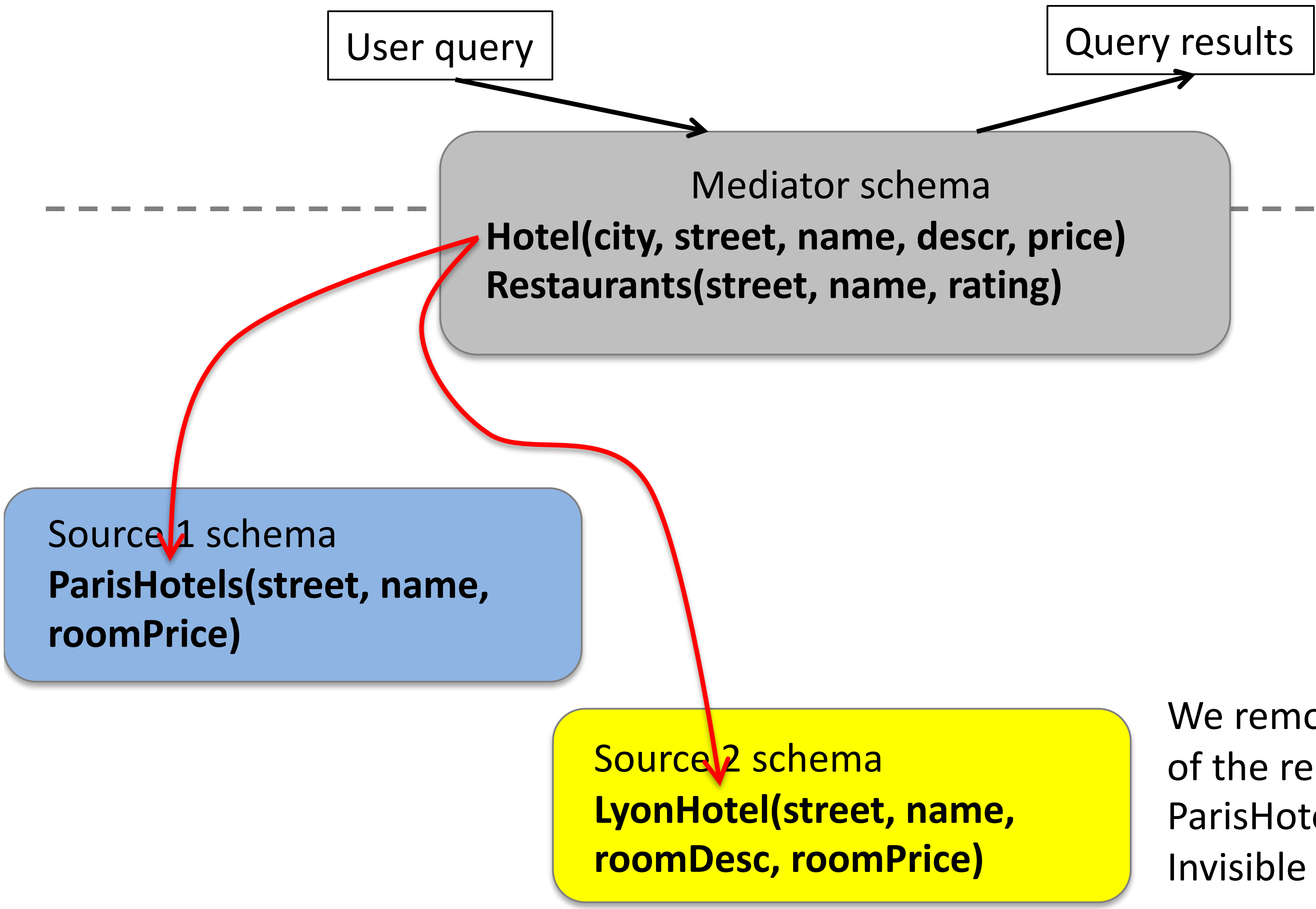
- One can independently add/remove sources from the system without the global schema being affected at all (see next)
- Thus, no application needs to be aware of the changes in the schema

Local-as-View: adding a new source



We add a new mapping relating the new source to the global schema.
Invisible to the applications!

Local-as-View: Removing a source



Mediator schema available to applications/users

Sources hidden to the applications/ users

We remove the mapping of the removed source. ParisHotels still available. Invisible to the applications!

Connecting the source schemas to the global schema: Global-Local-as-View (GLAV)

