Journal of Network and Computer Applications (1999) **22**, 215–232
Article No. jnca.1999.0096, available online at http://www.idealibrary.com on IDEAL®

Micro: A normalization tool for relational database designers

Hongbo Du and Laurent Wery

Department of Computer Science, University of Buckingham, Buckingham MK18 1EG, UK.

(Received 1 December 1998; accepted 13 August 1999)

Normalization is a major task in relational database design. Although normalization algorithms have been developed, very few commercial design tools are available to assist the normalization satisfactorily. In this paper, we present a prototype system Micro for automatic normalization. We have developed a simple algorithm for 2NF normalization, and used the abstract algorithms reported in the literature for 3NF and BCNF normalization. We employ efficient data structures on functional dependencies and relation schemes to improve the performance of these algorithms. Micro enforces a certain fixed order among functional dependencies to deal with the non-deterministic feature that is associated with the original algorithms. Micro provides a windowing user interface through which the database designer can specify functional dependencies easily and generate real normalized tables. Through Micro, we wish to demonstrate that the automation of normalization is practical. © 1999 Academic Press

1. Introduction

Most modern database management systems have improved the usability of their database implementation facilities. The creation of a database is becoming an easy task: tables, queries, forms and reports can be designed interactively by drag-and-drop movements of the pointing device. Many novice end users can therefore become proficient in constructing a simple database with a minimal amount of training. However, database implementation is only a small integral step in the whole database life cycle. The conceptual design of a database takes much more effort than the implementation.

In the conceptual design of a database, the database designers normally use high level conceptual models, such as the Entity-Relationship model, to analyse the structure of data and describe the conceptual schema of the database. Unfortunately, most commercial DBMS packages do not directly support high level conceptual models such as the ER model. Instead, these systems support record-based models such as the relational data model. This requires that the design in a high level conceptual model must be transformed to a design in a record-based model.

For the relational data model, a formal process called normalization is undertaken during this transformation. Relation schemes are *normalized* in the sense that attributes are grouped into relation schemes according to dependencies among the attributes clearly specified. Without proper normalization, instance tables of a relation scheme may contain redundant information. Tables with redundant information not only waste storage space, but also make it difficult for the database operators to maintain the database. This problem is known as updating anomalies [1] (pp. 395–397). Normalization is hence considered as an important step in the design of a relational database.

In the current practice of database design, normalization is largely carried out manually. Manual process of normalization can be demanding and prone to errors, particularly when the number of attributes is large and dependencies among attributes are complex. It is therefore desirable to develop tools that automate the normalization process. Abstract algorithms for normalization have been developed as a result of extensive studies ([1] (pp. 423–441), [2] (pp. 213–250)). However, convenient tools for automatic normalization are yet to feature in modern commercial DBMS products.

We have developed a prototype automatic normalization system named Micro to assist database designers in designing relational databases. It is based on the original abstract algorithms reported in the literature and a new algorithm for normalizing tables to 2^{nd} normal form. Our work is motivated mainly by the practical aspect and the potential application of such a system. Efficiency is therefore not the centre of the concern. In fact, the input space, i.e. the number of attributes in a database and the number of user specified functional dependencies, is normally small (within the range of hundreds). The process of normalization, which starts from functional dependency specification and ends with normal form relation schemes, is often a *one-pass* process rather than an interactive and iterative procedure. We do improve the efficiency of the original algorithms to a certain extent by using suitable data structures to represent functional dependencies and relation schemes. In order to deal with the non-determinism that is associated with the original abstract algorithms, Micro enforces a fixed order among functional dependencies. Micro provides a window-based environment where the database designer defines and edits functional dependencies among attributes easily and the system generates relation schemes in 2^{nd}, 3^{rd} or BCNF normal forms. In order for the system to be useful, Micro allows the database designer to specify the data type for each attribute of a scheme and create the table structure in Microsoft Access format. We have tested Micro on an ordinary PC computer (200 MHz and 32 MB RAM) with a number of real world database examples and a number of imaginative examples with a large number of attributes and complex functional dependencies. The performance is satisfactory.

Our work is related, yet different from existing works on efficient practical algorithms for testing normal form relations [3]. These works concentrate on discovering dependencies among attributes, testing prime attributes and consequently normalizing relations with respect to instance tables. We are interested in given functional dependencies from the database designer and algorithms to normalize relations according to the given functional dependencies.

The rest of the paper is organized as follows. Section 2 briefly describes some basic concepts of normalization. The original abstract algorithms are omitted from this paper due to extensive coverage in the literature ([1] (pp. 423–441), [2] (pp. 213–250)). Section 3 presents suitable data structures and our detailed designs of the algorithms and our own algorithm for 2NF normalization. The use of ordering in solving the non-determinism is also discussed. Section 4 gives an overview of the Micro system and its user interface. We use an example to demonstrate the effectiveness of Micro in automating the normalization.

2. Functional dependencies and normalization

2.1 *Functional dependencies*

Due to the extensive study of the subject, we assume that the reader is familiar with the concept of the relational data model. We briefly describe some basic concepts for the reader's reference.

2.1.1 *Functional dependency*. A relation scheme R, denoted by $R(A_1, A_2, \ldots, A_n)$, is a sequence of attributes $A_1, A_2, \ldots, A_n$. Let r be any instance table of R. Let $t[A_i](1 \leq i \leq n)$ denote the projection of tuple t $(t \in r)$ on attribute A_i. Also let X and Y be any arbitrary subsets of R. We also use t' to represent a tuple of table r. Y is said to be *functionally dependent* on X, expressed by $X \rightarrow Y$, if and only if for any two tuples, t and t' of r, such that $t[X]=t'[X]$, we must also have $t[Y]=t'[Y]$. X is called the *determinant*, and Y the *dependent* of the dependency. Any instance table r of R is a *legal extension* of R provided it satisfies all functional dependencies on R.

