Object Relational

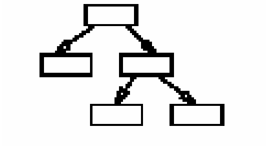
Paolo Cappellari

History of DB Models

- 1950 File Systems, Punched Cards

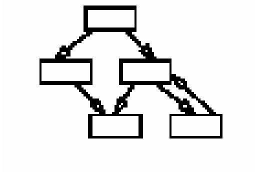
- 1960 Hierarchical

  (IMS IBM Mainframes)



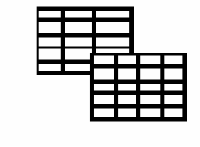
- 1970 Network

  (CODASYL, IDMS)



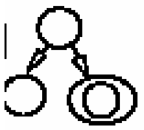
- 1980 Relational

  (ORACLE, DB2, Sybase)



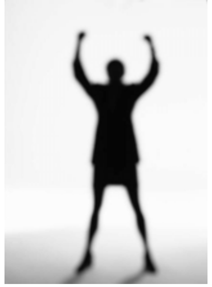
- 1990 Object-Oriented, Object-Relational

  (O2, ORACLE9i)



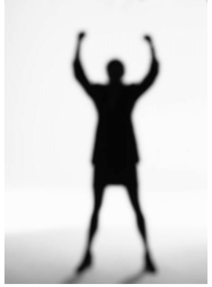
Relational Model

- Emergence of data model
- Data independence
- High-level approach
- Standardization



- Built-in data types
- Little abstraction
- Separation between data and operations

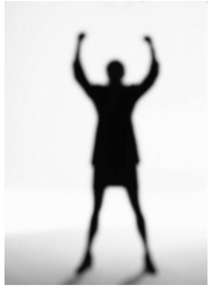
Object-Oriented Model



- Complex application datatypes
- Object Abstraction
- Encapsulation of behavior
- High performance for specific application

- No backwards compatibility
- Closely tied to language and application

Object-Relational Model



- Synthesis of two worlds
- Upward compatibility
- Robustness of mature systems

- Hybrid approach
- Legacy problems

Evolution of DBMS’s

- Object-oriented DBMS’s failed because they did not offer the efficiencies of well-entrenched relational DBMS’s.
- Object-relational extensions to relational DBMS’s capture much of the advantages of OO, yet retain the relation as the fundamental abstraction.

Main Features

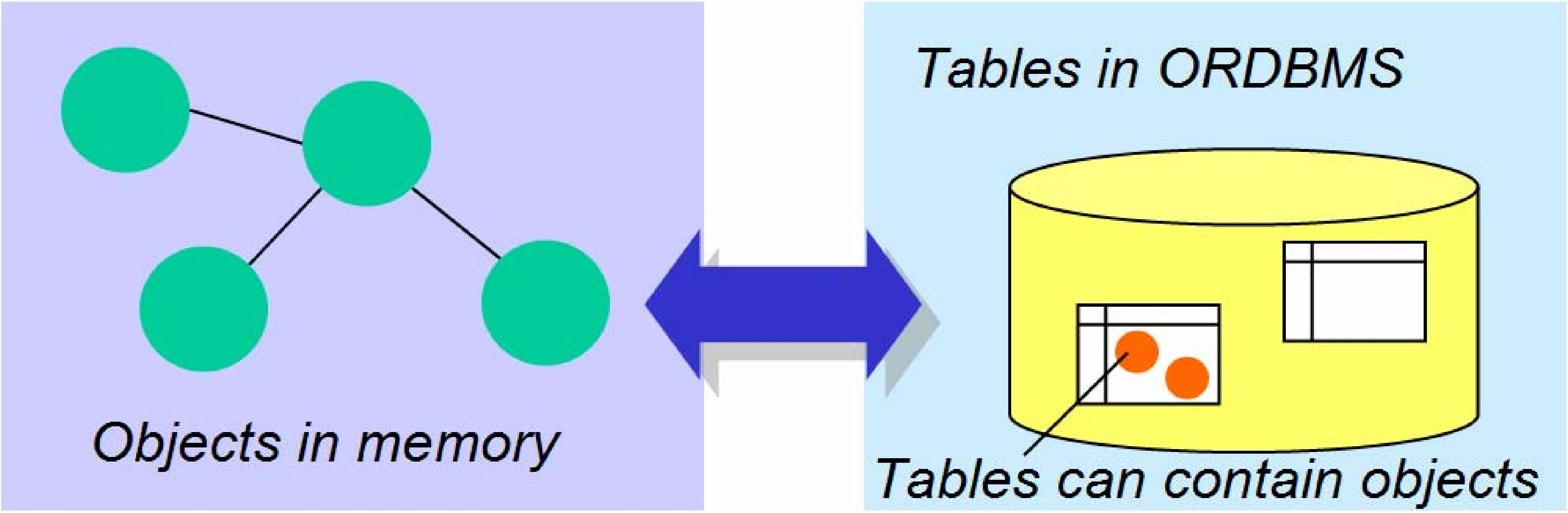
Relation is still the fundamental abstraction, with integration of OO features

- Structured Types
  - Non-atomic types
- Methods
  - Special operations can be defined for a type
- References
  - Pointers to tuples

Object Orientation

- Abstract data types
  - defining classes with data structure and operation on them whose details are hidden
- Object Identity
  - every entity is uniquely identifiable
- Polymorphism and overloading
  - to distinguish between two or more operation having the same name that have different semantics or that operate on values of different types
- Inheritance
  - share structure and behaviour among related types

OO – OR Mapping



Structured Types– SQL99

- UDT – User Defined Type
  - A UDT can be the type of a table
  - A UDT can be the type of an attribute belonging to some table
  - A UDT can inherit from another UDT

Three kinds of UDTs

- Distinct types
- Structured UDTs as values
- Structured UDTs as objects

Distinct types

```
CREATE TYPE shoe_size AS INTEGER FINAL
CREATE TYPE iq AS INTEGER FINAL

CREATE TABLE demograph_people (
    name                 CHARACTER VARYING (50),
    footsies             shoe_size,
    smarts               iq,
    last_purchase        DECIMAL(5,2)     )
```

Distinct types

- Incorrect use of distinct types

```
SELECT name
FROM demograph_people
WHERE footsies > smart
```

- Correct use: CAST

```
…
WHERE
        CAST (footsies TO INTEGER)
        >
        CAST (smarts TO INTEGER)
```

Distinct types (DB2)

CREATE **DISTINCT** TYPE shoe_size AS INTEGER
WITH COMPARISONS

- Keyword DISTINCT is required
- Generates functions to cast between the distinct type and its source type
- WITH COMPARISONS is required for almost all base types and generates support for the comparison operators (=, <>, <, <=, >, >=).

Structured UDTs as values

- Address
  - Number, street_name, apartment_number, city, state, postal_code
- Movie
  - Title, length, description, …

- Can be used as first-class type
- Can have functions associated
  - Comparison (*equals*) function, …

Create type syntax

CREATE type-name

AS *representation*

[[NOT] INSTANTIABLE]

[[NOT] FINAL]

[*reference-type-specification*]

[*method-specification-list*]

Type definition

```
CREATE TYPE addressLongT AS (
Number              CHARACTER(6),
Street              ROW (
    street_name              CHARACTER VARYING(35),
    street_type              CHARACTER VARYING (10)
                                    DEFAULT ‘Street’       ),
City                CHARACTER VARYING (35),
State               CHARACTER(2) NOT NULL,
Zip_code            ROW (
    base                      CHARACTER(5),
    plus4                     CHARACTER(4)    )       )
NOT FINAL
```

Type definition (DB2)

```
CREATE TYPE addressT AS (
Number      CHARACTER(6),
City        CHARACTER VARYING(35),
State       CHARACTER(2) )
NOT FINAL
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
```

Type definition (DB2)

```
CREATE TYPE PersonT AS (
name VARCHAR(50),
address addressT )
NOT FINAL
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
```

Type definition (DB2)

```
CREATE TYPE PersonT AS (
name VARCHAR(50),
address REF(addressT) )
NOT FINAL
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
```

Type definition (DB2)

```
CREATE TYPE addressT AS (
Number CHARACTER(6),
City      CHARACTER VARYING(35),
State     CHARACTER(2) )
NOT FINAL
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
   METHOD DISTANCE (addressT)
   RETURNS FLOAT
   LANGUAGE JAVA
   PARAMETER STYLE DB2GENERAL  -- to be used with structured and JAVA
   NO SQL                      -- no SQL statement allowed in method
```

Accessing attributes

- Suppose to have a relation customer (c) with a column (cust_addr) defined of type *addressLongT*
- c.cust_addr.number
- c.cust_addr.zip_code.base
- c.cust_addr[2].zip_code.base

- Columns can be defined as ROW and also as ARRAY.