Generalizes both GAV and LAV

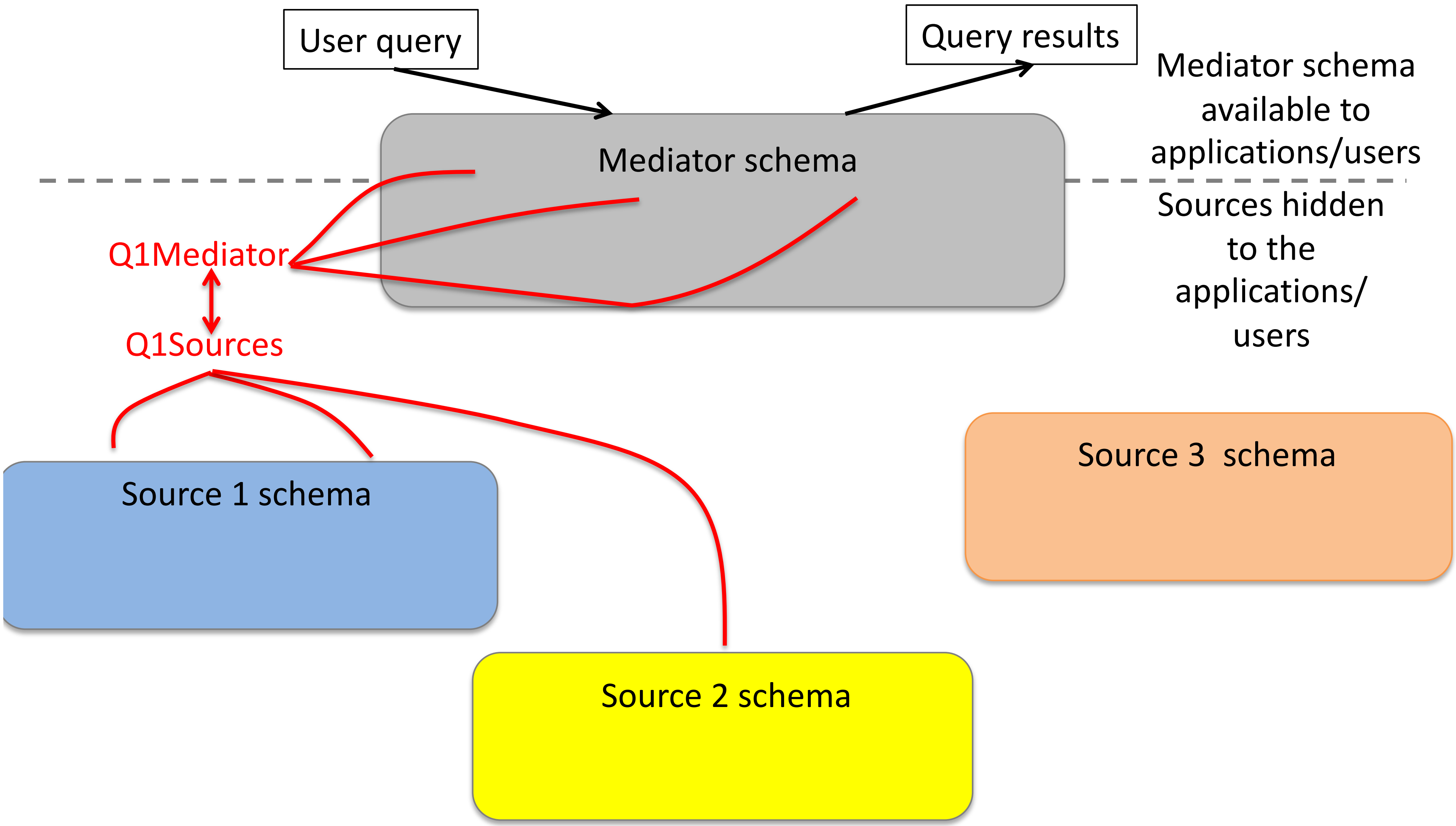
1 mapping = 1 pair (query over 1 or several sources schemas,
query over the mediator schema)

Q1Mediator(m:r1, m:r2, m:r3, ...) ←→ Q1Sources(s1:t1, s2:t1, ...)
Q2Mediator(m:r1, m:r2, m:r3, ...) ←→ Q2Sources(s1:t1, s2:t1, ...)
Q2Mediator(m:r1, m:r2, m:r3, ...) ←→ Q3Sources(s1:t1, s2:t1, ...)

