Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

Technische Universität Kaiserslautern

Chapter 13
Virtual Data Integration

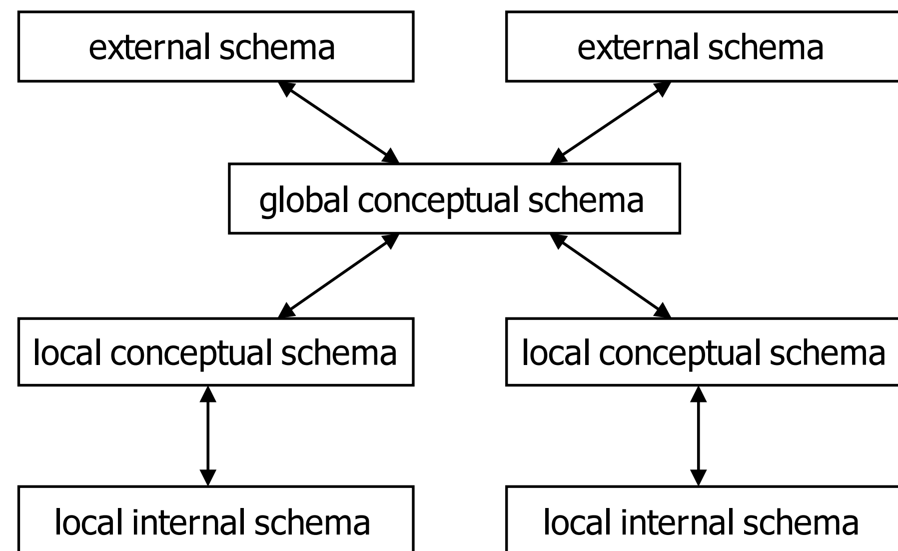
Outline

- Accessing multiple, distributed data sources in an integrated manner
  - architectures
  - types of transparency achieved
- Wrapper architecture as an infrastructure for overcoming heterogeneity
  - wrapper tasks
  - Garlic
  - SQL/MED
- Multi-database languages
  - SchemaSQL
  - FIRA/FISQL

© *Prof.Dr.-Ing. Stefan Deßloch*

Distributed DBMS

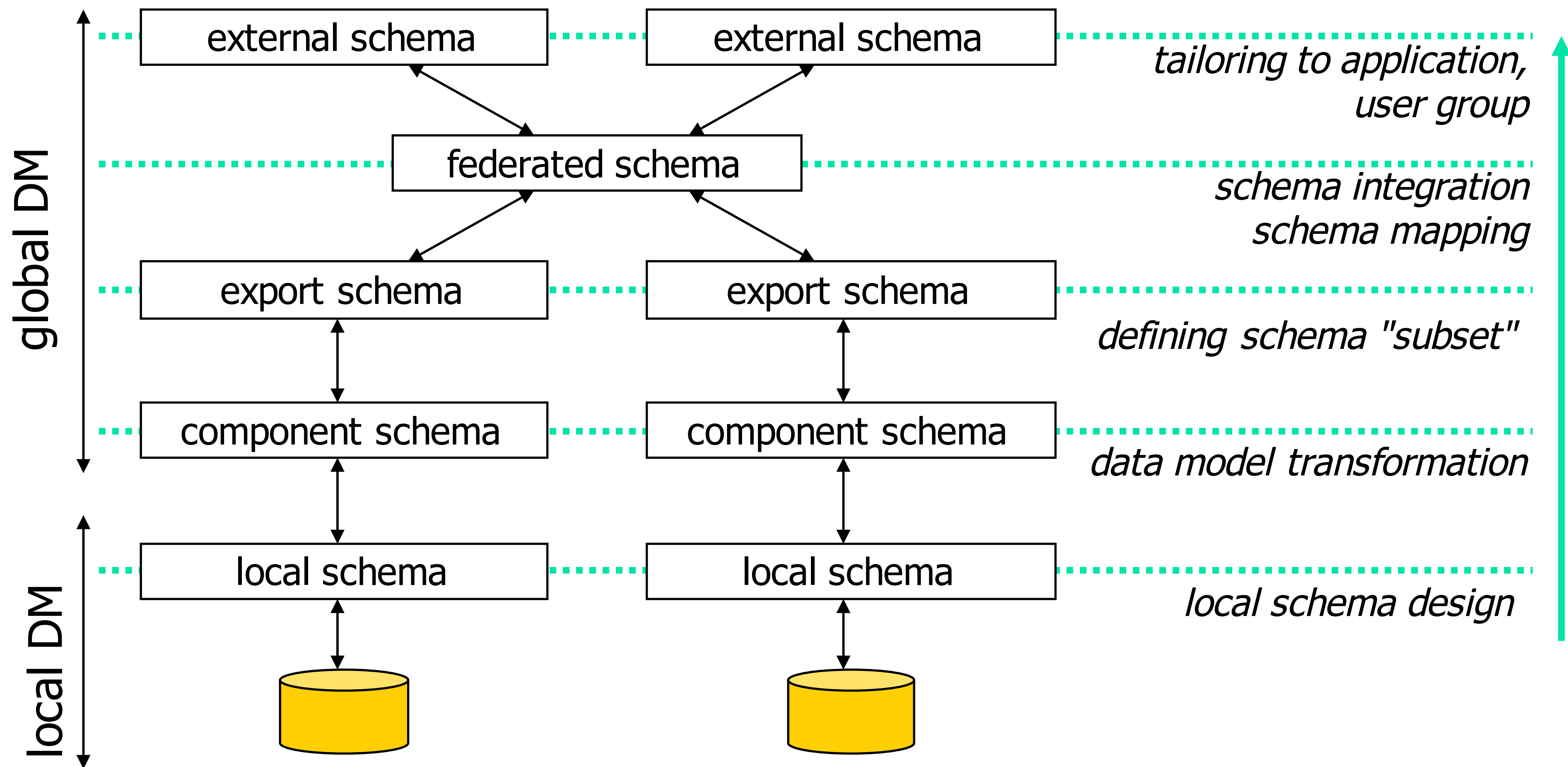
- Data is distributed across multiple systems
  - goal: distribution transparency for applications
- Distribution "by design"
  - distribution is intended, planned
  - participating systems give up autonomy
- 4-layer schema architecture
- Distribution strategies
  - horizontal partitioning
  - vertical partitioning
  - (partial) replication
    - to improve performance
- Tightly coupled
- Heterogeneity is not an issue



Federated DBMS

- Based on a global, federated conceptual schema
  - describes integrated view of all participating data sources using the canonical DM
    - global data model (and query language)
  - realizes distribution transparency
  - application can access multiple data sources within the same query
- Distribution is "given", resulting heterogeneity has to be dealt with
  - alternatives for federated schema creation
    - bottom-up: schema integration
    - top-down: schema design, schema mapping
- Preserves high degree of autonomy of participating data sources

Schema Reference Architecture

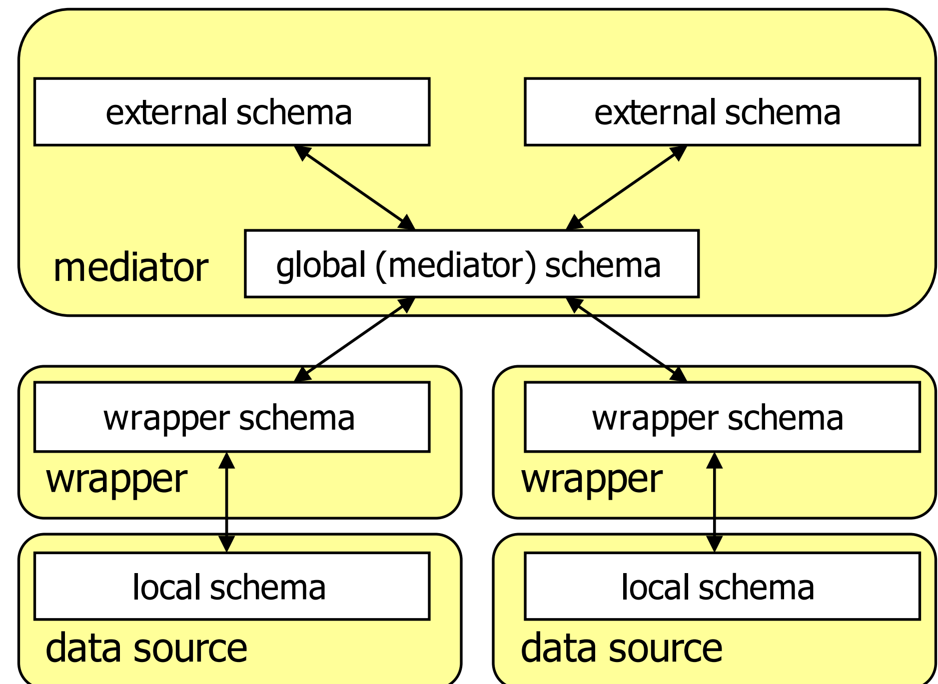


More Architecture Components

- Local Schema
  - corresponds to the conceptual schema of the local DB
  - based on local data model (e.g., relational DM)
- Component Schema
  - describes the local DB using global (canonical) data model
  - overcome data model heterogeneity
- Export Schema
  - describes subset of local data/schema to be made available for global applications
  - may be under the control of the local system (component system)
    - may result in swapping the export and component schemas in the architecture
- Federated Schema (global schema)
  - includes (and possibly just renames) export schema elements
  - may provide an integrated schema
    - resolve structural, schematic, and semantic heterogeneity (e.g., using view mechanism)
- External Schema
  - corresponds to classic external schema, now for the federated system

Mediator-based Information Systems

- Generalization of previous architectures based on
  - wrappers for accessing data sources
  - mediators that access one or more wrappers and provide useful services
    - search/query
    - data transformation
    - providing meta-data
    - data-level integration
    - ...
- Data sources remain autonomous
- Global mediator schema to achieve distribution transparency
- Architecture variations
  - nesting of mediators
  - single wrapper for multiple sources (of the same type)
  - applications accessing wrappers directly



