Brief Introduction to Database Concepts

Andrea Rodríguez

Summer School - Castellón 2004

Department of Computer Science
University of Concepción, Chile
http://www.inf.udec.cl/∼andrea
andrea@udec.cl

1 Introduction

Information and data are different. Information is understood by a person. Data are values stored on a passive medium like a computer disk. The purpose of a database management system (DBMS) is to bridge the gap between information and data - the data stored in memory or on disk must be converted to usable information.

The basic processes that are supported by a DBMS are:

- Specification of data types, structures and constraints to be considered in an application.
- Storing the data itself into persistent storage.
- Manipulation of the database.
- Querying the database to retrieve desired data
- Updating the content of the database

A database is a model of a real world system. The contents (sometimes called the extension) of a database represent the state of what is being modeled. Changes in the database represent events occurring in the environment that change the state of what is being modeled. It is appropriate to structure a database to mirror what it is intended to model.

Databases can be analyzed at different levels:

- Conceptual Modeling
- Logical Modeling
- Physical Modeling

2 Conceptual Modeling

Conceptual-level concepts permit us to model the application world in terms that are independent of any particular data (logical) model. Conceptual models provide a framework for developing a

database schema from the top to the bottom in the process of a database design. This Section examines the *entity-relationship* model and the *object-oriented* model as representatives of conceptual modeling. The entity-relationship model is widely used and the object-oriented model is gaining more acceptance for non-traditional databases.

2.1 The Entity-Relationship Model

The entity-relationship model is a tool for analyzing the semantic features of an application that are independent of events. This approach includes a graphical notation, which depicts entity classes as rectangles, relationships as diamonds, and attributes as circles or ovals. For complex situation, a partial entity-relationship diagram may be used to present a summary of the entities and relationships that do not include the details of the attributes.

The entity-relationship diagram provides a convenient method for visualizing the interrelationships among entities in a given application. This tool has proven to be useful in making the transition from an information application description to a formal database schema. The entity-relationship model is used for describing the conceptual schema of an enterprise without attention to the efficiency of the physical database design. The entity-relationship diagrams are then turned into a logical schema in which the database is actually implemented.

Short definitions of some of the basic terms that are used for describing important entity-relationship concepts are:

1. **Entity**. An entity is a thing that exists and is distinguishable.

   (a) *Entity instance.* An instance is a particular occurrence of an entity. For example, each person is an instance of an entity Person, each car is an instance of an entity Car, etc.

   (b) *Entity class.* A group of similar entities is called an *entity class* or *entity type.* An entity class has common attributes.

   In this review, I will not make distinction between *entity* and *entity class.* In other readings you may find that what I call *entity* is called *entity class* and what I call *entity instance* is just called *entity.*

2. **Attributes**. Attributes describe properties of entities and relationships.

   (a) *Simple and composite attributes.* A simple attribute is the smallest semantic unit of data, which are atomic (no internal structure). A composite attribute can be subdivided into parts, e.g., address (street, city, state, zip).

   (b) *Single and multivalued attributes.* Single attributes have a single value for a particular entity. Multivalued attributes have multiple values of an attribute for a particular entity; e.g., degrees or courses that a student can have or take.

   (c) *Domain.* Conceptual definition of attributes: a named set of scalar values, all of the same type, and a pool of possible values.

3. **Relationships**. A relationship is a connection between entities. For example, a relationship between PERSONS and AUTOMOBILES could be an "OWNS" relationship. That is to say, automobiles are owned by people.

   - *Isa hierarchies.* A special type of relationship that allows attribute inheritance. For example, to say that a truck is an automobile and an automobile has a model and serial number implies that a truck also has a model and serial number.

4. **Keys**. The key uniquely differentiates one entity instance from all others in the entity. A key is an identifier.

   (a) *Primary Key.* Identifier used to uniquely identify one particular instance of an entity. A primary key

      i. can be one or more attributes (e.g., consider substituting a single concatenated key attribute for multiple attribute key)
      ii. must be unique within the domain (not just the current data set),
      iii. its value should not change over time,
      iv. must always have a value, and
      v. is created when no obvious attribute exists. Each instance has a value.

   (b) *Candidate Key.* When multiple possible identifiers exist, each of them is a candidate key.

   (c) *Concatenated Key.* Key made up of parts which when combined become a unique identifier. Multiple attribute keys are concatenated keys.

