Invited Paper

Database Design Methodology: Converting a Relational Schema into an Object-Relational Schema

Sunit Gala* and Won Kim*

Abstract

The objective of this paper is to give a brief review of relational design methodology, and show how a relational schema can be converted into an object-relational schema. We first introduce relational terminology and describe the relational design process along with its limitations. Next we show how the relational model can be extended to overcome these limitations; these extensions form the core of an object-relational data model; we use the UniSQL/X data model as an example. We illustrate the process or converting a relational schema into an object-relational schema, and show the benefits of this conversion with respect to queries. The conversion process is then summarized as a set of guidelines. Finally, we make our conclusions.

1. Introduction

The relational model represents data in a database as a collection of relations. A relation is a table of values; each row in the table represents an ordered collection of related data values. In other words, a data value is the row-column intersection of a table. The table name and column name help interpret the meaning of values in each row of the table. A row is also called a tuple;it corresponds to a record or an instance of a table. Also, the terms column and attribute are used interchangeably. Consider the following example:

Employee					
Name	Age	SSN	Salary	Mgr	Dept
Tim	29	789789	50000	NULL	Acct
Jim	27	456456	40000	789789	Sales
Joe	25	123123	45000	456456	Sales

Figure 1.

* UniSQL, Inc. Their address is 9390 Research Ⅱ, Suite 200 Austin, Texas 78759-6544, USA.

The Employee table has 6 attributes(columns):Name, Age, SSN, Salary, Mgr, and Dept, and 3 tuples or rows.

An important question in relational design is what constitutes a goodrelational schema. That is, how does one decide whether one grouping of relations and attributes in these relations is quantitatively superior than another. Various formal techniques have been developed to determine the "goodness" of one schema design over another for the same application. A good relational schema should help users clearly understand the meaning of data value (tuples) in each table, and hence formulate correct queries, appropriately create new data, and delete or update existing data. Further, a good design must avoid, or at least minimize, anomalies for the various data manipulation operations. These anomalies can be minimized by:

1. reducing the number of redundant values in tuples,
2. reducing the number of null values in tuples, and
3. disallowing spurious tuples.

While a complete description of relational design theory [Date 1986, Elmasri & Navathe 1989 Ullman 1982], is beyond the scope of this paper, we briefly outline a few important concepts, and mention the various limitations of relational design.

Constraints in a relational schema are defined in terms of dependencies, the most important type being functional dependencies. Before defining a functional dependency, we define the notion of keys. A key is a time-invariant property of a table, for example, the SSN of an Employee. A table may have many keys, each called a candidate key. An Employee's Name is a possible candidate key. One key is chosen as the primary key of a table, for example, SSN. All candidate key values must be unique for every row in the table; a primary key value can never be null. Thus values for Name and SSN must be unique, and further, values for SSN cannot be null.

A table T_1 can refer to another table T_2 via one or more columns of T_1 which is the primary key of T_2. Attributes in this column are called a foreign key of T_1. For example, the Dept column in Employee refers to theprimary key of a Department table, and is a foreign key of the Employee table. A row (or table) in one table can refer to an existing row in another table via the foreign key--primary key relationship. This is called referential integrity. Enforcing this referential integrity avoids dangling or spurious tuples.

A primary key value is used to uniquely identify a row in the table. Values for other columns in a given row of the table are said to functionally depend upon the primary key value. Thus, it makes sense to group such dependent columns (or attributes) into one table. Information about primary keys, func

tional dependencies and foreign keys for all tables in a relational schema is used to normalize the schema. This information is also used to decompose unsatisfactory table definitions into smaller tables (that is, tables containing fewer columns) to reduce redundancy and spurious tuples.

A number ofqr algorithms have been devised to normalize and decompose a relational schema based on various dependencies. However, a functional problem is that the database designer must specify all functional dependencies before these algorithms can be applied. This is a non-trivial task for a large application schema with hundreds of attributes. Failure to specify even one or two important dependencies can result in an understandable design. Also, these algorithms sometimes depend on the order in which the dependencies are specified. Thus many possible designs may arise corresponding to the same set of functional dependencies. For the above reasons, the relational design methodology based on these algorithms is not popular.

Additionally, there are some fundamental problems with the relational paradigm itself. First the relational approach does not provide any ontological basis for representing the application, hence the need for inventing dependency theory. An ontological basis provides a one-to-one mapping between real-world objects and their machine representation (that is, the database system). Second, the relational model can capture only structural aspects of a real-world entity, not its behavior. Most relational vendors do provide stored procedures and triggers as a work-around to model behavior, but without a sound theoretical basis. Third, data values in row-column intersections are limited to atomic or primitive data types only. This implies that the domain of an attribute cannot be a user-defined table. It also rules out the ability to capture one-to-many relationships in a natural manner. Fourth, all tables in a relational schema are free-floating structure; they can be related to one another only via foreign keys and primary keys. This makes it impossible to arrange the tables in inheritance or aggregation (that is, compression) hierarchies. In other words, a relational schema lacks a natural structure which can give users meaningful insight into the application.

The above problems can be either entirely eliminated or greatly ameliorated by carefully making object-oriented extensions to the relational model[Kim 1991].

2. Extending the Relational Model

An object-relational model makes 4 fundamental extensions to infuse the relational paradigm with object-orientation. Below, we describe each extension and look at an example in the next section.

Extension Ⅰ : Tables can be valid data types.

The data type of an attribute or column can be any existing table in the database. This allows the user to define and manage arbitrary data types. Complex, nested data can be modeled and accessed directly by applications. This is made possible by the notion of system-wide object identity (OID), that is, every object in the database has a unique, persistent identifier. The OID is also used as mechanism for one tuple (or object) to point to another. The OID provides an ontological bases, that is, a one-to-one mapping between real-world objects and their machine representations.

Further, this extension eliminates the need for foreign keys;instead a data value can be the OID of another tuple to provide direct access. Thus data residing in multiple tables can be accessed directly and more efficiently without an explicit join;this is also known as pointer chasing. This extension also permits common data to be stored only once so it can be shared;it reduces maintenance overhead associated with redundant data.

Extension Ⅱ : A row-column intersection can contain more than one value.

The domain of an attribute can be specified as a "set of" values. That is, a row-column intersection can now be multivalued. This set can consist of primitive domains such as strings and integers, or of user-defined tables; for the latter case, the value would be a set of OIDs. This simplifies the creation of one-to-many and many-to-many data relationships.

