Multi-databases, schematic heterogeneity and queries over multiple schemata

Yannis Roussos

Outline

- **Motivation**
- Multi database systems
- MSQL
- SchemaSQL
- Discussion: Data Management in web 2.0, wiki-dbs etc.

Motivating Example

- Worldwide electronic marketplace
- Sellers:
  - Retail companies.
  - Suppliers.
  - Independent users.
- Buyers: any registered user

  Have context information attached to them:
  - Automatically retrieved from the user profile.
  - Sent from the users in real time (as they browse the site).

Motivating Example

- Sellers: dynamically construct a structured schema for
  - Products
  - Catalogue
  - Collaborating stores
  - Other basic predefined relations
- How?
  - Set of predefined relation names
  - Set of predefined attribute names
  - Can request for the addition of new ‘building blocks’

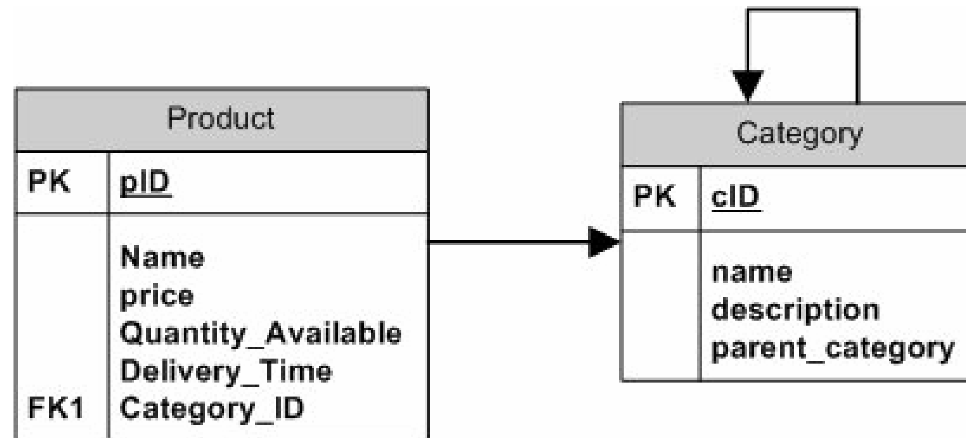
all with well defined global semantics…

Motivating Example

- Sellers: dynamically construct a structured schema.
- Why?
  - Each seller must be able to create more complex representations for his conceptual data model and business logic.
  - Loose intra-site interoperability → Intra site queries
  - Structured models are better (?).

Motivating Example

- An example of a simple schema created from supplier SA ….



Motivating Example

- Not enough for a motivating example…
- Buyers are able to:
  - Navigate the automatically generated catalogues of suppliers
  - Check offers
  - Order products
  - Pose more advanced queries like:
    - "Find me the best prices for CD-players"
    - Request for suppliers with delivery time below a threshold
    - Compare suppliers or products based on parameters

Motivating Example

- Registered users are characterized by a set of metadata defining their context:
  - Current user location
  - City and country
  - Current date and time
  - Browsing device
  - User tech expertise (e.g. naïve, advanced, expert)
  - Bandwidth

  etc…

Motivating Example

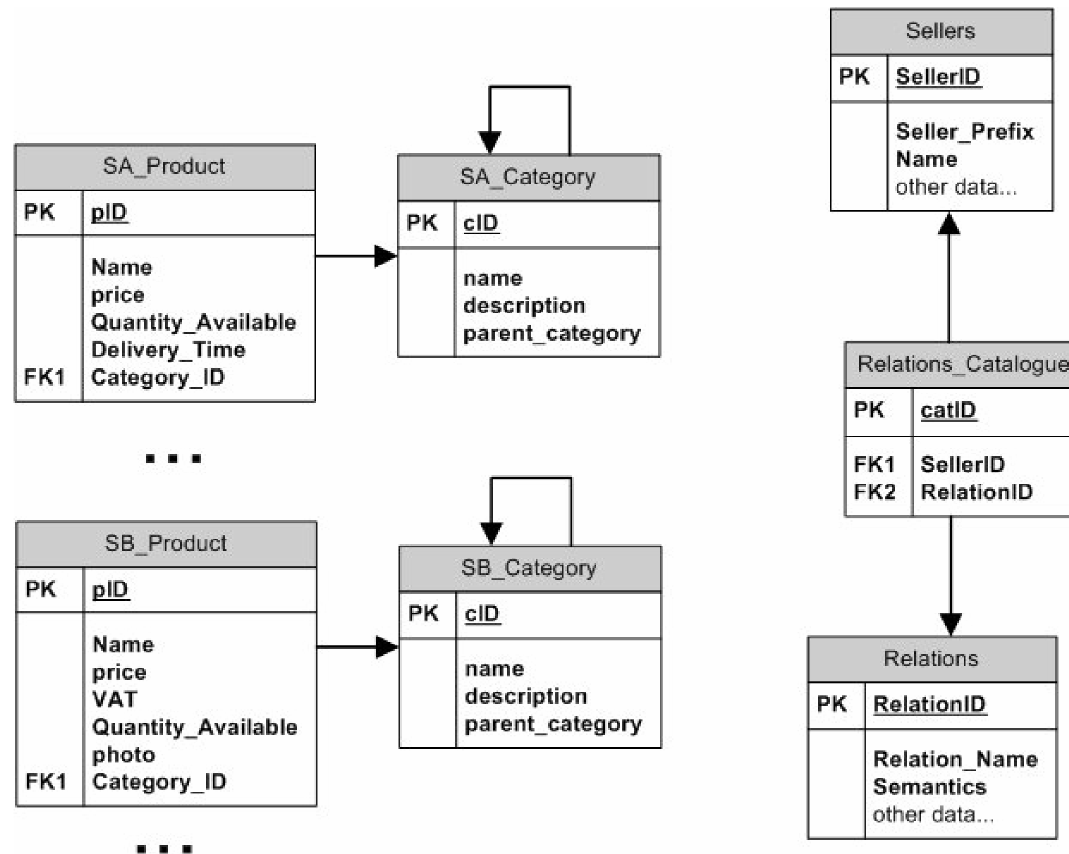
- Exploiting context information, sellers are able to create different Schemata for different contexts…
- Why? (examples)
  - offer specialized services or products for specific user categories.
  - a specific product from supplier SA is sold in Europe but not in USA.
  - price, available quantity and delivery time change according to country or even city.
  - special price offers can be arranged in advance for holidays, specific days or even hours of day.
  - etc…