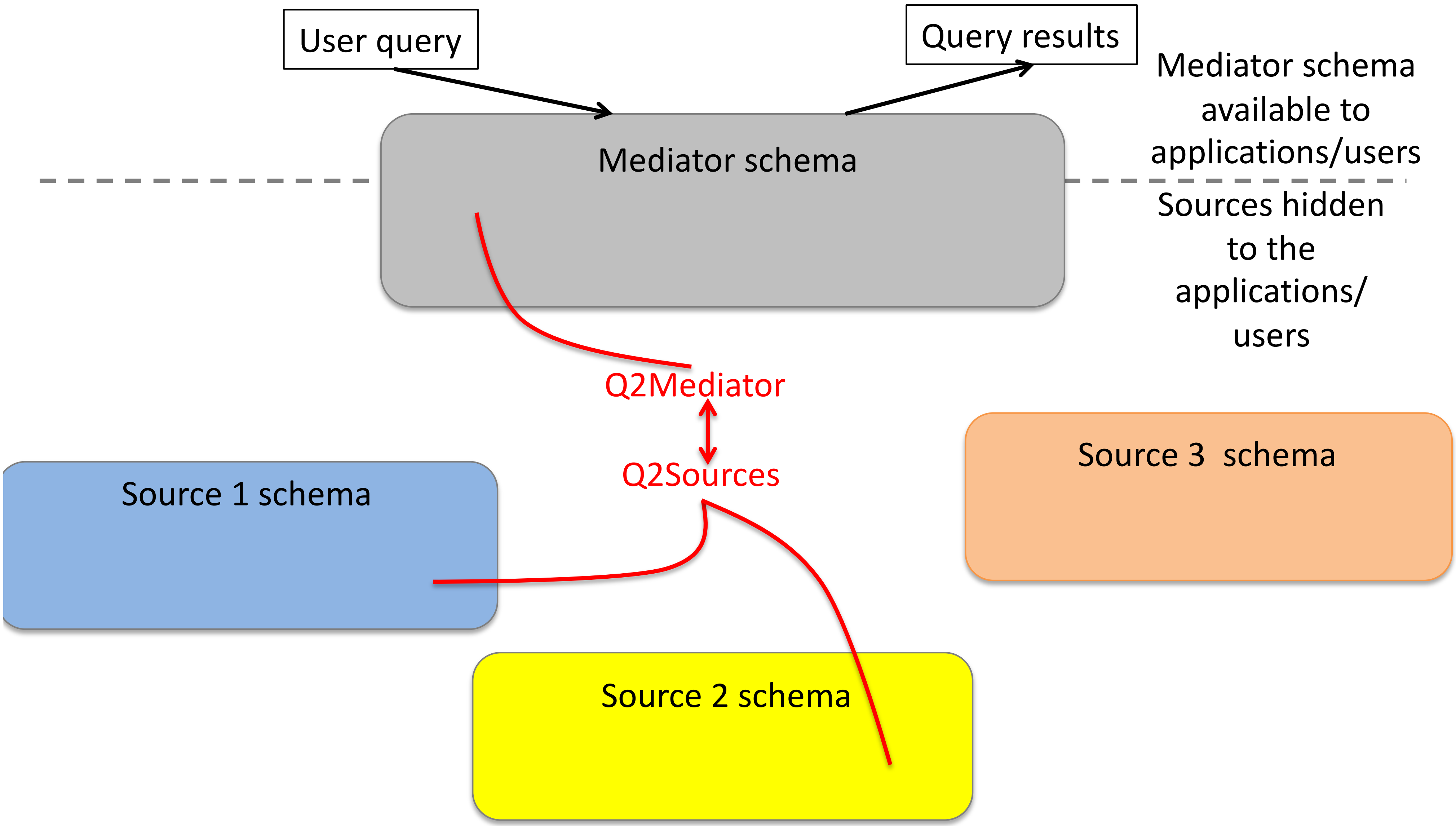
Semantics: *there is a tuple in QiMediator(...) for each result of QiSources(...)*

- A GAV mapping is a particular case of GLAV mapping where QMediator is exactly one mediator relation
- A LAV mapping is a particular case of GLAV mapping where QSources is exactly one source relation

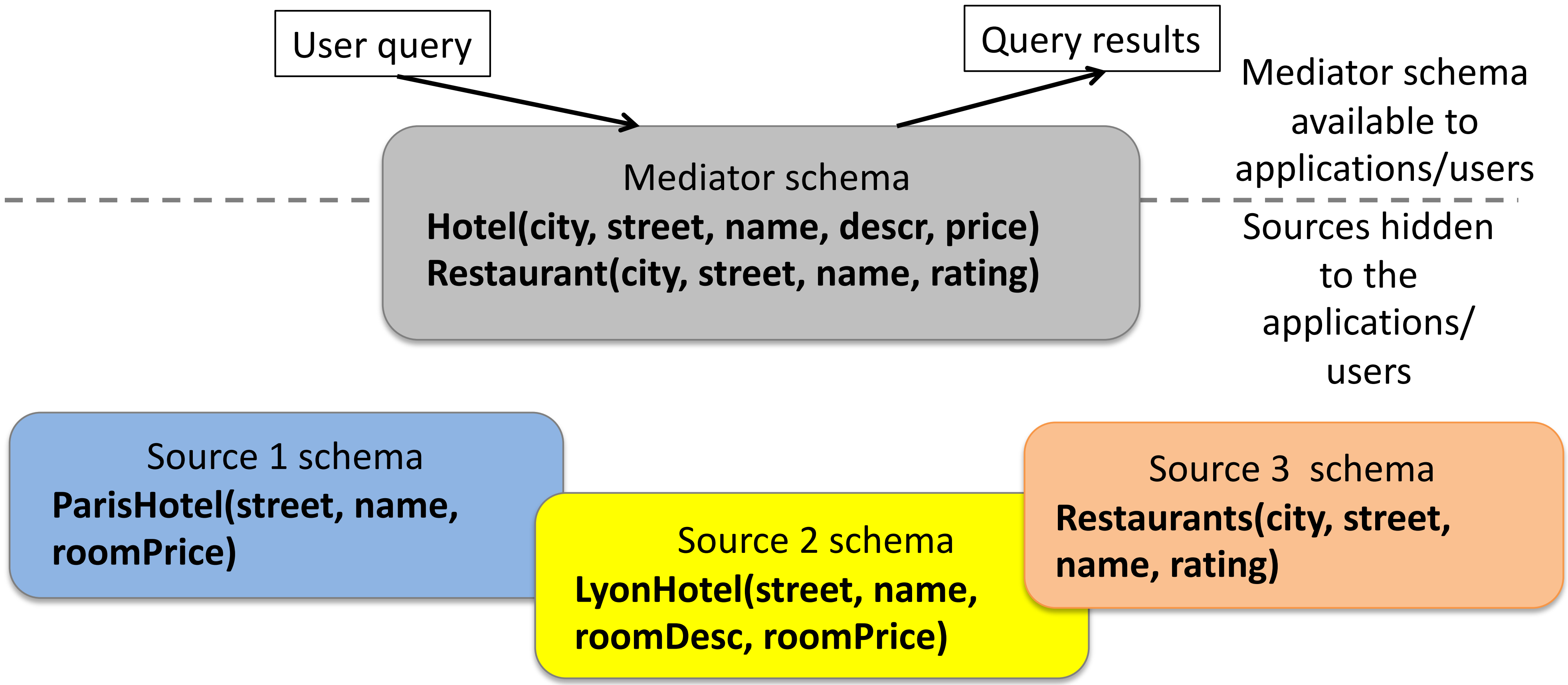
Connecting the source schemas to the global schema: Global-Local-as-View (GLAV)