Extension Ⅲ : Tables can have procedures.

User-written C programs and other applications can be registered with a specific table. These procedures can be invoked by other users and applications, and executed by the object-relational database system. Procedures can be reduced to transparently support table-specific behavior. True application independence from procedural data is thus provided because table-specific behavior can be hidden form calling applications. This feature is usually exploited in conjunction with the next extension.

Extension Ⅳ : Tables can be organized in an inheritance hierarchy.

Unlike a relational schema which consists of a set of free-floating tables, they can now be organized in an inheritance hierarchy. This hierarchy is reproduced as a directed acyclic graph. Thus a table can have may

inherit(reuse) the columns and procedures of the parent table(s); a child table may also redefine (that is, make table-specific) the columns and procedures of the parent table(s).

Thus, we have shown how the object-oriented paradigm can form the basis for a data model that subsumes the relational (and pre-relational) data models. Solutions to most of the data modeling or design related difficulties of relational database systems are inherent in an object-oriented data model. Relational systems are designed to manage only limited types of data, suchas integer, floating-point number, string, Boolean, date, time, and monetary. In other words, they are not designed to manage arbitrary user-defined data types. On the other hand, a central tenet of an object-oriented data model is the uniform treatment of arbitrary data types and the facility to add new data types. Like the relational model, an object-oriented data model allows the representation of data, and relationships and constraints on that data. Additionally, it also allows the encapsulation of data with programs that operate on the data, to provide a uniform framework for the treatment of arbitrary user-defined data types.

Further, the object-oriented paradigm, through the notions of encapsulation and inheritance (reuse), is fundamentally designed to reduce the difficulty of designing, developing and evolving complex software systems or designs. This is precisely the goal that has driven the data management technology from file systems to relational database systems during the past three decades. An object-oriented data model thus inherently satisfies the objective of reducing the difficulty of design and evolving very large and complex database applications. The extensions defined above are quintessential for enhancing productivity in database application development. They have been implemented in the UniSQL/X object-relational database management system. Below is a table showing equivalent terms for the relational and object-oriented paradigms.

Relational Term	Object-oriented Term
Domain	(Abstract) Data Type
Column	Attrribure
Row(tuple)	Instance
Table	Class
Procedure	Method
Table Hierarchy	Class Hierarchy
Child Table	Subclass
Parent Talbe	Superclass
Extension Ⅰ	Type system
Extension Ⅱ	Set-valued attributes
Extension Ⅲ	Encapsulation
Extension Ⅳ	Reuse/Inheritance

Figure. 2.

3. Converting a Relational Schema into an Object-Relational Schema

We now illustrate the conversion of a relational schema into a corresponding object-relational schema with the help of an example. We first describe a simple banking application and its relational implementation, which we then incrementally extend into an object-relational implementation. Consider a bank which has three kinds of accounts: Checking, Savings, and Checking Plus (an interest bearing checking account). A client can have more than one account, possibly of different kinds. The system must support four fundamental operations: opening a new account, transferring money from one account to another, depositing cash into an account, and withdrawing cash from an account. For each transaction recorded for each account, the date, type, amount and check number (if applicable) are recorded, and the current balance is computed. Shown below is a relational implementation of this application.

The Client table has 5 attributes describing a client (client number, name and address), of which *client_#* is the primary key. The three kinds of accounts are described by the Savings, Checking and check

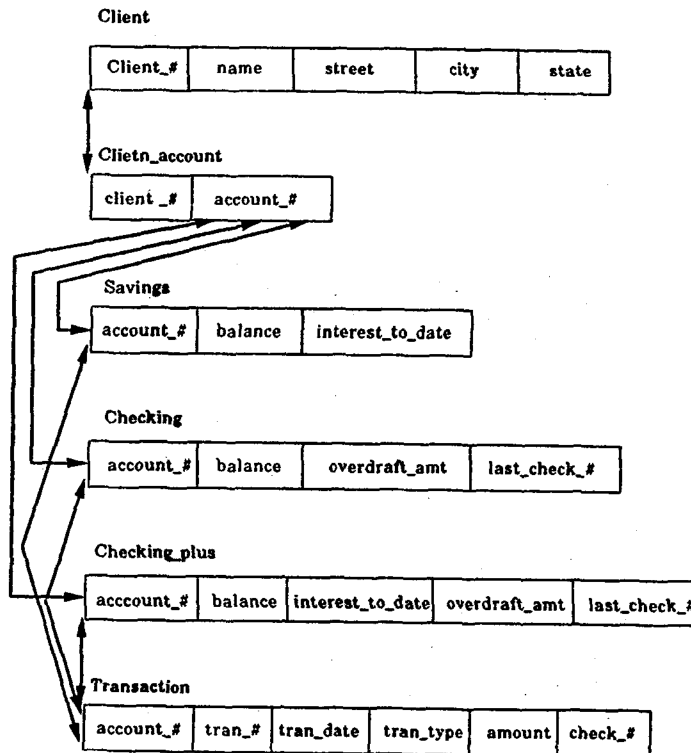

Figure. 3

ing_plus tables respectively. The primary key in eachtable is acount_#. Recall that a client can have one or more accounts, but the relational model does not support set-valued attributes. Therefore, an artificial table called Client_account is created to capture this 1:m relationship. Its primary key is the pair(client_#, account_#);both these attributes are also foreign keys. If a given client has 2 or more accounts, there will be 2 or more entries in the Client_account table, an entry for each account belonging to each client. Thus retrieving information about a given client's balance in one or all accounts will result in at lease a 3 table join. Further, a history of all transactions associated with a given account is maintained in the Transaction table, whose primary key is tran_#, and foreign key is account_#.

The ANSI SQL syntax for defining the above schema is shown below.

```
create table Client
(       client_# integer,
        name char(25),
        street varchar(50),
        city char(15),
        state char(2)               );

crete table Client_account
(       account_# integer,
        client_# integer            );

create table Savings
(       account_# integer,
        balance monetary,
        interest_to_date float      );

create table Checking
(       account_# integer,
        balace monetary,
        overdraft_amt monetary,
        last_check_# integer        );

create table Checking_plus
(       account_# intger,
        balance monetary,
        interest_to_date float,
        overdraft_amt monetary,
        last_check_# integer        );

create table Transaction
(       account_# integer,
        tran_# integer,
        tran_date date,
        tran_type chat(5),
        amount monetary,
        check_# integer             );
```

3.1 Extension I: Tables can be valid data types

Very often, spouses bank at the same institution, implying that they both are clients and they share an address. Since each have an account (they may or may not share an account), their address information will be redundant. If a couple moves, one of them will call the bank to inform the move; the bank will update the address of the calling spouse. To update the other spouse's address, the calling spouse either has to have the spouses's client_# or the bank must maintain spouse information. Thus, there is a potential for inconsistency.

