Advanced Databases

Lectures
November 2015.

3. Object-oriented and object-relational databases

Overview

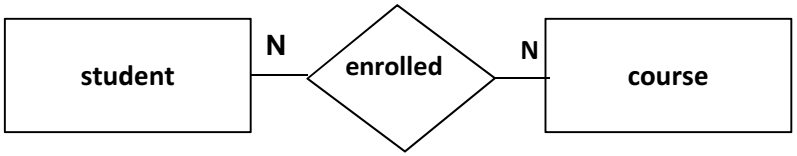
- Object-oriented database
  - The principles of object-oriented database
  - Object-oriented database management systems
  - ODMG standard

- Object-relational database
  - Object-relational data model
  - Object-relational features of SQL Standard
  - Object-relational extensions in PostgreSQL

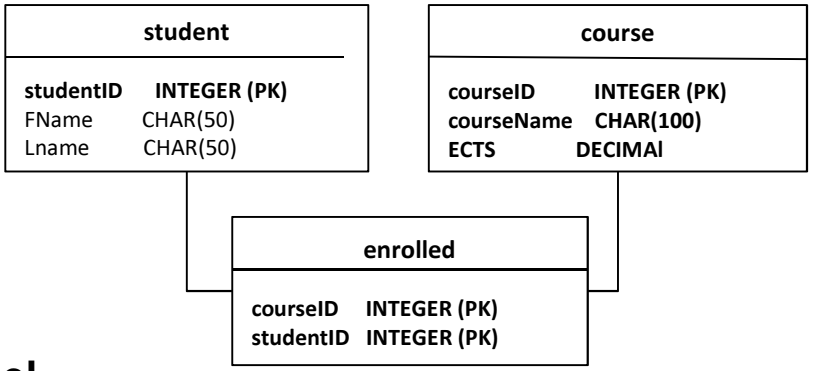
Object-oriented databases - motive (1)

- Relational databases are not suitable for applications that use complex data types or new data types for large unstructured objects (unstructured text, images, multimedia, GIS objects, ...)
- Relational database model is very different from the object model implemented in object-oriented languages (Java, C #)

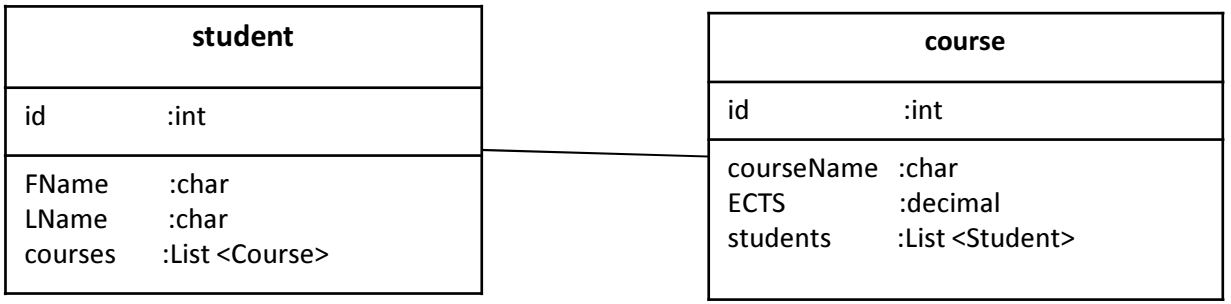
ER model



Relational model



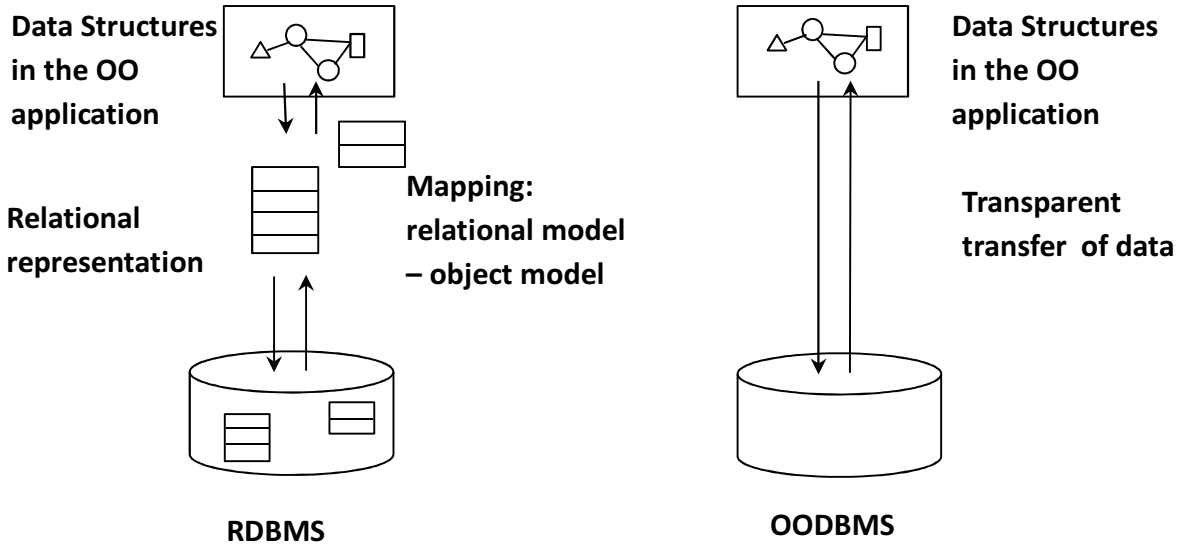
Object model



Object relational impedance mismatch

Object-oriented databases - motive (2)

- Object relational impedance mismatch
- Mapping between the two models is a tedious job – there is a need for transparent handling of data from the relational database, using the paradigm of object-oriented languages



Object-oriented databases - motive (3)

- **Object relational impedance mismatch:**
  - **Relationships between entities**:

    Relational model: relations student, studentCourse, course
    - using primary and foreign keys

    Object model: `student.getCourses()`
    - References to other objects
  - **Querying data**:

    Relational model:
    ```
    SELECT course.courseName
      FROM student, studentCourse, course
     WHERE ......
    ```
    - SQL (DDL, DML)

    Object model: OQL (Object Query Language), SODA (Simple Object Data Access), using object graf (`student.getCourses().get(0).getCourseName()`)
  - **Inheritance is not supported in the relational model**

Object model – relational model

Object-oriented model	Relational model
Class	Relational schema
Object	Entity, tuple
Member variable	Attribute
Method	Procedure
-	Primary key
OID	-

What element(s) from object model suits to relation from relational model?

Object-oriented database

- Object-oriented databases are sometimes called *object databases*
  - In the database are stored objects - the database model does not differ from the one in the application
  - Implementation of object-oriented database management system (OODBMS) is generally programmed for a specific programming language, and differ quite with each other
- *OODBMS is a database management system that implements object-oriented data model*
- *The Object-oriented Database System Manifesto, Atkinson at all,* 1989. – in scientific paper described the properties which OODBMS must satisfy
  - Object-oriented concepts
  - Database management system concepts

The basic principles of the object-oriented database/database management system

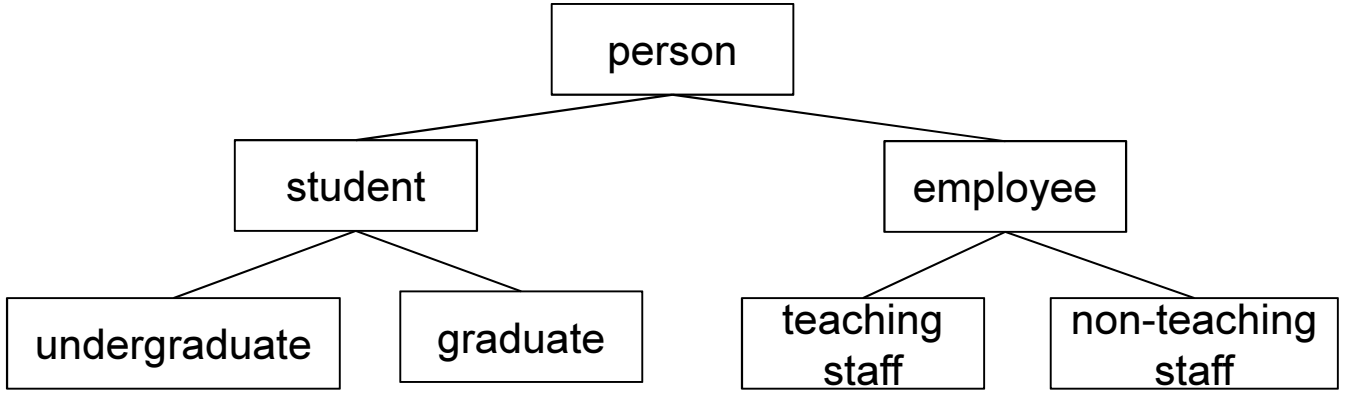
- Object-oriented concepts
  - Classes
  - Complex objects
  - Object identity
  - Class hierarchy
  - Encapsulation
  - Overriding, overloading and late binding
- Database management system concepts
  - Data persistence
  - Physical organisation of the data (*secondary storage management*)
  - Concurrency control
  - Database recovery
  - Ad Hoc Query Facility