Motivating Example

- Trivial Implementation?
  - A lot of application level programming..
  - How can you automate all this process?
    - 100s of schemata for each of 1000s of sellers…
    - Query rewritings for cross schema queries?
    - Maintenance?
    - Incremental update of different schemata?
  - How can you optimize your implementation?
  - Why not have a ***Database Model*** incorporating context? → let the DBMS do all the work!

Motivating Example

- Forget about context for a while…



Motivating Example

- What happens if we also add context?
- Model Suppliers as part of context for uniform treatment…
- Better implementations?

Motivating Example

denormalized relation solution	3-ary relation solution

Product (denormalized relation solution)

	Product
PK	productID
PK	SuplierID
PK	User_Country
PK	Date
PK	User_Type
PK	Device_Type
PK	{more context ...}
	Name
	price
	Quantity_Available
	Delivery_Time
	Category_ID
	VAT
	photo1
	photo2
	...
	description
	more_detailed_description
	...
	{ALL attributes ...}

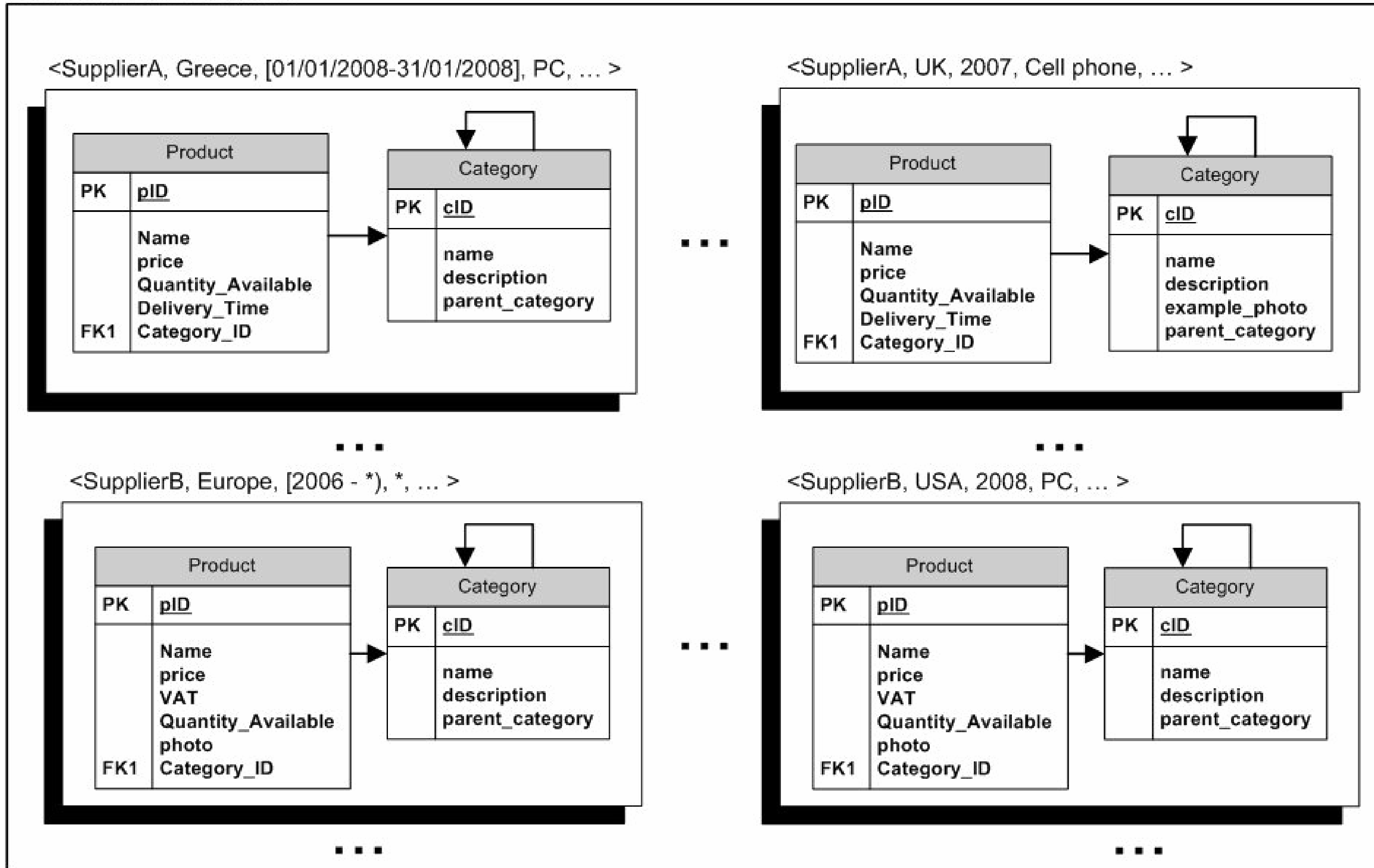
Product (3-ary relation solution)