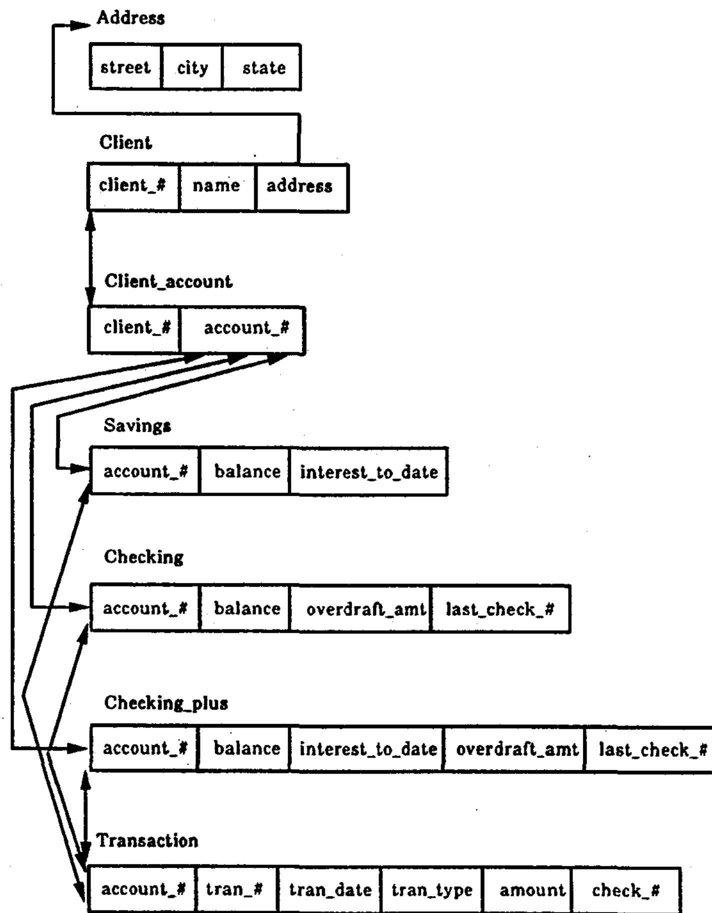

Figure. 4

To avoid such problems, it is better to eliminate redundant data. The above scenario can be avoided if the Client table entries for spouses shared address information. Therefore, we define a new table called Address, and replace the street, city and state attributes in Client with a single attribute address whose domain is the user-defined table Address. This extension is shown below. Now either spouse can call in with address-change information without needing to know the other's client_#.

The Client table is appropriately modified to reflect this change as follows:

```
create table Address
(       street string,
        city string,
        state char(2)           );

create table Client
(       client_# integer,
        name string,
        address Address         );
```

As another example, let us redefine the Employee table seen earlier as follows:

Employee						
OID	Name	Age	SSN	Salary	Mgr	Dept
OID $_1$	Tim	29	789789	50000	NULL	Acct
OID $_2$	Jim	27	456456	40000	OID$_1$	Sales
OID $_3$	Joe	25	123123	45000	OID$_2$	Sales

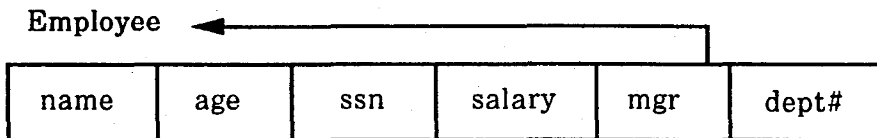

Figure 5.

The table now includes a recursive relationship since the domain of mgr is the Employee table itself. The object-relational syntax for its definition is:

```
create table Employee
(       name string,
        age integer,
        ssn string,
        salary moneraty,
        mgr Employee,
        dept string            );
```

3.2 Extension II: A row-column intersection cn contain more than one value.

Recall that a client can have one or more accounts, but the relational model does not support set-valued attributes, and that the Client_account table was created to capture this relationship. The first two extensions in conjunction allow us to eliminate this artificial table by defining an attribute called accounts in the Client table, whose domain is the set of Savings, Checking or Checking_plus tables. That is, the accounts attribute in Client can have a set of values such that one or more values can be the object identifier of an instance of any of these three tables. Similarly, the other one-to-many relationship between Account and Transaction can be captured by defining an attribute in each account table called trans whose domain is a set of Transaction tuples. These modifications are shown below.

