Basis Data Non Relasional

Object-Oriented Databases

Tricya Widagdo

Program Studi Teknik Informatika

Institut Teknologi Bandung

Motivation 1

- Consider a relational database in which we have information about Books:

ISBN	Title	Author	Keyword	Publisher
0-201	Compilers	J.D. Ullman	Compiler	Addison Wesley
0-201	Compilers	J.D. Ullman	Grammar	Addison Wesley
0-201	Compilers	A.V. Aho	Compiler	Addison Wesley
0-201	Compilers	A.V. Aho	Grammar	Addison Wesley
0-201	Compilers	J.E. Hopcroft	Compiler	Addison Wesley
0-201	Compilers	J.E. Hopcroft	Grammar	Addison Wesley

- Problem: Redundancy in design is forced by the fact that we cannot have set valued attributes.

"Normalized" Design

- Imagine another design:

ISBN	Title	Publisher
0-201	Compilers	Addison Wesley

ISBN	Author
0-201	J.D. Ullman
0-201	A.V. Aho
0-201	J.E. Hopcroft

ISBN	Keyword
0-201	Compiler
0-201	Grammar

Problems with Normalized Design

- The "object" book has been split over different relations
- Artificial keys are introduced when modeling sets of sets
- Impossible to represent order in relational data model
- While SQL can perform "group-by", it cannot return a grouped result. E.g. "Give me all the authors together with a list of books written by that author".
- SQL can be called from a "host" PL
  - How about calling external programs from SQL?

The natural solution

ISBN	Title	Authors	Keywords	Publisher
0-201	Compilers	{J.D. Ullman, A.V. Aho, J.E. Hopcroft}	{Compiler, Grammar}	Addison Wesley

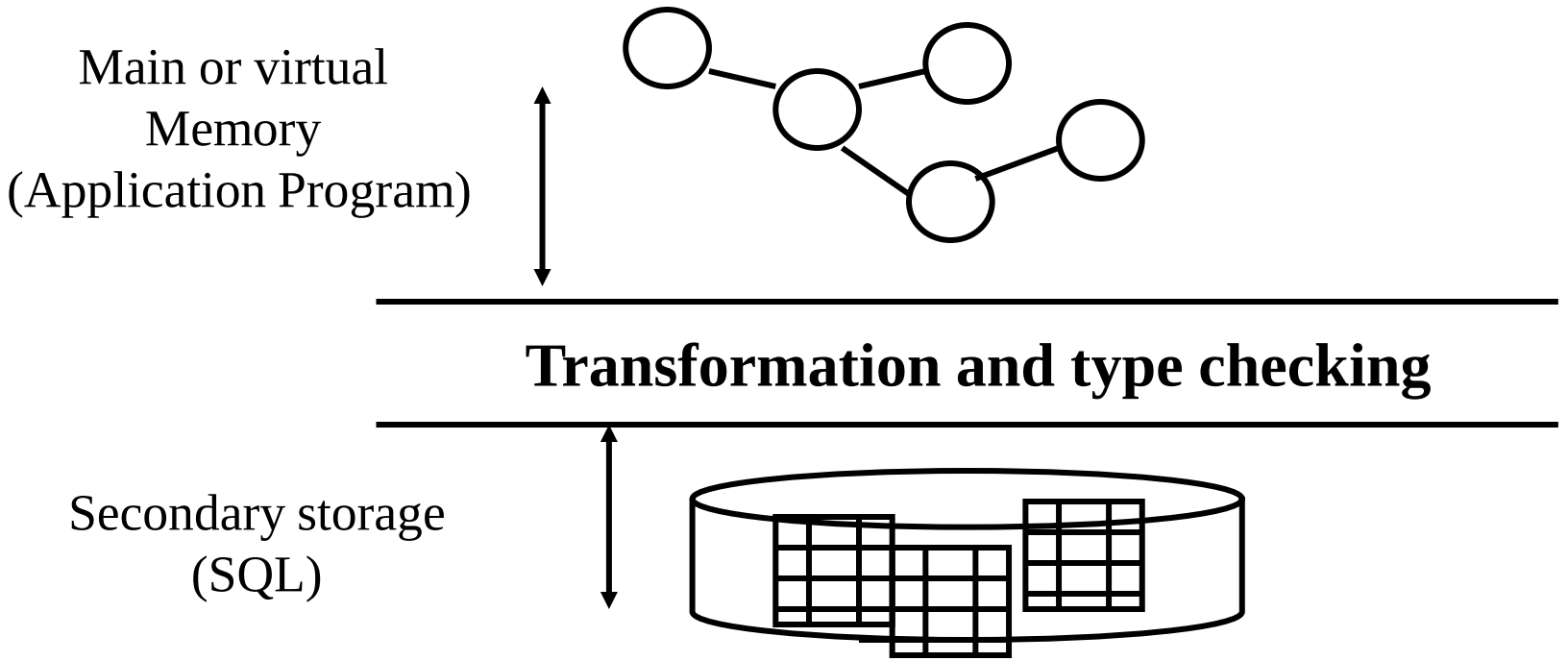
- This is an example of the nested relational model, in which values may themselves be sets or tuples.
- What needed is a type system that allows freely combining sets and tuples (non-1NF relations)
- This implies new operators: nest and unnest.

Motivation 2

- OO applications need object persistence
  - Persistence – ability to exist beyond the running time of an application. Implemented by storing the objects off-line on a secondary storage device
  - No strong support for persistence in OOPL's
- more and more application areas require systems that offer support (implement) for both traditional programming languages and database capabilities

One solution...

- Could use a relational database (RDBMS)
  - Store objects in the form of attribute tables