2.1.2 *Closure of functional dependencies*. Let F be a set of functional dependencies on R. A function dependency $X \rightarrow Y$ can be inferred from F, denoted by $F| = X \rightarrow Y$, if it holds for every instance r of R that satisfies all dependencies in F. The *closure* of F, denoted by F^+, is a set of all functional dependencies inferred from F. The inference of functional dependencies is governed by the Armstrong's rules of inference [1] (pp. 403–405).

The *closure of attribute set X of R under F*, denoted by X^+, is a set of all attributes of R that are dependent on X in F. Calculating X^+ is a frequent operation in the normalization algorithms. A straightforward method to calculate X^+ is given in [1] (p. 405). A modified version is given in Section 3.2.

2.1.3 *Minimal cover of functional dependencies*. Two sets of functional dependencies E and F are *equivalent* if $E^+=F^+$. A set of functional dependencies F is *minimal* if

(a) for every $f \in F$, the dependent in f has only one attribute;
(b) for no $X \rightarrow Y$ in F is the set $F-\{X \rightarrow Y\}$ equivalent to F;
(c) for no $X \rightarrow Y$ in F and $Z \subset X$ is $F-\{X \rightarrow Y\} \cup \{Z \rightarrow Y\}$ equivalent to F.

A *minimal cover* of F is a minimal set F_{min} such that F_{min} is equivalent to F. All algorithms for normalization need to calculate a minimal cover of a given set of functional dependencies. An algorithm for calculating the minimal cover is given in [1] (p. 426).

Depending on the order of functional dependencies in F, there can be more than one minimal cover for F (see the example on pp. 224–225 of [2]). In practice, if the end user specifies the same set of functional dependencies in different orders, the algorithms for normalization can yield different sets of normalized relations. This non-deterministic feature must be dealt with in practical systems.

Figure 1 shows the attributes and functional dependencies of an example taken from [4] (p. 317). In this database, `Jobtitle` is functionally dependent on `Date` and `Emp#`, and `Area` is transitively dependent on `Emp#`. The closure of {`Office#`} under F is {`Area, Dbudget, Dept#, Mgr#, Office#`}.

2.2 *Normalization*

The normalization process is to decompose a given relation scheme to a set of smaller relation schemes. It is a repeated process in which a decomposed relation scheme may be further decomposed. Normally, the decomposition starts from a universal relation scheme that includes all possible attributes and all functional dependencies. The universal table also enables thorough study of functional dependencies. According to the constraints on the functional dependencies, the universal relation can be decomposed into 2NF relation schemes where non-prime attributes are fully dependent on the primary key, 3NF relation schemes where no transitive dependencies exist among non-prime attributes or BCNF schemes where all determinants must be at least a candidate key for the relation [4] (pp. 410–416). Other dependencies among attributes among attributes (such as multi-valued dependencies) are possible. There are higher order normal forms

Attribute Names	*Functional Dependencies (F)*	*Minimal Cover of F*
`Area`	`Date, Emp#`→`Jobtitle, Salary`	`Data, Emp#`→`Jobtitle`
`Date`	`Dept#`→`DBudget, Mgr#`	`Date, Emp#`→`Salary`
`Dbudget`	`Emp#`→`Dept#, Office#, Phone#, Proj#`	`Dept#`→`Dbudget`
`Dept#`	`Mgr#`→`Dept#`	`Dept#`→`Mgr#`
`Emp#`	`Office#`→`Dept#, Area`	`Mgr#`→`Dept#`
`Jobtitle`	`Phone#`→`Office#`	`Emp#`→`Phone#,`
`Mgr#`	`Proj#`→`Pbudget, Dept#`	`Emp#`→`Proj#`
`Office#`		`Phone#`→`Office#`
`Pbudget`		`Proj#`→`Dept#`
`Phone#`		`Proj#`→`Pbudget`
`Proj#`		`Office#`→`Dept#`
`Salary`		`Office#`→`Area`

Figure 1. Functional dependencies in an example database.

(e.g. 4NF, 5NF, etc.). However, these higher normal forms deal with practical situations that are very rare [5] (p. 193). This is why Micro only deal with functional dependencies and normal forms up to BCNF.

Let R be a relation scheme before decomposition, and $R_1, R_2, \ldots, R_m$ be the decomposed schemes from R. Let F be the set of functional dependencies on R and $\pi_F(R_i)$ be the projection of F on $R_i (1 \leq i \leq n)$, i.e. $\pi_F(R_i) = \{X \rightarrow Y \mid (X \rightarrow Y) \in F^+$ and $(X \cup Y) \subseteq R_i\}$. During the normalization, the following three properties must hold:

(a) Attribute preservation, i.e. $R_1 \cup R_2 \cup \ldots \cup R_m = R$;
(b) Functional dependency preservation, i.e. $(\pi_F(R_1) \cup \ldots \cup \pi_F(R_m))^+ = F^+$;
(c) Lossless join property, i.e. for every instance table r of R, $\pi_{R_1}(r) \infty \ldots \infty \pi_{R_m}(r) = r$ where ∞ denotes the natural join operation.