   (d) *Borrowed Key Attributes.* If an *isa* relationship exists, the key of the more general entity is also a key of the sub entities. For example, if serial number is a key for automobiles, it would also be a key for trucks.

   (e) *Foreign Keys.* Foreign keys reference a related table through the primary key of that related table.

An ER schema may identity certain constraints to which the content the data must conform. Two of the most important types of constraints are:

1. The *mapping cardinality* of a relationship indicates the number of instances in entity E_1 that can or must be associated with instances in entity E_2:

   (a) *One-One Relationship.* For each entity instance in one entity there is at most one associated entity instance in the other entity. For example, for each husband there is at most one current legal wife (in this country at least). A wife has at most one current legal husband.

   (b) *Many-One Relationships.* One entity instance in entity E_2 is associated with zero or more entity instances in entity E_1, but each entity instance in E_1 is associated with at most one entity instance in E_2. For example, a woman may have many children but a child has only one birth mother.

   (c) *Many-Many Relationships* There are no restrictions on how many entity instances in either entity are associated with a single entity instance in the other. An example of a many-to-many relationship would be students taking classes. Each student takes many classes. Each class has many students.

   Mapping cardinality is derived from *cardinality constraints.* The cardinality constraint between two entities E_1 and E_2, denoted by (m,n), specifies that an instance in E_1 appears in E_2 at least m and at most n times. Mapping cardinality takes the maximum number of the cardinality constraint for each entity in a relationship.

2. *Existence dependence.* If the existence of an entity instance x depends on the existence of an entity instance y, then x is said to be *existence dependent* on y.

2.2 Entity-Relation Diagram

Symbols used in entity-relationship diagrams include:

- **Rectangles** represent ENTITY or ENTITY CLASSES
- **Circles** represent ATTRIBUTES
- **Diamonds** represent RELATIONSHIPS
- **Arcs** connect entities to relationships. Arcs are also used to connect attributes to entities. Some styles of entity-relationship diagrams use arrows and double arrows to indicate the one and the many in relationships. Some use forks etc.
- **Underlined attributes** identify keys of entities.

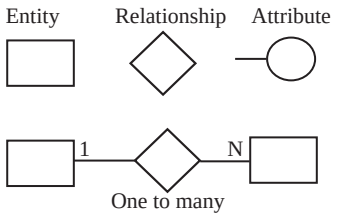

Figure 1: Diagram component of a ER conceptual schema

Consider, for example, a model of a cadastral application that does not consider the geometry of spatial objects. Such model can be described by the diagram of Figure 2.

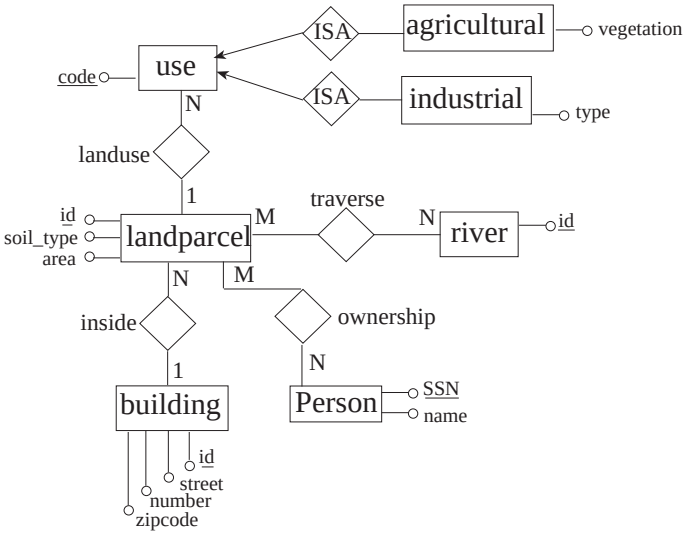

Figure 2: A simplified ER Conceptual schema for a cadastral application

Additional material for entity-relationship modeling in found in [6]

2.3 Object-Oriented Database Model OO

The aspects associated with an object-oriented modeling are:

1. **Object Structure**. Loosely speaking, an object corresponds to an entity instance in the ER model. Objects both know things (they have attributes) and they do things (they have methods). The object-oriented paradigm is based on encapsulating data and code related to an object into a single unit. Conceptually, all interactions between an object and the rest of the system are via messages. Thus, the interface between an object and the rest of the system is defined by a set of allowed messages. In general, an object has associated with it:

   (a) a set of variables that contain the data for the object; variables are attributes in ER;

   (b) a set of messages to which the object responds; and

   (c) a set of methods, each of which is a body of code to implement messages.

