Object-Relational Databases, or ORDBMSs
Object-Relational Data Example

California Department of Water Resources

• 500,000 photographs
• Captions: “picture of Auburn Dam taken during scaffold construction”
### Natural Representation: Schema with Complex Data

```sql
CREATE TABLE slides (  
id INT,  
date_taken DATE,  
caption DOCUMENT,  
picture JPEG_IMAGE,  
PRIMARY KEY (id));
```

```sql
CREATE TABLE landmarks (  
name VARCHAR(30),  
location POINT,  
PRIMARY KEY (name));
```
Example OR-Style Query over Schema

“Find sunset pictures of landmarks within 20 miles of Sacramento”

Can use:

- Predicate function `sunset(JPEG_IMAGE)`, which analyzes input image, and returns “true” if it is likely to correspond to a sunset and “false” otherwise
- Predicate function `contains(DOCUMENT, string)`, which returns “true” if the input string occurs in the input document and “false” otherwise
- Function `distance(POINT, POINT)`, which returns the distance, in miles, between the two input points
Example OR-Style Query over Schema

“Find sunset pictures of landmarks within 20 miles of Sacramento”

- Predicate function `sunset(JPEG_IMAGE)`, which analyzes input image, and returns “true” if it is likely to correspond to a sunset and “false” otherwise
- Predicate function `contains(DOCUMENT, string)`, which returns “true” if the input string occurs in the input document and “false” otherwise
- Function `distance(POINT, POINT)`, which returns the distance, in miles, between the two input points

```sql
CREATE TABLE slides (  id INT,  date_taken DATE,  caption DOCUMENT,  picture JPEG_IMAGE,  PRIMARY KEY (id));

CREATE TABLE landmarks (  name VARCHAR(30),  location POINT,  PRIMARY KEY (name));
```
Example OR-Style Query over Schema

“Find sunset pictures of landmarks within 20 miles of Sacramento”

SELECT P.id
FROM slides P, landmarks L1, landmarks L2
WHERE sunset (P.picture) AND contains (P.caption, L1.name) AND distance (L1.location, L2.location) < 20 AND L2.name = ‘Sacramento’;
What’s Special about the Example?

- **Query language**: user-defined functions (e.g., sunset) and operators (e.g., distance)
- **Data types**: relatively “complex” types (e.g., JPEG_IMAGE, DOCUMENT, POINT)
- **Performance**: a query optimizer aware of “expensive functions” like sunset
Another Motivating Example: The Dinky Entertainment Company

- Location: Hollywood, CA
- Main assets: cartoon characters (e.g., Herbert the Worm)
- Products: film shows; image, voice, and video-footage licenses (e.g., for action figures, video games)

DBMS manages everything!
An “Old-Fashioned” RDBMS is not Adequate for this Application

CREATE TABLE Frames (  
  frameno INTEGER,  
  image BLOB,  
  category INTEGER,  
  PRIMARY KEY (frameno));

• Application code has to handle BLOBs
• BLOBs transmitted to client unprocessed (efficiency!)
“Old-Fashioned” Relational DBMSs

• Small, fixed set of data types (e.g., integers, dates, strings)
• Fixed set of operators and functions

Good for administrative data processing
Bad for more complex or diverse kinds of data
New Data Types Required

• **User-defined abstract data types (ADTs):** image, sound, video, with functions and operators

• **Type constructors:** multisets, arrays, …

• **Inheritance:** low- and high-resolution images are images
ORDBMS DDL Statements

CREATE TABLE Frames (frameno integer, image jpeg_image, category integer, PRIMARY KEY(frameno))

CREATE TABLE Categories (cid integer, name text, lease_price float, comments text, PRIMARY KEY(cid))

CREATE TYPE theater_t AS ROW(tno integer, name text, address text, phone text)
INSTANTIABLE

CREATE TABLE Theaters OF theater_t (REF is tid SYSTEM GENERATED, PRIMARY KEY (tid))

CREATE TABLE Nowshowing (film integer, theater ref(theater_t) SCOPE Theaters, start date, end date, PRIMARY KEY (film, theater, start))

CREATE TABLE Films (filmno integer, title text, stars VARCHAR(25) ARRAY [10], director text, budget float, PRIMARY KEY (filmno))

CREATE TABLE Countries (name text, boundary polygon, population integer, language text, PRIMARY KEY (name))
ORDBMSs: “Extended” SQL

• User-defined data types possible for attributes

• Complex attributes possible (not 1NF)
  First Normal Form: A relation is in 1NF if every attribute contains only “atomic” values (i.e., no lists, sets, etc.)

