The logic of group operations on complex objects in RxO-system.

Evgeniy Grigoriev (C) 2011

Grigoriev.E @ gmail.com

Group operations (queries and methods) on complex objects of RxO-system [1] are described in the paper formally. The state of any object is presumed to be represented as set of relation values. This presumption makes possible the formal transformation of complex object data to set of relations. Operations on these relations are given which realize base principle [1] of representing complex object data in arbitrary O-views which are base for ad-hoc queries. The principle of formal translation as base of class method group execution is given. Conclusion is drawn that all described transformations can be implemented in object-oriented translator for programmable systems equipped with associative relational memory such as existing relational DBMSs.

Further discussion bases on relational data model [2,3,4] mainly. Next denotations of relational operations are used in the article

$R_1 \times R_2$- Cartesian product of R relations,

$R_1 \cup R_2$- union,

$R_1 - R_2$ - difference,

$R[a_1, a_2, \ldots]$ – projection (a_i is attribute),

R **WHERE** *criteria* - selection with criteria,

(LEFT) JOIN$_{condition}$ – join (common attributes of joined relations or join criteria operation are written as join condition).

R **RENAME** *a* **AS** *b* – attribute renaming. Let's note that RDM does not restrict complexity of names used for naming relations and their attributes. The only requirement is the name uniqueness in context.

> On object complexity.

Relational data model assumes that enterprise must be described as a set of relations which form relational database (the "enterprise" database). Any enterprise can be described as object; any object composing the enterprise can itself be considered as an enterprise. It means that any object can be described too as a set of relations forming relational database which is a subset of "enterprise" one. Thus complexity of any object is similar to complexity of relational database. Next formula is supposed as base one to unite properties of both objects and relations in single system:
"Object = relational DB".
Let's note that both sides of the formula allow nesting. The object can consist of many objects and according to this the relational database can be divided into many subsets being relational databases. Other links between object and relation concepts can be found according to the formula such as

"Object value = total DB value",
"Object structure = DB schema",
"Object component = relation",
"Class method= DB transaction", etc.

According to this, specification of RxO class **T** lists valued components which have one of next types

1) Scalar type (base or reference one). Scalar component definition can be treated as short recording of unary relation with single cardinaity.
2) SET type. The component of SET type is a set of tuples defined on scalar types. Keys can be defined for the sets. Thus the set component equals to relation.

Value of any class **T** object is described as set C_T of component C_1, C_2, … .

(<u>Remark</u>. It's possible to say that the state of any object can be described as set of values which are *not more complex* than relations. E.g. RxO class component could be also defined as typle (as recording of relations with single cardinality) or as collection (as recording of unary relations). Further we will not consider such possibilities because they are logically intervening between scalars and relations.)

Each object has unique object identifier (**OID**) defined in **dOID** domain. **OID** value is generated by system at object creation and stays unchanged during all object life.

> Relations which evidently follow from class structure.

For class **T**, all object values can be represented in several relations (further – class relations) in the following way

- Values of all scalar components $^{sc}C_i$ of all class **T** objects are represented in single relation R_T (further – class scalar relation)
  $(_1OID \times {}_1{}^{sc}C_1 \times {}_1{}^{sc}C_2 \times ... \times {}_1{}^{sc}C_n) \cup (_2OID \times {}_2{}^{sc}C_1 \times ... \times {}_2{}^{sc}C_n) \cup (...) \cup ... \text{ -> } R_T$
  here $_jOID$ is identifier of some object, $_j{}^{sc}C_i$ is some scalar component of this object, **n** is number of scalar components in class **T**

  One class scalar relation conforms to each class. One record of the scalar relation conforms to each object of the class. **OID** attribute is mandatory in the scalar relation. Other attributes of the scalar relation conform to class scalar components exactly.
  o **OID** is key field.
  o If scalar class components form class key, then corresponding attributes form scalar relation key.

- Values of each set component $^{set}C_i$ of all class **T** object are represented in relation $R_{T.seti}$ (further – class set relation).
  $(_1OID \times {}_1{}^{set}C_1) \cup (_2OID \times {}_2{}^{set}C_1) \cup ... \text{ -> } R_{T.set1}$
  $(_1OID \times {}_1{}^{set}C_2) \cup ... \text{ -> } R_{T.set2}$
  …
  $(_1OID \times {}_1{}^{set}C_m) \cup ... \text{ -> } R_{T.setm}$
  here $_jOID$ is identifier of some object, $_j{}^{set}C_i$ is some scalar component of this object, **m** is number of set components in class **T**.