Connecting the source schemas to the global schema: Global-Local-as-View (GLAV)



Global-Local-as-View: example

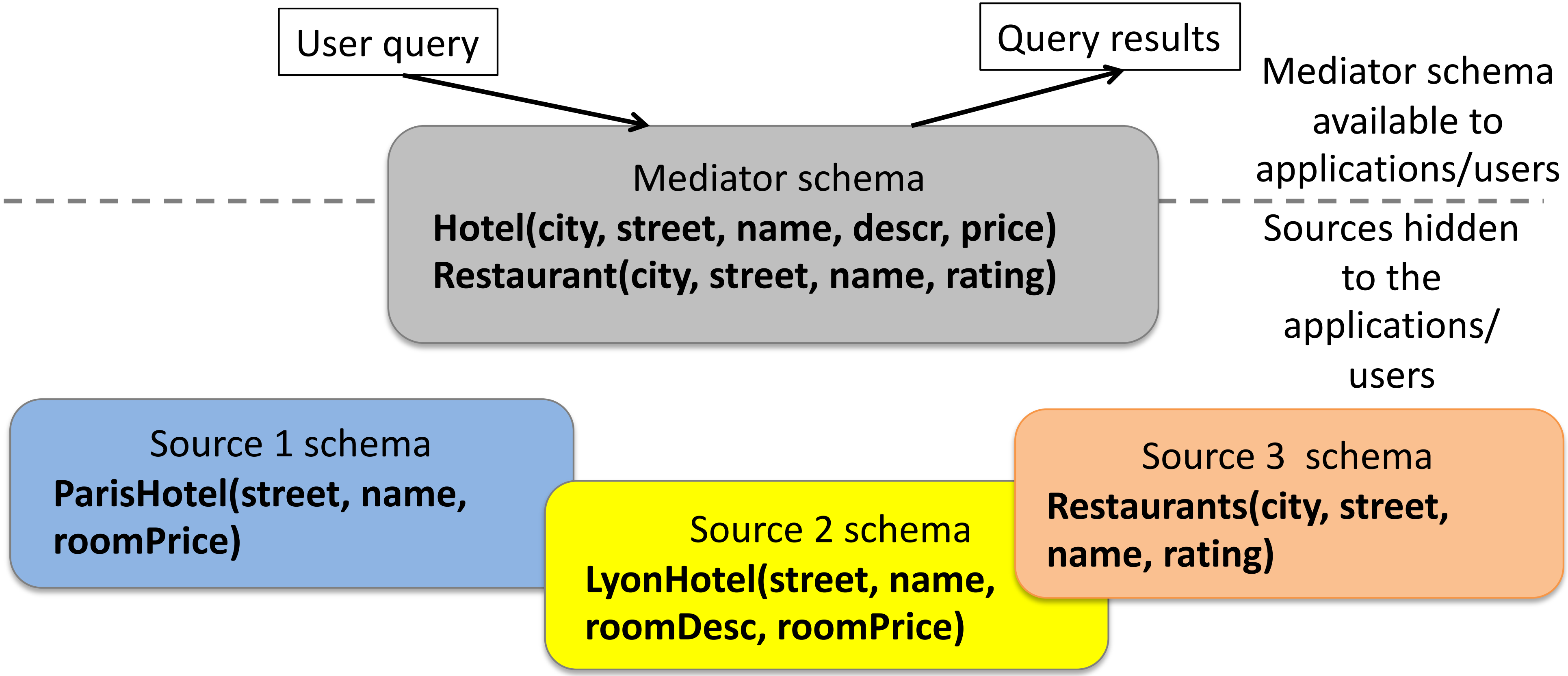


Previous LAV mapping of Source 1:

Q1Mediator: select street, name, price as roomPrice from Hotel where city='Paris'