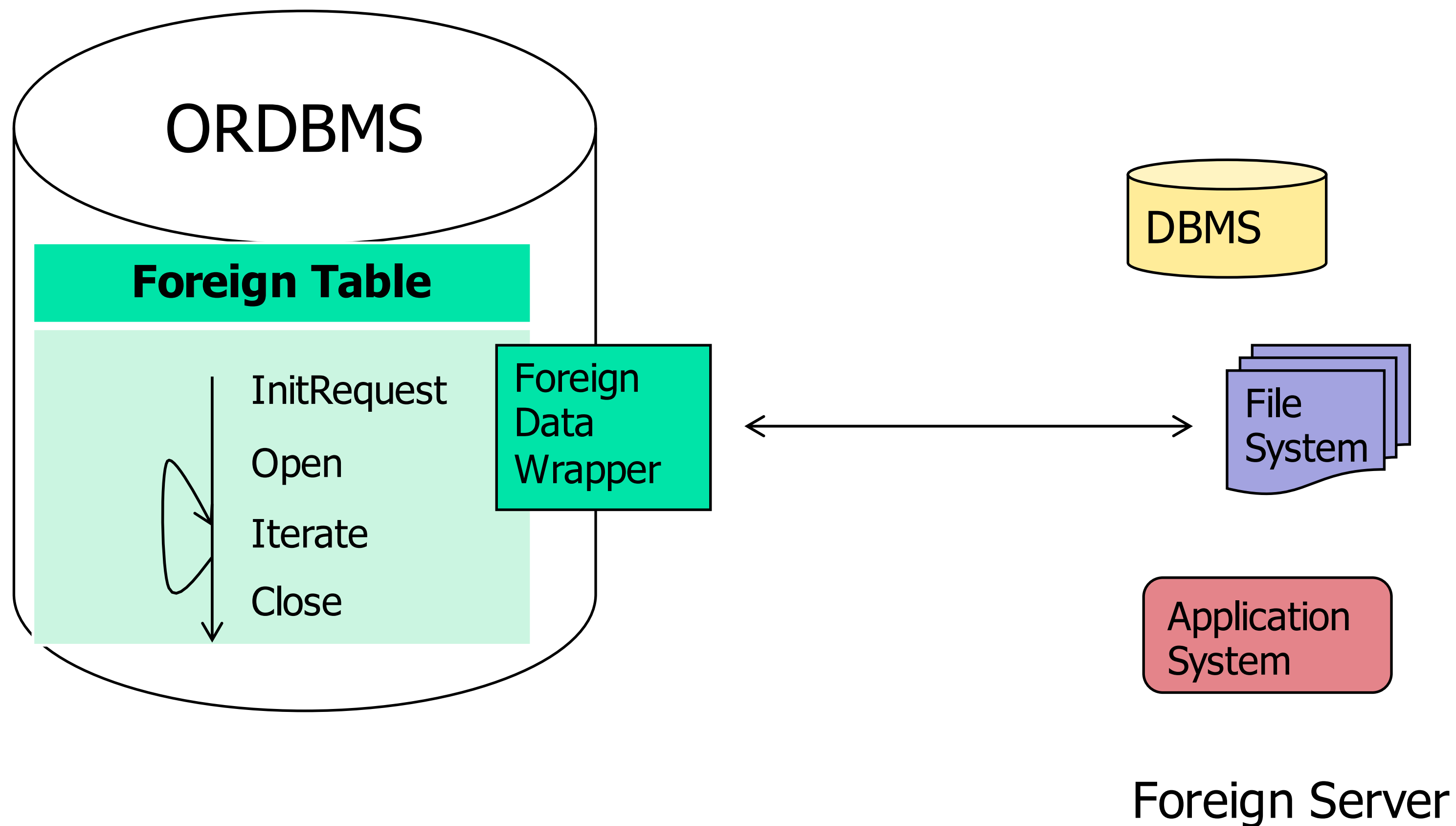
Wrapper Tasks

- Encapsulate a data source and provide uniform access to data
- Provide infrastructure for overcoming heterogeneity among data sources:
  - wrapper architecture
  - wrapper interfaces
- Help overcome heterogeneity of data sources regarding
  - data model
  - data access API
  - query language and capabilities
    - query language and expressiveness (simple scan, sort, simple predicates, complex predicates, aggregation, binary joins, n-way joins, …)
    - class/function libraries
    - proprietary query APIs
- Support global query evaluation and optimization
  - provide information about ability to process parts of a query
    - cost information

➨ Useful infrastructure for Federated DBMS

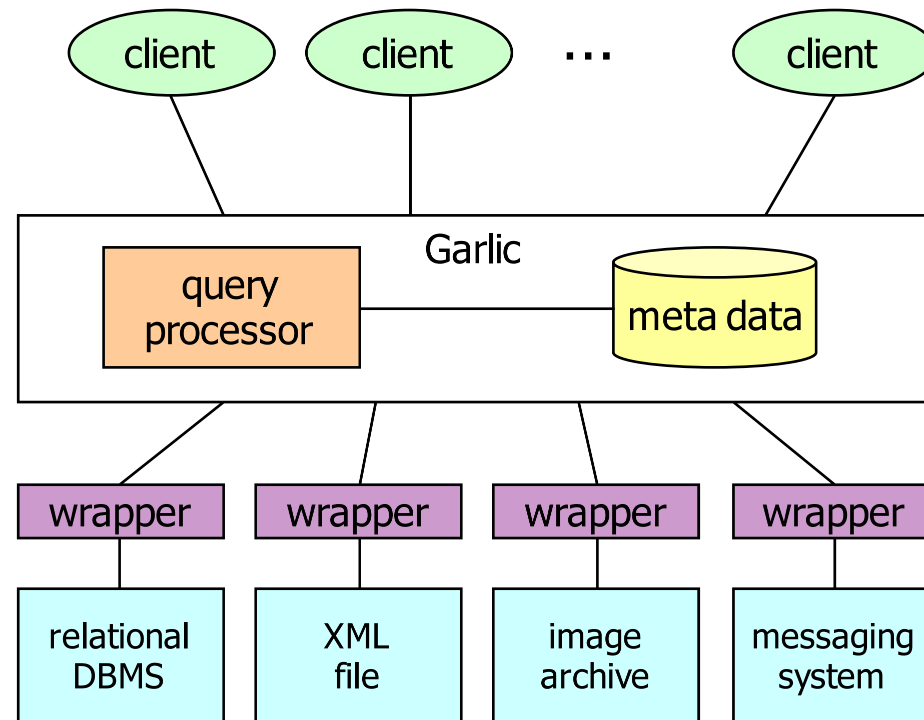
Example: SQL – MED

- *Foreign Data Wrapper* in 'SQL – Management of External Data (MED)'



Garlic

- "The wrapper architecture of Garlic ... addresses the challenge of diversity by standardizing how information in data sources is described and accessed, while taking an approach to query planning in which the wrapper and the middleware dynamically determine the wrapper's role in answering a query"



M. T. Roth, P. Schwarz: "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources", VLDB'97

Wrapper Architecture

- Garlic and wrappers cooperate for query processing
  - wrapper provides information about its processing capabilities
  - Garlic query engine compensates for (potential) lack of wrapper query functionality
    - function compensation
- Extensibility
  - add new data sources (accessed using existing wrappers)
  - add new wrappers for supporting new types of data sources
- Wrapper evolution
  - start with simple wrappers (equivalent of a table/collection scan)
    - low cost
  - expand query processing capabilities of the wrapper until it provides full support of the data source functionality

Wrapper-Services

Garlic

modelling data as object collections

method calls

query planning

query execution

hotel object

hotel object

wrapper plan

getCategory

request

iterator::advance()

Wrapper

wrapper plan

Data Source

name	category	rate	location	city

Modeling Data as Object Collections

- Registration
  - wrapper supplies description of data source in GDL (Garlic Data Language, derived from ODMG-ODL)
  - 'export/global schema' at the garlic level
- Garlic object
  - interface
  - at least one implementation (multiple are possible, but only one per data source)
  - identity: OID consists of
    - IID (implementation identifier)
    - key (identifies instance within a data source)
  - root objects (collections) serve as entry into data source, can be referenced using external names

Example: Travel Agency Schema

Relational Repository Schema:

```
interface Country {
    attribute string name;
    attribute string airlines_served;
    attribute boolean visa_required;
    attribute Image scene}

interface City {
    attribute string name;
    attribute long population;
    attribute boolean airport;
    attribute Country country;
    attribute Image scene}
```

Web Repository Schema:

```
interface Hotel {
    attribute readonly string name;
    attribute readonly short category;
    attribute readonly double daily_rate;
    attribute readonly string location;
    attribute readonly string city}
```

Image Server Repository Schema:

```
interface Image {
    attribute string file_name;
    double matches (in string file_name);
    void display (in string device_name)}
```

Method Calls

- Method can be called by Garlic query execution engine or by the application, based on an object reference
- Methods
  - implicitly defined get/set-methods (accessor methods)
  - explicitly defined methods
- Invocation mechanisms
  - stub dispatch
    - natural if data source provides object class libraries
    - example: *display* (see previous charts)
      wrapper provides routine that extracts file name from OID, receives device name as parameter, calls class library for display operation
  - generic dispatch
    - wrapper provides one entry point
    - schema-independent
    - example: relational wrapper (see previous charts)
      access methods only; each call is translated into a query:
      - method name -> attribute
      - IID -> relation name
      - value -> assignment value (SET)

Query Planning

- Garlic optimizer builds the query plan using a bottom-up approach
  1. create plan fragments for single collections
  2. plan the join processing for local joins (collections within the same data source)
  3. plan the join processing across multiple sources
  4. finalize the plan (additional processing within Garlic query engine)
- Fundamental Idea: wrappers participate in query planning process
- Query planning steps (in each phase)
  - Garlic optimizer identifies for each data source the largest possible query fragment that does not reference other data sources, sends it to the wrapper
  - wrapper returns one or more query plans that can be used to process the full query fragment or parts of the query fragment
    - providing all objects in a collection is minimal requirement
  - optimizer generates alternative plans, estimates execution costs, provides for compensation of fragments not supported by the wrapper