Object identity - OID

- Unique, unchangeable object identifier generated by the OO system
- Independent of the values of the object attributes
- Invisible to the user
- Used for referencing objects
- Two objects are identical if they have the same identity of the object - property that uniquely identifies them

- In relational databases
  - the identity of the entity is based on the data values
  - primary key is used to ensure uniqueness
    - primary keys do not provide the kind of unity that is required for the OO systems:
      - keys are unique in the relation, not in the entire database
      - keys are mainly based on the attributes of the relation, which makes them dependent on the state of the object

Class hierarchy




```
public abstract class Person {
    private String personID;
    private String FName;
    private String LName;
     ...
    // access methods and constructors
     ...
}
```

```
public class Student extends Person {
    private String studentIDNumber;
     ...
    // access methods and constructors
     ...
}
public class Employee extends Person {
    private float salary;
     ...
    // access methods and constructors
     ...
}
...
```

- Available attributes and methods are inherited from the parent class
- **Subclasses can define new attributes and methods**
- **Generalization (*person*) and specialization (*undergraduate student*)**

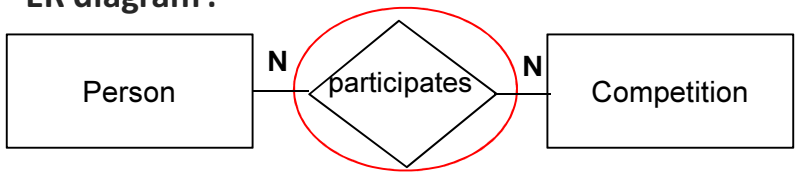
Relationships between objects

- Achieved with **referencing:**
- 1:1 relationship
  - `person.getPassport()`
  - `passport.getPerson()`
- N:1 (1:N) relationship
  - `customer.getResidenceCity()`
  - `city.getCustomers()`
- N:N relationship
  - `customer.getArticles()`
  - `article.getCustomers()`
- All connections can be two-way

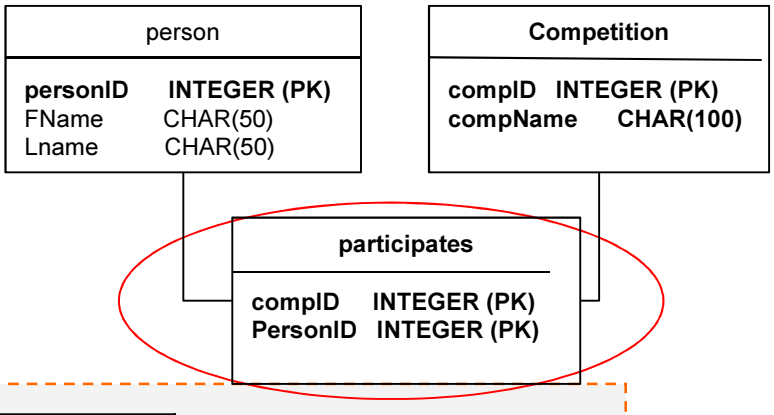
- Mutual references (two way) is not necessary to express in the object model, if it is not important for business process applications

Relationships between objects (2)

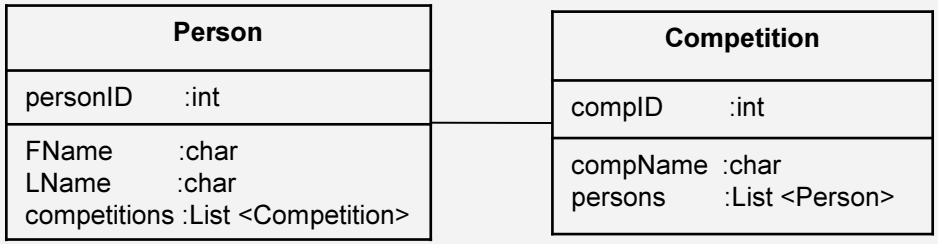
ER diagram :



relational diagram :



Object model



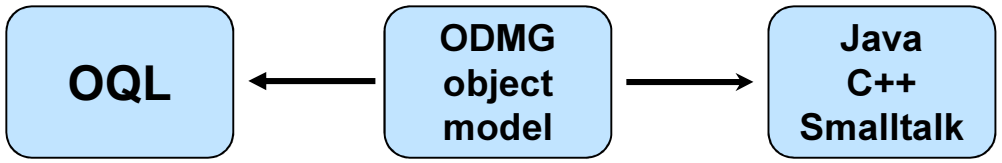
Class definition:

```
public class Person {
    private String personID;
    private String FName;
    private String LName;
    private List<Competition> competitions;
    …
    // access methods and constructors
    …
}
```

```
public class Competition {
    private int compID;
    private String compName;
    private List<Person> persons;
    …
    // access methods and constructors
    …
}
```

ODMG standard

- ODMG standard consists of the following:
  - Object model (OM)
  - Language for specifying objects (ODL - Object Definition Language)
  - Object query language (OQL)
  - Binding between the ODL and the object programming languages
    - Includes ODL which is dependent on the selected programming language
    - Provides an application programming interface (API) for the mapping of data types



OQL - Object Query Language (1)

- Query language for object databases modelled on SQL
- Flexible, but because of its complexity no manufacturer has fully implemented it
- Differences between OQL i SQL:
  - OQL supports referencing to objects within the table. Objects can be nested in other objects.
  - OQL does not support all of the keywords from the SQL
  - OQL supports mathematical calculations within OQL expression
- Syntax:
  - Queries in form using *Select-From-Where*

  or

  - Navigation in complex structures :

    ```
    book.publisher.contact.email
    ```

OQL - Object Query Language (2)

The result of the OQL query is an object whose type depends on the operands involved in the query.

Example 1: Get the names and price of food on offer in the restaurant "Snack"

```
public class Restaurant {
  private Int    IdRestaurant;
  private String name;
  ...
  private List<Dish> dish;
  ...
  //methods & constructors
}
```

```
public class Dish {
  private Int    IdDish;
  private String name;
  private Decimal price;
  ...
  //methods & constructors
}
```

```
public class Sells {
  private Int    IdDish;
  private String name;
  private Decimal price;
 ...
  private Restaurant restaurant;
  private List<Dish> dish;
  ...
 //methods & constructors}
```

```
SELECT s.dish.name, s.dish.price
  FROM Sells s
 WHERE s.restaurant.name = "Snack"
```

OODBMS – advantages

- Better and faster to manage complex objects and relationships in comparison to relational systems
- Support hierarchy, class and inheritance
- Single data model
- Objects in the database and the objects in the application are the same, so there is no need for the two models
  - No object-relational mismatch
- There is no need for a primary key (?)
  - The identification of objects is hidden from the user
- Used only one programming language (application and database access)
- There is no need for a special query language

OODBMS – shortcomings

- No logical data independence
  - Modifications to the database (schema evolution) require changes to the application and vice versa
- Lack of agreed standards, the existing standard (ODMG) is not fully implemented
- Dependence on a single programming language. Typical OODBMS is tied to a single programming language with it's programming interface
- Lack of interoperability with a large number of tools and features that are used in SQL
- Lack of Ad-Hoc queries (queries on the new tables that are obtained by joining new tables with the existing ones)

OODBMS in real world

- Chicago Stock Exchange - management in stocks trade (Versant)
- Radio Computing Services – automation of radio stations (POET)
- Ajou University Medical Canter in South Korea – all functions of the hospital, including those critical such as pathology, laboratory, blood bank, pharmacy and radiology
- CERN – big scientific data sets (Objectivity/DB)
- Federal Aviation Authority – simulation of passengers and baggage
- Electricite de France – management in electric power networks

OOSUBP products

- Versant
- Progress ObjectStore
- Objectivity/DB
- Intersystems Cachè
- POET fastObjects
- **db4o**
- Computer Associates Jasmine
- GemStone

Object-relational databases

Motivating example 1
Presenting geographic data in a relational database

- To determine the position of a point on Earth we use geographical coordinates:
  - latitude
  - longitude
- Distance in km between two geographic coordinates (lat1, long1) and (lat2, long2), taking into account the curvature of the Earth, can be calculated using eg. Haversin formula:

$$a = \sin^2\left(\frac{lat2-lat1}{2}\right) + \cos(lat1) * \cos(lat2) * \sin^2\left(\frac{long2-long1}{2}\right)$$

$$d = R * 2 * atan2\left(\sqrt{a}, \sqrt{1-a}\right) \qquad R = 6371 \text{ the average of the Earth radius}$$

Look at the examples in exercises.

