# OBJECTS AND DATABASES

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 21

# Relational Model and 1NF



- Relational model specifies that all attribute domains must be atomic
  - A database schema is in 1NF if all attribute domains are atomic
- Not always preferred approach in real world use
- In relational model:

  *employee(emp_id, emp_name)*

  *emp_phone(emp_id, phone_num)*

  *emp_deps(emp_id, dependent)*

  - Need two joins just to get all data for an employee!

| *employee* |
|---|
| *emp_id*<br>*emp_name*<br>{ *phone_num* }<br>{ *dependent* } |

# Composite Types



- Also, frequently have composite types that are reused
- Example:
  - Add home/work addresses to design
- In relational model, composite types are decomposed

  *employee(emp_id, emp_name,*
  *work_street, work_city, … )*

- …but programming languages typically provide structures or classes for composite types!

| *employee* |
|---|
| *emp_id*<br>*emp_name*<br>*work_address*<br>    *street*<br>    *city*<br>    *state*<br>    *zipcode*<br>*home_address*<br>    *street*<br>    *city*<br>    *state*<br>    *zipcode*<br>{ *phone_num* }<br>{ *dependent* } |

# Database Applications



- Programming languages have support for non-atomic types
  - Address class or structure:
    - street, city, state, zipcode
  - Arrays of phone numbers and dependents
- Application has to translate between relational model version and programming language representation
  - Annoying to deal with, at the least…
  - At worst, can have substantial application quality and performance impacts!

# SQL User-Defined Types



- SQL:1999 includes User-Defined Types
  - Allows users to define non-atomic types where appropriate
  - (Make sure it's actually appropriate!)
  - Frequently abbreviated as UDT
- Multivalued types – arrays, sets, lists, etc.
  - Elements are all the same type
- Structured types – composite attributes
  - Elements may be different types

# Non-Atomic Types for Employees



- Declare new UDT for addresses:

```
CREATE TYPE Address AS (
  street  VARCHAR(60),
  city    VARCHAR(40),
  state   CHAR(2),
  zipcode CHAR(9)
);
```

  - Only specify types, not constraints!
  - Defines a new structured type within the database schema
- For arrays, just add **`ARRAY[`*`n`*`]`** to column type
  - ***n*** is optional
  - Array elements have indexes 1 to ***n***

# Using Non-Atomic Types



- Employee table:

```
CREATE TABLE employee (
  emp_id       INTEGER      PRIMARY KEY,
  emp_name     VARCHAR(100) NOT NULL,
  work_address Address      NOT NULL,
  home_address Address      NOT NULL,
  phone_nums   CHAR(12)     ARRAY[],
  dependents   VARCHAR(100) ARRAY[]
);
```

- Now all details of an employee are contained within a single table
  - E-R model maps directly into this design
- Retrieving all details of an employee will be fast

# Structured Types in DML



- Accessing elements of a structured type:

```
SELECT emp_id, emp_name FROM employee
WHERE work_address.city = home_address.city;
```

- Specifying all values of a structured type:

```
UPDATE employee SET work_address =
  ('123 Main St.', 'Springfield', 'OH', '45505')
  WHERE emp_id = 5352;
```

- Specifying individual values of a structured type:

```
UPDATE employee SET work_address.city = 'Akron',
  work_address.zipcode = '44310'
  WHERE emp_id = 5352;
```

# Array Types in DML



- Specifying all values of an array type:

```
UPDATE employee SET phone_nums =
  ARRAY['800-555-1234', '800-555-5678']
  WHERE emp_id = 5352;
```

- Specifying individual values of an array type:

```
  UPDATE employee
    SET phone_nums[1] = '800-555-2345'
    WHERE emp_id = 5352;
```

- Order of elements in array is preserved!
  - Useful when order of values is meaningful
  - e.g. author-list in a database of research papers

# Array Types in DML (2)



- Array columns are like nested relations
  - A nested relation is stored within a single column
- SQL:1999 provides nesting and unnesting operations for arrays
- To unnest an array value:

```
SELECT emp_id, emp_name, p.phone
  FROM employee AS e,
       UNNEST(e.phone_nums) AS p(phone)
  WHERE emp_id = 5352;
```

| *emp_id* | *emp_name* | *phone* |
|---|---|---|
| 5352 | Bob Smith | 800-555-2345 |
| 5352 | Bob Smith | 800-555-5678 |

