2. OBJECT-BASED DATABASES

An object-oriented database system is a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.

Complex Data Types

Traditional database applications have conceptually simple datatypes. The basic data items are records that are fairly small and whose fields are atomic.

In recent years, demand has grown for ways to deal with more complex data types. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries.

On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow structured datatypes that allow a type address with subparts street address, city, state, and postal code.

Structured Types

Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute name with component attribute *firstname* and *lastname*:

```
create type Name as
(firstname varchar(20),
lastname varchar(20))
final;
```

Such types are called user-defined types in SQL. The final and not final specifications are related to subtyping.

The components of a composite attribute can be accessed using a “dot” notation; for instance, name. firstname returns the firstname component of the name attribute. An access to attribute name would return a value of the structured type Name.

We can also create a table whose rows are of a user-defined type. For example, we could define a type Person Type and create the table person as follows

```
create type PersonType as (
name Name,
```

address Address,

dateOfBirth date)

not final

create table person of PersonType;

Type Inheritance

Suppose that we have the following type definition for people:

create type Person

(name varchar(20),

address varchar(20));

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

create type Student

under Person

(degree varchar(20),

department varchar(20));

create type Teacher

under Person

(salary integer,

department varchar(20));

Both Student and Teacher inherit the attributes of Person—namely, name and address. Student and Teacher are said to be subtypes of Person, and Person is a supertype of Student, as well as of Teacher.

Table Inheritance

Sub tables in SQL correspond to the E-R notion of specialization/generalization. For instance, suppose we define the people table as follows:

create table people of Person;

We can then define tables students and teachers as sub tables of people, as follows:

create table students of Student

under people;

create table teachers of Teacher

under people;

The types of the sub tables (Student and Teacher in the above example) are subtypes of the type of the parent table (Person in the above example). As a result, every attribute present in the table people is also present in the sub tables students and teachers.

Array and Multiset Types in SQL

SQL supports two collection types: arrays and multisets

A multiset is an unordered collection, where an element may occur multiple times. Multisets are like sets, except that a set allows each element to occur at most once.

Suppose we wish to record information about books, including a set of keywords for each book. Suppose also that we wished to store the names of authors of a book as an array; unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author, and so on. The following example illustrates how these array and multiset-valued attributes can be defined in SQL:

```
create type Publisher as
(name varchar(20),
branch varchar(20));
create type Book as
(title varchar(20),
Autho_arrray varchar(20) array [10],
Pub_date date, publisher Publisher, keyword_set varchar(20) multiset);
create table books of Book;
```

The first statement defines a type called Publisher with two components: a name and a branch. The second statement defines a structured type Book that contains a title, an author array, which is an array of up to 10 author names, a publication date, a publisher (of type Publisher), and a multiset of keywords. Finally, a table books containing tuples of type Book is created.

Object-Identity and Reference Types in SQL

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type Department with a field name and a field head that is a reference to the type Person, and a table departments of type Department, as follows:

```
create type Department (
name varchar(20),
```

head ref(Person) scope people);

create table departments of Department;

Here, the reference is restricted to tuples of the table people. The restriction of the scope of a reference to tuples of a table is mandatory in SQL, and it makes references behave like foreign keys.

2.1 Object-relational Features

Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system. However, to minimize changes to the storage-system code (relation storage, indices, etc.), the complex datatypes supported by object-relational systems can be translated to the simpler type system of relational databases.

Sub tables can be stored in an efficient manner, without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes that are defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the super table, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the super tables. Access to all attributes of a tuple is faster, since a join is not required.