Motivating example 1
Presenting geographic data in a relational database

What do we want from DBMS:

- built-in data type (eg. point) for the presentation of the point on the Earth by means of geographical coordinates
- built-in function to determine
  - the distance between two points on Earth
  - the line connecting two points on Earth
  - the shortest path that connects N points in the order they are listed
  - closed polygon whose vertices are given N points
  - the area of the upper polygon
  - …
- everything works quickly ☺

Traditional relational DBMS do not support none of these:

- A point is a complex data type - consists of 2 elementary data
- The line as a set of points is also a complex data type, as well as polygon

Traditional relational DBMS must support the new complex data types and functions for working with them. Probably the new types of indexing.

Motivating example 1

Presenting and searching documents in a relational database

What do we want from DBMS:

- built-in data type for the presentation of
  - the document as a list of lexemes, together with the positions where it appears in the text *
  - query text as a list of lexemes connected with logical operator & or || **
  - Document with the list of q-grams
- built-in function for
  - converting text in a foreign language to the embedded data type *
  - converting query text in a foreign language to the embedded data type **
  - determining the similarity between the query text and the documents on the basis of morphology, syntax and semantics of the language
  - rank the documents according to similarities between the conditions and documents
  - determine the similarity between the conditions and the documents on the basis of the number of matched q-grams
  - …
- everything works quickly ☺

Motivating example 1
Presenting and searching documents in a relational database

Traditional relational DBMS do not support none of these

- List of lexemes is a complex data type: list of strings, and each string is further linked with list of integers - positions in which string/word appears in the text
- List of q-grams is also complex data type
- Traditional indexing methods (B-tree) would not be appropriate

Traditional relational DBMS must support:

- **the new complex data types and**
- **functions for working with them**
- **new types of indexing**
- **…**

Object-relational DBMS

- **object-relational system** (*object-relational DBMS* - ORDBMS) or *enhanced relational systems*
  - attempt to get the best of relational model and object-oriented approach
  - way of enhancing the capabilities of relational DBMS's with some of the features of object DBMS's, relation is still the fundamental abstraction
  - prototype of object-relational systems: Postgres (PostgreSQL)
  - example of extended commercial DBMS's : **Oracle, IBM** Informix

DBMS Matrix

M. Stonebreaker, D. Moore: Object-relational DBMSs – the next great wave, Morgan Kaufmann, 1996

	simple data	complex data
ad hoc query	relational database	object-relational database
no ad hoc query	file system	object oriented database

Object-relational data model (1)

- Based on the relational data model
  - preserve relational foundations, such as declarative access to data
  - upward compatibility with existing relational languages
- Extend the relational data model by including object orientation and constructs to deal with added data types
  - attributes can contain complex values, including nested relation
  - increase modeling power, increase the range of applications
- no single extended relational model - all models have some concept of "object"

Object-relational data model (2)

- Extensions of relational model:
  - abstract data types (object types, structured user-defined types, ...)
  - object identity and reference types
  - methods for object types, encapsulation
  - user-defined CAST
  - object table, typed tables
  - type and table inheritance
  - nested tables (complex attributes, collection types)

- there is no DBMS's with all of extensions of relational model
  - different DBMS’s use different approaches
  - all major commercial DBMS’s implement some of extensions

SQL Standard: object-relational extensions

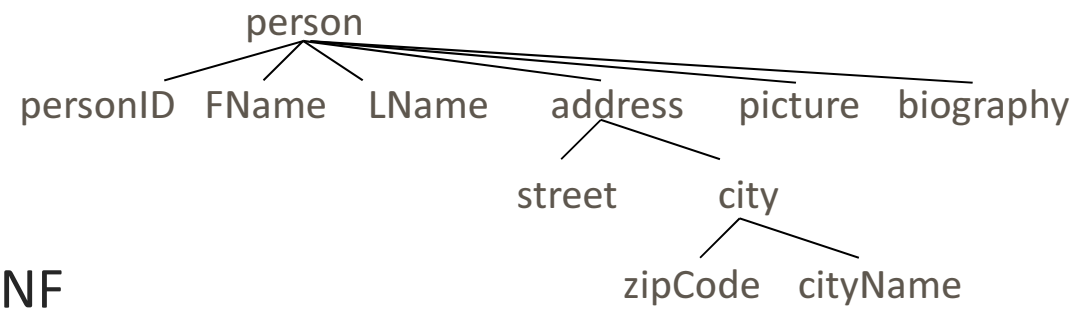
- SQL:1999 includes a number of object-oriented features
  - type constructor for specifying complex objects, object identity, encapsulation, inheritance
- every data type is either:
  - **predefined type**
    - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
  - **constructed type**
    - constructed atomic type
      - *reference*
    - constructed composite type
      - collection: *array*, *multiset*
      - *row*
  - **user-defined type (UDT)**
    - *distinct type*
    - *structured type*

Example: Student administration system

- *person* relation is not in 1NF

person

personID	FName	LName	address			picture	biography
			street	city			
				zipCode	cityName		
11001	Hrvoje	Novak	Ilica 25	10000	Zagreb		Born in …..
78936	Ana	Kolar	Marmontova 18	21000	Split		



- Course relation is not in 1NF

course

courseID	courseName	lectCh	lecturers	students	dep (depID, depName)
1	Advanced databases	123	{123,111, 345,24}	{13503, 14111, 9000, 14678, …}	(DAC, Department for applied computing)

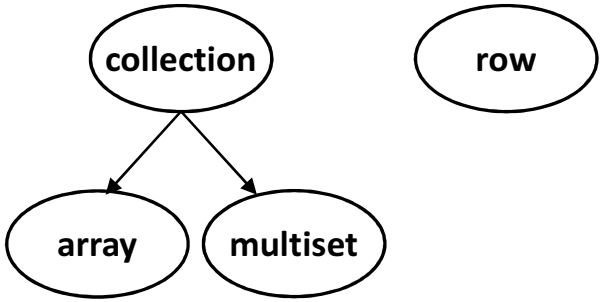
- hierarchy of tables *person*, *lecturer* and *student*

SQL Standard: data types

- predefined type
  - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
- **constructed type**
  - constructed atomic type
    - *reference*
  - constructed composite type
    - collection: *array*, *multiset*
    - *row*
- user-defined type (UDT)
  - *distinct type*
  - *structured type*

SQL Standard: Constructed types

- constructed types:
  - **atomic**
    - *reference type* - **REF** *type*
      - value of REF type references (or points to) some site holding a value of the referenced type
      - the only sites that may be so referenced are the rows of typed tables (every referenced type is a structured type)
  - **composite**
    - data type whose values are composed of values than can all be of the same data type (*collection*) or of different data types (*row*)



- name of a constructed data type is a reserved word specified by standard
- specified by an type constructor (REF, ARRAY, ROW)

SQL Standard: data types

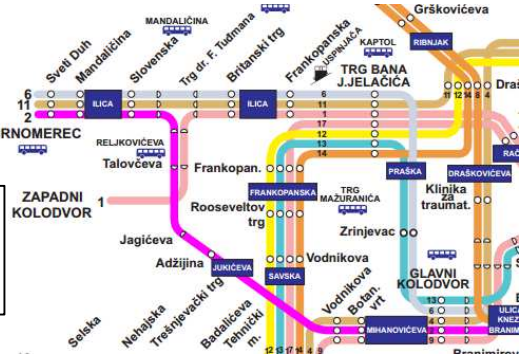
- predefined type
  - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
- **constructed type**
  - constructed atomic type
    - *reference*
  - constructed composite type
    - collection: *array*, *multiset*
    - *row*
- user-defined type (UDT)
  - *distinct type*
  - *structured type*

SQL Standard: Collection

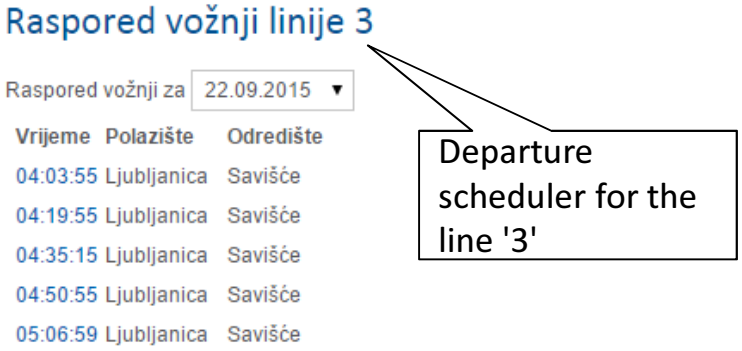
- constructed composite type
- comprises zero or more *elements* of a specified data type (*element type*)
- collection types:
  - ARRAY
    - one-dimensional array with a maximum number of elements
    - SQL:1999 standard
  - MULTISET
    - unordered collection that does allow duplicates
    - SQL:2003 standard
  - SET
    - unordered collection that does not allow duplicates
  - LIST
    - ordered collection that allows duplicates

SQL Standard: ARRAY

- ordered collection of not necessarily distinct values, whose elements are referenced by their ordinal position in the array
- accessing elements of an array by specifying the array index
- index of element ∈ [1, maximum cardinality]