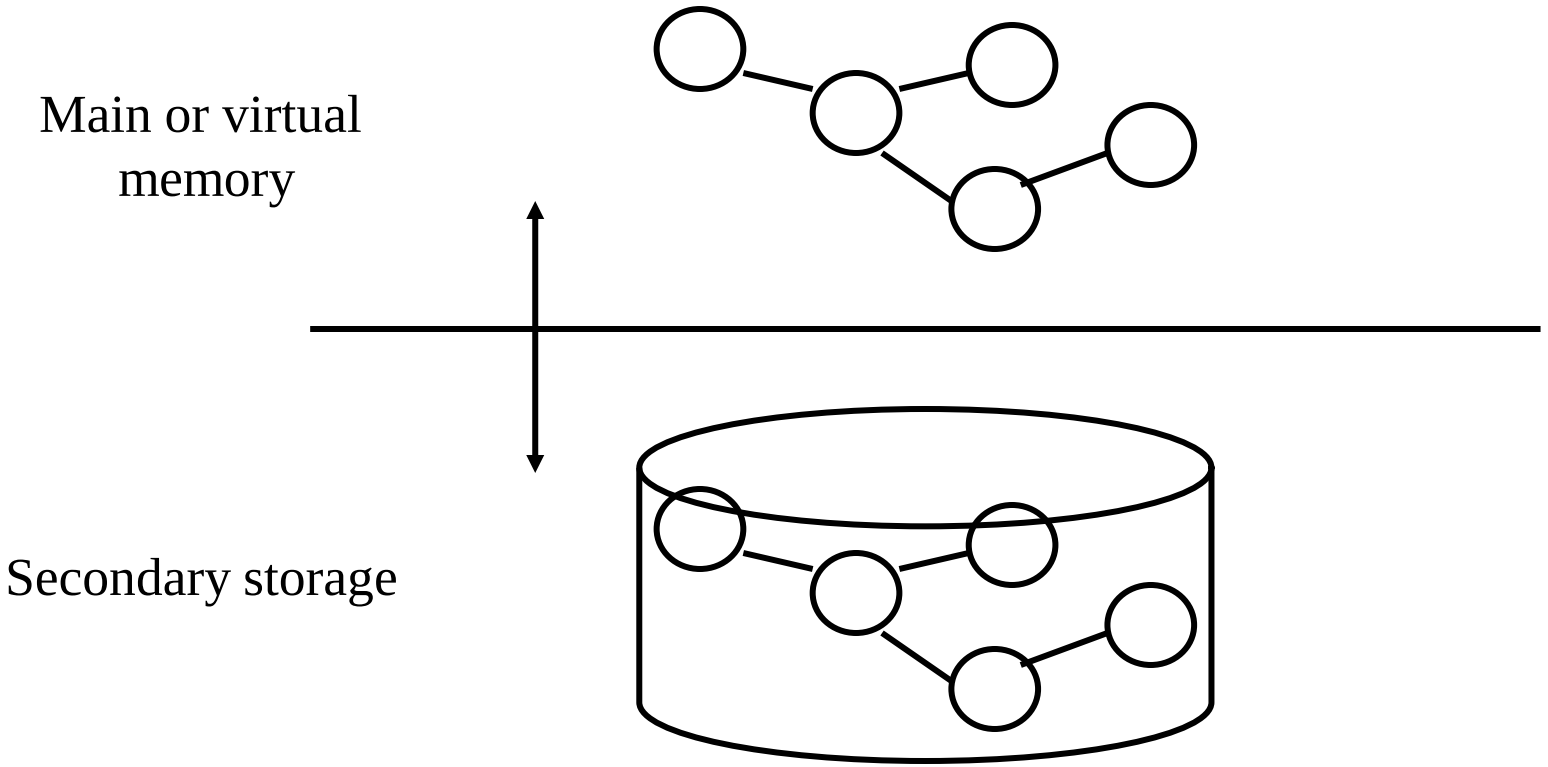
Problems with using relational model

- Impedance mismatch
  - the programmer has to write code to translate between the host language's data model and the DB's language data model (could take up to 30 % of coding effort)
  - Type checking
- Additional type-checking
- In relational model, typically many normalized tables are used to represent a real-world object.
  - Especially problematic for restructuring complex objects
  - It is always the programmer's responsibility to decide WHEN to read and write on secondary space

Benefits of relational model

- Well grounded in theory, thus
  - Portability: once written, the database application can be easily converted to another vendor's RDBMS
- Powerful query mechanism

Object-oriented DBMS



Intuitively clear idea, but needs a clever way of managing representations of objects in memory an on disk to achieve the illusion of transparency (*pointer swizzling*)

OODB Definitions

OO DBMS Definition

- Khoshafian and Abnous, 1990:
  OO DBMS = object orientation + database capabilities
- Kim 1991:
  - OODM (data model)
    - a logical data model that captures the semantics of objects supported in OO programming languages
    - NO UNIQUE DATA MODEL
  - OODB (database)
    - a **persistent** and **sharable** collection of objects defined by an OODM
  - OODBMS
    - the manager of an OODB

OO DBMS Definition 2

- Zdonic and Maier:
  - database functionality
  - support object identity
  - provide encapsulation
  - support objects with complex state
- Parsaye, 1989
  - high level query language + optimisation capabilities
  - support for persistent and atomic transactions
  - support for complex object storage + mechanisms of efficient access
  - OODBMS = OO system + the above characteristics

OO DBMS Definition 3

- Kroenke:

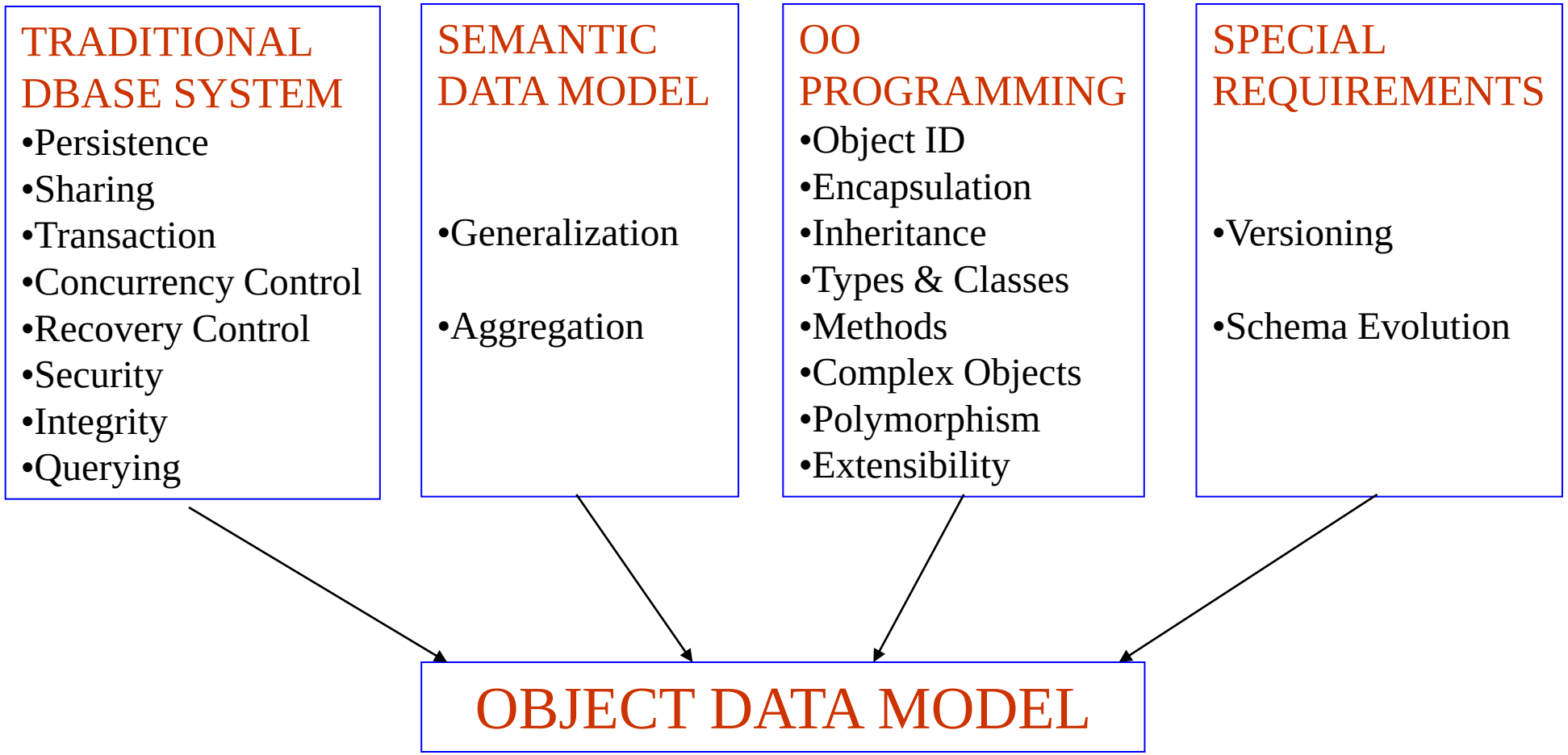
  provide the ability to persistently store objects created in Object Oriented programming languages like Java, C#, and C++

  - Objects have several complex structures that cannot be represented in tables, rows, or columns.
  - These structures include executable statements (i.e., methods) and pointers

Object-orientation

- Complex type system, including atomic types, collection types, and the record type
- Object identity gives identification of objects independent of value
- Values with identity are called objects
- Methods model the behavior of an object
  - For example, modifying the value of an object is often dangerous, and is encapsulated by methods
- Types + Identity + Methods are Classes
- Each of these new types gives rise to new operations on those types

Object Data Model



Object-Oriented Data Model

- Loosely speaking, an **object** corresponds to an entity in the E-R model
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit
- The object-oriented data model is a logical data model (like the E-R model)
- Adaptation of the object-oriented programming paradigm (e.g., Smalltalk, C++) to database systems

Object Structure

- An object has associated with it:
  - A set of **variables** that contain the data for the object. The value of each variable is itself an object.
  - A set of **messages** to which the object responds; each message may have zero, one, or more *parameters.*
  - A set of **methods***,* each of which is a body of code to implement a message; a method returns a value as the *response* to the message
- The physical representation of data is visible only to the implementor of the object
- Messages and responses provide the only external interface to an object.
- The term message does not necessarily imply physical message passing. Messages can be implemented as procedure invocations.

Messages and Methods

- Methods are programs written in general-purpose language with the following features
  - only variables in the object itself may be referenced directly
  - data in other objects are referenced only by sending *messages*.
- Methods can be read-only or update methods
  - Read-only methods do not change the value of the object
- Strictly speaking, every attribute of an entity must be represented by a variable and two methods, one to read and the other to update the attribute
  - e.g., the attribute *address* is represented by a variable *address* and two messages *get-address* and *set-address*.
  - For convenience, many object-oriented data models permit direct access to variables of other objects.

