Session-7: Object-Relational DBMS

Cyrus Shahabi

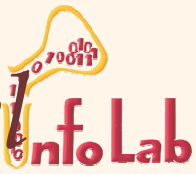
Motivation

- Relational databases (2nd generation) were designed for traditional banking-type applications with well-structured, homogenous data elements (vertical & horizontal homogeneity) and a minimal fixed set of limited operations (e.g., set & tuple-oriented operations).
- New applications (e.g., CAD, CAM, CASE, OA, and CAP), however, require concurrent modeling of both *data* and *processes* acting upon the data.
- Hence, a combination of database and software-engineering disciplines lead to the 3rd generation of database management systems: Object Database Management Systems, ODBMS.
- Note that a classic debate in database community is that do we need a new model or relational model is sufficient and can be extended to support new applications.

Motivation …

- People in favor of relational model argue that:
  - New versions of SQL (e.g., SQL-92 and SQL3) are designed to incorporate functionality required by new applications (UDT, UDF, …).
  - Embedded SQL can address almost all the requirements of the new applications.
- "Object people", however, counter-argue that in the above-mentioned solutions, it is the *application* rather than the inherent capabilities of the *model* that provides the required functionality.
- *Object people* say there is an *impedance mismatch* between programming languages (handling one row of data at a time) and SQL (multiple row handling) which makes conversions inefficient.
- *Relational people* say, instead of defining new models, let's introduce set-level functionality into programming languages.