• Reference types

• Inheritance
Defining New ADTs

- Define how to **read** in and how to **output** objects of the new type
- Define the **size** of the objects of the new type
- Define new **methods** for the new type (e.g., compress, rotate, shrink, crop, etc. for an image type)
“Packaged” ADTs Common and Useful

• DataBlades (IBM’s Informix)
  • time series, spatial extensions, …

• Data Cartridges (Oracle)
  • Oracle Text, …

• DB2 Extenders (IBM)
  • …
Object Relational Features of SQL

- **Type constructors** to specify complex objects: ROW, ARRAY, MULTISSET (plus LIST and SET)
- **Encapsulation of operations** through user-defined types that might include operations as part of their definition
- **Object identity** via REFERENCE type
- Inheritance
ROW Type Examples

CREATE TYPE Addr_Type AS ROW (street VARCHAR(45),
city VARCHAR(25),
zip CHAR(5));

CREATE TYPE Emp_Type AS ROW (name VARCHAR(35),
addr Addr_Type,
age INTEGER)
INSTANTIABLE;
ARRAY Type Example

CREATE TYPE Comp_Type AS ROW (compname VARCHAR(20), location VARCHAR(20) ARRAY[10]) INSTANTIABLE;

- “location” can hold a maximum of 10 elements (number can also be left unspecified)
- location[1]: first element in array
- location[CARDINALITY(location)]: last element in array
Defining Typed Tables: Examples
Defining Typed Tables: Examples

CREATE TABLE Employee OF Emp_Type (REF IS emp_id SYSTEM GENERATED, PRIMARY KEY (emp_id));
CREATE TABLE Company OF Comp_Type (REF IS comp_id SYSTEM GENERATED, PRIMARY KEY (comp_id));
Example Query: “List the name of each employee who works at company ‘ABCD’”

First, “non-OR” design:

CREATE TABLE Employee OF Emp_Type (REF IS emp_id SYSTEM GENERATED, PRIMARY KEY (emp_id));

CREATE TABLE Company OF Comp_Type (REF IS comp_id SYSTEM GENERATED, PRIMARY KEY (comp_id));
Example Query: “List the name of each employee who works at company ‘ABCD’”

First, “non-OR” design:

CREATE TABLE Employment (  
eid INTEGER,  
cid INTEGER,  
FOREIGN KEY (eid) REFERENCES Employee(emp_id),  
FOREIGN KEY (cid) REFERENCES Company(comp_id),  
PRIMARY KEY (eid, cid));

SELECT M.name  
FROM Employment E, Employee M, Company C  
WHERE E.eid=M.emp_id AND E.cid=C.comp_id AND C.compname='ABCD'
Now, OR-Style “Employment” Table

CREATE TABLE Employee OF Emp_Type (  
    REF IS emp_id SYSTEM GENERATED,  
    PRIMARY KEY (emp_id));
CREATE TABLE Company OF Comp_Type (  
    REF IS comp_id SYSTEM GENERATED,  
    PRIMARY KEY (comp_id));
CREATE TABLE Employee OF Emp_Type (  
    REF IS emp_id SYSTEM GENERATED,  
    PRIMARY KEY (emp_id));
CREATE TABLE Company OF Comp_Type (  
    REF IS comp_id SYSTEM GENERATED,  
    PRIMARY KEY (comp_id));
CREATE TYPE Employment_Type AS ROW (  
    employee REF(Emp_Type) SCOPE (Employee),  
    company REF(Comp_Type) SCOPE (Company))  
INSTANTIABLE;
CREATE TABLE Employment OF Employment_Type (  
    PRIMARY KEY (employee, company));
Example Query: “List the name of each employee who works at company ‘ABCD’”