Tram network map in Zagreb



Departure scheduler for the line '3'

```
CREATE TABLE tramRoute
(routeId          INTEGER       PRIMARY KEY,
 routeAbrev       CHAR(3),
 routeName        VARCHAR(120),
 routeStations    INTEGER ARRAY[50] REFERENCES tramStation (stationId),
 departureTimes   TIME ARRAY[300]
);
```

- Index of the array element corresponds with the order of tram stations

- two arrays of comparable element types are equal if they have the same cardinality and equal element value on every position

SQL Standard: MULTISET (1)

- unordered and unlimited collection of elements, with duplicates permitted

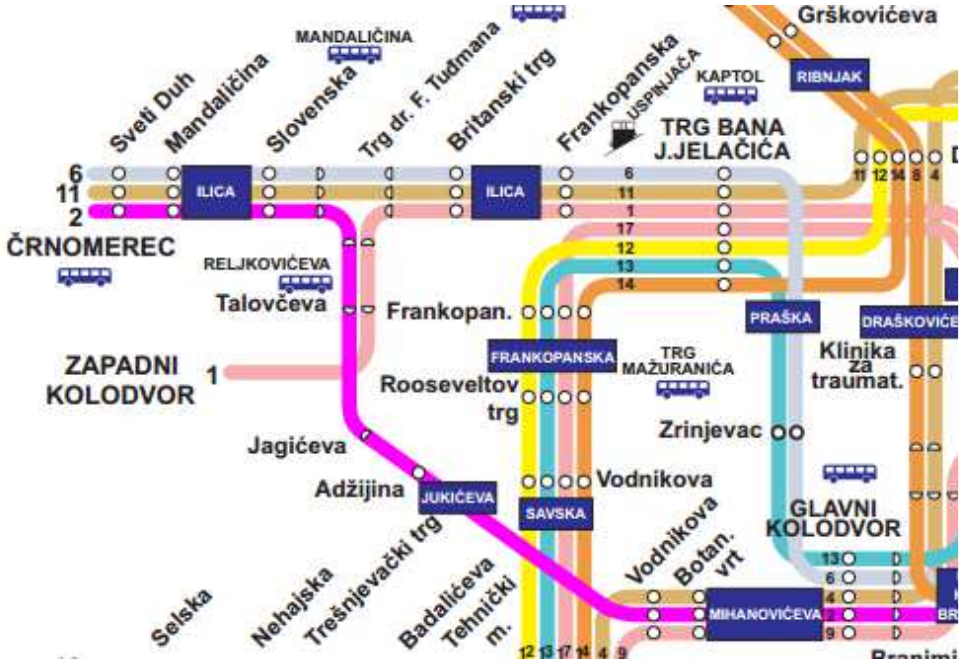
```
CREATE TABLE tramRoute
(routeId          INTEGER       PRIMARY KEY,
 routeAbrev       CHAR(3),
 routeName        VARCHAR(120),
 routeStations    INTEGER ARRAY[50] REFERENCES tramStation (stationId),
 departureTimes   TIME ARRAY[300]
);
```

  - there is no ordinal position to reference individual elements of a multiset

- two multisets of comparable element types are equal if they have the same cardinality and have the same elements, even if the elements are in different positions within the set

PostgreSQL: ARRAY (1)

- While defining attribute of an array type the size need not be specified
  - if the size of the array attribute is stated, the system ignores it




```
CREATE TABLE tramRoute
(routeId         SERIAL        PRIMARY KEY,
 routeAbrev      CHAR(3)       UNIQUE,
 routeName       VARCHAR(120)  UNIQUE,
 routeStations   INTEGER [] );
```

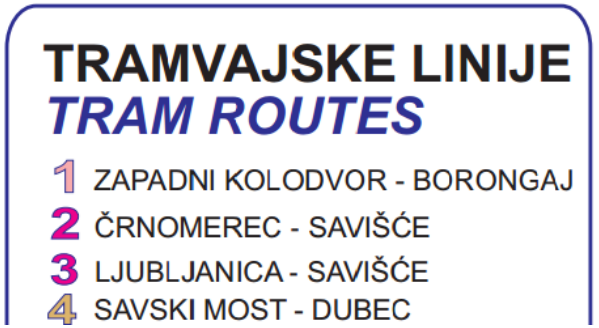
```
...
routeStations INTEGER[]
       REFERENCES tramStation(stationId)
...
```

```
CREATE TABLE tramStation
(stationId   SERIAL PRIMARY KEY,
 stationName VARCHAR(120) NOT NULL
                          UNIQUE);
```

```
INSERT INTO tramStation (stationName)
         VALUES ('Zapadni kolodvor');
INSERT INTO tramStation (stationname)
         VALUES ('Talovčeva');
...
```

- Foreign key (jet) is not possible to define for the elements of the array

PostgreSQL: ARRAY (2)



- Inserting tuple with ARRAY type attribute:

```
INSERT INTO tramRoute (routeAbrev, routeName, routeStations)
 VALUES ('1', 'Zapadni kolodvor - Borongaj',
        '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ,11, 12, 13, 14, 15, 16}')
```

The station order corresponds to the index of array fields

ili :

```
INSERT INTO tramRoute (routeAbrev, routeName, routeStations)
 VALUES ('2', 'Črnomerec - Savišće',
        ARRAY[17, 18, 19, 20, 3 , 2,  21, 22, NULL, NULL, NULL])
```

ili :

```
INSERT INTO tramRoute (routeAbrev, routeName, routeStations)
 VALUES ('3', 'Ljubljanica - Savišće',
        '{}')     /* ili ARRAY[]::integer[] */
```

```
SELECT * FROM tramRoute
```

routeId integer	routeAbrev character(3)	routeName character varying(120)	routeStations integer[ ]
1	1	Zapadni kolodvor – Borongaj	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
2	2	Črnomerec - Savišće	{17, 18, 19, 20, 3 , 2, 21, 22, NULL, NULL, NULL}
3	3	Ljubljanica – Savišće	{}

PostgreSQL: ARRAY (3)

- Inserting tuple with ARRAY type attribute - using SELECT statement to gather elements:

```
INSERT INTO tramRoute (routeAbrev, routeName, routeStations)
 VALUES ('4', 'Savski most - Dubec',
         ARRAY (SELECT stationId FROM tramStation
                  WHERE stationId BETWEEN 56 AND 70)
         )
```

- accessing array elements

```
SELECT routeName,
       routeStations[1] AS first,
       routeStations[2] AS second,
       routeStations[array_length (routeStations, 1)] AS last
  FROM tramRoute
 WHERE routeStations[2] = 2 OR routeStations[1:2] = ARRAY [17,18]
```

Using indeks of the element

array_length (array,dim) – number of elements in the array (for the given dimensiondim)

accessing elements whose indices are within the specified limits

routeName character varying(120)	first integer	second integer	last integer
Zapadni kolodvor – Borongaj	1	2	16
Črnomerec - Savišće	17	18	

PostgreSQL: ARRAY (4)

Operator	Description	Example
=, <>	comparison	ARRAY[1,2,3] = ARRAY[1,2,3] → true ARRAY[1,2,3,4] = ARRAY[1,2,4,3] → false
@>	contains	ARRAY[3,1,4,2] @> ARRAY[2,1] → true ARRAY[3,1,4,2] @> ARRAY[4,4] → true
<@	is contained by	ARRAY[3,2] <@ ARRAY[4,7,1,2,3] → true
&&	overlap (have elements in common)	ARRAY[4,3] && ARRAY[2,1,4] → true ARRAY[4,3] && ARRAY[2,1] → false
||	concatenation	ARRAY[1,2] || 3 → {1,2,3} ARRAY[1,2] || ARRAY[3,4] → {1,2,3,4}

Function	Description	Example
array_length (*array*,*dim*)	the length of the requested array dimension	array_length(ARRAY[1,2,3], 1) → 3
array_lower (*array,dim*) array_upper (*array*,*dim*)	lower/upper bound of the requested array dimension	array_lower(ARRAY[0,1,8], 1) → 1 array_upper(ARRAY[0,1,8], 1) → 3
array_append (*array*, *el*) array_prepend (*el*, *array*)	append an element to the end/beginning of an array	array_append(ARRAY[1,2], 3) → {1,2,3} array_prepend(3, ARRAY[1,2]) → {3,1,2}
array_remove (*array*, *el*)	remove all elements equal to the given value from the array (array must be one-dimensional)	array_remove(ARRAY[3,1,3,2],3) → {1,2}

Motivating example: unnesting

For the tram route with the abbreviation „1” list the ordinal and the names of tram stations. Sort them according to the station order (desc).

tramRoute

routeId integer	routeAbrev character(3)	routeName character varying(120)	routeStations integer[ ]
1	1	Zapadni kolodvor – Borongaj	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
2	2	Črnomerec - Savišće	{17, 18, 19, 20, 3 , 2, 21, 22, NULL, NULL, NULL}
3	3	Ljubljanica – Savišće	{}

tramStation

stationId integer	stationName varchar(120)

- Tram stations are stored in the collection ***routeStations INTEGER[ ]***.
- The station name must be obtained by joining the value of each element of an array with relation ***tramStation***.
- Sorting should be done according to the index of array elements.