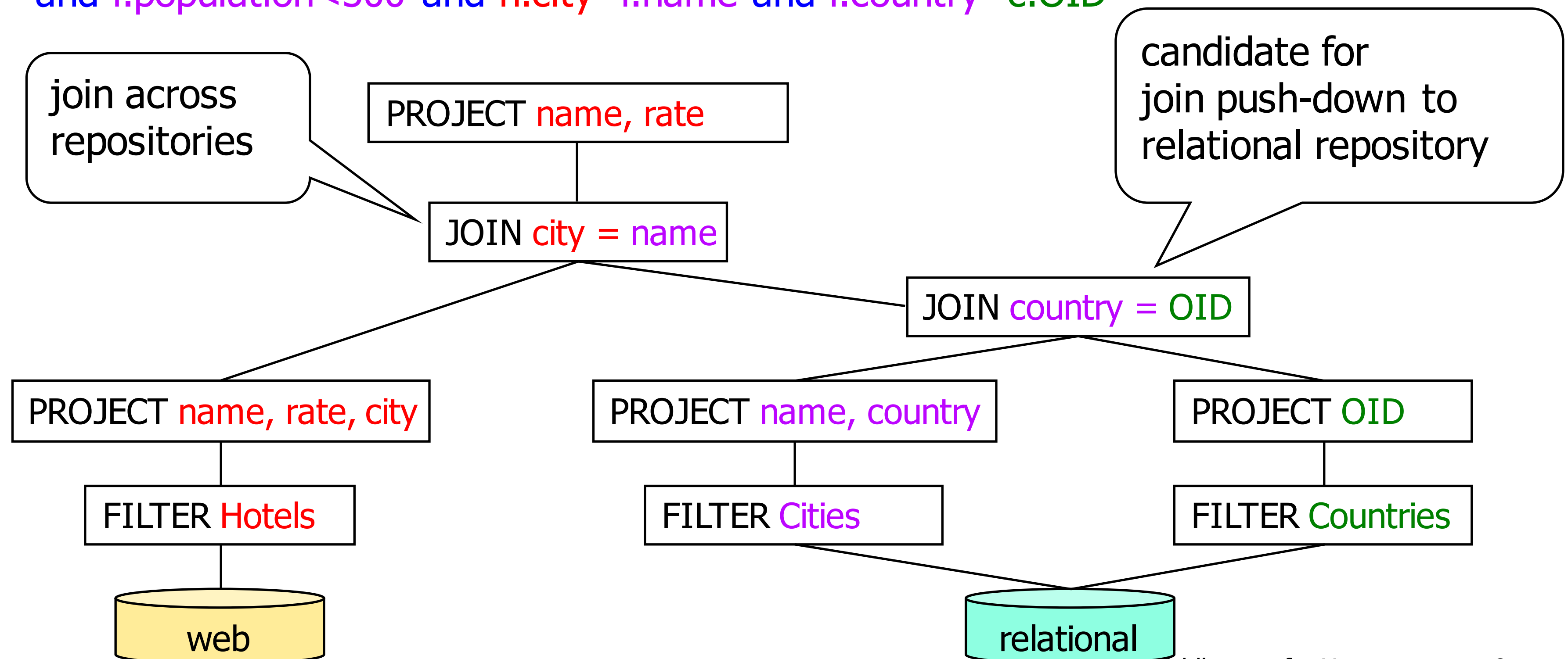
Query Planning (continued)

- Wrapper provides the following methods to be used by Garlic *work requests*:
  - *plan_access()*: generates *single-collection access plans*
  - *plan_join()*: generates *multi-way join plans* (joins may occur in application queries or in the context of resolving path expressions)
    - tables to be joined all reside in the same data source
  - *plan_bind()*: generates special plan, which can be used to process the *inner stream* of a *bind join*
- Result of a *work request*:
  - sets of *plans*
  - each *plan* contains a list of properties that describe which parts of the work request are implemented by the plan and what the costs are

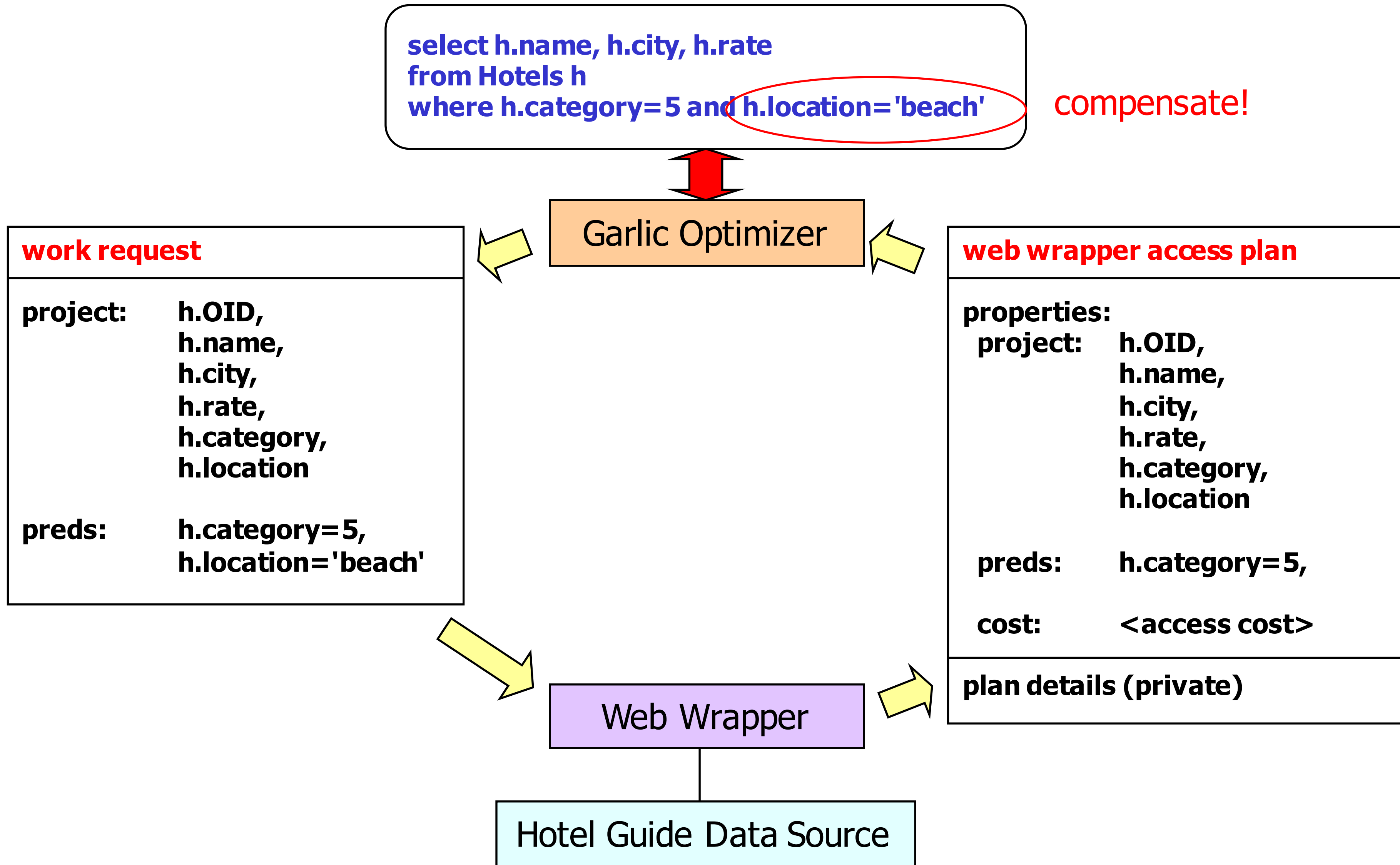
Example

- Query: *Find the name and room rate of all 5-star hotels in small towns in Greece located on the beach*
- Query statement:

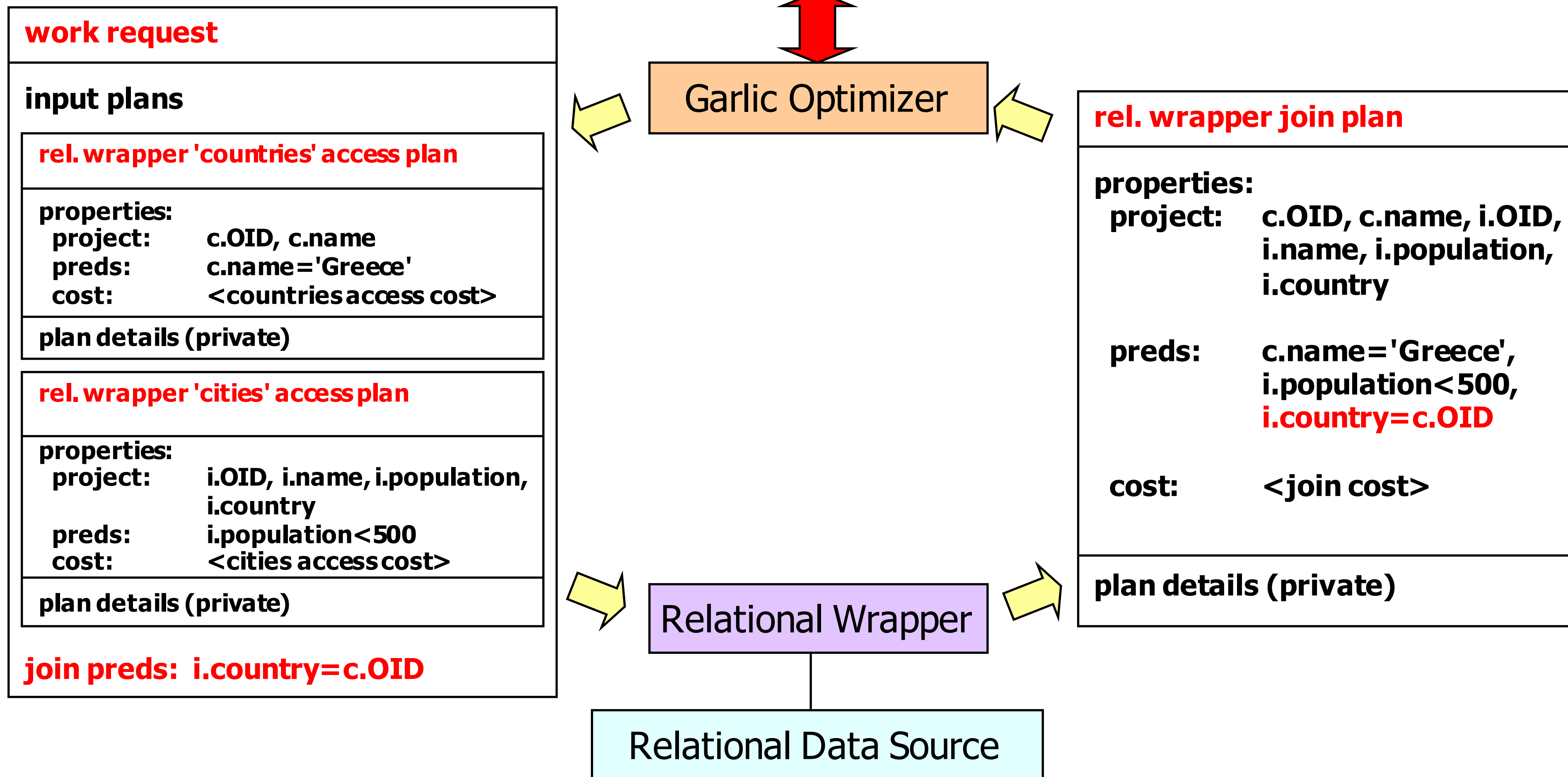
```
select h.name, h.rate
from Hotels h, Countries c, Cities i
where h.category=5 and h.location='beach' and c.name='Greece'
  and i.population<500 and h.city=i.name and i.country=c.OID
```



Single Collection Access Plan

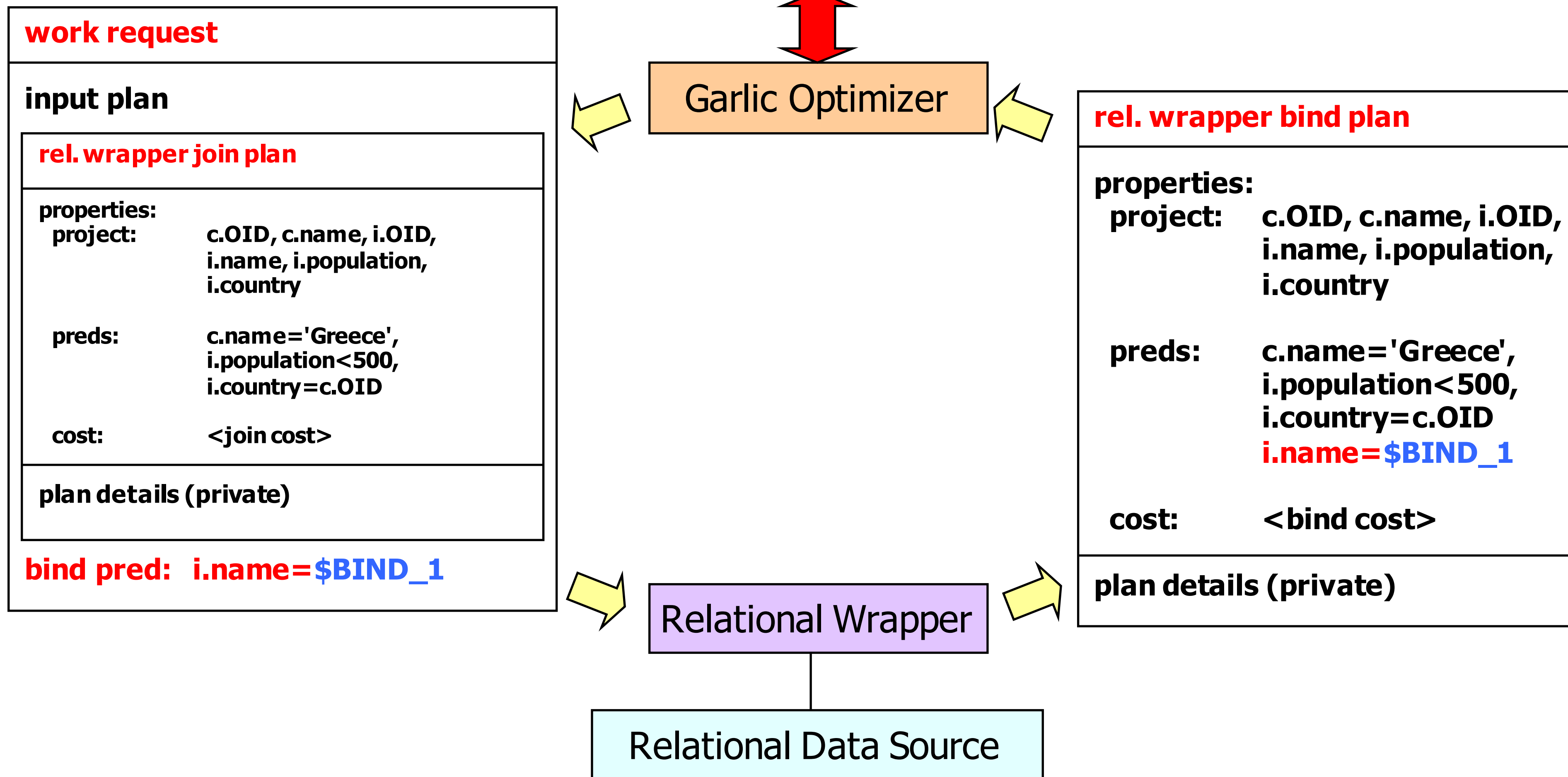


Join Plan



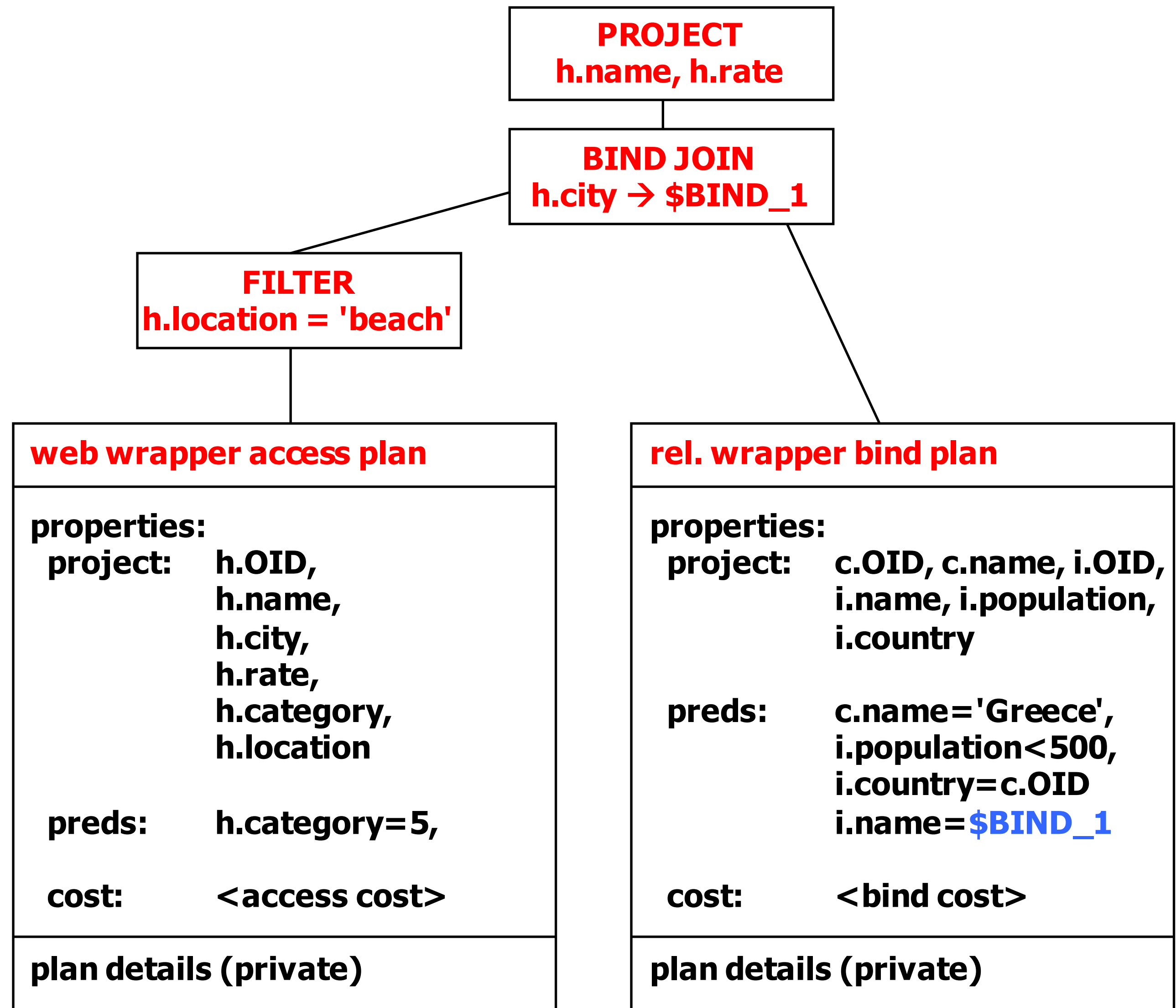
© Prof.Dr.-Ing. Stefan Deßloch

Bind Plan



Wrapper Plan Synthesis

- Plan generation needs to be supported by wrapper methods
- Plan execution has to be supported by wrapper as well (Iterator methods)