Accessing attributes (DB2)

- Suppose to have a relation customer (c) with a column (cust_addr) defined of type addressLongT
- c.cust_addr..number
- c.cust_addr..zip_code..base
- c.cust_addr[2]..zip_code..base

Accessing attributes (DB2)

```
CREATE FUNCTION addrT_transform (
                                    addr addressT)
RETURNS VARCHAR(100) LANGUAGE SQL
RETURN
  addr..number || ', ' || addr..city || ', ' || addr..state

CREATE TRANSFORM FOR addressT
  DB2_PROGRAM (
  FROM SQL WITH FUNCTION addrT_transform )
```

Accessing attributes

- Use the alias (or correlation) name to avoid ambiguities:

```
SELECT
        customers.name,
        customers.cust_addr.street_name
FROM
        customers, customers.cust_addr
```

- Should the expression *cust_addr.street_name* be resolved as *schema.table.column* or *table.column.attribute*?

Accessing attributes

- Use the correlation name to avoid ambiguities:

```
SELECT
        c.name,
        c.cust_addr.street_name
FROM
        customers AS c
```

Observer and Mutators

- Allow to access, set and retrive, the attributes of UDTs.
- They are methods that the system automativcally provides.

Observer and Mutators

SELECT movie.runs
FROM movie_table
WHERE
title = ‘Star Wars’
	SELECT movie.runs()
	FROM movie_table
	WHERE
	title = ‘Star Wars’

UPDATE movie_table
SET movie.runs = 113
WHERE
title = ‘Star Wars’
	UPDATE movie_table
	SET movie = movie.runs(113)
	WHERE
	title = ‘Star Wars’

Observer and Mutators

SELECT movie..runs
FROM movie_table
WHERE
title = ‘Star Wars’
	SELECT movie..runs()
	FROM movie_table
	WHERE
	title = ‘Star Wars’

UPDATE movie_table
SET movie..runs = 113
WHERE
title = ‘Star Wars’
	UPDATE movie_table
	SET movie = movie..runs(113)
	WHERE
	title = ‘Star Wars’

Method definition

- Methods are defined in two ways and in two places, and both are required
  - Define the signature among the type definition
  - Define the implementation

Method definition

```
CREATE TYPE movieT AS (
   title          CHARACTER VARYING (100),
   description    CHARACTER VARYING (500),
   runs           INTEGER )
NOT FINAL
METHOD length_interval ( )
   RETURNS INTERVAL HOUR(2) TO MINUTE
```

Method definition

```
CREATE INSTANCE METHOD
   length_interval ( )
RETURNS INTERVAL HOUR(2) TO MINUTE
FOR movie
/*
   implementation
*/
RETURN …
```

Method invocation

```
CREATE TABLE movie_table (
    stock_number                CHARACTET(8),
    movie_info                  movieT,
    rental_quantity             INTEGER,
    rental_cost                 DECIMAL(5,2)     )

SELECT mt.movie_info.length_interval
FROM movie_table AS mt
WHERE mt.movie_info.title = ‘Star Wars’
```

Method definition (DB2)

```
CREATE TYPE addressT AS (
Number          CHARACTER(6),
City     CHARACTER VARYING(35),
State    CHARACTER(2) )
NOT FINAL
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
   METHOD SAMECITY (addr addressT)
   RETURNS INTEGER
   LANGUAGE SQL
```

Method definition (DB2)

```
CREATE METHOD SAMECITY (addr addressT)
RETURNS INTEGER
FOR addressT
RETURN (
    CASE WHEN (self..city = addr..city)
    THEN 1
    ELSE 0
END)
```

Constructors

- Each defined UDT has a constructor
- The system automatically provides for a *niladic* constructor
- Users usually need for more sophisticated constructors
- Constructors method are marked with the keyword CONSTRUCTOR in the method definition

Constructor definition

```
CREATE CONSTRUCTOR METHOD movieT (
    name CHARACTER VARYING(100),
    descr CHARACTER VARYING(500),
    length INTEGER )
RETURNS movieT
BEGIN
    SET SELF.title = name;
    SET SELF.description = descr;
    SET SELF. runs = length;
    RETURN SELF;
END
```

Constructor definition (DB2)

```
CREATE function addressT (
    num         CHARACTER(6),
    cit         CHARACTER VARYING(35),
    sta         CHARACTER(2) )
RETURNS addressT
RETURN addressT()..number(num)..city(cit)..state(sta)
```

Constructor definition

```
CREATE TABLE TestTable (
        col1 INTEGER,
        col2 address )

insert into TestTable values (                         2,
        address20()..number('a')..city('b')..
                                state('c'))

insert into TestTable values (8, address20('d','e','f') )
```

Storing in the database

```
INSERT INTO movie_table VALUES (
  ‘152208-A’,          -- stock-number
  NEW movieT(
      ‘Star Wars’,
      ‘Action-Fantasy. Part IV in a George   Lucas
       epic, Star Wars: …  ’
      125    ),        -- new MOVIE instance
  23,                  -- rental-quantity in stock
  2.99        )        -- rental-cost
```

Storing in the database (DB2)

```
INSERT INTO movie_table VALUES (
   ‘152208-A’,        -- stock-number
   movieT()..title(‘Star Wars’)..description(‘Action-
   Fantasy. Part IV in a George Lucas epic, Star Wars:
   …  ’)..runs(125) ),
               -- new MOVIE instance
   23,         -- rental-quantity in stock
   2.99        )       -- rental-cost
```

Storing in the database (DB2)

```
INSERT INTO movie_table VALUES (
    ‘152208-A’,          -- stock-number
    movieT(
        ‘Star Wars’,
        ‘Action-Fantasy. Part IV in a George   Lucas
         epic, Star Wars: …  ’
        125   ),           -- new MOVIE instance using the
                           -- constructor function
    23,                    -- rental-quantity in stock
    2.99         )         -- rental-cost
```