# Array Types in DML (3)



- Can also retrieve element ordering details

```
SELECT emp_id, emp_name, p.phone, p.p_index
  FROM employee AS e,
       UNNEST(e.phone_nums) WITH ORDINALITY
         AS p(phone, p_index);
```

| *emp_id* | *emp_name* | *phone* | *p_index* |
|---|---|---|---|
| 5352 | Bob Smith | 800-555-2345 | 1 |
| 5352 | Bob Smith | 800-555-5678 | 2 |

- Can use **COLLECT** to combine values into an array

```
SELECT emp_id, COLLECT(phone_num) AS phone_nums
FROM raw_employee_data GROUP BY emp_id;
```

  - Very similar to grouping and aggregation operation!
- Can also pass subquery to **ARRAY()** fn. to populate an array

# SQL:1999 User Defined Types



- SQL:1999 user-defined types help with composite and multivalued attributes…
  - Can create schemas that don't incur join overheads for multivalued attributes
  - Can represent composite attributes more naturally within the SQL schema
- Still not quite the same as what programming languages can provide

# Objects and Relations



- Many programming languages are object-oriented
  - Objects:  encapsulation of state and behavior
  - Classes:  specifications of objects' state and behavior
  - Also other features, such as class inheritance
    - A class can derive from a parent class
    - Child class has specialized capabilities and additional state
  - C++, Java, C#, Python, PHP, etc.  All <u>widely</u> used.
- Typical approach for storing objects in a relational database:
  - Classes usually map to tables
  - Objects map to individual rows in a table

# Objects and Relations (2)



- Relational databases aren't designed to store objects!
- "Object-relational impedance mismatch"
  - A number of serious issues arise when storing objects into a relational database
- Relational databases cannot enforce the same constraints that OOP languages can enforce!
  - Objects encapsulate and manage state very carefully
  - In a relational database, <u>all</u> values can be manipulated very easily
  - Storing object-data in a relational database increases potential for data corruption

# Objects and Relations (3)



- Objects can reference each other
  - More akin to the network data model, which preceded the relational model
    - Objects are accessed by following specific object-references
    - Tuples are retrieved en masse via a query language
- Representing object identities and object references in a relational database can be tricky
  - In OOP languages, object-references are usually <u>opaque</u> to the user, and not manipulated directly
  - In relational model, identifying values are intentionally very visible and meaningful
    - Part of Codd's original intent with relational model

# Objects and Relations (4)



- OOP languages also provide features not present in relational model
  - Ability to specify methods to be called on objects
  - Class inheritance and class hierarchies
  - Require careful modeling in a relational database, if it can be implemented at all!
    - Frequently have multiple choices for modeling, with different performance and space implications
- A number of other issues as well…
  - Some are more esoteric than others
- Can definitely live with most of these issues, but be aware of the mismatch in capabilities!

# Addressing The Mismatch



- Two main approaches to object-relational mismatch
- Object-Oriented Databases (ODBMS/OODBMS)
  - Further extend SQL's type system to support basic object-oriented constructs
  - Database supports object-oriented abstractions directly and internally
- Persistent Programming Languages
  - Hide (R)DBMS storage operations from programmers
  - Automate the translation between programming-language objects and database storage