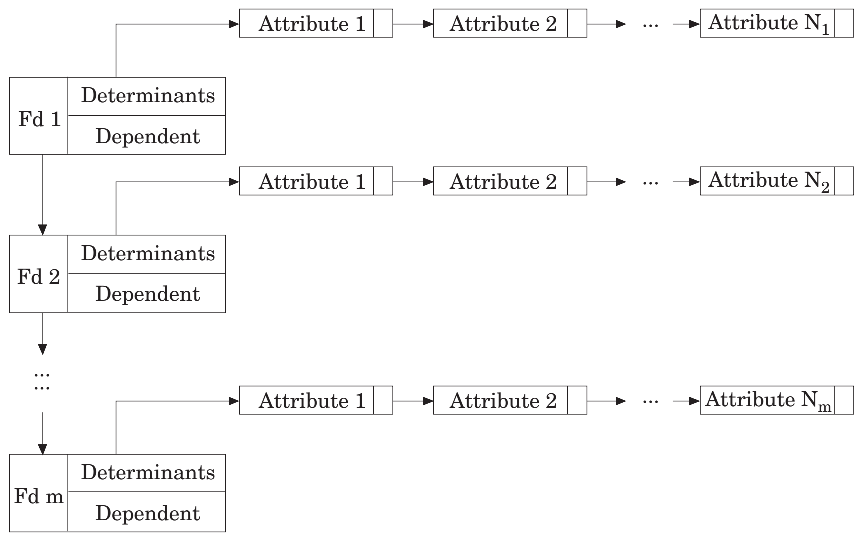
3. Normalization algorithms in micro

3.1 *Data structures*

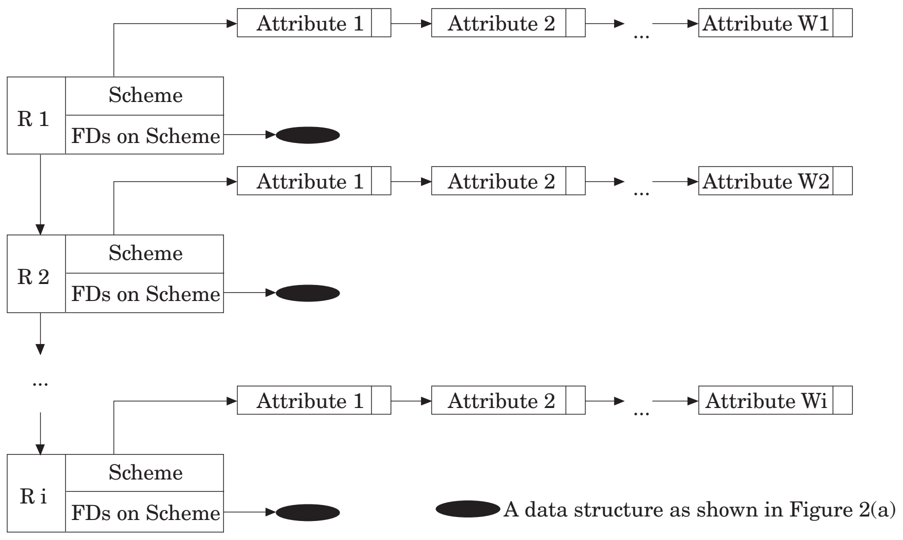
The computation performed by the normalization algorithms requires dynamic manipulations of sets of data, such as sets of attributes, sets of functional dependencies and sets of decomposed relation schemes. The results of the manipulations can be of any length. It is therefore necessary that attributes and functional dependencies are stored in flexible and dynamic data structures. In order to avoid the non-determinism of solutions, attribute sets and function dependency sets are implemented as sorted lists so that there can only be one representation for the same set.

An attribute set is a sorted list of comparable attribute names (i.e. strings) $a_1, a_2, \ldots, a_n$ where $a_i < a_j (1 \leq i < j \leq n)$. A set of functional dependencies is a list of pairs. Each pair consists of the determinant and dependent of a dependency. The determinant is an attribute set. The dependent is a single attribute so that the algorithm for calculating a minimal cover can be simplified. Elements in a set of functional dependencies are sorted according to the attribute names of the determinant. For the functional dependencies that have the same determinant, the dependencies are sorted according to the alphabetic order on the attribute names of the dependents. A set of decomposed relations is a list of pairs. Each pair consists of a decomposed scheme (i.e. an attribute set) and a set of functional dependencies projected on the scheme. The purpose to hold the functional dependencies projected on a relation scheme together with the decomposed relation scheme is to enable further normalization (e.g. from 3NF to BCNF).

As shown in Fig. 2(a), we use a pointer-based linear linked list structure to implement an attribute set. Each node of the list contains the name of an attribute and a reference pointer to the next node. For control purposes of the algorithms, each attribute has an additional Boolean property of being 'marked' or 'unmarked'. An operation is responsible for the setting of the property. The



(a)



(b)

Figure 2. Data structures for normalization. (a) The data structure for a set of functional dependencies; (b) The data structure for decomposed relations.

functional dependency set is implemented with a pointer-based multiple linked list structure. Each node, designated for one functional dependency, contains a unique dependency identity, a pointer to a linked list for all determinant attributes and a field for the name of the dependent attribute. Each node also has a reference pointer link to the next dependency node in the list.

As illustrated in Fig. 2(b), a pointer-based multiple linked list structure is used to implement a set of decomposed relation schemes. Each node of the linked list contains a unique scheme identity, a pointer to a linked list of the attribute names for the scheme and a pointer reference to the multiple linked list structure for a set of functional dependencies as shown in Fig. 2(a). Each attribute node in the relation scheme list has an additional boolean property, indicating if the attribute is a prime attribute for the key of the relation scheme.

Our normalization algorithms are built on the basis of a generic sorted list abstract data type. This ADT is applicable to both data structures listed above and the attribute set. The operations of the ADT, which are used later to describe the algorithms, are given below.

- subset(X, Y)—a Boolean operation to test if $X \subset Y$;
- member(a, X)—a Boolean operation to test if $a \in X$;
- size(X)—an operation to calculate and return the size of list X;
- insert(a, X)—an operation to insert a into list X. After the insertion, $a \in X$ and the order in X must be preserved;
- remove(a, X)—an operation to remove a from list X. After the removal, $a \notin X$ and the order in X must be preserved;
- copySet(X)—an operation to return a duplicate of list X;
- union(X, Y)—an operation to perform the set union on lists X and Y and return the result;
- differ(X, Y)—an operation to perform the set difference from X to Y and return the result;
- first(X)—an operation to return the first element of list X;
- next(X)—an operation to return the next element of list X from a given current position;
- end(X)—an operation to return the end flag of list X.

3.2 *Normalization algorithms*

In this section, the algorithms for normalization are presented in a Pascal-like pseudo code. The original abstract algorithms use intensively set operations such as set union and difference. Such operations can be costly to perform. To reduce the cost, we minimize the use of these operations in our normalization algorithms. At the same time, for those parts of the algorithms where set operations are essential, we exploit the situation where sets are implemented as sorted lists and use the merge method to implement the set operations. The time complexity becomes O(N).

