

Relational Data Model and ER-Relational Mapping

Relational Data Model

1/70

The *relational data model* describes the world as:

- a collection of inter-related *relations* (or *tables*)

Goal of relational model:

- a simple, general data modelling formalism
- which maps easily to file structures (i.e. implementable)

Can be viewed as an attempt to formalise the file organisations that were in common use at the time the model was developed.

... Relational Data Model

2/70

The relational data model has existed for over 30 years.

(The original description is Codd, *Communications of the ACM*, 13(6), 1970)

The relational model has provided the basis for:

- research on the theory of data/relationships/constraints
- numerous database design methodologies
- the standard database access language SQL
- almost all modern commercial database management systems

It is a very influential development in CS, for which Codd received a Turing award.

Terminology

3/70

A note on the terminology used in the relational model ...

The relational model is a mathematical theory; it has no "standard".

However, it also has a close connection to file/data structures.

There are thus two kinds of terminology in use:

- mathematical: relation, tuple, attribute, ...
- data-oriented: table, record, field/column, ...

Textbooks alternate between the two; treat them as synonyms.

Relations

4/70

The relational model has one structuring mechanism ...

- a *relation* corresponds to a mathematical "relation"
- a *relation* can also be viewed as a "table"

Each relation schema (denoted R, S, T, \dots) has:

- a *name* (unique within a given database)
- a set of *attributes* (which can be viewed as column headings)

Each *attribute* (denoted A, B, \dots or a_1, a_2, \dots) has:

- a *name* (unique within a given relation)
 - an associated *domain* (set of allowed values)
-

... Relations

5/70

Attribute values are *atomic* (no composite or multi-valued attributes).

Attribute domains are typically: numbers, strings, booleans.

There is a distinguished value NULL that belongs to all domains.

A *database* is a collection of associated relations.

... Relations

6/70

Example relation (bank accounts):

branchName	accountNo	balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

Example Database

7/70

Account

branchName	accountNo	balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700

Branch

branchName	address	assets
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

Customer

name	address	customerNo	homeBranch
Smith	Rye	1234567	Mianus
Jones	Palo Alto	9876543	Redwood
Smith	Brooklyn	1313131	Downtown
Curry	Rye	1111111	Mianus

Depositor

account	customer
A-101	1313131
A-215	1111111
A-102	1313131
A-305	1234567
A-201	9876543
A-222	1111111
A-102	1234567

... Example Database

8/70

A *tuple* is a set of values; a relation is a set of tuples.

Since a relation is a set, there is no ordering on rows.

Normally, we define a standard ordering on components of a tuple.

The following are different presentations of the same relation:

branchName	accountNo	balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Redwood	A-222	700

accountNo	branchName	balance
A-305	Round Hill	350
A-222	Redwood	700
A-215	Mianus	700
A-102	Perryridge	400
A-101	Downtown	500

... Example Database

9/70

Consider a relation R :

- which has n attributes a_1, a_2, \dots, a_n
- with corresponding domains D_1, D_2, \dots, D_n

$R(a_1, a_2, \dots, a_n)$ (alternatively, $D_1 \times D_2 \times \dots \times D_n$)

- is a *schema* for the relation (intensional)

A particular subset r of $D_1 \times D_2 \times \dots \times D_n$

- is an *instance* of the schema (extensional)

... Example Database

10/70

Schema names are typically unique within a given database.

So, we often use R as a synonym for $R(a_1, a_2, \dots, a_n)$.

$r(R)$ is used to denote that r is an instance of the schema R .

The number of attributes (n) in a schema is its *degree* (arity).

Note: the phrase " the relation R " can refer to either

- the schema for R or
- the current instance of R stored in a DBMS

The intended usage is generally clear from the context.

... Example Database

11/70

E.g. the Accounts schema has type $String \times String \times Int$:

Account(branchName, accountNo, balance)

E.g. the Account instance (set of tuples) from the diagram:

```
{
  (Downtown, A-101, 500), (Mianus, A-215, 700),
  (Perryridge, A-102, 400), (Round Hill, A-305, 350),
  (Brighton, A-201, 900), (Redwood, A-222, 700),
  (Brighton, A-217, 750)
}
```

Notes:

- values in tuples are comma-separated, so we don't normally quote strings
- choose an order for attributes/values in tuples and use that consistently
- relations are sets \Rightarrow no duplicates, order of tuples is not important

Changing Relations

12/70

In making changes to relations, it is ...

- easy to add new tuples (rows) (relation update)
- difficult to add new attributes (columns) (schema update)

The reasons:

- relation update \Rightarrow insertion of one new tuple into a set
(in file terms: writing one record to the end of a data file)
- schema update \Rightarrow insertion of new data into every tuple
(in file terms: re-writing the entire file to modify each record)

Schema update is a well-known and not well-solved problem in RDBMSs.