How it can be done using SQL?

Motivating example: unnesting

For the tram route with the abbreviation „1” list the ordinal and the names of tram stations. Sort them according to the station order (desc).

In relational model collection ***routeStations*** would be presented with relation ***routeStation***.

tramRoute

routeId	routeAbrev	routeName

tramStation

stationid	stationName

routeStation

routeId	stationId	stationOrd

The solution is trivial:

```
SELECT stationOrd, stationName
  FROM tramStation
  JOIN routeStation ON routeStation.stationId = tramStation.stationId
  JOIN tramRoute    ON tramRoute.routeId      = routeStation.routeId
 WHERE routeAbrev = '1'
ORDER BY stationOrd
```

The idea: transform a collection into "temporary" table and use the SQL in the usual way.

PostgreSQL: unnesting

Function	Description	Example
unnest (array) [WITH ORDINALITY] [AS] table_alias [(column_alias [, ...])]]	Expand an array to a set of rows. If the WITH ORDINALITY clause is specified, an additional column of type bigint will be added to the function result columns. This column numbers the rows of the function result set, starting from 1.	`unnest(ARRAY[3,2])` → two tuples: 3 2

tramRoute

routeId	routeAbrev	routeName	routeStations Integer[ ]

tramStation

stationid	stationName

```
SELECT routeName,
       UNNEST(routeStations) AS stationId
  FROM tramRoute
WHERE routeAbrev = '1'
```

or

```
SELECT routeName,
       routeStation.stationId
  FROM tramRoute,
  UNNEST(routeStations) AS routeStation (stationId)
WHERE routeAbrev = '1'
```

routeName	stationId
Zapadni kolodvor – Borongaj	1
Zapadni kolodvor – Borongaj	2
…	…
Zapadni kolodvor – Borongaj	16

PostgreSQL: unnesting

For the tram route with the abbreviation „1” list the ordinal and the names of tram stations. Sort them according to the station order (desc).

tramRoute

routeId	routeAbrev	routeName	routeStations Integer[ ]

tramStation

stationid	stationName

```
SELECT routeStation.stationOrd,
       tramStation.stationName
  FROM tramRoute,
       UNNEST(tramRoute.routeStations) WITH ORDINALITY
           AS routeStation (stationId, stationOrd),
       tramStation
 WHERE tramStation.stationId = routeStation.stationId
   AND routeAbrev = '1'
ORDER BY stationOrd DESC
```

stationOrd	stationName
16	Borongaj
15	Svetice
14	Harambašićeva
…	…
2	Talovčeva
1	Zapadni kolodvor

Motivating example: nesting

Expand the table ***tramStation*** with the attribute ***tramRoutes*** - collection storing the names of all the route lines that pass through the station. Update the value of the attribute in accordance with the data stored in the tramRoute table.

tramStation

stationId	stationName	tramRoutes character varying(120) [ ]
1	Zapadni kolodvor	{"Zapadni kolodvor - Borongaj"}
2	Talovčeva	{"Zapadni kolodvor - Borongaj", "Črnomerec - Savišće"}
…	…	

```
ALTER TABLE tramStation ADD tramRoutes VARCHAR(120)[];
```

```
UPDATE tramStation
   SET tramRoutes =
  (SELECT routeName
     FROM tramRoute
    WHERE tramRoute.routeStations ... tramStation.stationId)
```

3. Convert into array of strings

2. Convert into element of integer array

1. operator „contains”

1. Operator „contains” is @>

2. Ineger array can be obtained with ARRAY [sifPostaja]

```
SELECT ARRAY[stationId]
 FROM tramStation
```

array Integer [ ]
{1}
{2}
…

Motivating example: nesting

```
UPDATE tramStation
   SET tramRoutes =
  (SELECT routeName
     FROM tramRoute
    WHERE tramRoute.routeStations ... tramStation.stationId)
```

3. Convert into array of strings

3. Array can be obtained using ARRAY

```
SELECT ARRAY[routeName]
  FROM tramRoute
 WHERE routeStations @> ARRAY[3]
```

array bpchar [ ]
{"Zapadni kolodvor - Borongaj"}
{"Črnomerec - Savišće"}

I need one tuple , not 2 or more

array character varying(120) [ ]
{"Zapadni kolodvor - Borongaj", "Črnomerec - Savišće"

```
SELECT tramStation.*,
       (SELECT ARRAY[routeName]
          FROM tramRoute
         WHERE routeStations @> ARRAY[stationId])
  FROM tramStation
 WHERE stationId = 3
```

Correlated subquery

```
ERROR:  more than one row returned by a subquery used as an expression
```

I need "something" (eg. function) that will transform multiple separate elements fields into a collection.

SQL standard: nesting

- nesting - transformation of unnested relation into nested relation
  - Using grouping - function COLLECT


```
SELECT routeId, routeName,
       COLLECT(stationId) AS tramStations
  FROM tramRoute, routeStation
 WHERE tramRoute.routeId = routeStation.routeId
GROUP BY routeId, routeName
```

  - Using subqueries

```
SELECT routeId, routeName,
       ARRAY(SELECT stationId
               FROM routeStation
              WHERE tramRoute.routeId = routeStation.routeId)
             AS routeStations
  FROM tramRoute
```

PostgreSQL: nesting

Function	Description
array_agg (*expression*)	input values, including nulls, concatenated into an array

```
UPDATE tramStation
  SET tramRoutes =
 (SELECT array_agg (routeName)
    FROM tramRoute
   WHERE tramRoute.routeStations @> ARRAY[stationId])
```

```
SELECT *
  FROM tramStation;
```

stationId	stationName	tramRoutes character varying(120) [ ]
1	Zapadni kolodvor	{"Zapadni kolodvor - Borongaj"}
2	Talovčeva	{"Zapadni kolodvor - Borongaj", "Črnomerec - Savišće"}

The result of the expression (array_agg argument) can not be eg. tuple. Therefore, using the array_agg it is not possible to create a collection of more than one dimension.

```
SELECT ARRAY_agg[routeAbrev, routeName]
  FROM tramRoute
 WHERE routeStations @> ARRAY[3]
```

```
ERROR:  syntax error at or near "," LINE 1: SELECT ARRAY_agg[routeAbrev, routeName]
```

Is it possible that PostgreSQL has so modest support for nesting?

No. Elements of the collection can be of a complex type. What is a composite/complex type - later.

PostgreSQL: indexing ARRAY attributes

- index on attributes of array type is used for operations: &&, @>, @@ ,…
  - GIST index (*Generalized Search Tree*) : gist__int_ops - for small data sets (default), gist__intbig_ops
  - GIN index (*Generalized Inverted Index*)

```
CREATE INDEX routeStIdx ON tramRoute USING GIST (routeStations gist__int_ops);
```

```
CREATE INDEX routeStIdx ON tramRoute USING GIN (routeStations);
```

Examples

- ensure that the grade stored in the array type attribute, can take on only integer values between 1 and 5

```
CREATE TABLE examResult(
   teacherId integer,
   grades    integer[] CHECK (grades <@ ARRAY[1,2,3,4,5]) );
```

- ensure not more than 3 marks

```
... CHECK (icount(grades) <= 3)
```

SQL Standard: data types

- predefined type
  - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
- **constructed type**
  - constructed atomic type
    - *reference*
  - constructed composite type
    - *collection: array, multiset*
    - row
- user-defined type (UDT)
  - *distinct type*
  - *structured type*

SQL Standard: ROW type

- constructed composite type
- sequence of one or more (*field name*, *data type*) pairs, called *fields*
  - *degree* - number of elements

```
ROW ( zipCode   INTEGER,
      cityName  VARCHAR(20)
    )
```

- ROW constructor is used to create an instance of a ROW data type
- value of ROW element can be literal, and also result of query
- eg. above ROW type can be assigned value (10000,'Zagreb') with the following construct:

```
ROW(10000, 'Zagreb')
```

SQL standard: ROW type

ROW type can be used to define complex attributes in a table:

person

personId	FName	LName	address		
			street	city	
				zipCode	cityName
11001	Hrvoje	Novak	Ilica 25	10000	Zagreb
78936	Ana	Kolar	Marmontova 18	21000	Split

```
CREATE TABLE person(
  personId  INTEGER,
  FName     VARCHAR(25),
  LName     VARCHAR(25),
  address   ROW ( street VARCHAR(50),
                  city ROW (zipCode    INTEGER,
                            cityName VARCHAR(40))
                )
);
```

PostgreSQL: ROW constructor

In PostgreSQL ROW is constructor for instantiation of a composite type value

```
ROW (literal, ...)
```

```
ROW ('HR', 'Hrvatska')
```

or

```
'( val1 , val2 , ... )'
```

```
'("HR", "Hrvatska")'
```

- allowed to omit ROW if it consists of multiple attributes
  http://www.postgresql.org/docs/9.4/static/rowtypes.html

CREATE TABLE syntax from the previous slide is not supported in PostgreSQL.

