Object Databases

Esteban ZIMÁNYI
Department of Computer & Decision Engineering (CoDE)
Université Libre de Bruxelles
ezimanyi@ulb.ac.be

Info-H-415 Advanced Databases
Academic Year 2012-2013

Object Databases: Topics

- Object-Oriented Databases
  - **ODMG**
  - Linq
- Object-Relational Databases
  - SQL
  - Oracle
- Summary

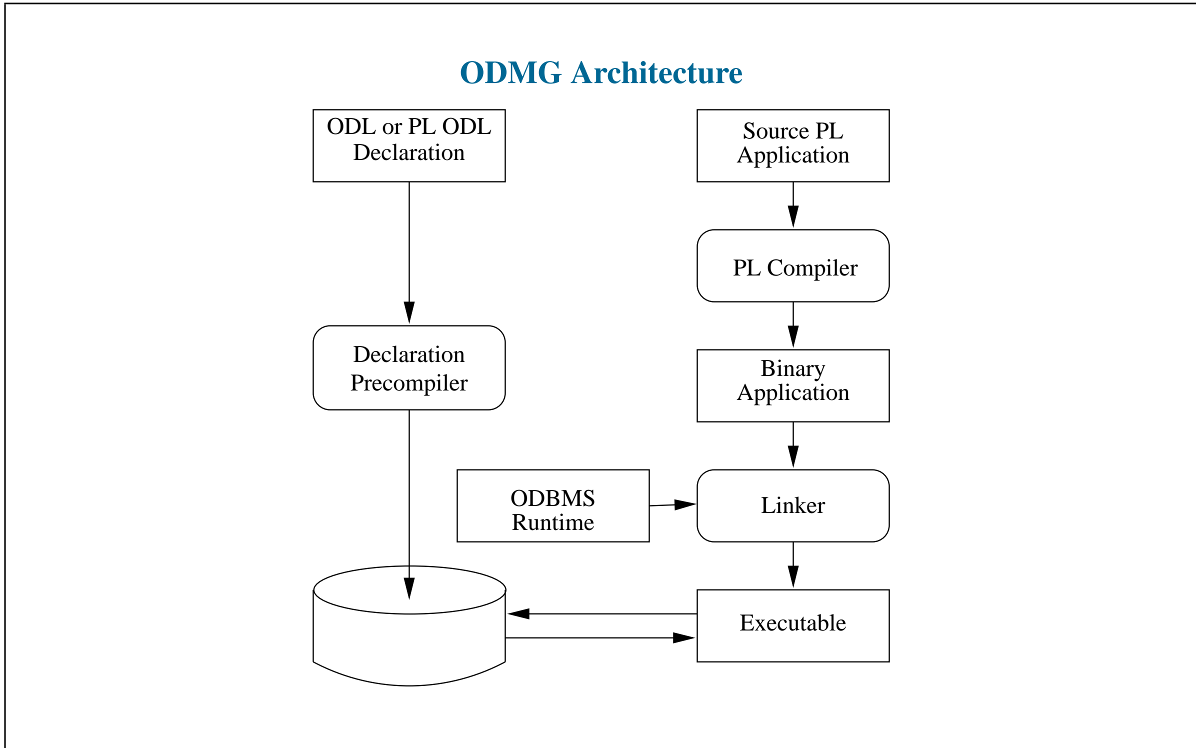
Object Databases

- ODBMS: DBMS that integrates database (DB) and object programming language (PL) capabilities
- Make DB objects appear as PL objects in one or more PLs
- Persistent store for objects in a multiuser environment
- Combines object properties with traditional DBMS properties: locking, protection, transactions, querying, versioning, concurrency, persistence
- Objects referenced through persistent identifiers

Object Data(base) Management Group

- Founded in 1991 by OODB constructors
- Objective: create a standard in OODBs, that applies to
  - ODBMSs: store objects directly
  - Object-to-Database Mappings: convert and store objects in, e.g., RDBs
- Releases: 1.0 in 1993, 1.1 in 1994, 1.2 in 1995, 2.0 in 1997, 3.0 in 2000
- Proposal available as a book from Morgan Kaufmann Publishers
- In February 2006, OMG formed the Object Database Technology Working Group (ODBT WG) for creating new specifications based on the ODMG 3.0 specification
- The work of the ODBT WG was suspended in March 2009

ODMG Architecture



ODMG Standard

- Builds upon the existing OMG, SQL-92, INCITS (formerly ANSI) PL standards, JavaSofts Java specification
- Define a framework for application portability between compliant data storage products
- Functional components
  - Object Model
  - Object Definition Language
  - Object Query Language
  - Language Bindings to Java, C++, and Smalltalk

ODMG: Object Model

- OMG's Common Object Model (COM)
  - common denominator for ORB, OPL, object storage systems, . . .
  - associated with Interface Definition Language (IDL)
- ODMG's OM
  - main component of standard,
  - extends COM with a ODBMS profile to support object storage needs
- Central features
  - Dual model: object and literals
  - Atomic, structured, and collection literals
  - Object with properties (attributes, relationships), operations, exceptions
  - Object naming, lifetime identity
  - Classes with extent and keys, collection classes
  - Multiple inheritance
  - Concurrency control, object locking, database operations

ODMG: Object Definition Language (ODL)

- Strict superset of OMG's IDL for DB schema definition
- Objects defined in terms of type, attributes, relationships, and operations
- Abstraction layer: ODL-generated schemas are PL- and DB-independent
- Allow applications to be
  - moved between compliant DBs
  - different language implementations
  - translated into other Data Definition Languages (DDLs), e.g., SQL3

ODMG: Object Query Language (OQL)

- SQL-like declarative language
- Allows efficient querying of database objects, including high-level primitives for object sets and structures
- Based on query portion of SQL-92: superset of SQL-92 SELECT syntax
- Object extensions for object identity, complex objects, path expressions, operation invocation, inheritance
- OQLs queries can invoke operations in ODMG language bindings, OQL may be embedded in an ODMG language binding
- No associated OML: object integrity maintained using operations of objects

ODMG: Language Bindings

- Define Object Manipulation Languages (OMLs) extending PL to support persistent objects
- Defined for Java, C++, and Smalltalk
- Bindings also include support for OQL, navigation, transactions
- Enable developers to work inside a single language environment without a separate DB language
- Example: Java Binding
  - Adds classes and other constructs to Java environment for supporting ODMG's OM: e.g., collections, transactions, databases
  - Instances of classes can be made persistent without changes to source code
  - Persistence by reachability: at transaction commit, objects reached from root objects in DB automatically made persistent

ODMG Object Model: Summary

- Differentiates objects (with identity) and literals
- Literals: atomic, collection, structured
- Classes with attributes, methods, subtyping, extension (inheritance of state)
- Binary relationships, referential integrity maintained by DBMS
- Generic collection classes, structured object classes
- Names can be given to objects
- Persistent and transient objects
- Exception model
- Metadata stored in a Schema Repository
- Conventional lock-based approach to concurrency control
- Transaction model
- Database operations

ODMG Object Model

- Basic modeling primitives: objets (with unique identifiers) and literals
- Objects and literals categorized by their types
- All elements of a type have common state (properties) and behavior (operations)
- Properties: attributes and relationships
- Operations: may have a list of input and output parameters, each with a type; may also return a typed result
- DB stores objects shared by multiple users and applications
- DB based on a schema defined in ODL, contains instances of types defined by its schema

Types: Specifications and Implementations

- Type: external specification, one or more implementations
- **Specification**: external characteristics of type, visible to its users
  - implementation-independent description of operations, exceptions, properties
- **Implementation**: internal aspects of type
  - implementation of operations, other internal details
- Encapsulation = separating specifications from implementation, helps
  - multilingual access to objects of a single type
  - sharing objects accross heterogeneous computing environments

Type Specification

- Interface: abstract behavior of an object type

```
interface Employee {
  float salary()
  void hire();
  void fire(); }
```

- Class: abstract behavior and state of an object type

```
class Person {
  attribute string name;
  relationship Person spouse inverse Person::spouse;
  void marriage(Person with); }
```

- Literal: abstract state of a literal type

```
struct Address {
  string street;
  string City;
  string Phone; }
```

Type Implementation

- Implementation of a type = a representation and a set of methods
- **Representation**: data structure derived from abstract state by PL binding
  - property in abstract state $\Rightarrow$ instance variable of appropriate type
- **Methods**: procedures derived from abstract behavior by PL binding
  - operation in abstract behavior $\Rightarrow$ method defined
  - read or modify representation of an object state, invoke other operations
  - also, internal methods without associated operation
- A type can have more than one implementation, e.g.,
  - in C++ and in Smalltalk
  - in C++ for different machine architectures

Language Bindings

- Each language binding defines an implementation mapping for literal types
  - C++ has constructs to represent literals directly
  - Smalltalk and Java map them to object classes
- Example: floats
  - represented directly in C++ and Java
  - instances on class Float in Smalltalk
- No way to represent abstract behavior on literal type $\Rightarrow$ language-specific operations to access their values
- OPL have language constructs called classes
  - implementation classes $\neq$ abstract classes in Object Model
  - each language binding defines a mapping between both

Subtyping and Inheritance of Behavior

- Type-subtype (is-a) relationships represented in graphs

```
interface Employee { ... }
interface Professor : Employee { ... }
interface AssociateProfessor : Professor { ... }
```

- An instance of a subtype is also logically an instance of the supertype
- An object's most specific type: type describing all its behavior and properties
- A subtype's interface may add characteristics to those defined on its supertypes
- Can also specialize state and behavior

```
class SalariedEmployee : Employee { ... }
class HourlyEmployee : Employee { ... }
```


Subtyping and Inheritance of Behavior (cont.)

- Multiple inheritance of object behavior
  - a type could inherit operations of the same name but different parameters
  - precluded by model: disallows name overloading during inheritance
- Classes are directly instantiable types, interfaces not
- Subtyping pertains to inheritance of behavior only: classes and interfaces may inherit from interfaces
- Inefficiencies and ambiguities in multiple inheritance of state ⇒ interfaces and classes may not inherit from classes

Inheritance of State: Extends

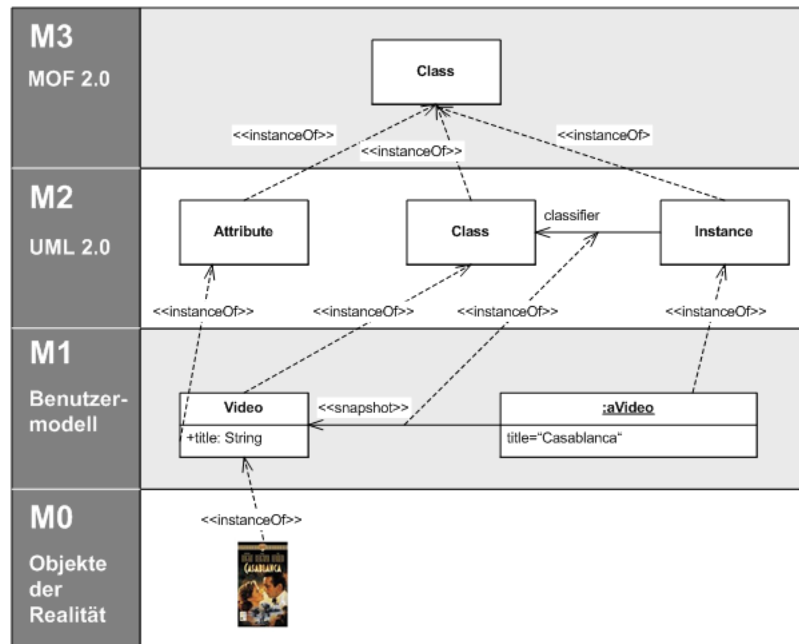
- In addition to isa, for inheritance of state
- Applies only to object types: only classes (not literals) may inherit state
- Single inheritance relationship between two classes: subordinate class inherits all properties and behavior of the class it extends

```
class Person {
  attribute string name;
  attribute Date birthDate;
};
class EmployeePerson extends Person : Employee {
  attribute Date hireDate;
  attribute Currency payRate;
  relationship Manager boss inverse Manager::subordinates;
};
class ManagerPerson extends EmployeePerson : Manager {
  relationship set<Employee> subordinates inverse Employee::boss;
};
```


