# CHAPTER 9

# DATABASE NORMALIZATION DESIGN TOOL

## 9.1 INTRODUCTION

Since this work is centered on designing of database tables, a comprehensive tool has been developed which automates the design. This software helps in normalizing the database relational schemas without going through a rigorous manual process. *Normalization* is a process that reduces the redundancies and removes any possible anomalies. A number of normal forms are well defined in the literature. The most significant of these are 3NF and BCNF. The normalization process initially accepts a relational schema, *R* and checked for a series of normal forms. It is generally desired to normalize the relations up to BCNF, though it is sufficient to stop at 3NF. However, dependency preservation is not guaranteed in BCNF [Ullman, 1988] [Peter, 2004].

There are two ways by which normalization process can be done. First, the relation along with the Functional Dependencies (FDs) may be given as the input. Second, the instance of the relation filled with sample records could be given. In the first case, normalization can be carried out using classical algorithms and in the second case the FDs have to be discovered in *synthesis* method. Both these problems are addressed and implemented here.

In order to normalize any arbitrary relation, one would compute first the *candidate keys* and employ this in the normalization algorithms. An efficient algorithm is used in order to determine the candidate keys of a given relation. This algorithm is based on building a *dependency graph* for a given set of FDs.

Though there are many classical algorithms to compute the candidate keys, the most efficient one is given by Saiedian and Spencer [Saiedian, 1996]. Once the candidate keys are computed, theseare used in 3NF and BCNF normalization algorithms. For 3NF the algorithm given in [Saiedian, 1996] is used. Similarly, for BCNF the algorithm suggested by Tsou and Fischer [Tsou, 1980] is used. To discover the exact FDs from a given relation, an extension of

the SQL-based algorithm of Victor Matos and Becky Grasser [Victor, 2004] is used with the Oracle database.

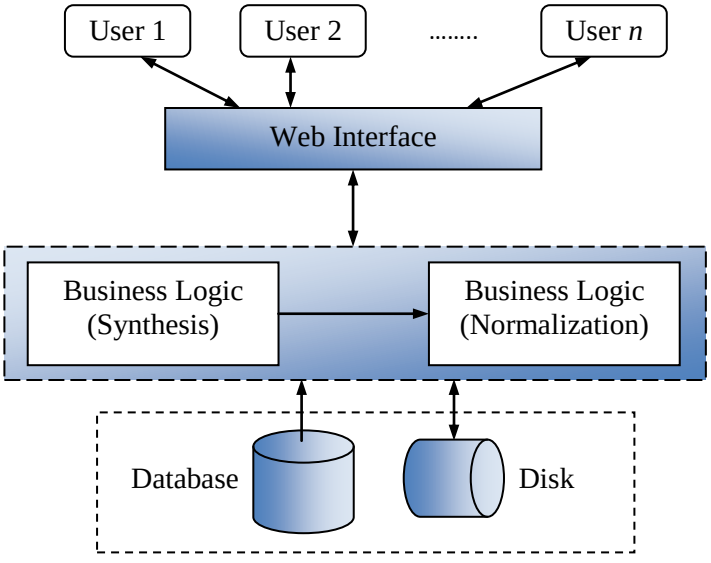


Fig. 9.1 Schematic showing the Architecture of the proposed Normalization Software

The architecture of the software tool is as shown in Figure 9.1. It is basically a three-layer architecture. The users, from the home page, can go to the appropriate web page after registering themselves to the web site. Every user has two options to select namely *normalize* or *synthesis*. If the normalize hyper link is clicked, they can either type their relation attributes and FDs or load some pre-defined relations. If the synthesis hyper link is selected, a set of Oracle tables will be displayed from which appropriate table of their interest, for which the FDs are to be discovered, may be selected. Then the output (all exact FDs of *R*) is displayed.

From the *normalize* approach, once the input data is typed, the decomposed tables can be seen on a separate web page called the *result page*. We can visualize the whole system having three layers: users in the top layer, the computational part for normalization and *synthesis* engine in the middle layer, and the physical database in the bottom layer.

Having decided on the various algorithms to be used, the next issue is to implement these algorithms. The major points to be considered for implementation are the language, platform, and other supporting tools or utilities. As explained earlier, the tool is an online web-based software. The Microsoft Visual Studio 2005 has been used as the major tool to design. In particular, C# is used for the entire development along with ASP.NET and ADO.NET for web design and data access respectively.

## 9.2 DEFINITIONS AND BACKGROUND

A relational database schema $R$ is denoted by $R$ ($A_1$, $A_2$, …,$A_n$). Here, $A_1$, $A_2$, …, $A_n$ are the attributes of the database schema. Here, the term *relation* to mean relational database schema. The elements of the relation are called as the rows or *tuples*. Every relation has one or more *Functional Dependencies* (FDs). An FD is to describe the relationship between attributes in a given relation, $R$ denoted as $X \rightarrow Y$, which means that $Y$ is functionally dependent on $X$. Let $F$ be the set of FDs that are defined for $R$.

By definition, a relation can't have two tuples with the same value. This is known as the *uniqueness* property. In practice, it is often the case that some proper subset of the set of all attributes of the relation $R$ has the uniqueness property.

**Definition:** Let $K$ be a subset of attributes of the relation $R$. Then, $K$ is a candidate key if and only if it satisfies the following two conditions:

**Uniqueness:** No legal values of $R$ ever contain two distinct tuples with the same values. That is, for any two tuples $t_1$ and $t_2$, $t_1[K] \neq t_2[K]$.

**Irreducibility:** No proper subset of $K$ has the uniqueness property.

Every relation has at least one candidate key. $K$ is considered as a *superkey* if it satisfies the uniqueness property but not the irreducibility property. Any attribute that is a member of a key is called as *prime* attribute and the other attributes are called as *nonprime* attributes.