Structured UDTs as objects

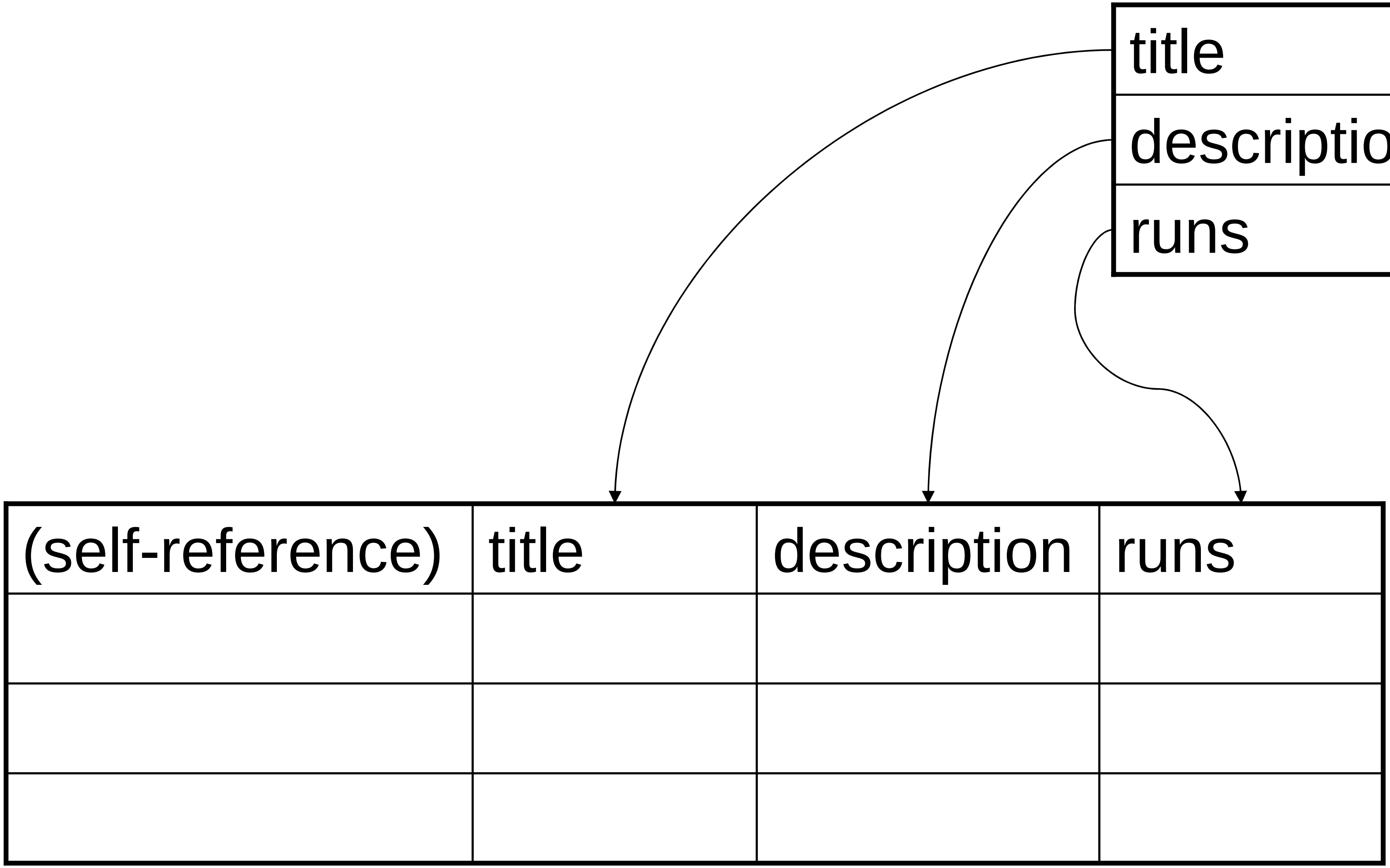
- Define a special sort of table (typed table or table of type) to represents instances of a type.
- Each instance is unique and has its own indenty.
  - Each instance behaves exactly as an object
- Each row stored in the table is an instance, or a value, of the associated structured UDT.

Typed tables

- The typed table has a column for each attributes in the UDT associated, plus an *object-identifier* known as *Self-referencing column*.

```
CREATE TABLE movie_TypedTable OF movieT
    REF IS oidName SYSTEM GENERATED
```

Typed tables



title

description

runs

(self-reference)	title	description	runs

self-reference specification

- In the type definition the reference type has to be specified:

  CREATE TYPE movieT AS (*attributes*)

  NOT FINAL

  <*reference-type specification*>

Types of *self-reference specification*

- <*reference-type specification*> can be:
  - System-generated ::= REF IS SYSTEM GENERATED
  - User-defined ::= REF USING <*predefined SQL type*>
  - Derived ::= REF FROM <*attribute-list from the structured type*>

- When defining a typed table, the <*reference-type specification*> must be specified again(redundantly), associating it with a name (the name of the *self-referencing* column).

Values for *self-reference*

- REF IS <*selfColumnName*>
  - SYSTEM GENERATED: generated by the system.
  - USER GENERATED: it is responsability of the application to choose the values stored in each row of the column.
  - DERIVED: the system uses the values in the specified columns (in the type definition) to derive the reference value. The columns should be under a PRIMARY KEY or a UNIQUE constraint.

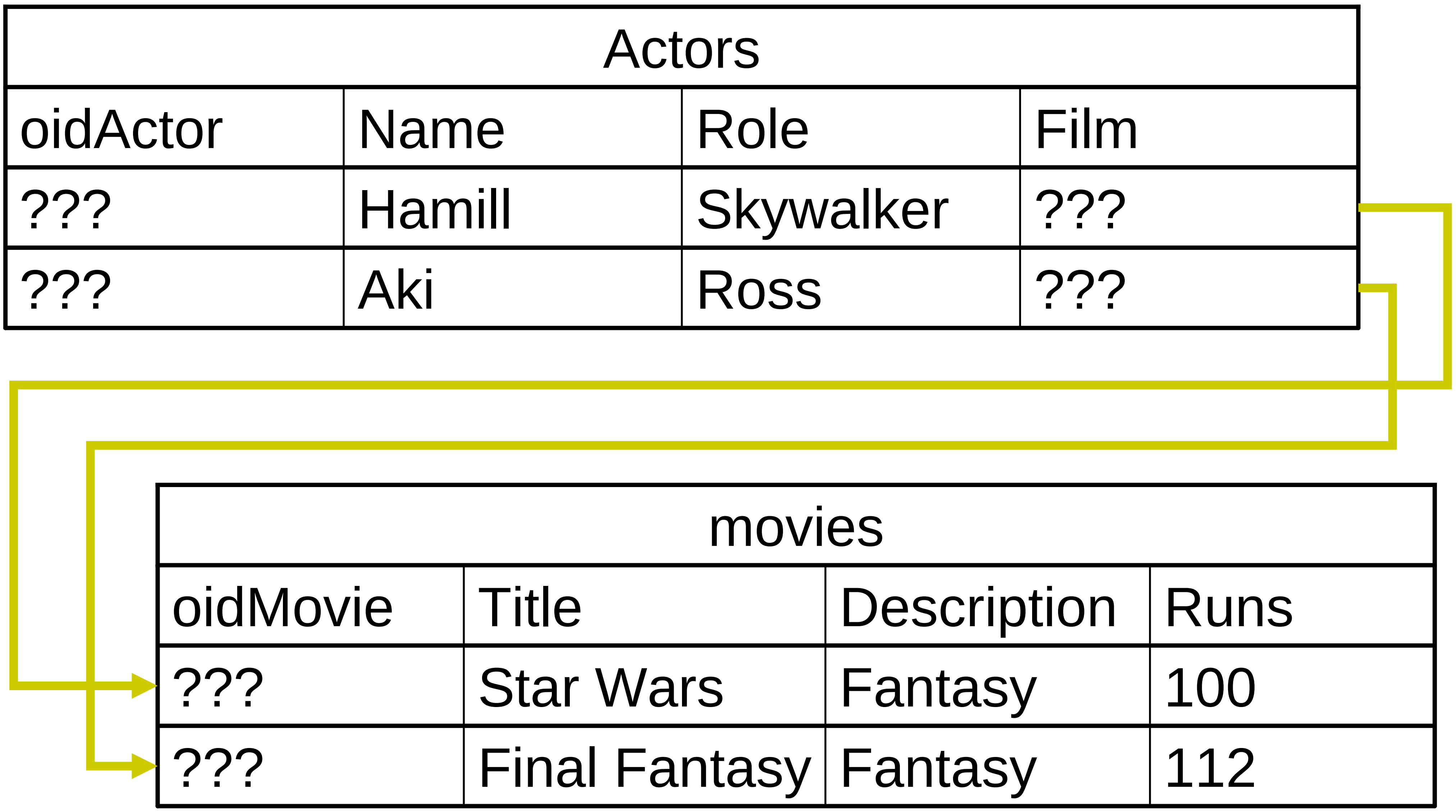
References

```
CREATE TYPE movieT AS (
    title         CHARACTER VARYING (100),
    description   CHARACTER VARYING (500),
    runs          INTEGER )
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED

CREATE TYPE playerT AS (
    name          CHARACTER VARYING (35)
    role          CHARACTER VARYING (35) )
    film          REF (movieT) )
NOT FINAL
REF IS FROM (name, role, film)
```

References

```
CREATE TABLE movies OF movieT
title WITH OPTION CONSTRAINT NOT NULL,
REF IS oidMovie SYSTEM GENERATED

CREATE TABLE actors OF playerT
PRIMARY KEY (name, role, film),
film WITH OPTION SCOPE movies,
REF IS oidActor DERIVED
```

