# Relations

## Relations

A *relation* is an ordered set of tuples. A *binary relation* is a set of ordered pairs.

| | |
|---|---|
| *Domain:* | the set of first elements in the ordered pair |
| *Range:* | the set of second elements in the ordered pair |
| *Cartesian Product:* | the set of all possible ordered pairs (domain X range) |
| *Empty Relation:* | the set of no ordered pairs |
| *Inverse:* | the relation formed by exchanging the range and the domain |
| *Relation on a Set:* | the domain and the range are the same set |

| | |
|---|---|
| *Identity Relation:* | `(x,x) \| x inR` |
| *Equivalence of ordered pairs:* | `(a,b) = (c,d)   iff   a = c   and   b = d` |

## Relations as Graphs

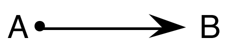
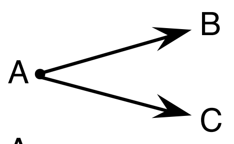
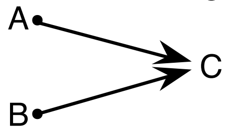
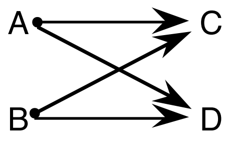
Relations are a way of pointing from one set to another. Relations establish directional pointers between elements of the *domain* and elements of the *range*. Thus, every relation is isomorphic to a directed graph with elements as nodes and concrete relations as arcs.

### *Types of Existence*

Some elements in the domain do not map onto a element in the range.
Some elements in the range do not correspond to an element in the domain.

### *Types of Connectivity*

Relations support any type of connectivity
between elements in the domain and those in the range.

| *GRAPH* | *NAME* | *Example* |
|---|---|---|
|  | *one-to-one* | `{(a,1),(b,2),(c,3)}` |
|  | *one-to-many* | `{(a,1),(a,2),(b,3)}` |
|  | *many-to-one* | `{(a,1),(b,1),(c,2)}` |
|  | *many-to-many* | `{(a,1),(a,2),(b,1),(c,2)}` |

## Types of Relational Structure

### *Relations on a Set*

| | |
|---|---|
| *reflexive* | `all x | (x,x) inR` |
| *symmetric* | `if (x,y) inR, then (y,x) inR` |
| *transitive* | `if (x,y) inR and (y,z) inR, then (x,z) inR` |
| *antisymmetric* | `if (x,y) inR and (y,x) inR, then x=y` |
| *trichotomy* | `(x,y) inR xor (y,x) inR  xor x=y` |
| *irreflexive* | `not reflexive` |
| *asymmetric* | `not symmetric` |

### *Functions*

| | |
|---|---|
| *identity* | `Id op A = A op Id = A` |
| *inverse* | `A op iA = iA op A = Id` |
| *associative* | `(A op B) op C = A op (B op C)` |
| *commutative* | `A op B = B op A` |
| *distributive* | `A op1 (B op2 C) = (A op1 B) op2 (A op1 C)` |
| *idempotent* | `A op A = A` |

## Equivalences

An *equivalence set* is a relation which is

| | |
|---|---|
| *reflexive* | `xRx` |
| *symmetric* | `xRy -> yRx` |
| *transitive* | `xRy and yRz -> xRz` |

## Partitions

A *partition* of a set (or a relation) is a collection of disjoint subsets of the set. The union of partitions is the entire set.

The *equivalence relation* determines a partition, and each partition of a set defines an equivalence relation.

## Orderings

A *partial order* is a relation which is

| | |
|---|---|
| *reflexive* | `xRx` |
| *antisymmetric* | `xRy and yRx -> x=y` |
| *transitive* | `xRy and yRz -> xRz` |

A *total order* is a relation which is

| | |
|---|---|
| *trichotomous* | `xRy xor x=y xor yRx` |
| *transitive* | `xRy and yRz -> xRz` |

## Cartesian Product

A relation is between two sets. The Cartesian Product of two sets is the set of ordered pairs consisting of all possible combinations of elements from each set. *Example:*

```
S1 = {1,2,3}            S2 = {a,b}

S1xS2 = {(1,a),(2,a),(3,a),(1,b),(2,b),(3,b)}
```

The set of *all possible relations* between S1 and S2 is defined by all the possible combinations of the product elements. In the example above, there are six product pairs, so the total number of possible relations is

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 things taken | 0 | at a time | = | 1 | | |
| “ | 1 | “ | = | 6 | | |
| “ | 2 | “ | = | 15 | | |
| “ | 3 | “ | = | 20 | | |
| “ | 4 | “ | = | 15 | | |
| “ | 5 | “ | = | 6 | | |
| “ | 6 | “ | = | 1 | | |
| Total number of relations | | | = | 64 | = | 2^6 |

There are two ways of analyzing the structure of relations illustrated above.

