Abstract—DBNorma [1] is a semi automated database normalization tool, which uses a singly linked list to store a relation and functional dependencies hold on it. This paper describes possible algorithms that can be used to normalize a given relation represented using singly linked list. These algorithms are tested on various relational schemas collected from several research papers/resources and output validated. We have also determined the time required to normalize a given relation in 2NF and 3NF and found that it is proportional to number of attributes and number of functional dependencies present in that relation. The time required, on average, is in order of 186 msec for 2NF and 209 msec for 3NF. By observing time required, one can conclude that these algorithms can be used for normalizing relations within the shorter time; this is specifically needed when database designer is using a universal relation.

Index Terms—Relation, functional dependency, normalization, normal forms.

I.  INTRODUCTION

Profits of any commercial organization are dependant on its productivity and quality of the products. Naturally, to improve profitability it needs to increase productivity without sacrificing quality. To achieve this, it is necessary for an organization to automate the tasks involved in the design and development of its product. From the past few decades relational databases proposed by Dr. Codd [2] are widely used in almost all commercial applications to store, manipulate and use the bulk of data related with a specific enterprise, for decision making. [Detailed discussion on relational database can be found in [3]). Their proven capability to manage the enterprise in a simple, efficient and reliable manner opened a new arena for software industries needing good backend database systems for development of various IT systems for their clients. The success of a relational database systems modeled for a given enterprise is dependant on the design of relational schema and an important step in the design of relational database is Normalization, which takes a roughly defined bigger relation as input along with attributes and functional dependencies and produces more than one smaller relational schema in such a way that they will be free from redundancy, insertion and deletion anomalies. Normalization is mostly carried out manually in the software industries, which demand skilled personnel with expertise in normalization. To model today’s enterprise we require large number of relations, each containing large number of attributes and functional dependencies. So, generally, more than one person’s involvement is needed in manual process of normalization. Following are the obvious drawbacks of normalization carried out manually.

1) It is time consuming and thus less productive: To model an enterprise a large number of relations containing large number of attributes and functional dependencies may be required

2) It is prone to errors: due to reasons stated in 1.

3) It is costly: Since it needs skilled persons having expertise in Relational database design.

To eliminate these drawbacks several researchers already tried for automation of this process. Ali Yazeri, et.al [4] proposed a tool called JMathNorm, which is designed using inbuilt functions provided by Mathematica and thus dependent on Mathematica. This tool provides facility to normalize a given relation up to Boyce-codd normal form including 3NF. Its GUI is written in Java and linked with Mathematica using Jlink library. Hongbo Du and Laurent Wery [5] proposed a tool called Micro, which uses two linked lists to represent a relation along with functional dependencies. One list stores all the attributes and other stores functional dependencies holding on it. Our tool DBNorma uses a novel approach of only one linked list to represent a relation as well as functional dependencies holding on it and thus requires less space as compared to Micro [5].

We have also seen a US patent [6], where a database normalizing system is proposed. This system takes input as a collection of records stored in a table and by observing a record source it normalizes the given database. But our system DBNorma works at schema level i.e. it normalizes a relation before defining a table and entering records into it.

Remaining parts of the paper are organized as follows. Section 2 describes node structure with example. Possible algorithm of 2NF and 3NF used in DBNorma are described in section 3. Standard relational schemas used for experimentation and Experimental results are discussed in Section 4. Conclusions based on empirical evidences are drawn in section 4 and references are cited at the end.
II. SINGLE LINKED LIST NODE STRUCTURE AND REAL WORLD EXAMPLE

As mentioned in [1] node structure used in DBNorma and its description is as follows.

![Node Structure Diagram]

**nodeid**: It is a node identifier (a unique number) assigned to each newly generated node and is stored inside the node itself. This number can be generated by using a counter, which need to be reset for normalizing a new database. When new node is added on a linked list counter will be incremented by 1.

**attribute name**: This field is used to hold the attribute name. It allows underscores and special character and size can at least 50 characters.

**attribute type**: This field is used to hold type of the attribute and will hold m for multivalued attribute, s for atomic attribute, c for composite attribute and p for prime attribute. It will be of size 1 character long.

**determiner_id[]**: This field holds all the determiners of this attribute. A Determiner can be composite or atomic. E.g. Consider node represents an attribute C and we have AB → C and D→ C then the two determiners of C are (A,B) and (D) and thus their nodeid’s will be stored in determiner_ id [] of c.

**determiner**: Determiner is an attribute which takes part in left hand side of FD. This field indicates whether this attribute is determiner or not and of binary valued a size of 1 character will be more than sufficient. If this filed is set to 1 indicates that this attribute is a determiner otherwise it is dependant.

**keyattribute**: This is a binary field and holds 1 if this attribute is taking part in primary key else it is 0. Size of 1 character is sufficient for this purpose.

**ptrToNext**: This field hold pointer (link) to next node and will be NULL if this is the last node on the list.

An example of representing a real word relation and its FD’s using a signally linked list is shown below.

Consider a relation employee containing e_id as primary key e_s_name as employee surname, j_class indicating job category and CPH represent charging per hour. This relation and all FD’s holds on it are shown below.

```
Employee = (e_id, e_s_name, j_class, CPH)
e_id → e_s_name, j_class, CPH
j_class → CPH
```

We have decided to enter all the prime attributes first and then non prime ones. This specific order helps us to get determiners of non prime attributes since they will be already entered in linked list. Hence a new and first node will be created for the prime attribute e_id. Let that Counter is set to 001. A node for e_id attribute is as shown in fig2.