An FD is *trivial* if and only if the attributes in the right-hand side is a subset of the left-hand side. That is, $X \rightarrow Y$ is trivial if $Y \subseteq X$. For example, $AB \rightarrow B$ is trivial whereas $AC \rightarrow BC$ is nontrivial. The set of all FDs that are implied by a given set of FDs, say $F$, is called as the *closure* of $F$ and is denoted by $F^+$. The new FDs that are inferred are based upon the *Armstrong Axioms* [Navathe, 2009] [Zaniolo, 1982]. Given a relation $R$ with a set of attributes $X$ and a set of FDs $F$ that hold on $R$, in addition to computing $F^+$, also the attribute closure, denoted as $X^+$ is computed.

The relation $R$ is in 3NF if and only if it is already in Second Normal Form (2NF) [Ramamkrishnan, 2004] and if, for every FD $X \rightarrow A$ in $F^+$, at least one of the following statements is true:

- $A \subseteq X$ (trivial functional dependency)
- $A$ is a part of a candidate key $K$ of $R$ ($A$ is prime)
- $X$ is a superkey

A relation $R$ is in BCNF, if for every FD $X \rightarrow A$ in $F^+$, at least one of the following statements is true:

- $A \subseteq X$ (trivial functional dependency)
- $X$ is a superkey

When the relation is neither in 3NF nor in BCNF, then appropriate decomposition algorithms is to be applied [Zaniolo, 1982].

## 9.3 ALGORITHM FOR COMPUTING THE CANDIDATE KEYS

Though many algorithms were proposed over the years, the algorithm developed by Saiedian and Spencer has been used here. The method is based on the attribute graph being drawn for the given FDs as a directed graph. A modified algorithm is proposed by redefining the graph as *dependency graph*. A *dependency graph* is a digraph with one vertex for each attribute of $R$ and one directed edge for each functional dependency in the set $F$. The direction of the edge is drawn as per the given dependency. For example, the FD $X \rightarrow Y$ is represented as a directed edge from vertex $X$ to vertex $Y$. In case of multiple attributes on left-hand side and/or right-hand side, edges are drawn for each of the attributes.

To determine the candidate keys, the algorithm shown below is followed. Let $L_S$ denote an attribute set which contains all the attributes that appear *only* on LHS, $R_S$ denote all the attributes that appear *only* on the RHS, and $B_S$ denote all the attributes that appear on *both* the sides of $F$. Also assume that $C_K$ is an array of attribute set to hold all the candidate keys. With this representation, the following algorithm can find all the candidate keys of $R$:

In the below algorithm, Step 1 is the initialization of the attribute set variables which would hold the various attributes and candidate keys. Step 2 takes care of building the dependency graph. From the graph we can easily find the indegree and outdegree of each vertex. Accumulate these vertices (i.e. attribute set) to the appropriate variables $L_S$, $R_S$, and $B_S$.

**Algorithm** *CandidateKey*(*R, F*)
1. $L_S \leftarrow \phi$; $R_S \leftarrow \phi$; $B_S \leftarrow \phi$; $C_K \leftarrow \phi$
2. Build the dependency graph for *R* and *F*.
3. From the graph compute the following:
   $L_S \leftarrow L_S \cup$ all vertices with zero indegree
   $R_S \leftarrow R_S \cup$ all vertices with zero outdegree
   $B_S \leftarrow B_S \cup$ all vertices with nonzero indegree and outdegree
4. **If** $L_S = \phi$ and $R_S = \phi$ **Then**
   use the algorithm of [Saiedian, 1996] // all attributes in $B_S$
5. **If** $L_S^+ = R$
   $C_K \leftarrow C_K \cup L_S$;
   **goto** step 7.
6. **For** each attribute $A \in B_S$ **do**
   $L_S \leftarrow L_S \cup A$;
   **If** $L_S^+ = R$
   $C_K \leftarrow C_K \cup L_S$;
   $L_S \leftarrow L_S - A$;
7. **Return** $C_K$.

Suppose, if all the attributes are in $B_S$ ($L_S$ and $R_S$ are empty), then every subset of attributes in $B_S$ may be a candidate to be part of the key. This requires exponential time algorithms and handled by using the algorithm of [Saiedian, 1996]. If $L_S$ includes all the attributes of *R* (closure), then this is the only candidate key. Otherwise, add one attribute at a time from $B_S$ to $L_S$ and compute the closure to find all the candidate keys.

### 9.3.1 COMPUTING STRONGLY CONNECTED COMPONENTS

One of the important steps in the algorithm proposed by Saiedian [Saiedian, 1996] is to calculate the S*trongly Connected Components* of the given dependency graph *G*. For this the *Depth-First Search* (DFS) method is used as shown in Figure 9.2 and call it as *Strongly_Connected_Components*(*G*) [Cormen, 1999]. In this algorithm, it is assumed that *f* is the finish times, *d* is the discovered vertices, and *S* is a set of strongly connected components. Note that *G*[*R, F*] represents the dependency graph for the given *R* and *F*.

In order to represent the graph an adjacency list is used and hence the algorithm takes only time complexity of linear time, $\Theta(V + E)$, where *V* is the number of vertices and *E* is the number of edges. To compute $G^T$ the brute-force method of traversing the *adjacency list* of *G* and construct a new adjacency list by adding reversed edge information is adopted.