Wrapper Packaging

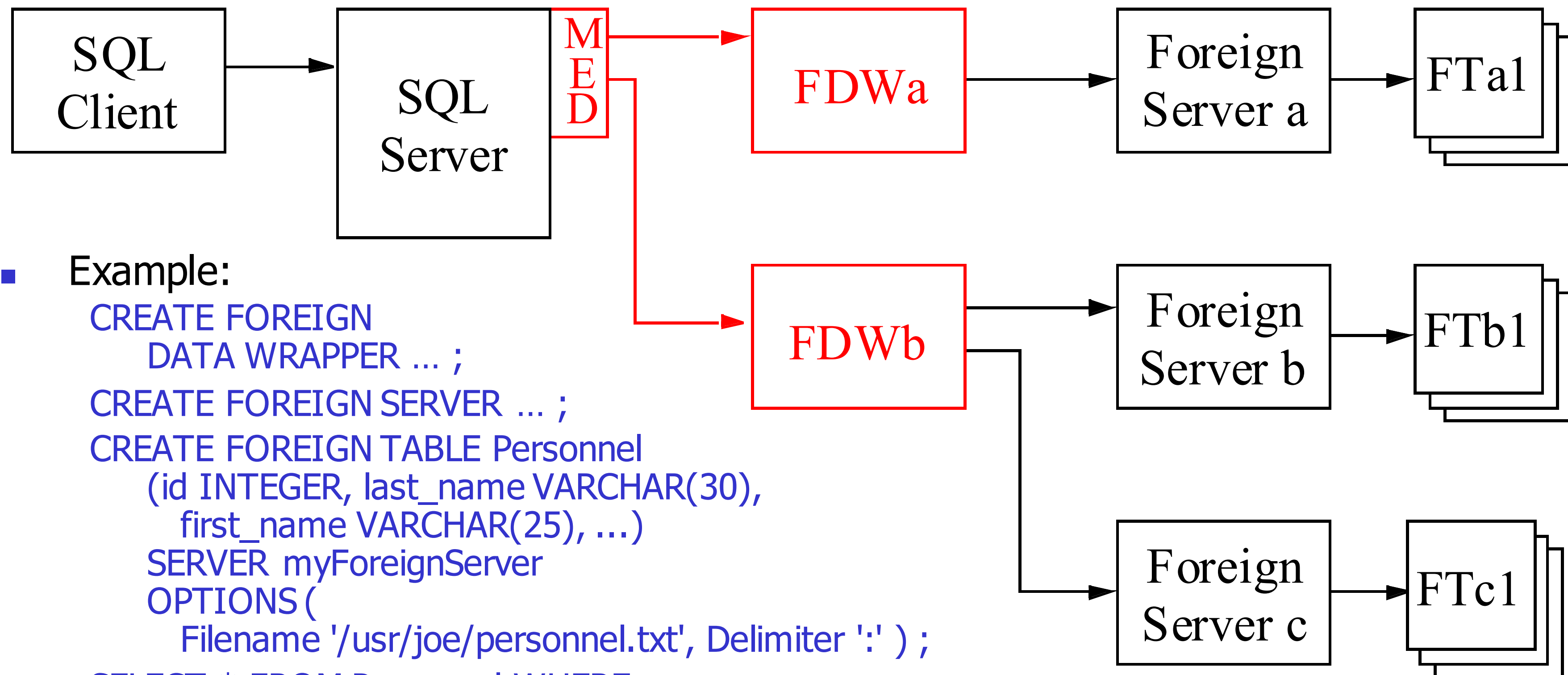
- Wrapper program provides the following wrapper components in a package:
  - interface files
    - GDL definitions
  - environment files
    - support for data-source-specific information
  - libraries
    - schema registration
    - method calls
    - query processing interfaces

SQL/MED

- Part 9 of **SQL**:1999: **M**anagement of **E**xternal **D**ata
  - extended in SQL:2003
- Two major parts
  - Datalinks
  - Foreign Data Wrapper / Foreign Data Server

Foreign Data Wrapper/Server

- Concept based on Garlic idea
  - data provided as tables instead of object collections
- Model:



- Example:

```
CREATE FOREIGN
    DATA WRAPPER … ;
CREATE FOREIGN SERVER … ;
CREATE FOREIGN TABLE Personnel
    (id INTEGER, last_name VARCHAR(30),
      first_name VARCHAR(25), ...)
    SERVER myForeignServer
    OPTIONS (
      Filename '/usr/joe/personnel.txt', Delimiter ':' ) ;
SELECT * FROM Personnel WHERE … ;
```

Foreign Data Server

- Manages data stored outside the SQL server
- SQL server and SQL client use foreign server descriptors (catalog elements) to communicate with foreign servers
- Catalog (implementation-specific):
  - SQL schemas
  - Foreign server descriptors
  - Foreign table descriptors
  - Foreign wrapper descriptors
- Foreign table
  - stored in a (relational) foreign server or dynamically generated by foreign wrapper capabilities
- Modes of interaction
  - *Decomposition*
    - SQL query is analyzed by SQL server, communicating with foreign data wrapper using *InitRequest*
  - *Pass-Through* (see discussion of *TransmitRequest*)

Foreign Data Wrapper Interface

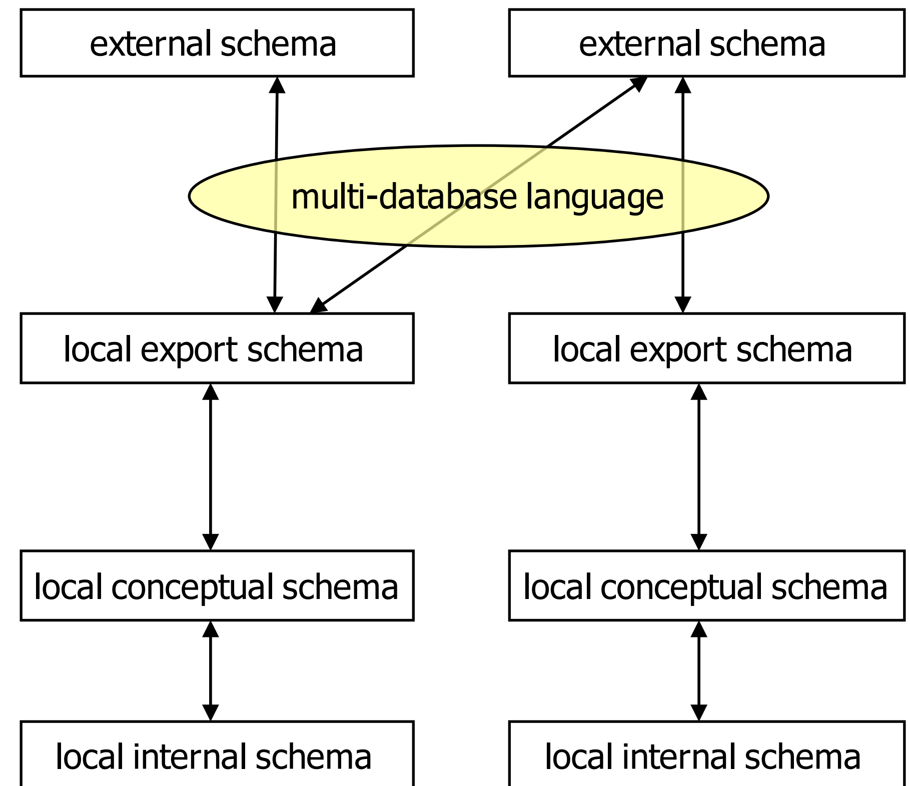
- Handle routines
- Initialization routines
  - AllocDescriptor
  - AllocWrapperEnv
  - ConnectServer
  - GetOps: request meta data about
    - foreign data wrapper/server capabilities
    - foreign table (columns)
  - InitRequest: initializes processing of a request (query)
- Access routines
  - Open
  - Iterate: for delivering foreign data to SQL server
  - ReOpen
  - Close
  - GetStatistics
  - TransmitRequest: „pass-through" of a query/request using the proprietary language of the foreign server

Security, Updates, and Transactions

- User Mapping
  - defines mapping of SQL server user-ids to corresponding concept of a foreign server
  - example:
    - ```
      CREATE USER MAPPING FOR dessloch
                SERVER myforeignserver
                OPTIONS
                              (user_id 'SD',
                               user_pw 'secret')
      ```
- Updates, transactions on external data
  - not supported in SQL/MED
    - goal for future version of the standard
  - provided as product extensions
    - usually, updates on non-relational data sources are not supported
      - distributed TAs are useful, read-only optimization can be used for foreign data source
    - updates on relational data sources
      - pass-through
      - transparent
      - distributed TAs supported
      ```

Multi-database Systems

- Loose coupling of systems
  - systems are autonomous
    - schema design autonomy
  - permit external/global applications to access data
- No global schema
  - local export schemas describe available data
  - actual integration needs to be performed by the application
  - only location transparency and physical distribution transparency
- Multi-database language
  - allows access of multiple data bases in a single query
  - directly references export schemas
- Data model heterogeneity needs to be handled either by the local data source or the multi-database language



Limitations of SQL

- Standard SQL is unable to generically solve most forms of schematic heterogeneity
- Comp. Person – Men/Women example

Schema A

Person

ID	Name	Gender
1234	Bob	male
4567	Jane	female

⟺

Schema B

Men

ID	Name
1234	Bob

Women

ID	Name
4567	Jane

- Can be solved with relational view(s)...

A to B

```
CREATE VIEW Men AS
SELECT ID, Name
FROM Person
WHERE Gender='male'

CREATE VIEW Women AS
SELECT ID, Name
FROM Person
WHERE Gender='female'
```

⟺

B to A

```
CREATE VIEW (ID, Name, Gender)
AS
SELECT ID, Name, 'male'
FROM Men
UNION
SELECT ID, Name, 'female'
FROM Women
```

- … but only because the number of different "categories" (here: genders) is known a priori (and fixed)

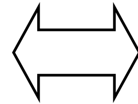
Limitations of SQL (cont.)