References



Actors			
oidActor	Name	Role	Film
???	Hamill	Skywalker	???
???	Aki	Ross	???

movies			
oidMovie	Title	Description	Runs
???	Star Wars	Fantasy	100
???	Final Fantasy	Fantasy	112

Following the reference

- Retrive values:

```
SELECT film -> runs
FROM actors
WHERE name = ‘Hamill’ and role = ‘Skywalker’
```

- The statement retrives a value from the movies table without specifing that table in the FROM clause. The

- Retrive structured type instance:

```
SELECT DEREF (film)
FROM actors
WHERE name = ‘Hamill’ and role = ‘Skywalker’
```

Typed table (DB2)

```
create table Address of addressT (
   ref is oidAddress system generated,
   number  WITH OPTIONS NOT NULL,
   state      WITH OPTIONS NOT NULL,
   CONSTRAINT pk PRIMARY KEY (
         number, state)  )
```

Typed table (DB2)

```
CREATE TYPE addressT AS (
    street          varchar(50),
    city            varchar(50),
    zip             varchar(4) )
NOT FINAL
INSTANTIABLE
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
```

```
CREATE TYPE personT AS (
    name            varchar(50),
    age             varchar(50),
    address         REF(addressT)
          )
NOT FINAL
INSTANTIABLE
MODE DB2SQL
WITH FUNCTION ACCESS
REF USING INTEGER
```

Typed table (DB2)

```
create table Address of addressT (
ref is oidAddress system generated,
street WITH OPTIONS NOT NULL,
city WITH OPTIONS NOT NULL,
CONSTRAINT pkAddress PRIMARY KEY (street, city)  )

create table Person of personT (
ref is oidPerson system generated,
name WITH OPTIONS NOT NULL,
age WITH OPTIONS NOT NULL,
CONSTRAINT pkPerson PRIMARY KEY (name, age),
address WITH OPTIONS SCOPE Address)
```

Typed table (DB2)

```
create table Address of addressT (
ref is oidAddress system generated,
street WITH OPTIONS NOT NULL,
city WITH OPTIONS NOT NULL,
CONSTRAINT pk PRIMARY KEY (street, city)  )
```

```
create table X (a varchar(50) NOT NULL, b varchar(50), c
    varchar(50), d varchar(50), e varchar(50),
PRIMARY KEY (a),
FOREIGN KEY (b,c) REFERENCES Person (name, age),
FOREIGN KEY (d,e) REFERENCES Address (street, city) )
```

Insert into typed table (DB2)

```
insert into Address values
   (AddressT(5),'a','b','c');

insert into Person values
   (PersonT(5),'nome','eta',AddressT(5));
```

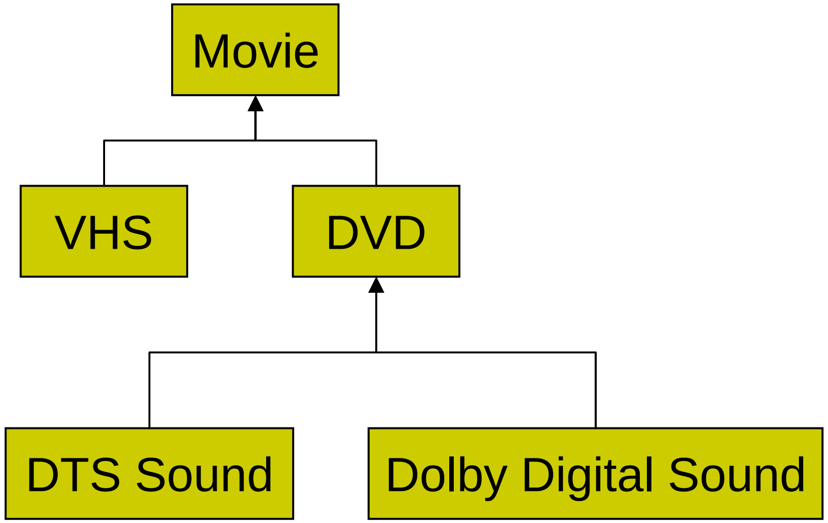
Following references DB2

```
select name, address->city, address->zip
from Person
```

- No mention to the Address table!

Inheritance

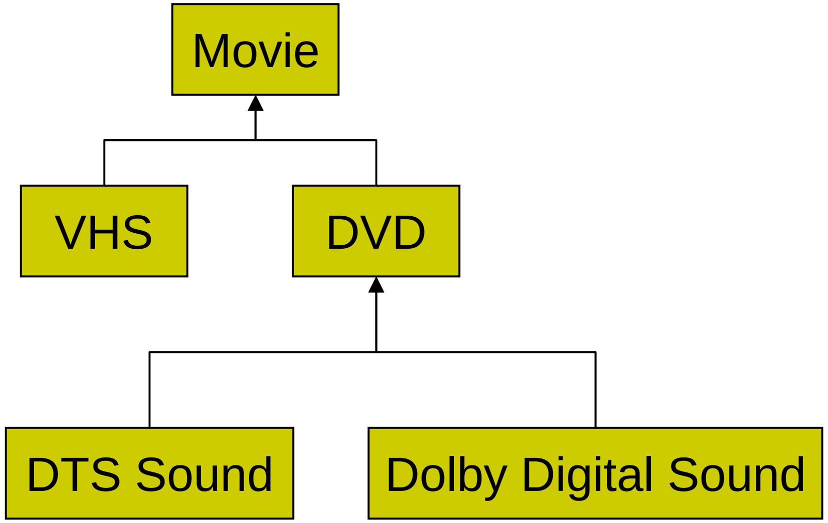
- A type hierarchy in SQL is a collection of UDTs.



Inheritance on types

- Super-type definition

```
CREATE TYPE movieT AS (
   title          CHARACTER VARYING (100),
   description    CHARACTER VARYING (500),
   runs           INTEGER )
NOT INSTANTIABLE
NOT FINAL
```



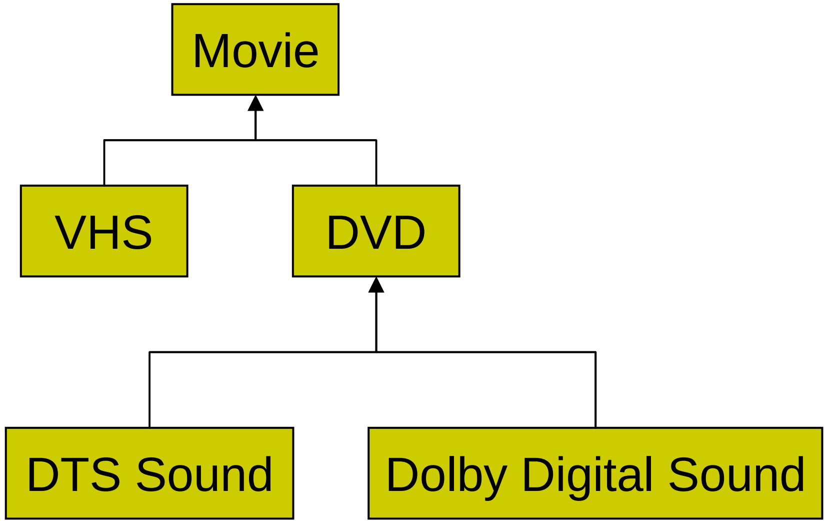
Inheritance on types

- Sub-type definition

```
CREATE TYPE dvdT UNDER movieT AS (
   stock-number INTEGER,
   rental_price    DECIMAL(5,2),
   extra_feature  feature_desc ARRAY[10] )
```

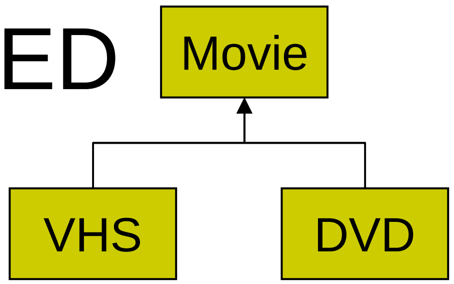
INSTANTIABLE

NOT FINAL



Inheritance on typed-tables

```
CREATE TABLE short_movies OF movieT
REF IS oidMovie SYSTEM GENERATED,
runs WITH OPTION CONSTRAINT smc_runs
   CHECK (runs < 90)

CREATE TABLE short_dvd OF dvdT
UNDER short_movies
REF IS oidDvd SYSTEM GENERATED
```

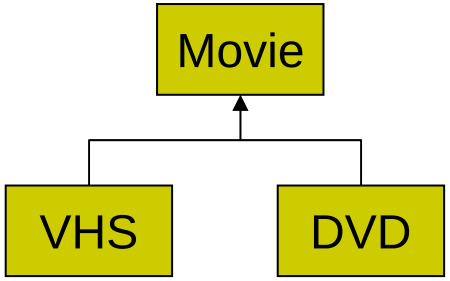


Retrieval in hierarchies

- The query:

```
SELECT titles, runs
FROM short_movies
WHERE runs < 60
```

- Retrives title and runs from short_movies tables, then retrives title and runs from short_dvd table!