Q1Sources: select * from ParisHotel

Global-Local-as-View: example

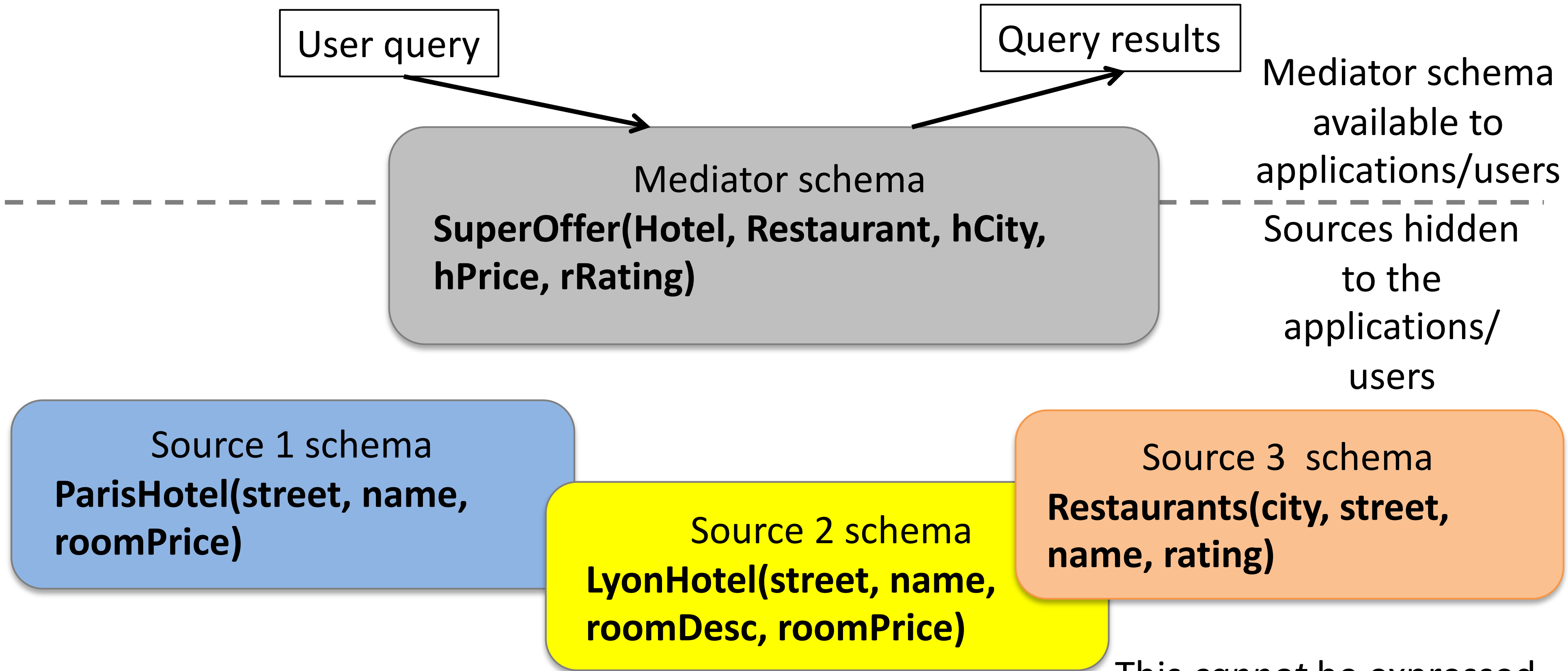


Previous GAV mapping of Hotel:

Q2Mediator: select * from Hotel

Q2Sources: select 'Paris' as city, street, name, null as descr, roomPrice as price from ParisHotel
union
select 'Lyon' as city, street, name, roomDesc as descr, roomPrice as price from LyonHotel

Global-Local-as-View: example



This *cannot* be expressed either in LAV or GAV.