Constraints on Relational Data

13/70

Constraints

14/70

A very important feature of the relational model:

- well-defined theory of constraints on attributes/tables

This is useful because it allows

- formal reasoning about databases and operations on them
- designers to specify precisely the semantics of the data
- DBMSs to check that new data satisfies the semantics

Integrity Constraints

15/70

Relations allow us to represent data and associations.

Domains limit the values that attributes can take.

However, to fully represent the semantics of real-world problems, we need more detailed ways of specifying

- what values are/are not allowed
- what combinations of values are/are not allowed

Integrity constraints are logical statements about data that provide such information.

Some examples:

- employees must be over 16 and under 65 years of age
- account numbers must be unique
- each account is held at one particular branch

... Integrity Constraints

16/70

Several kinds of constraints exist e.g.

<i>key</i>	combination of attributes must be unique
<i>entity integrity</i>	no attribute in key may be NULL
<i>referential integrity</i>	references to tuples in other relations must be valid
<i>domain</i>	value of attribute must satisfy certain property

Functional dependencies are another important kind of constraint, related to database design; we cover them in considerable detail later.

... Integrity Constraints

17/70

Associating an attribute to a domain restricts its possible values to a well-defined set (e.g. integer).

Domain constraints allow more "fine-grained" definition of potential attribute values.

Example:

An age attribute is typically defined as integer ...

- but integer values like -5 and 199 are not valid ages
- better modelled by adding a condition ($15 < \text{age} < 66$)

Note: the NULL value satisfies any domain constraint.

... Integrity Constraints

18/70

Relational tuples have no notion of identity like OIDs.

Identity is value-based (as in ER model)

- *keys* are a way of uniquely identifying tuples.

Relational model supports same notions of key as ER model:

- superkey - set of attributes that distinguishes tuples
- candidate key - minimal super-key (no unnecessary attributes)
- primary key - distinguished/chosen candidate key

Keys are often implemented by introducing an artificial attribute specifically for the purpose of being a key (e.g. student ID, account number).

... Integrity Constraints

19/70

Referential integrity constraints are relevant for inter-relation references.

Example:

- the Account relation needs to take note of the branch where each account is held
- implemented by storing information in each Account tuple to identify the associated branch (e.g. primary key branchName)
- it would not make sense to store a branchName that did not refer to one of the existing branches

The notion that the branchName must *refer* to a valid branch is a referential integrity constraint.

Foreign Keys

20/70

Referential integrity is related to the notion of a *foreign key*.

A set of attributes *FK* from a relation schema R_1 is a foreign key if:

- the attributes in *FK* correspond to the attributes in the primary key of another relation schema R_2
 - the value for *FK* in each tuple of R_1
 - either occurs as a primary key in R_2
 - or is entirely NULL
-

... Foreign Keys

21/70

Foreign keys are critical in relational databases

- they provide the "glue" that links individual relations into a cohesive database structure
- they provide the basis for "reconnecting" individual relations to assemble query answers

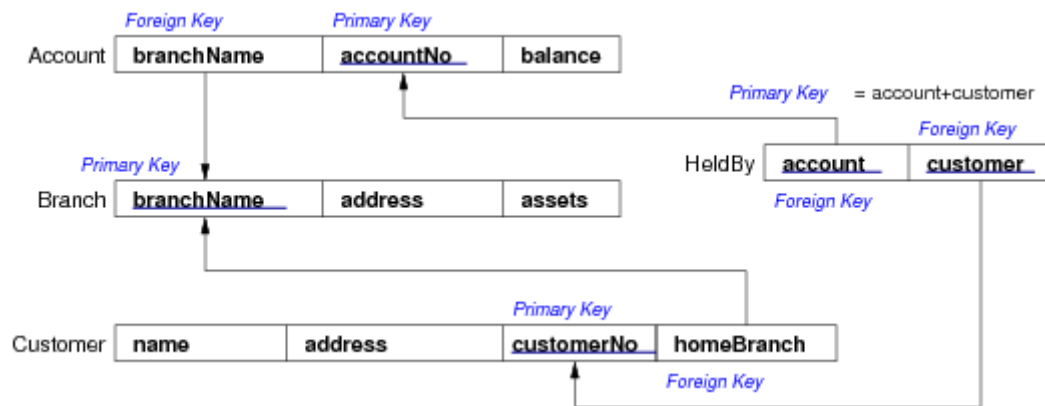
Special notation for foreign/primary keys:

- each relation is a sequence of "attribute boxes"
 - attributes that are part of primary key are underlined
 - arrows are drawn from foreign key attributes to their corresponding primary key attributes
-

... Foreign Keys

22/70

Foreign key examples:



Relational Databases

23/70

Relations, keys, foreign keys, and integrity constraints provide a complete toolkit for building relational databases.