# Object-Oriented Database Systems



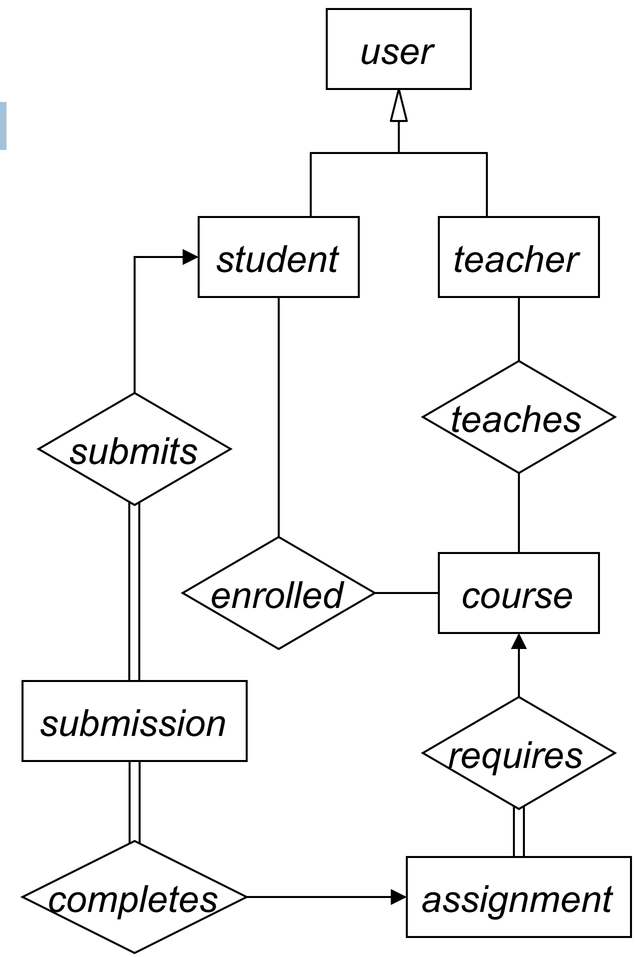
- ODBMSes provide direct support for classes and objects
  - Define constructors and methods for classes
  - Define type hierarchies for classes
  - Provide object-reference support
- Inclusion of objects requires significant changes to the query language
  - Objects can refer to collections of objects
  - Must support path-based queries

# ODBMS Example



- Course management system database schema
- Entities are objects in the ODBMS
  - Relationships specify object-references
- Retrieve names of students enrolled in CS121:

```
SELECT s.name FROM student s
WHERE s.enrolled.course.name = 'cs121';
```



# ODBMS Operations



- ODBMSes must provide capabilities for:
  - Object-definition, similar to SQL DDL
  - Queries on objects, similar to SQL DML
- Object Data Management Group
  - Consortium founded in 1991, to create ODBMS specifications
- Standardization effort has had limited success
  - Several DB vendors offer ODBMS capabilities, but syntax and feature-sets vary pretty widely.
  - (not unlike SQL standard…)

# ODBMS Operations (2)



- Object Description Language (ODL)
  - For specifying object-database schemas
  - Specify classes, class-members, and class inheritance hierarchies
- Object Query Language (OQL)
  - A SQL-like query language for querying object-databases
  - Most significant change is ability to specify "path expressions" that follow relationships between objects

# Object-References



- All objects in an ODBMS have a unique identifier of some kind
  - Object Identifier (OID)
- Objects reference other objects using their OIDs
  - Akin to a pointer or reference to the object
- ODBMSes generally load referenced objects from disk, <u>as needed</u>.
  - Lazy loading, not eager loading
  - Objects tend to reference many other objects…
  - Object-loading must be done carefully, to avoid unnecessary resource usage!

# ODBMS Summary



- Hasn't been widespread adoption of ODBMSes
  - Cost of switching is very high
    - A company's data is extremely valuable
    - Relational model is satisfactory for most needs, so why change?
- Many commercial databases provide a hybrid data model now
  - Object-Relational Database Management System (ORDBMS)
    - Blends object capabilities and relational database capabilities
  - Typically provide capabilities for simple type hierarchies, and simple class-method declarations

# Persistent Programming Languages



- Most popular approach to object-relational impedance mismatch:
  - Create or enhance OOP languages to provide persistent objects directly in the language itself
- Normally, when a program terminates, all objects it created go away
  - These objects are <u>transient</u>
- <u>Persistent</u> objects are stored before termination
  - (in a database of some kind…)
  - Next time the program runs, persistent objects can be retrieved and used

# Persistent Programming Languages (2)



- Persistent programming languages usually store objects in relational databases
  - PostgreSQL, MySQL, SQLite, Oracle, etc.
  - Also called "object-relational mapping layers" or simply "object-relational mappers" (abbrev. ORM)
- Type specification is <u>entirely</u> in the OO programming language itself
  - Able to leverage most types and OO capabilities of the programming language itself
  - Usually very few differences in capabilities between transient and persistent objects

# Database Operations



- Database operations are usually entirely obscured from the programmer
  - Persistent object storage and retrieval is handled entirely by the framework itself
- Many ORM layers also provide automatic data-definition capabilities
  - Given a set of persistent objects, ORM layer can generate a SQL schema for those objects
  - Persistent objects are typically annotated to indicate "primary key" values, etc.