As a constructor, ROW is used with user-defined type - complex data type (discussed later).

SQL Standard: data types

- predefined type
  - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
- constructed type
  - constructed atomic type
    - *reference*
  - constructed composite type
    - *collection: array, multiset*
    - row
- **user-defined type (UDT)**
  - distinct type
  - structured type

SQL Standard: User-defined types

- User-defined types (UDT)
  - the name of a user-defined type is provided in its definition
  - persistent
  - sometimes called *abstract data types*

  (1) **distinct type**
    - based on a single predefined data type called *source type*
    - type inheritance is not allowed

  (2) **structured type**
    - expressed as a list of attributes
    - type inheritance is allowed

SQL Standard: DISTINCT type

- user-defined type - more limited variant
- based on some predefined type (source type)
- renamed type, with different behavior than its source type
  - allows differentiation between the same underlying base types
- methods can be defined over DISTINCT types
- type inheritance is not allowed (FINAL means - subtypes are not allowed)

```
CREATE TYPE heightT AS DECIMAL (5,2) FINAL;
CREATE TYPE weightT AS DECIMAL (5,2) FINAL;
```

```
CREATE TABLE person(
  personId    INTEGER PRIMARY KEY,
  LName       VARCHAR(25),
  height      heightT,
  weight      weightT);
```

- values cannot be directly mixed in operations with source type or other distinct types
- using CAST function is needed

```
SELECT personId,
       weight/height/height AS BMI,
      'Overweight'
  FROM person
 WHERE weight > height*height*30;
```

ERROR

```
SELECT personId,
       CAST(weight AS DECIMAL (5,2))/
       CAST((height*height) AS DECIMAL (5,2)) AS BMI,
      'Overweight'
  FROM person
 WHERE CAST(weight          AS DECIMAL (5,2)) >
       CAST((height*height) AS DECIMAL (5,2)) *
       CAST(30              AS DECIMAL (5,2));
```

BMI = weight/height$^{2;}$ <20 Underweight; >30 Overweight

PostgreSQL: DISTINCT

- CREATE DOMAIN - scalar type based on a built-in data type - alias for embedded data type with the possibility of specifying DEFAULT NULL and CHECK constraints

Simplified syntax:

```
CREATE DOMAIN domName [AS] dataType
   [ DEFAULT expression ][ NOT NULL ]
   [ CHECK expression ]
```

Example:

```
CREATE DOMAIN dCelsius AS DECIMAL(4,1)
   CHECK (VALUE BETWEEN -100 AND 100);

CREATE TABLE euroTemp
(dateTemp DATE,
 cityName CHAR(20),
 temp     dCelsius);

INSERT INTO euroTemp VALUES
('03.01.2015', 'Rovaniemi', -20.0);
```

```
CREATE DOMAIN dFahrenheit AS DECIMAL(4,1)
    CHECK (VALUE BETWEEN -140 AND 210);

CREATE TABLE USATemp
(dateTemp DATE,
 cityName CHAR(20),
 temp     dFahrenheit);

INSERT INTO USATemp VALUES
('03.01.2015', 'Anchorage', -20.0);
```

```
SELECT USAtemp.dateTemp,
       euroTemp.cityname euro,
       USATemp.cityName usa,
euroTemp.temp tempC, USATemp.temp tempF
  FROM USAtemp, euroTemp
 WHERE USAtemp.temp = euroTemp.temp
   AND USAtemp.dateTemp = euroTemp.dateTemp
```

dateTemp	euro	usa	tempC numeric(4,1)	tempF numeric(4,1)
03.01.2015	Rovaniemi	Anchorage	-20	-20

Not in accordance with the SQL standard.
An implicit conversion is conducted into built-in type.
+ result does not make sense!

SQL Standard: Structured types

- named, user-defined data type
- composed of one or more *attributes*
  - attribute has name and data type
    - example: definition of *cityT* structured type

```
CREATE TYPE cityT AS (
        zipCode     INTEGER,
        cityName  VARCHAR(40))
  NOT FINAL;
```

- **NOT FINAL** - creating of subtypes of *cityT* type is allowed

- structured type can be used as:
  - type of attributes of other structured types
  - type of parameters of functions, methods, and procedures
  - type of SQL variables
  - type of columns in tables and type of table rows

SQL standard: Typed tables

- ***typed table*** - table that is declared to be based on some structured type
  - contains columns that correspond to the associated type's attributes
  - additional column - *self-referencing column* - value that uniquely identifies the row in which it is stored
    - rows in different typed tables may have equal values in their self-referencing column
- Example:

```
CREATE TYPE cityT AS
(   zipCode    INTEGER,
    cityName   VARCHAR(40)
 ) NOT FINAL
    REF IS SYSTEM GENERATED;
```

Way of generating *object reference* - REF IS clause

```
CREATE TABLE city OF cityT
(PRIMARY KEY (zipCode),
 REF IS cityID SYSTEM GENERATED);
```

```
INSERT INTO city VALUES
(10000, 'Zagreb');
```

cityID	zipCode	cityName
1023456734	10000	Zagreb

- REF IS clause :
  - SYSTEM GENERATED - automatically generated by system
  - USING - user generated
  - FROM - derived from one or more attributes of the structured type

PostgreSQL: *Composite Type*

- structure consisting of attributes (elements, fields) that can be distinguished by type
  - For the attribute, name and data type should be stated; integrity constraint cen not be defined (eg. NOT NULL, CHECK, DEFAULT)
    - attributes can be of any type (including other complex types and ARRAY)

- CREATE TYPE statement

```
CREATE TYPE typeName AS ( atribName atribType [, ... ] ] )
```

eg.:

```
CREATE TYPE stateT AS (stateCode CHAR(2)
                     , stateName VARCHAR(20));
```

  - automatically when creating the table

```
CREATE TABLE state (stateCode CHAR(2)
                  , statename VARCHAR(20));
```

→ Type *state* is created

PostgreSQL: *Composite Type*

- Created type can be used while creating:
  - another type

```
CREATE TYPE stateT AS
 (stateCode    CHAR(2),
  stateName
VARCHAR(20));
```

```
CREATE TYPE cityT AS
 (zipCode    INTEGER,
  cityName   VARCHAR(20),
  state      stateT);
```

  - relation

```
CREATE TABLE firm (firmName CHAR(20),
                   residence cityT);
```

- Tuple insert:

```
INSERT INTO firm
     VALUES ( 'INA',
              ROW ( 10000, 'Zagreb',
                    ROW ('HR', 'Hrvatska')
                  )
            );
```

ROW key word may be omitted when the complex type has more than one attribute

- accessing attributes of composite type - *dot* notation
  - the name of the composite type attribute must be enclosed in parentheses

```
SELECT firm.residence,
       (firm.residence).zipCode,
       (residence).state,
       (residence).state.stateName
  FROM firm;
```

residence cityT	zipCode integer	state stateT	state varch
(10000,Zagreb,"(HR,Hrvatska)")	10000	(HR,Hrvatska)	Hrv

Motivating example – referencing from structured type

city

zipCode	cityName
10000	Zagreb
21000	Split

person

personId	fName	LName	Address	
			street	zipCode?? cityREF??
11001	Hrvoje	Novak	Ilica 25	
78936	Ana	Kolar	Marmontova 18	

- I don't want to store all city attribute values in address attribute in person. I want to store the value that would allow me to easily access attributes from the city table.
- In relational model similar requirement is solved using foreign key.
- SQL standard provides REF type for this problem

```
CREATE TYPE cityT AS
(  zipCode      INTEGER,
   cityName   VARCHAR(40)
 ) NOT FINAL;
```

```
CREATE TABLE city OF cityT
(PRIMARY KEY (zipCode)
 REF IS cityID SYSTEM GENERATED)
```

```
CREATE TYPE addressT AS
(  street      VARCHAR(40),
   city        REF(cityT)
) NOT FINAL;
```

```
CREATE TYPE personT AS
(  personId      INTEGER,
   FName         VARCHAR(25)
   LName         VARCHAR(25)
   address       addressT)
 ) NOT FINAL;
```

```
CREATE TABLE person OF personT
(PRIMARY KEY (personId)
 REF IS personID SYSTEM GENERATED)
```

Connection with objects of *cityT type* is established using REF type

SQL standard: REF type

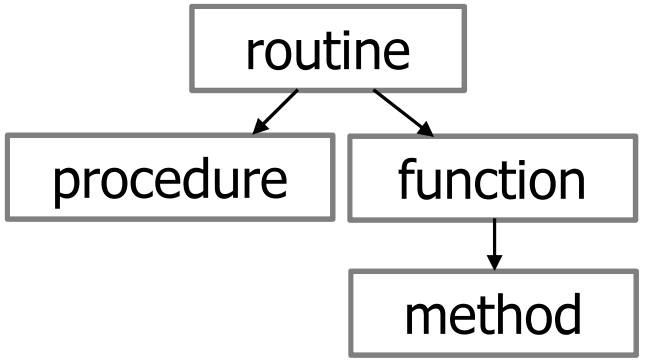
- value of REF type references (or points to) some site holding a value of the referenced type
  - if T is a type, then REF T is the type of a reference to T, that is, a pointer to an object of type T
- pointers only to rows in typed tables
- can be used as type of:
  - columns in ordinary SQL tables
  - attributes of structured types
  - SQL variables, parameters

personId	FName	LName	address	
			street	zipCode?? cityREF??
11001	Hrvoje	Novak	Ilica 25	