  Number of set relation is equal to number of set components. **OID** attribute is mandatory in the set relation. Other attributes of the set relation conform to attributes of corresponding set component exactly.
  o Key of the set relation aggregates **OID** field and fields defined as corresponding component key ones.

o If attributes of set component form class key, then corresponding attributes form set relation key.

(Remark. Component keys are not mandatory in RxO system described in [1] because it is realized over SQL server which allows rows duplication in table. This realization feature is not important in current discussion.)

• At that all references (reference components and reference attributes of set components) are represented in class relations as attributes defined in domain of object identifiers **dOID**.

Thus all data about all class **T** objects are represented in one scalar relation $\mathbf{R_T}$ and several set relations $\mathbf{R_{T.seti}}$ which together form a set $\boldsymbol{R_T}$ of class relations. At that all constraints defined for class T are represented in constraints set on $\boldsymbol{R_T}$.

The set $\boldsymbol{E}$ of classes $\mathbf{T_i}$ which form full description of some enterprise is represented as union $\boldsymbol{D}$ of sets $\boldsymbol{R_{Ti}}$ containing all data about the enterprise.
$$\mathbf{T_i} \in \boldsymbol{E} \quad => \quad \boldsymbol{R_{Ti}} \subset \boldsymbol{D}$$

Class relations are axiom for further discussion.

(Remark on scalar components. Here all scalar components of class **T** were represented in single scalar relation $\mathbf{R_T}$. Let's note that projection $\mathbf{R_T[OID, \,^{sc}C_i]} \rightarrow \mathbf{R_{T.sci}}$ will return relation $\mathbf{R_{T.sci}}$ which conforming to scalar component $^{sc}C_i$ exactly. Such operation allows representation where any class component (both scalar and set one) is represented in own class relation. However class key formed of set of scalar components cannot be represented in class relations in this case. Further we will use this representation variant too.)

E.g. each object of class SHIPMENT

```
CREATE CLASS SHIPMENTS
{
  DocN STRING;
  Cntr CONTRACTORS;
  ...
  Items SET OF
  {
    Art STRING;
    Qty INTEGER;
  }KEY uniqArt (Art);
}KEY uniqDocN (DocN)
REFERENCE ToUniqArt
Items.(Art) ON GOODS.UniqArt;
```

is described by scalar components DocN, Cntr and set component Items with attributes (Art, Qty). Component DocN is a class key. Attribute Art is a local key of component Items. As it described upper all data about all class SHIPMENT objects are represented in two class relations. First relation is scalar one
R_{SHIPMENTS}(OID: dOID, DocN:String, Cntr:dOID)
where attribute **OID** is key and attribute **DocN** is other key.
Second one is set relation
R_{SHIPMENTS.Items}(OID: dOID, Art:Stirng, Qty:Integer)
where key consists of attributes **OID** and **Art**.

> Base principle.

Given in [1] the base principle defines semantic link between classes and relations (O-views) which represent values describing states of the classes objects.
Any non-terminal path can be treated as a name of existing O-view, some post-paths of this path can be treated as names of attributes of this view.
Here the path is name sequence determined by structures and references defined in class specification.

System is supposed to analyze the O-view signature got in command to check if it satisfies to the base principle and then to calculate O-view. Operations used in the O-view calculation are described below.

> Simple projection.

If O-view signature is subset of one of class relations R_i then simple projection $R_i[a]$ is used to calculate it.
For example O-view `SHIPMENTS.Items(.Art)` is calculated as projection $R_{SHIPMENTS}$[<u>DocN</u>, Cntr]. O-view `SHIPMENTS.Items(.Art)` is calculated as $R_{SHIPMENTS.Items}$[**Art**].

> In-class JOIN operation.

Suppose O-view combines data from different class relations of some class **T;** then result is calculated by **JOIN** operation using common OID attribute. Also attributes of class set relations are renamed to keep path semantics.
R_T **LEFT JOIN$_{OID}$ … ($R_{T.seti}$ RENAME a as set$_i$.a …) …**
Here **RENAME** operation specifies attribute name **a** with set component name **set$_i$**. New complex name **set$_i$.a** keeps full semantic of the attribute, given in class **T** specification. Also new complex name is certainly unique among other O-view attributes. **LEFT JOIN** is used to present objects in O-view even if their set components $R_{T.seti}$ contain no record.