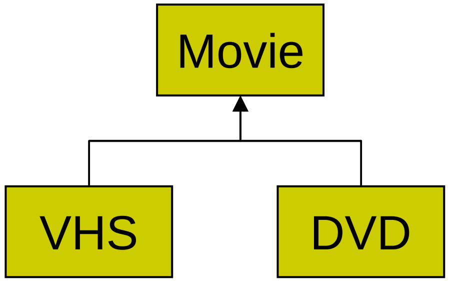


Retrieval in hierarchies

- The query:

```
SELECT titles, runs
FROM ONLY (short_movies)
WHERE runs < 60
```

- Retrives title and runs from short_movies that are ***not*** available on dvd (and on VHS).



The type predicate

- Allows to determine the type of a structured type instance.

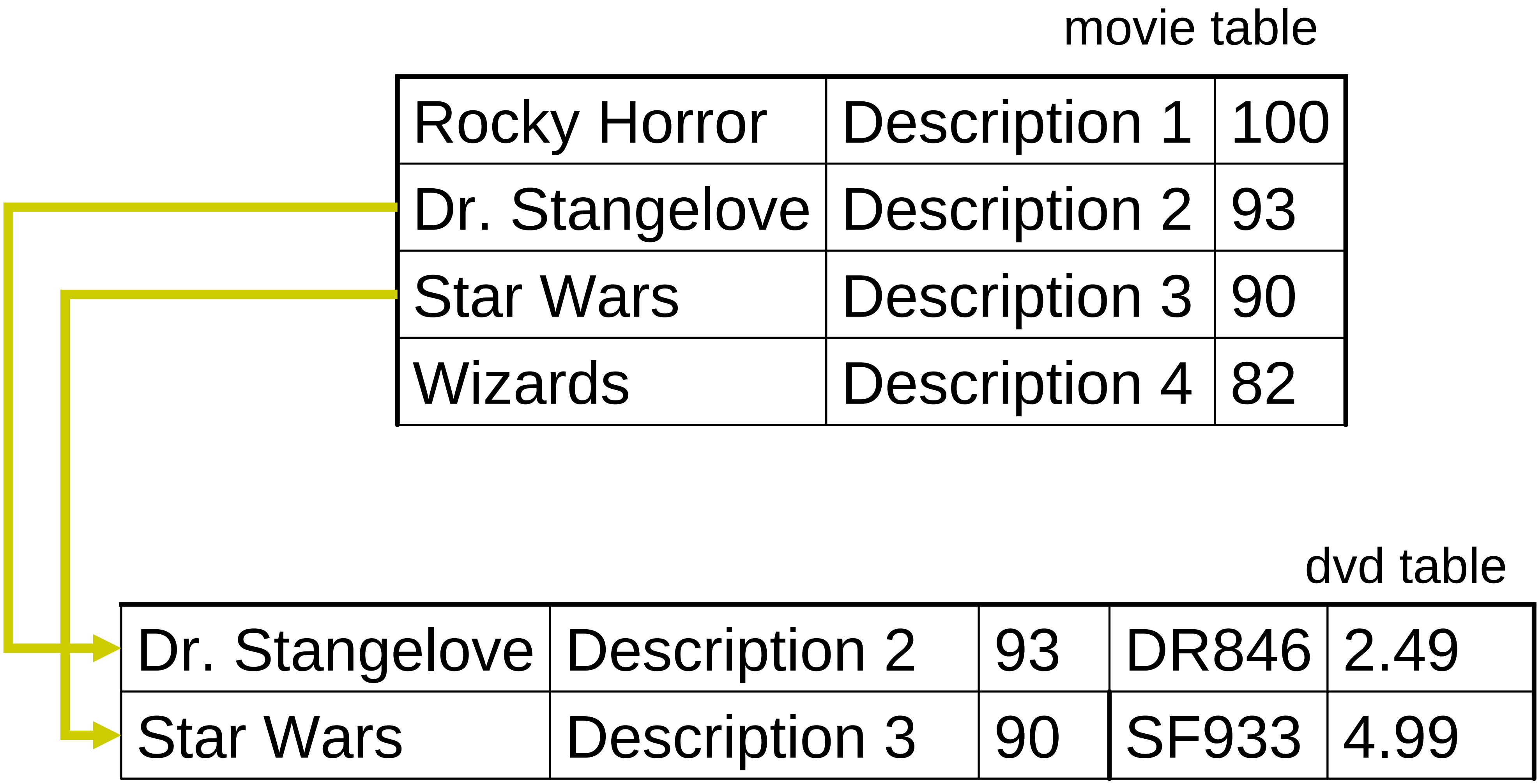
```
SELECT name, title
FROM actors
WHERE film IS OF (dvd)
```

- WHERE film IS NOT OF (dvd)
- WHERE film IS OF (ONLY dvd)

The hierarchy model

- There are several mental models to represent relationships between the tables in a table hierarchy and the rows in those tables.
  - Duplicate-row model
  - Single-table model
  - Union model

Duplicate-row model



movie table

Rocky Horror	Description 1	100
Dr. Stangelove	Description 2	93
Star Wars	Description 3	90
Wizards	Description 4	82

dvd table

Dr. Stangelove	Description 2	93	DR846	2.49
Star Wars	Description 3	90	SF933	4.99

Single-table model

movie + dvd table

movie	Rocky Horror	Description 1	100		
DVD	Dr. Stangelove	Description 2	93	DR846	2.49
DVD	Star Wars	Description 3	90	SF933	4.99
movie	Wizards	Description 4	82		

