APPENDIX A1

Introduction to Informix

Jonas Mureika and Edriss N. Merchant
University of Southern California Brain Project,
University of Southern California, Los Angeles, California

Informix SQL Tutorial: A Practical Example

As a practical example of SQL in Informix, we will review the steps to create a portion of the NeuroCore database (see Chapter 3.1 and 3.2 for a complete description of the NeuroCore database). A sub-structure of the contact portion of the database will be featured. Specifically, our example database will be used to store contact information for researchers in the lab. In order to construct the best possible (and least confusing) database schema of this type, the following questions should be addressed: *who* are our contacts, *where* do they work, and *how* does one contact them?

This exemplar database will demonstrate the following versatile features of an Informix database: referential integrity, set backing tables, and inheritance. These elements will be explicitly defined when they are introduced later in the text. For a quick-reference guide to the syntax used, please see Appendix B1.

The first step in the creation of our database is to create the database. This will initialize the database and create the necessary internal information needed by the database server. To create this database, which we will name *contactDB*, we can execute the following SQL command:

```
CREATE DATABASE contactDB;
```

The database will contain a set of relational (inheritance-based) tables that will store the information necessary to contact the persons stored in this database. Each table is defined as containing certain columns where one can store information related to the concept defined by the table. An entry into the table is referred to as a *tuple*. Relations between tables can be defined so that columns in a table can reference values stored in columns residing in other tables. The structure of a database is known as the *schema*. The schema for our exemplar database can be seen in Fig. 1.

In defining the schema for our contact database, the first objects we must define are the contacts we wish to store. In our case, we will store the information concerning both persons and the labs to which they belong. Inheritance is immediately evident in this structure. In our example, the tables, "lab" and "person" are children of "contact." Both labs and persons are contacts and should inherit all the properties of a contact. For example, all contacts in the database will have a phone number. Instead of defining this at each individual table, we can define this relation for a parent-table (i.e., "contact") which will then be passed down to its children, namely "person" and "lab." However, we must now define how persons and labs are related. A set backing table is a special table that allows a many-to-many relationship between other tables to be defined. For example, many persons may be associated with a lab and, similarly, many labs may be associated with a person. The set backing table holds the identification elements (relations) to link the two tables. The "_labPerson" table is such a set backing table that identifies which individuals in the database belong to which lab. Once we have defined the contacts we wish to store in the database, we need to define how we want to contact them. In our example, we will store telephone contact information. For each contact, we can define various types of phone numbers (e.g., office, home, mobile). To accomplish this, one can define a "phone" table, which can contain the individual tuples that define the phone contact information for each contact. However, in order to classify the types of phone information stored in the database, we will need to define a list of allowable phone

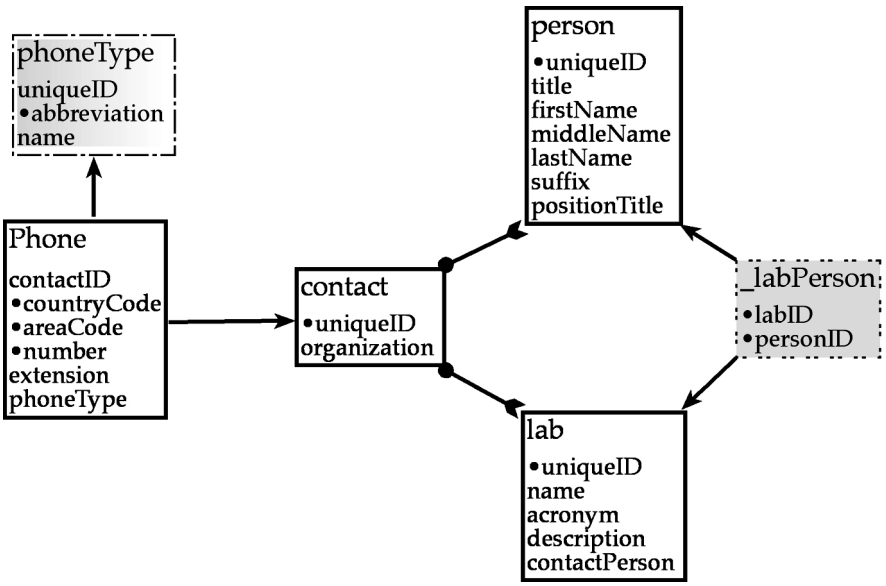

Figure 1 Database schema for the SQL tutorial. Each table is represented as a single entity, with the relations depicted as arrows. Inheritance (parent-child relations) are depicted as arrows with round heads.

types. The "phoneType" table is a dictionary table that is referred to by the "phone" table. Each tuple stored in the "phone" table must reference a valid entry in the "phone" type table.