For example O-view
`SHIPMENTS(.DocN, .Items.Art)`
is calculated as
($R_{SHIPMENTS}$ LEFT JOIN$_{OID}$ ($R_{SHIPMENTS.Items}$ RENAME Art AS <u>Items.Art</u>))[DocN, Items.Art]

> In-reference JOIN operation in O-view attributes.

Suppose O-view attribute contains reference inside; then **JOIN** operation is used to join referencing and referenced class relations. Also **RENAME** operation is used to keep full semantics of referenced attribute.
R_{refT} LEFT JOIN$_{ref = OID}$ (R_T RENAME a AS ref.a ….)
Here **R_{retT}(… , ref …)** is class relation containing reference field **ref**; **R_T(…, a …)** is referenced class relation. New complex name **ref.a** is certainly unique among other O-view attributes. **LEFT JOIN** is used to present objects in O-view even if their fields **ref** are empty.

E.g. class `SHIPMENTS` contains field `Cntr` referencing to class `CONTRACTORS`.

```
CREATE CLASS CONTRACTORS
{
  Name STRING;
  Bank BANKS;
  BankAcc STRING;
  INN STRING;
}KEY uniqINN (INN);
```

The class `CONTRACTORS` contains only scalar components so all data about the class object is represented as single scalar relation.
$R_{CONTRACTORS}$(OID:OID, Name:STRING, BANK: OID, BankAcc:STRING, INN:STRING)

In next O-view signature
`SHIPMENTS(.DocN, .Cntr.Name)`

first attribute is conformed to scalar component DocN of class SHIPMENTS represented in scalar relation **R_{SHIPMENTS}**, second attribute is conformed to scalar component Name of referenced with Cntr class CONTRACTORS represented in scalar relation **R_{CONTRACTORS}**.

SHIPMENTS(.DocN, .Cntr `-> CONTRACTORS`.Name)

This O-view is calculated as
(
R_{SHIPMENTS} JOIN_{Cntr=OID} (**R_{CONTRACTORS} RENAME Name AS Cntr.Name**)
)[**DocN, Cntr.Name**]

New complex name **Cntr.Name** keeps semantic of reference expression as it given in O-view signature.

This operation can be nested. For example next O-view presented data of three classes SHIPMENTS, CONTRACTORS and BANKS
SHIPMENTS(.DocN, .Cntr.Name, .Cntr.Bank.Name)
is calcilated as (nested operation is underlined)

(**R_{SHIPMENTS} JOIN_{Cntr=OID} (**
 (R_{CONTRACTORS} JOIN_{Bank=OID} (R_{BANKS} RENAME Name AS Bank.Name))
 [OID_{Shipments}, Name, Bank.Name])
RENAME Name AS Cntr.Name, Bank.Name AS Cntr.Bank.Name)
[DocN, Cntr.Name, Cntr.Bank.Name]

So, nested construction can have any length in O-view attributes.

> In-reference JOIN operation in O-view headers.

Path SHIPMENTS.Cntr can be treated as recording of simple O-view
SHIPMENTS(.Cntr)
which is calculated as
R_{SHIPMENTS}[Cntr]
The calculation result is unary relation containing set of **OID** of class CONTRACTORS objects which are referenced by existing class SHIPMENTS objects. Let's name such relations as group references. Expressions to generate the group reference will be named as *group reference expression* further. Any path ended with reference is group reference expression. So, SHIPMENTS.Cntr is the one.

O-view
SHIPMENTS.Cntr(.Name, .INN)
presents data of class CONTRACTORS objects which are referenced with existing class SHIPMENTS objects.
SHIPMENTS.Cntr `->  CONTRACTORS`(.Name, .INN)
This O-view is calculated as
((**R_{SHIPMENTS}[Cntr]**) **JOIN_{Cntr=OID} R_{CONTRACTORS}**)[**Name, INN**]

This operation can be nested. For example next O-view presented data of three classes SHIPMENTS, CONTRACTORS and BANKS
SHIPMENTS.Cntr.Bank(.Name, BIC)
is calcilated as (nested operation is underlined)
 ((((**R_{SHIPMENTS}[Cntr]**) **JOIN_{Cntr=OID} R_{CONTRACTORS}**)[**Bank**]) **JOIN_{Bank=OID} R_{BANKS}**) [**Name, BIC**]

So, nested construction can have any length in O-view headers.