**Algorithm** Strongly_Connected_Components($G[R, F]$)
1. $f \leftarrow \phi$; $d \leftarrow \phi$; $S \leftarrow \phi$;
2. **Foreach** vertex $u$ in $G[R, F]$ **do**
   Compute finish times $f[u]$ and $d[u]$ using DFS($u$).
3. Compute $G^T$ ($G^T$ *is a graph of G with all edges reversed*)
4. Sort $f$ in descending order.
5. **Foreach** vertex $u$ in $f$ **do**
   Call DFS($u$);
6. $S \leftarrow S \cup$ {Strongly Connected Components of trees in DFS forest}.
7. **Return** $S$.

Fig. 9.2 Algorithm to Compute Strongly Connected Components

The second step in the algorithm is to compute the *source* component of $G$ among all the strongly connected components that have been found using the above algorithm. This is obtained by deleting the edges in the adjacency list of $G$ within any given strongly connected graph. Here only the edges between strongly connected components are retained. From this reduced graph determining the source component is easy.

The algorithm for computing the source components of the given dependency graph can be stated as follows:

**Algorithm** *Source_Components*($G[R, F]$, $S$)
1. $C \leftarrow \phi$; $I \leftarrow \phi$;
2. **Foreach** strongly connected component $Z$ in $S$ **do**
   Delete all edges in Z;
3. Build the reduced graph $G^R$
4. **Foreach** vertex $u$ in $G^R$ **do**
   Compute $I[u]$; // Indegree of each vertex
   **If** $I[u] = 0$
   $C \leftarrow C \cup u$;
5. **Return** $C$.

The algorithm takes two arguments: the first one is the graph and the second is a set of all strongly connected components. Step 2 deletes all the edges that are within the same component so that a reduced graph $G^R$ consisting of each vertex corresponding to the strongly connected component is obtained. From $G^R$ the indegree of each vertex is found and stored in a

vector called *I*. All vertices with indegree zero are collected and returned as the source components.

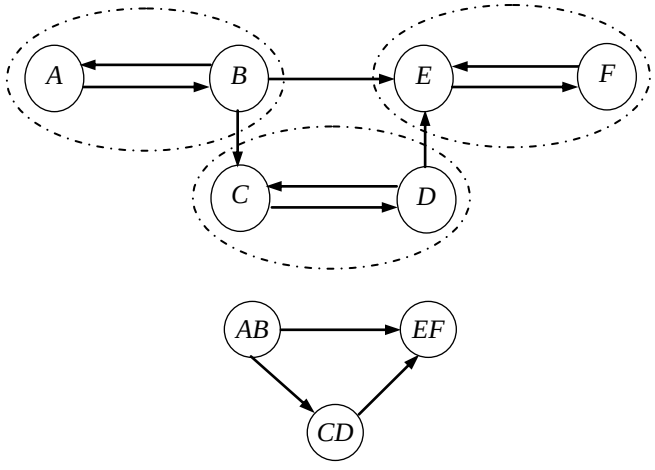


Fig. 9.3 Strongly Connected Components and $G^R$

Applying the above two algorithms to the graph of Figure 9.3, a three vertex graph is obtained as there are only three strongly connected components, shown as dotted lines. After step 4, *C* contains {*AB*} as it is the only source component. In case if $G^R$ has more than one source component, all such vertices are returned.

### 9.3.2 FINAL ALGORITHM

This section presents the complete algorithm for computing ***all*** the candidate keys, irrespective of whether the graph is strongly connected or not. Let $S_r$ represent the set of keys of restrict, $S_c$ represent the set of keys of contract, *S* stores the final set of all candidate keys, *Z* to store all the strongly connected components, and *X* stores the source components.

**Algorithm** SS_CandidateKeys(*R, F*)
1. $S_r \leftarrow \phi$; $Sc \leftarrow \phi$; $S \leftarrow \phi$; $X \leftarrow \phi$; $Z \leftarrow \phi$;
2. Build the dependency graph G[R, F].
3. **If** IsSCC(*G*)
   $S_c = S_c \cup$ MinimalKeys(*R, F*)
   **Return** $S_c$;
4. $Z \leftarrow$ Strongly_Connected_Components(*R, F*);
5. $X \leftarrow$ Source_Components(*R, F, Z*);
6. // $F_r$ is the set of FDs of Restrict
   Compute restrict(*X*, $F_r$);
7. $S_r \leftarrow S_r \cup$ MinimalKeys(*X*, $F_r$);
8. $T \leftarrow R - X^+$

9. // Compute Contract
   **Foreach** *fd* in *F* **do**
     **Foreach** attribute *A* in *fd* **do**
       If $A \notin T$ Then remove *A* from *fd*;
10. **Foreach** *fd* in *F* **do**
     **If** $fd_{LHS} = \phi$ or $fd_{RHS} = \phi$
       $F \leftarrow F - fd$;
11. $S_c \leftarrow S_c \cup$ SS_CandidateKeys(*T*, *F*);
12. **Foreach** key $K_r \in S_r$ **do**
     **Foreach** key $K_c \in S_c$ **do**
       $S \leftarrow K_r \cup K_c$;
13. **Return** *S*.

The algorithm first tests whether the given relation *R* is strongly connected. If it is so then it uses the Lucchesi and Spencer [Lucchesi, 1978] method of computing all the candidate keys. Otherwise, it finds the *restrict* and *contract* components; then calls *SS_CandidateKeys* algorithm recursively. Finally, *S* returns the set of all candidate keys.

## 9.4 ALGORITHM FOR 3NF NORMALIZATION

Once all the candidate keys of *R* for a set of FDs *F* are known, finding it being in 3NF is easy [Douglas, 2004]. The problem can be stated as:

*For all FDs $X \rightarrow A$ in F, either X is superkey or A is prime, then R is in 3NF. When at least one FD violates this condition, then R is not in 3NF and requires decomposition.*

*SS_CandidateKeys*(*R*, *F*) and *MinimalCover*(*R*, *F*) are used to decompose *R* into 3NF relations. Below is the algorithm that uses *CK* for a set of candidate keys, *MC* for minimal cover [Maier, 1980] FD set, *Z* for the set of decomposed relations.