Now, “OR” design:

```
CREATE TABLE Employee OF Emp_Type (  
    REF IS emp_id SYSTEM GENERATED,  
    PRIMARY KEY (emp_id));
CREATE TABLE Company OF Comp_Type (  
    REF IS comp_id SYSTEM GENERATED,  
    PRIMARY KEY (comp_id));
CREATE TYPE Employment_Type AS ROW (  
    employee REF(Emp_Type) SCOPE (Employee),  
    company REF(Comp_Type) SCOPE (Company))  
    INSTANTIABLE;
CREATE TABLE Employment OF Employment_Type (  
    PRIMARY KEY (employee, company));
```
Example Query: “List the name of each employee who works at company ‘ABCD’”

Now, “OR” design:

SELECT E.employee → name
FROM Employment E
WHERE E.company → compname = ‘ABCD’

→ is used for “dereferencing”
MULTISETs

• INTEGER MULTISET can have values 5, 30, 45, -8, 9, 5, for example
• Nesting is allowed, so an element type can itself be a collection type
• Creating MULTISET values:
  • By enumeration: MULTISET [1, 2, 3, 4, 2, 5]
  • By a query expression: MULTISET (SELECT C.grades FROM Courses C)
MULTISET Can Be Used As a Table, Using UNNEST

```sql
SELECT T.A, T.A*2 AS TIMES_TWO
FROM UNNEST (MULTISET [4, 3, 2, 1]) AS T(A)
```

<table>
<thead>
<tr>
<th>A</th>
<th>TIMES_TWO</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
MULTISET Operations

• **UNION, EXCEPT, INTERSECT**
  Options: ALL (default: keeps multiset semantics with duplicates), DISTINCT

• New aggregate functions:
  • **COLLECT**: creates a multiset from attribute values
  • **FUSION**: creates a multiset by combining multiset attribute values
  • **INTERSECTION**: creates a multiset as intersection of multiset attribute values
## MULTISET Example

### Friends

<table>
<thead>
<tr>
<th>Friend</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>MULTISET ['reading', 'pop music', 'running']</td>
</tr>
<tr>
<td>Susan</td>
<td>MULTISET ['movies', 'opera', 'reading']</td>
</tr>
<tr>
<td>James</td>
<td>MULTISET ['movies', 'reading']</td>
</tr>
</tbody>
</table>

### All_Friends

<table>
<thead>
<tr>
<th>All_Friends</th>
<th>All_Hobbies</th>
<th>Common_Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTISET ['John', 'Susan', 'James']</td>
<td>MULTISET ['reading', 'pop music', 'running', 'movies', 'opera', 'reading', 'movies', 'reading']</td>
<td>MULTISET ['reading']</td>
</tr>
</tbody>
</table>
## MULTISET Example

### Friends

<table>
<thead>
<tr>
<th>Friend</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>MULTISET ['reading', 'pop music', 'running']</td>
</tr>
<tr>
<td>Susan</td>
<td>MULTISET ['movies', 'opera', 'reading']</td>
</tr>
<tr>
<td>James</td>
<td>MULTISET ['movies', 'reading']</td>
</tr>
</tbody>
</table>

### Select Query

```sql
SELECT COLLECT (F.Friend) AS All_Friends,
       FUSION (F.Hobbies) AS All_Hobbies,
       INTERSECTION (F.Hobbies) AS Common_Hobbies
FROM Friends F
```

<table>
<thead>
<tr>
<th>All_Friends</th>
<th>All_Hobbies</th>
<th>Common_Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTISET ['John', 'Susan', 'James']</td>
<td>MULTISET ['reading', 'pop music', 'running', 'movies', 'opera', 'reading', 'movies', 'reading']</td>
<td>MULTISET ['reading']</td>
</tr>
</tbody>
</table>
MULTISETs: Additional Operations

- Removing duplicates (SET (<multiset>))
- Returning number of elements, counting repetitions (CARDINALITY (<multiset>))
- Checking membership (<value> [NOT] MEMBER OF <multiset>)
- Checking inclusion (<multiset> SUBMULTISET OF <multiset>)
- Checking if multiset has no repeated elements (<multiset> IS SET)
- …
Inheritance

• To reuse and refine type definitions
• To create hierarchies of collections with similar but not identical objects

SQL standard doesn’t support multiple inheritance, so each subtype has only one immediate supertype
Type Inheritance Example
CREATE TYPE Person_Type AS (  
    name VARCHAR(20),
    address VARCHAR(20))

INSTANTIABLE NOT FINAL;
// INSTANTIABLE: can have tables of this schema; NOT FINAL: subtypes possible