The first step in implementing the database for our contact information is to create a parent table for the contacts we want to maintain, namely "contact." All types of contacts (labs, people) are stored in tables that are children of the "contact" table. This allows us to associate various entities and relations with all contacts (e.g., phone numbers). Because a row is actually a datatype, it facilitates matters to manipulate the tables in terms of row datatypes created by the syntax:

```
CREATE ROW TYPE contact_t
(
    uniqueID        INT8            NOT NULL,
    organization    VARCHAR(128)
);

CREATE TABLE contact OF TYPE contact_t
(
    PRIMARY KEY (uniqueID)
);
```

These statements create a new row type that defines what a contact is and then creates the actual table to store that information. In order to reference tuples within the database, one must define a collection of columns that contain unique values. This allows any item within the table to be uniquely referred to. A collection of such columns is termed a *primary key* and is defined in the "create table" statement. The primary key for all tables within the contact hierarchy is the uniqueID. We next create a table called "lab" to store the names of the locations/institutes at which the persons work:

```
CREATE ROW TYPE lab_t
(
    name            VARCHAR(128)    NOT NULL,
    acronym         VARCHAR(32),
    description     LVARCHAR,
    contactPerson   INT8
)
under contact_t;

CREATE TABLE lab OF TYPE lab_t
(
    UNIQUE (uniqueID)
    UNIQUE (name, organization)
)
under contact;
```

Notice that, because the lab is a sub-table of contact (it inherits information from the contact super-table), the syntax of the statement has changed slightly. For a sub-table we need to define *under* which table the new information will reside in the inheritance hierarchy. Each lab must have a unique name (the keyword UNIQUE ensures a one-to-one correspondence between the pair [name, organization] and "lab"). That is, there cannot be two labs with the same name in the same organization.

The table that describes the actual person will be denoted as "person":

```
CREATE ROW TYPE person_t
(
    title           VARCHAR(16),
    firstName       VARCHAR(64) NOT NULL,
    middleName      VARCHAR(64),
    lastName        VARCHAR(64) NOT NULL,
    suffix          VARCHAR(16),
    positionTitle   VARCHAR(128)
)
UNDER contact_t;

CREATE TABLE person OF TYPE person_t
(
    UNIQUE (uniqueID)
)
UNDER contact;
```

The major stipulations for this table are that the person must have a first and last name; all other entries can be blank (NULL). To further develop the structure of the database, we need to create a set backing table to relate persons and labs. In this case, the table "_labPerson" (and associated datatype "_labPerson_t") relates the entries in the tables "lab" and "person". This implements a many-to-many relationship between the entries in the lab table and the entries in the person table. The syntax for such a table is

```
CREATE ROW TYPE _labPerson_t
(
    labID       INT8        NOT NULL,
    personID    INT8        NOT NULL
);

CREATE TABLE _labPerson OF TYPE
    _labPerson_t
(
PRIMARY KEY (labID, personID),
FOREIGN KEY (labID) REFERENCES lab
    (uniqueID),
FOREIGN KEY (personID) REFERENCES person
    (uniqueID)
);
```

The table consists entirely of primary keys. Each tuple stored in this table contains an ID for a person and the lab they belong to. A set backing table can be used as a way of joining tables in one-to-many or many-to-many relationships. In other words, we can store information on people who belong to a single lab, people who belong to multiple labs, and labs containing a single person or multiple people. We have now implemented the information necessary to define a contact.

In our example, we chose the telephone as the best method of contacting an individual; however, to store the phone data we need to know what type of phone is being stored (e.g., cellular, home, office). To accomplish this, a dictionary table was implemented:

```
CREATE ROW TYPE phoneType_t
(
    abbreviation    VARCHAR(16) NOT NULL,
    name            VARCHAR(64) NOT NULL
);

CREATE TABLE phoneType OF TYPE
    phoneType_t
(
UNIQUE (name),
PRIMARY KEY (abbreviation)
);
```

In this case, the new datatype "phoneType_t" has been assigned to contain two strings of variable length (16 and 64 characters, respectively). Note the stipulation NOT NULL; this is a check to ensure that the entries *do* contain some value. The keyword UNIQUE acts as a check to the database contents and structure. Unique instructs the database that the name variable in the table "phoneType" cannot be assigned the same value more than once. That is, no phone type should have exactly the same name (cellular, beeper, etc.).