- e.g., replace gender with department:

Schema A

Person

<u>ID</u>	Name	Department
1234	Bob	Accounting
4567	Jane	Sales
9876	**Joe**	**Service**



Schema B

Accounting

<u>ID</u>	Name
1234	Bob

Sales

<u>ID</u>	Name
4567	Jane

Service

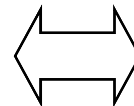
<u>ID</u>	**Name**
9876	**Joe**

- Departments might change over time
- When using static views as before
  - Each new department in A requires its own view definition to transform to schema B
  - Each new department in B requires a modification of the view to transform to schema A
- ➡ Expensive maintenance

A to B

```
CREATE VIEW Accounting AS
…
CREATE VIEW Sales AS
…

CREATE VIEW Service AS
SELECT ID, Name
FROM Person
WHERE Department ='Service'
```



B to A

```
CREATE VIEW (ID, Name, Department)
AS
SELECT ID, Name, 'Accounting'
FROM Accounting
UNION
SELECT ID, Name, 'Sales'
FROM Sales
UNION
SELECT ID, Name, 'Service'
FROM Service
```

Schematic Query Languages

- Solution: Extend SQL to be able to transform data to metadata (and v.v.)
- ➨ Schematic Query Languages (a.k.a. Multi-database QLs)
- Examples
  - SchemaSQL
  - FIRA/FISQL
- Challenge:
  - The schema of the result of a query is now dependent on the data actually present in the input relations
  - To allow such *dynamic schemas*, schematic query languages have to extend the relational model
- In addition, schematic query languages provide mechanism to access different databases in a single query

Example Databases

Kaiserslautern (KL)

Sales

Store	Department	AvgSales
Innenstadt	TV	139000
Innenstadt	Computer	156000
Innenstadt	Hifi	118000
Gewerbegbt	TV	112000
Gewerbegbt	Computer	180000
Gewerbegbt	Hifi	57000

Mannheim (MA)

AvgSales

Store	TV	Computer	Hifi
Quadrate	205000	234000	108000
Kaefertal	90000	76000	87000
Sandhofen	73000	81000	98000

Trier (TR)

Eisenbahnstr

Dept	AvgSales
TV	67000
Computer	51000
Hifi	78000

Hauptstr

Dept	AvgSales
TV	74000
Computer	103000
Hifi	89000

SchemaSQL

- Lakshmanan, Sadri & Subramanian [LSS96, LSS01]
- First approach that addresses the issue of schematic heterogeneity with SQL
- Built on top of SQL by providing an extended FROM clause:
  - Specify range variables ("aliases") not only over tuples of relations, but also over
    - the databases of the (M)DBMS `->`
    - the relation names of a database `db->`
    - attribute names of a relation `db::rel->`
    - tuples of a relation (-> SQL) `db::rel`
    - distinct values of an attribute `db::rel.attr`
  - Elements of the FROM clause can be nested, e.g.
    `FROM xdb-> xdbtables, xdbtables-> atts`
    to iterate over the relations of database xdb and then over the relations' attributes
  - Variables in the FROM clause can be used in view definitions for dynamic result schemas

SchemaSQL – Example

- Transform KL database to MA format:

```
CREATE VIEW KL2MA::AvgSales(Store, KD) AS
SELECT KS.Store, KS.AvgSales
FROM KL::Sales KS, KS.Department KD
```

1
3
2

1. Dynamic result schema: number of attributes depends on number of attribute values in the source relation's department attribute
2. Nesting of sets in FROM clause
3. A source tuple's value for AvgSales is placed in the result column depending on the value of the tuple's Department attribute (merge into one result tuple is implicit)

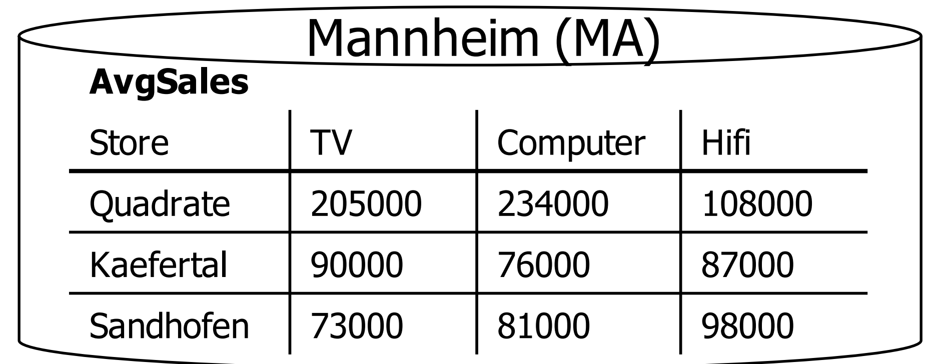
- Problem: Operation (the merge) is not well-defined for all source relations
  - What happens if there was an additional tuple ("Innenstadt","Hifi", 97500) in the KL database? Which value (11800 or 97500) to place into the "Hifi" column?
  - SchemaSQL does not answer this question

SchemaSQL – Example (cont.)

- Aggregation over a variable number of columns
- e.g. “What are the average sales of the Mannheim stores, across all departments?”
- Number of departments cannot assumed to be fixed!

```
SELECT MS.Store, AVG(MSAtts)
FROM MA::AvgSales MS, MA::AvgSales-> MSAtts
WHERE MSAtts<>'Store'
GROUP BY MS.Store
```

- Use of attribute set in aggregate function



Store	TV	Computer	Hifi
Quadrate	205000	234000	108000
Kaefertal	90000	76000	87000
Sandhofen	73000	81000	98000

SchemaSQL – Criticism

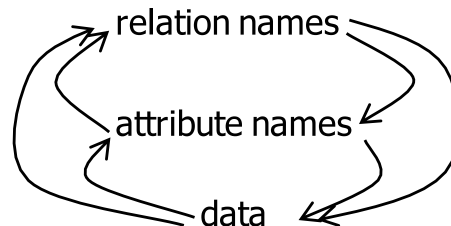
- Semantics of a SchemaSQL SELECT statement differs depending on context:
  - e.g., query from Example 2, placed in a view definition:

```
CREATE VIEW MA::PerDeptAvgs(Store, MSAtts) AS
    SELECT MS.Store, AVG(MSAtts)
    FROM MA::AvgSales MS, MA::AvgSales-> MSAtts
    WHERE MSAtts<>'Store'
    GROUP BY MS.Store
```

- Query now computes the averages for each department *individually*!

FIRA/FISQL

- Presented by Wyss and Robertson [WyRo05]
- Extends the relational model to the *federated relational model*
  - Number of output relations and their attributes is fully dynamic
- Provides an extended SQL syntax (Federated Interoperable SQL, FISQL)
- Provides a sound theoretical foundation by specifying the underlying algebra operators (Federated Interoperable Relational Algebra, FIRA)
- FIRA/FISQL is *transformationally complete*:
  - Transform any form of relational metadata to data and v.v.



- FISQL allows nesting of queries

FIRA/FISQL Data Model

- *Federated* relational data model:
  - Extends the relational model to incorporate metadata
    - A federated tuple is a mapping from a finite set of names S (=attribute names) to values; S is known as the schema of the tuple.
    - A federated relation has a name and contains a finite set of federated tuples
    - A federated database has a name and consists of a finite set of federated relations
    - A federation consists of a finite set of federated databases
    - The schema of a federated relation is the union of the schemas of the tuples
      - ➨ Operations that add/change/delete tuples may modify the relation schema
- Defines federated counterparts of the six standard relational operators, e.g.
  - Renaming of relations (in addition to attributes)
  - Cartesian product/union/difference of databases
- Introduces six new operators
- Most operators are defined on federated relations and on federated databases, i.e. operators take a relation/database as input and produce a relation/database as output

FIRA/FISQL – Operators

- Drop-projection $\mathfrak{u}_A(R)$, $\mathfrak{u}_A(D)$
  - Two variants: one for relations, one for federated databases
  - Parameter A is the set of attributes to be *removed* from the relation/fed. DB
  - Required to generically handle relations/fed. DBs with variable schema
- Down $\downarrow_I(R)$, $\downarrow_I(D)$
  - Two variants: one for relations, one for federated databases
  - "Demotes" a table R's metadata to data by creating a relation *metadata*$_i$ and forming its crossproduct with R.
  - For a relation R with name N and attributes $A_1 \ldots A_n$, the relation *metadata*$_i$ is defined as:
  - Ignores metadata columns: $\downarrow_i(R) = metadata_i(R) \times \mathfrak{u}_{r_i, a_i}(R)$

metadata$_i$(R)=

r_i	a_i
N	A_1
N	A_2
…	…
N	A_n

- Attribute Dereference $\Delta_A^B(R)$
  - The value of attribute B of the target tuple t is determined by using the value found in the attribute named equal to t's value in column A, values of all other attributes of t are equal to the respective value of those in source tuple s
  - Let t[X] denote the value of attribute X of tuple t. The attribute values of a result tuple t are obtained from the values of its respective source tuple s like this:

$$t[X] = \begin{cases} s[s[A]] \text{ if } X = B \\ s[X] \text{ otherwise} \end{cases}$$

FIRA/FISQL – Operators (cont.)