Object Classes

- Similar objects are grouped into a **class**; each such object is called an **instance** of its class
- All objects in a class have the same
  - Variables, with the same types
  - message interface
  - methods

  They may differ in the values assigned to variables
- Example: Group objects for people into a *person* class
- Classes are analogous to entity sets in the E-R model

Class Definition Example

```
class employee {
    /*Variables */
      string     name;
      string     address;
      date       start-date;
      int          salary;
         /* Messages */
      int          annual-salary();
      string     get-name();
      string     get-address();
      int          set-address(string new-address);
      int          employment-length();
};
```

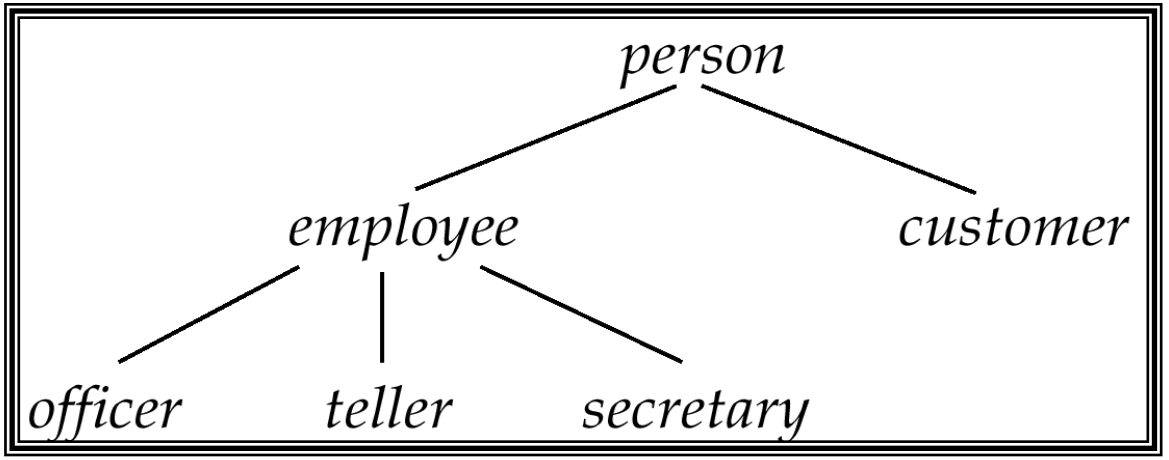
- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
  E.g.
  - `int employment-length() { return today() - start-date;}`
  - `int set-address(string new-address) { address = new-address;}`

Inheritance

- E.g., class of bank customers is similar to class of bank employees, although there are differences
  - both share some variables and messages, e.g., *name* and *address.*
  - But there are variables and messages specific to each class e.g., *salary* for employees and *credit-rating* for customers.
- Every employee is a person; thus *employee* is a specialization of *person*
- *Similarly, customer* is a specialization of *person.*
- Create classes *person, employee* and *customer*
  - variables/messages applicable to all persons associated with class *person.*
  - variables/messages specific to employees associated with class *employee*; similarly for *customer*

Inheritance (Cont.)

- Place classes into a specialization/IS-A hierarchy
  - variables/messages belonging to class *person* are *inherited* by class *employee* as well as *customer*
- Result is a **class hierarchy**



Note analogy with ISA Hierarchy in the E-R model

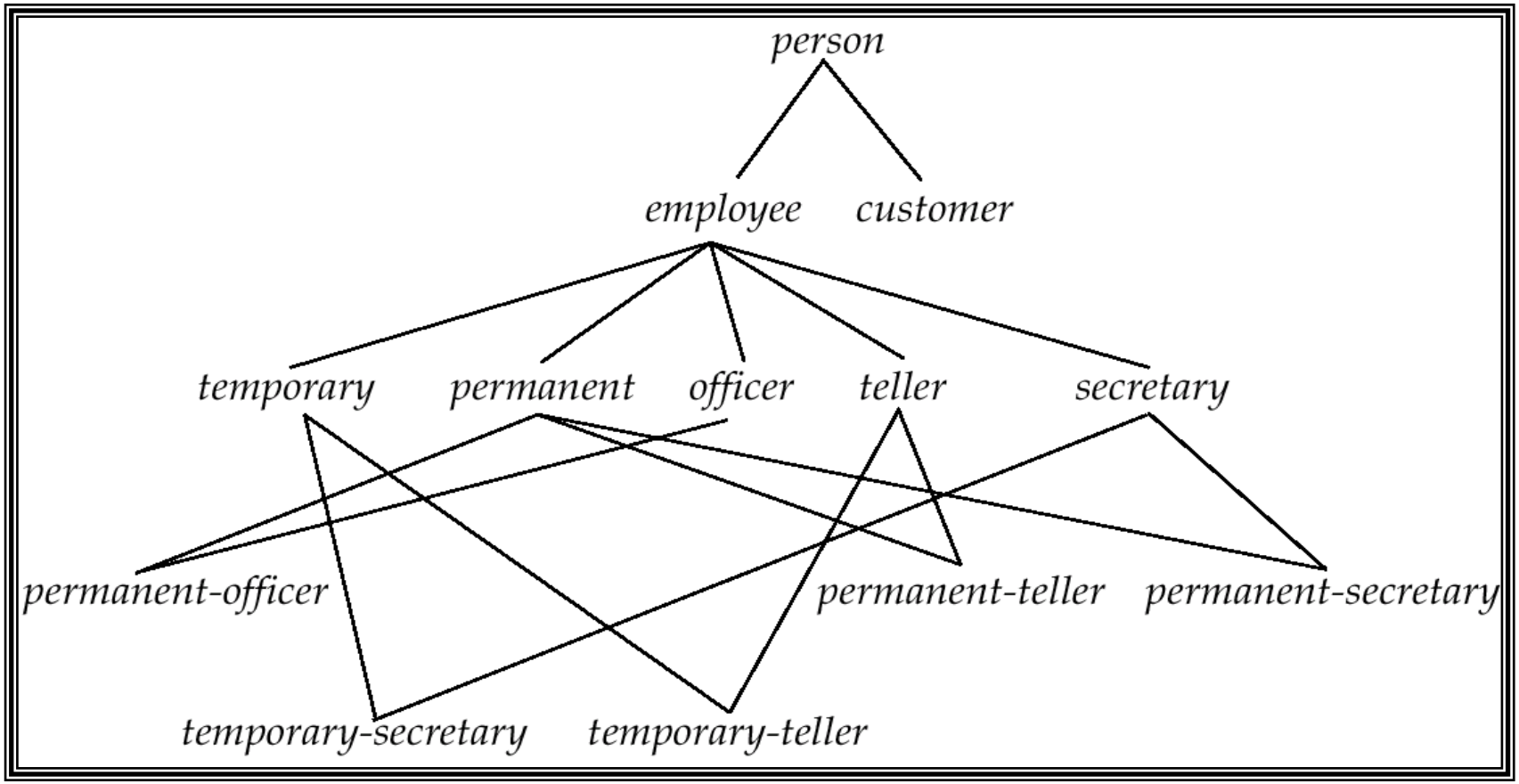
Class Hierarchy Definition

```
class person{
  string    name;
  string    address:
  };
class customer isa person {
  int credit-rating;
  };
class employee isa person {
  date start-date;
  int salary;
  };
class officer isa employee {
  int office-number,
  int expense-account-number,
  };
.
.
.
```

Class Hierarchy Example (Cont.)

- Full variable list for objects in the class *officer:*
  - *office-number, expense-account-number:* defined locally
  - *start-date, salary:* inherited from *employee*
  - *name, address:* inherited from *person*
- Methods inherited similar to variables.
- **Substitutability** — any method of a class, say *person,* can be invoked equally well with any object belonging to any subclass, such as subclass *officer* of *person.*
- **Class extent***:* set of all objects in the class. Two options:
  1. Class extent of *employee* includes all *officer, teller* and *secretary* objects.
  2. Class extent of *employee* includes only employee objects that are not in a subclass such as *officer, teller,* or *secretary*

Example of Multiple Inheritance



Class DAG for banking example.