**Algorithm** *TNF*(*R*, *F*)
1. $CK \leftarrow \phi$; $MC \leftarrow \phi$; $Z \leftarrow \phi$; $J \leftarrow \phi$; $Y \leftarrow \phi$;
2. $CK \leftarrow$ *SS_CandidateKeys*(*R*, *F*);
3. $MC \leftarrow$ *MinimalCover*(*R*, *F*);
4. *test* $\leftarrow$ *Is3NF*(*R*, *F*);
5. **If** *test* = true
     **Return** *F*;
6. // Merge FDs whose LHS are same
   **Foreach** *fd* in *MC* **do**
     $S \leftarrow \{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$;

$S \leftarrow \{X \rightarrow A_1A_2A_3\ldots.A_n\}$;
$Z \leftarrow Z \cup S$;

7. // Put all unplaced attributes into a new relation
   **Foreach** *fd* in $Z$ **do**
   $J \leftarrow J \cup \{fd_{LHS} \cup fd_{RHS}\}$;
   $Y \leftarrow R - J$;
   **If** $Y \neq \phi$
   $T_{LHS} \leftarrow CK$;
   $T_{RHS} \leftarrow \delta$; // dummy attribute
   $Z \leftarrow Z \cup T$; // for dependency preservation
8. // If no relation contains a key *CK*, Then create a
   // new relation *U* that contains the attributes to form a key.
   $Z \leftarrow Z \cup U$;
9. **Return** $Z$.

With this algorithm, the decomposed relations are guaranteed to be in 3NF with lossless-join and dependency preserving properties satisfied. Decomposing the 3NF relations into BCNF with lossless join has been implemented using Tsou and Fischer algorithm [Tsou, 1980].

## 9.5 ALGORITHM FOR DATABASE SYNTHESIS

A simple strategy is proposed to discover the FD match from the tuples of *R* [Brockhausen, 1995]. A SQL query is used to determine the exact rules $X \rightarrow A$ and is illustrated as shown in Figure 9.4(a). The following query checks the validity of the FD $AB \rightarrow C$. It retrieves the distinct count of *C* with *AB* in the GROUP BY clause.

```
SELECT    A||B AS "AB", COUNT(DISTINCT C) AS "CountC"
FROM      R1
GROUP BY A||B;
```

*R1*

| A | B | C |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 2 | 1 |
| 2 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 3 | 1 |
| 3 | 4 | 2 |
| 3 | 4 | 3 |

→

| AB | CountC |
|---|---|
| 11 | 1 |
| 12 | 1 |
| 22 | 1 |
| 23 | 1 |
| 33 | 1 |
| 34 | 2 |

(a) (b)

Fig. 9.4 (a) Relation instance (b) Output

The output of the above query is as shown in Figure 9.4(b). The last two tuples in *R1* have two distinct values for *AB*. By modifying the above query, as shown here, the validity of the FD can be checked.

```
SELECT    A||B AS "AB", COUNT(DISTINCT C) AS "CountC"
FROM      R1
GROUP BY A||B
HAVING    COUNT(DISTINCT C) > 1;
```

When the output of this query is NULL, it signifies that the FD holds. Similarly, SQL statements are written for all the combinations of the attributes but with a constraint that the right hand side of the FD should be a single character.

For the relation instance of Figure 9.4(a), 7 combinations are obtained as follows: *A*, *B*, *C*, *AB*, *AC*, *BC*, and *ABC*. Each combination defines the X attribute set and $R - X$ gives the *Y* attribute set. Here the FDs to be checked are: $A \rightarrow BC$, $B \rightarrow AC$, $C \rightarrow AB$, $AB \rightarrow C$, $AC \rightarrow B$, $BC \rightarrow A$. However, when any particular FD is checked the RHS attribute should be a single character. In $A \rightarrow BC$, both the FDs $A \rightarrow B$ and $A \rightarrow C$ must be tried.

## 9.6 SOFTWARE TOOL: THE FORMAL DESIGN

The formal design of this tool is based on object oriented strategy for which a language like C# is used. This tool is named as *DNorm* version 1.0 and the software requirement specifications are listed below:

- Given a relational schema, *R* with a set of attributes and FDs, the tool is expected to normalize up to BCNF.
- In case the FDs are not available, the relational instance must be given.
- Users must be able to login, logout, normalize their relations, see the output in a readable form, save their output, load some sample relations.
- Authentication of users shall be done by storing the encrypted passwords, may be by using XML files or database table.
- It should work in Web browsers like Internet Explorer or Netscape Navigator.

### 9.6.1 USE CASE MODEL

Figure 9.5 shows the Use Case diagram for the *DNorm* version 1.0 package. The only actor in this system is the user and the possible use cases are *Normalization*, *Synthesis*, and *Log In*. As mentioned earlier, any legal user (with proper user name and password), will be able to interact with the Normalization and Synthesis use cases.

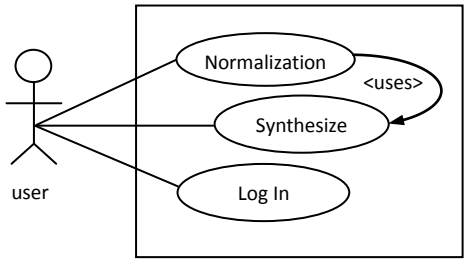


Fig. 9.5 Use Case Diagram of *DNorm* 1.0 System

When users provide just the relational instance, the *Normalization* use case *uses* the *Synthesis* use case to discover the exact FDs. Since the main objective of this system is to decompose the relations, no other features like the other commercial sites are included. Next, Figure 9.6 shows the detailed use case diagram of Normalization process.