REF type is **not implemented** in PostgreSQL.
Oracle has REF type.

```
INSERT INTO person (personId, FName, LName, address)
           VALUES (11001, 'Hrvoje', 'Novak',
                     ROW ('Ilica 25', (SELECT cityId
                                         FROM city
                                        WHERE zipCode = 21000)
                          )
                  );
```

SQL-invoked routines



- SQL-invoked routines - routine that is invoked from SQL code
- three principle classes of SQL-invoked routine:
  - **Function:**
    - only input parameters (return a single value as the "value" of a function invocation)
    - can be called using function notation: **`functionName (parameters)`**
  - **Method:**
    - special sort of function - function that is closely associated with a single structured type
    - every method always has one implicit parameter, whose data type must be the associated type
  - **Procedure:**
    - input and output parameters
    - typically invoked using some form of CALL statement

PostgreSQL: Function with composite type

Example

```
CREATE TABLE state (stateCode CHAR(2) PRIMARY KEY
                  , stateName VARCHAR(20));

CREATE TABLE city (zipCode    INTEGER,
                   cityName   VARCHAR(20),
                   stateCode  CHAR(2)
                   REFERENCES state(stateCode));

INSERT INTo state VALUES ('HR', 'Hrvatska');
INSERT INTo city  VALUES (10000, 'Zagreb', 'HR');
```

```
CREATE OR REPLACE FUNCTION ctateOfTheCity(city) RETURNS state AS $$
    SELECT * FROM state
      WHERE stateCode = $1.stateCode;
$$ LANGUAGE SQL;
```

```
SELECT m.*, ctateOfTheCity(m) FROM city m;

Or function notation:

SELECT m.*, m.ctateOfTheCity  FROM city m;
```

zipCode integer	cityName varchar(20)	stateCode char(2)	stateT state
10000	Zagreb	HR	(HR,Hrvatska)

PostgreSQL: Function with composite type

Example

```
CREATE TABLE state (stateCode CHAR(2) PRIMARY KEY
                    , stateName VARCHAR(20));
CREATE TABLE city (zipCodeINTEGER,
                   cityName VARCHAR(20),
                   stateCode CHAR(2)
                   REFERENCES state(stateCode));
```

```
CREATE FUNCTION city(int)
RETURNS city AS $$
    SELECT * FROM city
      WHERE zipCode = $1;
$$ LANGUAGE SQL;
```

```
CREATE FUNCTION city(text)
RETURNS city AS $$
  SELECT * FROM city
   WHERE cityName = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM city('Zagreb');
SELECT * FROM city('10000'::int);
```

```
SELECT * FROM city('10000');
```

zipCode integer	cityName varchar(20)	stateCode char(2)
10000	Zagreb	HR

zipCode integer	cityName varchar(20)	stateCode char(2)

- Conversion of the integer type into *city type* eg. 10000 into **(10000,Zagreb,HR)**

```
CREATE CAST (int AS city)
  WITH FUNCTION city(int);
```

```
SELECT 10000::city
or
SELECT CAST(10000 AS city)
```

city city
(10000, Zagreb, HR)

PostgreSQL: user defined CAST (for DOMAIN and composite type)

°C = (°F -32) / 1.8
°F = °C *1.8 + 32

Example

```
CREATE DOMAIN dCelsius
    AS DECIMAL(4,1)
 CHECK (VALUE BETWEEN -100 AND 100);
```

```
CREATE DOMAIN dFahren
    AS DECIMAL(4,1)
 CHECK (VALUE BETWEEN -140 AND 210);
```

Enable conversion from Celsius to Fahrenheit and vice versa on the DBMS level => CAST

```
CREATE FUNCTION FToC(dFahren)
    RETURNS dCelsius AS $$
    SELECT CAST(($1-32)/1.8 AS dCelsius);
$$ LANGUAGE SQL;
```

```
CREATE FUNCTION CToF (dCelsius)
  RETURNS dFahren AS $$
  SELECT CAST(($1*1.8 +32) AS dFahren);
$$ LANGUAGE SQL;
```

```
CREATE CAST (dFahren AS dCelsius)
  WITH FUNCTION FToC(dFahren);
```

```
CREATE CAST (dCelsius AS dFahren)
  WITH FUNCTION CToF(dCelsius);
```

```
WARNING:  cast will be ignored because the source data type is a domain
```

```
SELECT FtoC(32), CAST(CAST (32 AS dFahren) AS dCelsius)
```

ftoc numeric(4,1)	dcelsius numeric(4,1)
0	32.0

Conclusion: for DOMAIN type in PostgreSQL user defined CAST can not be obtained.

PostgreSQL: CAST for composite type

Example

```
CREATE TYPE tFahren AS
(temperature DECIMAL(4,1) );
```

°C = (°F -32) / 1.8

°F = °C *1.8 + 32

```
CREATE TYPE tCelsius AS
(temperature DECIMAL(4,1) );
```

Enable conversion from Celsius to Fahrenheit and vice versa on the DBMS level => CAST

```
CREATE OR REPLACE FUNCTION FToCType (tFahren)
RETURNS tCelsius AS $$
    SELECT CAST( ROW (($1.temperature-32)/1.8) AS tCelsius);
$$ LANGUAGE SQL;
```

```
CREATE CAST (tFahren AS tCelsius)
  WITH FUNCTION FToCType(tFahren);
```

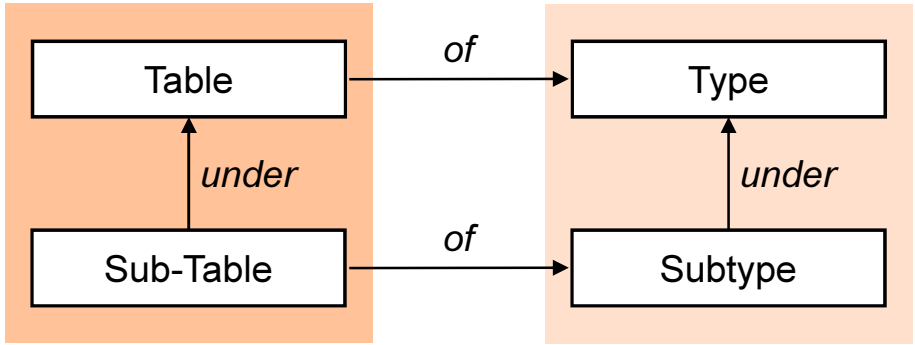
```
SELECT FToCType((ROW(32))::tFahren),
       (ROW (32) :: tFahren) :: tCelsius
```

FToCType tcelsius	row tcelsius
(0.0)	(0.0)

Conclusion: for composite type , user defined CAST can be obtained.
To convert Celsius to Fahrenheit and vice versa, composite type is not suitable as a data such as the temperature is too simple to be stored in a composite type.
Better example for CAST on composite type can be found in exercises.

SQL standard: Inheritance

- type inheritance
  - allowed only for structured types
  - type hierarchy
- table inheritance
  - allowed only for typed tables
  - table hierarchy
  - analogy with specialization/generalization in the E-R model
  - allows multiple types of the same object and simultaneous existence of the same entity in more than one table



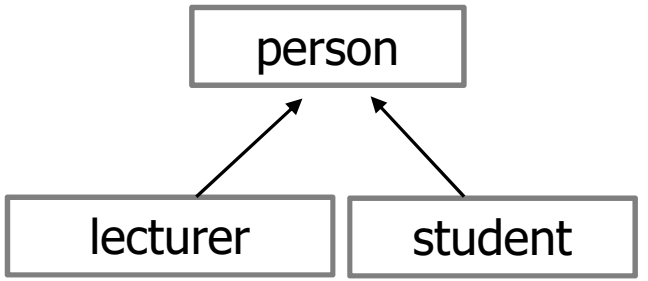
PostgreSQL: Inheritance

- inheritance is enabled only for (base) tables (not for composite types)
  - can be inherited:
    - Attribute definitions (name, type, NULL constraint)
    - *default* attribute values
    - CHECK constraints (can not be overridden in the child-tables)
    - table methods
  - can not be inherited:
    - indexes
    - UNIQUE, PRIMARY KEY, FOREIGN KEY
    - triggers
- the multiple inheritance is allowed