Extents and Keys

- Extent of a type: set of all instances of the type in a particular DB
- If type A is a subtype of type B $\Rightarrow$ extent of A is a subset of extent of B
- Maintaining the extent of a type is optional ($\neq$ RDBMS)
- Extent maintenance
  - insert newly created instances, remove instances
  - create and manage indexes to speed up access
- Key: instances of a type uniquely identified by values of some property or set of properties
- As in relational model: simple vs compound keys
- A type must have an extent to have a key

OMG's Meta-Object Facility



Object Creation

- Objects created by invoking creation operations on factory interfaces provided on factory objects supplied by language binding

```
interface ObjectFactory {
  Object new();
};
```

- All objects have the following interface

```
interface Object {
  enum Lock_Type { read, write, upgrade };
  exception LockNotGranted {};
  void lock(in Lock_Type mode)
    raises (LockNotGranted); // obtain a lock
  boolean try_lock (in Lock_Type mode); // True if lock obtained
  boolean same_as (in Object anObject); // identity comparison
  Object copy();
  void delete()}; // remove objects from memory and the DB
};
```

- Critique: transaction deadlock exception is missing

Object Identifiers and Names

- Objects retain same object identifiers for its entire lifetime
- Literals do not have their own identifiers
- Object identifiers geneterated the the ODBMS, not applications
- Bit pattern representing an object identifier ⇒ implementation issue
- An object may be given name(s) meaningful for users
- ODBMS provides a function mapping an object name to an object
- Names used to refer to “root” objects: provide entry points to the DB

Object Lifetime

- Specifies, at object creation, how memory allocated to the object is managed
- **Transient**: allocated in memory managed by the PL run-time system, e.g.,
  - declared in a procedure and allocated on the stack
  - for a process and allocated on the heap
- **Persistent**: allocated in memory managed by the ODBMS run-time system
  - continue to exists after procedure or process that creates it terminates
- Object lifetimes independent of types
  - a type may have transient and persistent instances
  - persistent and transient objects manipulated using the same operations
- ⇒ Persistence is orthogonal to type
- This addresses the “impedance mismatch” of the relational model:
  - SQL for defining and using persistent data
  - PL for defining and using transient data

Collections

- Generic type generators for structured values
- All members of a collection object must be of the same type
  - Critique: must have a common ancestor type
- Collections
  - Set<t>: unordered collection of elements, no duplicate allowed
  - Bag<t>: unordered collection of elements, may contain duplicates
  - Array<t>: ordered collection of elements
  - List<t>: dynamically sized ordered collection of elements that can be located by oposition
  - Dictionary<t,v>: unordered sequence of key-value pairs with no duplicate keys

Manipulating Collection Objects

```
interface CollectionFactory : ObjectFactory {
  Collection new_of_size(in long size);
};
interface Collection : Object {
  exception InvalidCollectionType {};
  exception ElementNotFound {any element; };
  unsigned long cardinality();
  boolean is_empty();
  boolean is_ordered();
  boolean allows_duplicates();
  boolean contains_element(in any element);
  void insert_element(in any element);
  void remove_element(in any element) raises(ElementNotFound);
  Iterator create_iterator(in boolean stable);
  BidirectionalIterator create_bidirectional_iterator(in
    boolean stable) raises(InvalidCollectionType);
};
```

Manipulating Iterators

```
interface Iterator {
  exception NoMoreElements {};
  exception InvalidCollectionType {};
  boolean is_stable();
  boolean at_end();
  void reset();
  any get_element() raises(NoMoreElements);
  void next_position() raises(NoMoreElements);
  void replace_element(in any element)
    raises(InvalidCollectionType);
};
interface BidirectionalIterator : Iterator {
  boolean at_beginning();
  void previous_position() raises(NoMoreElements);
};
```


Structured Objects

- `Date`
- `Time`
  - internally stored in GMT
  - time zones : number of hours that must be added/subtracted to local time to get time in Greenwich, England
- `Timestamp`: encapsulated `Date` and `Time`
- `Interval`: duration in time
  - used to perform some operations on `Time` and `Timestamp` objects
  - created with the `substract_time` operation in `Time`

Structured Objects: Example

```
interface Date : Object {
  typedef unsigned short ushort;
  enum Weekday {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
  enum Month {January,February,March,April,May,June,July,August,
    September,October,November,December };
  struct asValue { ushort month, day, year; };
  ushort year();
  ushort month();
  ushort day();
  ushort day_of_year();
  ushort month_of_year();
  ushort day_of_week();
  boolean is_leap_year();
  boolean is_equal(in Date a_date);
  boolean is_greater(in Date a_date);
  boolean is_greater_or_equal(in Date a_date);
  boolean is_less(in Date a_date);
  boolean is_less_or_equal(in Date a_date);
  boolean is_between(in Date a_date,in Date b_date);
  Date next(in Weekday day);
  Date previous(in Weekday day);
  Date add_days(in long days);
  Date substract_days(in long days);
  long substract_date(in Date a_date);
}
```


Literals

Atomic Literals

- `long`, `short`, `unsigned long`, `unsigned short`, `float`, `double`, `boolean`, `octet`, `char` (character), `string`, `enum` (enumeration)
- All supported by OMG's Interface Definition Language (IDL)
- An `enum` is a type generator: named literal type that can take only the values listed in the declaration

```
attribute enum gender { male, female };
attribute enum state_code { AK,AL,AR,AZ,CA,...,WY };
```

Collection Literals

- `set<t>`, `bag<t>`, `array<t>`, `list<t>`, `dictionary<t,v>`
- Analogous to collection objects, but without identifiers
- Their elements can be of literal or object types

Table Type

- Express SQL tables, semantically equivalent to a collection of structs

Structured Literals (Structures)

- Fixed number of elements, each with a variable name and can contain a literal value or an object
- Predefined structures: `date`, `interval`, `time`, `timestamp`

```
struct Address {
  string dorm_name;
  string room_no;
}
attribute Address dorm_address
```

- May be freely composed
  - sets of structures, structures of sets, arrays of structures, ...

```
struct Degree {
  string school_name;
  string degree_type;
  unsigned short degree_year;
}
typedef list<Degree> Degrees;
```


Null Literals

- For every literal type (e.g., `float`, `set<>`) $\Rightarrow$ another literal type supporting null value (e.g., `nullable_float`, `nullable_set<>`)
- Nullable type = literal type augmented by the null value “nill”
- Semantics of null as defined by SQL-92

Modeling State: Attributes

```
class Person {
  attribute short age;
  attribute string name;
  attribute enum gender {male, female};
  attribute Address home_address;
  attribute set<Phone_no> phones;
  attribute Department dept;
}
```

- An attribute value: a literal or an object identifier
- Attribute ≠ data structure:
  - attribute = abstract, data structure = physical representation
  - attributes may be implemented by data structures or by methods (e.g., age)
- Attributes are not "first class"

Modeling State: Relationships

- Binary relationships only
- Between objects only, literals cannot participate in relationships
- May be one-to-one, one-to-many, many-to-many
- Relationships are not "first class": not objects, without identifier
- Defined implicitly by declaration of traversal paths declared in pairs

```
interface Professor { ...
  relationship set<Course> teaches inverse Course::is_taught_by;
}
interface Course { ...
  relationship Professor is_taught_by inverse Professor::teaches;
}
```

- Traversal paths "many" may be unordered or ordered: indicated by type of collection

Modeling State: Relationships (cont.)

- ODBMS responsible for maintaining referential integrity
  - if object is deleted, any traversal path to that object is also deleted
- Ensures that applications cannot dereference traversal paths leading to nonexistent objects
- Relationships vs references

```
attribute Student top_of_class
```

- Object-valued attributes reference an object, without inverse traversal path or referential integrity: “unidirectional relationships”

Implementation of Relationships

- Encapsulated by public operations that
  - form and drop members of the relationship
  - provide access and manage referential integrity constraints
- Possible exception: set already contains reference to the object

```
interface Course {
  ...
  attribute Professor is_taught_by;
  void form_is_taught_by(in Professor aProfessor);
  void drop_is_taught_by(in Professor aProfessor);
}
interface Professor {
  ...
  readonly attribute set<Course> teaches;
  void form_teaches(in Course aCourse) raises(IntegrityError);
  void drop_teaches(in Course aCourse);
  void add_teaches(in Course aCourse) raises(IntegrityError);
  void remove_teaches(in Course aCourse);
}
```

Modeling Behavior: Operations

- Specification identical to OMG CORBA specification for operations
- Operation signatures: name, name and type of each argument, types of values returned, names of any exceptions (error conditions) that may be raised
- An operation defined on only a single type
- An operation name need be unique only within a single type defintion
  - Critique: static polymorphism not supported
- Overloaded operations can be defined in different types
- Operations name resolution or operation dispatching: based on the most specific type of the object supplied as first argument of the call
- Single-dispatch model ⇒ consistency with C++, Java, and Smalltalk
- Object model assumes sequential execution of operations

Metadata

- Descriptive information about DB objects, defines the schema of the DB
- Used by the ODBMS to define the structure of the DB and at runtime to guide its access to the database
- Stored in an ODL Schema Repository, accessible to tools and applications
- Meta-produced, e.g., during ODL source compilation
- Defines interfaces for, e.g., `MetaObject`, `Specifier`, `Operation`, `Exception`, `Constant`, `Property`, `Attribute`, `Relationship`, `Type`, `Interface`, `Class`, `Collection`, `Literal`, `Expression`, ...

Metadata: Examples

```
interface MetaObject {
  attribute string name;
  attribute string comment;
  relationship DefiningScope definedIn inverse DefiningScope::defines;
}
interface Operation : ScopedMetaObject {
  relationship list<Parameter> signature inverse Parameter::operation;
  relationship Type result inverse Type::operations;
  relationship list<Exception> exceptions inverse Exception::operations;
}
interface Property : MetaObject {
  relationship Type type inverse Type::properties;
}
interface Attribute : Property {
  attribute boolean isReadOnly;
}
enum Cardinality {c1_1, c1_N, cN_1, cN_M};
interface Relationship : Property {
  exception integrityError {};
  relationship Relationship traversal inverse Relationship::traversal;
  Cardinality getCardinality();
}
```


Object Specification Languages

- Language independent
- Used to define the schema, operations, and state of an object DB
- Objectives of these languages
  - facilitate portability of DBs across ODMG-compliant implementations
  - provide a step toward interoperability of ODBMS's from multiple vendors
- Two specification languages
  - Object Definition Language (ODL)
  - Object Interchange Format (OIF)

Object Definition Language: Principles

- Support all semantic constructs of the ODMG object model
- Not a full PL, rather a definition language for object specifications
- Programing language independent
- Compatible with the OMG's Interface Definition Language
- Extensible for future functionality and for physical optimizations
- Practical to application developpers, easily implementable by ODBMS vendors

Object Definition Language

- Defines characteristics of types, including properties and operations
- Defines only the signatures of operations, not methods of these operations
- ODMG does not provide an Object Manipulation Language (as RDBMS do)
- Define standard APIs to bind conformant ODBMSs to C++, Java, and Smalltalk
- Provides a degree of insulation for application against variations in PLs and underlying ODBMS products

Class Definition: Example

```
class Person
( extent people ) {
  attribute string name;
  attribute struct Address { unsigned short number, string street,
    string cityName } address;
  relationship Person spouse inverse Person::spouse;
  relationship set<Person> children inverse Person::parents;
  relationship list<Person> parents inverse Person::children;
  void birth (in string name);
  boolean marriage (in string personName) raises (noSuchPerson);
  unsigned short ancestors (out set<Person> allAncestors) raises (noSuchPerson);
  void move (in string newAdress);
}
class City
( extent cities ) {
  attribute unsigned short cityCode;
  attribute string name;
  attribute set<Person> population;
}
```


ODMG Schemas: Notations

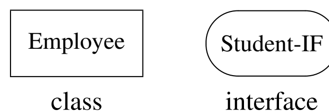
- Object-valued attributes (unidirectional relationships)