2. **Object Classes**. A class group objects in a database that share a common definition. The notion of a class corresponds to the notion of entity-class, or just entity, in the ER model. A class is a representation of an object and, in many ways, it is simply a template from which objects are created. Classes form the main building blocks of an object-oriented application. A class object includes:

   (a) a set-value variable whose value is the set of all objects that are instances of the class;

   (b) implementation of a model for the message new, which creates a new instance of the class;

   (c) inheritance upon which variables and methods of a class are inherited from a super-class. A super-class establishes a hierarchy similar to the concept of specialization (ISA, relation). For example, an employee is a person such that employee inherits variables and methods from person;

   (d) multiple inheritance in which case a class inherits from multiples super-classes; and

   (e) and object identity.

2.4 UML: Unified Modeling Language

UML is a visual modeling language of general purpose. The UML combines certain number of graphical elements into diagrams. Because it is a language, the UML has rules for combining these elements. UML consists of nine basic diagrams, where UML class diagrams (Object Management Group 2003) are the mainstay of object-oriented analysis and design. UML class diagrams show the classes of the system, their interrelationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Class diagrams are used for a wide variety of purposes, including both conceptual/domain modeling and detailed design modeling.

Classes are typically modeled as rectangles with three sections: the top section for the name of the class, the middle section for the attributes of the class, and the bottom section for the methods of the class. Attributes are the information stored about an object (or at least information temporarily maintained about an object), while methods are the things an object or class do. For example, a class hospital has attributes such as name, number of beds and address. Hospitals also check-in patient, check-out patient, and transfer patient. Those are all examples of the things that happen at a hospital, which get implemented (coded) as methods.

An important consideration is the appropriate level of detail. In class *Hospital*, the attribute *address* is composed of street, number, zip code, city and country. One could, therefore, model this

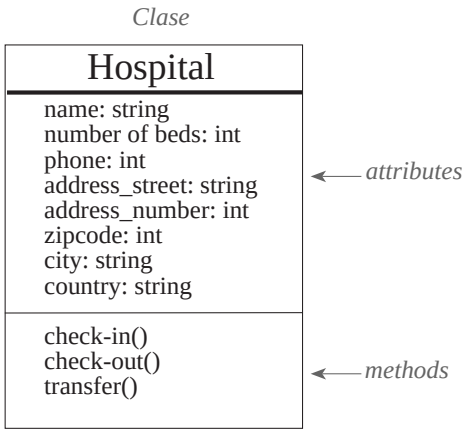

Figure 3: A class in UML class diagram

situation by using a different class *Address* and associating *Hospital* with *Address*. By introducing the *Address* class, the *Hospital* class has become more cohesive. It no longer contains logic (such as validation) that is pertinent to addresses. In addition, the *Address* class could now be reused in association with other classes.

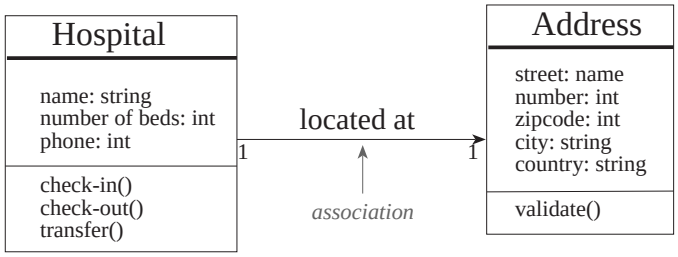

Figure 4: Influence of the detail in modeling classes

Methods in the class may be of several types;

- *public* (+) if the method is visible for all classes,
- *private* (-) if the method is only visible to the members of the class, and
- *protected* (#) if the method is visible to the member of the class and all subclasses

Objects are often associated with, or related to, other objects. When you model *associations* in UML class diagrams, you show them as a thin line connecting two classes. Associations can become quite complex; consequently, you can depict some things about them on your diagrams. The *label*, which is optional, although highly recommended, is typically one or two words describing the association. *Multiplicity* of an association is the degree in which each class participates in the association. The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction (See Table 1).