	Product
PK	Oid (object identifier)
	Key (attribute name)
	Val (attribute value)

Motivating Example

- The problem with all of the above solutions, even when using an extended higher order DDL and Query language is that:
  - The semantics of context are lost.
  - A full knowledge of the specific solution and schemata is needed in order to be able to store and query the context aware information.
  - Complex Query rewritings are needed for even the simplest context queries.
  - Answering queries that refer to attributes not defined under specific contexts is not trivial.
  - They are ***implementation oriented*** and not **model oriented**.
  - They are difficult to manage and maintain under the assumption of changing context schema, e.g. addition of more context parameters.

Motivating Example – Solution?



Outline

- Motivation
- **Multi database systems**
- MSQL
- SchemaSQL
- Discussion: Data Management in web 2.0, wiki-dbs etc.

Multi database systems

- Multiple databases created for the same functionality
  - Different operating systems, data formats, query languages etc
- Typically DBs managed by DBMSs running on heterogeneous computing platforms
- Information sharing across dissimilar platforms
  - Interconnect previously isolated software systems (DBMS)
  - Not only invoke but also coordinate interactions

Interoperating with heterogeneous databases - requirements

- Distributed transparency-users must access a number of different databases in the same way as accessing a single database.
- Heterogeneity transparency-users must access other schemas in the same way they access their local database (using a familiar model and language).
- The existing database systems and applications must not be changed.

Interoperating with heterogeneous databases - requirements

- Addition of new databases must be easily accommodated into the system.
- The databases have to be accessed both for retrievals and updates.
- The performance of heterogeneous systems has to be comparable to the one of homogeneous distributed systems.

Solutions to integrating Heterogeneous Distributed Databases

- Global Schema Integration
- Federated Database systems
- Multi database language approach
- Views: LAVs, GAVs, GLAVs, etc