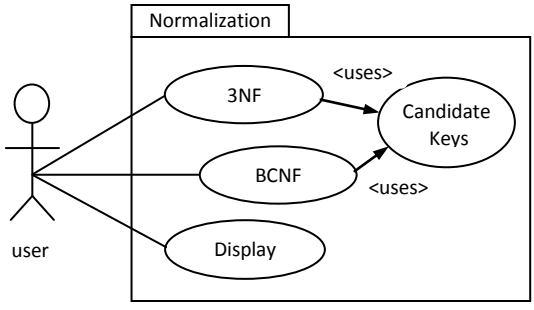


Fig. 9.6 Use Case Diagram of Normalization Process

When the user submits the required input to the system the main task of the Normalization use case is to invoke 3NF and BCNF algorithms. Note that Candidate Key use case is being used by both these algorithms. Yet another process, minimal cover, may also be used to compute 3NF which is not shown in the use case diagram. Similarly, the *Display* use case helps the user to display the decomposed relations for further processing.

Following are the assumptions made while designing the tool:

- The Table name is a simple text, the attributes are set of characters (uppercase or lowercase) but restricted to 'A' to 'Z' or 'a' to 'z'.
- The FDs are generally written as two different items: left-side attribute set and right-side attribute set.
- Any number of attributes may be written; but attributes are selected from check boxes to avoid erroneous input data (e.g. duplicates).
- No flexibility to dynamically upload user's sample data to the tool from disk files.
- All the candidate keys of *R* and the normalized relations shall be shown.
- The BCNF is final [Douglas, 2004] and no attempt is made to normalize *R* into 4NF and 5NF.

### 9.6.2 ACTIVITY DIAGRAM

The *DNorm* version 1.0 system is built keeping in mind the following major activities as shown in Figure 9.7 for the activity diagram:

a) Handle the users. It should authenticate the logged on users and provide with proper messages. For instance, if the user is not a registered user, then show the registration page.
b) Once the user is successfully logged in, there are two major activities that could be interacted. The first one is the normalization and the other is the synthesis.
c) If the user selects the normalization link, he would be prompted with data entry panel where the attributes and FDs of *R* can be entered.

d) Alternatively, the synthesis activity again prompts the user to select the relation which contains sample tuples.
e) For every activity the results are to be displayed. A separate web page must take care of displaying the results in a tabular form.
f) After applying synthesis we get the output in the form of FDs. Now it is left to the user to send these FDs to the normalization process or simply take a look at the FDs (display). Also notice that the FDs are obtained using SQL-based pattern search algorithm as explained already.

The diagram may not depict all the activities of the system, but it does bring out the major activities.

## 9.7 IMPLEMENTATION OF *DNORM* SYSTEM

The implementation of the tool is broadly encompasses the following modules:

**(a) Normalization**: Computing Candidate Keys, Minimal Cover, 3NF, BCNF, etc.

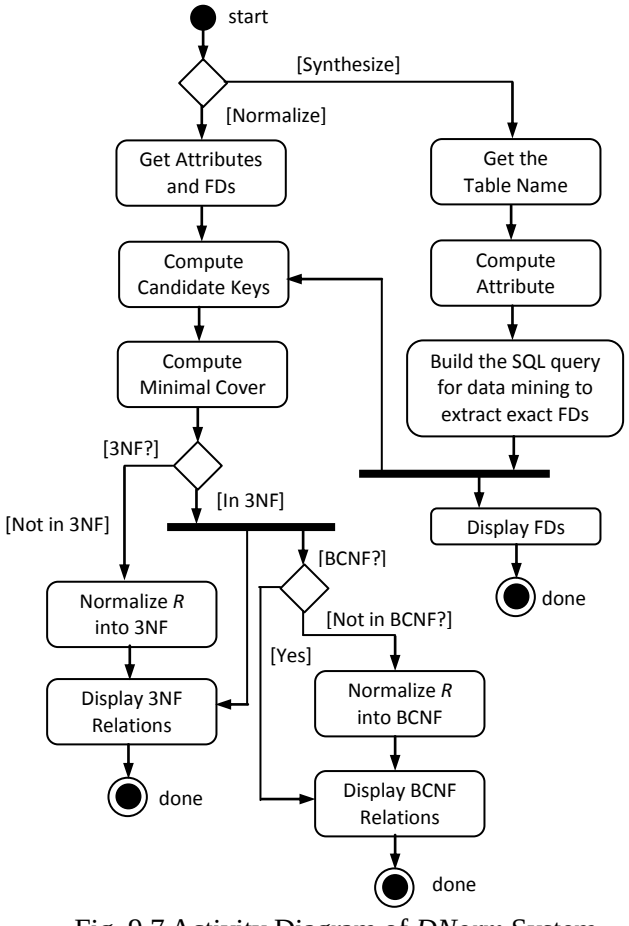


Fig. 9.7 Activity Diagram of *DNorm* System

**(b) Synthesis**: Generating subsets of the attributes, Building SQL query, Mining for the pattern, etc.

**(c) Web Interface**: User management, reading input data, display the results, etc.

In order to apply Object Oriented technique for this system, C# and ASP.NET under Visual Studio 2005 framework is preferred. To store the relation instance for synthesis, Oracle 10g database is used. In fact, ASP.NET is ideally suited for web application development as C# can be used for implementing the business logic.

The next three sections will explain the implementation logic of all these modules.

**(a) Business Logic for Normalization**

As explained earlier, there are several classes required to build the system. The major tasks are to store the attributes, FDs, dependency graph, et al. The class diagram is as shown in Figure 9.8.