# Automatic DDL Generation



- Persistence frameworks are becoming quite sophisticated with auto-DDL generation…
- Two main issues to consider!
- Database schema migration:
  - It's easy to change classes, or add new ones
  - Absolutely essential to have a migration path for existing data!
- Database performance:
  - ORM layers don't usually generate a schema tuned for high-performance and scalability

# ORM Layers and Schema Migration



- Most ORM layers don't yet provide schema/data migration capabilities directly…
- Typically, external libraries/tools are available, to add schema-migration support to ORM layers
- General approach:
  - Take a snapshot of every stable version of data model
  - When the schema changes in *simple* ways, tool can generate the needed SQL DDL to migrate the schema
  - Tools also support manual data-migration steps for more involved changes
- <u>Always</u> back up data before using these tools! ☺

# ORM Layers and Performance



- Most ORM layers also provide ability to run custom SQL on database before/after schema-generation
  - Add stored procedures and user-defined functions
  - Add indexes
  - Populate database with initial data
- To facilitate this, ORM layers frequently document exactly how table/column names are generated
- DDL that isn't specifically managed by the ORM can make data models significantly more fragile!
  - May need to change names/structure in multiple places

# Manual DDL Creation



- Because of these issues, ORM layers also frequently support mapping objects to an existing schema
- You design the schema with specific needs in mind
  - Specify table indexes, partition large tables, etc.
- When a schema needs to change, it's easier to design a migration plan for your data
  - You have the "old" schema definition…
  - You provide the new schema definition yourself…
  - You can design the migration process for upgrading the database and preserving your data.

# Persistence Framework Limitations



- Many persistence frameworks impose limitations on the kinds of schemas they support
  - Make sure to understand these limitations <u>before</u> designing schemas for these frameworks!
- Multiple-column primary keys
  - Supported by more advanced ORM layers…
  - …but they may not support multi-column foreign keys!
- Ternary relationships
  - Many ORM layers only support binary relationships
  - Need to model ternary relationships as a combination of binary relationships

# Persistence Framework Challenges

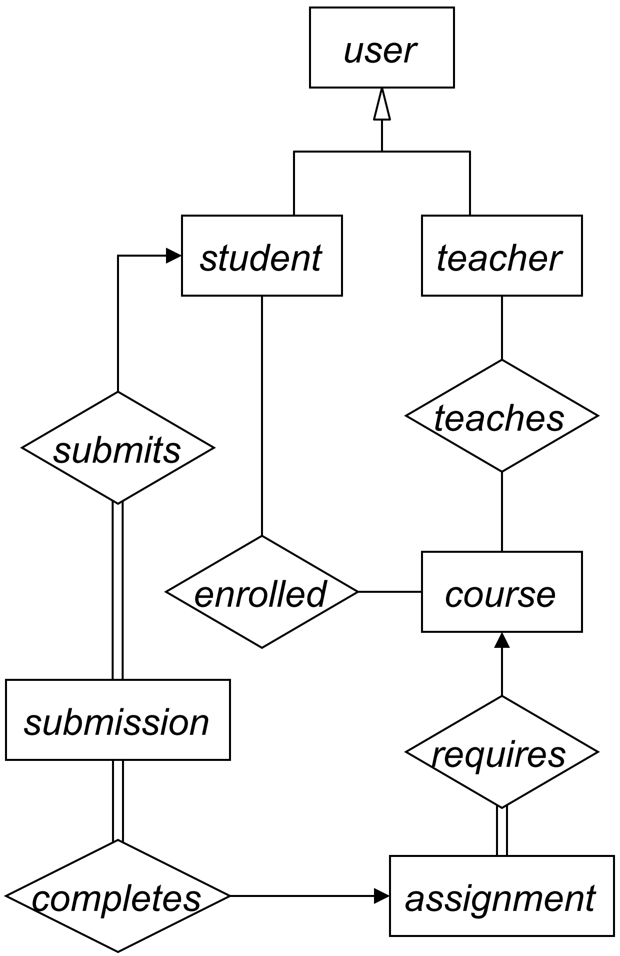


- Many persistence frameworks also have limited support for database constraints
- Virtually all can handle referential integrity constraints
- Not all ORM layers can handle objects with multiple candidate keys
  - …but these days, most of them can.
- Be careful about general `CHECK` constraints!
- ORM layer must identify the cause of database errors generated by violating these constraints
  - Hard to do for a wide range of database vendors