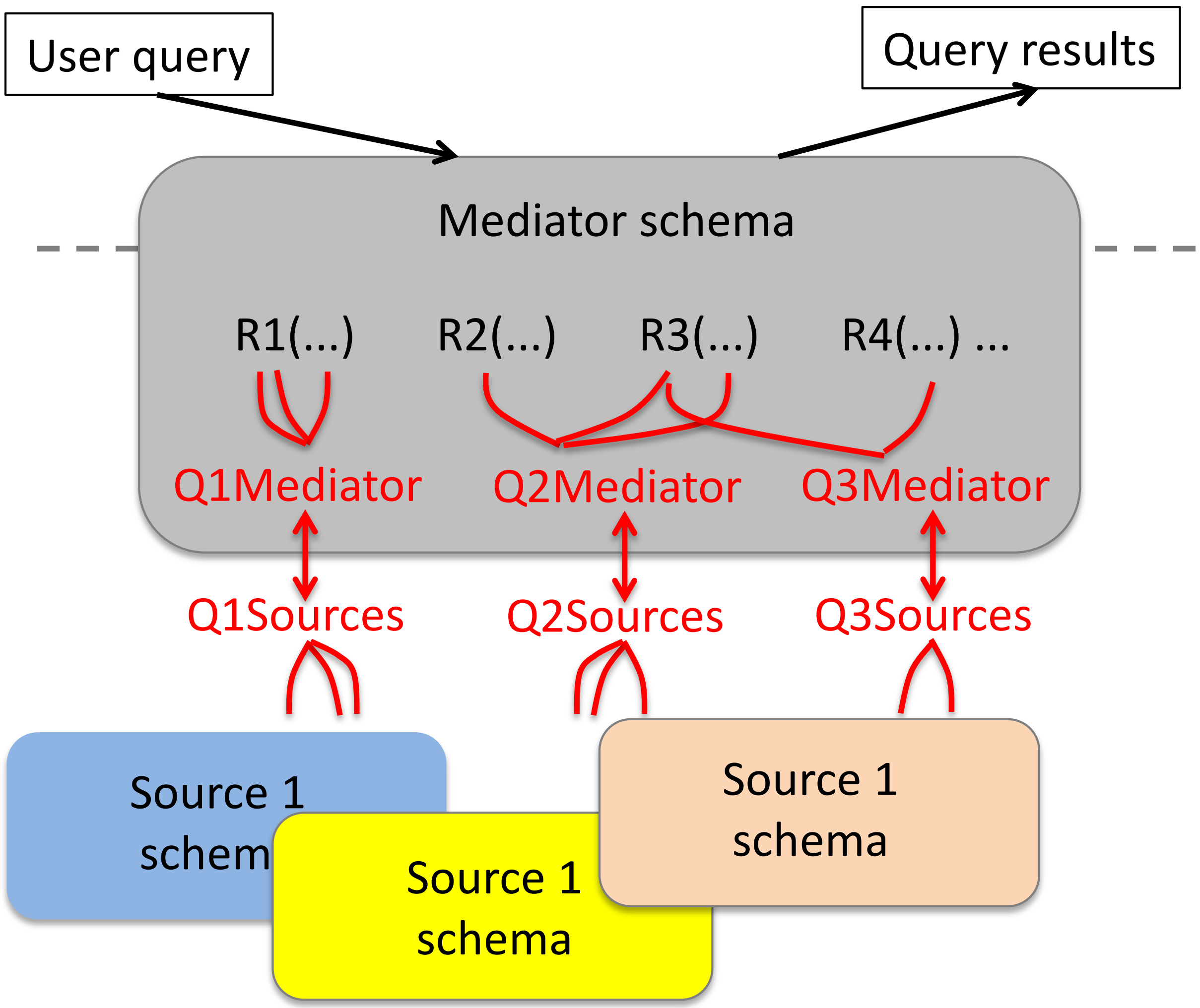
New GLAV mapping:

Q3Mediator: select * from SuperOffer where hCity='Lyon'

Q3Sources: select lh.name, r.name, h.roomPrice * 0.5 as hPrice, r.rating as rRating
from LyonHotel lh, Restaurants r
where r.city='Lyon' and name='Lion d'Or' and r.street=lh.street

This mapping says: "each result of Q3Sources leads to a SuperOffer in Lyon".
Other mappings could define more SuperOffers in Lyon, or in other cities, or with rRating=3...

Query Processing in GLAV



User queries asked on the mediator schema.

Q1Mediator, Q2Mediator, ... are queries over this schema

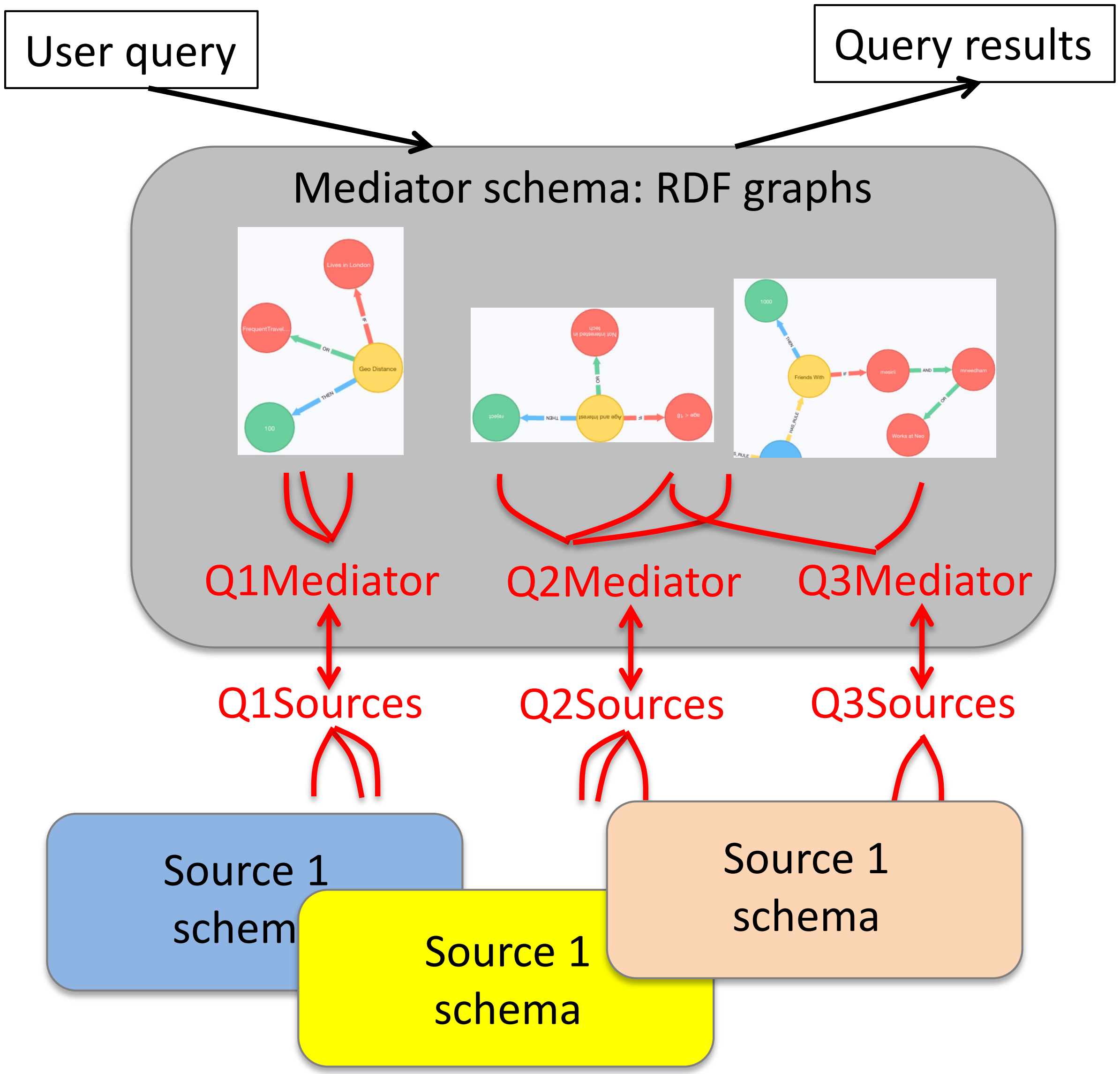
1. Apply **LAV**-style rewriting considering each QiMediator as a view over the mediator schema.
   – This leads to rewritings of Q over QiMediator relations (Q1Mediator, Q2Mediator, ...)
2. For each such rewriting, in **GAV** style, replace the symbol QiMediator by the query QiSources.
   – Then unfold → query over the sources themselves.