3.2.1 *Find the closure of X under F*. The algorithm for calculating the closure of attribute set X under functional dependency set F is presented as follows.

```
algorithm closure(X: attribute set; F: functional dependency set):
         attribute set;
begin
    tempX:=copySet(X);
    repeat
      oldX:=copySet(tempX);
      f:=first(F);
      while f≠end(F) do
      begin
        if subset(f.determinant, tempX) then tempX:=union({f.
             dependent}, tempX);
        f:=next(F)
      end;
    until size(oldX)=size(tempX);
    return tempX
end;
```

The main modification to the original algorithm is to replace condition oldX= tempX by the condition size(oldX)=size(tempX) in the **until** statement. It is obvious from the algorithm that the condition size(oldX)=size(tempX) is semantically equivalent to the condition oldX=tempX. The condition, size(oldX)= size(tempX), however, takes only O(n) time. This is because calculating the sizes of oldX and tempX takes O(n) time when the linear search method is used on the linked lists.

3.2.2 *Find the minimal cover of F*. The algorithm for calculating the minimal cover of a functional dependency set F is presented below. The first stage of the original algorithm is removed since the data structure for functional dependency has only one dependent. To avoid too much intermediate results, we use a strategy of removing an element from a list and inserting it back in the list when needed. This method simplifies certain operations of the algorithm.

```
Algorithm minimalCover(F: functional dependency set): functional
         dependency set;
begin
    f:=first(F);
    while f≠end(F) do begin (*remove transitive and redundant dependencies*)
    remove(f, F);
    cl:=closure(f.determinant, F);
    if not member(f.dependent, cl) then insert(f, F);
    f:=next(F)
    end;

    f:=first(F);
    while f≠end(F) do begin  (*remove redundancy in determinants*)
      a:=first(f.determinant);
      while a≠end(f.determinant) do
```

```
        begin
          temp:=f; remove(f, F);
          remove(a, temp.determinant);
          attSet:=closure(temp.determinant, F);
          if member (a, attSet) then f:=temp;
          insert(f,F);
          a:=next(f.determinant)
        end;
        f:=next(F)
      end;
      return F
    end;
```

3.2.3 *Decomposition algorithms*. The algorithm for decomposing a relation scheme into a set of 3NF schemes with dependency preservation and lossless join properties is presented below. The algorithm finds all functional dependencies that have the same determinant and construct a relation scheme using the determinant as the key and all dependants as non-key attributes. To preserve the attributes, a default table, which includes all unused attributes, is created.

```
Algorithm 3NF(R: attribute set; F: functional dependency set): decomposed
          relation set;
begin
    G:=minimalCover(F); Result:=∅;
    f:=first(G);
    while f≠end(G) do (*find functional dependencies with the same
     determinant*)
    begin
      r:=∅;
      r:=union(f.determinant, r); r:=union(f.dependent, r);
      ff:=next(G);
      while (f.determinant=ff.determinant) and ff≠end(G) do
      begin
        r:=union(ff.dependent, r);
        ff:=next(G);
      end;
      mark f.determinant as the key;
      insert(r, Result);
      f:=ff
    end;
    place all unplaced attributes in Xp;
    insert(Xp, Result);
    return Result;
end;
```

The algorithm for decomposing a 3NF relation scheme into a set of BCNF schemes with the lossless join properties is presented below. The Boolean function violate BCNF(f, r) checks if the functional dependency f violates the constraint of BCNF upon the relation r. The method used is to calculate f.determinant$^+$, and then test if f.determinant$^+$ includes all attributes from relation r. If it does, then the relation r does not violate the BCNF constraints; otherwise, it violates the BCNF constraints.

```
Algorithm BCNF(R: attribute set; F: functional dependency set):
          decomposed relation set;
(*Constraints: R is already in 3NF*)
begin
    D:={R}; P:=∅;
    while (D≠∅) do begin
      remove (r,D); f:=first(F); found:=false;
      while f≠end(F) and not found do (*find a dependency that violate
          BCNF constraints*)
        if violatesBCNF(f, r) then found:=true
        else f:=next(F);
      if found then
        begin
          insert(differ(r, {f.dependent}), D);
          insert(union(f.determinant, {f.dependent}), D);
        end;
      else
        insert(r, P);
    end;
    return P;
end;
```

The algorithm for decomposing a relation scheme into a set of 2NF schemes with the attribute preservation property is presented below. We define a function, *fullClosure(X, F)*, which returns all attributes that are *fully* dependent on X with repsect to dependencies in F.

```
algorithm fullClosure(X: attribute set; F: functional dependency set):
         attribute set;
begin
    tempX:=copySet(X);
    repeat
      oldX:=copySet(tempX);
      f:=first(F);
      while f≠end(F) do begin
        if subset(f.determinant, tempX) then
          if not subset(f.determinant, X) then tempX:=union({f.
          dependent}, tempX)
          else if f.determinant=X then tempX:=union({f.dependent},
          tempX);
        f:=next(F)
      end;
    until size(oldX)=size(tempX);
    return tempX
end;
```

This function is very similar to the *closure* function. However, the closure function returns an attribute set in which there can be partial dependencies among the attributes.

The algorithm also makes use of a procedure called mark(X, Y). This procedure sets the MARK properties of those attributes in attribute set Y that are listed in X to ‘marked.’

```
Algorithm 2NF(R: attribute set; F: functional dependency set): decomposed
          relation set;
begin
    P:=∅;
    G:=minimalCover(F); (*remove redundant functional dependencies in F*)
    f:=first(G);
    while f≠end(G) do begin (*find functional dependencies with the same
        determinant*)
      XP:=fullClosure(f.determinant,G);
      mark(XP,R);
      insert(XP,P);
      repeat ff:=f; f:=next(G);
      until ff.determinant≠f.determinant
    end;
    XP:=∅;
    for all attributes in R do
      if attribute is not marked then insert(attribute, XP);
    if XP≠∅ then insert(XP,P);
    return(P);
end;
```

4. Overview of Micro system

4.1 *System architecture*

Micro consists of an user interface, a dynamically linked library for normalization algorithms, a context sensitive help facility and a data gateway to Microsoft Access DBMS. The conceptual model of the system is shown in Fig. 3. The use of dynamically linked library for normalization algorithms makes the system modular and portable. It is therefore convenient to further enhance the system’s functionality.