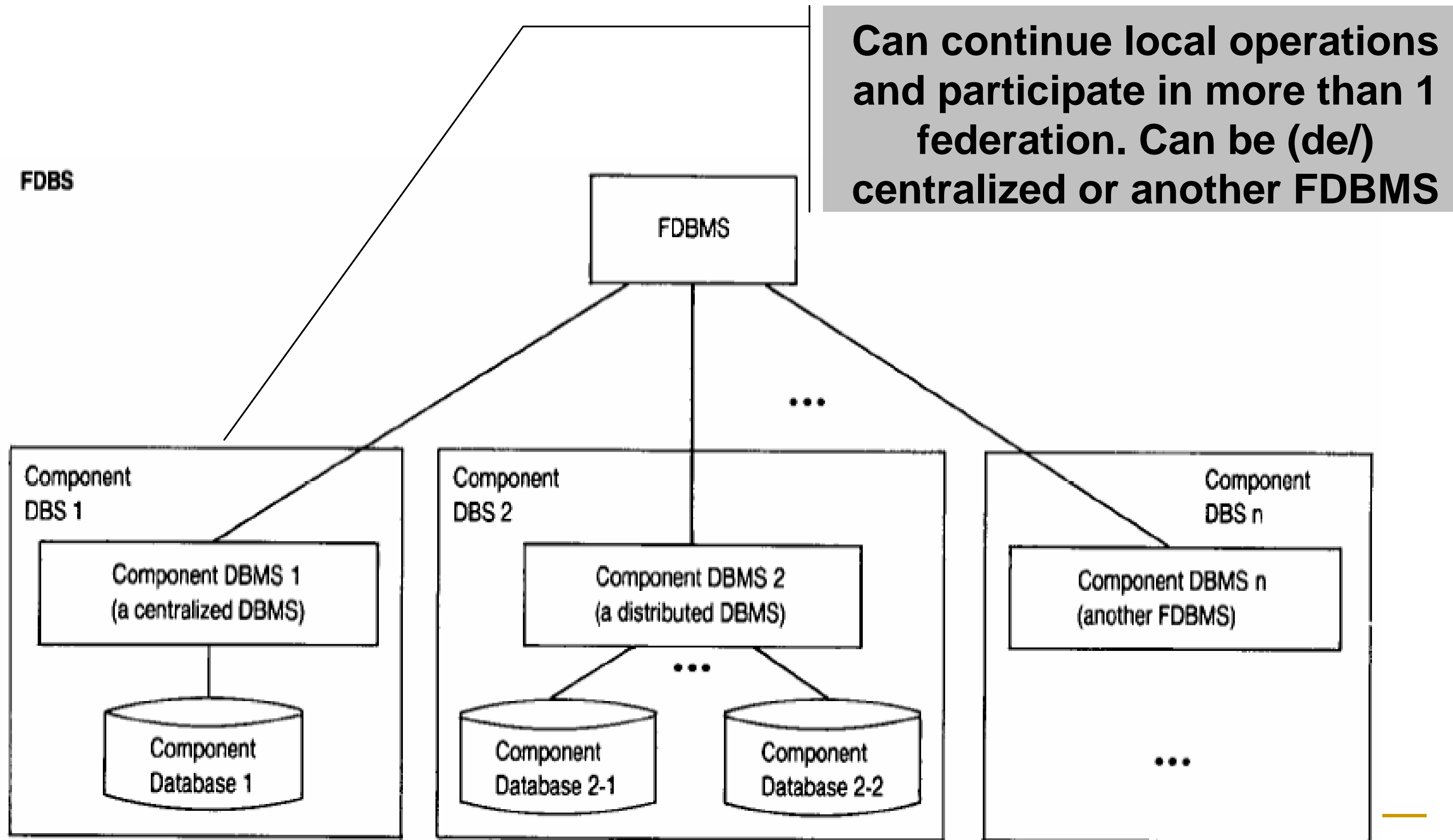
Global Schema Integration

- Based on complete integration to provide a single view
- Advantages:
  - Consistent, uniform view of and access to data for users
  - Users unaware of existing multiple existing DBs

Federated Database systems

- Aim: remove the need for static global schema integration
- Allows each local DB to have more control over the shareable information
- Control is decentralized
- Integration need not be complete but depends on needs of users

A FDBS and its components – cooperation among independent systems



FDBs

- Compromise between
  - no integration in which users must explicitly interface between multiple autonomous DBs

  AND
  - Total integration in which autonomy of each component DBS is sacrificed so that users can access data through a single global interface but not as a local user
- Support local and global (federated) operations

FDBSs Schemas

- Local schema
  - Conceptual schema of a component DB
- Component schema
  - Local schema translated to a common data model of the FDBS. Alleviates data model heterogeneity.
- Export schema
  - Specify shareable objects to other members or classes of members of the FDBS.
- Federated schema
  - A statically integrated schema or dynamic view of multiple export schemas. Can be multiple federated schemas.
- External schema
  - For customization when the federated schema is large and complicated. Another level of abstraction for class of users for example.

Multi database language approach

- Aim: provide constructs that perform queries involving several DBs at the same time
- Has features not supported by traditional languages. Ex: a global name can be used to identify a group of DBs
- DBs covering same subject are grouped under a collective name. Inter DB relationships are specified in the dependency schemas

Multi database language approach - disandvantages

- Lack of distribution and location transparency for users.
- Users responsible for
  - finding relevant DBs,
  - understanding schemas,
  - detecting and resolving semantic conflicts
  - performing view integration
- Some support offered by the language constructs

Outline

- Motivation
- Multi database systems
- **MSQL**
- SchemaSQL
- Discussion: Data Management in web 2.0, wiki-dbs etc.

MSQL

(Litwin, Abdellatif, Nicolas, Zeroual, 1989
L. Suardi, M. Rusinkiewicz, 1992)

- An extension to SQL
  - Contains by definition every SQL-x
- Allows for non-procedural multidatabase base manipulations
  - ∃ MSQL queries impossible to formulate in SQL
  - An MSQL query may replace several SQL queries
- Developed in 1986-89
  - INRIA, projet B A BA,
  - Dr. Thesis of MM. Abdellatif, Nicolas, Zeroual
- Compiler implemented at Houston University
  - Team of Prof. M. Rusinkiewicz, 1990-1993

MSQL

(Litwin, Abdellatif, Nicolas, Zeroual, 1989
L. Suardi, M. Rusinkiewicz, 1992)

- Research vehicle for functions for the MBD environment
  - to address relations in different databases
  - to manipulate semantically heterogeneous data
  - to create MDB views
  - to transfer data (and schemas) between DBs
  - to define MDB dependencies
- Present to limited extent in most of commercial DBMSs

MSQL

(Basic new properties)

- SQL Query
  - Uses 1st order predicate calculus
  - Is compiled for optimization into the relational algebra
  - Result is a table
- MSQL Query
  - May use higher-order predicate calculus
  - Is compiled for optimization into the *multirelational* algebra
  - Result is a *multitable*
    - A set of relations (tables)
    - May be constituted from one or no tables

MSQL

(More on functions specific to the MDB environment)