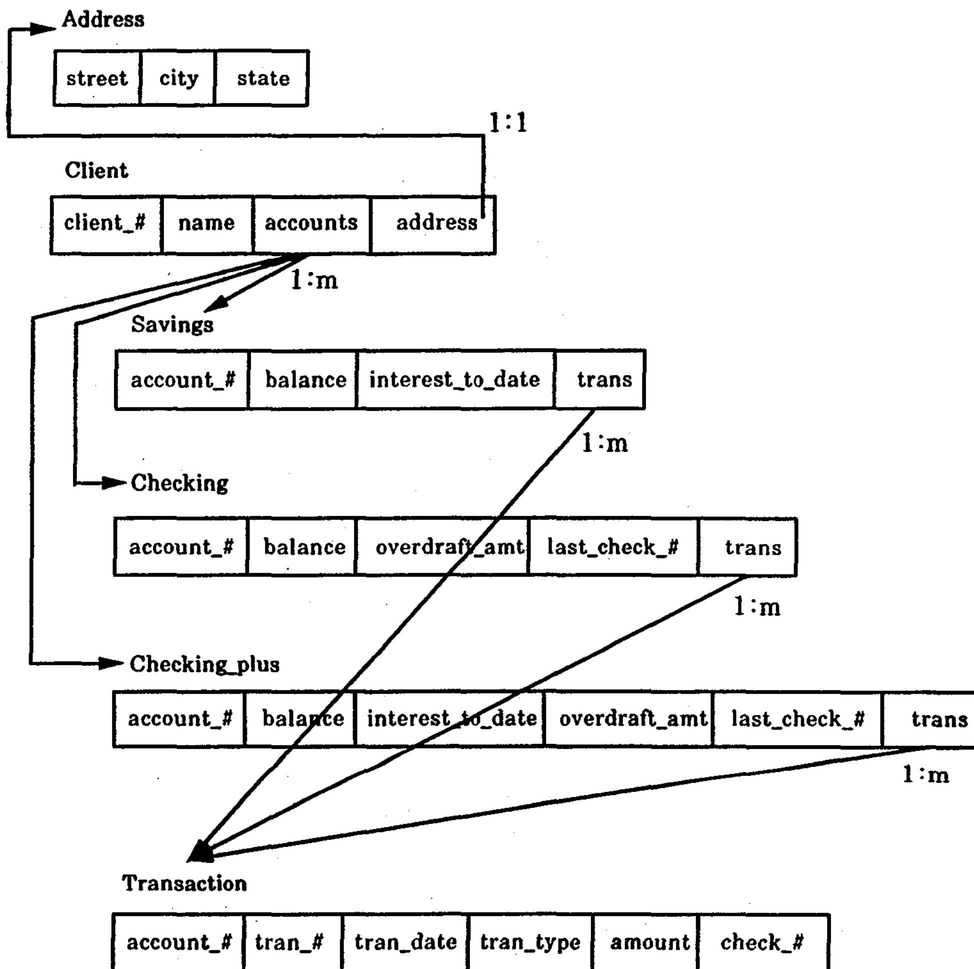

Fig. 6.

The corresponding schema modifications are:

```
drop table Client_account;

create table Client
(       client_# integer,
        name string,
        address Address,
        accounts set (Savings, Checkng,
        Checking_plus)                  );

create class Savings
(       account_# integer,
        balance monetary,
        transactions set(Transaction),
        interest_to_date monetary      );

create class Checking
(       account_# integer,
        balance monetary,
        transactions set(Transaction),
```

```
        overdraft_amt monetary,
        last_check_# integer          );

    create class Checking_plus
    (   account_# integer,
        balance monetary,
        transactions set(Transaction),
        interest_to_date monetary,
        overdraft_amt monetary,
        last_check_# integer          );
```

3.3 Extension Ⅲ : Tables can have procedures

This extension allows us to encapsulate behavior along with structure of a table by registering one or more procedures (that is, methods) as part of the table definition. Therefor, we define a method called Open_account for the Client table; this method creates a new account for a client. Also, we define three methods called Deposit, Transfer and Withdraw for each kind of account. The fundamental operations of depositing money to an account, transferring money from one account to another, and withdrawing cash from an account are the same for each kind of account, even though each may have different constraints. For example, the minimum balance may be $500 for opening a checking account, $2000 for savings, and $2500 for checking plus. This scenario is shown in the figure below.

The schema is modified as follows:

```
create table Client
(   client_# integer,
    name string,
    address Address,
    accounts set(Savings, Checkings,
    Checking_plus)                )
method
    open_account(string, monetary)
    function open_account, add file 'client_methods.o';

create class Savings
(   account_# integer,
    balance monetary,
    transactions set(Transaction),
    interest_to_date monetary     )
method
    deposit(monetary) function deposit,
    withdraw(monetary) function withdraw,
    transfer(integer, integer, monetary) function transfer,
    add file 'account_methods.o';
```