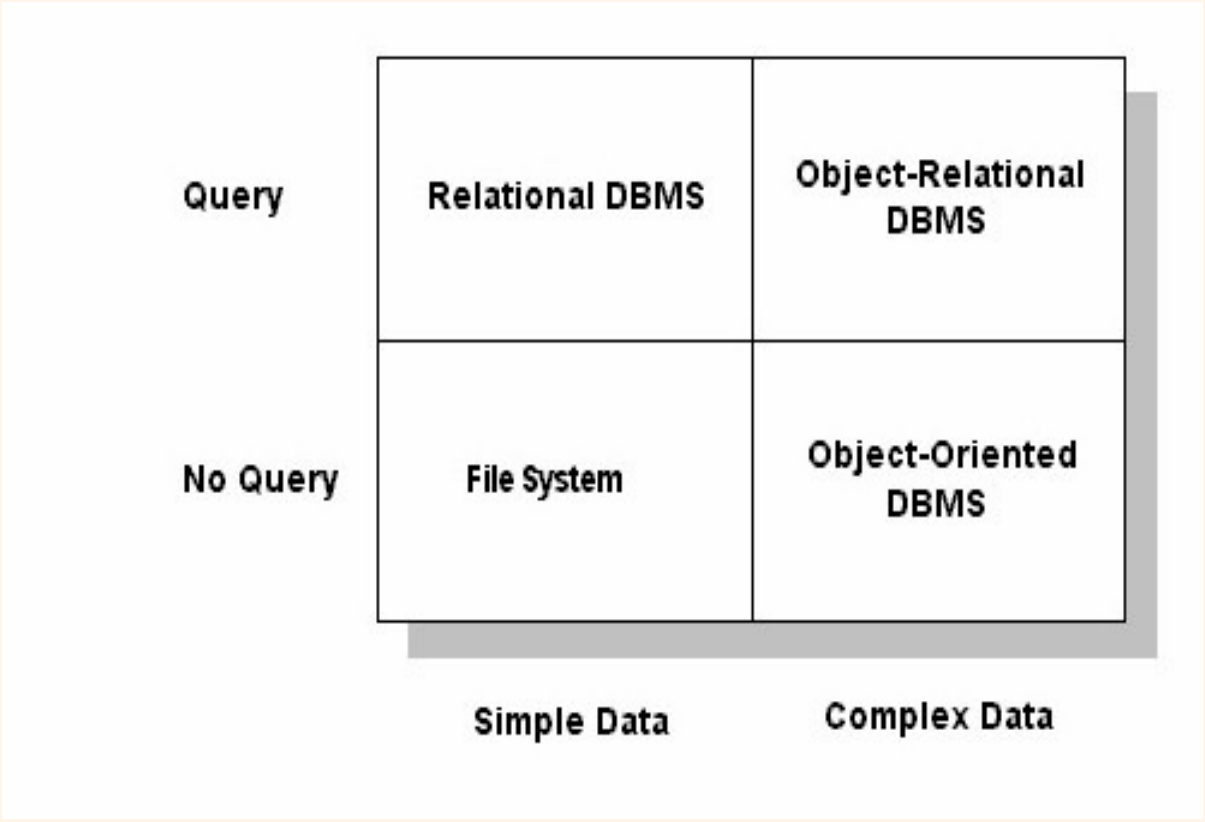
Weaknesses of Relational Data Model



- Poor representation of 'real world' conceptual model
  - Usually the relational schema does not correspond to real world entities
- Difficult to change schema without affecting the applications; e.g., Y2K
- Semantic overloading
  - The same relation is used to represent entities as well as relationships
- Poor support for integrity and business rules
- Fixed number of attributes & all attribute values must be atomic
- Limited operations
- Difficult to handle recursive queries
- Impedance mismatch (when SQL is embedded in PLs)
  - Type System mismatch, Evaluation Strategy mismatch
- Poor navigational access
- Short-lived transactions (strict locking and recovery mechanisms

Michael Stonebraker's Classification

- Michael Stonebraker presents this four-quadrant matrix in the book entitled *"Object-Relational DBMSs: The Next Great Wave"*
  - This is a classification of both database applications and systems.



Lower-Left Quadrant

- Those application that process simple data and require no query capability e.g. text processors (word, emacs)
  - Information has little internal structure.
  - Document updates are relatively infrequent.
  - Documents are of modest size.
  - Queries are simple string or pattern searches.

Upper-Left Quadrant

- Those application that process simple data and require complex query capability e.g. a typical business application require RDBMS.
  - Information has straightforward and fixed structure.
  - Information collection may be large.
  - Information storage must be reliable.
  - Queries are relatively complex.
  - Updates are frequent and Security is vital.

Lower-Right Quadrant

- Those application that process complex data and require no query capability e.g. a CAD application requires OODBMS.
  - Information has complex structure.
  - Analysis are complex.
  - Information is moderate in quantity.
  - Updates are periodic.

Upper-Right Quadrant

- Those application that process complex data and require complex query capability e.g. an Image Data Archive requires ORDBMS.
  - Information has complex structure.
  - Information may include special data types.
    - Images, Spatial information
  - Information is large in quantity.
  - Queries are important.
  - Updates are periodic.

Object-Relational Databases

- Object-Relational databases (ORDBSs) seek to:
  - Retain the relational model as a subset.
  - Retain the strengths of the relational model and all the technologies that come with it.
  - Supports complex data types (BLOBS, ADTs, Spatial, and Multimedia, …).
  - Supports object-oriented design.
  - Reduces impedance mismatch (type system).

Advantages of ORDBMSs

- Resolves many of known weaknesses of RDBMS.

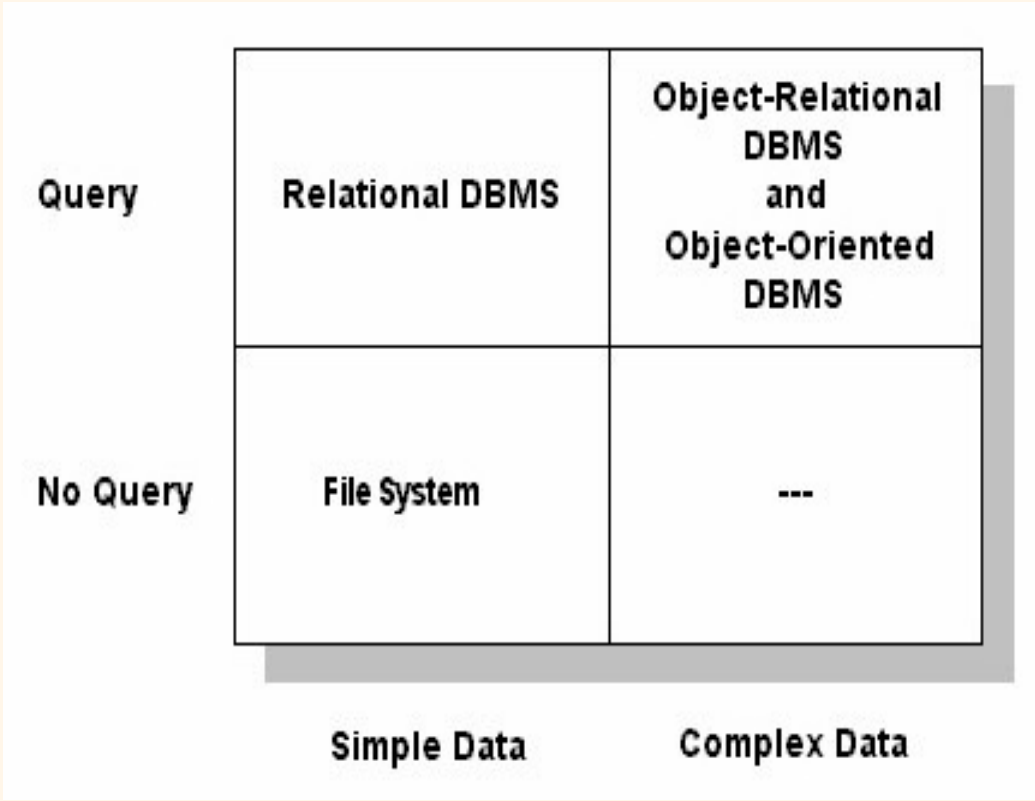
- Preserves significant body of knowledge and experience gone into developing relational applications.