We can use inheritance to define two subtypes:
CREATE TYPE Student_Type UNDER Person_Type AS (  
    degree VARCHAR(20),
    department VARCHAR(20))

INSTANTIABLE; // all attributes of Person_Type plus 2 new ones

CREATE TYPE Teacher_Type UNDER Person_Type AS (  
    salary INTEGER,
    department VARCHAR(20))

INSTANTIABLE; // all attributes of Person_Type plus 2 new ones

Subtypes can add new attributes, add new methods, change implementation of inherited methods (but signature of methods cannot change); subtypes cannot change inherited attributes, though
Table Inheritance Example

“Supertable”/“subtable” relationships specified also with UNDER:
Table Inheritance Example

“Supertable”/“subtable” relationships specified also with UNDER:

CREATE TABLE Person OF Person_Type
    REF IS person_id SYSTEM GENERATED;
CREATE TABLE Student OF Student_Type UNDER Person;
CREATE TABLE Teacher OF Teacher_Type UNDER Person;

• When a tuple is inserted into a subtable (e.g., Student), it is also automatically inserted into its supertable(s) (e.g., Person)
• INSERT, DELETE, and UPDATE are appropriately propagated
Inheritance Requirements

All tuples “corresponding to each other” (i.e., tuples with the same values for inherited attributes) must be derived from the insertion of just one tuple in one table

- In our example, we cannot then have a Person tuple corresponding to both a Student tuple and a Teacher tuple
- Then, Student and Teacher cannot overlap (unless Person is not INSTANTIABLE)
- Do Student and Teacher cover Person (i.e., does every Person tuple have to be either a Student or a Teacher)? Should be specified: Yes if Person_Type is declared as NOT INSTANTIABLE
Substitution Principle

• Given a supertype A and a subtype B, it is always possible to substitute an object of type B into a legal expression written for objects of type A.

• Then dynamic binding of methods is needed at query time (recall that subtype definitions can change implementation of inherited methods).
PostgreSQL 14 Support of OR Features

• **Type constructors:**
  - **ROW:** not really supported, but can use CREATE TYPE mechanism
  - **ARRAY:** somewhat supported, with array size limits and number of dimensions ignored (allowed for documentation only)
    
    ```
    CREATE TABLE sal_emp (  
        name TEXT,  
        pay_by_quarter INTEGER [],  
        schedule TEXT [] []);  
    ```
  - **MULTISET:** not supported

• **User-defined types:** supported (composite, enumerated, range, “base” types—similar to ADTs but only for superusers)
  
  ```
  CREATE TYPE ... AS ...  
  ```

• **Object references:** not really supported (primitive object ids present, but not in full OR sense)
Inheritance: limited support, just to specify a list of tables from which a new table automatically inherits all columns

CREATE TABLE Cities (
    name TEXT,
    population FLOAT);

CREATE TABLE Capitals (
    state CHAR(2))

INHERITS (Cities); // if more than 1 table listed, attributes are unioned/merged, and error is returned if problem with incompatible types of attributes named the same

SELECT C.name, C.population FROM Cities C WHERE C.population>1000000
// executed over both Cities and Capitals (i.e., over Cities and all its descendants)

… FROM ONLY Cities …
// executed over Cities that are not Capitals

• **Problem:** PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints not inherited

• **Problem:** tuples are inserted exactly in table mentioned, not “propagated” to supertables
Serious Limitations of PostgreSQL Inheritance

- PRIMARY KEY and UNIQUE constraints apply to only one table, not to their inherited children.
- So if PRIMARY KEY(name) is specified in Cities table, nothing prevents a “duplicate” city —with conflicting values for other inherited attributes— from appearing in subtable Capitals. Different from behavior in SQL standard, where tuples inserted in Capitals get automatically propagated to Cities. And duplicate tuples would appear by default on queries over Cities that do not say FROM ONLY Cities!
- FOREIGN KEY constraints also apply to only one table, not to their inherited children.
  Foreign keys from other tables onto Cities (and Capitals) are then a problem: FOREIGN KEY ... REFERENCES Cities(name) would not capture or include Capitals.

https://www.postgresql.org/docs/14/ddl-inherit.html
Project 2

You will expand your project by adding three out of:

- TEXT
- ARRAY
- Composite TYPE
- TRIGGER