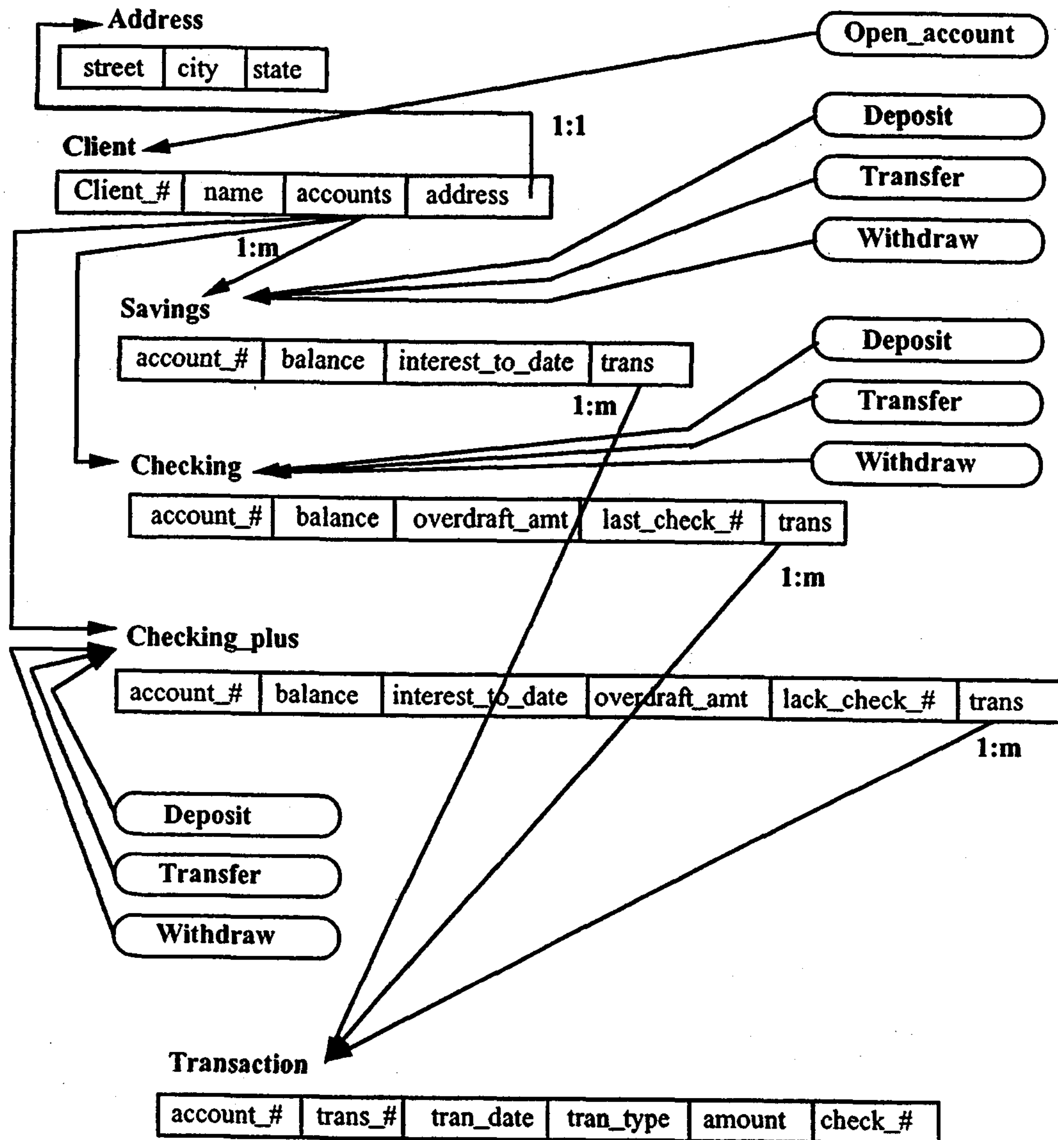

Fig. 7.

```
create class Checking
(     account_# integer,
      balance monetary,
      transactions set(Transaction),
      overdraft_amt monetary,
      last_check_# integer           )
method
   deposit(monetary) function deposit,
   withdraw(monetary) function withdraw,
   transfer(integer, integer, monetary)
   function transfer,
   add file 'account_methods.o';
```

```
create class Checking_plus
(       account_# integer,
        balance monetary,
        transactions set(Transaction),
        interest_to_date monetary,
        overdraft_amt monetary,
        last_check_# integer          )
method
        deposit(monetary) function deposit,
        withdraw(monetary) function withdraw,
        transfer(integer, integer, monetary) function transfer,
        add file 'account_methods.o';
```

The open_account method has two arguments: the first one denotes the kind of account, and the second one denotes the starting balance to be deposited in the new account. The method is implemented by a function called open_account which is found in a file called client_methods.o. The deposit and withdraw methods have an argument which denotes the amount of money to be deposited or withdrawn; they are implemented by functions called deposit and withdraw respectively, and reside in a file called account_methods.o. The transfer method has three arguments: the first one denotes the "from" account number, the second one denotes the "to" account number, and the third one denotes the amount of money to be transferred. The method is implemented by a function called transfer which is found in a file called account_methods.o.

3.4 Extension IV: Tables can be organized in an inheritance hierarchy.

It can be seen from the above that the Savings, Checking and Checking_plus tables share a similar structure, namely, account_# and amount attributes, and similar behavior, namely, Deposit, Transfer and Withdraw procedures. Further, there are resemblance between the three tables, which can be exploited to organize them into an inheritance hierarchy. Thus, we reorganize the above schema to create a new table called Account with child tables Savings and Checking. Further, note that a Checking_plus account allows a client to write checks (like a Checking account) and also bears interest(like a Savings account). This relationship is exploited to redefine Checking_plus as a child table of Savings and Checking, that is, is has two parent tables. This is known as multiple inheritance. The modified schema is shown below.

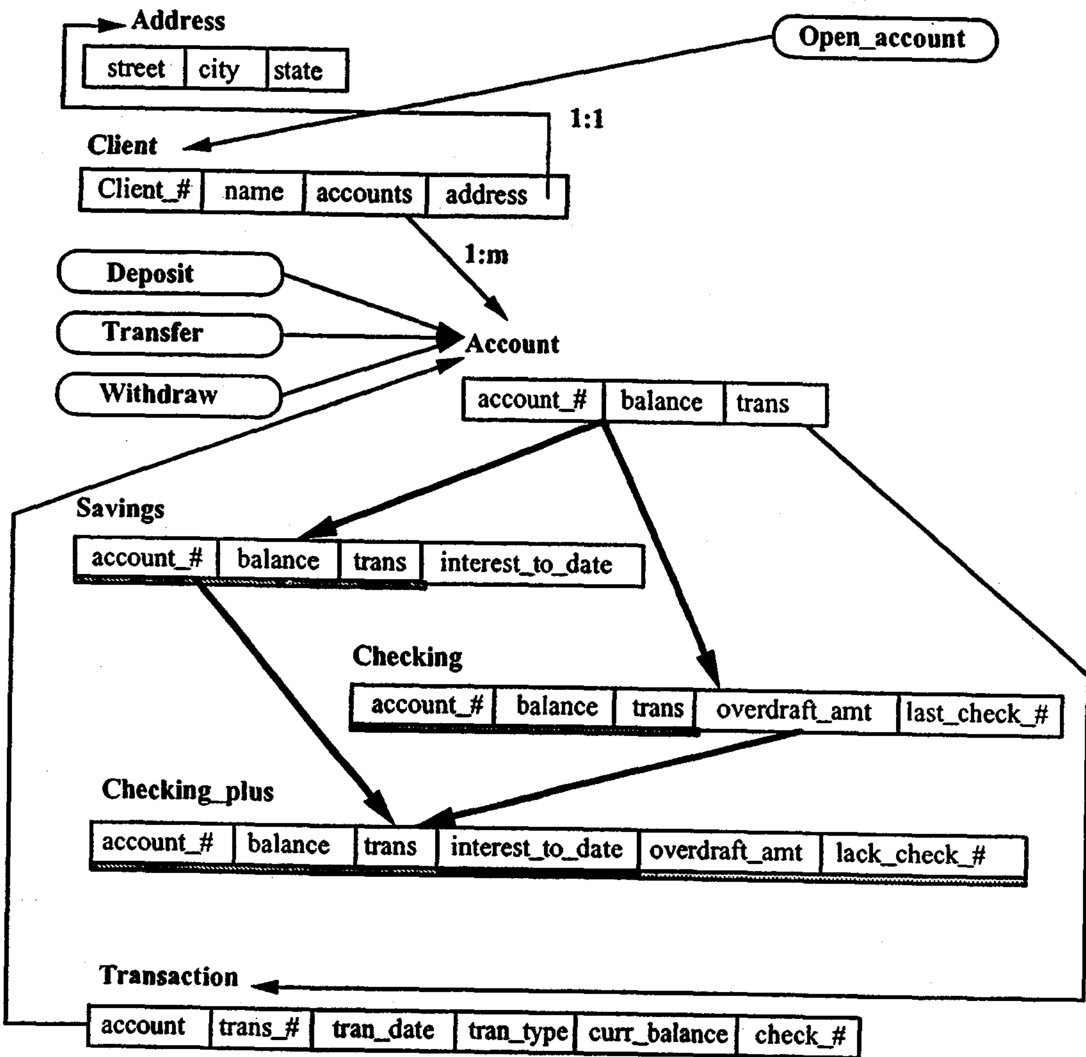

Fig. 8.