The interface of Micro offers a window-based menu driven style of interaction. The design of the interface closely follows the Microsoft’s HCI guidelines on user control, clarity and directness [6]. The main window and various menu options are illustrated in Fig. 4.

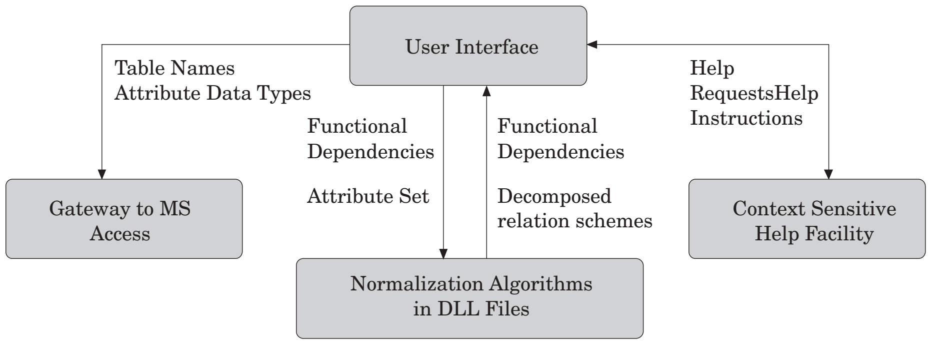

User Interface
Table Names
Attribute Data Types
Functional Dependencies
Attribute Set
Functional Dependencies
Decomposed relation schemes
Help RequestsHelp Instructions
Gateway to MS Access
Normalization Algorithms in DLL Files
Context Sensitive Help Facility

Figure 3. Conceptual view of Micro.

The process of normalization in Micro follows four main steps. The first step is to name attributes in the new database (Fig. 4(a)). The second step is to specify functional dependencies among the attributes (Fig. 4(b)). Editing facilities are available in these two steps so that the database designer can change the attribute names and modify functional dependencies. The third step is to choose a normal form and execute the normalization procedure (Fig. 4(c)). The final step is to define data types for the attributes in the normalized relations and create the actual table structure in Microsoft Access format (Fig. 4(d)).

4.2 *An example*

We now use the example in Section 2.2 to illustrate the use of Micro. First, the user starts by selecting the `New Database` option from the `File` pull-down menu. In the dialogue box as show in Fig. 5, the user enters the names of attributes, one at a time, in the top input box. Any unwanted attribute can be deleted by highlighting the attribute in the attribute list and pressing the `Delete` button. Eventually the user obtains a final list of attributes. The attribute list can also be saved, and modified later on. This is done by selecting the `Save` and `Open` options from the `File` pull-down menu.

The user then selects the `New Functional Dependency` option from the `Dependency` pull-down menu. A dialogue box as shown in Fig. 6 is displayed. This dialogue box consists of a top display panel, two lists of attributes on the left and right sides and a `Determines` button at the centre. Using the pointing device, the user highlights attributes that would be determinants from the left attribute list and attributes that would be dependents from the right attribute list, and then presses the `Determines` button. The functional dependency is then displayed in the top panel in $X \rightarrow Y$ format. The user can delete a functional dependency from the display list by highlighting the dependency in the top panel and pressing the `Delete` button. Like attributes, functional dependencies can also be saved, and then later on loaded and modified.

After completing the specification of the functional dependencies, the user selects a desirable normal form option (2nd Normal Form, 3rd Normal Form, or