  1: ⟶ n: ⟶⟶

- Relationships

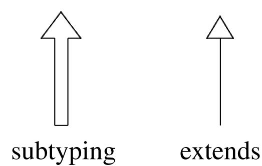
  1-1: ⟷ 1-n: ⟷⟶ m-n: ⟵⟵⟶⟶

- Class and interfaces



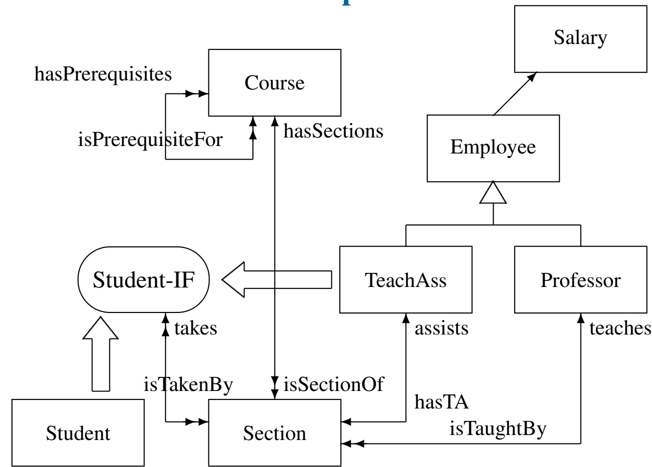
class interface

- Subtyping and extents



subtyping extends

ODMG: Example Schema



ODL: Example Schema

```
class Course
( extent courses ) {
  attribute string name;
  attribute string number;
  relationship list<Section> hasSections inverse Section::isSectionOf;
  relationship set<Course> hasPrerequisites inverse Course::isPrerequisiteFor;
  relationship set<Course> isPrerequisiteFor inverse Course::hasPrerequisites;
  boolean offer (in unsigned short semester) raises (alreadyOffered);
  boolean drop (in unsigned short semester) raises (notOffered);
}
class Section
( extent sections ) {
  attribute string number;
  relationship Professor isTaughtBy inverse Professor::teaches;
  relationship TeachAssist hasTA inverse TeachAssist::assists;
  relationship Course isSectionOf inverse Course::hasSections;
  relationship set<Student> isTakenBy inverse Student::takes;
}
```

ODL: Example Schema, cont.

```
class Salary {
  attribute float base;
  attribute float overtime;
  attribute float bonus;
}
class Employee
( extent employees ) {
  attribute string name;
  attribute short id;
  attribute Salary annualSalary;
  void hire();
  void fire() raises (noSuchEmployee);
}
class Professor extends Employee
( extent professors ) {
  attribute enum Rank {full, associate, assistant} rank;
  relationship set<Section> teaches inverse Section::isTaughtBy;
  short grantTenure() raises (ineligibleForTenure);
}
```


ODL: Example Schema, cont.

```
interface Student-IF {
  struct Address { string College, string roomNumber };
  attribute string name;
  attribute string studentId;
  attribute Address dormAddress;
  boolean registerForCourse(in unsigned short course, in unsigned short section)
    raises (unsatisfiedPrerequisites, sectionFull, CourseFull);
  void dropCourse(in unsigned short Course)
    raises (notRegisteredForThatCourse);
  void assignMajor(in unsigned short Department);
  short transfer (in unsigned short oldSection, in unsigned short newSection)
    raises (sectionFull,notRegisteredInSection);
}
```

ODL: Example Schema, cont.

```
class TeachAssist extends Employee : Student-IF {
  relationship Section assists inverse Section::hasTA;
  attribute string name;
  attribute string studentId;
  attribute struct Address dormAddress;
  relationship set<Section> takes inverse Section::isTakenBy;
}
class Student : Student-IF
(extent students) {
  attribute string name;
  attribute string studentId;
  attribute struct Address dormAddress;
  relationship set<Section> takes inverse Section::isTakenBy;
}
```


Object Interchange Format

- Specification language for dumping and loading an ODB to or from a (set of) file(s)
- Can be used to
  - exchange objects between DBs
  - seed DBs
  - provide DB documentation
  - drive DB test suites
- Principles
  - support all ODB states compliant with the ODMG object model and ODL schema definition
  - not a full programming language, rather a specification language for persistent objects and their states
  - designed according to the related standards STEP or ANSI
  - needs no keywords other than the type, attribute, and relationship identifiers provided with the ODL definition of a DB schema

Object Interchange Format: Object Database States

- State of all objects in a DB characterized by
  - object identifiers (with object tag names)
  - type bindings
  - attribute values
  - links to other objects
- Critiques
  - Object names as entry points in the database are missing
  - No correlation beween OIF and the corresponding schema

Object Interchange Format: Object and Attributes

Object Definitions

- OIDs specified with object tag names unique to the OIF files(s)
- Object definitions: `Jack Person {}`
- Physical clustering: `Paul (Jack) Person {}`
- Copy initialization: `MacBain (MacPerth) Company(MacPerth)`

Attribute Initialization

- Format for dumping Boolean, character, integer, float, and string literals is specified
- Range overflow controlled for integers and floats
- Critique
  - Format for dumping structured literals Date, Time, Timestamp, and Interval, as well as enumeration types not specified

Initializing Attributes: Example

```
struct PhoneNumber {
  unsigned short CountryCode;
  unsigned short AreaCode;
  unsigned short PersonCode;
}
struct Address {
  string Street;
  string City;
  PhoneNumber Phone;
}
interface Person {
  attribute string Name;
  attribute Address PersonAddress
}
Sarah Person { Name "Sarah",
  PersonAdress { Street "Willow Road",
    City "Palo Alto",
    Phone { CountryCode 1,
      AreaCode 415,
      PersonCode 1234 } } }
```


Initializing Arrays

```
interface Engineer {
  attribute unsigned long PhoneNo[3];
}
```

- Fields of the array indexed starting from zero, attributes not specified remain uninitialized

```
Jane Engineer { PhoneNo { [0] 450123, [2] 270456} }
```

- Index specifier can be omitted ⇒ continuous sequence starting from zero

```
Marc Engineer { PhoneNo { 12345, 234234 } }
```

- Dynamic arrays: if one of the indices used in the object definition exceeds the current size of the variable size array, the array will be resized

```
struct Point {
  float X; float Y;
}
interface Polygon {
  attribute array<Point> RefPoints;
}
P1 Polygon { RefPoints { [5] { X 7.5, Y 12.0 },
                         [12] { X 22.5, Y 23.0 } } }
```

Initializing Arrays and Collections

Multidimensional arrays

```
interface PolygonSet {
  attribute array<float> PolygonRefPoints[10];
}
P2 PolygonSet { PolygonRefPoints {
 [0] { [0] 12.5 , [1] 8.98, ...}
  ...
 [1] { [0] 23.7 , [1] 3.33 } } }
```

Collections

```
interface Professor : Person {
  attribute set<string> Degrees;
}
Feynman Professor { Degrees { "Masters", "PhD" }
```


Initializing Links

```
interface Person {
  relationship Company Employer inverse Company::Employees;
  relationship Company Property inverse Company::Owner;
}
interface Company {
  relationship set<Person> Employees inverse Person::Employer;
  relationship Person Owner inverse Person::Property;
}
```

- Links for relationships with cardinality 1 treated as attributes

```
Jack Person { Employer McPerth }
```

- Links for relationships with cardinality m treated as collections

```
McPerth Company { Employees { Jack, Joe, Jim } }
```

- Example of a cyclic link

```
Jack Person { Employer McPerth }
McPerth Company { Owner Jack }
```

- Critique: not explicitly stated whether both traversal directions of a link have to be dumped of if one suffices

Command Line Utilities

Dump database

- To create an OIF representation of the specified database

```
odbdump <database name>
```

- Object tag names created automatically using implementation-dependent name generation algorithms

Load database

- Populates the database with objects defined in the specified files

```
odbload <database name> <file1> .. <filen>
```


Object Interchange Format: Example

```
Eve Person {
  name "Eve",
  address { number 25, street "Apple Road", city "Paradise" },
  spouse Adam,
  children { Cain, Abel },
  parents { God }
  }
Paradise City {
 cityCode 6969,
 name "Paradise",
 population { Adam, Eve, Cain, Abel },
}
```

Object Query Language: Principles

- Relies on ODMG's object model
- Close to SQL 92
- Provides high-level primitives to deal with sets of objects, structures, lists, arrays
- Functional language where operators can freely be composed
- Not computationally complete
- Can be invoked from within PL for which a binding is defined, can also invoke operations programmed in these languages
- Does not provide explicit update operators
- Provides declarative access to objects

Object Query Language: Example DB

- Types `Person` and `Employee` with extents `Persons` and `Employees`
- One of these persons is the chairman: entry point `Chairman`
- `Person` defines
  - attributes: `name`, `birthdate`, and `salary`
  - operation: `age`
- `Employee` subtype of `Person`, defines
  - attribute: `empNo`
  - relationship: `subordinates`
  - operation: `seniority`

Query Input and Result

- Query: function that delivers an object or a literal
- Type of a query inferred from query expression, schema type declarations, and type of named objects and literals
- Some objects generated by the QL interpreter, others produced from current DB
- A query may return
  - a collection of objects with identity

    ```
    select x from Persons x where x.name="Pat"
    ```

  - an object with identity

    ```
    element(select x from Employees x where x.empNo=123456)
    ```

  - a collection of literals

    ```
    select x.empNo from Employees x where x.name="Pat"
    ```

  - a literal

    ```
    Chairman.salary
    ```


Type of Query Result

- The set of ages of persons named Pat

  ```
  select distinct x.age
  from Persons x
  where x.name = "Pat"
  ```

  returns a literal of type `set<integer>`

- The distinct couples of age and sex for persons named Pat

  ```
  select distinct struct(a:x.age, s:x.sex)
  from Persons x
  where x.name = "Pat"
  ```

  returns a literal of type `set<struct(a:x.age,s:x.sex)>`

Orthogonality

- The name of each employee with the set of his highly paid subordinates

```
select distinct struct(name:x.name, hps:
                select y from x.subordinates as y
                where y.salary > 100000))
from Employees x
```

  - returns a literal of type `set<struct(name:string, hps:bag<Employee>)>`
  - nested query in `select` clause

- The couples of age and sex for persons having seniority of 10 and named Pat

```
select struct(a:x.age,s:x.sex)
from (select y from Employees y where y.seniority=10) as x
where x.name = "Pat"
```

  - nested query in `from` clause

Compatibility: Definition

Defined recursively

(1) `t` is compatible with `t`

(2) If `t` is compatible with `t'` then

- `set(t)` is compatible with `set(t')`
- `bag(t)` is compatible with `bag(t')`
- `list(t)` is compatible with `list(t')`
- `array(t)` is compatible with `array(t')`

(3) If there exist `t` such that `t` is a supertype of `t1` and `t2`, then `t1` and `t2` are compatible

Compatibility: Consequences

- Literal types are not compatible with object types
- Atomic literals types are compatible only if they are the same
- Structured literal types are compatible only if they have a common ancestor
- Collection literal types are compatible if they are of the same collection and the types of their memebers are compatible
- Atomic object types are compatible only if they have a common ancestor
- Collection object types are compatible if they are of the same collection and the types of their members are compatible

Navigations: Path Expressions

- "." or "->" notation to go inside complex objets and follow 1-1 relationships
- For a person `p`, give the name of the city where `p`'s spouse lives

```
p.spouse.address.city.name
```

- For m-n relationships: use `select from where` clause as in SQL
- The names of the children of person `p`: result of type `bag<string>`

```
select c.name
from p.children c
```

- Result of type `set<string>`

```
select distinct c.name
from p.children c
```

- The set of addresses of the children of each `Person` in the DB

```
select c.address
from Person p, p.children c
```

Predicates

- Restrict previous result to people living on Main Street, having at least two children who do not live in the same city as their parents

```
select c.address
from Person p, p.children c
where p.address.street = "Main Street" and
     count(p.children) >= 2 and
     c.address.city != p.address.city
```

- Joins: In the `from` clause, collections not directly related
- Example: the people who bear the name of a flower

```
select p
from Person p, Flowers c
where p.name = f.name
```


Null Values

- Access to properties (with `.` or `->`) applied to an `Undefined` left operand produce `Undefined` as result
- Comparison operations with either of both operands being `Undefined` produce `False` as result
- `is_undefined(Undefined)` returns `true` `is_defined(Undefined)` returns `false`
- Any other operation with any `Undefined` operans results in a run-time error

Null Values: Examples

- Three employees: one lives in Paris, another in Milan, third has a nil address

```
select e
from Employees e
where e.address.city = "Paris"
```

returns a bag containing the employee living in Paris

```
select e.address.city
from Employees e
```

generates a run-time error

```
select e.address.city
from Employees e
where is_defined(e.address.city)
```

returns a bag containing the city names Paris and Milan

```
select e
from Employees e
where not(e.address.city = Paris)
```

returns a bag containing the employee living in Milan and the one without address

Method Invocation

- Same notation as accessing an attribute or traversing a relationship
- Parameters given between parentheses
- Frees the user from knowing whether a property is stored or computed
- The age of the oldest child of all persons with name Paul

```
select max(select c.age from p.children c)
from Persons p
where p.name = "Paul"
```