Multiple Inheritance

- With multiple inheritance a class may have more than one superclass.
  - The class/subclass relationship is represented by a **directed acyclic graph** (**DAG**)
  - Particularly useful when objects can be classified in more than one way, which are independent of each other
    - E.g. temporary/permanent is independent of Officer/secretary/teller
    - Create a subclass for each combination of subclasses
- A class inherits variables and methods from all its superclasses
- There is potential for ambiguity when a variable/message N with the same name is inherited from two superclasses A and B
  - No problem if the variable/message is defined in a shared superclass
  - Otherwise, do one of the following
    - flag as an error,
    - rename variables (A.N and B.N)
    - choose one.

More Examples of Multiple Inheritance

- Conceptually, an object can belong to each of several subclasses
  - A *person* can play the roles of *student*, a *teacher* or *footballPlayer,* or any combination of the three
    - E.g., student teaching assistant who also play football
- Can use multiple inheritance to model "roles" of an object
  - That is, allow an object to take on any one or more of a set of types
- But many systems insist an object should have a **most-specific class**
  - That is, there must be one class that an object belongs to which is a subclass of all other classes that the object belongs to
  - Create subclasses such as *student-teacher* and *student-teacher-footballPlayer* for each combination
  - When many combinations are possible, creating subclasses for each combination can become cumbersome

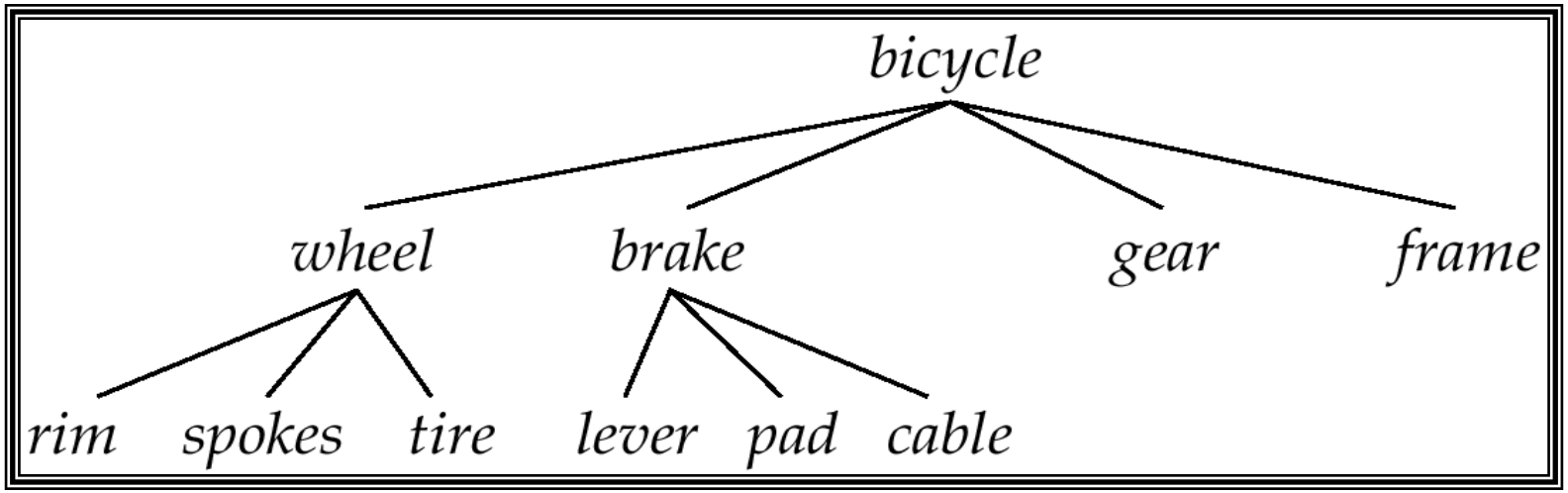
Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
  - Value – data value; e.g. primary key value used in relational systems.
  - Name – supplied by user; used for variables in procedures.
  - Built-in – identity built into data model or programming language.
    - no user-supplied identifier is required.
    - Is the form of identity used in object-oriented systems.

Object Identifiers

- **Object identifiers** used to uniquely identify objects
  - Object identifiers are unique:
    - no two objects have the same identifier
    - each object has only one object identifier
  - Can be stored as a field of an object, to refer to another object.
  - Can be:
    - system generated (created by database) or
    - external (such as social-security number)
  - System generated identifiers:
    - Are easier to use, but cannot be used across database systems
    - May be redundant if unique identifier already exists

Object Containment

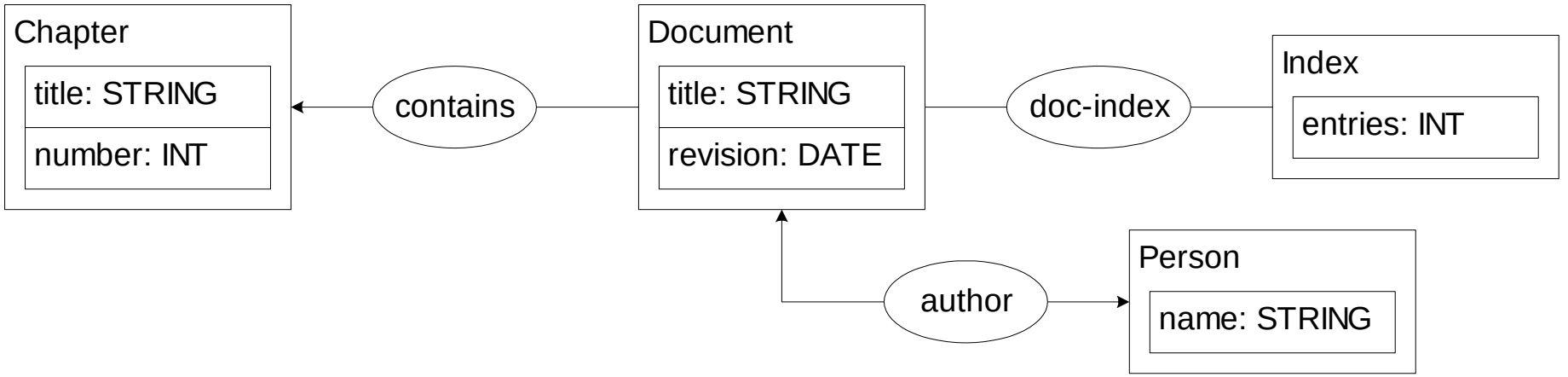


- Each component in a design may contain other components
- Can be modeled as containment of objects. Objects containing; other objects are called **composite** objects.
- Multiple levels of containment create a **containment hierarchy**
  - links interpreted as **is-part-of,** not **is-a.**
- Allows data to be viewed at different granularities by different users.

Object-Oriented Data Model

Data Schemas

- The derivative of *entity-relationship diagrams*
  - Boxes represent types of objects
    - Each attribute of each type is drawn by using a box inside the type box
  - Ovals and associated lines (possibly with arrows) represent types of relationships between objects
    - Arrows are used for the many types



OO vs. EER Data Modeling

Object Oriented

Class
Object
Association
Inheritance of attributes
Inheritance of behavior

EER

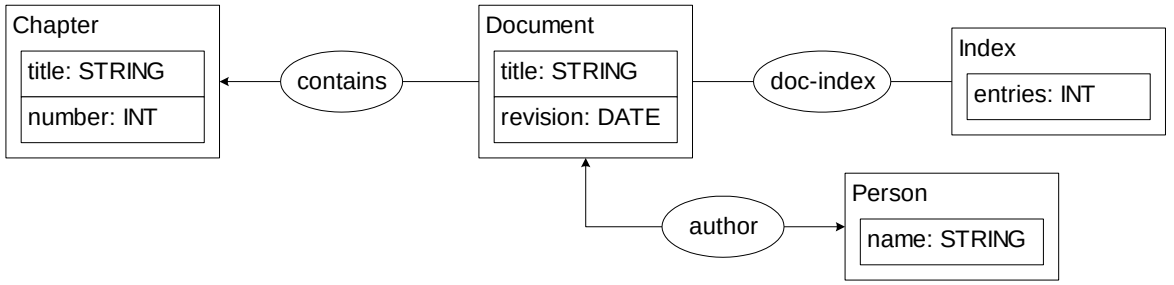
Entity type
Entity instance
Relationship
Inheritance of attributes
No representation of behavior

Object-oriented modeling is frequently accomplished using the **Unified Modeling Language (UML)**

Binary Relationships

- In general, relationships are represented using the complex attributes: references and collections of references
  - The 'back reference' syntax: "<->" (refer to Inverse Attributes)
- One-to-one relationship: by adding a reference attribute to each type
- Many-to-one relationship: by adding a reference attribute to the 'many' type and a reference-set attribute to the 'one' type
- Many-to-many relationship: by adding a reference-set attribute to each type

Binary Relationships (Cont.)