A *relational database schema* is

- a set of relation schemas $\{R_1, R_2, \dots, R_n\}$, and
- a set of integrity constraints

A *relational database instance* is

- a set of relation instances $\{r_1(R_1), r_2(R_2), \dots, r_n(R_n)\}$
- where all of the integrity constraints are satisfied

Constraint Checking

24/70

If we have a database instance that satisfies all integrity constraints, what can go wrong?

The data might change \Rightarrow constraints need to be re-checked.

Possible changes:

- *insert* (add) a new record
- *delete* (remove) an existing record
- *update* (modify) an existing record

... Constraint Checking

25/70

For *domain* constraints ...

Insert:

- check each attribute value for type and additional domain constraints

Delete:

- no need to check any domain constraints

Update:

- check each modified attribute value for type and additional domain constraints

... Constraint Checking

26/70

These changes satisfy domain constraints:

```
insert Account(Downtown, A-456, 600)
insert Account(Perryridge, A-321, 200)
insert Account(Perryridge, A-102, 750)
  # but note duplicate key value
insert Account(Perryridge, A-131, 450)
  # value looks ok, but isn't correct
```

These changes do **not** satisfy domain constraints:

```
insert Account(Downtown, A-321, money)
  # 3rd attribute ( $a_3$ ) fails type check
insert Account(XYZZY, Hello, 300)
  # if we check for "lexically sensible" values on  $a_1, a_2$ 
insert Account(Brighton, A-402, -500)
  # if we check for positive opening balance
```

... Constraint Checking

27/70

For *key* constraints ...

Insert:

- check that key does not occur in any tuple already in the relation

Delete:

- no need to check anything

Update:

- if key attributes modified, same check as for insertion

... Constraint Checking

28/70

These changes satisfy key constraints:

```
insert Account(Downtown, A-456, 600)
insert Depositor(A-101, 9876543)
  # ok, only part of key duplicated
update Account(Downtown, A-101, 500)
  to Account(Downtown, A-101, 600)
  # ok, key attributes were not changed
insert Depositor(A-305, 8888888)
  # but no such customer
```

These changes do **not** satisfy key constraints:

```
insert Account(Perryridge, A-102, 750)
  # key A-102 already exists in relation
update Account(Downtown, A-101, 500)
  to Account(Downtown, A-201, 500)
  # key A-201 already exists in relation
```

... Constraint Checking

29/70

For *referential integrity* constraints ...

Insert:

- check that any foreign keys occur as primary keys in their own relation

Delete:

- check all relations that have foreign keys referring to this relation

Update:

- treat as delete-then-insert for constraint checking

... Constraint Checking

30/70

Example of deletion with foreign keys:

branchName	accountNo	balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700

branchName	address	assets
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

name	address	customerNo	homeBranch
Smith	Rye	1234567	Mianus
Jones	Palo Alto	9876543	Redwood
Smith	Brooklyn	1313131	Downtown
Curry	Rye	1111111	Mianus

account	customer
A-101	1313131
A-215	1111111
A-102	1313131
A-305	1234567
A-201	9876543
A-222	1111111
A-102	1234567

... Constraint Checking

31/70

How to handle violation of referential constraints on deletion?

One approach:

- simply disallow the deletion
- user must then find referring tuples and
 - either remove each one manually
 - or change their foreign keys to an acceptable value

Another approaches:

- remove all referring tuples automatically (cascade)
- set foreign key attributes to NULL in all referring tuples
(not possible if the foreign key also forms part of the primary key)

... Constraint Checking

32/70

These changes satisfy referential integrity constraints:

```
insert Account(Downtown, A-456, 600)
insert Depositor(A-215, 9876543)
update Account(Downtown, A-101, 500)
  to Account(Perryridge, A-101, 500)
# ok, a1 changed to valid FK
```

These changes do **not** satisfy referential integrity constraints:

```
insert Account(Wombatville, A-987, 500)
# no such branch
insert Depositor(A-305, 8888888)
# valid account, but no such customer
update Account(Downtown, A-101, 500)
  to Account(Nowhere, A-101, 500)
# no such branch
```

... Constraint Checking

33/70

These changes satisfy referential integrity constraints:

```
delete Depositor(A-102, 1234567)
delete Depositor(A-101, 1313131)
# although A-101 now has no "owner"
delete Branch(North Town, Rye, 3700000)
# ok, since no accounts or customers (but assets?)
```

These changes do **not** satisfy referential integrity constraints:

```
delete Branch(Perryridge, Horseneck, 1700000)
# some accounts are held at Perryridge
delete Customer(Smith, Rye, 1234567, Mianus)
# Depositor records become invalid
```

Mapping ER Designs to Relational Schemas

34/70

ER to Relational Mapping

35/70

As noted earlier, one useful strategy for database design:

- perform initial data modelling using ER or OO
(conceptual-level modelling)
- transform conceptual design into relational model
(implementation-level modelling)

By examining semantic correspondences, a formal mapping between the ER and relational models has been developed.