PostgreSQL: inheritance

Example:

```
CREATE TABLE person (
    personId INTEGER PRIMARY KEY,
    FName    VARCHAR(25),
    LName    VARCHAR(25)) ;
```



student and lecturer inherits attributes *personID, LName and FName*

from *person*

```
CREATE TABLE lecturer(
 accountNo    CHAR(11),
 employedFrom DATE NOT NULL,
 employedTo   DATE
) INHERITS (person);
```

```
CREATE TABLE student (
 JMBAG      CHAR(10),
 enrolDate DATE
) INHERITS (person);
```

```
INSERT INTO person    VALUES (1, 'Ana'   , 'Ban');
INSERT INTO student   VALUES (2, 'Mia'   , 'Nel', '0036000001', '01.07.2013');
INSERT INTO student   VALUES (3, 'Ivo'   , 'Puž', '0036000002', '12.07.2014');
INSERT INTO lecturer  VALUES (4, 'Petar', 'Par', '23456187902', '01.05.2001', NULL);
```

```
SELECT * FROM person;
```

personId	FName	LName
1	Ana	Ban
2	Mia	Nel
3	Ivo	Puž
4	Petar	Par

```
SELECT * FROM ONLY person;
```

personId	FName	LName
1	Ana	Ban

```
SELECT * FROM lecturer;
```

personId	FName	LName	accountNo	employed From	employed To
1	Ana	Ban	23456187902	01.05.2001	

PostgreSQL: Inheritance of the methods

```
CREATE OR REPLACE  FUNCTION description (person)
RETURNS VARCHAR(250) AS $$
    SELECT FName|| ' ' || LName FROM person
      WHERE personId = $1.personId;
$$ LANGUAGE SQL;
```

student and *teacher* inherits method *description* from *person*

```
SELECT lecturer, lecturer.description
  FROM lecturer;

SELECT student, student.description
  FROM student
```

lecturer	description
(4,Petar,Par, 23456187902 ,01.05.2001, null)	Petar Par

student	description
(2, Mia, Nel, 0036000001, 01.07.2013)	Mia Nel
(3, Ivo, Puž, 003600002, 12.07.2014)	Ivo Puž

Polymorphism - redefining methods (functions) at the table level:

```
CREATE OR REPLACE FUNCTION description (student)
RETURNS VARCHAR(250) AS $$
    SELECT JMBAG|| ', ' || enrolDate FROM student
      WHERE personId = $1.personId;
$$ LANGUAGE SQL;
```

```
SELECT student, student.description
  FROM student
```

student	description
(2, Mia, Nel, diplomski, Psihologija)	0036000001, 01.07.2013
(3, Ivo, Puž, diplomski, Medicina)	0036000002, 12.07.2014

PostgreSQL: Inheritance

Problem with primary key (UNIQUE constraint)

- subtable does not inherit primary key
- subtable can contain tuples with same primary key as tuple from supertable

```
CREATE TABLE person(
   personId      INTEGER PRIMARY KEY,
   FName         VARCHAR(25),
   LName         VARCHAR(25)) ;
```

```
CREATE TABLE student (
   JMBAG      CHAR(10),
   enrolDate DATE
) INHERITS (person);
```

```
INSERT INTO person     VALUES (1, 'Ana'   , 'Ban');
INSERT INTO person     VALUES (1, 'Tena'  , 'Pale');                    →
ERROR

INSERT INTO student   VALUES (1, 'Mia'   , 'Nel', '0036000001', '01.07.2013'); → OK
INSERT INTO student   VALUES (1, 'Ivo'   , 'Puž', '0036000002', '12.07.2014'); → OK
```

```
ALTER TABLE student ADD CONSTRAINT studentPk PRIMARY KEY (personId);
```

Is the problem solved?

See in exercises how this problem can be solved .

PostgreSQL: Inheritance

Problem with foreign key

1. Foreign keys are not inherited

```
CREATE TABLE school(
   schoolId   INTEGER PRIMARY KEY,
   schoolLName VARCHAR(250));
```

```
CREATE TABLE person(
   personId  INTEGER PRIMARY KEY,
   FName     VARCHAR(25),
   LName     VARCHAR(25),
   schoolId  INTEGER REFERENCES
                 school(schoolId));

CREATE TABLE student (
   JMBAG     CHAR(10),
   enrolDate DATE
) INHERITS (person);
```

```
INSERT INTO school VALUES (1, 'Gimnazija');

INSERT INTO person  VALUES (1, 'Ana'   , 'Ban', 2);                          → ERROR
INSERT INTO student VALUES (2, 'Mia'   , 'Nel', 2, '0036000001', '01.07.2013');→ OK
```

Solution:

```
ALTER TABLE student ADD CONSTRAINT studentSchoolFk
                    FOREIGN KEY (schoolId) REFERENCES school(schoolId);
```

Prior to creating foreign key the tuple which does not satisfy the reference integrity should be deleted.

PostgreSQL: Inheritance

Problem with foreign key

2. inability to create foreign key referencing all the tuples in the hierarchy (from supertable and all its subtables)

   only the tuples from directly referenced table are taken into account (*most-specific table*)

```
CREATE TABLE person(
   personId     INTEGER PRIMARY KEY,
   FName        VARCHAR(25),
   LName        VARCHAR(25));
```

```
CREATE TABLE student (
   JMBAG      CHAR(10),
   enrolDate DATE
) INHERITS (person);
```

```
CREATE TABLE roomAttendance(
   personId  INTEGER REFERENCES person(personId),
   room      VARCHAR(10),
   time      TIMESTAMP);
```

```
INSERT INTO person   VALUES (1, 'Ana'   , 'Ban');
INSERT INTO student  VALUES (2, 'Mia'   , 'Nel', '0036000001', '01.07.2013');

INSERT INTO roomAttendance VALUES (1, 'B5', '2016-10-22 10:50');   → OK
INSERT INTO roomAttendance VALUES (2, 'B5', '2016-10-22 10:50');   → ERROR
```

ERROR: insert or update on table "roomattendance" violates foreign key constraint "roomattendance_personid_fkey"
DETAIL: Key (personid)=(2) is not present in table "person".

PostgreSQL: OID (*Object identifier*)

- Object unique identifier; hidden attribute
- Can not be updated (eg. With UPDATE statement)
- PostgreSQL uses OID as primary keys in system tables
- OID exists in user tables:
  - when the table is created with parameter: `WITH (OIDS=TRUE)`, or
  - when the configuration variable is set (*default_with_oids*)

```
CREATE TABLE course (courseId    INTEGER PRIMARY KEY,
                     courseName  VARCHAR(250))
                      WITH (OIDS=TRUE);
INSERT INTO course VALUES (1, 'Databases');
```

```
SELECT * FROM course;
```

courseId integer	courseName varchar(250)
1	Databases

oid is hidden attribute

```
SELECT oid, * FROM course;
```

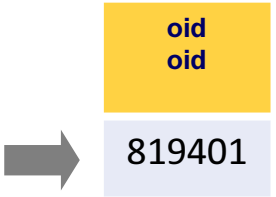
oid oid	courseId integer	courseName varchar(250)
819395	1	Databases

PostgreSQL: OID (*Object identifier*)

- implemented as unsigned integer (4 byte)
  - not enough to provide uniqueness on the database level of
  - when it reaches the maximum value, starts from 1 - possible duplicates

OID as a primary and/or foreign key:

```
CREATE TABLE course (courseName VARCHAR(250))
                        WITH (OIDS=TRUE);
ALTER TABLE course ADD PRIMARY KEY (oid);
INSERT INTO course VALUES ('Databases') RETURNING oid;
```




```
CREATE TABLE courseEnroll(courseOID INT REFERENCES course(oid),
                          studentId INT REFERENCES student(personId),
                         PRIMARY KEY (courseOID, studentId));
INSERT INTO courseEnroll VALUES (819401, 100);
-- OR
INSERT INTO courseEnroll VALUES ((SELECT oid FROM course
                                    WHERE courseName = 'Databases')
                                  , 100);
```

ORDBMS advantages and disadvantages

- advantages
  - main advantages come from *reuse* and *sharing*
    - functionality embedded in the server can be shared by all applications
  - all possibilities of relational databases preserved
    - the extended relational approach preserves the significant body of knowledge and experience that has gone into developing relational applications
- disadvantages
  - complexity
  - dissatisfaction of relational model supporters
    - basic simplicity and purity of relational model is lost
    - lower performance than the current relational technology
  - dissatisfaction of object-oriented model supporters
    - dissatisfaction with used terminology and object concepts

Literature

- J. A. Hoffer, R. Venkataraman, H.Topi: **Modern Database Management (11th Edition)**, Prentice Hall, 2013
  - Chapter 13: Overview: Object-Oriented Data Modeling
    http://wps.prenhall.com/wps/media/objects/14735/15089538/M13_HOFF2253_11_SE_C13WEB.pdf
- Pearson Education: **Advanced Database Topics**: **Object-Relational Databases**
  - http://wps.pearsoned.co.uk/wps/media/objects/10977/11240737/Web%20chapters/Chapter%2014_WEB.pdf
- PostgreSQL 9.4 Documentation - http://www.postgresql.org/docs/9.4/static/index.html
- S.W. Dietrich, S.D. Urban: **An Advanced Course in Database Systems : Beyond Relational Databases**, Prentice Hall, 2005