- Method `oldestChild` returns an object of class Person, `livesIn` is a method with one parameter
- The set of street names where the oldest children of Parisian people are living

```
select p.oldestChild.address.street
from Persons p
where p.livesIn("Paris")
```

Polymorphism

- Major contribution of object orientation: manipulate polymorphic collections and, with late binding, carry out generic actions on their elements
- Suppose set `Persons` contains objects of class `Person`, `Employee`, and `Student`
- Properties of a subclass cannot be applied to an element of `Person` except in late binding or explicit class declaration
- Late binding: give activities of each person

```
select p.activities
from Persons p
```

  Depending on the kind of person of the current `p` the right method `activities` is called
- Class indication: grades of students (assuming that only students spend their time in following a course of study)

```
select ((Student)p).grade
from Persons p
where "course of study" in p.activities
```


Collection Expressions

- Existential and universal quantification: return a Boolean value

```
exists x in Doe.takes: x.taught_by.name = "Turing"
for all x in Students: x.student_id > 0
```

- Membership testing: return a Boolean value

```
Doe in Students
```

- Aggregate operators: min, max, count, sum, avg

```
max( select salary from Professors )
```

Cartesian Product

- Give the couples of student name and the name of the full professors from which they take classes

```
select couple(student: x.name, professor: z.name)
from Students as x, x.takes as y, y.taught_by as z
where z.rank = "full professor"
```

- Type of result: bag of objects of type `couple`
- A different query

```
select *
from Students as x, x.takes as y, y.taught_by as z
where z.rank = "full professor"
```

- Type of result: bag of structures, giving for each student object the section object followed by the student and the full professor object teaching in this section

```
bag<struct(x:Student, y:Section, z:Professor)>
```


Group-by Operator

- Partition employees according to salary in 3 groups: low, medium, and high

```
select *
from Employees e
group by low:    salary < 1000,
         medium: salary >= 1000 and salary < 10000,
         high:   salary >= 10000
```

- Gives a set of three elements, each with a property `partition` contaning the bag of employees entering in the category
- Type of result:

```
set<struct(low:boolean, medium:boolean, high:boolean,
           partition: bag<struct(e:Employee)> )>
```

Group-by, cont.

- Give for each department the employees working in this department

```
select department,
       employees: (select p from partition p)
from Employees e
group by department: e.deptno
```

- Result type:

```
set<struct(department: integer,
           employees: bag<struct(e:Employee)> )>
```


Group-by and Having Operator

- Give a set of couples composed of a department and the average of the salaries of the employees working in that department, when this average is more than 30000

```
select department,
       avg_salary: avg(select e.salary from partition)
from Employees e
group by department: e.deptno
having avg(select x.e.salary from partition x) > 30000
```

- Result type: `bag<struct(department:integer, avg_salary:float)>`

Indexed Collection Expressions

- Third prerequisite of course Math 101

```
element( select x
         from Courses x
         where x.name="Math" and x.number="101" ).requires[2]
```

- First prerequisite of course Math 101

```
first ( element( select x
                 from Courses x
                 where x.name="Math" and x.number="101" ).requires )
```

- First three prerequisites of course Math 101

```
element( select x
         from Courses x
         where x.name="Math" and x.number="101" ).requires[0:2]
```


Set Expressions

- Union, intersection, difference (`except`)
- The set of students who are not teaching assistants

```
Students except TechAss
```

- Defined for set and for bags
  - `bag(2,2,3,3,3) union bag(2,3,3,3)` returns `bag(2,2,3,3,3,2,3,3,3)`
  - `bag(2,2,3,3,3) intersect bag(2,3,3,3)` returns `bag(2,3,3,3)`
  - `bag(2,2,3,3,3) except bag(2,3,3,3)` returns `bag(2)`
- Set inclusion: `<`, `=<`, `>`, `>=`,
  - `set(1,2,3) < set(3,4,2,1)` is true
- Converson expressions
  - `element(e)`: extracts the element of a singleton
  - `listoset(e)`: turns a list into a set
  - `distinct(e)`: removes duplicates
  - `flatten(e)` : flattens a collection of collections

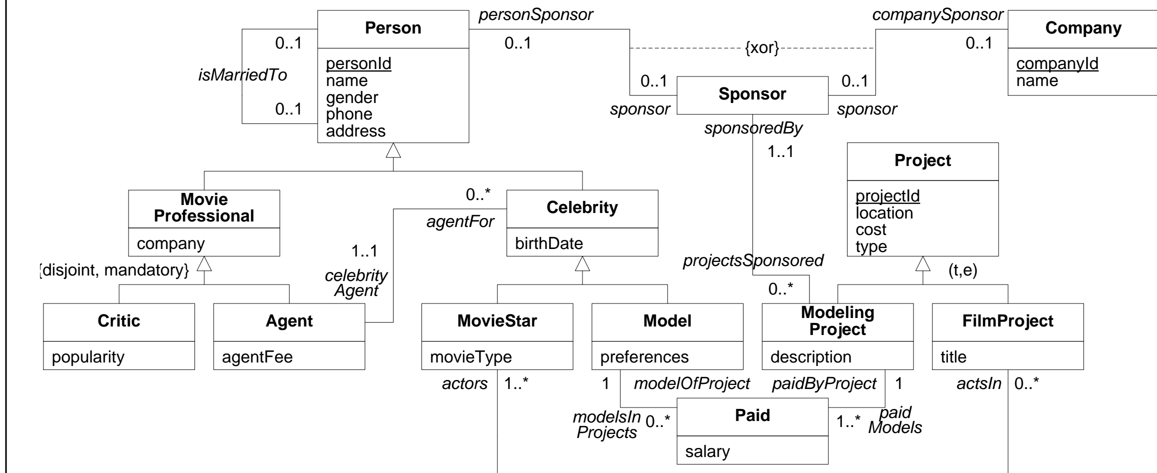
ODBMSs Today: Summary

- Provide a more powerful and flexible alternative to relational DBMSs
- Early commercial products appeared in the 1990s, open source implementations in the 2000s
- Very few of the original products remain on the market, in particular due to merging and acquisitions
- OODBMS never get momentum and remained a niche market
- Current systems differ considerably in the functions provided
- Their performance and scalability should be assessed
- Building complex applications requires significant expertise
- Proprietary application interface and class libraries ⇒ porting an application requires significant re-work

Object Databases: Topics

- Object-Oriented Databases
  - ODMG
  - ➡ **Linq**
- Object-Relational Databases
  - SQL
  - Oracle
- Summary

Hollywood Database Example



81

Language INtegratedQuery (LINQ)

- **Declarative query language** seamlessly integrated into the OOPLs of the .NET Framework, which include C# and Visual Basic
- As OQL, **relies on the well-typed features of the underlying OOPL**, e.g., applying aggregate operators to a collection of the appropriate type
- **Can query collections of objects, tuples, or XML elements**
- Here we focus on querying a collection of objects using the C# language
- Therefore, the fields of the classes are implemented as properties in C#
- These are conventionally indicated by a method with an initial uppercase letter that defines get and set accessors for the private instance field that the property is encapsulating
- The private instance field usually has the same name but starts with an initial lowercase letter

Queries

```
from specifies collections relevant to answering the query and
     identifiers to iterate over those collections
where specifies conditions for filtering objects
select defines the structure of the data to be returned
```

- LINQ uses path expressions with dot notation to access properties or methods associated with objects
- Find the titles of the movies filmed in Phoenix

```
from f in filmProjects
where f.Location == "Phoenix"
select f.Title;
```

- `from` and `where` clauses are not limited to occur once in the expression, and the clauses can be intertwined
- Find the actors who starred in projects filmed in Phoenix

```
from f in filmProjects
where f.Location == "Phoenix"
from a in f.Actors
select a.Name;
```


Alternative Syntax

- LINQ's querying capabilities are provided by the underlying .NET framework
- **Extension method**: static method in a static class that can be invoked as if it were an instance method on the object itself
- The `from-where-select` syntax is called **query comprehension syntax**
- This is compiled into an **extension method syntax**: e.g., `Where` and `Select` are extension methods
- Underlying extension method syntax for the previous query:

```
filmProjects.Where(f => f.Location == "Phoenix").Select(f => f.Title);
```

- The `Where` extension method is invoked on the `filmProjects` extent, taking as an argument an anonymous function that filters the extent
- The `=>` syntax is read as "goes to" and represents a way to pass an argument into an anonymous function similar to lambda expressions
- The `Select` method returns the `Title` of the filtered collection
- Extension methods that perform operations on LINQ query expressions are called **query operators**
- Linq queries typically use a combination of query comprehension syntax with query operators

Query Results (1)

- Variables are implicitly typed
- Defining the variable `phxFilms` as the collection of film projects located in Phoenix

```
var phxFilms =
    from f in filmProjects
    where f.Location == "Phoenix"
    select f;
```

- In C#, the type of the collection returned is usually an instance of `IEnumerable<T>`
- To remove duplicate elements from a collection in LINQ, the `Distinct` operator is used
- The set of movie stars that appeared in Phoenix film projects

```
( from f in phxFilms
from a in f.Actors
select a).Distinct();
```


Query Results (2)

- To return a single object, with its type being implicitly defined, the `Single` operator is used
- Define a variable `daysOfThunder` that represents the unique `FilmProject` object with the title "Days of Thunder"

```
var daysOfThunder =
    ( from f in filmProjects
    where f.Title == "Days of Thunder"
    select f ).Single();
```

- An **exception is thrown** if there is more than one object in the collection
- Query expressions return structured results using anonymous types and object initialization
- The following query returns an anonymous type that contains the names of movie stars in the `daysOfThunder` film along with the name of each movie star's agent

```
from a in daysOfThunder.Actors
select new { movieStarName = a.Name, agentName = a.CelebrityAgent.Name };
```

- The `new` operator in the select clause creates an anonymous type
- The braces enclose the initialization of the anonymous object using names

Methods Calls and Embedded Queries

- Queries can use **method calls**
- Finds celebrities who are younger than 30

```
from c in celebrities
where c.Age() < 30
select c.Name;
```

- LINQ supports **embedded queries** in the from or select clauses
- Find the Phoenix film projects with a cost greater than $10 million

```
from phx in ( from f in filmProjects
              where f.Location == "Phoenix" select f)
where phx.Cost > 10,000,000
select new { phx.Title, phx.Cost };
```

- Names of the celebrities that an agent manages

```
from a in agents
select new { agentName = a.Name,
    agentForCelebrities = (from c in a.AgentFor select c.Name) };
```


Set Membership and Quantification (1)

- The `Contains` operator supports set membership
- Does the collection `phxFilms` contain the film project `daysOfThunder`?

```
phxFilms.Contains(daysOfThunder);
```

- The `Any` and `All` operators supports existential and universal quantification, respectively
- Is there any `f` in `filmProjects` having a `Location` value equal to Phoenix?

```
filmProjects.Any(f => f.Location == "Phoenix");
```

- Finds the models who have had at least one modeling project located in Phoenix

```
from m in models
where (m.ModelsInProjects).Any(p => p.PaidByProject.Location == "Phoenix")
select new {modelname = m.Name};
```

- Determine whether all `f` objects in `filmProjects` cost more than 10 million

```
filmProjects.All(f => f.Cost > 10000000);
```

Set Membership and Quantification (2)

- Find the names of movie stars for which all of their film projects cost more than $20 million

```
from m in movieStars
where (m.ActsIn).All(fp => fp.Cost > 20000000)
select m.Name;
```

- Find the sponsors that are companies for which all of their sponsored modeling projects include at least one male model

```
from s in sponsors
where s.CompanySponsor ! = null &&
     (s.ProjectsSponsored).All(mp => (mp.PaidModels).
             Any(p => p.ModelOfProject.Gender == "M"))
select s.CompanySponsor.Name;
```


Ordering

- Expressed with the `orderby` clause, which occurs before the `select` clause
- This allows the results to be ordered by data that is not contained in the query result
- Keywords `ascending` and `descending` specify the result order, with `ascending` as the default
- List the film projects filmed in Phoenix sorted in descending order by the cost of the project

```
from fp in filmProjects
where fp.Location == "Phoenix"
orderby fp.Cost descending
select new { phxTitle = fp.Title, phxCost = fp.Cost};
```

- The `orderby` clause can also be used in embedded queries
- List of movie stars in alphabetical order for each film project filmed in Phoenix

```
from fp in filmProjects
where fp.Location == "Phoenix"
orderby fp.Title ascending
select new { phxTitle = fp.Title, phxMovieStars =
    ( from ms in fp.Actors orderby ms.Name select ms.Name ) };
```