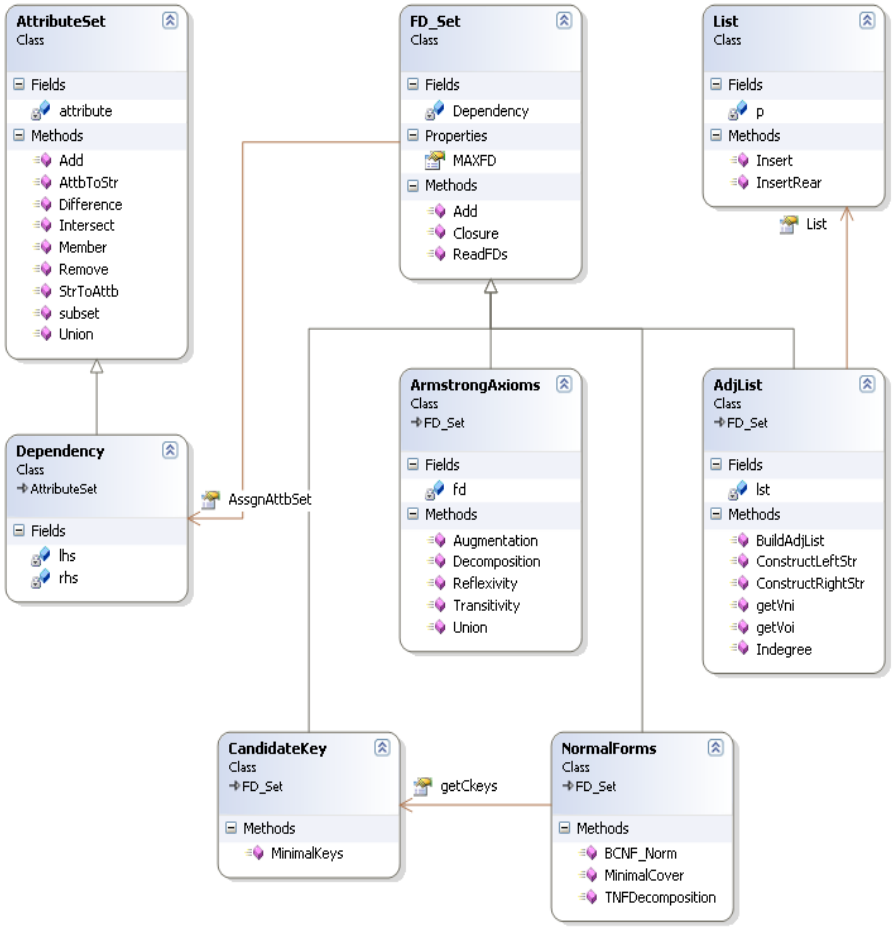


Fig. 9.8 Class Diagram for Normalization

The *attributes* of the LHS or RHS of the FD is stored in the *AttributeSet* class. This class has only one field called *attribute* which is a Boolean array. The object of this class can store a maximum of 26 distinct attributes '*A*' to '*Z*'. Each attribute is converted to an integer which in turn is used as an index in the attribute array to mark it *true*. For example, to store the attribute '*AB*', the attribute field would look like:

| | *attribute*[] | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | ..... | 25 |
| initial | *false* | *false* | *false* | *false* | *false* |
| *A* | *true* | *false* | *false* | *false* | *false* |
| *B* | ***true*** | ***true*** | *false* | *false* | *false* |

After adding *AB*, *attribute*[0] and *attribute*[1] will change from their initial values of *false* to *true*. To store both LHS and RHS attribute sets, another class called *Dependency* is used. Every object of Dependency can store an FD consisting of *lhs* and *rhs*. Each FD_Set object is capable of storing an array of FDs (MAXFD) each one being a *Dependency* object.

To build the adjacency list of the dependency graph, *List* and *AdjList* classes are designed. The methods in *AdjList* class can take care of building the adjacency list, computing the indegree, and many more tasks needed to compute the candidate keys. The key tasks of normalization algorithms are included in a separate class called *NormalForms* which in turn uses *FD_Set* class as indicated in the class diagram.

All these classes are stored in a single namespace called '*DependencyGraph*' and Visual Studio 2005 has an option to build this project as a class library. Therefore, a dll file called DNorm-DLL.dll is created so that this can be easily added in the design of web forms using ASP.NET.

### 9.7.1 BUSINESS LOGIC FOR SYNTHESIS

As explained in the earlier sections, when a user does not supply the FDs, but the instance, an elegant method must be devised to discover the FDs based on SQL queries. The requirement is that the attributes of the relation must be single character and the rows can remain as it is, assuming that the relation is available in the default user area Scott.

Two major classes are used: *GenerateSubsets* and *FindFD*. The logic for generating the subset of attributes is based upon a classical algorithm. Converting the *n* attributes into binary

and generating all combinations, i.e. $2^n - 1$, an array of size $2^n - 1 \times n$ is built by adding attribute into the *ArrayList*, whenever the value is one.

The *FindFD* class prepares the LHS attribute set and RHS attribute set and uses the SQL query to check whether this FD holds or not. When the queries yield no rows, only those FD combinations are stored in an *ArrayList* collection. This data structure can then be sent to the Normalization module for further processing.

### 9.7.2 WEB INTERFACE

ASP.NET 2.0 is used for web interface design. The following are the main web pages:

**a)** **Home Page** (*index.aspx*): This page is to give an introduction to the *DNorm* 1.0 and enables the user to log in or register.

**b)** **Data Entry Page** (*Default.aspx*): Users can enter the relation name, attributes (through check boxes), and FDs. Alternatively, they can load certain predefined examples too.