Because it is formal, it can be automated, and commercial tools now exist to perform it.

... ER to Relational Mapping

36/70

If we have tools, why worry about the mapping process itself?

It is still useful to understand how mapping occurs because:

- tools produce *correct* but (sometimes) *incomprehensible* relational descriptions
- to do performance tuning, you need to understand these descriptions
- you may need to use a different mapping to improve DB performance

Also, you're CSE students and you like to know how things work.

Relational Model vs ER Model

37/70

The relational and ER data models have some obvious correspondences:

Entity/Relationship	Relational
Attributes	Attributes (atomic)
Entity Relationship	Relation schema
Entity instance Relationship instance	Tuple (row, record)
Entity set Relationship set	Relation instance

38/70

... Relational Model vs ER Model

There are also differences between relational and ER models.

Compared to ER, the relational model:

- uses relations to model both entities *and* relationships
- has no composite or multi-valued attributes (only atomic-valued)
- has no object-oriented notions (e.g. subclasses, inheritance)

... Relational Model vs ER Model

39/70

At first glance, Relational looks less powerful than ER:

- less "mechanisms" and "weaker" data structuring tools

However, the relational model:

- can be used to represent any ER design
(although relational design may not be as "natural" as ER one)
- is simple, elegant and formal
⇒ provides a theory for evaluating relational designs
- has a model for query processing
⇒ provides a basis for efficient implementations

Mapping Strong Entities

40/70

An *entity* consists of:

- a collection of attributes;
attributes can be simple, composite, multi-valued

A *relation schema* consists of:

- a collection of attributes;
all attributes have atomic data values

So, even the mapping from entity to relation schema is not simple.

... Mapping Strong Entities

41/70

In one special case, there is an obvious mapping:

- an entity set E with atomic attributes a_1, a_2, \dots, a_n

maps to

- a relation R with attributes (columns) a_1, a_2, \dots, a_n

Each row in the relation R corresponds to an entity in E .

The *key* for the relation is the same (set of attributes) as for the entity set.

... Mapping Strong Entities

42/70

ER supports composite (hierarchical) attributes.

The relational model supports only atomic attributes.

Composite attributes consist of

- structuring attributes (non-leaf attributes)
- data attributes (containing atomic values)



... Mapping Strong Entities

43/70

One approach to mapping composite attributes:

- remove structuring attributes
- map atomic components to a set of atomic attributes (possibly with renaming)

E.g. Struct A {W, Struct B {X,Y}, Z} → (W,X,Y,Z)

It is common to retain structuring attribute as part of name to resolve name conflicts.

E.g. Struct Addr {number,street,suburb,pcode}
maps to (AddrNumber,AddrStreet,AddrSuburb,AddrPcode)

... Mapping Strong Entities

44/70

Alternative approach to mapping composite attributes:

- concatenate atomic attribute values into a string

E.g. Struct name {"John","Smith"} → "John Smith"

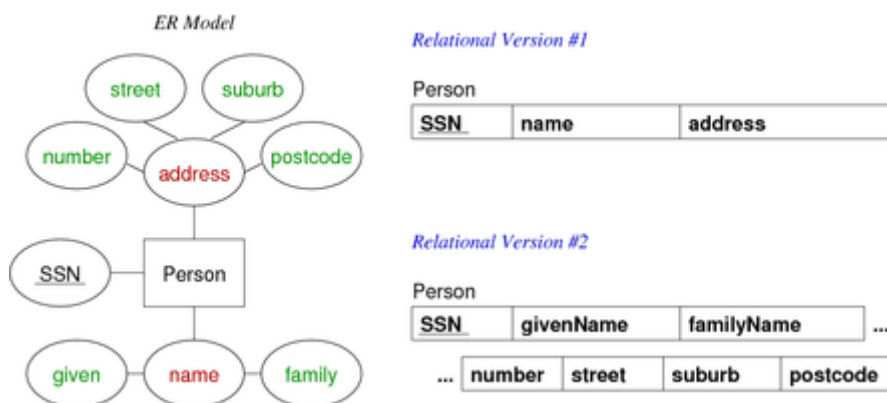
However, this approach "hides" information from data manipulation languages:

- requires extra extraction effort if components *are* required
- cannot exploit efficient query capabilities on components

... Mapping Strong Entities

45/70

Example:



Mapping Weak Entities

46/70

A weak entity set W

- has some attributes that form a *discriminator*, BUT
- is dependent on some other entity set E to form a key

If we simply form a relation for W by mapping its attributes, it would not be a valid relation because it would not have a key.