Because the table "phoneType" only describes the type of phone number being stored, we need to create another table with more explicit contents in which to store the actual phone contact data. This will be the table "phone," which has the associated datatype "phone_t":

```
CREATE ROW TYPE phone_t
(
    contactID       INT8            NOT NULL,
    countryCode     VARCHAR(8)      NOT NULL,
    areaCode        VARCHAR(8)      NOT NULL,
    number          VARCHAR(32)     NOT NULL,
    extension       VARCHAR(8),
    phoneType       VARCHAR(16)     NOT NULL
);

CREATE TABLE phone OF TYPE phone_t
(
PRIMARY KEY (countryCode, areaCode, number),
FOREIGN KEY (phoneType) REFERENCES
    phoneType (abbreviation),
FOREIGN KEY (contactID) REFERENCES
    contact (uniqueID)
);
```

Here, the variable "phoneType" should not be confused with the table of the same name. Note, however, that in the table creation syntax the varchar "phoneType" introduced in the definition of the datatype "phone_t" is assigned as a FOREIGN KEY that REFERENCES "abbreviation" in the table phoneType (likewise for "contactID" and "uniqueID"). This is an example of *referential integrity*. The value of the VARCHAR(16) "phoneType" is the same as the value of "abbreviation" in the "phoneType" table. They are

referentially linked. The foreign key is the same value as the primary key it references. Ideally, such primary/foreign key links can propagate down through the database and create an intricate and delicate web of joined tables. It would be easy to destroy such a fragile structure if the primary key were to be (accidentally) deleted. Luckily, this is usually not permitted in a database.

Another of the versatile features of SQL is the ability to create indices associated with a data entry. Creating an index speeds up queries to the database by providing the query processor with additional information to search a specific table. To demonstrate this feature, we index parts of the datatype "person:"

```
CREATE INDEX person_asc_ix ON person
    (title ASC, firstName ASC, middleName ASC,
    lastName ASC, suffix ASC)
    USING btree;

CREATE INDEX userInfo_asc_ix ON person
    (username ASC) USING btree;
```

This is the general form of the syntax used to create an index. An index is a structure of pointers that point to rows of data in a table. An index optimizes the performance of database queries by ordering rows to make access faster. The end result of the index is that it provides a secondary access method to the data. Creating indices on data can help speed search and retrieval times for arbitrarily large databases. Indices can list their contents in ascending order (ASC) or descending order (DESC), and one can index the same data using both methods.

We have now finished the implementation of our contact database. The next step in the implementation is to actually store data in the database. To illustrate this process, we will add two individuals to the database who work in the same laboratory. First, we add the laboratory and then the individuals:

```
INSERT INTO lab (uniqueID, name, organization,
    acronym, description) values (1, 'Our Example
    Lab', 'Our University', 'OEL', 'A lab for our
    example database');
```

Notice that we only insert data into the lab table and not into the lab and contact table, even though the organization column is defined in the contact table. We must then add our two individuals:

```
INSERT INTO person (uniqueID, firstName, middle-
    Name, lastName) values (2, 'John', 'Q', 'Public');

INSERT INTO person (uniqueID, firstName, middle-
    Name, lastName) values (3, 'Jane', NULL, 'Doe');
```

As we inserted data into the person table, notice that we did not define all the values contained within the person table. We defined only the values we were required to and had values for. Also notice that Jane does not have a middle name, so we have declared that value to be NULL, containing no value. Once we have our people and labs defined, we will need to define which lab our people belong to. In our example, John does not belong to a lab, but Jane does. To define this, we can insert the following values into the "_labPerson" table:

```
INSERT INTO _labPerson values (1,3);
```

Notice that we did not define the actual columns we were inserting data into. Because we did not select specific columns in the insert statement, we must provide data for all columns defined in the table and list them in the order in which they were defined. In our example above, we have defined that the person with the uniqueID = 3 belongs to the lab with uniqueID = 1. Now that we have defined our contacts, we must define the contact information:

```
INSERT INTO phoneType values ('office', 'Office
    Number');
INSERT INTO phone values (2, '001', '213–555',
    '1212', 'office');
```

Here we have inserted a new phone type, namely a person's office phone. We are now able to insert an office phone number into the phone table. If we had not defined a phone type, we would not be able to insert any values into the phone table, as each phone number must be of a particular type (this policy is enforced through the use of referential integrity defined by the foreign key – -primary key structure). We have entered a phone number for John. Notice that the contactID refers to John's uniqueID in the contact hierarchy. Now that we have entered some contact information, we can now perform queries to extract information from the database. To list all persons in the database we can perform the simple query:

```
SELECT * FROM person;
```

Here we have defined that we wish to select all (*) columns from the persons table. This will return information for John and Jane. A more complex query would allow us to select John's phone numbers:

```
SELECT person.firstName, phone.areaCode, phone.
    number FROM
phone,
person
WHERE
            phone.contactID == person.uniqueID
            and person.firstName == 'John';
```

Here we are selecting for all tuples in the phone table where the contact ID is equal to the ID of persons with a firstName of John. Notice that we are only returning the area code and phone number from the records stored in the phone table. It is hoped that the previous section has demonstrated various features of a typical database and how a user would design, implement, and query such a database.

For more information on SQL and Informix's implementation of SQL, users can visit the documentation available at the Informix website http://www.informix.com.