**c)** **Synthesis Page** (*Synthesize.aspx*): By default the system connects to Oracle database under the default user Scott and displays all the tables. One can select the table from the *ListBox* and the columns and the tuples are displayed. By clicking the Compute button the exact FDs computed are displayed.

**d)** **New Users Page** (*Register.aspx*): New users can register themselves by typing the relevant details (details are shown later).

**e)** **Examples Page** (*examples.aspx*): It is a simple HTML page containing the predefined example attributes and FDs.

**f)** **Results Page** (*Resultsnewp.aspx*): This page uses a *GridView* to display the various decomposed tables.

A novel technique for storing the user name and password in a XML file is used. The Users.xml file would like:

```
<Users>
   <Users>
     <UserName>gayan</UserName>
       <UserPassword>
         134B20FFF068BEBD6014C15D1BF2F3BEC3FDC8A6
       </UserPassword>
   </Users>
   <Users>
     <UserName>sng</UserName>
       <UserPassword>
         40BD001563085FC35165329EA1FF5C5ECBDBBEEF
```

```
        </UserPassword>
    </Users>
</Users>
```

When a new user registers, the password is hashed using the *HashPasswordForStoringInConfigFile*() method and stored along with the user name in the XML file (see the <*UserPassword*> tag above). This file is stored in the web server's folder *C:\Inetpub\wwwroot\Users.xml*. Web-based programming distinguishes itself as a programming idiom in which the applications are managed serving multiple users distributed over a wide area. This web application will be made available to a large number of clients, and hence it is necessary to use Session State in ASP.NET. All the objects that are needed for the decomposition and synthesis are stored in these Session objects. Normally these session variables are stored in a separate file called *Global.asax* including whether a user has logged in or not.

## 9.8 TESTING AND RESULTS

Testing web based multiuser kind of software is a difficult task. A systematic testing procedure and rigorous evaluation has been carried out. Since .NET Framework works on the Windows-XP platform, this tool also is hosted on the Windows platform [Troelsen, 2003]. Following are the steps used:

**a)** Normalization module has been tested by giving input of all possible examples that appear in the Reference section of this paper. Before integrating with the web interface, the results are seen as a console output (See Table 9.1 to 9.4).

Table 9.1 Candidate Keys

| Sl. No. | Relation Attributes and FDs | Expected Output | Result |
|---|---|---|---|
| 1 | $R(A, B, C, G, H, I)$<br>$F = \{A \rightarrow BC, CG \rightarrow HI, B \rightarrow H\}$ | $\{A\ G\}$ | OK |
| 2 | $R(A, B, C, D, E, H)$<br>$F = \{A \rightarrow B, AB \rightarrow E, BH \rightarrow C,$<br>$C \rightarrow D, D \rightarrow A\}$ | $\{A\ H\}$<br>$\{B\ H\}$<br>$\{C\ H\}$<br>$\{D\ H\}$ | OK |

Table 9.2 Minimal Cover

| Sl. No. | Relation Attributes and FDs | Expected Output | Result |
|---|---|---|---|
| 1 | $R(A, B, C, G, H, I)$<br>$F = \{A \rightarrow BC, CG \rightarrow HI, B \rightarrow H\}$ | $\{A\ G\}$ | OK |
| 2 | $R(C, S, Z)$<br>$F = \{CS \rightarrow Z, Z \rightarrow C\}$ | $\{C\ S\}$<br>$\{S\ Z\}$ | OK |

Table 9.3 Third Normal Form

| Sl. No. | Relation Attributes and FDs | Expected Output | Result |
|---|---|---|---|
| 1 | $R(C, S, Z)$<br>$F = \{CS \rightarrow Z, Z \rightarrow C\}$ | CK = $\{C\ S,\ S\ Z\}$ | in 3NF |
| 2 | $R(S, A, I, P)$<br>$F = \{SI \rightarrow P, S \rightarrow A\}$ | CK = $\{S\ I\}$<br>$D = \{SIP, SA\}$ | OK |
| 3 | $R(A, B, C)$<br>$F = \{A \rightarrow B, C \rightarrow B\}$ | CK = $\{AC\}$<br>$D = \{AB, BC, AC\}$ | OK |
| 4 | $R(A, B, C, D, E, F, G, H, I, J)$<br>$F = \{AB \rightarrow C, A \rightarrow DE,$<br>$B \rightarrow F,\ F \rightarrow GH, D \rightarrow IJ\}$ | CK = $\{ABD, ABF\}$<br>$D = \{ABC, ADE,$<br>$BF, FGH, DIJ\}$ | OK |

Table 9.4 Boyce-Codd Normal Form

| Sl. No. | Relation Attributes and FDs | Expected Output | Result |
|---|---|---|---|
| 1 | $R(C, S, Z)$<br>$F = \{CS \rightarrow Z, Z \rightarrow C\}$ | CK = $\{CS, SZ\}$<br>$D = \{SZ, ZC\}$ | OK |
| 2 | $R(A, B, C)$<br>$F = \{C \rightarrow A, C \rightarrow B\}$ | CK = $\{C\}$<br>$D = \{$ in BCNF$\}$ | |