> Object selection expression.

Group reference expression can be added with object selection expression.
...name_of_class_or_reference[*condition_list*]*...*
Its result is group reference to the objects referenced by *name_of_class_or_reference*, which satisfy to *condition_list*.

Each of conditions listed in *condition_list* is the one available in WHERE operation.

Next expression
```
SHIPMENTS[.DocN = "N01"]
```
references on class `SHIPMENTS` objects which component `DocN` contains value `"N01"` in. As soon as the component `DocN` is key one in the class this expression references to the single object (if the one exists). The selection condition `.DocN = "N01"` is calculated as
($R_{SHIPMENTS}$WHERE <u>DocN = "N01"</u>) [OID]
The calculation result is unary relation containing OID of objects satisfying to given condition. This relation restricts the object set defined with *name_of_class_or_reference* part using **JOIN** operation. For example O-view
```
SHIPMENTS[.DocN = "N01"](.Items.Art = "Tie")
```
is calculated as (selection part is underlined)
((<u>$R_{SHIPMENTS}$WHERE DocN = "N01"</u>) [OID] JOIN$_{OID}$ ($R_{SHIPMENTS}$ RENAME Art AS Items.Art) [Items.Art]

Object selection expression can include a number of conditions separated by commas denoting new low-level logical operation which can be named as "intertuples_AND" . Next path
```
SHIPMENTS[.Items.Art = "Tie", .Items.Art = "Axe"]
```
defines group reference to class `SHIPMENTS` objects which contain in rows of set component `Items` both values `"Tie"` and `"Axe"` in attribute `Art`.

The necessity of this operation is dictated by the fact that class components are relations containing set of tuples, so selection cases inexpressible with usual in-tuple logical operation are possible. The example of such case is given above. The result of "intertuples_AND" operation is **INTERCEPT** of unary relations containing OID of objects satisfying to each of conditions separated by the comma. The group reference expression given above is calculated as

**((R$_{SHIPMENTS.Items}$ RENAME Art AS Items.Art) WHERE <u>Items.Art = "Tie"</u>) [OID]
INTERCEPT
((R$_{SHIPMENTS.Items}$ RENAME Art AS Items.Art) WHERE <u>Items.Art = "Axe"</u>) [OID]**

Described operations realize base principle which defines semantical link between classes and relations (O-views) which represent values describing states of the classes' objects. Let's attend that the operations use classes' relations **R** as base operands.

> Translation statement.

RxO system does not use iterators in operations on group of objects. The same is true for procedures which realize calculated components and methods. Next expression
```
EXEC someClass[condition].p();
```
is executed in two steps
    1) Group reference expression `someClass[condition]` is calculated.

2) Procedure p' is executed taking the group reference found at previous step as parameter. Its single execution changes the system just as source method p was executed in each of objects referenced by the group reference.

The principles of translation of p to p' is given below.

Let's illustrate the idea of such translation with simple example. Values of two scalar components a and b are summed in realization procedure.
```
a + b
```

This operation can be applied to the same values represented in terms of class relations R as
(**R**$_{scalar}$ **WHERE OID = someOID)[a+b]**
Here **someOID** is identifier of some object which the operation is executed for. It's easy to see that the operation can be executed for group of objects referenced by group reference **these** as
(**R**$_{scalar}$ **JOIN**$_{OID}$ **these)[OID, a+b]**
The result contains sums of a and b components in all objects from group referenced by **these**. The structure of resulted relation is same as the one of class relations, so it can be used in calculations together with the class relations.

Let's prove that <u>any procedure p on components C of class **T** can be translated in such procedure p' on the class relation R_T that result of single execution of p' is equal to execution of p in each object from given set of object</u> (translation statement).

Further the set *THESE* of class **T** objects is given by group reference relation **these** containing single attribute **OID**.

Let's note that logic of representing component C values in form of class relation R can be also applied to parameters and to local variables of procedures. Parameters **par**$_1$, **par**$_2$, ... of procedure p can be represented as relation **R**$_{par}$**(OID, par**$_1$**,par**$_2$**,…)** for set *THESE* of objects which the procedure executed in.
R$_{par}$ **[OID] -> these**
The parameter relation **R**$_{par}$ is single parameter of procedure p'. It must be created before execution of p'.

Local variable can be represented in relations **R**$_{local}$ in the same way. They must be created at the beginning of p' execution.