Declarations:

```
Document: {
        title: STRING,
        revision: DATE,
        chaps: LIST[Chapter] <-> doc,
        authors: LIST[Person] <-> pubs,
        index: Index <-> for
}

Chapter: {
        title: STRING,
        number: INTEGER,
        doc: Document <-> chaps
}

Index: {
        entries: INTEGER,
        for: Document <-> index
}

Person: {
        name: STRING,
        pubs: LIST[Document] <-> authors
}
```

Binary Relationships (Cont.)

Declarations:

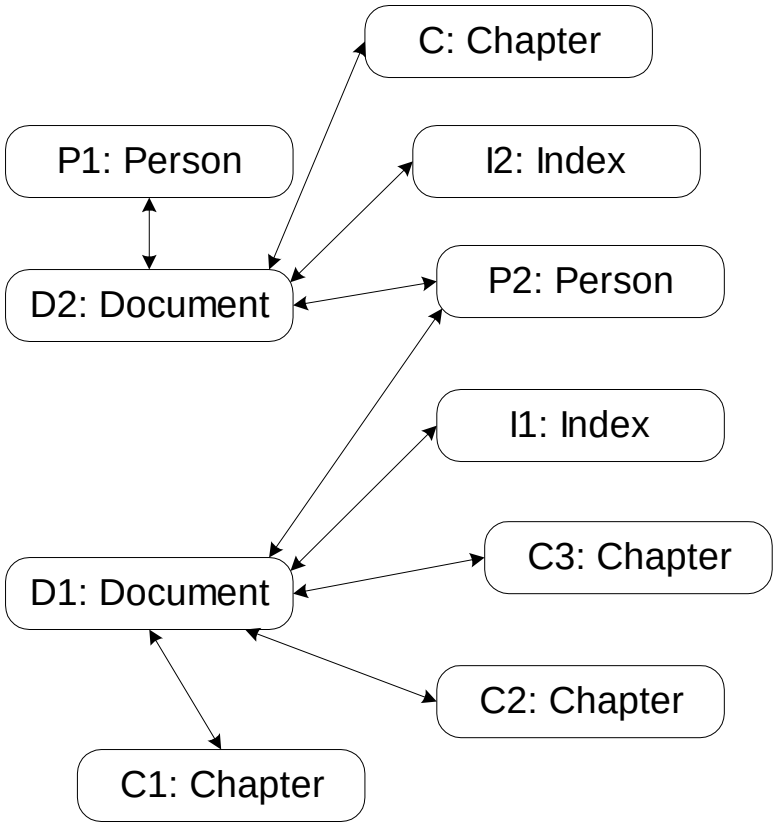
```
Document: {
        title: STRING,
        revision: DATE,
        chaps: LIST[Chapter] <-> doc,
        authors: LIST[Person] <-> pubs,
        index: Index <-> for
}

Chapter: {
        title: STRING,
        number: INTEGER,
        doc: Document <-> chaps
}

Index: {
        entries: INTEGER,
        for: Document <-> index
}

Person: {
        name: STRING,
        pubs: LIST[Document] <-> authors
}
```

Instance example:



Inverse Attributes

- Names must be given to the new attributes used to represent relationships
- The syntax used to define these attributes not only define the attribute of the type, but also the inverse attribute (from the other type), using the double arrow "<->"
- Both attributes are called *inverses*, and represent exactly the same information
- The inverse attribute pair must be kept in sync with each other.

Referential Integrity

Levels of referential integrity:

1. No integrity checks
   - The references are maintained by the application program
2. Reference validation
   - The system ensures that references are to objects that exist and are of the correct type
   - Achieved in either of two ways:
     - The system may delete objects automatically when they are no longer accessible by the user
     - The system may require that objects be deleted explicitly when they are no longer used, but may detect invalid references automatically

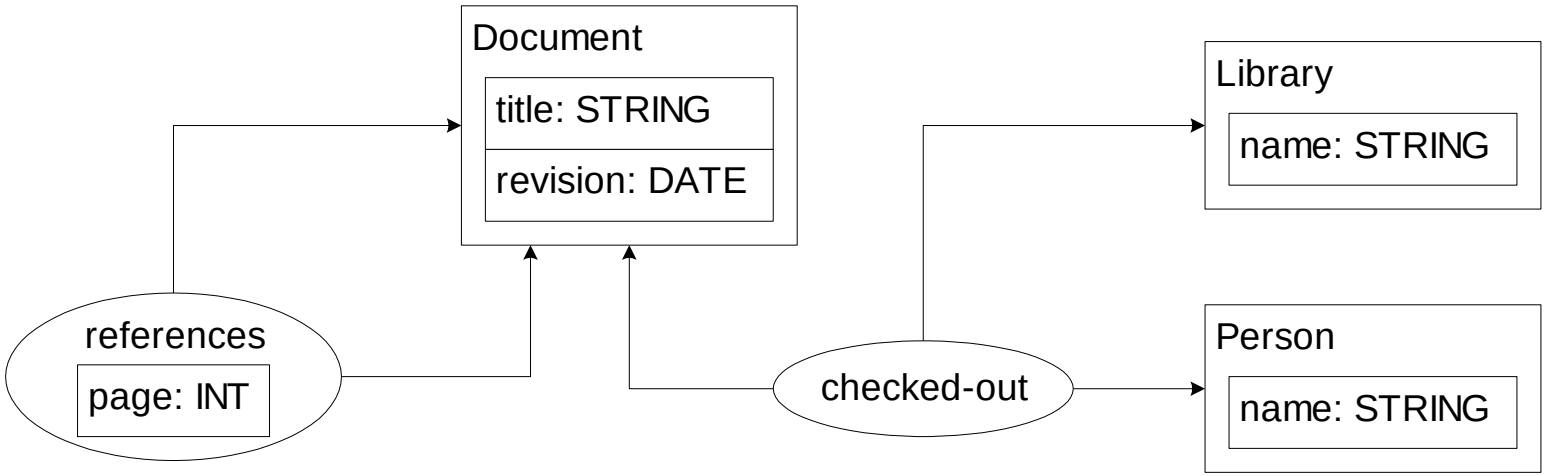
Referential Integrity (Cont.)

Levels of referential integrity (Cont.):

3. Relationship integrity
   - The system may allow explicit deletion and modification of objects and relationships, and may maintain automatically the correctness of relationships as seen through all objects.
     - Provided through inverse attributes
4. Custom reference semantics
   - The systems allow the database designer to specify custom-tailored referential integrity semantics for each type of object or relationship
   - Custom reference semantics could be encoded in methods associated with objects

Non Binary Relationships

- It is necessary to create new object types to represent these non binary relationships
  - Even if one of the attributes of the relationship is literal value

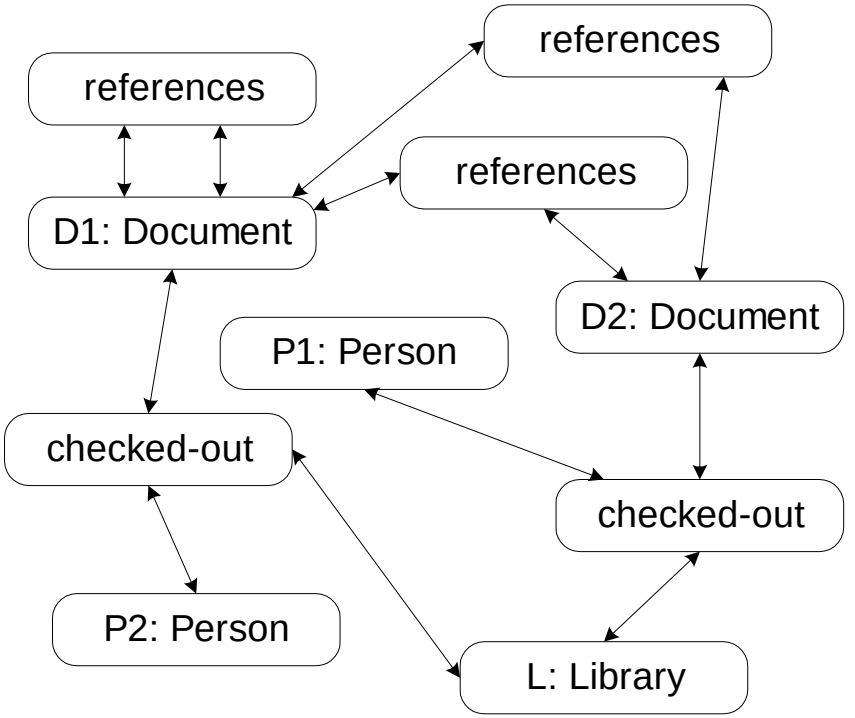


Non Binary Relationships (Cont.)