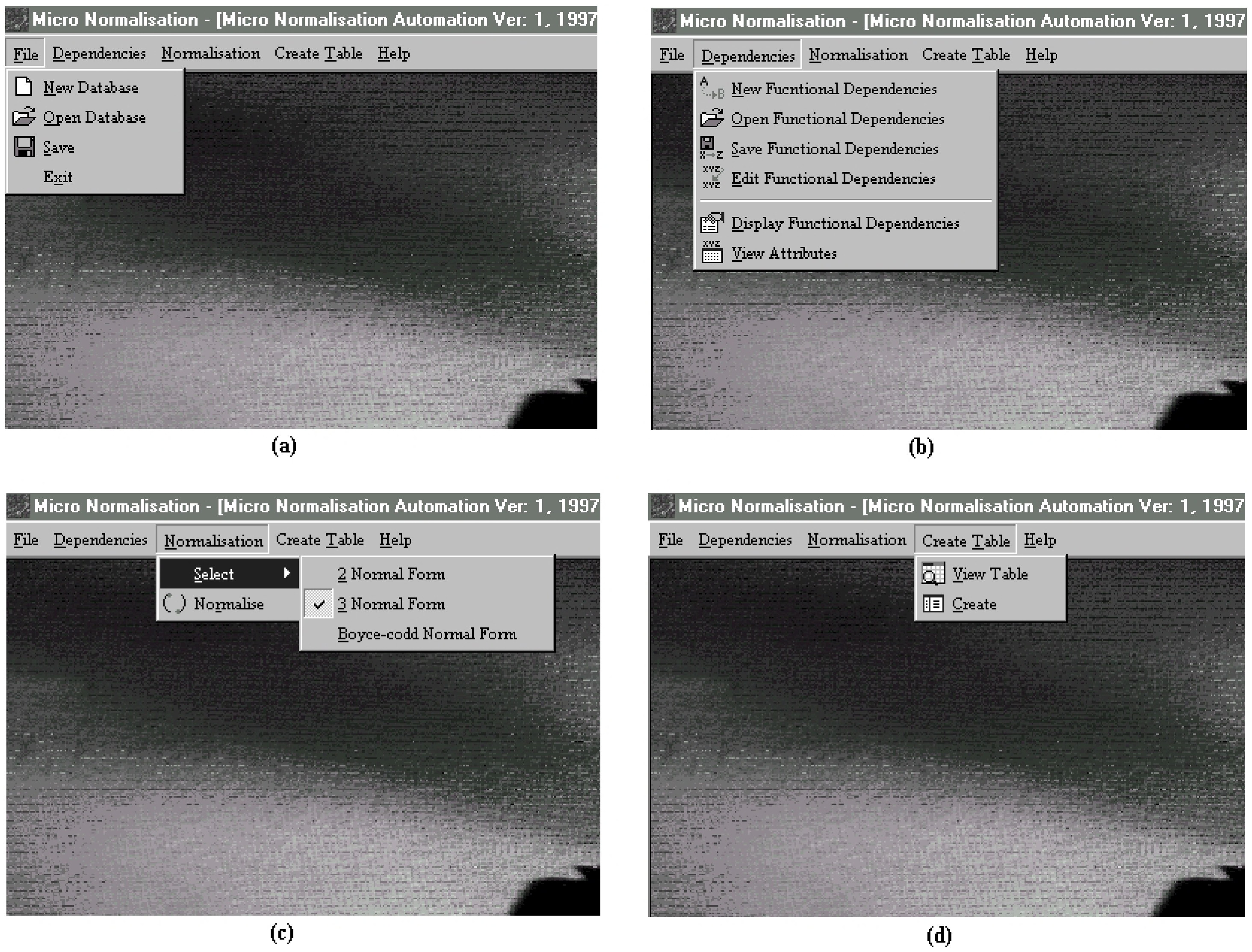

Micro Normalisation - [Micro Normalisation Automation Ver: 1, 1997
File Dependencies Normalisation Create Table Help
New Database
Open Database
Save
Exit
(a)
New Fucntional Dependencies
Open Functional Dependencies
Save Functional Dependencies
Edit Functional Dependencies
Display Functional Dependencies
View Attributes
(b)
Select
Normalise
2 Normal Form
3 Normal Form
Boyce-codd Normal Form
(c)
View Table
Create
(d)

Figure 4. The main window and menu options of Micro.

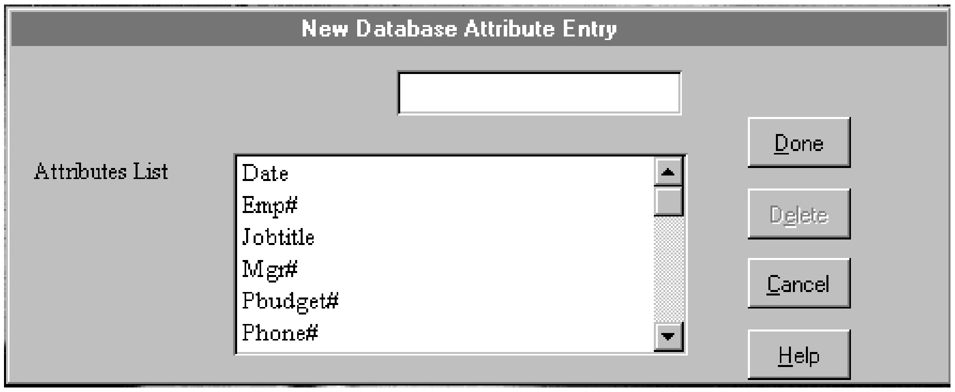
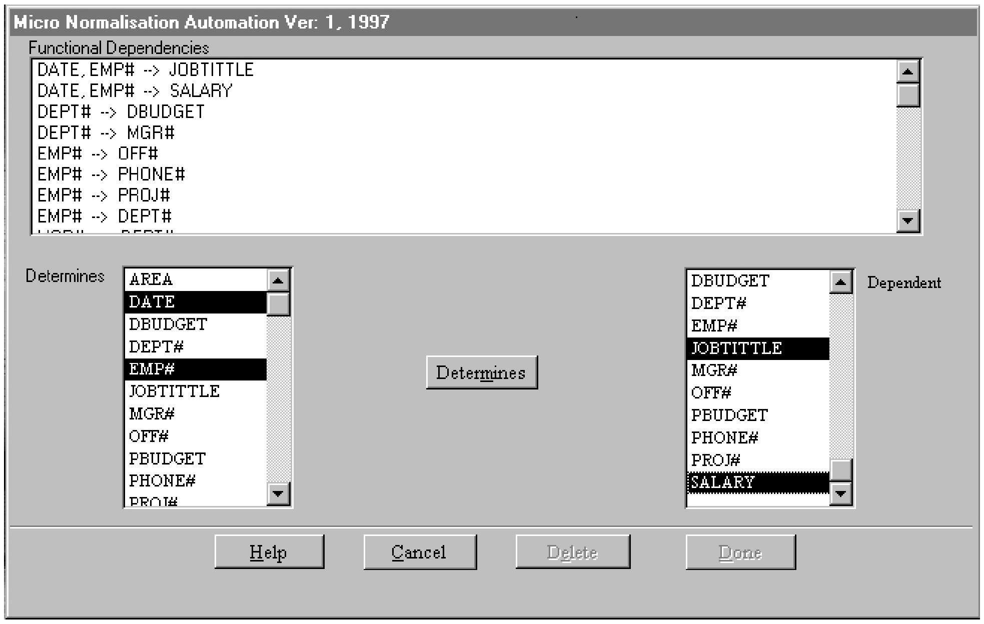


Figure 5. Attribute specification.