Using Collections

- Operators such as `First`, `Last`, and `Take` are used for accessing elements in a collection
- Finds the highest paid Phoenix model

```
( from m in models
from p in m.ModelsInProjects
where p.PaidByProject.Location == "Phoenix"
orderby p.Salary ascending
select new {modelName = m.Name, modelSalary = p.Salary}).Last();
```

- Find the top ten models based on the salary received for a modeling project in Phoenix

```
( from m in models
from p in m.ModelsInProjects
where p.PaidByProject.Location == "Phoenix"
orderby p.Salary descending
select new {modelName = m.Name, modelSalary = p.Salary}).Take(10);
```

- Query operators for collections include the set operations `Union`, `Intersect`, and `Except` (difference)

Aggregation

- Determine the number of films for movie stars

```
from m in movieStars
orderby m.Name
select new {name = m.Name, filmCount = m.ActsIn.Count()};
```

- Find the total salary for a model

```
from m in models
select new { name = m.Name, salaryTotal =
    ( from p in m.ModelsInProjects select p.Salary ).Sum()};
```

- An anonymous function can be used to generate the numeric collection as an argument to the operator
- Find the total cost of Phoenix film projects

```
phxFilms.Sum(phx => phx.Cost);
```

- A `let` clause to define a variable for the scope of a query
- Order the results of the total model salary in descending order

```
from m in models
let salaryTotal = m.ModelsInProjects.Sum(p => p.Salary)
orderby salaryTotal descending
select new { name = m.Name, salaryTotal };
```

Grouping (1)

- A `group` clause specifies **what to group** and **what to group by**
- Returns a collection of groups, each group has a `Key` property consisting of the value on which the group was created
- Group film projects by their location

```
var filmsByLocation =
    from f in filmProjects
    group f by f.Location;
```

- Give the location, count of the number of films associated with that location, and the group of films

```
from locFilms in filmsByLocation
orderby locFilms.Key
select new { location = locFilms.Key, numberOfFilms = locFilms.Count(),
    films = locFilms };
```


Grouping (2)

- Optional `into` specification allows to continue the iteration over the groups in one query expression

```
from f in filmProjects
group f by f.Location into locFilms
orderby locFilms.Key
select new { location = locFilms.Key, numberOfFilms = locFilms.Count(),
    films = locFilms };
```

- Group models by their agent, returning for each agent that represents more than three models, the count and names of the models in the group

```
from m in models
group m by m.CelebrityAgent into agentGroup
where agentGroup.Count() > 3
select new { agentName = agentGroup.Key.Name, modelCount = agentGroup.Count(),
    modelNames = (from mg in agentGroup select mg.Name) };
```

Query Execution (1)

- The query comprehension syntax is converted to an underlying extension method syntax, resulting in an expression tree that represents the definition of the query
- **Deferred query execution**: a query is not executed until it is referenced, decoupling the query construction from its execution
- Thus, a query is **similar to a database view definition**: it is materialized on each reference
- This provides an opportunity for changing the values of parameters between query invocations
- Defining a variable `locationOfInterest` and a query expression `filmsAtLocation` that returns the films filmed at the `locationOfInterest`

```
var locationOfInterest = "Phoenix";
var filmsAtLocation =
    from f in filmProjects
    where f.Location == locationOfInterest
    select f;
```

- The first time that `filmsAtLocation` is referenced it returns the films filmed in Phoenix
- If the value of the variable `locationOfInterest` is changed to San Diego, then a subsequent reference to `filmsAtLocation` returns San Diego films

Query Execution (2)

- Some query operators force the **immediate execution of a query**, such as the operators that return a scalar value (e.g., `Average`, `Count`, `Max`, `Min`, `Sum`) or a single element of a collection (e.g., `First`, `Last`, `Single`)
- The query developer can also **force** immediate execution of any query by materializing the query using a conversion operator, such as `ToArray` or `ToList`, to cache the results in the designated data structure
- These materialized results can then be reused for subsequent query references **provided that** the underlying data on which the materialized query depends has not changed
- Otherwise, the materialized results would be **out of date**

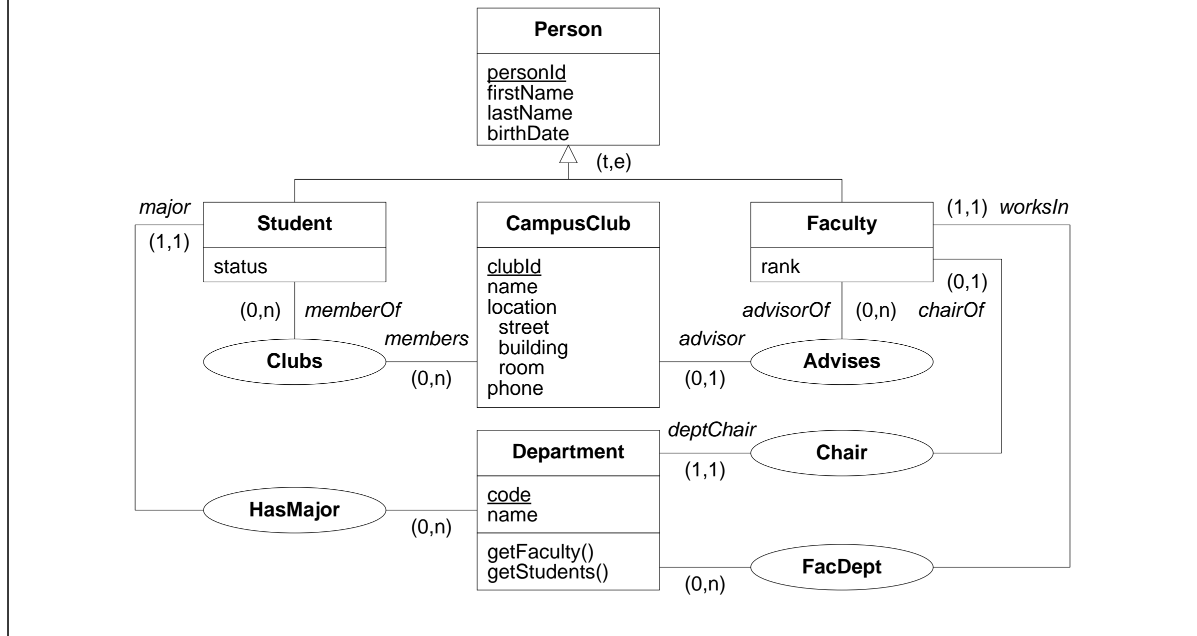
Linq: Summary

- LINQ provides a **declarative object-based query language** within the .NET Framework
- Here we illustrated how to use LINQ to **query collections of objects**
- LINQ can also be used to query collections of relational tuples or collections of XML elements, the syntax is essentially the same
- **For relational databases**, once the connection to the database is established, the attributes are accessed using familiar dot notation
- **For XML**, after loading the XML document, the methods `Descendants` and `Elements` return a collection of elements for querying
- LINQ also provides a programmer with a **declarative language for programming in C#**
- LINQ uses a combination of the familiar `from-where-select` query comprehension syntax in combination with query operators, such as
  - Set membership and quantification: `All`, `Any`, `Contains`
  - Accessing elements: `First`, `Last`, `Single`, `Take`
  - Set operations: `Distinct`, `Except`, `Intersect`, `Union`
  - Aggregation: `Average`, `Count`, `Max`, `Min`, `Sum`

Object Databases: Topics

- Object-Oriented Databases
  - ODMG
  - Linq
- Object-Relational Databases
  - ➥ **SQL**
  - Oracle
- Summary

School Database Example: Entity Relationship Schema



Row Types

- In traditional relational tables each column value in each row must be atomic (First Normal Form)
- The **row type** allows a row to be stored as a column value inside of another row

```
create table campusClub
( clubId varchar(10),
  name varchar(50) not null,
  location row (street varchar(30), building varchar(5), room varchar(5)),
  advisor varchar(11) references faculty(personId),
  primary key (clubId));
```

- Each element of the column is referred to as a **field**
- The `row` constructor is used to assign values to the fields of a row
- The types of the values must conform to the field types in the row type definition

```
insert into campusClub values ( 'CC123', 'Computer Club',
  row('Mill Avenue', 'Brickyard Building', 'Rm 222'), 'FA123');
```

- The values of a row type can be retrieved using dot notation to access the individual fields

```
select c.location.street, c.location.building, c.location.room
from campusClub c
where c.name = 'Computer Club';
```

Arrays as Collections (1)

- For collections, the SQL standard only supports the array type (not set, list, or bag)
- Arrays are useful for representing multivalued attributes
- Arrays can also be used to directly model the many side of a 1:N or M:N association
- A column in a table can be specified as an array by following the column type with the `array` keyword
- The maximum number of elements in an array is enclosed in square brackets
- The first position in an array is accessed with an index value of one
- Example: A `members` array that stores the identifiers of the club members

```
create table campusClub ( ...
  members varchar(11) array[50] references student(personId), ... );
```

- The `array` constructor is used to reserve space for an array

```
insert into campusClub values ( 'CC123', ..., array[]);
```

- The identifiers of members can be added by using the `array` constructor in an `update` statement

```
update campusClub
set members = array['ST111', 'ST222', 'ST333']
where name = 'Computer Club';
```


Arrays as Collections (2)

- Assignments can also be made to individual positions in an array by using a specific index value

```
update campusClub
set members[4] = 'ST444'
where name = 'Computer Club';
```

- An index is used to access a specific position of an array

```
select members[2]
from campusClub
where name = 'Computer Club';
```

- The `cardinality` function can be used to get the size of an array
- `cardinality(c.members)` returns the current size of the array for a row `c` in the `campusClub` table
- The `cardinality` function can be used for iterating through array contents within SQL routines
- The cardinality of an array ranges from an empty array to the maximum size specified in the definition
- Given the previous `update` statements, the current cardinality of the array is four
- If an additional `update` statement sets position six to the value `'ST666'`, the cardinality of the array becomes six, with position five set to null

User-Defined Types

- User-defined type (UDT) is the SQL standard terminology for abstract data type
- UDTs in the SQL standard are somewhat different from the strict definition of an abstract data type
- All internal attributes of a UDT and their associated methods are public and cannot be marked as protected or private as in languages such as Java or C++
- UDTs allows database developers to define new application-oriented types, beyond built-in atomic and constructed types, that can be used in the definition of relational tables
- UDTs also provide the basis for the creation of objects in the relational world
- There are two basic types of UDT in the SQL standard: the distinct type and the structured type

Distinct Types (1)

- Provides a way of attaching special meaning to an existing atomic type
- Use of a distinct type cannot be freely mixed with the atomic type on which it is based
- The distinct type essentially defines a new form of the atomic type
- Example: A table `person` table that maintains the age and weight of persons
- Columns for `personAge` and `personWeight` can be defined of type integer
- The age and weight of the person can be added even though this is meaningless
- Distinct types can be defined using the `create type` to prevent meaningless calculations with values that are of the same type but have different semantics
- Example: age and weight defined as two different distinct types, both based on the integer atomic type

```
create type age as integer final;
create type weight as integer final;
```

- Keyword `final` (required) means that a subtype of the distinct type cannot be defined

Distinct Types (2)

- Use of the above distinct types in table `person`

```
create table person ( personId varchar(3),
  personAge age, personWeight weight, primary key (personId));
```

- Retrieve the `personId` of persons younger than the person with `personId` '123'

```
select p1.personId
from person p1, person p2
where p2.personId = '123' and p1.personAge < p2.personAge;
```

- The following query is invalid since it mixes the use of age, weight, and integer values

```
select personId
from person
where (personAge * 2) < personWeight;
```

- The `cast` function can be used to solve the problem

```
select personId
from person
where cast (personAge as integer) * 2 < cast (personWeight as integer);
```

- Methods can also be defined on distinct types to create specialized operations for manipulating and comparing distinct types

Structured Types

- UDTs composed of several internal components, each component can be of a different type
- An instance of a structured type is a composite value
- Example: defining a campus club location as a structured type

```
create type locationUdt as ( street varchar(30), building varchar(5),
  room varchar(5)) not final;
```

- The components are the **attributes** of the structured type, their type can be a built-in atomic or a constructed type as well as any user-defined type
- The keywords `not final` (required, a current restriction of the SQL standard) indicates that it is possible to define subtypes of the type
- Use of this type in table `campusClub`

```
create table campusClub ( clubId varchar(10), name varchar(50) not null,
  location locationUdt,
  advisor varchar(11) references faculty(personId),
  members varchar(11) array[50] references student(personId),
  primary key (clubId));
```