The Account table has three methods associated with it, and they are reused by its two subclasses(Savings and Checking), and their subclass(Checking_plus). Inherited attributes are underlined by thick lines.

The final and complete schema is shown below:

```
create class Address
(       street string,
        city string,
        state char(2)                    );

create class Client
(       client_# integer,
        name string,
```

```
      address Address,
      accounts set of (Account)       )
method
      open_account(string,    monetary)
      function open_account
      file 'client_methods.o';

create class Account
(     account_# integer,
      balance monetary,
      transactions set(Transaction)   )
method
      deposit(monetary) function deposit,
      withdraw(monetary)       function
      withdraw,
      transfer(integer, integer, monetary) function transfer,
      add file 'account_methods.o';

create class Savings
as subclass of Account
(     interest_to_date float          );

create class Checking
as subclass of Account
(     overdraft_amt monetary,
      last_check_# integer            );

create class Checking_plus
as subclass of Savings, Checking;

create table Transaction
(     account Account,
      account_# integer,
      tran_# integer,
      tran_date date,
      tran_type char(5),
      amount monetary,
      check_# integer                 );
```

4. Writing Object-Relational Queries

We have discussed the various extensions to the relational model and these benefits are made clear by writing some sample queries. We first retrieve all employees who earn more than their managers. The relational query is shown below:

```
select e.name, e.salary, m.name, m.salary
 from Employee e, Employee m
where e.mgr = m.ssn
  and e.salary > e.mgr.salary;
```

The corresponding object-relational query, as defined in UniSQL/X, is written as :

```
select e.name, e.salary, e.mgr.name, e.mgr.salary
 from Employee e
where e.salary > e.mgr.salary;
```

Note that no join is necessary in the latter case. The two path expression in the query, namely, e.mgr.name, e.mgr.salary, eliminate the need for a join. A path expression is a simple mechanism which provides associative access(that is, pointer chasing) to date.

As another example, we retrieve the names, account numbers and current balance for all accounts of clients living in Austin. The relational query looks like:

```
select c.name, a.account_#, s.balance
 from Client    c,    Client_account    a,
```

```
       Savings s
where c.city = 'Austin'
  and a.client_# = c.client_#
  and a.account_# = s.account_#
union
select c.name, a.account_#, k.balance
 from Client c, Client_account a, Check-
       ing k
where c.city = 'Austin'
  and a.client_# = k.client_#
  and a.account_# = s.account_#
union
select c.name, a.account_#, p.balance
 from Client c, Client_account a, Check-
       ing_plus p
where c.city = 'Austin'
  and a.client_# = k.client_#
  and a.account_# = p.account_#;
```

The corresponding object-relational query looks like:

```
select c.name, tbl.acct.account_#, tbl.
       acct.balance
 from Client c, table(c.accounts) as tbl
       (acct)
where c.address.city = 'Austin';
```

The from clause above uses derived tables (a SQL 92 construct) to coerce the accounts set-valued attribute of each Client tuple to a temporary table denoted by tbl with attribute acct(whose domain is the table Account). The select clause can now print each client's name, account number and balance; the client's name will appear for each account owned. The where clause contains a path expression: c.address.city which eliminates the need for a join. The object-relational query is not only more succinct and readable, it is also more efficient to process.

5. Conversion Guidelines

There are many ways to convert a relational schema into a corresponding object-relational schema. The final schema depends upon how much relational data and application code needs to be salvaged, While it is a relatively easier task to salvage the data, one usually needs to rewrite most of the application code. There are various techniques available for designing object-oriented systems [Rumbaugh et al. 1991, Booch 1991]. However, we describe below a few simple steps which can be used to convert a relational schema. The first step is reorganize the tables into an inheritance hierarchy. This helps in organizing and reducing the number of composition relationships to be considered. The second step is to identify one-to-one and one-to-many relationships; the third step is to identify many-to-many relationships. The fourth step is to attach methods to tables. While it is recommended that the steps be followed in the order described below, the designer can judiciously modify the order. The four steps are as follows:

1. *Identify tables in the schema that share common structure*(that is, attributes), and have similar behavior. Since there is no equivalent notion of inheritance in the relational model(and inclusion dependencies are almost never known), one must use discretion in discerning a semantic resemblance between many tables. Common structure is a simple means to identify possible inheritance relationships. When two or more tables share the same primary of candidate key attribute(s), it is very likely that they bear a semantic resemblance. Further, since attribute names are not unique across the entire schema, a large number of tables may share attributes with the same name. For example, Employee, Department and Project tables may share an attribute called manager which is not a primary key attribute in any of these three tables. On the other hand, Employee and Manager tables may share an attribute called SSN which can be either a primary or candidate key in both tables. This enables the schema designer to identify an inheritance relationship between the Employee and Manager tables. Thus limiting common structure to primary or candidate keys helps in keeping the inheritance relationships being considered to meaningful possibilities. There are two cases to be considered:

i. If there exist two(or more) tables T_i and T_j *with the same primary key attribute* (s), these tables can be redefined as subtables of a new table called T. The attributes of T include the primary key and other common attributes shared by T_i and T_j, these attributes are then removed from the definitions of T_i and T_j.