Union model

movie table

Rocky Horror	Description 1	100
Star Wars	Description 3	90

dvd table

Dr. Stangelove	Description 2	93	DR846	2.49
Star Wars	Description 3	90	SF933	4.99

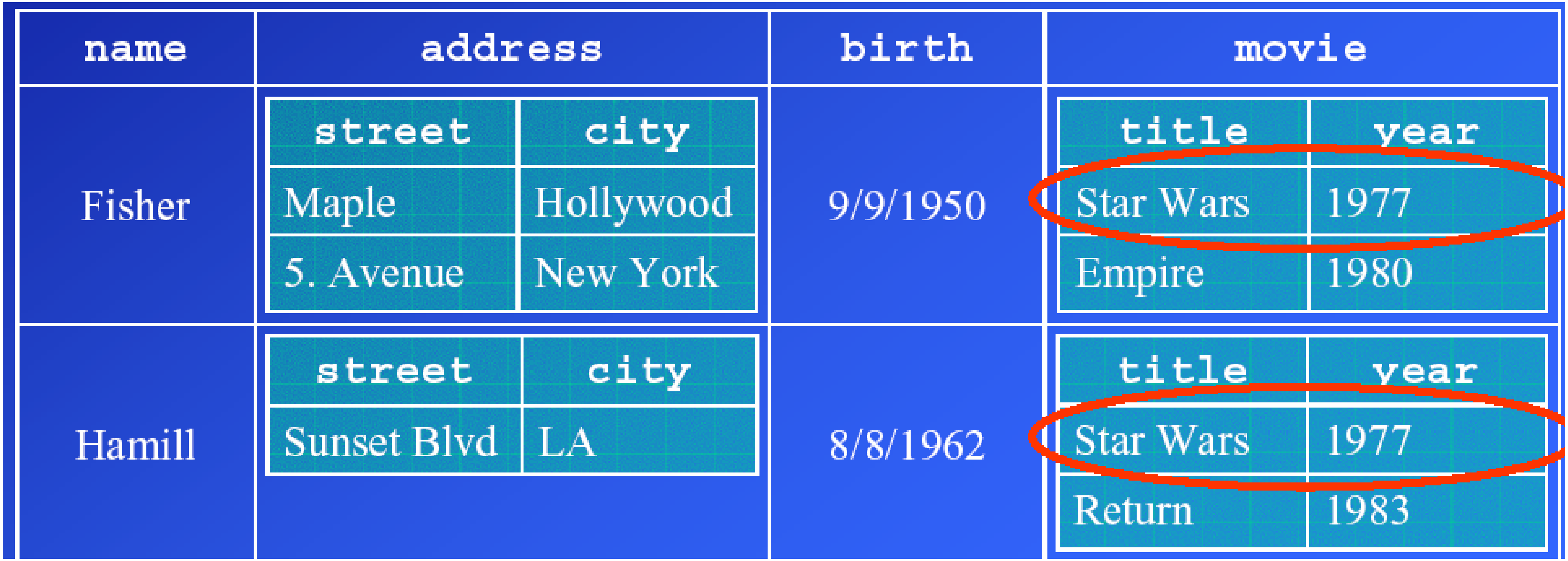
Hierarchies in DB2

- Conform to the standard!

FINE

References - I

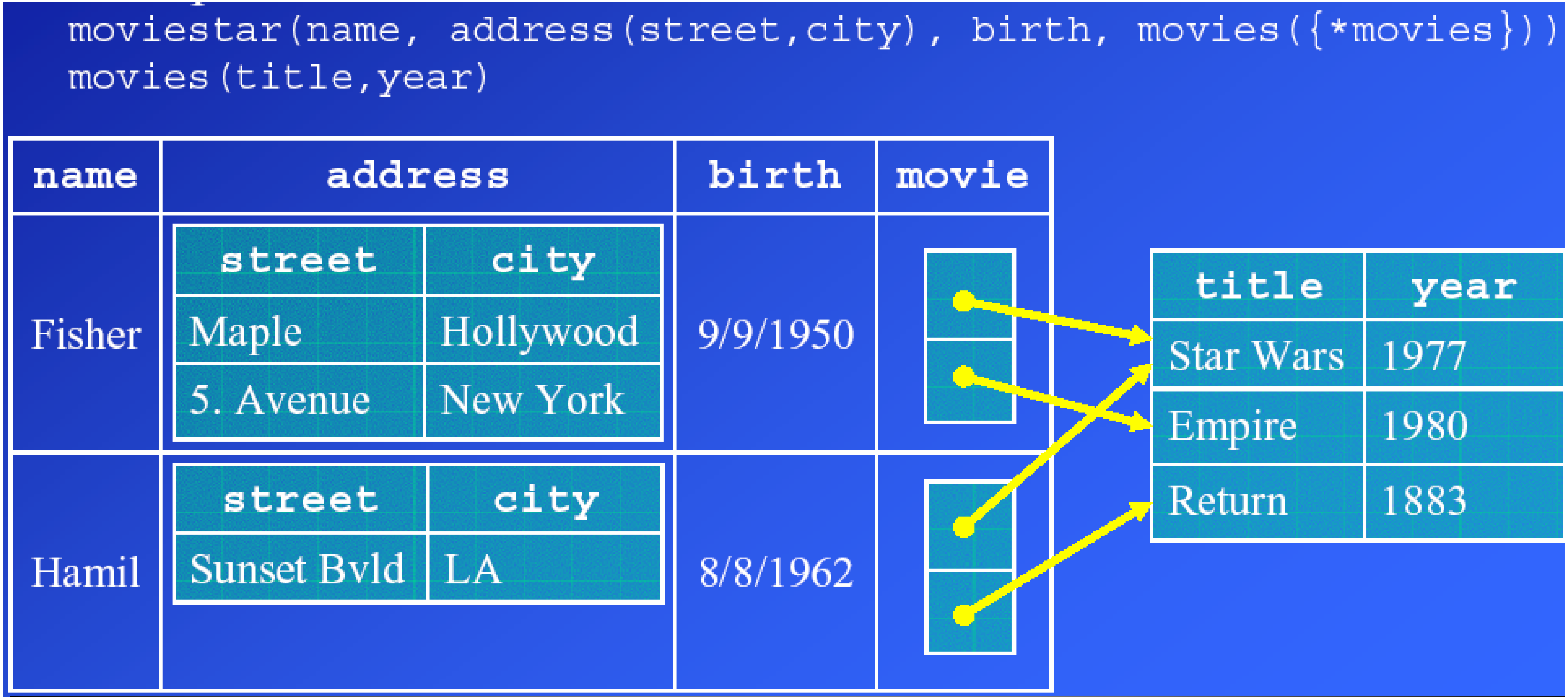
- Allow a tuple *t* refer to a tuple *s* rather than including *s* in *t*



name	address		birth	movie	
	street	city		title	year
Fisher	Maple	Hollywood	9/9/1950	Star Wars	1977
	5. Avenue	New York		Empire	1980
Hamill	Sunset Blvd	LA	8/8/1962	Star Wars	1977
				Return	1983

References - II

- If attribute A has a type that is a reference to a tuple in relation with schema R, we denote A as *A(*R)*
- If A is a set of references, we denote A as *A({*R})*



```
moviestar(name, address(street,city), birth, movies({*movies}))
movies(title,year)
```

name	address	birth	movie
Fisher	street | city Maple | Hollywood 5. Avenue | New York	9/9/1950	
Hamil	street | city Sunset Bvld | LA	8/8/1962	

title	year
Star Wars	1977
Empire	1980
Return	1883

References – SQL99 - I

- A table which type is a UDT may have a reference column that serves as its “ID”

In CREATE TABLE statement, add

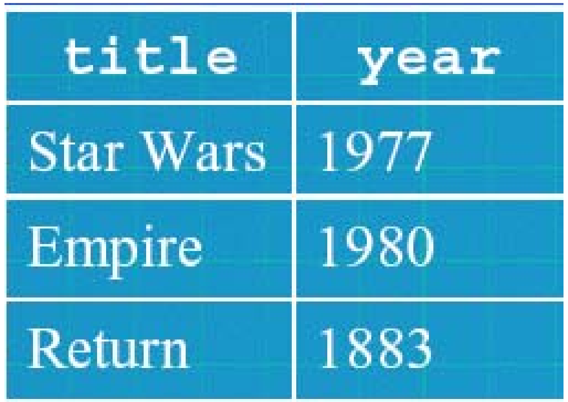
```
REF IS <attribute name> <how generated>
```

Where <how generated> is either

- SYSTEM_GENERATED : DBMS generates unique IDs
- DERIVED: DBMS uses primary key of the relation for IDs