The solution:

- map the weak entity set to a relation, BUT also
- augment the relation by including E 's key

This always yields a relation with a valid key.

... Mapping Weak Entities

47/70

More formally:

- let W be a weak entity set with attributes w_1, w_2, \dots, w_n
- let E be its strong entity set with key e_1, e_2, \dots, e_m
- represent W by a table with columns $\{w_1, w_2, \dots, w_n\} \cup \{e_1, e_2, \dots, e_m\}$

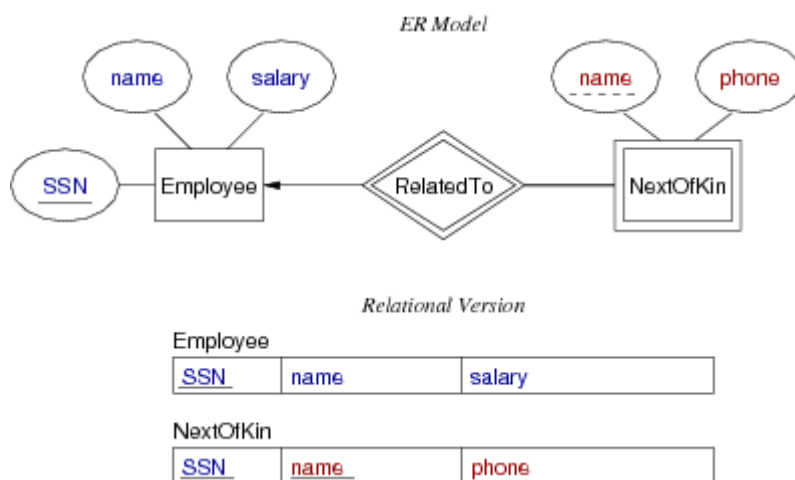
The key is E 's key (foreign key in W) plus the discriminator of W .

The weak relationship set between W and E is not explicitly represented.

... Mapping Weak Entities

48/70

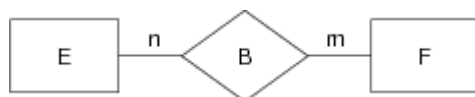
Example:



Mapping N:M Relationships

49/70

A **binary relationship set** B between entity sets E and F gives associations between pairs of entities in E and F



We can represent

- entity set E by relation S (using attribute mappings as above)
- entity set F by relation T (using attribute mappings as above)

But how to represent B ?

... Mapping N:M Relationships

50/70

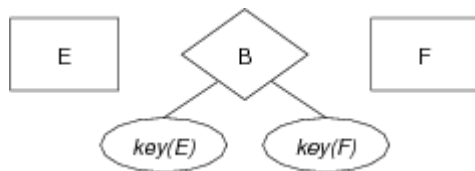
One possibility: represent the **relationship set** B explicitly by a **relation** R .

Each tuple (row) in R represents the **relationship** between a specific pair of entities from E and F .

For this to work, the tuple would need to contain information to identify the entities involved

This is achieved by storing the keys of the related entities.

It is somewhat like breaking the ER diagram up as follows:



... Mapping N:M Relationships

51/70

A relationship set $B(E, F)$ is represented by a relation R containing:

- all attributes from the primary keys of S and T
- all attributes associated with the relationship set B

where S and T are relations representing entity sets E and F .

The key for R is the union of the key attributes for S and T .

... Mapping N:M Relationships

52/70

This approach for representing relationships works generally:

- relationship degree ≥ 2
- relationship multiplicity 1:1, 1:N, N:M
- associated attributes are simply included in R

but requires a new relation to be created for each relationship set.

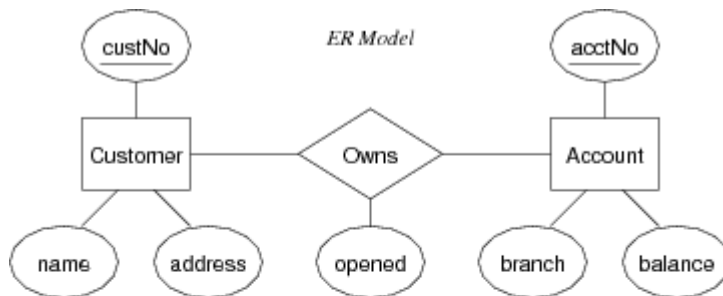
This can slow down query processing *considerably*.

In certain special cases, we do not need to create a new relation (see later).