Declarations:

```
Person: {
        ...
        borrowed: LIST[checked-out] <-> who
}
Document: {
        ...
        refs: LIST[references] <-> from,
        refby:LIST[references] <-> to,
        borrowed: LIST[checked-out] <-> book
}
references: {
        from: Document <-> refs,
        to: Document <-> refby,
        page: INTEGER
}
checked-out: {
        who: Person <-> borrowed,
        from: Library <-> checkouts,
        book: Document <-> borrowed
}
Library: {
        name: STRING,
        checkouts: LIST[checked-out] <-> from
}
```

Instance example:



Relationships in OO vs Relational

- Relationships in Relational Model is represented using primary (and foreign) keys, while in OO Model is using OIDs
- Relationship tables are approximately equivalent to the intermediate objects
- In OO, all relationships are packaged in attributes, while in Relational, it is needed to examine the entire schema to get all foreign keys
- OO models allow ordering of relationships using lists

Aggregation
(more on Object Containment)

- Reasons to single out aggregation from other relationships are:
  - The DBMS can provide conveniences automatically on the basis of the composite-object abstraction (e.g. on deletion and copying)
  - Composite objects can be used for physical clustering of objects
- Most ODMSs provide a declaration mechanism to specify that a user-defined relationship is the aggregation

  E.g.: `doc: PARENT Document <-> chaps;`

Procedures

- ODMSs provide a language for database access that is computationally complete, i.e. the database language can perform the same operations as a programming language can. Additionally:
  - To associate procedures with database objects
  - To store procedures in a database
- Issues in implementation:
  - Procedures may be defined and bound at compile time or it may be possible to define new ones at run time
  - Mostly, procedures are stored in conventional programming-language binary files, not in the database

Schema Evolution

- Implications of making changes in a data schema:
  - Modification to programs that use the old data schema $\rightarrow$ most substantial, most of ODMSs offer no assistance
  - Modification of existing instances of the modified types
  - Effects of the changes on the remainder of the schema

Taxonomy of Schema Changes

- Changes to the components of a type:
  - Changes to attributes: add, drop, change the name, change the type, inherit a different attribute definition
  - Changes to methods: same as above
- Changes to inheritance graph/tree:
  - Add a new supertype/subtype relationship
  - Remove a supertype/subtype relationship
  - Change the inheritance ordering
- Changes to types themselves:
  - Add a new type
  - Drop an existing type
  - Change the name of a type

Schema Evolution (Cont.)

- Changes can be effected on existing objects by:
  - Fixing them immediately, at the time the schema is changed
  - Using a type representation that allows object type instances with different sets of attributes or relationships to coexist, and be updated as the instances are used → lazy evaluation
- The previous taxonomy has not covered all the changes possibilities.
  - E.g. splitting an attribute

Dynamic Type Definition

- Is also called run-time types
- Is used to avoid necessity to declare all types a program will use before the program is executed
- Two problems:
  - Defining the new schema
    - Solved if the ODMS allows dynamic schema operations
  - Manipulating objects using the new schema
    - Existing programs are not aware of new schema → requires schema concurrency control or versions

ODMS Architecture

Database Architecture

The way in which an ODMS is implemented and integrated with other system components

- Extended database systems
  - Provide new functionality through new or extended database query languages that incorporate procedures and other ODMS features
  - Programmers are presented with two separate environments:
    - The application programming language
    - The extended database query language

    Either language can invoke the other, but the two have different type systems and run-time execution environments

Database Architecture (Cont.)

- Database programming languages
  The term **database/programming system architecture** is more accurate
  - Extend existing programming languages (such as C++, Java) to provide persistence, concurrency control, and other database capabilities
  - Both the query language and the programming language execute in the application program environment, sharing the same type system and data workspace
  - Also called **persistent programming language**
  - It is distinguished with embedded SQL in:
    - Query language is fully integrated with the host language and both share the same type system.
    - Any format changes required in databases are carried out transparently.
  - Drawbacks:
    - Easy to make programming errors that damage the database
    - Harder to do automatic high-level optimization
    - Do not support declarative querying well

Database Architecture (Cont.)

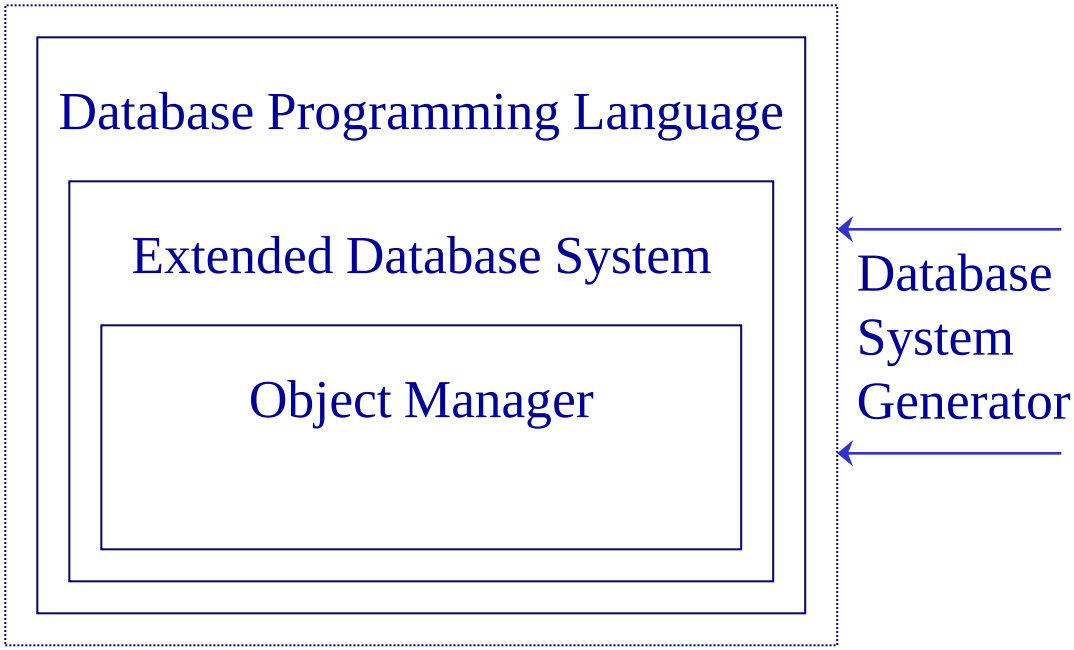
- Object managers (persistent object stores)
  - Packages that may be regarded as extensions of existing file systems or virtual memory
  - Provides a repository for persistent objects
  - Do not include a query or programming language
  - Important for simple applications

Database Architecture (Cont.)

- Database system generators
  - Allows a database implementor to construct a DBMS tailored to particular needs
    - Logical data model, includes the query and representation capabilities of the system, e.g. keys, relationships.
    - Physical data model, includes the access methods and the ways in which logical data schemas can be mapped to them
  - Variations:
    - Toolkits: provides a powerful language for writing a DBMS together with prewritten (most commonly used) packages.
    - Constructors: provides a formalization of database architecture. A DBMS is generated by linking together existing modules and using tools and high-level languages to generate DBMS modules automatically
  - Example: MAGE2 that uses OOAG in defining the specifications of the ODBS.