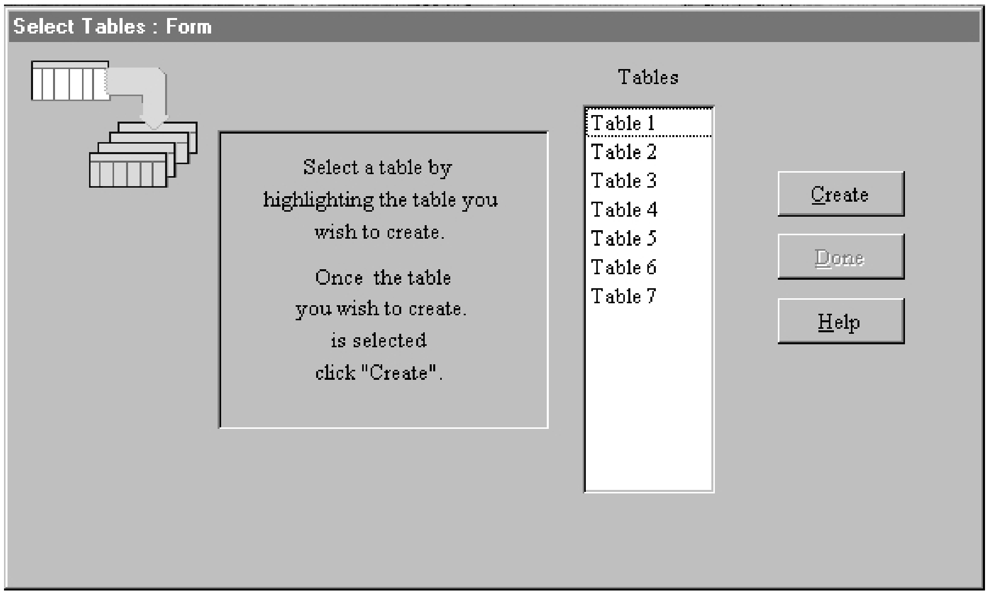


```
DATE, EMP# --> JOBTITTLE
DATE, EMP# --> SALARY
DEPT# --> DBUDGET
DEPT# --> MGR#
EMP# --> OFF#
EMP# --> PHONE#
EMP# --> PROJ#
EMP# --> DEPT#
```

Figure 6. Functional dependency specification and editing window.

BCNF) from the `Normalization` pull-down menu of the main frame, and starts the process of normalization. As a result, a list of normalized relation schemes with default names are created.

Finally, the user can select `View Tables` or `Create Tables` option from the `Table` pull-down menu. The `View Tables` option allows the user to view the normalized relation schemes by selecting any of them from a list box. The `Create Tables` option allows the user to choose relation schemes upon which an MS Access table is to be created. Figure 7 shows the dialogue box that enables the selection.


Select Tables : Form

Select a table by highlighting the table you wish to create.

Once the table you wish to create. is selected click "Create".

Figure 7. Selecting tables to be created in MS Access format.