Built-In Methods (1)

- Methods are associated with structured types, thus supporting encapsulation of structured types
- In the SQL standard, methods cannot be defined as procedures
- Some systems (e.g., Oracle) allow methods to be defined as functions or procedures
- Three types of built-in methods for structured types
- **Constructor function**: has the same name as the type and is used for creating instances of the type
- Must always be invoked using the new expression
- **Observer functions**: are used for retrieving the attribute values of a structured type
- There is an observer function for every attribute of the type, where the function has the same name as the attribute
- **Mutator functions**: used to modify the attribute values of a structured type
- There is a mutator function for every attribute of the type having the same name as the attribute

Built-In Methods (2)

- Observer and mutator functions are invoked using dot notation `variable.functionName`
- Functional notation `functionName(parameters)` is only used for functions that are not methods
- Routine that uses the constructor and mutator functions to create an instance of `locationUdt`

```
begin
  declare loc locationUdt;
  set loc = new locationUdt(); /* invoking the constructor function */
  set loc.street = 'Mill Avenue'; /* invoking the mutator functions */
  set loc.building = 'Brickyard Building';
  set loc.room = 'RM 222';
  insert into campusClub values ( 'CC123', 'Computer Club',
    loc, /* initializing location */
    'FA123', array[]);
end;
```

Built-In Methods (3)

- The system-defined constructor function has no parameters: the new instance has all of its attribute values set to either null or to a default value specified in the type definition
- The new instance created above is **a value and not an object** (see later)
- The value returned by a mutator function is a new instance of the type with the modified attribute value
- The query below uses of observer functions to retrieve the attribute values of the structured type

```
select name, c.location.street, c.location.building, c.location.room
from campusClub c
where name = 'Computer Club';
```

- Because of potential naming ambiguities, structured types and observer functions can only be accessed through the use of alias names in queries (i.e., `c` in the previous query)
- A reference such as `location.street` or `campusClub.location.street` is not allowed

User-Defined Methods

- Users can define their own methods on structured types
- Method **specification** (the signature) is defined in the `create type` statement
- It indicates the name of each method together with the names and types of its parameters
- The **implementation** of the method is defined separately using the `create method` statement
- Example

```
create type threeNumbers as
( one integer, two integer, three integer)
  not final
  method sum() returns integer;
create method sum() returns integer for threeNumbers
begin
  return self.one + self.two + self.three;
end
```

- Every method has one implicit parameter: the instance of the type on which the method is defined
- The value of the implicit parameter is accessed in the method implementation using the `self` keyword

Method Overriding

- Users can override the constructor function of a structured type
- This can be used to set the attribute values at the time an instance of the type is created

```
create type locationUdt as ( ... ) not final
  overriding constructor method locationUdt (street varchar(30),
     building varchar(5), room varchar(5)) returns locationUdt;
create method locationUdt(st varchar(30), bl varchar(5), rm varchar(5))
  returns locationUdt for locationUdt
begin
  set self.street = st;  set self.building = bl;  set self.room = rm;
  return self;
end;
```

- Since the overridden method is a constructor function, the `constructor` keyword must be specified
- The name of the method must be the same as the name of the system-defined constructor function
- This function can then be used to construct a new instance and to set the values of its attributes

```
declare loc locationUdt;
set loc = new locationUdt('Mill Avenue', 'Brickyard Building', 'Rm 222');
```


Typed Tables (1)

- To create the notion of an object, a UDT must be used together with a typed table
- **Typed table**: a new form of table that is always associated with a structured type
- It has a column for every attribute of the structured type on which it is based
- In addition, it has a **self-referencing column** that contains a unique object identifier, known as a reference, for each row in the table
- Other than the self-referencing column and the attributes of the structured type, additional columns cannot be added to the table definition
- When a structured type is used to define a typed table, an instance of the type is viewed as an object, with the self-referencing column providing the object identifier
- Unlike object identifiers in OODBs, an object identifier is only **unique within a specific typed table**
- As a result, it is possible to have two typed tables with rows that have identical self-referencing values

Typed Tables: Defining the Structured Type

- Defining the UDT associated to the `department` table

```
create type departmentUdt as ( code varchar(3), name varchar(40) )
  instantiable not final ref is system generated;
```

- The `instantiable` clause indicates that it is possible to directly create instances of the type
- The use of `not instantiable` only makes sense in the context of a type that has a subtype
- A structured type that is used together with a typed table must always be specified as instantiable
- There are three ways for generating the value for the object reference
- **System generated**: the DBMS generates a unique object reference for instances of the type
- **User defined**: the user must provide a unique reference for each instance using a predefined type
- **Derived**: the user must specify the attribute(s) from the type used to provide a unique object reference
- For user-defined and derived references, the user must ensure the uniqueness of the reference
- System generated references is the most natural approach

Defining a Typed Table

- Once a structured type is defined, a corresponding typed table may be defined

```
create table department of departmentUdt
( primary key (code), ref is departmentID system generated);
```

- Typed tables supports the same constraints (e.g., primary key, not null) as those of regular tables
- Only the attributes with constraints are listed in the typed table definition
- A reference generation specification that is consistent with that of the structured type must be repeated
- The `ref is` specification must assign a **name** to the self referencing column to manipulate it
- Rows are inserted in the same manner as for any relational table

```
insert into department values ('cse', 'Computer Science and Engineering');
insert into department values ('ece', 'Electrical and Computer Engineering');
insert into department values ('mae', 'Mechanical and Aerospace Engineering');
```

- If the reference is user-defined, the value for the self-referencing column must also be given
- If the reference is derived, the primary key or the unique and not null constraints can be used in the table definition to ensure a unique reference value for the appropriate attributes

Type and Table Hierarchies (1)

- Structured types and typed tables can be formed into superclass/subclass hierarchies
- Structured types are formed into a hierarchy using the `under` clause in the `create type` statement
- This supports the inheritance of attributes and methods of the supertype to the subtypes
- Typed tables are then created to correspond to the type hierarchy, also using an `under` clause
- Inheritance is also supported among the object instances of typed tables
- Structured type definitions for the `Person`, `Student`, and `Faculty` classes

```
create type personUdt as
( personId varchar(11), firstName varchar(20), lastName varchar(20),
  birthDate date) instantiable not final ref is system generated;
create type facultyUdt under personUdt as
( rank varchar(10)) instantiable not final;
create type studentUdt under personUdt as
( status varchar(10)) instantiable not final;
```

- Recall that structured types must always be defined as `not final`
- The `instantiable` clause is required since these types are used with typed tables

Type and Table Hierarchies (2)

- With the type hierarchy defined, the corresponding typed tables can be created
- Definition of the `person`, `faculty`, and `student` typed tables

```
create table person of personUdt
( primary key (personId), ref is personID system generated);
create table faculty of facultyUdt under person;
create table student of studentUdt under person;
```

- The `under` specification of the subtables must be consistent with that of the corresponding types
- Only **single inheritance** is supported ⇒ every subtype/subtable has one maximal supertype/supertable
- A primary key can only be defined for a maximal supertable, it is inherited by all of the subtables
- Subtables can indirectly define keys through the use of the not null and unique constraints
- A self-referencing column can only be defined at the supertype/supertable level
- Constraints can only be defined on originally-defined attributes and not on inherited attributes
- Originally-defined attributes: attributes introduced in the structured type on which the table is based

Type and Table Hierarchies (3)

- Type hierarchies can be used independently of table hierarchies
- In this case, use of the type hierarchy supports valued-based rather than object-based inheritance
- The use of the `not instantiable` clause can only be used in type hierarchies that are not associated with typed tables
- For example, a user can define a supertype `A` as not instantiable and then define `B` and `C` as instantiable subtypes of `A`
- Since `A` is not instantiable, users cannot directly create instances of `A` but can directly create instances of `B` and `C` that inherit the attributes and methods of `A`
- All types associated with a typed table hierarchy must be defined as instantiable
- As a result, abstract supertables and the total specialization constraint cannot be inherently enforced

Inserting Rows in a Table Hierarchy

- Example: `insert` statements over the `Person` hierarchy

```
insert into person values ('PP111', 'Joe', 'Smith', '2/18/82');
insert into person values ('PP222', 'Alice', 'Black', '2/15/80');
insert into student values ('ST333', 'Sue', 'Jones', '8/23/87', 'freshman');
insert into student values ('ST444', 'Joe', 'White', '5/16/86', 'sophomore');
insert into faculty values ('FA555', 'Alice', 'Cooper', '9/2/51', 'professor');
```

- Each row has a **most specific table** that defines the type of the row
- This type corresponds to the type of the table in which the row is directly inserted
- Inserting a row in a table makes the row visible in all supertables of the table
- The row is not visible in any of the subtables of the table
- Once inserted, a row cannot migrate to other tables in the hierarchy
- This can only be achieved by deleting the row and reinserting it into a different table
- If the self-referencing column is system generated, the row will have a different reference value
- The contents of subtables are therefore always disjoint and cannot represent overlapping constraints

Querying a Table Hierarchy

- In subtable/supertable relationships, an instance of a subtable is an instance of its supertable
- The query below over the person table returns the five names previously introduced

```
select firstName || ' ' || lastName
from person;
```

- The query performs the union of the person table with all of the common attributes from rows of the subtables to return all direct instances of the person table as well as all instances of its subtables
- The following query over the faculty table returns only one name

```
select firstName || ' ' || lastName
from faculty;
```

- Unless otherwise specified, a query over a typed table returns the direct instances of the table as well as instances of its subtables and not instances of its supertables
- The only option can be used in the from clause to retrieve only the direct instances of a table
- The following query returns two names

```
select firstName || ' ' || lastName
from only (person);
```


Deleting Rows from a Table Hierarchy

- The following statement over the person table will delete any row from the table hierarchy with the first name of Alice, even if the most specific type of the row is a student or a faculty

```
delete from person where firstName = 'Alice';
```

- Deleting a tuple from a subtable also implicitly deletes the tuple from its supertables
- Once a row is removed from its most specific type, it is removed from the entire table hierarchy
- In the statement below the deletion of 'Sue Jones' from the student table will also result in 'Sue Jones' no longer being visible from the person table

```
delete from student where firstName = 'Sue' and lastName = 'Jones';
```

- It is possible to restrict the rows that are deleted using the only clause
- The following statement will delete rows from the person table only if the person does not have a most specific type of student or faculty

```
delete from only (person) where firstName = 'Joe';
```

- As a result, the row for 'Joe Smith' is the only row deleted, the row for 'Joe White' still remains in the table hierarchy

Updating Rows in a Table Hierarchy

- The update statement operates in a manner similar to the delete statement
- Example: change the first name of each row in the person table and in any of its subtables

```
update person
set firstName = 'Suzy'
where firstName = 'Sue';
```

- As a result, the name of 'Sue Jones' in the student table will be changed
- To restrict an update operation to rows that do not appear in any subtables, the only clause is used
- The query below will change the first name of Joe Smith in the person table and not the first name of Joe White in the student table

```
update only (person)
set firstName = 'Joey'
where firstName = 'Joe';
```

- The condition in the where clause of an update statement also applies to all rows that are in supertables of the table identified in the update statement
- The set clause is ignored if it refers to attributes that do not appear in the supertable

Reference Types

- The self-referencing column is the internal object identifier, or reference, to a row
- This reference is a data type known as a **reference type**, or ref
- Reference types can be used to model associations between typed tables that are based on object identity rather than on foreign keys
- Using reference types for defining relationships of a club with faculty and student objects

```
create type campusClubUdt as ( ...
  advisor ref(facultyUdt),
  members ref(studentUdt) array[50]) ... ;
create table campusClub of campusClubUdt
( primary key (clubId), ref is campusClubId system generated);
```

- The name of the self-referencing column of the faculty table can be used to assign a ref value

```
update campusClub
set advisor = ( select personID from person
    where firstName = 'Alice' and lastName = 'Cooper')
where name = 'Computer Club';
```

- Arrays of references (e.g., members) are generally manipulated through the use of SQL routines

Scopes and Reference Checking

- A `scope` clause can be used to restrict reference values to those from a specific typed table
- Redefinition of the `advisor` attribute in `campusClubUdt`

```
advisor ref(facultyUdt) scope faculty
  references are checked on delete set null;
```