Examples: find all super offers in Paris? in Lyon?

Concluding remarks on GLAV

- The most flexible approach
  - Can express LAV, GAV, and more
- If a source changes or sources are added, as long as Q1Sources can be rewritten, applications will not be impacted
  - Only the "invisible" part of the system (the mappings) may have to be adapted
- Query rewriting remains expensive because it includes view-based query rewriting (NP-hard) as well as query unfolding (simple)

Modern mediators: GLAV with RDF global schema



Idea 1: RDF global schema

- Flexible!
- We can use ontologies to add semantics

Idea 2: write GLAV mappings, e.g.:

1. **Q1Sources**: an SQL query returning (x, y, z) tuples
   Q1Mediator:
   (x, 'friend', y), (y, 'worksfor' z)
   Q1Mediator "*creates RDF out of relational data*"
2. **Q2Sources**: a JSON query returning (z) nodes
   Q2Mediator:
   (z, 'type', Company)

If common z value, the graphs built by Q1,2Mediator **connect**!

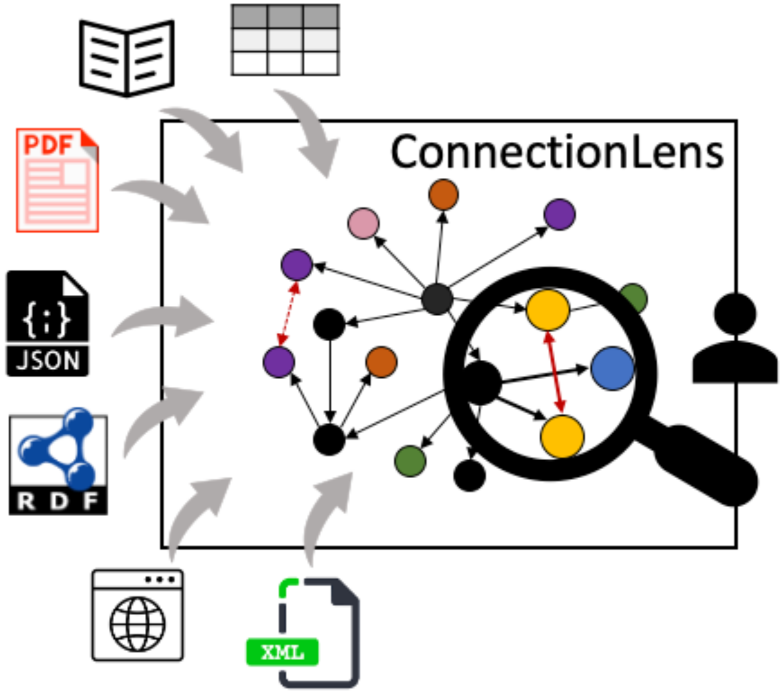
OTHER FORMS OF HETEROGENEOUS DATA MANAGEMENT:
DATA SPACE, DATA LAKE, DATA MESH

Data spaces

- "Data spaces" (Franklin, Halevy, Maier, 2005):
  - Many heterogeneous data sources...
  - On a single or on multiple machines
  - But, unlike data integration systems, the sources
    - May not be **structured**: text, email, Web pages, directories...
      - Therefore, different data models, or unstructured (text)
    - May not reside in **databases**
      - Therefore, limited query language
- Too many sources, too heterogeneous → integrated schema hard or impossible to define→ no integrated schema!