The parameters and local variables of procedure can be treated as unpersistent class components with different lifetime. But anyway they are certainly defined inside procedure as well as other class components. So, further they will be considered as elements of set C of class components. Accordingly parameters relation **R**$_{par}$ and local variables relations **R**$_{local}$ will be considered as elements of set R of class relations.

> Operation translation.

Let's consider operation
$f(C_1, C_2, …)$ -> C_n (1)
C_n is a result of operation.

Here f is superposition of primitive relational operation $_{prim}op$
$_{prim}op_1(C_1, {}_{prim}op_2(C_2, … (…)))$

on set of class **T** components $C_1, C_2, \ldots$. Each $_{prim}op$ can contain scalar operations on attributes of operands.

With remark on scalar components, any scalar component C_i conforms to class relation R_i. Component C_i value is calculated as

  $(R_i$ WHERE OID $=$ theOID$)[!OID]$ –> C_i

Here **$[!OID]$** is projection operation which excludes attribute OID existing in any class relation **R**.

For each of primitive relational operations $_{prim}op$ the next is true.
- Union $C_1 \cup C_2$ is equal to **$((R_1 \cup R_2)$ WHERE OID $=$ theOID$)[!OID]$**
- Difference $C_1 - C_2$ is equal to **$((R_1 - R_2)$ WHERE OID $=$ theOID$)[!OID]$**
- Cartesian product $C_1 \times C_2$ is equal to **$((R_1$ JOIN$_{OID}$ R_2 WHERE OID $=$ theOID$)[!OID]$**
- Selection **C WHERE condition** is equal to **$((R$ WHERE condition$)$ WHERE OID $=$ theOID$)[!OID]$**
- Projection **C[a1, a2, ...]** is equal to **$((R[OID, a1, a2, ...])$ WHERE OID $=$ theOID$)$ $[!OID]$**

At that, all possible in $_{prim}op$ scalar operations on components **C** attributes are applied to corresponding relations **R** attributes without changes.

So, all primitive operations

  $_{prim}op(C_1 \ldots)$ **-> C_{res}**   (C_{res} is result of the operation)

is deduced to expression

  $(op'$ $(R_1 \ldots)$ WHERE OID $=$ theOID$)$ $[!OID]$ -> C_{res}   (2)

Let's note that logic of representing component **C** values in form of common class relation **R** can be applied to values C_{res}, so result relation R_{res} exists which unites results of $_{prim}op(C_1 \ldots)$ executed for all class T objects.

  $(R_{res}$ WHERE OID $=$ theOID$)[!OID]$ –> C_{res}   (3)

Comparison of (2) and (3) gives that any primitive operation $_{prim}op$ on **C** can be deduces to operation op' on **R** which produces result relation R_{res}.

  $op'(R_1 \ldots)$ -> R_{res}

Relational algebra closure means that any result relation R_{res} can be used as operand of other operation

  $op'(\ldots R_{res} \ldots)$ -> R_{res+1}

Thus, for any operation *f* on **C**

  $f(C_1, C_2, \ldots)$ -> C_n

such operation *f'* on **R** can be found

  $f'(R_1, R_2, \ldots)$ -> R_n

that its result R_n unites all results C_n of source expression *f* executed in each of class **T** objects

  $(R_n$ WHERE OID $=$ theOID$)[!OID]$ –> C_n

Let's name *f'* as translation of source operation *f*.

Result of operation

  $f'(R_1, R_2, \ldots)$ JOIN$_{OID}$ these -> theseR$_n$

is relation **theseR$_n$** which unites result of source operation *f* executed in each object of given set *THESE*.

> Procedure translation.

Let's consider procedure p as algorithmic sequence of assignment operations

set $C_n = f(C_1, C_2, …)$

and procedure calls

call proc(par_1, par_2, …)

(here **proc** – procedure name, **par_i** – procedure parameters)

Assigment operation

set $C_n = f(C_1, C_2, …)$

is translated in

set $R_n =$
$f'(R_1, R_2, …)$ JOIN$_{OID}$ these
UNION
(R_n WHERE JOIN$_{OID}$ (R_n[OID] MINUS these))

Here
- part $f'(R_1, R_2, …)$ **JOIN$_{OID}$ these** is result of translation f' execution in objects of set **THESE**
- part **(R_n WHERE JOIN$_{OID}$ (R_n[OID] MINUS these))** is unchanged part of R_n, which unites component C_n of class **T** objects not included in **THESE**.
Per se this translation updates relation R_n by means of replacing tuples containing data of set **THESE** objects with new ones. Further it is written as

update R_n with ($f'(R_1, R_2, …)$ JOIN these)

Procedure call

call proc(par_1, par_2, …)

is translated in its translation call

call proc'(R_{par})

with parameter relation **R_{par}(OID, par_1,par_2,…)** as single parameter.