Database Architecture (Cont.)

- These architectures are nested according to their respective capabilities.
- **An object manager** provides the most basic functionality: storage of persistent objects.
- **An extended database system** adds new capabilities, including a query language and more sophisticated concurrency control, on top of an object manager.
- **A database programming language** adds integration with a persistent programming language on top of an extended database system.
- **A database generator** can be used to construct all these system levels.



ODMSs Examples

- Database progr. languages
  - O_2
  - ObjectStore
  - Objectivity/DB
  - ONTOS
  - VERSANT
  - GBase
  - GemStone
  - STATICE
  - ZEITGEIST
  - MCC ORION
  - ITASCA
  - db4o
- Database system generators
  - EXODUS
  - GENESIS
- Extended relational database system
  - POSTGRES
  - Starburst
  - ALGRES
  - INGRES
  - SYBASE
  - Xidak Orion
- Object managers
  - POMS
  - Mneme
  - ObServer
  - Camelot
  - LOOM Smalltalk-80 virtual memory

Query and Programming Languages

- ODMSs are vary the most in this area
- The major dimensions to compare the languages are:
  - Environments
    - Database programming languages are extensions of existing programming languages
    - Extended database systems, in contrast, have a database language separate from the application programming language
  - Query results
    - ODMSs differ in the kind of results that can be obtained by nonprocedural or declarative queries
      - Some provide no declarative queries
      - Some provide one that produces a set or list of objects
      - Some produce relations
      - Some can produce any type of result

Query and Programming Languages (Cont.)

- Major dimensions (Cont.):
  - Encapsulation
    - Some adhere to strict encapsulation of objects
      - Provides a mechanism for data independence
    - Some violate encapsulation for the purposes of the query language → better if the access is retrieval-only
    - Some provide no encapsulation mechanism at all
  - Virtual data
    - Some allow data fetched by the database language to be defined using derived attributes and relationships
    - Virtual data provide another mechanism for data independence
  - Data model
    - The database languages differ according to data model: relational, functional, or object-oriented.
    - The data model affects syntax and semantics of language

Query and Programming Languages (Cont.)

- Major dimensions (Cont.):
  - Extended operations
    - ODMSs generally extend conventional declarative query languages with new operations (e.g. transitive closure, recursion, and rules)
  - Standardization
    - An important issue of using ODMSs is compatibility with existing query-language standards and compatibility with other ODMSs

Language Environments

- Extended database systems
  - Provide a query language and programming language with separate type systems and execution environments, which still leads to impedance mismatch
  - Query language statements must be embedded in the programming language in order to copy data between the environments
  - Applications are effectively written in two different languages
- Database programming languages
  - Provide one execution environment, procedural language, and type system
    - The database language has access to all the necessary application variables, operations, and system procedures – i.e. resource complete

Impedance Mismatch

Approaches to deal with impedance mismatch:

- Query download
  - Write queries that fetch all the objects that might be required, translate to the programming representation, and copy them back afterward
  - Problems:
    - Significant overhead on startup and completion
    - Do not utilize the power of DBMS, and limits concurrent access
- Database representation
  - Copy no more than a single data field or object from the database at a time, operating on the data in the database
  - Problem:
    - Overhead in performing each data operation

Impedance Mismatch (Cont.)

- BLOBs
  - All the information about a large object can be stored as a single BLOB field in the database
  - Problem:
    - ODMS is unaware of the internal structure of the BLOB
- Procedural database language
  - Write all or most of the application in the query language
  - The query language provides programming language operations (e.g. SYBASE)
  - Lesser extent: procedures in a programming language can be called from the query language (e.g. INGRES, POSTGRES, Starburst)
  - Problem:
    - Lack of resource completeness

Impedance Mismatch (Cont.)

- Extended programming data types
  - Relation is made to behave much like other data structures in the language (added as a new data type)
  - Problem:
    - Performance: relational model is inefficient for representation of complex data structure
- Persistent programming languages
  - Transparently persistent data structures: storing the data in an extended relational ODMS
  - Problem:
    - When the programming language data model is very different from the DBMS data model, the impedance mismatch is still occured

Query Results

- No queries
  - It is necessary to write programs that navigate access paths manually
- Collection results
  - Query facility for associative lookup that does not operate on an entire database
    - Are used to select objects satisfying constraints from a particular collection of objects
- Relation results
  - Provide a query language that results in relations from any data in database
- Open results
  - Can result in any type of data

Encapsulation

- Provide (logical) data independence through the implementation of methods
- There are 2 requirements for encapsulation and the data independence it affords:
  - Hiding: some components must be public, others are private
  - Mapping: methods define mapping between public and private components
- Strict definition of encapsulation:
  - Only methods may be in the public portion
  - Methods are defined in the procedural language
  - Methods can only see and manipulate data within the object

Virtual Data

- The most important use of derived data is to define virtual data for logical data independence:
  - When a data schema is changed, derived data can be defined for old attributes and relationships, so that existing programs and queries continue to work
- To provide full logical data independence, the mapping mechanism must work for both storage and retrieval of data values
- An alternative to deriving update methods automatically for derived data: to allow the user explicitly specify procedures to fetch and store derived data values
- Important factor for physical data independence is the use of a declarative query language
  - That is why most ODMSs some what loose this feature

Extended Operations

- Transitive closure
  - Allows relationships to be traversed iteratively to find all objects reachable from a given starting point through specific relationships
- Text pattern matching
- Rule system
  - Used in knowledge management, and defining business rules on data

Standards Issues

- Data model
  - There are many models, depending on the kind of encapsulation, inheritance, procedures, and other features provided
- Query language
  - There is no equivalent of a standard query language for any of those models
- Programming language
  - This standard is needed to allow programs to be portable across systems
- SQL
  - It is currently the only standard for database access that is widely implemented. Thus, it affects new ODMS products as well

Object Query Language

Object Query Language

- To browse easily an object database
  - either interactively (ad-hoc queries)
  - or embedded in programming language
- OQL is declarative (as opposed to procedural) which allows ODBMS seamless operations
  - algebraic transformation: expressions factorization, priority to selections, constant expressions
  - access path: index, clustering
- Stand alone:

```
select struct(n: p.name,
  s: p.sex)
from People as p
where p.age < 18
```

- Embedded in C++:

```
d_oql_execute(teenagers,
"select p from People as
  p where p.age < 18"
```

Operands

- Constants

  1.0; true; "A String"; nil/null

- Named objects

  People; president

- Variables

  p; p.age or p->age; I[i]

  – Path expressions: Let *x* be an object of class *C*.
    - If *a* is an attribute of *C*, then *x.a* = the value of *a* in the *x* object.
    - If *r* is a relationship of *C*, then *x.r* = the value to which *x* is connected by *r*.
      – Could be an object or a collection of objects, depending on the type of *r*.
    - If *m* is a method of *C*, then *x.m*(…) is the result of applying *m* to *x*.

Operators

- Numerical + - * / mod abs
- Comparison = != (or <>) > < >= <=
- Boolean not and or
- Constructors

  Object: Employee(name: "John", salary: 15000)

  Collection: Vector(1, 2, 3)

  Structure: struct(name: p.name, age: p.age)

  Literal collection: set(1, 2, 3)

Operators on Collections

- Set
  - intersect union except
- Selection
  - select result (x, y, …) from x in col1, y in col2, … where predicate(x, y, …)
- Quantificators
  - for all x in collection: predicate(x)
  - exists x in collection: predicate(x)
- Conversion
  - element(singleton)
  - flatten(collection_of_collections)

OQL Select-From-Where

SELECT <list of values>

FROM <list of collections and typical members>

WHERE <condition>

- Collections in FROM can be any expression that evaluate to a collection
- Preceding a collection is a name for a typical member (alias), followed by IN
- Typical usage of attributes:
  - If *x* is an object, the path expression can be extended to access an attribute's value
  - If *x* is a collection, it is used in the FROM list in order to access attributes of *x*.

OQL Select-From-Where, cont.