1. Using the coefficients of the binomial expansion

```
sum[i choose n] for i=0..n
```

2. Using the power set of the relational pairs

```
2^n
```

## Ways to View Relations

I. A set of *ordered pairs*: `{(a,b),(a,c),(b,d)…}`

II. A *set of points* matching elements in the Domain set with elements in the Range set.

```
      |
      |  *
Range |        *
      |    *
      |___________
          Domain
```

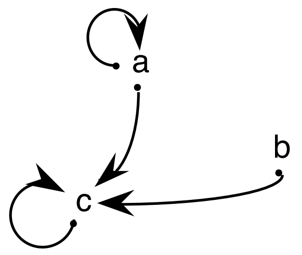
III. A *lookup table* between two sets

| R | a | b | c |
|---|---|---|---|
| a | x |   | x |
| b |   | x |   |
| c |   |   | x |

IV. A *matrix*:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

V. A *connection graph*:



VI. A *relational database*:

| | | |
|---|---|---|
| R | a | a |
| R | a | c |
| R | b | b |
| R | c | c |

VII. A *link table* (a possibility database):

| A | B | C | n |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

## Relations on a Set

When both the Domain and the Range of a relation are the same Set, the relation is *on a set*.

The lookup table and matrix representations can contain *common patterns*, which define the concepts associated with relations.

```
                     _R_|_a_b_c_
REFLEXIVE             a | x
                      b |   x
                      c |     x

                     _R_|_a_b_c_
SYMMETRIC             a |   x
                      b | x
                      c |     x

                     _R_|_a_b_c_
TRANSITIVE            a |   x
                      b |     x
                      c | x

                     _R_|_a_b_c_
ANTISYMMETRIC         a |   x
                      b | x        iff  a = b
                      c |
```

## Composition of Relations

Relational composition is very similar to functional composition.

```
(R o S) =def=  all pairs (x,z) exists y | (x,y) inS and (y,z) inR
```

Note that the range of S is a subset of the domain of R

```
(R o S)(A) = R(S(A))
```

*associative:* `(R o S) o T = R o (S o T)`

*not commutative:* `R o S =/= S o R`

*inverse of a composition:* `(R o S)^-1 = S^-1 o R^-1`

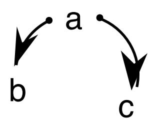
## Transitive Closure

A relation is *transitive* when it is possible to follow relations for one pair to another in a cycle.

The transitive closure is computed by multiplying the matrix of a relation by itself N times. A matrix will return to its original configuration after N multiplies if there is transitive path of N steps between the elements.

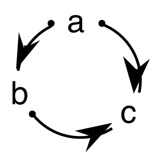
Example: non-transitive, no path between `b` and `c`

```
  a b c
a 0 1 1     0 1 1     0 0 0
b 0 0 0  *  0 0 0  =  0 0 0       no paths
c 0 0 0     0 0 0     0 0 0
```



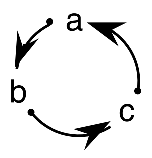
Example: non-transitive, no path to `a`

```
               A^1       A^2                  A^3
  a b c
a 0 1 1     0 1 1     0 0 1       0 1 1     0 0 0
b 0 0 1  *  0 0 1  =  0 0 0    *  0 0 1  =  0 0 0
c 0 0 0     0 0 0     0 0 0       0 0 0     0 0 0
```



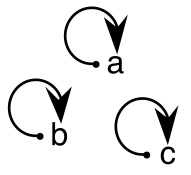
Example: transitive with cycle length 3

```
               A^1       A^2                  A^3                  A^4
  a b c
a 0 1 0     0 1 0     0 0 1       0 1 0     1 0 0       0 1 0     0 1 0
b 0 0 1  *  0 0 1  =  1 0 0    *  0 0 1  =  0 1 0    *  0 0 1  =  0 0 1
c 1 0 0     1 0 0     0 1 0       1 0 0     0 0 1       1 0 0     1 0 0
```



Example: identity, degenerate transitive with cycle length 1

```
               A^1       A^2
  a b c
a 1 0 0     1 0 0     1 0 0
b 0 1 0  *  0 1 0  =  0 1 0
c 0 0 1     0 0 1     0 0 1
```



## Relations or Functions?

Functions are a subset of relations. Using functions provides the advantages that functions place on relational structures: *existence* and *uniqueness*. However, since functions are less general, these same constraints make generalization impossible. Thus functional encoding is often brittle and difficult to modify. For example:

Domain D = students `{s1,s2,s3}`

Range R = chairs `{c1,c2,c3,c4,c5}`

When every student sits in a chair, the uniqueness (no student sits in two chairs) and existence (a chair for every student) constraints on a function are met.

F[students] = `{(s1,c1),(s2,c2),(s3,c5)}`

Note that it is still permitted for two students to sit in one chair.

If the situation changes, the functional constraints may be violated. For example, if one student lounges across two chairs, uniqueness is violated. If some student stands rather than sits in a chair, then existence is violated.

Most generally, whenever a specification may change (almost always the case), more general data structures achieve better code modularity, portability, and maintenance.

Many concepts do not permit a functional approach. For example, consider:

Proof-of[theorem] = <proof-sequence>

Domain = logical formulas as theorems
Range = proof-sequences

The above assertion has *relational semantics*. There is no assurance that a proof does exist for any formula (existence violated), and certainly there are many different proof sequences for any formula (uniqueness violated). In contrast:

Theorem-of[proof-sequence] = <theorem>

Domain = proof-sequences
Range = theorems as formulas

This assertion is (or can be) functional, since any proof leads to a unique theorem, and every proof-sequence leads to some theorem. The assertion, of course, can also be seen as relational.