- Addressing of tables in different DBs
  - Implicitly or by qualification by (multi)database names
    - Introduced around 1985 by relational multibase system prototype MRDSM
    - Unknown at that time of any relational language
      - See the overview of relational DBMSs existing in 1987 (M. Brodie)

Conceptual Schemas

(the multischema)

DB bnp :

br (br#, brname, street, street#, city, zipcode, tel)
account (acc#, cl#, balance, br#)
client (cl#, clname, cltel, cltype, street, street#, city, zipcode)
spe-acc (acc#, br#, cl#, balance, curr)

DB sg :

branch (bra#, braname, street, s#, town, zip, t#, class)
acc (acc#, bra#, c#, balance)
client (c#, cname, ct#, ctype, street, s#, town, zip)

DB cic :

br (br#, brname, street, street#, city, zipcode, tel)
account (ac#, br#, cl#, balance, open_date)
client(cl#, clname, cltel, cltype, street, street#, city, zipcode)

Semantic Heterogeneity

In Banks

- Same names can designate different data
- Different names can designate same data
  - same client, same town..
- The value of a primary key is valid only in one DB
  - how to identify same client in diff. banks ?

MSQL Commands

- CREATE TABLE — CREATE DATABASE
- CREATE MULTIDATABASE — CREATE VIEW
- ALTER TABLE or — ALTER VIEW
- ALTER MULTIDATABASE
- DROP TABLE — DROP VIEW
- DROP MULTIDATABASE — DROP DATABASE

MSQL

CREATE DATABASE

```
> MSQL
        CREATE DATABASE  boulogne ;

        CREATE DB  |.com.org.user.boulogne ;
        CREATE MULTIDATABASE Banks (bnp cic sg );
```

Query scope

```
        USE Banks;
        CREATE DATABASE boulogne FROM bnp ;
```

MSQL

CREATE TABLE

Import

```
use banks ;
CREATE TABLE  boulogne.loan  FROM bnp.loan ;

CREATE TABLE  fake_checks
   (Chq#          INT,
    Montant_Euro       CURRENCY  [EURO]
    .... );
```

Unit of measure

One has created four (empty) tables :

bnp. fake_checks , cic. fake_checks ... boulogne.fake_checks

```
CREATE TABLE boulogne.client (c#, cn, ct#)
   FROM bnp.client (cl#, clname, cltel)
   PRIMARY KEY (c#)
   (cn, ct#)  OUTER REFERENCES (clname, cltel);
```

MSQL

CREATE TABLE with References

```
USE  AuPrintemps                          /* MDB AuPrintemps
CREATE TABLE  MusicDep.Inventory
    ….
    FOREIGN KEY (Item#)  REFERENCES Central.Stock(I#);
```

No unauthorized Item# in the inventory of the Music Department

- Other options
- PRIMARY KEY (…) REFERENCES T(…) ;
- [T1(A)] [LEFT|RIGHT] REFERENCES T2(B) ;
  - Generates implicit equijoin, or left or right implicit outerjoins when a query selects attributes A and B.

MSQL

Elementary queries

```
USE bnp cic
SELECT bnp.br.brname, cic.br.brname, bnp.br.street
    FROM bnp.br, cic.br
    WHERE bnp.br.street = cic.br.street ;
```

bnp.br.brname	cic.br.braname	bnp.br.street
vaugirard 3	bd. montparnasse	vaugirard

Prefixing with DB names   was unknown to SQL

Updates

```
USE  (bnp b)  sg ;
UPDATE account
SET account.balance = account.balance + 500
        WHERE account.balance > acc.balance
        AND b.client.clname = sg.client.cname AND
        b.client.street = sg.client.street ;
```

- What does it mean ?

Multiple Queries

```
USE Banks
SELECT *
    FROM br%
    WHERE street = 'champs elysées'  ;
```

```
USE bnp
    SELECT *
    FROM br
    WHERE street = 'champs elysées' ;
USE sg
    SELECT *
    FROM branch
    WHERE street = 'champs elysées' ;
USE cic
    SELECT *
    FROM br
    WHERE street = 'champs elysées' ;
```

Results (a multitable)

bnp.br

br#	brname	street	street#	city	zipcode	tel
123	sembat	champs elysées	130	Boulogne	92100	12 34
456	sevres	champs elysées	120	Sevres	92105	12 56

cic.br

bra#	braname	street	st#	town	zip	t#	class
123	jaures	champs elysées	153	Boulogne	92100	3214	A
765	sevres	champs elysées	20	Sevres	92105	1243	B

sg.branch

br#	brname	street	s#	city	zipcode	tel
abc	sembat	champs elysées	110	Boulogne	92100	12.45
a1f	gare	champs elysées	30	Chaville	92110	34.56

Semantic Variables in MSQL

```
use bnp sg
    let x be   town
    select *
    from b%
    where x = 'Paris' and street = 'r. de Rivoli'
```

```
use bnp
select *
from br
where town = 'Paris' and street = 'r. de Rivoli'
```

```
use sg
select *
from branch
where town = 'Paris' and street = 'r. de Rivoli'
```

DB bnp : br (br#, brname, street, street#, **city**, zipcode, tel)

DB sg : branch (bra#, braname, street, s#, **town**, zip, t#, class)