... Mapping N:M Relationships

53/70

Example:



Relational Version

Customer		
<u>custNo</u>	name	address

Account		
<u>acctNo</u>	branch	balance

Owns		
<u>custNo</u>	<u>acctNo</u>	opened

Mapping 1:N Relationships

54/70

Consider a 1:N relationship R between entity sets E and F

- an entity in F is associated with at most one entity in E
- an entity in E may be associated with many entities in F

As above, we represent E and F by relations S and T .

How to capture the association between an entity in F and the corresponding entity in E ?

We have already seen one solution: introduce a new relation for R .

... Mapping 1:N Relationships

55/70

Since there is (at most) one corresponding entity, add attributes in F :

- to identify the corresponding entity (i.e. E 's key)
- to represent any attributes associated with R

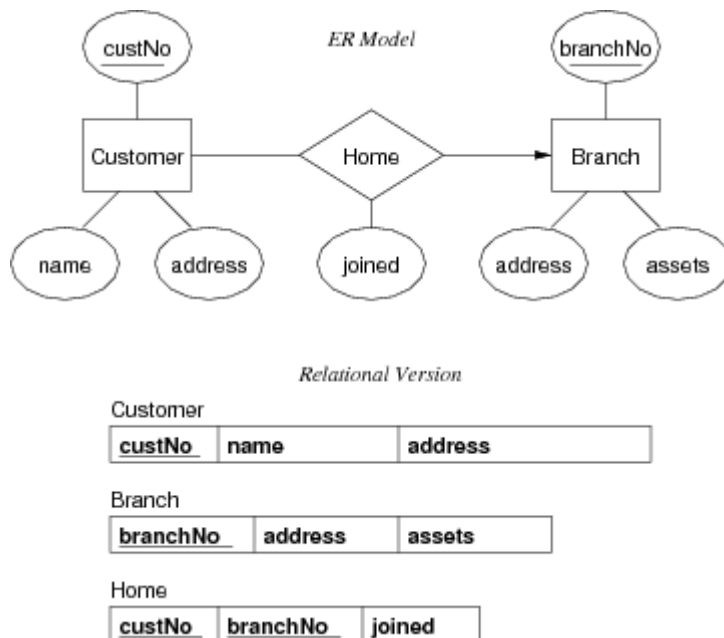
In other words, we insert a foreign key for E into F , along with any attributes for the relationship R .

If an entity in F has no relationship with E give NULL values to the "extra" attributes in F .

... Mapping 1:N Relationships

56/70

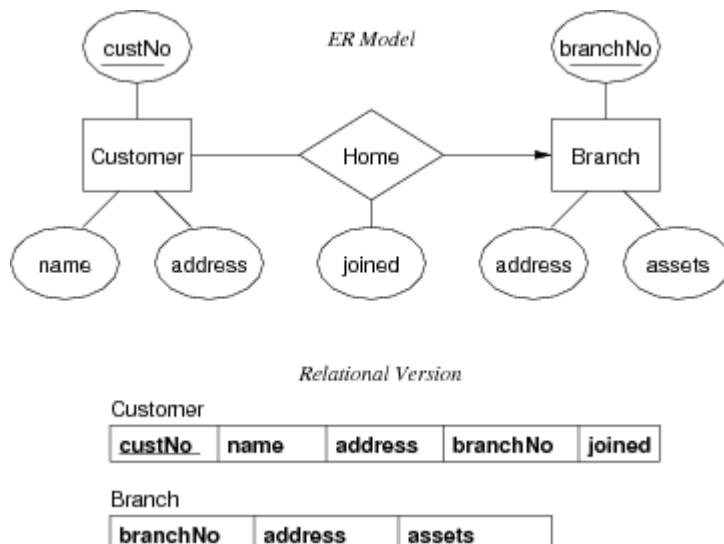
Example (generic mapping):



... Mapping 1:N Relationships

57/70

Example (optimised mapping):



Mapping 1:1 Relationships

58/70

1:1 relationships are handled in a similar manner to 1:N relationships.

The difference is that we could choose either relation to hold the key of the other relation, to represent the correspondence.

Choose the entity set that participates totally, if only one of them does.