- Default type of the result: bag of structs, field names taken from the ends of path names in SELECT clause
- The fields can be renamed by giving prefix to the path with the desired name and a colon
- Changing collection type:
  - Use SELECT DISTINCT to get a *set* of structs
  - Use ORDER BY clause to get a *list* of structs
  - We can extract from a list as if it were an array

OQL Select-From-Where Examples

- Print all females in the database

```
select p
from p in Persons
where not(p.male)
```

Result is of type
set(Person)

- Print the values of all females in the database

```
select *p
from p in Persons
where not(p.male)
```

Result is of type

```
set(tuple(name:string,
          male: boolean,
          spouse: Person,
          children: list(Person),
          dob: Date))
```

OQL Select-From-Where Examples, cont.

- Print the names of all females in the database

```
select distinct p.name
from p in Persons
where not(p.male)
```

Result is of type
set(string)

- Print the name and birth-date of all females in the database who were born in July

```
select struct(name: p.name, dob: p.dob)
from p in Persons
where not(p.male)  and p.dob.month= 7
```

Result is of type
set(tuple(name:string, dob: Date))

Path Expressions Examples

- Print the names of all spouses

```
select p.spouse.name
from p in Persons
```

Result is of type set(string)

- Print the names and dob of all spouses who were born in June

```
select struct(name: p.spouse.name,
              dob: p.spouse.dob)
from p in Persons
where p.spouse.dob.month=6
```

Result is of type

set(tuple(name:string, dob:Date))

Method Invocation

- Methods are functions which implicitly have as a parameter "self" and optional additional parameters. None of the example methods have any additional parameters
- Example: Print the names and age of all employees

```
select struct(name: e.name, age: e.age())
from e in Employees
```

Aggregation

- Aggregate operator count operates on collections of elements of any type, min and max operate on collections of ordered types, avg and sum operates on collections of numbers
- Example

```
select *
from e in Employees
group by e.department as departments
order by avg(select e.salary from partition)
```

Result is of type

```
List<struct(department: integer,
partition: Bag<struct(e:Employee)> >
```

Aggregation, cont.

- Print the number of Employees

  count(Employees) Result is of type integer.

- Print the maximum number of children of any person

  max(select count(p.children) from p in Persons)

- Print the name and names of children of the persons with the maximum number of children

```
select struct(name: p. name,
              children: select c.name from c in p.children)
from p in Persons
where count(p.children)=
      max(select count(p1.children) from p1 in Person)
```

Unnesting

- Print all pairs (manager, employee) of managers in Employees.

```
select struct(manager: e.name, employee: s.name)
from e in Employees, s in e.subordinates
where count(e.subordinates) >0
```

- Print the names of all subordinates of Joe Brown

```
select s.name
from e in Employees, s in e.subordinates
where e.name="Joe Brown"
```

What do you think of the following query?

```
select e.subordinates.name
from e in Employees
```

Flatten

- Flatten gets rid of one level of set (collection) nesting, e.g. it takes something of type set(set(element)) and produces something of type set(element) by taking the union of all sets of elements

  flatten({{1,2},{3,4},{1,5}}) = {1,2,3,4,5,1}

- Print the names of employees who are supervised by someone

```
select s.name
from s in flatten(select e.subordinates
                  from e in Employees)
```

Group by

- Group by introduces a level of set nesting, and produces something of type set(tuple(label:..., partition: set(...))).
- For each salary, print the employees earning that salary.

```
select * from e in Employees
group by e.salary
```

Result is of type

```
set(tuple(salary: integer,
    partition: set(struct(e:Employee))))
```

We could also have written the previous query as:

```
select struct(salary: e.salary,
              partition:(select * from e1 in Employees
                         where e.salary=e1.salary))
from e in Employees
```

Group by … having

- "For each salary earned by more than 2 employees, print the set of names of employees earning that salary."

```
select struct(salary: e.salary,
                  names: (select e1.name
                          from e1 in partition))
from e in Employee
group by e.salary
having count(partition)>2
```

Result is of type

set(tuple(salary:integer, name:set(string))).

Indexing into a list.

- Lists are ordered sets, and are treated as arrays indexed starting from 0.
- "Print all first-born children"

```
select p.children[0]
from p in Person
where p.children<>list()
```

Result is of type set(Person)

- "Print the names of all second-born children"

```
select p.children[1].name
from p in Person
where count(p.children)> 1
```

Result is of type set(string)

Sort... in... by

- "select...from...where" creates a bag of elements from an input set, "select distinct...from...where" creates a set. To create a list of element as output from an input set, we use "sort...in...by".
- "Print the names and ages of employees by increasing age."

```
sort x in (select struct(name:e.name, age:e.age())
                  from e in Employees)
by x.age()
```

Result type is list(tuple(name:string, age:integer)).

Creating sets from lists

- A list can be converted to a set using listtoset. For example, suppose we want to print each person with children represented as a set rather than a list.

```
select struct(name:p.name,
                    children: listtoset(p.children))
from p in Person
```

Result type is set(tuple(name:string,children:set(Person)).

Set operations: difference

- "Print the employees who are supervised by none."

```
select e from Employees where not(e in
      flatten(select e1.subordinates
                  from e1 in Employees))
                        OR
(select e in Employees) -flatten(select e.subordinates
                                    from e in Employees)
```

Result type is set(Employee).

Casting

- Casting is sometimes necessary to support type inferencing.
- For example, suppose we know that all people in our database are either Employees or Consultants: "Print the names of all consultants."

```
select c.name from c in
(Persons - (select (Person)e from e in Employees))
```

Result type is set(string).

- We could also have said

```
Persons - Employees
```

- "Print names and earnings of consultants."

```
select struct(name:p.name,
              earnings:((Consultant)p).earnings())
from p in (Persons -
           (select (Person)e from e in Employees))
```

Quantifiers

- Boolean-valued expressions for use in WHERE-clauses.

  FOR ALL *x* IN <collection>:
  <condition>

  EXISTS *x* IN <collection>:
  <condition>

- The expression has value TRUE if the condition is true for all (resp., at least one) elements of the collection.

- "Print the names of people who have had at least one boy."

```
select struct(name:p.name)
from p in Persons
where exists c in p.children: c.male
```

- "Print the names of people who had all boys."

```
select struct(name: p.name)
from p in Persons
where forall c in p.children: c.male
```

Element

- Sometimes we want to turn a collection containing one element into the element type.
- Note that this is not always safe, i.e. testing whether a collection has a single element is not something that can be checked at compile time, so a run-time error may occur.

a) A collection with a single member:
Extract the member with `ELEMENT`.

b) Extracting all elements of a collection, one at a time:
   1. Turn the collection into a list.
   2. Extract elements of a list with <list name>`[i]`.

Element, an example

- "For all people with single-child families, print the person's name and their child's name."

```
select struct(name:p.name,
                    cname: element(select c.name
                                   from c in p.children))
from p in Persons
where count(p.children)=1
```

- "Print the tuple of information about John Kramer."
- If we know "name" is a key for Person, then the following query will be correct:

```
element(select *p from p in Persons
            where p.name="John Kramer")
```

Define

- The result of a query can be named by using DEFINE clause
- Example:

```
define TheFemale as
select p
from p in Persons
where not(p.male)
```

- The name can then be used in other queries within the same query session, i.e. up to the commit/abort point

Summary

- OQL is a language for complex object databases. The types include:
  - base types- integer, string, boolean
  - tuple type - tuple($l_1 : \tau_1, l_2 : \tau_2, ..., l_N : \tau_N$)
  - the collection types - unique set (set), set (bag) and list
  - objects (class types)
- For each type, basic operations (including constructor and destructor operations) are defined.

Summary, cont

- For example,
  - For objects o, we can dereference (*o) and cast ((Person)o).
  - For tuples, we can project over attributes (e.g. p.name).
  - For sets, we can select elements, flatten, extract an element from a singleton set, take unions (+ or union), set difference (- or except) and intersection (* or intersect).
  - For lists, we can extract indexed elements and create lists by specifying an ordering on elements.

OQL and SQL

- Fully compliant to ODMG, C++ and Smalltalk models
- OQL accepts path expression:
  - person.address.city.name
  - person.fornames[2]
- Constructors
  - Person(name: "Smith", fornames: array("John", "Paul"))
- Operator and operands are orthogonal
- Methods call:
  - person.age()