- Generalized Union $\Sigma(D)$
  - Creates a relation holding the outer union of all relations in the database D
- Transpose $\tau_A^B(R)$
  - For each distinct value of the parameter column B in the input relation R, create a column in the result relation (whose name is the respective value of B)
  - For each tuple t of the result relation, obtain the value of column X (denoted t[X]) from the respective source tuple s like this:

$$t[X] = \begin{cases} s[A] \text{ if } X = s[B] \\ s[X] \text{ if } X \in \text{schema(s)} \\ \text{NULL otherwise} \end{cases}$$

  - i.e.: for each new attribute N_i, its value is that of the source tuple's A attribute if the source tuple's B attribute value is equal to the name of attribute N_i, NULL otherwise
  - other attributes remain unchanged
- Partition operator $\wp_A(R)$
  - Roughly the opposite of Generalized Union
  - Creates a federated database with one relation for each distinct value in column A of input relation R

FIRA/FISQL example – KL2TR

- Transform the Kaiserslautern database to the format of the Trier database
- Requires the Partition operator $\wp_A(R)$ and (drop) projection

Kaiserslautern (KL)

Sales

Store	Department	AvgSales
Innenstadt	TV	139000
Innenstadt	Computer	156000
Innenstadt	Hifi	118000
Gewerbegbt	TV	112000
Gewerbegbt	Computer	180000
Gewerbegbt	Hifi	57000

$KL2TR' = \wp_{Store}(KL)$

KL2TR'

Innenstadt

Store	Department	AvgSales
Innenstadt	TV	139000
Innenstadt	Computer	156000
Innenstadt	Hifi	118000

Gewerbegbt

Store	Department	AvgSales
Gewerbegbt	TV	112000
Gewerbegbt	Computer	180000
Gewerbegbt	Hifi	57000

$KL2TR = \text{\u{2C6F}}_{Store}(KL2TR')$

OR

$KL2TR = \pi_{Department, AvgSales}(KL2TR')$

KL2TR

Innenstadt

Department	AvgSales
TV	139000
Computer	156000
Hifi	118000

Gewerbegbt

Department	AvgSales
TV	112000
Computer	180000
Hifi	57000

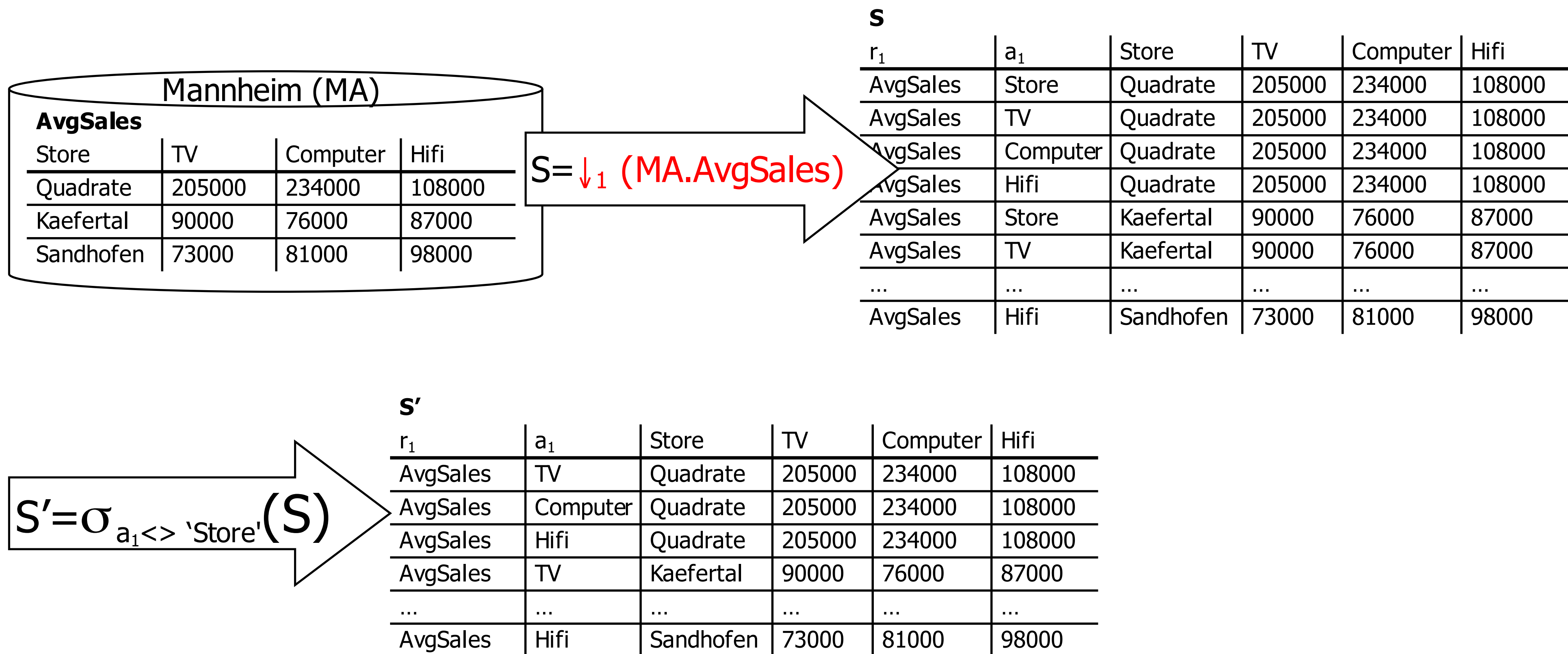
- FISQL statement:

```
SELECT KS.Department AS Dept, KS.AvgSales INTO KS.Store
FROM KL.Sales KS
```

A

FIRA/FISQL example – MA2KL

- Transform the Mannheim database to the format of the Kaiserslautern database
- Requires a combination of the down and attribute deference operator



AvgSales

Store	TV	Computer	Hifi
Quadrate	205000	234000	108000
Kaefertal	90000	76000	87000
Sandhofen	73000	81000	98000

S

r_1	a_1	Store	TV	Computer	Hifi
AvgSales	Store	Quadrate	205000	234000	108000
AvgSales	TV	Quadrate	205000	234000	108000
AvgSales	Computer	Quadrate	205000	234000	108000
AvgSales	Hifi	Quadrate	205000	234000	108000
AvgSales	Store	Kaefertal	90000	76000	87000
AvgSales	TV	Kaefertal	90000	76000	87000
…	…	…	…	…	…
AvgSales	Hifi	Sandhofen	73000	81000	98000

S'

r_1	a_1	Store	TV	Computer	Hifi
AvgSales	TV	Quadrate	205000	234000	108000
AvgSales	Computer	Quadrate	205000	234000	108000
AvgSales	Hifi	Quadrate	205000	234000	108000
AvgSales	TV	Kaefertal	90000	76000	87000
…	…	…	…	…	…
AvgSales	Hifi	Sandhofen	73000	81000	98000

FIRA/FISQL – MA2KL (cont.)

$S'' = \Delta^{AvgSales}_{a_1}(S')$

S''

r_1	a_1	Store	TV	Computer	Hifi	**AvgSales**
AvgSales	**TV**	Quadrate	**205000**	234000	108000	**205000**
AvgSales	**Computer**	Quadrate	205000	**234000**	108000	**234000**
AvgSales	**Hifi**	Quadrate	205000	234000	**108000**	**108000**
AvgSales	**TV**	Kaefertal	**90000**	76000	87000	**90000**
…	…	…	…	…	…	
AvgSales	**Hifi**	Sandhofen	73000	81000	**98000**	**98000**

- Cleanup:
  MA2KL= $\pi_{\text{Store, Department, AvgSales}}\ \rho_{\text{a1->Department}}(S'')$

MA2KL

Store	Department	AvgSales
Quadrate	TV	205000
Quadrate	Computer	234000
Quadrate	Hifi	108000
Kaefertal	TV	90000
…	…	
Sandhofen	Hifi	98000

FIRA/FISQL example – TR2KL

- Use Down (on DB) with Generalized union, renaming and projection:
  TR2KL= $\rho^{\varepsilon\text{->Sales}} \pi_{\text{Store, Dept, AvgSales}}\ \rho_{\text{r1->Store}}\ \Sigma(\downarrow_1 (\text{TR}))$

Trier (TR)

Eisenbahnstr		Hauptstr	
Dept	AvgSales	Dept	AvgSales
TV	67000	TV	74000
Computer	51000	Computer	103000
Hifi	78000	Hifi	89000

$\text{TR}' = \downarrow_1 (\text{TR})$

TR`

Eisenbahnstr

r_1	a_1	Dept	AvgSales
Eisenbahnstr	Dept	TV	67000
Eisenbahnstr	AvgSales	TV	67000
Eisenbahnstr	Dept	Computer	51000
…	…	…	…
Eisenbahnstr	AvgSales	Hifi	78000

Hauptstr

r1	a
Hauptstr	D
Hauptstr	A
…	..

$\text{TR}'' = \Sigma\ (\text{TR}')$

TR``

ε

r_1	a_1	Dept	AvgSales
Eisenbahnstr	Dept	TV	67000
Eisenbahnstr	AvgSales	TV	67000
Eisenbahnstr	Dept	Computer	51000
…	…	…	…
Hauptstr	AvgSales	Hifi	89000

TR2KL= $\rho^{\varepsilon\text{->Sales}}\ \pi_{\text{Store, Dept, AvgSales}}\ \rho_{\text{r1->Store}}\ (\text{TR}'')$