Another option for associations is to indicate the direction in which the label should be read. This is depicted using an arrow, called a direction indicator. Direction indicators should be used

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only n (where $n > 1$)
0..n	Zero to n (where $n > 1$)
1..n	One to n (where $n > 1$)

Table 1: Multiplicity Indicators

whenever it isn't clear which way a label should be read. My advice, however, is if your label is not clear, then you should consider rewording it. The arrowheads on the end of the line indicate the directionality of the association. A line with one arrowhead is uni-directional whereas a line with either zero or two arrowheads is bidirectional. Officially you should include both arrowheads for bi-directional assocations, however, common practice is to drop them. At each end of the association, the role, the context an object takes within the association, may also be indicated. This indication may be used only in cases when it isn't clear from the association label what the roles are, if there is a recursive association, or if there are several associations between two classes.

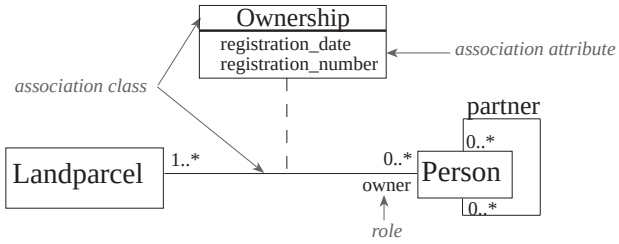

Figure 5: Notation for associations

Two other characteristics of associations are qualifiers and constraints. A qualifier is a value that selects an unique object among the set of objects related by the association. Constraints applies rules that the associations must satisfy. These properties in the design of associations are illustrated in the following example:

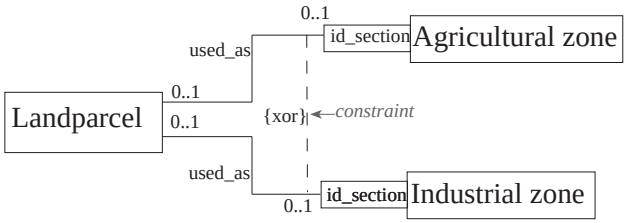

Figure 6: Qualifiers and constraints in associations

Similarities often exist between different classes. Very often two or more classes will share the same attributes and/or the same methods. *Inheritance* models "is a" and "is like" relationships, enabling you to reuse existing data and code easily. When A inherits from B, we say A is the subclass of B and B is the superclass of A. Furthermore, we say we have *pure inheritance* when A inherits all the attributes and methods of B. The UML modeling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass. In same cases, it is reasonable to create an *abstract class* that joins common attributes and methods of subclasses. In abstract classes, objects are not created directly from it, they capture the similarities between subclasses. Abstract classes are modeled with their names in italics, as opposed to concrete classes, classes from which objects are instantiated, whose names are in normal text.

Sometimes an object is made up of other objects, having *composition* or *aggregation* associations. For example, a building is composed of several rooms and a rooms may be composed of several sub-rooms, i.e., you can have recursive composition. A composition is a static association in the sense that the existence of the part depends on the existence of the whole. Such type of associated is represented by a filled diamond. In an aggregation, the existence of the part is independent of its whole. An aggregation is represented by an empty diamond.

As a general example of a model using UML, consider a cadastral application where different classes and associations are depicted (Figure 7).