# Loading Persistent Objects



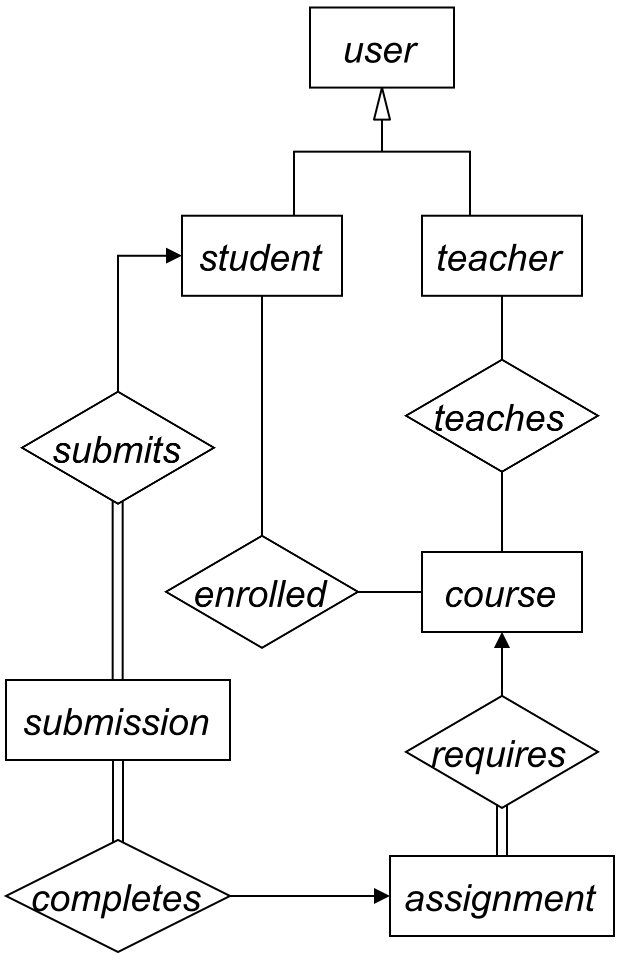
- Like ODBMSes, persistent object frameworks must carefully manage object retrieval
- Example:
  - Retrieve all students in CS121
- Step 1: retrieve the **`Course`** object with name of "cs121"
  - The **`Course`** object will have a set of **`Student`**-references, a set of **`Assignment`**-refs, etc.



# Loading Persistent Objects (2)



- Should the **`Course`** object eagerly or lazily load **`Student`** objects from the database?
  - We said we want <u>all</u> students…
  - Makes most sense to get all of them in one query.
- **`Student`** objects will have a collection of **`Submission`** objects, each of which has an **`Assignment`** object, …
  - ORM layer must get exactly what is needed, and no more!



# Persistent-Object References



- OO programming language already has a way of referencing objects
  - e.g. pointers in C++, or references in Java/Python/…
- ORM layers must map between in-memory reference type and database reference type
  - A persistent object may not be loaded into memory yet, but other in-memory persistent objects refer to it
- Two kinds of persistent-object references:
  - A database-reference for when object isn't loaded yet
  - An in-memory reference for when the object is already in memory

# Persistent-Object References (2)



- When a DB-reference is followed:
  - ORM layer loads object into memory from database
  - Then, ORM layer switches out the DB-reference for an in-memory reference
- In compiled languages, often implemented with <u>pointer-swizzling</u>
  - ORM layer uses special pointer-values for database-references to objects
  - When pointer is accessed, the ORM layer is notified (e.g. via a page-fault signal)
  - ORM layer loads the object, then directly changes the pointer value to point to the loaded object instead

# Persistent-Object References (3)



- In interpreted (or VM-based) languages, often implemented with hollow objects
  - Before a persistent object is loaded, the reference actually points to a proxy
  - When the proxy is accessed, ORM layer retrieves the object from the DB
  - Proxy is replaced with the loaded object
- In-memory objects must also track state changes!
  - Writes to the object must flag the object as "dirty"
  - ORM layer ensures that dirty objects are saved to DB

# Persistent Objects – Summary



- Much more popular solution to object-relational impedance mismatch
  - But, an admittedly incomplete solution to the mismatch
- Obscures much of the pain of moving objects to and from the database
- Also frequently provides ability to manage schema design directly (but watch out for limitations!)
- Some Java persistence frameworks:
  - Java Persistence API, Hibernate
- Some Python persistence frameworks:
  - Django models (+ South for migration), SQLAlchemy