References – SQL99 – I - Example

```
CREATE TYPE MovieType AS (
   title         CHAR(30),
   year          INTEGER
);

CREATE TABLE Movie OF MovieType (
   REF IS movieID DERIVED,
   PRIMARY KEY (title, year)
);
```



title	year
Star Wars	1977
Empire	1980
Return	1883

References – SQL99 - II

Reference to a tuple of type *T*

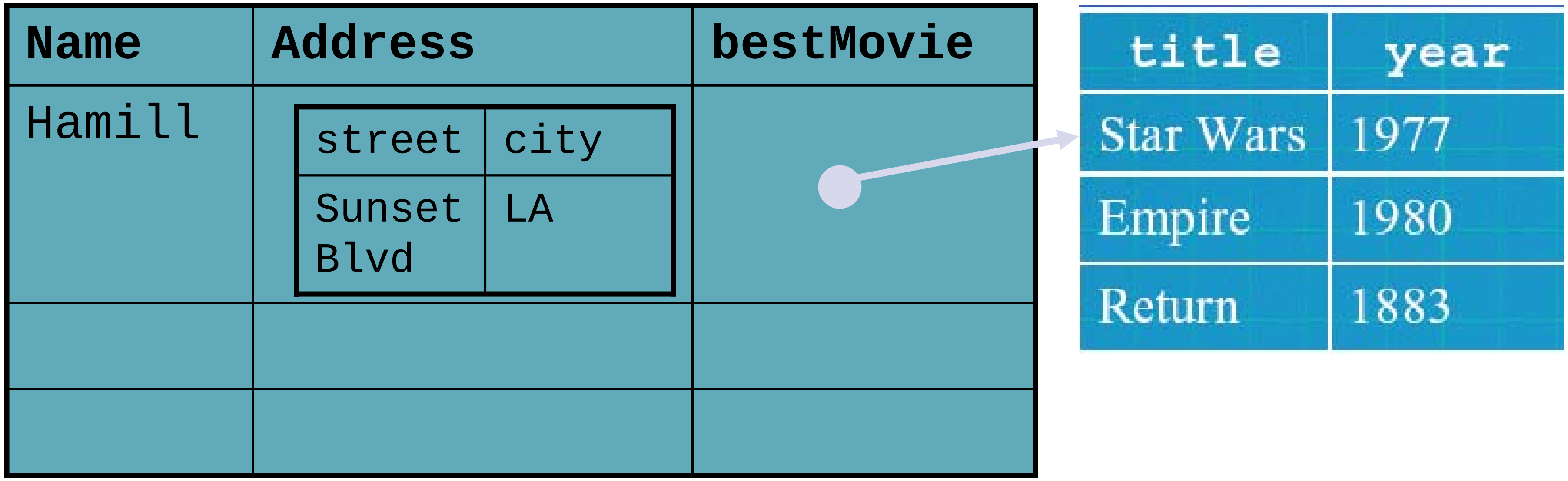
```
REF(T)
```

Reference to tuples in relation *R*, where *R* is a table whose type is the UDT *T*

```
REF(T) SCOPE R
```

References – SQL99 - II – Example

```
CREATE TYPE StarType AS (
   name        CHAR(30),
   address     AddressType,
   bestMovie   REF(MovieType) SCOPE Movie
);
```



Name	Address	bestMovie
Hamill	street city Sunset Blvd LA	●

street	city
Sunset Blvd	LA

title	year
Star Wars	1977
Empire	1980
Return	1883

ORDB Example - Oracle

```
CREATE TYPE Name AS OBJECT (
    first_name CHAR (15),
    last_name CHAR (15),
    middle_initial CHAR (1);
    MEMBER PROCEDURE initialize,;
```

Code to define operations – in this case simply a class constructor

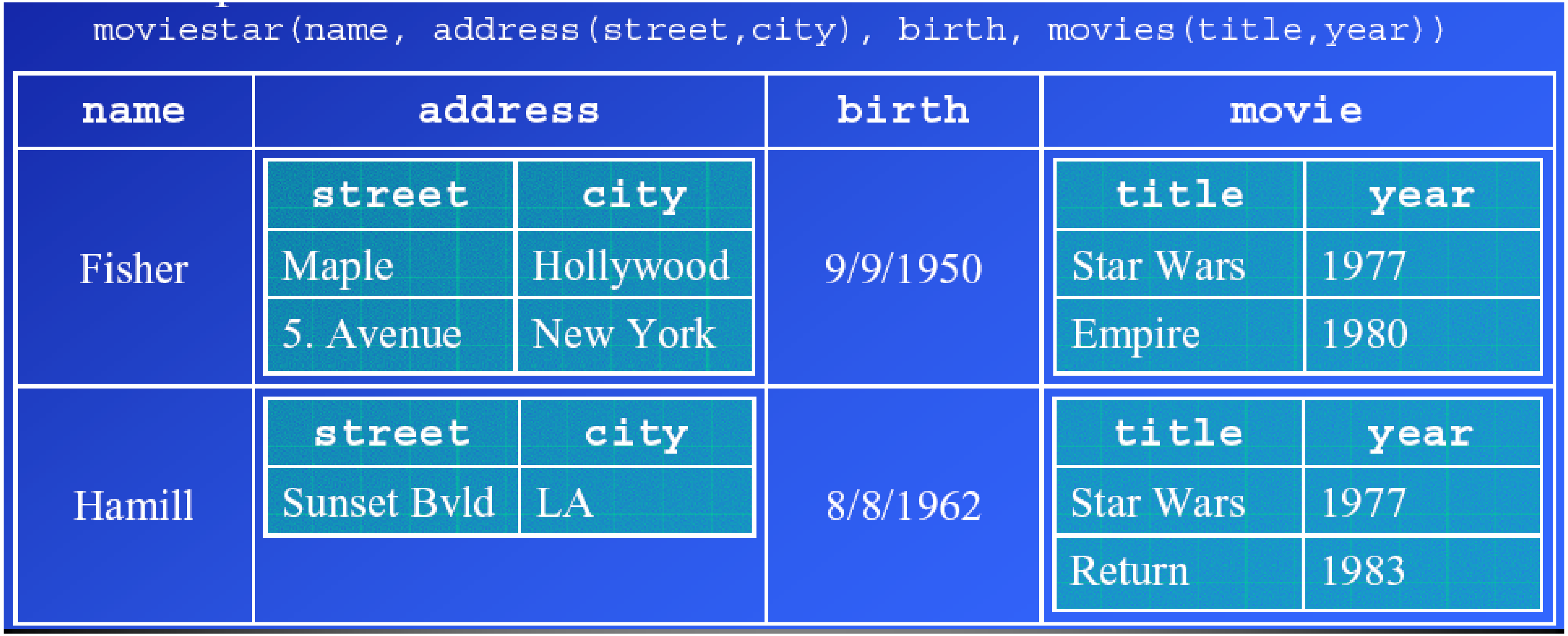
```
CREATE TYPE BODY Name AS
    MEMBER PROCEDURE initialize IS
    BEGIN
        first_name  := NULL;
        last_name := NULL;
        middle_initial := NULL;
    END initialize;
END;
```

Using the new type in a table

```
CREATE TABLE person(
    person_ID NUMBER;
    person_name Name,
    PRIMARY KEY (person_ID));
```

Structured Types - I

- Attributes of relation schemas can be
  - Atomic
  - Relation schemas: Nested Relations

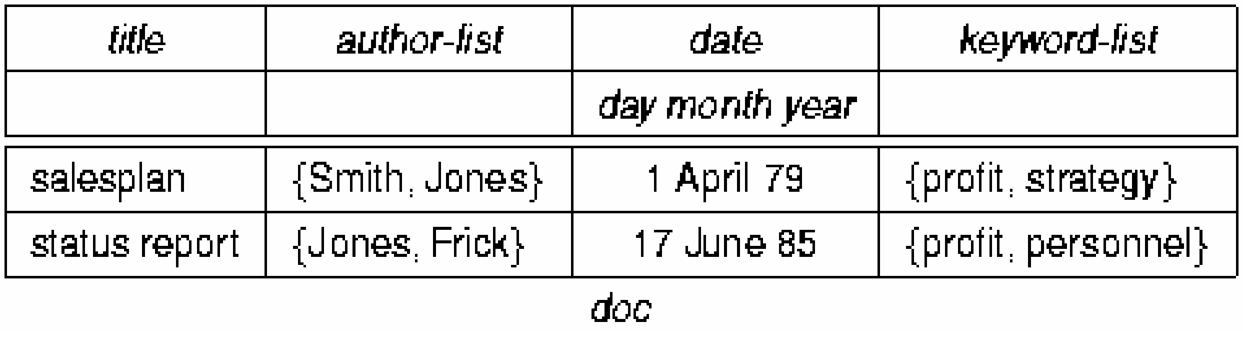


`moviestar(name, address(street,city), birth, movies(title,year))`

name	address		birth	movie	
	street	city		title	year
Fisher	Maple	Hollywood	9/9/1950	Star Wars	1977
	5. Avenue	New York		Empire	1980
	street	city		title	year
Hamill	Sunset Bvld	LA	8/8/1962	Star Wars	1977
				Return	1983