Disadvantages of ORDBMSs

- Complexity.
- Increased costs.
- Supporters of relational approach believe simplicity and purity of relational model are lost.
- Some believe RDBMS is being extended for what will be a minority of applications.
- OO purists not attracted by extensions either.
- SQL now extremely complex.

Classification Problems

- Most of OODBMSs claim to be in Upper-Right quadrant not just ORDBMSs.

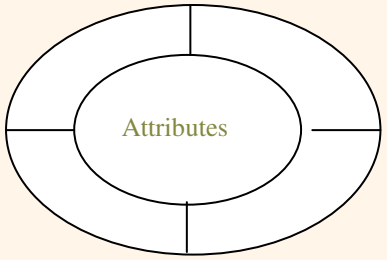


Object-Oriented Concepts

- **Abstraction and Encapsulation** (Provided by Abstract Data Types (ADT))
  - ***Abstraction*** is the process of identifying the essential aspects of an entity and ignoring the unimportant properties. Focus on what an object is and what it does, rather than how it should be implemented.
  - ***Encapsulation*** (or information hiding) provides data independence by separating the external aspects of an object from its internal details, which is hidden from the outside world.
- **Objects**
  - ***Object*** is a uniquely identifiable entity that contains both the attributes that describe the state of a real-world object and the actions that conceptualize the behavior of a real-world object. The difference between object and entity is that object encapsulates both state and behavior while entity only models state.
  - ***Attributes*** (or instance variables) describe the current state of an object (the notation for attribute: object-name.attribute-name).

Object-Oriented Concepts

- ***Methods:*** define the behavior of the object. They can be used to change the object's state by modifying its attribute values, or to query the value of the selected attributes. A method consists of a name and a body that performs the behavior associated with the method name (notation: object-name.method-name).



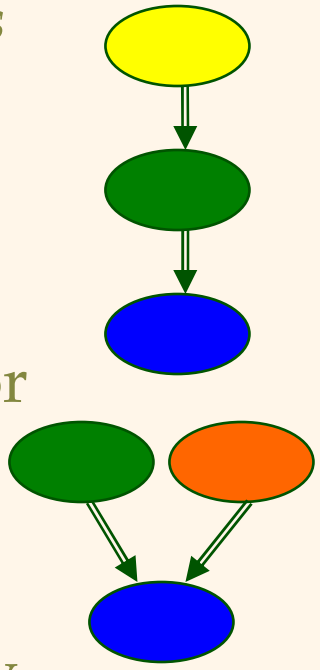
- **Classes:** A group of objects with the same attributes and methods. Hence, the attributes and the associated methods are defined once for the class rather than separately for each object.
- The ***instances*** of a class are those objects belonging to a class.

OO Concepts - Inheritance

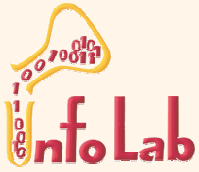
- ***Subclasses:*** A class of objects that is defined as a special case of a more general class (the process of forming subclasses is called *specialization*).
- ***Superclass:*** A class of objects that is defined as a general case of a number of special classes (the process of forming a superclass is called *generalization*). All instances of a subclass are also instances of its superclass.
- ***Inheritance:*** By default, a subclass inherits all the properties of its superclass (or it can redefine some (or all) of the inherited methods). Additionally, it may define its own unique properties.

OO Concepts - Inheritance

- ***Single inheritance:*** When a subclass inherits from no more than one superclass (note: forming *class hierarchies* is permissible here).
- ***Multiple inheritance:*** When a subclass inherits from more than one superclass (note: a mechanism is required to resolve conflicts when the Superclasses have the same attributes and/or methods). Due to its complexity, not all OO languages and database systems support this concept.
- ***Overriding****:* To redefine an inherited property by defining the same property differently at the subclass level.