Semantic Variables in MSQL

```
use bnp sg
    let x be   town   city
    select *
    from b%
    where x = 'Paris' and street = 'r. de Rivoli'
```

Alternatively:

```
use bnp sg
    let x be   to%   city
    select *
    from b%
    where x = 'Paris' and street = 'r. de Rivoli'
```

Semantic Variables in MSQL

```
use banks
   let X be banks.*
   select a%, balance, c%name
   from X.a% a, X.c% c
where a. c% = c. c%
```

- The query illustrates the multitable *pair-wise* join

DB bnp :
account (acc#, cl#, balance, br#)
client (cl#, clname, cltel, cltype, street, street#, city, zipcode)

DB sg :
acc (acc#, bra#, c#, balance)
client (c#, cname, ct#, ctype, street, s#, town, zip)

DB cic :
account (ac#, br#, cl#, balance, open_date)
client(cl#, clname, cltel, cltype, street, street#, city, zipcode)

Semantic Variables in MSQL

```
use banks
   let x be   town   city
   let y be sg bnp
   select  Z.*
   from y.b% Z, cic.b% V
   where  V.x = 'Paris' and
          V.street =  Z.street and Z.x = 'Paris';
```

- What does it means ?
- Is the « natural » decomposition into SQL queries optimal ?
- Otherwise is there any better ?

DB bnp : br (br#, brname, street, street#, city, zipcode, tel)
DB sg : branch (bra#, braname, street, s#, town, zip, t#, class)
DB cic : br (br#, brname, street, street#, city, zipcode, tel)

Multirelational Algebra

- No, the natural decomposition is not optimal
  - the selection in **cic** is repeated three times uselessly
- It should be done once first, then one should proceed with the join
- One needs an algebra for multiple queries

Grant, Litwin, Selis, Roussopoulos. An Algebra and Calculus for Relational Multidatabases. The VLDB Journal, Vol. 2, No. 2, April 1993, 153-171.

Multirelational Algebra

- Multirelational operators
  - Select From M where (boolean condition)
  - Project M (A, B…)
  - Pair-wise Theta join On (M1.A θ M2.B AND …)
  - Theta Join On (M1.A θ M2.B AND …)
- These operators are typically commutative and associative as their relational counterparts
- Select can be moved through a join down the execution tree
- Project (C (Project M (A, B, C))) = Project M (C)
- Etc

Outline

- Motivation
- Multi database systems
- MSQL
- **SchemaSQL**
- Discussion: Data Management in web 2.0, wiki-dbs etc.

SchemaSQL: Criteria for the Language

- Expressive power independent from the schema
- Restructuring involving data and metadata
- Easy to use, yet sufficiently expressive
- Downward compatible with SQL syntax and semantics
- Efficient implementation
- Non-intrusive where extracting data from sources is concerned

Example

- Multi-database system consisting of floor salaries in various universities

univ-A:

salInfo

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
...	...	...

univ-B:

salInfo

category	cs	math	ece	...
Prof	72K	65K	78K	...
Assoc Prof	65K	54K	69K	...
...	...	...	...	...

Example

- Multi-database system consisting of floor salaries in various universities

univ-C:

cs

category	salFloor
Prof	74K
Assoc Prof	62K
...	...

math

category	salFloor
Prof	67K
Assoc Prof	56K
...	...

univ-D:

salInfo

dept	Prof	Assoc Prof	Asst Prof	...
cs	72K	65K	78K	...
math	65K	54K	69K	...
...	...	...	...	...

Expressive Power and Input Schema

Q1: What are the departments in *this* university?

- Expressible (in SQL) against univ-A and univ-D, but not against univ-B and univ-C
- Expressive power of most conventional query languages depends on input schema!

Attribute/Value Conflict

Q2: List the departments that have a salFloor exceeding 50K for Associate Profs in any university

Q3: List the pairs <dept, category> such that the salFloor for category in dept is more than 50K in both univ-B and univ-D

- "CS" – domain value in univ-A and univ-D, attribute in univ-B, and relation name in univ-C

- Manipulating both data and schema information is essential for achieving true interoperability

SchemaSQL

- Principled extension of SQL for multi-database interoperability and database restructuring
- Uniform manipulation of data and schema
- OLAP functionalities

Standard SQL : A SchemaSQL View

```
SELECT  name
FROM    employees
WHERE   dept = 'Marketing'
```

```
SELECT  emploees.name
FROM    employees
WHERE   employees.dept
        = 'Marketing'
```

```
SELECT  T.name
FROM    employees T
WHERE   T.dept = 'Marketing'
```

- Variables declared in the FROM clause range over **tuples**

Extending SQL Syntax for SchemaSQL

- Allow for distinction between the components of different databases
- Declaration of other types of variables in addition to tuple variables
- Generalized aggregate operations
- Dynamic output schema creation and restructuring
- Allow handling semantic heterogeneity by non-intrusive means

Kinds of Extensions

Kind of extension	SQL	SchemaSQL
Kinds of variable declarations allowed	only tuple variables	tuple, domain, attribute, relation and database variables
Conditions in WHERE clause	constraints on domain values	SQL's WHERE conditions plus conditions on schema variables
AS clause	only names as arguments	enhanced AS clause