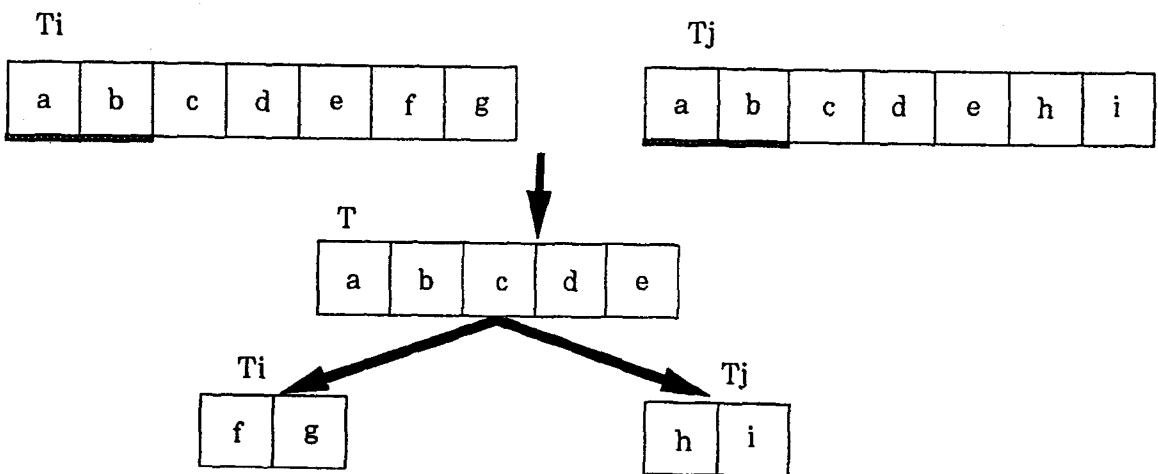

Figure 9.

ii. If there exist two (or more) tables T_i and T_j with the same primary key attribute (s) such that all attributes of T_i are contained in T_j, and that T_j has more attributes than T_i, these two tables can be reorganized such that T_j is a subtable of T_i, and common attributes including the primary key attributes are dropped from T_j.

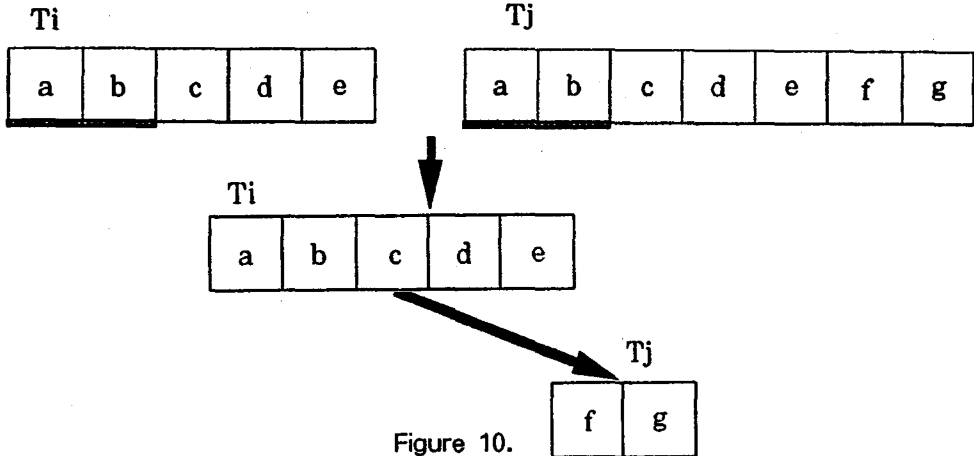

Figure 10.

2. The next step is to identify relationships by following primary key/foreign key references. The following procedure should be applied to all tables in the schema. Let K_p be the primary key (or candidate key) of a table T. Identify all tables T_i which contain attributes corresponding[1] to K_p; these attributes constitute T_i's foreign key If T_i contains foreign keys from T only, there exists relationship between the two tables. One must then determine whether it is a one-to-one or one-to-many relationship. For one-to-one relationships, there are four options:

i. Replace the foreign key attributes in Ti with a single attribute whose domain is T.

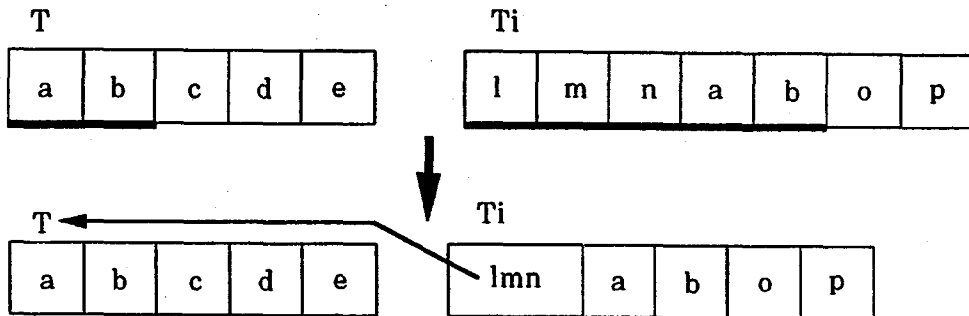

Figure 11.

1. Often, such attributes have different names in different tables; hence the correspondence must be "known" to the designer.
2. If Tj is the same as T, we have a recursive relationship -- see the Employee table in Section 1.

ii. Drop foreign key attributes in T_i, add a new attribute in T with domain T_i.

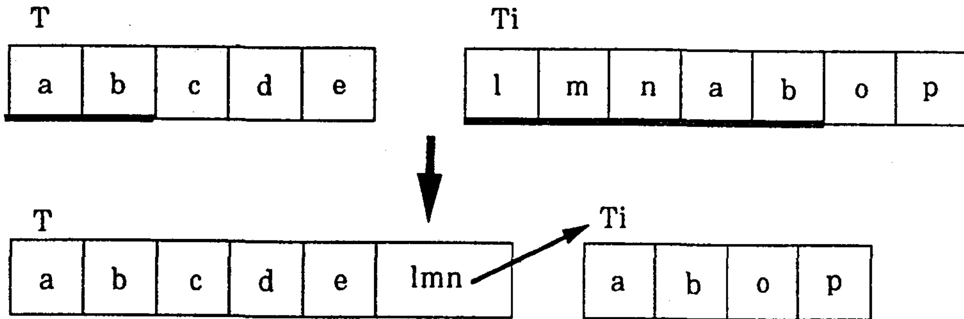

Figure 12.

iii. Replace the foreign key attributes in T_i with a single attribute whose domain is T, and add another attribute to T with domain T_i. This establishes an inverse link between T and T_i.