Chapter 9: Object-Relational Databases



- Nested Relations
- Complex Types
- Inheritence

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Nested Relations

- Motivation:
  - Permit non-atomic domains (atomic ≡ indivisible)
  - Example of non-atomic domain: set of integers,or set of tuples
    - Composite attributes; multi-valued attributes
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values — relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form.

Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a set of authors,
  - Publisher, and
  - a set of keywords
- Non-1NF relation *books*

title	*author-set*	*publisher*	*keyword-set*
		(*name*, *branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Complex Types and SQL:1999

- Extensions to SQL to support complex types include:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard
  - Not fully implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports
  - We present some features that are not in SQL:1999
    - These are noted explicitly

Collection Types

- Set type (not in SQL:1999)

  create table *books* (
  .....
  keyword-set **setof**(**varchar**(20))
  .....
)

- Sets are an instance of collection types. Other instances include
  - Arrays (are supported in SQL:1999)
    - E.g. *author-array* **varchar**(20) **array**[10]
    - Can access elements of array in usual fashion:
      - *E.g. author-array*[1]
  - Multisets (not supported in SQL:1999)
    - I.e., unordered collections, where an element may occur multiple times
  - Nested relations are sets of tuples
    - SQL:1999 supports arrays of tuples

Structured and Collection Types

- Structured types can be declared and used in SQL

```
create type Publisher as
    (name          varchar(20),
     branch        varchar(20))
create type Book as
    (title         varchar(20),
     author-array  varchar(20) array [10],
     pub-date      date,
     publisher     Publisher,
     keyword-set   setof(varchar(20)))
```

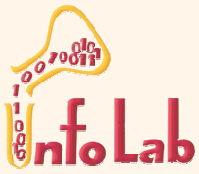
  - Note: **setof** declaration of keyword-set is not supported by SQL:1999
  - Using an array to store authors lets us record the order of the authors

- Structured types can be used to create tables

```
create table books of Book
```

  - Similar to the nested relation books, but with array of authors instead of set

Structured and Collection Types (Cont.)



- Structured types allow composite attributes of E-R diagrams to be represented directly.
- Unnamed row types can also be used in SQL:1999 to define composite attributes
  - **E.g.** we can omit the declaration of type *Publisher* and instead use the following in declaring the type *Book*

    publisher **row** (*name* **varchar**(20),
    branch **varchar**(20))
- Similarly, collection types allow multivalued attributes of E-R diagrams to be represented directly.

Structured Types (Cont.)

- We can create tables without creating an intermediate type
  - For example, the table *books* could also be defined as follows:

```
create table books
    (title varchar(20),
     author-array varchar(20) array[10],
     pub-date date,
     publisher Publisher
     keyword-list setof(varchar(20)))
```

- Methods can be part of the type definition of a structured type:

```
create type Employee as (
    name varchar(20),
    salary integer)
  method giveraise (percent integer)
```

- We create the method body separately

```
create method giveraise (percent integer) for Employee
begin
    set self.salary = self.salary + (self.salary * percent) / 100;
end
```

Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department   varchar(20))
create type Teacher
  under Person
  (salary       integer,
   department   varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Inheritance

- SQL:1999 does not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student-dept*),
  Teacher **with** (*department* **as** *teacher-dept*)

Table Inheritance

- Table inheritance allows an object to have multiple types by allowing an entity to exist in more than one table at once.
- *E.g. people* table: **create table** *people* **of** *Person*
- We can then define the *students* and *teachers* tables as **subtables** of *people*

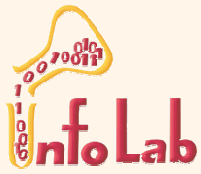
  create table *students* **of** *Student*
  under *people*
  create table *teachers* **of** *Teacher*
  under *people*

- Each tuple in a subtable (e.g. *students* and *teachers)* is implicitly present in its supertables (e.g. *people*)
- Multiple inheritance is possible with tables, just as it is possible with types.

  create table *teaching-assistants* **of** *Teaching Assistant*
  under *students, teachers*

  - Multiple inheritance not supported in SQL:1999

Table Inheritance: Consistency Requirements



- SQL:1999 Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
  - *That is, overlap participation is not supported (at most one, cannot be 2)*
  - *Partial participation is supported (at most one, can be 0)*