- Resolving attribute/value conflict and other schematic discrepancies
  - Powerful and novel forms of aggregation
  - Enhanced class of variable declaration
- Dynamic (i.e., data dependant) output schema

Variable Declarations

FROM <range> <var>

Range Expression	Correspondence
➔	db names in the federation
db➔	relation names in db
db::rel➔	attribute names in rel in …
db::rel	tuples in rel in …
db::rel.attr	column with name attr in …

Range specifications in SchemaSQL can be nested

- Example: FROM db➔ X, db::X T

Schema Variables – Example 1

Fixed Output Schema

Q: List the departments in univ-A that pay a higher salary floor to their technicians compared with the same department in univ-B

univ-A:

salInfo

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
...	...	...

univ-B:

salInfo

category	cs	math	ece	...
Prof	72K	65K	78K	...
Assoc Prof	65K	54K	69K	...
...	...	...	...	...

Schema Variables – Example 1

Fixed Output Schema

Q: List the departments in univ-A that pay a higher salary floor to their technicians compared with the same department in univ-B

```
SELECT   A.dept
FROM     univ-A::salInfo A, univ-B::salInfo B,
         univ-B::salInfo→ AttB
WHERE    AttB <> 'category' AND
         A.dept = AttB AND
         A.category = 'technician' AND
         B.category = 'technician' AND
         A.salFloor > B.AttB
```

- A, B – variables that range over tuples
- AttB – variable that range over attributes in relation salInfo of univ-B

Schema Variables – Example 2

Q: List the departments in univ-C that pay a higher salary floor to their technicians compared with the same department in univ-D

univ-C:

cs

category	salFloor
Prof	74K
Assoc Prof	62K
...	...

math

category	salFloor
Prof	67K
Assoc Prof	56K
...	...

univ-D:

salInfo

dept	Prof	Assoc Prof	Asst Prof	...
cs	72K	65K	78K	...
math	65K	54K	69K	...
...	...	...	...	...

Schema Variables – Example 2

Q: List the departments in univ-C that pay a higher salary floor to their technicians compared with the same department in univ-D

```
SELECT   RelC
FROM     univ-C➔ RelC,
         univ-C::RelC C,
         univ-D::salInfo D
WHERE    RelC = D.dept AND
         C.category = 'technician' AND
         C.salFloor > D.technician
```

- RelC – ranges over relation names in database univ-C
- C – ranges over **all** tuples in univ-C
- D – ranges over tuples of salInfo in univ-D

Aggregation

Fixed Output Schema

Q: Compute the average salary floor of each category of employees over ALL departments

In univ-B: Horizontal Aggregation

univ-B: ***salInfo***

category	cs	math	ece	…
Prof	72K	65K	78K	…
Assoc Prof	65K	54K	69K	…
…	…	…	…	…

```
SELECT      T.category, avg(T.D)
FROM        univ-B::salInfo→ D,
            univ-B::salInfo T
WHERE       D <> 'category'
GROUP BY    T.category
```

Aggregation

Fixed Output Schema

Q: Compute the average salary floor of each category of employees over ALL departments

In univ-C: Aggregation over multiple relations

univ-C:

cs

category	salFloor
Prof	74K
Assoc Prof	62K
…	…

math

category	salFloor
Prof	67K
Assoc Prof	56K
…	…

```
SELECT     T.category, avg(T.salFloor)
FROM       univ-C→ D,
           univ-C::D T
GROUP BY   T.category
```

Dynamic Output Schema & Restructuring Views

Result of query/view in SQL

- Single relation

SchemaSQL

- Dynamic output schema
  - Width of relations
  - Number of relations
- Restructuring queries and views
- Interaction between these and aggregation

Restructuring Examples

View of database univ-B in the schema of univ-A:

univ-B: *salInfo*

category	cs	math	ece	…
Prof	72K	65K	78K	…
Assoc Prof	65K	54K	69K	…
…	…	…	…	…

univ-A: *salInfo*

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
…	…	…

```
CREATE VIEW BtoA::salInfo(category, dept, salFloor) AS
    SELECT  T.category, D AS dept, T.D AS salFloor
    FROM    univ-B::salInfo➔ D,
            univ-B::salInfo T
    WHERE   D <> 'category'
```

- Convert to ‘normalized form’
- One tuple in univ-B ➔ Many tuples in univ-A

Restructuring Examples

... and vice versa:

univ-A: *salInfo*

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
...	...	...

univ-B: *salInfo*

category	cs	math	ece	...
Prof	72K	65K	78K	...
Assoc Prof	65K	54K	69K	...
...	...	...	...	...

```
CREATE VIEW AtoB::salInfo(category, D) AS
    SELECT  A.category, A.salFloor
    FROM    univ-A::salInfo A,
            A.dept D
```

- Dynamic schema: number of columns depending on data
- Many tuples in univ-A ➔ One tuple in univ-B

Restructuring Examples

View of database univ-A in the schema of univ-C:

univ-A: *salInfo*

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
...	...	...

univ-C: *cs*

category	salFloor
Prof	74K
Assoc Prof	62K
...	...

math

category	salFloor
Prof	67K
Assoc Prof	56K
...	...

```
CREATE VIEW AtoC::D(category, salFloor) AS
    SELECT  (A.category, A.salFloor)
    FROM    univ-A::salInfo A,
            A.dept D
```

- Splits univ-A's relation into many relations

Restructuring Examples

View of database univ-C in the schema of univ-B:

univ-C: *cs*

category	salFloor
Prof	74K
Assoc Prof	62K
...	...

math

category	salFloor
Prof	67K
Assoc Prof	56K
...	...

univ-B: *salInfo*

category	cs	math	ece	...
Prof	72K	65K	78K	...
Assoc Prof	65K	54K	69K	...
...	...	...	...	...