- If the clause is omitted, the reference value can be a row from any table having the specified type
- If the clause is specified, a reference scope check can also be specified, i.e., dangling references (invalid reference values) are not allowed
- By default, references are not checked
- The `on delete` clause allows the user to specify the same referential actions as used with referential integrity for foreign keys
- In the example above, the deletion of a referenced faculty object will cause the `advisor` attribute to be set to null
- Specification of table scopes and reference checks can also be done during the creation of typed tables

Querying Reference Types

- Reference values are typically used to accessing some attribute of the row that is being referenced
- The **dereference operator** `->` is used to traverse through object references
- Give the name of the advisor of the Computer Club

```
select advisor -> firstName, advisor -> lastName
from campusClub
where name = 'Computer Club';
```

- The dereference operator is used to implicitly join the `campusClub` and `faculty` tables based on the reference value
- The `deref()` function returns the entire structured type that is associated with a reference value
- Give the advisors of campus clubs that are located in the Brickyard Building

```
select deref (advisor)
from campusClub c
where c.location.building = 'Brickyard Building';
```

- The result of the query is a table with one column of type `facultyUdt`

School Database Example: SQL Object-Relational Schema (1)

```
create type personUdt as
( personId varchar(11),
  firstName varchar(20),
  lastName varchar(20),
  birthDate date)
  instantiable not final ref is system generated;
create type facultyUdt under personUdt as
( rank varchar(10),
  advisorOf ref(campusClubUdt) scope campusClub array[5]
    references are checked on delete set null,
  worksIn ref(departmentUdt) scope department
    references are checked on delete no action,
  chairOf ref(departmentUdt) scope department
    references are checked on delete set null)
  instantiable not final;
create type studentUdt under personUdt as
( status varchar(10),
  memberOf ref(campusClubUdt) scope campusClub array[5]
    references are checked on delete set null,
  major ref(departmentUdt) scope department
    references are checked on delete no action)
  instantiable not final;
```


School Database Example: SQL Object-Relational Schema (2)

```
create table person of personUdt
( primary key(personId),
  ref is personID system generated);
create table faculty of facultyUdt under person;
create table student of studentUdt under person;
create type departmentUdt as
( code varchar(3),
  name varchar(40),
  deptChair ref(facultyUdt) scope faculty
    references are checked on delete no action)
  instantiable not final ref is system generated
  method getStudents() returns studentUdt array[1000],
  method getFaculty() returns facultyUdt array[50];
create table department of departmentUdt
( primary key (code),
  deptChair with options not null,
  ref is departmentID system generated);
```

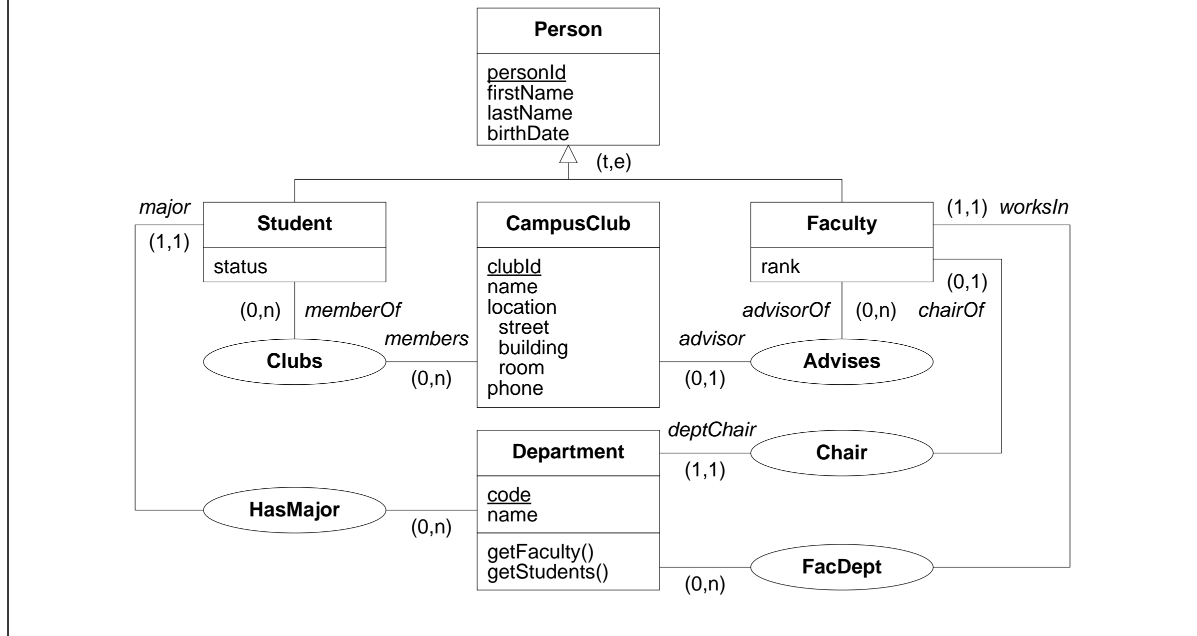
School Database Example: SQL Object-Relational Schema (3)

```
create type locationUdt as
( street varchar(30),
  building varchar(5),
  room varchar(5)) not final;
create type campusClubUdt as
( clubId varchar(10),
  name varchar(50),
  location locationUdt,
  phone varchar(12),
  advisor ref(facultyUdt) scope faculty
    references are checked on delete cascade,
  members ref(studentUdt) scope student array[50]
    references are checked on delete set null)
  instantiable not final ref is system generated;
create table campusClub of campusClubUdt
( primary key(clubIdd),
  ref is campusClubID system generated);
```


Object Databases: Topics

- Object-Oriented Databases
  - ODMG
  - Linq
- Object-Relational Databases
  - SQL
  - ➡ **Oracle**
- Summary

School Database Example: Entity Relationship Schema



Object Types

- **Object type**: a composite structure that can have a set of attributes and methods

```
create or replace type person_t as object
( personId varchar2(9),
  firstName varchar2(20),
  lastName varchar2(20),
  birthDate date);
```

- Oracle definition of an object type does not require the instantiation, finality, and reference generation clauses as in the SQL standard

- To assign values to an object type, a variable can be created having the type of the object type

```
declare
  p person_t;
begin
  p.personId := 'PR123456789';
  p.firstName := 'Terrence';
  p.lastName := 'Grand';
  p.birthDate := '10-NOV-1975';
end;
```

Embedded Objects

- Oracle does not support row types from the SQL standard
- When an object type is used as the attribute value of another object type, the attribute value is referred to as an **embedded object**
- Embedded objects do not have object identifiers, but provide a way to define an attribute as containing a composite value
- Example: the `location` attribute of the `campusClub_t` object type is used as an embedded object

```
create or replace type location_t as object
( street varchar2(30),
  building varchar2(5),
  room varchar2(5));
create or replace type campusClub_t as object
( clubId number,
  name varchar2(50),
  phone varchar2(25),
  location location_t,
  ... );
```


Column Objects

- When an object type is used as the type of a column in a traditional relational table, the value is referred to as a **column object**
- Embedded and column objects are accessed and manipulated in the same way as structured types
- Example: A table definition with a column object `loc`

```
create table courseOffering(code int, loc location_t);
```

- Data can be inserted into the table using the type constructor for `location_t`

```
insert into courseOffering
    values (123, location_t('Orange Mall', 'CPCOM', '207'));
```

Methods

- The methods of an object type are functions or procedures that model the behavior of objects
- Methods in the SQL standard are **limited to functions only**
- The `self` parameter is an instance of the object type on which a method is invoked
- Example: Definition of a function that determines whether a faculty member is an advisor of a club

```
create or replace type body campusClub_t is
  ...
  member function isAssociatedAdvisor (candidate_advisor in ref faculty_t)
    return boolean is
  begin
    return (self.advisor = candidate_advisor);
  end isAssociatedAdvisor;
end;
```

- Methods are implemented in the object type body definition for `campusClub_t`

Method Pragmas

- The declaration of functions are accompanied by a pragma declaration

```
create or replace type campusClub_t as object ( ...
  member function isAssociatedAdvisor
    (candidate_advisor in ref faculty_t) return boolean,
  pragma restrict_references(default, wnds, wnps));
```

- Pragmas are compiler directives, they do not affect the meaning of a program
- Pragma `restrict_references` is used to control side effects in a function, asserting that a function does not read/write any database tables and/or package variables
- Syntax: `pragma restrict_references (function_name, option [, option])`
- `default` can be used to specify that the pragma applies to all functions in the type definition
- The pragma options are the following
  - `wnds`: Writes no database state (does not modify database tables)
  - `rnds`: Reads no database state (does not query database tables)
  - `wnps`: Writes no package state (does not change the values of package variables)
  - `rnps`: Reads no package state (does not reference the values of package variables)

Constructor Methods

- Every object type has a system-defined **constructor method**
- It is a method that creates a new instance of an object type and sets up the values of its attributes
- The name of the constructor method is the name of the object type
- Its parameters have the names and types of the object type's attributes
- It is a function that returns an instance of an object type as its value
- Example: invocation of a constructor method that returns an instance of the `person_t` object type

  ```
  person_t('PR123456789', 'Terrence', 'Grand', '10-NOV-1975')
  ```
- Unlike the definition of constructor methods in the SQL standard, the built-in constructor for an object type in Oracle accepts parameter values for the attributes of the type

Type Inheritance

- A **type hierarchy** is specified by creating an object type as a supertype and by defining one or more subtypes of the supertype
- Example: `student_t` defined as subtype under `person_t`

  ```
  create or replace type student_t under person_t
  ( status varchar2(10),
    major ref department_t, --relation major
    memberOf campusClub_array, --relation memberOf
    member function getClubs return string_array,
    pragma restrict_references (default, wnds, wnps)) final;
  ```
- Objects of type `student_t` inherit the attributes of `person_t` and also define additional attributes and methods
- As a supertype, `person_t` must be defined as not final so that subtypes can be defined
- Unlike the SQL standard, a type at the bottom of the type hierarchy can be specified as final

Object Tables

- Tables in which each row represents an object, each object has a unique **object identifier** (OID)
- Objects that occupy complete rows in object tables are called **row objects**
- Row objects can be referenced in other row objects or relational rows using its OID
- The unique OID value specified as either **system-generated** (default) or **based on the primary key**
- Example: creation of an object table `person` based on the object type `person_t`

```
create table person of person_t
( personId primary key,
  firstName not null,
  lastName not null,
  birthDate not null)
object id system generated;
```

- The `ref is` clause of the SQL standard is replaced by the `object id` clause
- Table and column constraints can be defined in the specification of the object table
- As a deviation from the SQL standard, object tables in Oracle cannot be formed into table hierarchies
- Instead, Oracle supports the concept of substitutable tables

Substitutable Tables

- **Substitable tables** are capable of storing multiple object types from a type hierarchy
- They have the polymorphic capability to store an object of a supertype and any subtype that can be substituted for the supertype
- The `person` table above is a substitutable table: it is associated with the supertype of a type hierarchy
- It is capable of storing objects of type `person_t` as well as objects of its subtypes
- Inserting row objects in substitutable tables

```
insert into person_table values (person_t('101','Sue','Jones','05/16/1955'));
insert into person_table values (student_t('102','Jim','Duncan','03/12/1989',
    'senior',get _dref('CSE'),null);
insert into person_table values (faculty_t('103','Joe', 'Smith','10/21/1960',
    'Professor',null,get_dref('CSE'),null);
```

- Each `insert` statement indicates the specific type of the object and includes values for attributes of `person_t` as well as values for the attributes of the subtype
- The `insert` statements invoke the `get_dref` function, which is passed the name of a department and returns a reference to the associated department object (see later)

Reference Types (1)

- Reference types (refs) can be used to define object-to-object relationships
- A ref is a logical pointer to a row object
- Example: a ref type is used to model the chair relationship between department_t and faculty_t

```
create or replace type department_t as object ( ...
  deptChair ref faculty_t, ...);
```

- A ref column or attribute can be constrained using a scope clause or a referential constraint clause
- When a ref column is unconstrained, it may store object references to row objects contained in any object table of the corresponding object type
- Unconstrained references may also lead to dangling references
- Currently, Oracle does not permit storing object references that contain a primary-key based object identifier in unconstrained ref columns

Reference Types (2)

- A ref column may be constrained with a referential constraint similar to foreign keys
- Example: Specifying a column referential constraint on the deptChair column

```
create table department of department_t ( ...
  foreign key(deptChair) references person on delete set null ... )
```