**b)** Next, the Synthesis module is tested by creating few example tables in Oracle with different attributes and rows.

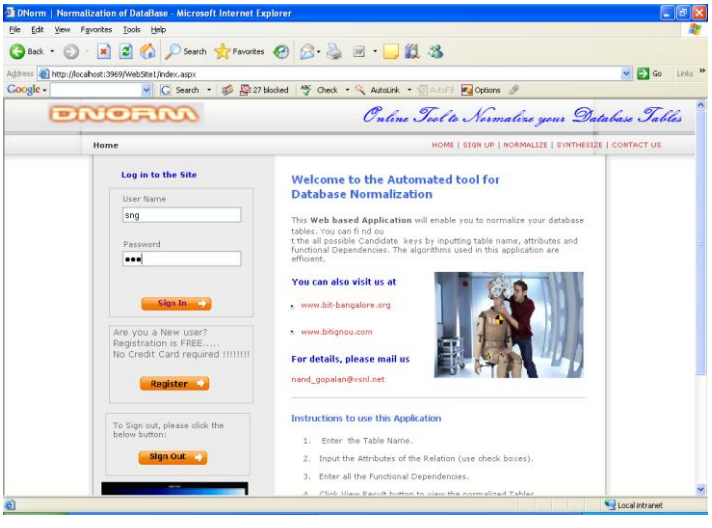


Fig. 9.9 Snapshot of the Home Page

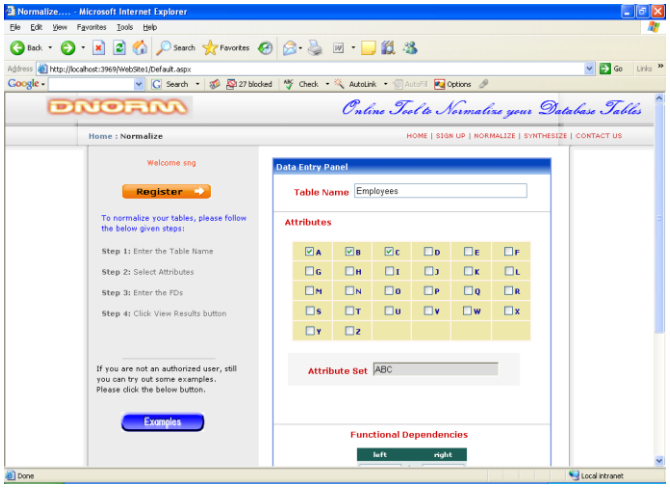


Fig. 9.10 Normalization Page

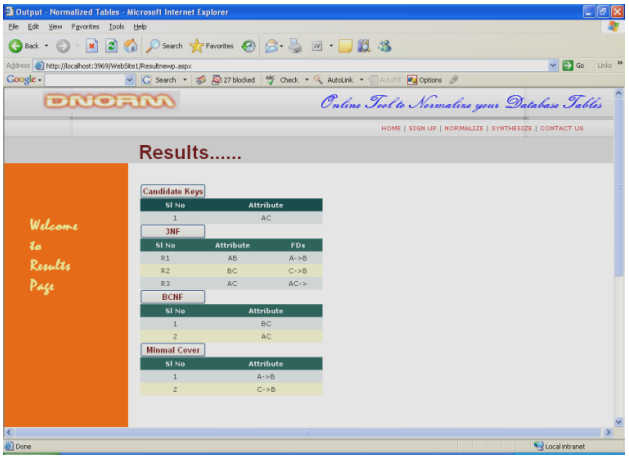


Fig. 9.11 Results Page

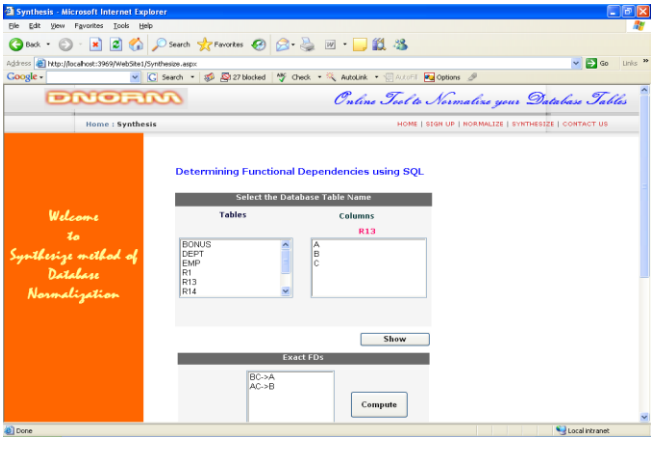


Fig. 9.12 Synthesis Page

All these modules are integrated to the ASP.NET environment and tested for various inputs. These web pages are tested first by executing the pages with *localhost*. To test in the actual internet or intranet environment, first the Internet Information Services (IIS) is configured in the web server. Then the site is published on the web server. The results of the web pages are shown in Figures 9.9 to 9.12. Apart from the sample test data several other FDs have been tried and the output has been verified.

## 9.9 DISCUSSIONS

This chapter presented a novel way of designing the database applications. It is mainly an experimental work where more emphasize was given to the implementation issues and the working of the tool. It is well known that the database application development is a tedious task, because generally there are many relations with complex functional dependencies due to stringent business rules. If the relations are not normalized properly then they have to pay heavy penalty in terms of storage space, time, and man power. This automated tool could solve these drawbacks.

Even when a relation is available with no functional dependencies, but with actual tuples then this tool discovers the FDs and provides normalized relations. Probably the tool could be integrated with the ER diagram preparation tools to make it as integrated design software. Currently the tool does not support 4NF and 5NF normalization processes. However, adding such facilities is not cumbersome. This tool has been used to design various database relations used in the implementation of SQL based K-Means clustering algorithms.

## 9.10 SUMMARY

*Normalization* is a process that reduces the redundancies and removes any possible anomalies. The prominent normal forms are 3NF and BCNF. In order to normalize any arbitrary relation the *candidate keys* are to be computed with efficient algorithms. A simple architecture of the web based database normalization tool called *DNorm* is discussed. ASP.NET 2.0 is used for web interface design under VS2008. A systematic testing procedure and rigorous evaluation is conducted on various relations and the results are as expected.