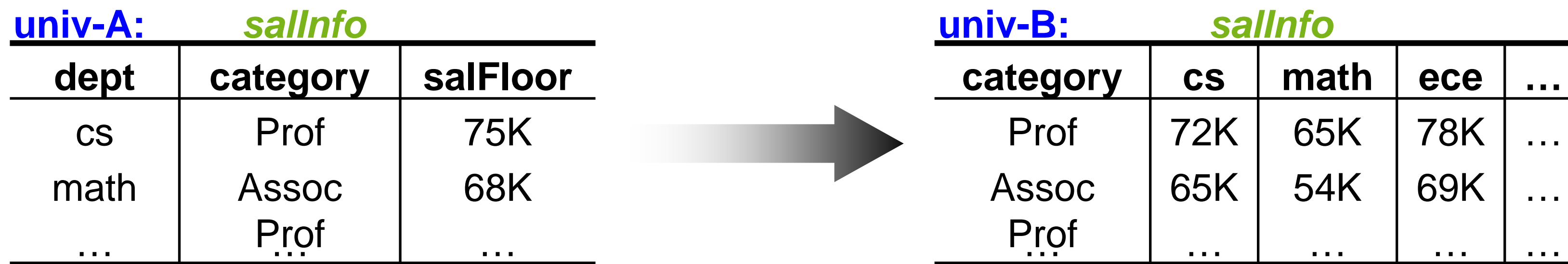
```
CREATE VIEW CtoB::salInfo(category, D) AS
       SELECT   T.category, T.salFloor
       FROM     univ-C➔ D,
                univ-C::D T
```

- Unites many relations into a single relational view

Summary of Restructuring Capabilities

Fundamental Operations:

- **Unfolding** a relation by an attribute on another attribute
  - Unfold salInfo by dept on salFloor



univ-A: *salInfo*

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
…	…	…

univ-B: *salInfo*

category	cs	math	ece	…
Prof	72K	65K	78K	…
Assoc Prof	65K	54K	69K	…
…	…	…	…	…

- **Folding** a relation on an attribute by another attribute
  - Fold salInfo on salFloor by dept

Summary of Restructuring Capabilities

Fundamental Operations:

- **Splitting** a relation by an attribute on another attribute
  - Split salInfo by dept on salFloor

univ-A: *salInfo*

dept	category	salFloor
cs	Prof	75K
math	Assoc Prof	68K
...	...	...

univ-C: *cs*

category	salFloor
Prof	74K
Assoc Prof	62K
...	...

math

category	salFloor
Prof	67K
Assoc Prof	56K
...	...

- **Uniting** a relation on an attribute by another attribute
  - Unite salInfo on salFloor by dept

Aggregation with Dynamic View Definition

- Assume a relation univ-D::faculty(dname, fname)
- Q: For each faculty in univ-D, compute the faculty-wide average salary floor of ALL employees (over all departments) in the faculty

```
SELECT      U.fname, avg(T.C)
FROM        univ-D::salInfo➔ C,
            univ-D::salInfo T,
            univ-D::faculty U
WHERE       C <> 'dept' AND
            T.dept = U.dname
GROUP BY    U.fname
```

- Block Aggregation

Aggregation with Dynamic View Definition

```
SELECT      U.fname, avg(T.C)
FROM  univ-D::salInfo→ C,
            univ-D::salInfo T,
            univ-D::faculty U
WHERE       C <> 'dept' AND
            T.dept = U.dname
GROUP BY    U.fname
```

intermediate-relation

faculty	dept	Prof	Asso	…
F1	CS	80K	75K	…
F1	Math	70K	60K	…
…	…	…	…	…

Aggregation with Dynamic View Definition

- Assume a relation univ-D::faculty(dname, fname)
- Q: For each faculty in univ-D, compute the faculty-wide average salary floor of EACH category of employees (over all departments) in the faculty

```
CREATE VIEW    averages::salInfo(faculty, C) AS
  SELECT       U.fname, avg(T.C) AS avgSal FOR C
  FROM         univ-D::salInfo➔ C,
               univ-D::salInfo T,
               univ-D::faculty U
  WHERE        C <> 'dept' AND
               T.dept = U.dname
  GROUP BY     U.fname
```

- Aggregation over dynamic number of columns

Outline

- Motivation
- Multi database systems
- MSQL
- SchemaSQL
- **Discussion: Data Management in web 2.0, wiki-dbs etc.**

Discussion