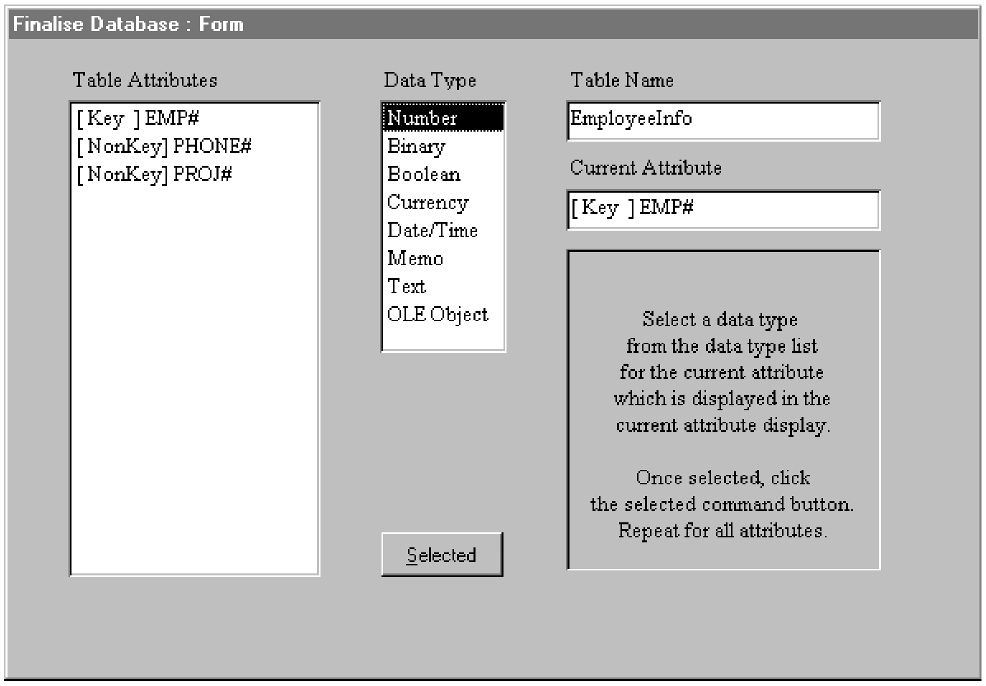

Finalise Database : Form

Table Attributes
[Key] EMP#
[NonKey] PHONE#
[NonKey] PROJ#

Data Type: Number

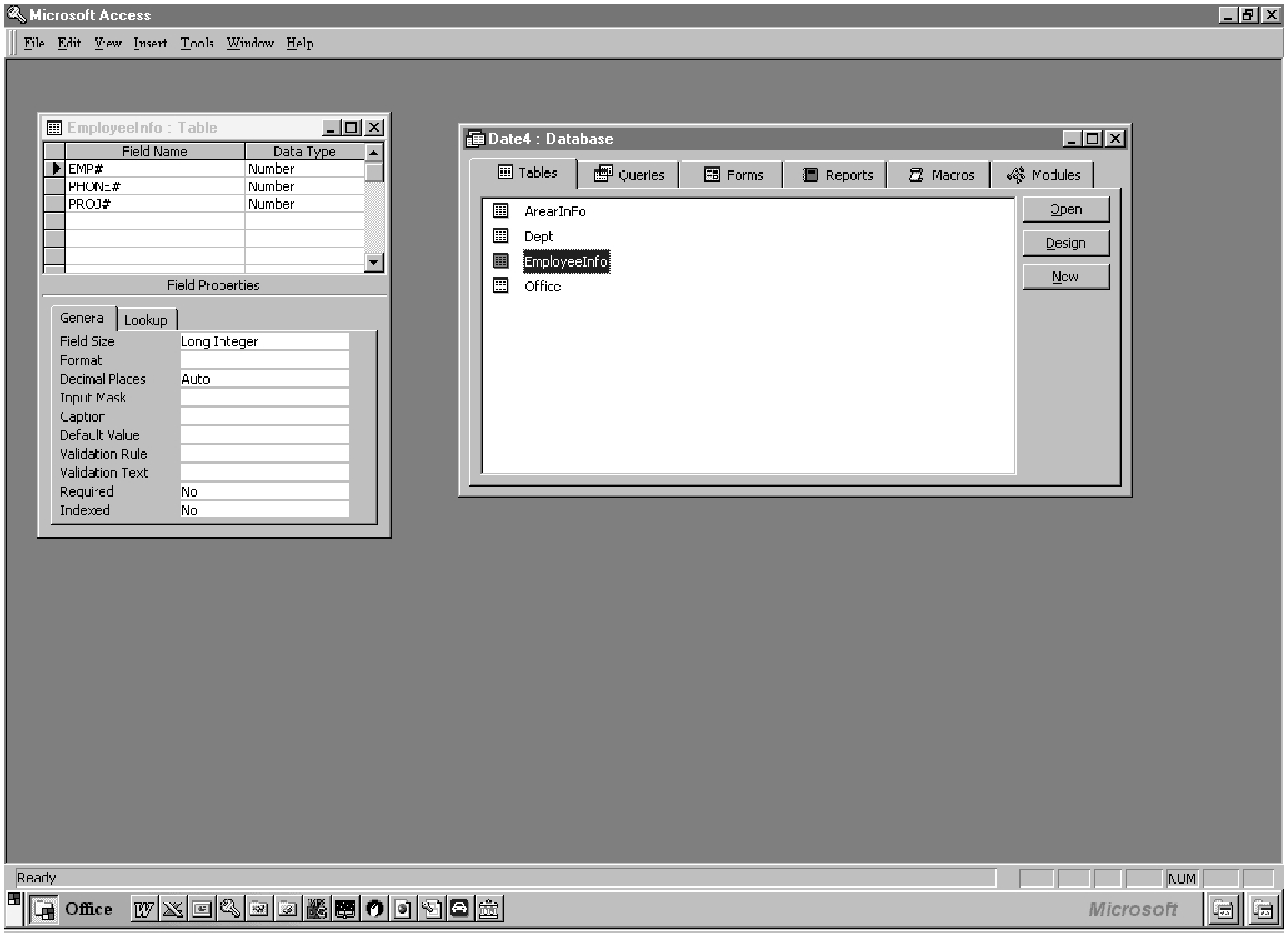
Table Name: EmployeeInfo

Current Attribute: [Key] EMP#

Select a data type from the data type list for the current attribute which is displayed in the current attribute display.

Once selected, click the selected command button. Repeat for all attributes.

Figure 8. Defining table name and attribute data types.



Field Name	Data Type
EMP#	Number
PHONE#	Number
PROJ#	Number

Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Indexed	No

AreaInFo
Dept
EmployeeInfo
Office

Figure 9. MS Access environment window showing the created tables.

In the window as shown in Fig. 8, the user enters the name of the table, highlights each attribute and selects a suitable data type from a given list. Once the process is complete, a `Create` button appears in the window. The user can then press the button to finally create a table definition in Access.

Figure 9 illustrates that four final tables chosen by the user are created and listed in the Table category of the MS Access DBMS environment.

4.3 *Discussions*

One problem associated with BCNF normalization is that the property for functional dependency preservation may be violated. In other words, certain functional dependencies may be lost after the decomposition. This problem is well documented in the literature [4] (pp. 309−311). Micro warns the user about this problem before any BCNF normalization. The user has the option to proceed with the decomposition or to remain at 3NF relation schemes.

In order to validate our algorithms, we have tested the Micro system using several examples from well-known textbooks (the example on p. 317 of [4] and the example on pp. 412−414 of [1]). The results produced by Micro exactly match those produced manually in the books. Other examples, which involve tens of attributes and tens of functional dependencies of various complexities, have also been performed. The system works satisfactorily.

5. Conclusions

In this paper, we presented a prototype system for automatic normalization. The system applies abstract algorithms in the literature into practical database design tools, liberating the database designer from a tedious manual process of normalization. The database designer only needs to identify and specify functional dependencies among a list of attributes. The system allows the designer to create real table structures in Microsoft Access format. It demonstrates that such an automatic normalization system is practical.

In order to further strengthen the belief that Micro can be really useful for designing large databases, bigger experiments that involve hundreds of attributes and functional dependencies are needed. However, it is unlikely in practice that the number of attributes and the functional dependencies are within the range of thousands or more.

References

1. R. Elmasri & S. B. Navathe 1994. *Fundamentals of Database Systems (2nd Edition)*. Benjaming/Cummings Publishing Company, pp. 395−441.
2. J. Ullman 1982. *Principles of Database Systems*. Computer Press, pp. 213−250.
3. H. Mannila & K. J. Raiha 1989. Practical Algorithms for Finding Prime Attributes and Testing Normal Forms. In *Proceedings of 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 128−133.

4. C. J. Date 1995. *An Introduction to Database Systems (6th Edition)*. Addison-Wesley, pp. 309–317.
5. T. Connolly & C. Begg 1999. *Database Systems, A Practical Approach to Design, Implementation and Management (2nd Edition)*. Addison-Wesley, p. 193.
6. Microsoft Corporation 1992. *The Window Interface: An Application Design Guide*. Redmond: Microsoft Press.

Hongbo Du received his BSc degree in Computer Science from Beijing University of Science and Technology in 1982. He received the MSc degree in Computer Studies in 1986 and the MPhil degree in Computing in 1990 both from the University of Essex, UK.

He is currently a lecturer at the Department of Computer Science, University of Buckingham, UK. His research interests include information discovery and data mining, visual languages, and information retrieval for multimedia databases.

Laurent Wery received the BSc degree in Computer Science from University of Buckingham UK in 1997. Since his graduation, he has been working in the IT industry first as analyst/programmer and then a freelance consultant. He is specialised in database design and database application development, and various programming languages.