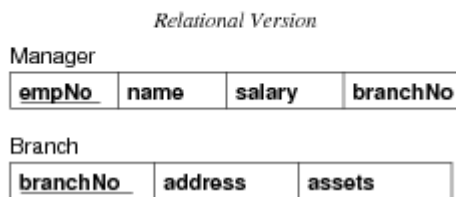
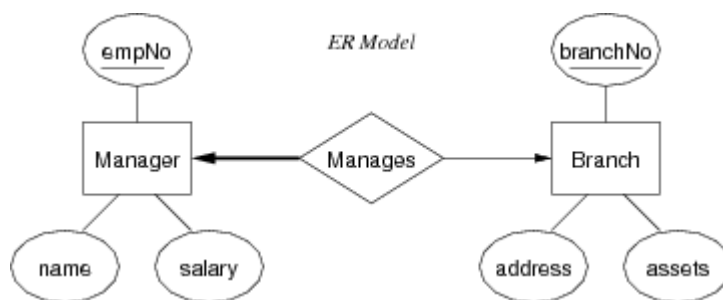
For a 1:1 relationship between entity sets E and F (S and T):

- choose one of S and T (e.g. S)
- add the attributes of T 's primary key to S as foreign key
- add the relationship attributes as attributes of S

... Mapping 1:1 Relationships

59/70

Example:



Mapping Multi-valued Attributes

60/70

An attribute in a relation may hold a single atomic value.

An attribute in an entity may hold multiple (structured) values.

A multi-valued attribute may be viewed as:

- a collection of values associated with an entity

so treat it like an N:M relationship between entities and values.

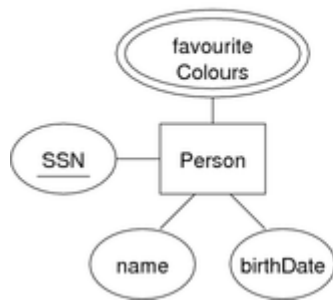
Create a new relation where each tuple contains:

- the primary key attributes from the entity
- one value for the multi-valued attribute from the corresponding entity

... Mapping Multi-valued Attributes

61/70

Example:



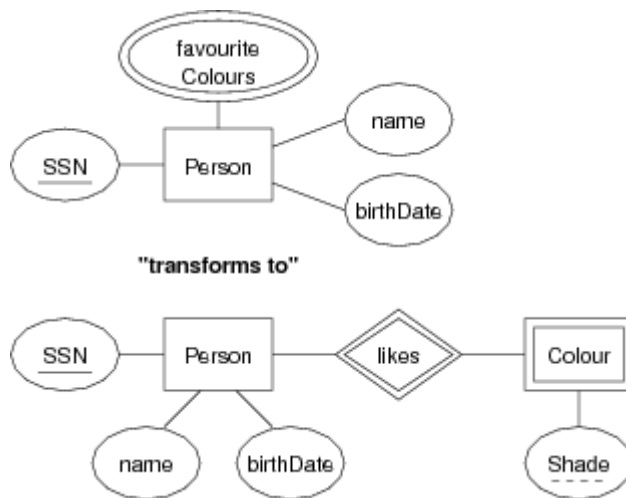
Person		
SSN	name	birthdate

FavColour	
SSN	colour

... Mapping Multi-valued Attributes

62/70

This approach is like altering the ER diagram as follows:



... Mapping Multi-valued Attributes

63/70

Example: the two entities

```

Person(12345, John, 12-feb-1990, [red,green,blue])
Person(54321, Jane, 25-dec-1990, [green,purple])
    
```

would be represented as

```

Person(12345, John, 12-feb-1990)
Person(54321, Jane, 25-dec-1990)
FavColour(12345, red)
FavColour(12345, green)
FavColour(12345, blue)
FavColour(54321, green)
FavColour(54321, purple)
    
```

Mapping Subclasses

64/70

Each subclass is represented as a separate relation.

Each entity in the subclass:

- contains its own subclass-specific information (attributes)
- needs to be associated with information in the superclass

Use the superclass entity's primary key to capture the association.

Each tuple in the subclass relation contains:

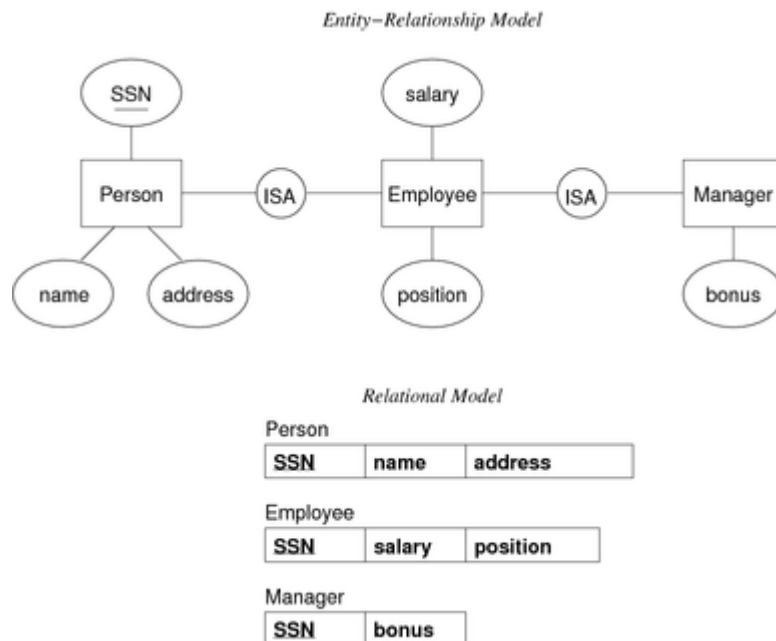
- all of the attributes from the parent's key