Linear sequence of operations

set $C_n = f_1(C_1, C_2, …)$
call proc(par_1, par_2, …)
set $C_{n+1} = f_2(C_1, C_2, …)$
…

is translated in the same sequence of their translations

update R_n with ($f'_1(R_1, R_2, …)$ JOIN these)
call proc'(R_{par})
update R_{n+1} with ($f'_2(R_1, R_2, …)$ JOIN these)
…

Translation of canonical algorithm structures **if…** and **while…** uses a condition given in the structures to find subset of set *THESE*.

Algorithm structure

if(condition) then set $C_n = f(C_1, C_2, …)$

is translated in

set theseTrue = these JOIN$_{OID}$ (R_i…WHERE condition)[OID]
update R_n with ($f'(R_1, R_2, …)$ JOIN theseTrue)

Here group reference **theseTrue** contains identifiers **OID** of that objects from **THESE** which satisfy to **condition** given in source code. **R_i…**is guessed to contain attributes used in condition calculation.

Algorithm structure **while** is translated in the same way. At that, the circle will be continuing while objects satisfying to condition exist. Next code

while condition
begin
   …
   set $C_n = f(C_1, C_2, …)$
   …
end
is translated in
set theseTrue = these JOIN$_{OID}$ (R_i…WHERE condition)[OID]
while COUNT(theseTrue)>0
begin
   …
   update R_n with ($f'(R_1, R_2, …)$ JOIN theseTrue)
   …
   set theseTrue = these JOIN$_{OID}$ (R_i…WHERE condition')[OID]
end
Here **COUNT(R)** is operation finding a number of relation **R** typles (i.e. cardinality of **R**).

Thus, any procedure p on components C of class **T** can be translated in such procedure p' on the class relations R_T that result of single execution of p' for given set of object is equal to execution of p in each object of the set. Let's attend that the translation executed p' on classes' relations **R**.

> Conclusion.

It was shown that set E of classes' T objects satisfying to formula "Object = Relational DB" can be represented in set D of class relations **R**. Both schemas and constrains of the relations are uniquely defined by the classes' structures and constrains. According to this, language expressions describing classes T can be uniquely translated into the ones describing class relations **R**.

The operations realizing the base principle were given, which links sematically the set E of classes T and set of O-view representing total enterprise data in from of arbitrary relations. The principle is a base for ad-hoc queries on data described in terms of complex classes. Class relations **R** are base operands for these operations.

Translation statement was proved, which bases ability of group execution of procedures p in given set of class **T** objects. The translation p' is procedure executed on the same class relations **R**.

So, the system able to define complex classes, to operate on groups of the classes objects and to query data about these objects must consist of two parts.
1) Translator part transforms incoming language expression applicable to the classes T into output commands applicable to relations **R** in accordance to principles describing upper.
2) Executive part maintains and manipulates both the relations **R** and procedures p' applicable to these relations. This part can be defined as programmable machine equipped with associative memory described by relational data model.
Existing relational DBMSs can be used as such machine without any changes. In this case the translator is an independent program generating SQL expressions executed by RDBMS

But SQL itself can be extended with new object-oriented expressions directly translated in sequences of DBMS kernel system calls inside the DBMS. This way is much more interesting for practical use. It makes possible evolution of existing relational DBMSs themselves towards the client-independent environment allowing both creation and persist existence of manageable object model of an enterprise.

References:

[1] Evgeniy Grigoriev (C) "RxO system. Simple semantic approach for representation complex objects data in table form." http://odbms.org/download/RxO.pdf 2011

[2] Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." CACM 13(6), June 1970. Republished in Milestones of Research -- Selected Papers 1958-1982 (CACM 25th Anniversary Issue), CACM 26(1), January 1983.

[3] David Mayer "The theory of Relational Database." Computer science press 1983

[4] C.J.Date "An introduction to Database Systems. 6th edition" Addison-Wesley Pub. Co. 1995