- A ref stored in a deptChair column must point to a valid row object in the person object table
- This also implicitly restricts the scope of deptChair to the person object table
- A scope clause can be used for constraining a ref column by being scoped to a specific object table
- The scope constraint ≠ referential constraint: the scope constraint has no implications on the referenced object (i.e., deleting a referenced object can still cause a dangling reference)
- Alternatively, the scope clause can be used to constrain the deptChair column to refer to objects in the person table

```
alter table department add (scope for (deptChair) is person);
```

- The unique and primary key constraints cannot be specified for ref columns
- A unique index may be created on a scoped ref column to ensure uniqueness of the ref values

Using Reference Types (1)

- Queries involving objects make a distinction between row objects, refs, and object types
- Oracle provides three functions to support queries involving objects
  - `ref()`: takes a row object as its argument and returns the ref to that object
  - `value()`: takes a row object as its argument and returns the instance of the object type
  - `deref()`: takes a `ref` to an object as its argument and returns the instance of the object type
- The only difference between `value` and `deref` is the input to each function
- In both cases, the output is the instance of the object type (i.e., a tuple of attribute values) associated with the object

Using Reference Types (2)

- Example code

```
declare
    club_ref ref campusClub_t;
    club campusClub_t;
    club_adv faculty_t;
begin
    select value(c), ref(c), deref(c.advisor) into club, club_ref, club_adv
    from campusClub c
    where c.name='The Hiking Club';
end;
```

- The table alias `c` contains a row object
- `value(c)` used to see the values of the attributes of the row object as defined in the object type
- `ref(c)` used to get the `ref` to the row object
- `deref()` used to get values of the attributes of the object type associated with a `ref`
- In the above query, `club` will contain a `campusClub_t` object type instance, `club_ref` will contain the `ref` of the object, and `club_adv` will contain an instance of the `faculty_t` object type
- Since `c.advisor` is a `ref` to `faculty_t`, the `deref` function is applied to return the object type

Using Reference Types (3)

- The `get_dref` function was used previously to retrieve the `ref` to a specific department object

```
create or replace function get_dref(d_code in varchar2)
    return ref department_t is
    d_ref ref department_t;
    cursor cr is
    select ref(d) from department d where d.code = d_code;
begin
    open cr; fetch cr into d_ref; close cr; return d_ref;
end get_dref;
```

- The `select` statement retrieves a department object using the `code` attribute and returns the reference to the object for assignment to an attribute that is defined to be of type `ref department_t`
- The query below returns the name of the department in which the advisor of the Computer Club works

```
select c.advisor.worksIn.name
from campusClub c
where c.name = 'Computer Club';
```

- The expression `c.advisor.worksIn.name` is a path expression, representing implicit joins between the object tables involved

Querying Substitutable Tables (1)

- A substitutable table contains a heterogeneous collection of objects from a type hierarchy
- Queries over the table may need to indicate the specific types of objects to be returned
- The following select statement will return information about all objects in the `person` table

```
select value(p)
from person_table p;
```

- The `is of` clause used to retrieve details only about objects that are subtypes of the type of the table
- The query below will only return objects of type `student_t` from the `person` table

```
select value(p)
from person_table p
where value(p) is of (student_t);
```

Querying Substitutable Tables (2)

- The `treat` function can be used to treat a supertype instance as a subtype instance
- This is useful for assigning a variable to a more specialized type in a hierarchy and accessing attributes or methods of a subtype
- The following query creates a view of student objects

```
create or replace view student(personId, firstName, lastName, birthDate,
        status, major, memberOf) as
    select personId, firstName, lastName, birthDate,
            treat(value(p) as student_t).status,
        treat(value(p) as student_t).major,
                treat(value(p) as student_t).memberOf
    from person p
    where value(p) is of (student_t);
```

- The `is of` clause is used to indicate that the view should include objects of type `student_t`
- The `treat` function is used to access the attributes that are specific to the `student_t` object type

Varrays and Nested Tables as Collections (1)

- Variable-sized arrays (`varrays`) and nested tables can be used to represent the many side of 1:N and M:N relationships
- Whereas a `varray` is an **indexed collection** of data elements of the same type, a nested table is an **unordered set** of data elements of the same data type
- Maximum size must be specified when an attribute of type `varray` is defined, it can be changed later
- To define an attribute as a `varray`, a `varray` type definition must first be created
- Creating a `varray` type does not allocate space, simply defines a data type that can be used elsewhere
- Example: `varray` type definition used to model the many side of the `Clubs` relationship

```
create or replace type campusClub_array as varray(50) of ref campusClub_t;
create or replace type student_t under person_t ( ...
  memberOf campusClub_array, ...)
```

- Example: `varray` definitions for `student_array`, used as output in the `getStudents` function

```
create or replace type student_array as varray(50) of ref student_t;
create or replace type department_t as object ( ...
  member function getStudents return student_array, ...)
```

Varrays and Nested Tables as Collections (2)

- A nested table has a single column, where the type of the column is a built-in type or an object type
- If the column in a nested table is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type
- DML statements on nested tables such as `select`, `insert`, and `delete` are the same as with regular relational tables
- In a nested table, the order of the elements is not defined
- Nested tables are stored in a storage table with every element mapping to a row in the storage table
- Example: a nested table type definition for the `student_ntable` nested table

```
create or replace type student_ntable as table of ref student_t;
create or replace type campusClub_t as object ( ...
  members student_ntable, ... );
```

- The contents of the nested table are of type `ref student_t`
- The type definition is used to define the `members` attribute in `campusClub_t`

Varrays and Nested Tables as Collections (3)

- Example: Specifying the storage table of the `club_members` nested table

```
create table campusClub of campusClub_t ( ... )
  object id system generated
  nested table members store as club_members;
alter table club_members add(scope of (column_value) is person);
create unique index club_members_idx on
  club_members(nested_table_id, column_value);
```

- The programmer accesses the nested table using the name `members`
- The name `club_members` is used internally by Oracle to access and manipulate the nested table
- The `scope` constraint indicates that the values stored in the `ref` column must be references to row objects in the `person` table
- Unique index restrict the uniqueness of the `ref` values
- `nested_table_id`: pseudocolumn that holds the row identifier of the storage table
- `column_value`: pseudocolumn that holds the contents of the nested table, in this case, a `ref` of type `student_t`

Comparison of Varrays and Nested Tables (1)

Varrays

- A varray cannot be indexed
- A varray declaration must specify the maximum number of objects to hold
- A varray is dense in that all positions from the first to the lastmust be filled. Individual elements cannot be deleted from a varray to leave a null value in an array position
- The elements of a varray are ordered
- If the size of the varray is smaller than 4000 bytes, Oracle stores the varray in line; if it is greater than 4000 bytes, Oracle stores it in a Binary Large Object (BLOB)

Comparison of Varrays and Nested Tables (2)

Nested Tables

- A nested table is an unordered set of data elements, all of the same data type having a single column
- The type of the column in the nested table is either a built-in type or an object type
- If an object type is used as the type of the column, the table can be viewed as a multi-column table, with a column for each attribute of the object
- Nested tables are stored in a separate storage table and can be indexed for efficient access
- Nested tables are sparse (i.e., individual elements can be deleted from a nested table)

In Summary

- The choice depends on the nature of the values or objects stored in the collection
- Nested table used when the number of objects in a multivalued attribute or relationship is large
- Varray used when the number of objects contained in a multivalued attribute is small and does not change
- Consult Oracle 11g documentation for further details about accessing and modifying varrays and nested tables

School Database Example: Oracle Object-Relational Schema (1)

```
create or replace type campusClub_t;
create or replace type campusClub_array as varray(50) of ref campusClub_t;
create or replace type string_array as varray(50) of varchar2(50);
create or replace type location_t as object
( street varchar2(30),
  building varchar2(5),
  room varchar2(5));
create or replace type person_t as object
( personld varchar2(9),
  firstName varchar2(20),
  lastName varchar2(20),
  birthDate date) not final;
create or replace type department_t;
create or replace type student_t under person_t
( status varchar2(10),
  major ref department_t, --relation major
  memberOf campusClub_array, --relation memberOf
  member function getClubs return string_array,
  pragma restrict_references (default, wnds, wnps)) final;
```


School Database Example: Oracle Object-Relational Schema (2)

```
create or replace type faculty_t under person_t
( rank varchar2(25),
  advisorOf campusClub_array, --relation advisorOf
  worksIn ref department_t, --relation worksIn
  chairOf ref department_t,
  member function getClubsAdvised return string_array,
  pragma restrict_references(default, wnds, wnps))
  final;
create table person of person_t
( personId primary key,
  firstName not null,
  lastName not null,
  birthDate not null)
object id system generated;
```

School Database Example: Oracle Object-Relational Schema (3)

```
create or replace type student_array as varray(50) of ref student_t;
create or replace type faculty_array as varray(50) of ref faculty_t;
create or replace type department_t as object
( code varchar2(3),
  name varchar2(40),
  deptChair ref faculty_t,
  member function getStudents return student_array,
  member function getFaculty return faculty_array,
  pragma restrict_references(default, wnds, wnps));
create table department of department_t
( code primary key,
  name not null,
  constraint department_chair
  foreign key(deptChair) references person on delete set null)
  object id system generated;
create or replace type student_ntable as table of ref student_t;
```


School Database Example: Oracle Object-Relational Schema (4)

```
create or replace type campusClub_t as object
( clubId number,
  name varchar2(50),
  phone varchar2(25),
  location location_t,
  advisor ref faculty_t, --relation advised by
  members student_ntable, --relation member of
  constraint club_advisor
    foreign key (advisor) references person on delete set null,
  member function isAssociatedMember
    (candidate_member in ref student_t) return boolean,
  member function isAssociatedAdvisor
    (candidate_advisor in ref faculty_t) return boolean,
  pragma restrict_references(default, wnds, wnps));
create table campusClub of campusClub_t
( clubId primary key,
  name not null)
  object id system generated
  nested table members store as club_members;
alter table club_members add(scope of (column_value) is person);
create unique index club_members_idx on
  club_members(nested_table_id, column_value);
```

Object-Relational Features: Summary

- Extension of the standard relational model with object features
- This includes row types for representing composite values, arrays for representing collections, and user-defined types
- User-defined types can be distinct types for defining domains, and structured types for defining the equivalent of classes in OPL
- Structured types allow to define inheritance hierarchies
- Structured types together with typed tables allow to define object-based inheritance hierarchies
- Reference types allow to store the identifier (or reference) of object rows
- Reference types can be restricted in a similar way as for referential integrity
- Nevertheless, the resulting model is hybrid, it aims at converging two paradigms
- The implementations of the object-relational features varies across systems

Bibliographic References

- The Object-Oriented Database System Manifesto was prepared by researchers of OODB technology [Atkinson et al., 1990]
- In response, researchers in relational database technology prepared the Third Generation Database System Manifesto [Stonebraker et al., 1990] which outlined the manner in which relational technology could be extended to support object-oriented features
- The feasibility of object-relational technology was demonstrated with the development of Postgres [Rowe and Stonebraker, 1987], an object-relational version of Ingres (an early relational database prototype and research tool)
- Additional sources of information about object-relational technology can be found in [Brown, 2001, Date and Darwen, 1998, Stonebraker, 1995]
- Object-relational features of the SQL standard are described in [Melton, 2002] with additional information about arrays and row types in [Gulutzan and Pelzer, 1999, Melton and Simon, 2001]
- A description of the object-relational features of Oracle 11g can be found in Oracle's documentation

References

- M. Atkinson et al. The object-oriented database system manifesto. In *Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan. Elsevier, 1990
- P. Brown. *Object-Relational Database Development: a Plumber's Guide*, Prentice Hall, 2001.
- C.J. Date and H. Darwen. *Foundations for Object/Relational Databases: The Third Manifesto*, Addison-Wesley, 1998.
- P. Gulutzan and T. Pelzer. *SQL-99 Complete Really!*, Miller Freeman, 1999.
- J. Melton. *Advanced SQL:1999 Understanding Object-Relational and Other Advance Features*, Morgan Kaufmann, 2002.
- J. Melton and A. Simon. *SQL:1999 Understanding Relational Language Components*, Morgan Kaufmann, 2001.
- L. Rowe and M. Stonebraker. The Postgres Data Model. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 83-96, 1987.
- M. Stonebraker, *Object-Relational DBMSs: The Next Great Wave*, Morgan Kaufmann, 1995.
- M. Stonebraker et al. Third Generation Database System Manifesto. In *SIGMOD Record*, vol. 19, no. 3, pages 31-44, 1990.