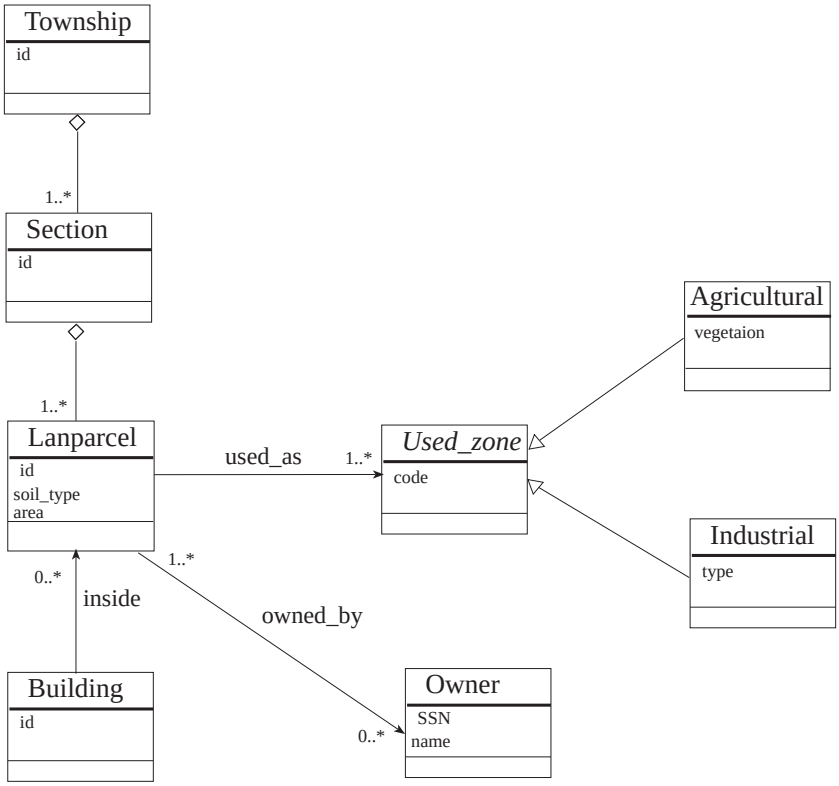

Figure 7: A cadastral application in UML

A more complete tutorial of UML can be found in [2].

3 Logical Data Model and Query Languages

At the logical level, the conceptual schema is translated into the data model of a particular DBMS. A logical model is described as a set of relatively simple structures. In addition to data representation, a DBMS needs to specify the data manipulation, which is done through expressing queries and other operations in a data manipulation language. The following Section reviews the main concepts associated with relational and objects-oriented databases.

3.1 Relational Data Model Concepts

In the relational data model the database is represented as a group of related tables. The relational data model was introduced in 1970. It is currently the most popular model. The mathematical simplicity and ease of visualization of the relational data model have contributed to its success. The relational data model is based on the mathematics of set theory, whose basic components are the following.

1. **Relation**. A two dimensional table. A relation is a collection of tuples, each of which contains values for a fixed number of attributes. Relations are sometimes referred to as flat files, because of their resemblance to an unstructured sequence of records. Each tuple in a relation must be unique; that is, there can be no duplicates.

2. **Attribute**. A table column. Other commonly used terms for attribute are *property* and *field*. The set of permissible values for each attribute is called the domain for that attribute.

3. **Tuple**. A table row. A tuple is an instance of an entity or relationship or whatever is represented by the relation.

4. **Key**. A single attribute or combination of attributes whose values uniquely identify the tuples of the relation. That is, each row has a different value for the key attribute(s). The relational model requires that every relation have a key and that:

   - no two tuples may have the same key value and
   - every tuple must have a value for the key attribute (the key fields have non-null values).

There are two restrictions on the relational model that are sometimes circumvented in practice:

1. Duplicate tuples are not permitted. If two tuples are entered with the same value for each and every attribute, they are considered to be the same tuple. In practice this restriction is sometimes overcome by assigning unique line or tuple numbers to each entry, thus assuring that it is unique.

2. No ordering of tuples within a relation is assumed. In practice, however, one method or another of ordering tuples is often used.

Following the example of the cadastral application of Figure 2, the ER conceptual model is mapped onto a relational model. Using the *relation schema* that corresponds to the programming notion of type definition, a portion of the relational schema of the cadastral application is:

$$
\begin{aligned}
Landparcel - schema &= (landparcel\_id, soil\_type, area) \\
Building - schema &= (building\_id, street, number, zipcode, city, landparcel\_id) \\
Person - schema &= (person\_SSN, person\_name) \\
River - schema &= (river\_id) \\
Ownership - schema &= (landparcel\_id, person\_SSN) \\
Traverse - schema &= (river\_id, landparcel\_id)
\end{aligned}
$$

Note that in this schema, two relations (i.e., *ownership* and *traverse*) are specified as independent tables, whereas the relation *inside* between *building* and *landparcel* is defined by incluiding a *foreign key* in the relation *building*, since each building is inside of only one landparcel.

SQL, or standard query language, is the widely accepted language used in all relational database. The SQL2 data definition language (DDL) can be used to specified the previous relational schema. SQL2 provides a specific clause (**foreign key**) to declare that one or several attributes reference a tuple in another relation.