Structured Types - II

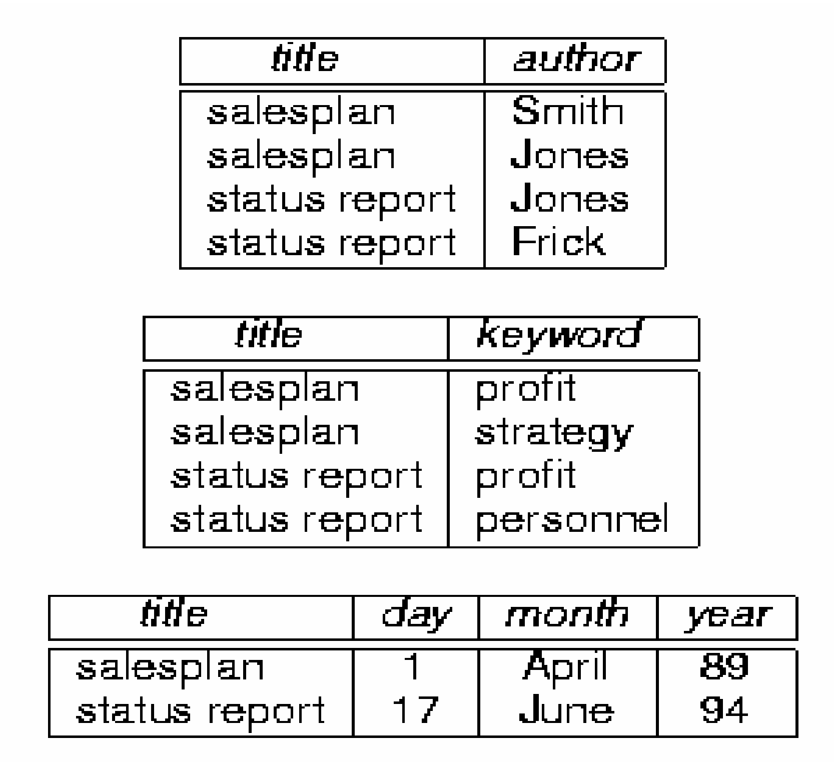
Nested



title	*author-list*	*date*	*keyword-list*
		day month year	
salesplan	{Smith, Jones}	1 April 79	{profit, strategy}
status report	{Jones, Frick}	17 June 85	{profit, personnel}

doc

4NF

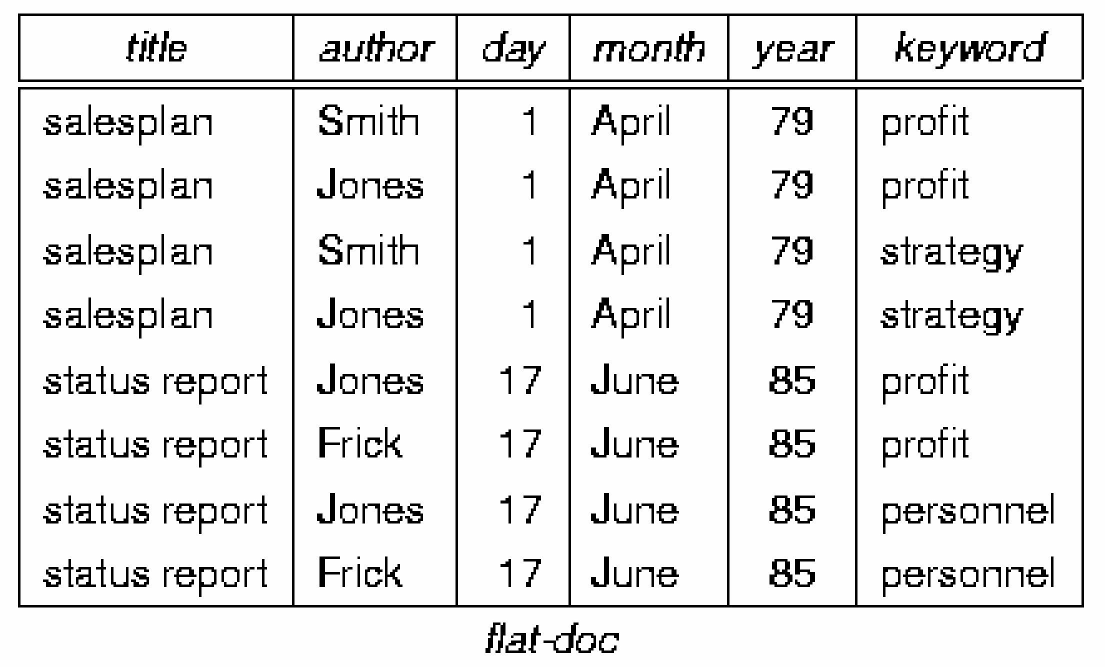


title	*author*
salesplan	Smith
salesplan	Jones
status report	Jones
status report	Frick

title	*keyword*
salesplan	profit
salesplan	strategy
status report	profit
status report	personnel

title	*day*	*month*	*year*
salesplan	1	April	89
status report	17	June	94

1NF

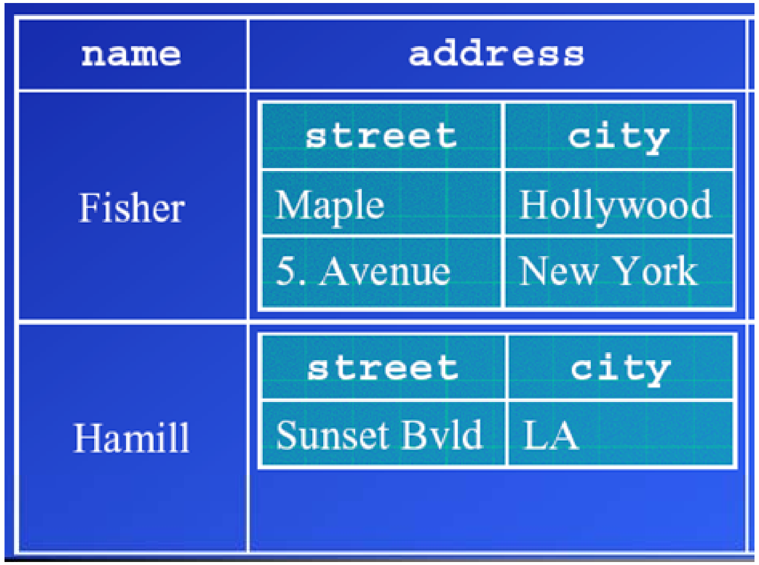


title	*author*	*day*	*month*	*year*	*keyword*
salesplan	Smith	1	April	79	profit
salesplan	Jones	1	April	79	profit
salesplan	Smith	1	April	79	strategy
salesplan	Jones	1	April	79	strategy
status report	Jones	17	June	85	profit
status report	Frick	17	June	85	profit
status report	Jones	17	June	85	personnel
status report	Frick	17	June	85	personnel

flat-doc

Nested Relations – SQL99 Example

```
CREATE TYPE AddressType AS (
   street      CHAR(50),
   city        CHAR(20)
);

CREATE TYPE AddressTypeTable
   AS TABLE OF AddressType;

CREATE TYPE StarType AS (
   name        CHAR(30),
   address     AddressTypeTable
);

CREATE TABLE MovieStar OF StarType;
```



name	address	
	street	city
Fisher	Maple	Hollywood
	5. Avenue	New York
	street	city
Hamill	Sunset Bvld	LA

Methods – SQL99

- Special operations defined for a type
- In SQL, implementation defined with Presistent Stored Modules (PSM) language

```
METHOD m() RETURNS <TYPE>;
```

Methods – SQL99 - Example

```
CREATE TYPE AddressType AS (
   street      CHAR(50),
   city        CHAR(20)
)
METHOD houseNumber() RETURNS CHAR(10);

CREATE METHOD houseNumber() RETURNS CHAR(10) FOR AddressType
BEGIN
  …
END;
```