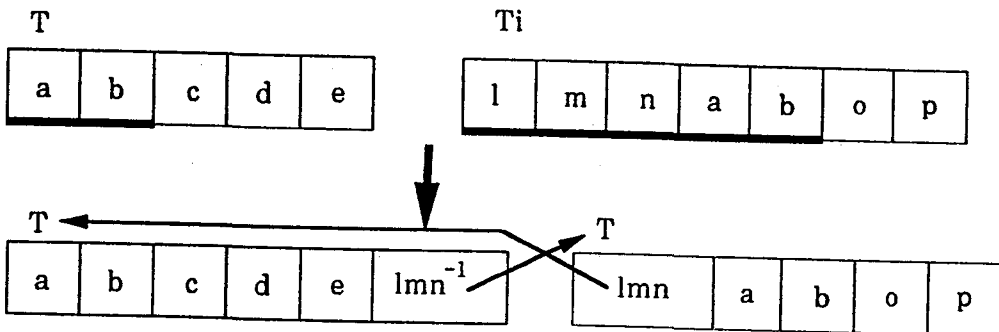

Figure 13.

iv. Retain the foreign key attributes in Ti, add another attribute to Ti with domain T, and add another attribute to T with domain Ti. In addition to establishing an inverse link, this option may be a factor if existing application code were to be reused with minimal changes -- this would be a situation-specific decision.

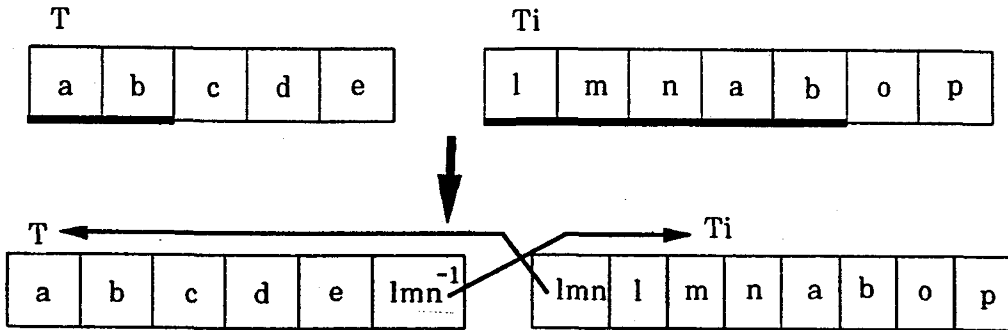

Figure 14.

The above four options for one-to-many relationships are modified as follows: if for each tuple of T, there may be zero or morerecords of Ti, the domain of the new attribute in T would be set(Ti); if for each tuple of Ti, there may be zero or more records of T, the domain of the new attribute in Ti would be set(T).

3. Discovering many-to-many relationships requires more work. If there exists a table T and two(or more) tables Ti and Tj such that T contains attributes which are the primary keys off Ti and Tj, there exists a many-to-many relationship between Ti and Tj. Two cases arise:

i. T may have additional attributes. The attributes of T corresponding to the primary key of Ti are replaced with a single attribute whose domain is Ti; the attributes of T corresponding to the primary key of Tj are replaced with a single attribute whose domain is Tj.

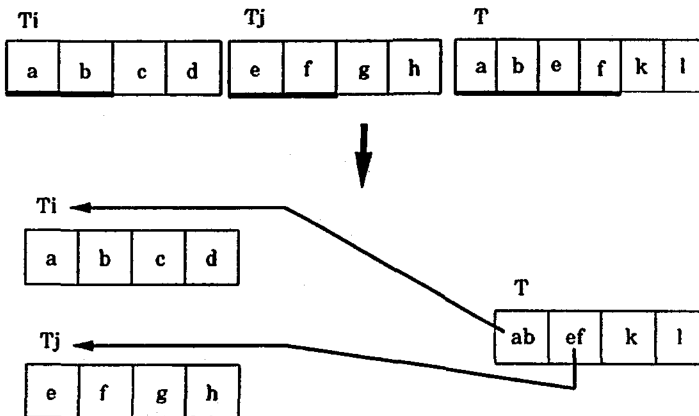

Figure 15.

ii. T may not have additional attributes. While the above solution would also work, there is a better alternative. The table T is dropped, a new attribute is added to T_i with domain set(T_j), and a new attribute is added to T_j with domain set(T_i).

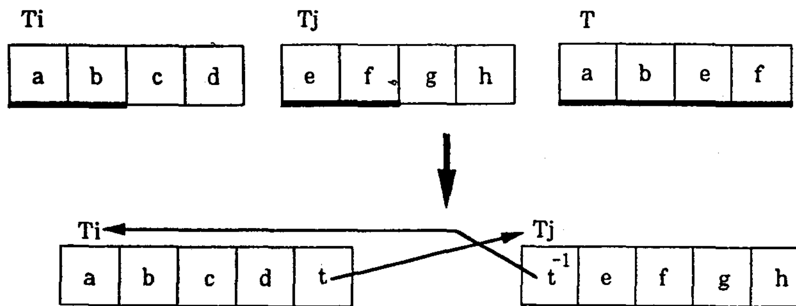

Figure 16.

4. The final step is to attach methods to various tables. Since a method is a routine that is specific to the behavior of individual tuples of the table to which it is attached, it may be possible to reuse an appropriate routine from an existing application. This decision would be situation-specific. Alternately, a new implementation for the method would be required.

Since the composition and inheritance hierarchies are two orthogonal dimensions off an object-oriented schema, care must be taken that the results of the above four steps are not mutually inconsistent. If an inconsistency is found, it must be resolved by appropriately modifying the inheritance and composition relationships between various tables, or redefining the tables.

6. Conclusions

To overcome the limitations of the relational paradigm, we have made four extensions in UniSQL/X, an object-relational data model, to the core relational model. The first extension eliminates redundant data by promoting data sharing. This reduces potential inconsistency and maintenance problems. It also eliminates the foreign key/primary key references; one object can directly point to another object. This replaces expensive joins in pointer chasing applications. The second extension eliminates the need for artificial tables necessary to capture one-to-many(or many-to-many) relationships. Such a table is usually identified by noting that its primary key contains attributes which are also foreign keys. The first two extensions together are most fruitfully employed in "parts-explosion" or "bill-of-materials" applications. The third extension promotes encapsulation of behavior, and the fourth extension promotes reuse of code and structure. We also provided some guidelines to convert a relational schema into a corresponding object-relational schema along with an illustration.

References

Booch, G., *Object-Oriented Design*, Benjamin /Cummings, 1991.

Date, C.J., *An Introduction to Database Systems*, Volume 1, Fourth Edition, Addison Wesley, 1986.

Elmasri, R. and S.B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings, Redwood City, CA, 1989.

Kim, W., *Introduction to Object-Oriented Databases*, MIT Press, 1991.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

Ullman, J., *Principles of Database Systems*, Second Edition, Computer Science Press, 1982.