Create Table *Landparcel*
(*landparcel_id* **integer**,
soil_type **varchar(30)**,
area **integer**
Primary Key (*landparcel_id*))

Create Table *Building*
(*building_id* **integer**,
street **varchar(30)**,
number **integer**,
zipcode **integer**,
city **varchar(30)**,
Primary Key (*building_id*),
Foreign Key (*landparcel_id*) **References** *Landparcel*))

Create Table *Ownership*
Primary Key (*building_id*, *person_SSN*),
Foreign Key (*landparcel_id*) **References** *Landparcel*),
Foreign Key (*person_SSN*) **References** *Person*))

SQL is declarative, that is, it expresses queries without specifying how the system must operate to compute the result of a query. In addition, SQL relies on solid theories such that what is expressed by SQL can be defined by two equivalent formal languages: *relation calculus*, essentially a first order language, and *relational algebra*, a set of operations that describes how relations are manipulated to answer queriers. These operation are:

1. *Select operation* ($\sigma_{predicate}$). It selects tuples in a relation that satisfy a given predicate. For example,

   $\sigma_{area>10000}(Landparcel)$

   will return all tuples from the entity *Landparcel* with $area > 10000$.

2. *Project operation*($\pi_{attributes}$). It extracts a subset of attributes of the relation. For example,

   $\pi_{landparcel\_id,soil\_type}(Landparcel)$

   will return the *id* and *soil_type* of all landparcels.

3. *Union operation* $\cup$. It results in the union of the data sets coming from the two input relations. For example,

   $\pi_{landparcel\_id}(Traverse) \cup \pi_{landparcel\_id}(Building)$

   will return the *id* of landparcels that contain buildings *or* are traversed by rivers.

4. *Cartesian product operation* ($\times$). It combines information from any two relations. For example,

$\pi_{street,number,zipcode}(\sigma_{building.building\_id=ownership.building\_id}(\sigma_{person\_SSN=249873245}(Building \times Ownership)))$

will return the address of buildings that are owned by a person with $SSN = 249873245$.

5. *Set intersection operation.* It results in the intersection of the data sets coming from the two input relations. For example,

   $\pi_{landparcel\_id}(Traverse) \cap \pi_{landparcel\_id}(Building)$

   will return the *id* of landparcels that contain building *and* are traversed by rivers.

6. *Set difference operation.* It finds tuples that is one but not in another relation. For example,

   $\pi_{landparcel\_id}(Traverse) - \pi_{landparcel\_id}(Building)$

   will return the *id* of landparcels that contain building but are not traversed by rivers.

7. *Join operation* ($\bowtie$). A join operation allows us to combine certain selection and cartesian product into one operator. For example, a query example for cartesian product:

   $\pi_{street,number,zipcode}(\sigma_{building.building\_id=ownership.building\_id}(\sigma_{person\_SSN=249873245}(Building \times Ownership)))$

   can be expressed with the *join* operation as:

   $\pi_{street,number,zipcode}(\sigma_{person\_SSN=249873245}(Building \bowtie Ownership))$

8. *Division operation.* It takes two relations, one binary relation and one unary relation, and gives the values of an attribute of the binary relation that correspond with values in the unary relation. For example,

   $\pi_{landparcel\_id,person\_name}(Ownership \bowtie Person) \div \pi_{landparcel\_id}(Building)$

   will return the *id* of landparcels and the *name* of these landparcels' owners for all landparcels that contain buildings.

SQL language expresses queries with "**select from where**" *clause*. Some of the previous queries may be expressed by:

Query 1:

select $landparcel\_id, area, soil\_type$
from $Landparcel$
where $area > 10000$

Query 2:

select all $landparcel\_id, soil\_type$
from $Landparcel$

Query 3:

(**select all** $landparcel\_id$
from $Traverse$)
union
(**select all** $landparcel\_id$
from $Building$)

Query 4:

select $street, number, zipcode$
from $Buildingb, Owershipo$
where $b.landparcel\_id = o.landparcel\_id$ **and** $o.person\_SSN = 249873245$
from $Building$

Query 5:

(**select all** $landparcel\_id$
from $Traverse$)
intersect
(**select all** $landparcel\_id$
from $Building$)

A more complete tutorial of SQL language can be found in [1].