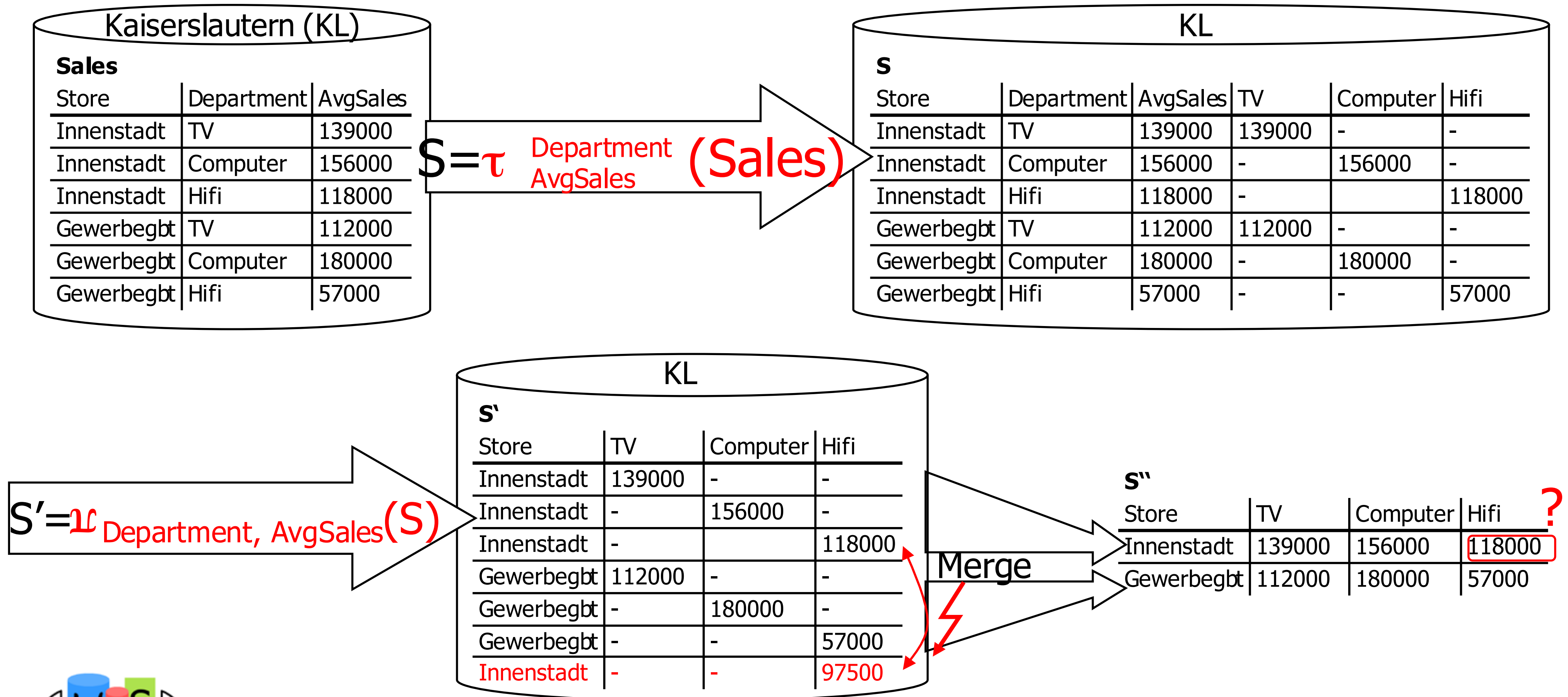
TR2KL

Sales

Store	Dept	AvgSales
Eisenbahnstr	TV	67000
Eisenbahnstr	TV	67000
Eisenbahnstr	Computer	51000
…	…	…
Hauptstr	Hifi	89000

FIRA/FISQL example – KL2MA

- Transform the Kaiserslautern database to the format of the Mannheim database
- Requires the transpose and drop-projection operators:



© Prof.Dr.-Ing. Stefan Deßloch

FIRA/FISQL – Merging

- Merging of tuples required
  - Merging is simple if no "conflicts" arise
  - Merge not uniquely defined if tuples conflict
  - Two tuples t_1, t_2 of a relation with n attributes are mergeable if either
    - $t_1[A_i] = t_2[A_i]$ or
    - one of $t_1[A_i]$ or $t_2[A_i]$ is a null value

    holds for $1 \leq i \leq n$
  - The merge t of two mergeable tuples t_1, t_2 (denoted $t = t_1 \odot t_2$) is defined as

$$t[A_i] = \begin{cases} t_1[A_i] \text{ if } t_1[A_i] \text{ not null} \\ t_2[A_i] \text{ otherwise} \end{cases} \quad \text{for } 1 \leq i \leq n$$

- Optimal tuple merge
  - For a relation schema R and two relations r_1 and r_2 that are instances of R, r_2 is a tuple merge of r_1, if it can be obtained from r_1 by a finite sequence of merge operations of mergeable tuples
  - A tuple merge r_2 of r_1 is an optimal tuple merge, if for every r_3 that is also a tuple merge of r1
    $|r_2| \leq |r_3|$ holds

FIRA/FISQL – Merge Operator

- (Unique optimal tuple) Merge Operator $\mu(R)$ [WyRo05b]
  - Let R be a relational schema, and r an instance of R
  - Let $\emptyset^R$ denote the empty relation of schema R
  - Then the unique optimal tuple merge of r is

$$\mu(r) := \begin{cases} \emptyset^R \text{if there is more than one optimal tuple merge of r} \\ \text{the unique optimal tuple merge otherwise} \end{cases}$$

- Merge was not part of the original FIRA/FISQL
  - ➨ No FISQL syntax specified
- FISQL statement (without merge):

```
SELECT DROP (KS1.Department, KS1.Avgsales)
FROM    (SELECT KS.*, KS.Avgsales ON KS.Department
         FROM KL.Sales AS KS)
        AS KS1
```

S' (whole statement), S (inner subquery), A (KS.Avgsales), B (KS.Department)

FIRA/FISQL example – KL2MA continued

KL

S'

Store	TV	Computer	Hifi
Innenstadt	139000	-	-
Innenstadt	-	156000	-
Innenstadt	-		118000
Gewerbegbt	112000	-	-
Gewerbegbt	-	180000	-
Gewerbegbt	-	-	57000

$S'' = \mu_{Store}(S')$

S''

Store	TV	Computer	Hifi
Innenstadt	139000	156000	118000
Gewerbegbt	112000	180000	57000

KL

S'

Store	TV	Computer	Hifi
Innenstadt	139000	-	-
Innenstadt	-	156000	-
Innenstadt	-		118000
Gewerbegbt	112000	-	-
Gewerbegbt	-	180000	-
Gewerbegbt	-	-	57000
Innenstadt	-	-	97500

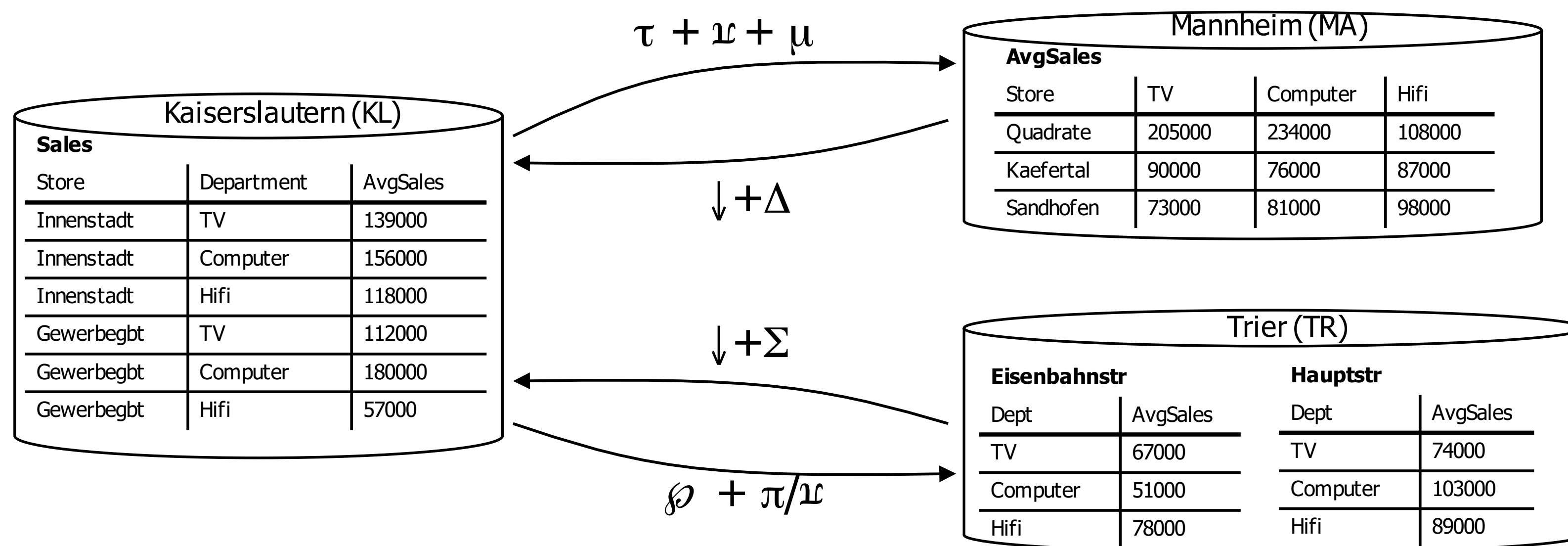
$S'' = \mu_{Store}(S')$

S''

Store	TV	Computer	Hifi

FIRA/FISQL – Summary

- Theoretically sound approach to resolve schematic heterogeneity
- Open questions:
  - How does grouping/aggregation fit into the model?
    - Group by/aggregate over an unknown set of attributes ?
    - Could allow the user to solve the merge problem for relations with conflicting tuples by explicitly specifying the desired merge semantics (using an aggregate function)
  - What does transformational completeness mean in the XML data model?



Summary

- Architectures for virtual data integration
  - distributed DBMS, federated DBMS, mediator-based systems
    - based on a global schema, can support location and distribution transparency
  - multi-database systems
    - no global schema, only support location transparency
- Wrappers as important infrastructure
  - Advantages
    - provide a common interface for integration middleware to interact with arbitrary data sources
    - overcomes heterogeneity regarding data model, API
  - Garlic (IBM)
    - almost any data source can be integrated
    - global query optimization
      - middleware (Garlic) and wrapper decide dynamically which query fragments are processed by the wrapper
      - specific capabilities of data sources can be utilized
      - function compensation by Garlic for query capabilities not supported by a wrapper

Summary (cont.)

- SQL/MED – Management of External Data
  - Foreign-Data-Wrapper/Server/Table
    - provides standardized support for extending SQL engine to access external data sources
    - follows the Garlic idea
    - Limitations: no standardized update operations, no transactional support
- Multi-database systems
  - loose coupling of systems, no global schema
  - multi-database languages
    - allows access of multiple data bases in a single query
    - directly references export schemas
    - may also support bridging schematic heterogeneity
  - SchemaSQL
    - variables in SQL FROM-clause can now also range over databases, tables, attributes, distinct values
  - FIRA/FISQL
    - solid theoretical foundation, providing new algebraic operators