- all of the subclass-specific attributes

... Mapping Subclasses

65/70

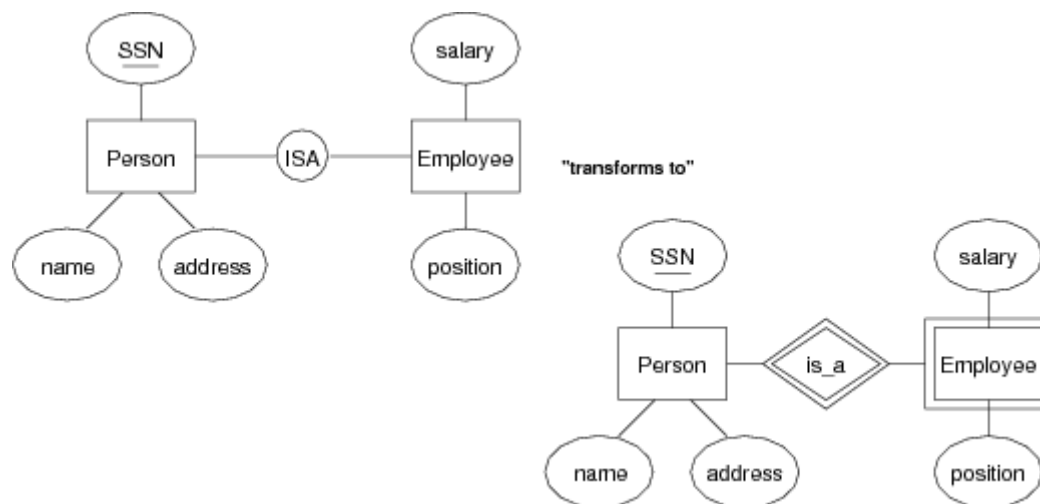
Example:



... Mapping Subclasses

66/70

This approach is like transforming the ER as follows:



... Mapping Subclasses

67/70

This approach to subclass mapping is called "ER style"

There are two other approaches to subclass mapping:

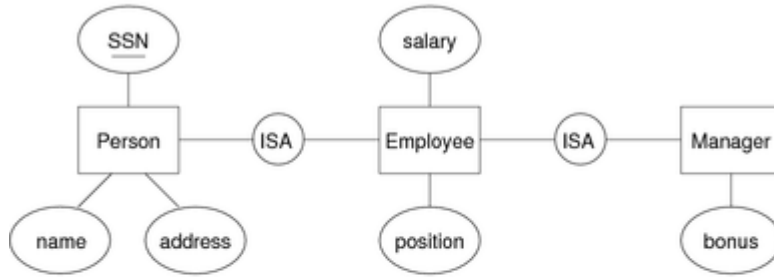
- object-oriented
 - each entity becomes a table, inheriting superclass attributes
- single table with nulls
 - one table, with all attributes of all subclasses

... Mapping Subclasses

68/70

Example of object-oriented mapping:

Entity-Relationship Model



Relational Model

Person		
SSN	name	address

Employee				
SSN	name	address	salary	position

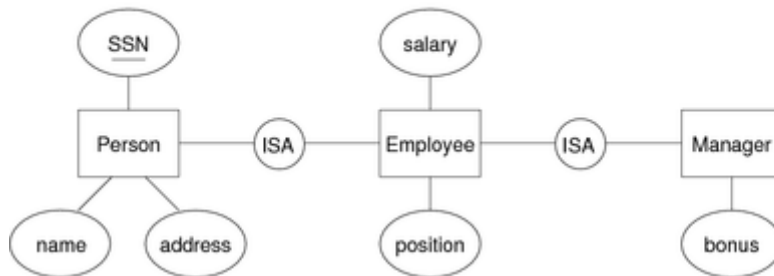
Manager					
SSN	name	address	salary	position	bonus

... Mapping Subclasses

69/70

Example of single-table-with-nulls mapping:

Entity-Relationship Model



Relational Model

Person					
SSN	name	address	salary	position	bonus

↑ NULL for Person who is not Employee
 ↑ NULL for Employee who is not Manager

... Mapping Subclasses

70/70

Which mapping is best depends on other requirements ...

- ER-style good for queries like "find average salary"
 - need to look only in (relatively small) Employee table
- OO-style good for queries like "find manager names and bonuses"
 - need to look only in Manager table
- Single-table saves space, unless many NULL values