3.2 Object-Oriented Model

In an object-oriented environment, each objects class in represented by a class using an object-oriented language. The following will describe the way a conceptual model based on UML is mapped onto a logical model using an object-oriented language. To this effect, we will use OQL proposed by the Object Database Management Group (ODMG) [4]. This language allows end users to access complex structures, to run methods, and return results that make use of specific constructors, such as sets, lists, bags, and arrays.

Consider the cadastral application described in UML of the previous Section. OQL works with the object description language (ODL) in much the same way that SQL. The ODL schema of the cadastral application is the following:

class *Landparcel* {
(extent Landparcel)

attribute integer *id*;
attribute string *soil_type*;
attribute integer *area*;
relationship *Section in_section*
inverse *Section*:: *landparcels*
relationship *Building contains*
inverse *Building*:: *inside*
relationship *Person owned_by*
inverse *Person*:: *owns*
relationship *Use used_as*
inverse *Use*:: *in_use*
};

class *Section* {
(extent Section)

attribute integer *id*;
relationship Set< *Landparcel* > *landparcels*
inverse Landparcel:: *in_section*
relationship*Township in_town*
inverse *Township*:: *sections*
};

class *Township* {
(extent Township)

attribute integer *id*;
relationship Set< *Section* > *sections*
inverse *Section*:: *in_town*
};

class *Building* {
(extent Building)

attribute integer *id*;
relationship *Landparcel inside*
inverse *Landparcel*:: *constains*
};

class *Person* {
(extent Person)

attribute integer *SSN*;
attribute string *name*;
relationship *Landpacel owns*
inverse *Landparcel*:: *owned_by*
};

class *Use* {
(extent Use)

attribute integer *code*;
relationship *Landparcel in_use*
inverse *Landparcel*:: *used_as*
};

class *Agricultural* : Use {
(extent Agricultural)

attribute string *vegetation*;
};

class *Industrial* : Use {
(extent Industrial)

attribute string *type*;
};

Based on these definitions, OQL can express different types of queries:

Query 1: retrieve *id*, *area* and *soil_type* of landparcels with area larger than 10000
select $l.id, l.area, l.soil\_type$
from l **in** *Landparcel*
where $l.area > 10000$

Query 2: retrieve *id*, *area* and *soil_type* of all landparcels
select $l.id, l.soil\_type$
from l **in** *Landparcel*

Query 3: Retrieve *id* of sections with landparcels that contain buildings
select *s.id*
from s **in** *Section*, l **in** *s.landparcels*
where exists b **in** *l.contains*

Query 4: Retrieve area of sections as an aggregation of the area of their landparcels grouped by use code
select *s.id*, *sumArea*: **sum**(**Select** *p.l.area* **from partition** *p*)
from s **in** *Section*, l **in** *s.landparcels*, u **in** *l.used_as* **group by** LUSE: u.code

Review [5, 3] for more details about ODL and OQL review.

4 Physical Model

Al the physical level, a DBMS is in charged of:

- *Storage.* The representation of efficient organization of data in a persistent secondary unit.
- *Access Methods.* Organization of data to accelerate data retrieval by defining data structures or index.
- *Query Processing.* The set of operations to answer a query. Such operations defines algorithms that make use of access methods.
- *Query Optimization.* Strategy of evaluation of query processing.
- *Concurrency and recovery.* Strategy to manage concurrent access to data and resources from several users and the recovery of the database after a system failure.

References

[1] A. Cumming. *A Gentle Introduction to SQL: An interactive tutorial*. School of Computing of Napier University, Edinburg, UK, URL: http://sqlzoo.net/, 2003.

[2] S. Kalajdziski. *UML Tutorial*. URL: http://odl-skopje.etf.ukim.edu.mk/uml-help/, 2004 (access).

[3] ODL. *EyeDB: The Object Definition Language*. Sysra Informatique, URL: http://www.infobiogen.fr/services/eyedb/pub/manual/node5.html, 1999.

[4] ODMG. *Object Database Management Group: The Standard for Storing Objects*. ODMG, URL: http://www.odmg.org/, 2000.

[5] OQL. *OQL - Object Query Language*. UCSD: Department of Computer Science and Engineering, URL: http://www.cs.ucsd.edu/classes/wi00/cse132a/oql.htm, 2000.

[6] Information Technilogy Services. *Introduction to Data Modeling*. University of Texas at Austin, URL: http://www.utexas.edu/its/windows/database/datamodeling/dm/erintro.html, 2004 (access).