Data spaces

- How to query the data space?
  Use keywords!
- User query: kw1, kw2, …, kwm
- Answers:
  - From a text file: **minimal text fragments** that contain all kwds
  - From a database:
    - **One tuple** it if contains all the kwds, or
    - **A few tuples** if they join and they contain all the kwds, or
    - A **minimal JSON tree** that contains all the kwds, etc.
  - *Score* to decide which answers to return first



Data lakes

- Popular term, started around 2010 (cca)
- Mostly in **companies**
- Many data sources: hundreds, thousands
  - Mostly *relational* data. Also: text, JSON
  - Independently produced by different authors
    - Different schemas; different names for same or similar attributes
  - Some relationships *probably* exist between the schemas of the different databases
  - ... but finding and expressing them is beyond human capacity at this scale
- Seen as a generalization of data warehouse
  - Common: very large scale, « contains all the data »
  - Difference: no single schema in data lakes

How to exploit a data lake?

Algorithms to automatically find **relationships between datasets**:

- Have a *common attribute* → may join
- Have *same-type attributes* → may be unioned
- Share a *subset of attributes* → the projections may be unioned, or the two sets may be joined
- Have common data items → dataset similarity, used for recommendation, etc.

How to exploit a data lake?

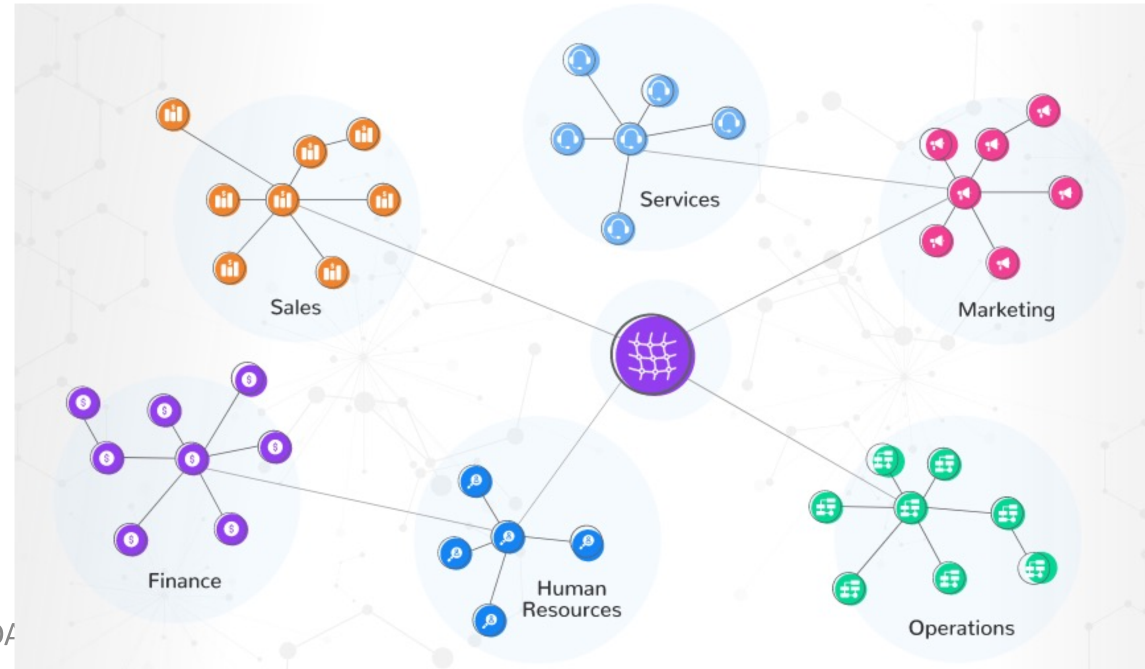
- **Query processing** over data sources whose relationships are well understood as in mediator or data warehouse

- Results of processing are typically stored again in the data lake, together with **provenance metadata**
  - **Metadata**: generic term referring to description of the data, e.g., what is it about, where does it come from, who produced it, when…
  - **Provenance**: a machine-understandable description of how a dataset was produced (which sequence of computations, on which inputs)
    - Often represented as a directed, rooted graph, with data as nodes and processing as edges

Data lake products: IBM, Oracle, Amazon

Data mesh

- Since 2019
- Adopted by Netflix, Paypal, Amazon…
- Four core principles:
  - **Domain ownership**: domain (application) specialists decide what data to store, how it should be structured, described, etc.



Data mesh

- Four core principles:
  - **Domain ownership**: domain (application) specialists decide what data to store, how it should be structured, described, etc. E.g., personnel, financial, marketing…
  - **Data as a product**: each dataset, original or derived, should be: discoverable, addressable, trustworthy, have self-describing semantics and syntax
  - **Self-serve data platform**: easy for domain teams to add/modify/work on data
  - **Federated computational governance** across the domain teams + technical infrastructure