![Figure 2. Snapshot of Linked List when first node is added in it]

In figure 2, first field is set to 001, since it is the nodeid of this node Second field indicates name of attribute i.e. e_id. Third field is set to 1, since e_id is an atomic attribute. Fourth field is set to NULL, indicating that each cell of this field is set to NULL. The fifth field is set to NULL as functional dependencies are not entered yet. The Sixth field is set to 1, since e_id is a key attribute. The last attribute is set to NULL indicating it is the last node on the list.

Now the next attribute to be added is j_class as it is nonprime as well as determiner attribute. Hence the linked list structure after adding this attribute is shown in fig 3.

![Figure 3. Snapshot of Linked List when second node is added in it]

At last all the dependent i.e. e_s_name and CPH will be added into linked list hence the linked list after adding all the attributes details is shown in fig 4.

![Figure 4. Linked List when all nodes are added in it]
After adding all the attributes we need to add information about all the FD's holds on the relation employee in the linked list representation of this relation. Note that FD's will be added one after the other. One more thing is that we need to convert FD into a format such that right hand side will contain only dependant. Thus FD (1) will be broken into three FD's as follows

\[ e\_id \rightarrow e\_s\_name \]
\[ e\_id \rightarrow j\_class \]
\[ e\_id \rightarrow CPH \]

Thus details of 4 FD's will be added. Linked list representation after adding FD's detail is shown in fig 5.

**III. Possible Algorithms for DB Norma**

It is assumed that reader is familiar with definition of different normal forms. On the other hand, tables of relational database are assumed to be in 1NF. Order of entering attributes in linked list is assumed as prime attributes are entered first, then all the determiner other than prime attributes are entered and all the dependents are entered at last.

A. Algorithms For Computing Second Normal Form Of Given Relation

\textit{SecondNormalForm}(listptr): This algorithm takes input as a 1NF relation represented in computers memory in a linked list format and cuts this linked list according to the definition of 2NF into two or more separate lists. These newly created lists represent decomposed relations of a bigger relation taken as input and are in 2NF. A listptr is a single linked list which represents a relation to be converted in to 2NF and FD’s hold on it. This algorithm internally uses other two algorithms namely \textit{PrimeKeyCount}(listptr) and \textit{OtherDependants}(listptr, trav). The function \textit{PrimeKeyCount}(listptr) returns total no of prime key attributes used in input relation. It also uses \textit{OtherDependants}(listptr, trav) function to find out all the dependents of the same determiner. This algorithm uses two variable pointer trav and temp for traversing the list and one integer variable counter. It also uses two arrays to store the newly created tables and attributes used in them. This algorithm is given below and Output of this algorithm is tables generated in 2NF.

\begin{verbatim}
BEGIN
1) Find total no of prime attributes in linked list
   Call PrimeKeyCount(listptr)
2) Node * trav
   trav = Head;
   counter = 0;
   while (trav -> ptrTonext != NULL)
      if (trav -> keyAttribute == 0)
         /* it means attribute is non key attributes */
         then find the determiner id[] of trav
         if (determiner_id[] of trav == primary key)
            then counter++ ;
         if (counter == keyCount)
            /* means full dependancy exists */
            then Call otherDependent(listptr, trav )
         elseif ( (counter < keyCount) &&
               (counter == sizeof(trav -> determiner_id)))
            /* means partial dependancy exist*/
            then Call otherDependent(listptr, trav )
         elseif ( (counter == 0 ) &&
               (counter < sizeof(trav -> determiner_id) )
            /* means transitive dependancy exist */
            then Call otherDependent(listptr, trav )
      Find on which prime arribute both determiner and dependent of trav (current node) depends also add both determiner and dependent in same table of that prime attribute.
3) Once all dependent of same determiner is found
   Create their table in oracle. Mark table name and table attributes in separate array.
   trav = trav -> ptrTonext
4) Repeat step1 to step 3
   for unmark table name and attributes.
END
\end{verbatim}
B. Algorithm for Finding Number of Prime Attributes

PrimeKeyCount(listptr): This algorithm takes a relation (listptr) represented as a single linked list and returns total no of prime attributes present in a relation.

BEGIN
  int keyCount=0;
  node* q;
  if (q keyAttribute ==1) then
    keyCount++;
  q=q->ptrTonext;
  return( keyCount)
END

C. Algorithm for Finding Other Dependent Of Same Determiner:

OtherDependent(listptr,trav): This algorithms is used to find all the dependent non key attributes of same determiner. Consider the relation employee[10], employee(e_id, e_s_name, j_class, CHPH) and functional dependencys hold on it as e_id → e_s_name, j_class, CHPH

j_class → CHPH

In this example while traversing a linked list made for relation employee, when the first non key attribute e_s_name will appear, we will first find its type of functional dependency, then all other non key attributes determined by determiner of e_s_name (i.e. j_class, CHPH) will be returned by using OtherDependent(listptr,trav) Function.

Input: determiner id [ ] of trav of linked list.
Output: Array of all the dependent of same determiner with determiner_id [ ].

BEGIN
  int counter=0;
  String dependentArray[ ];
  Node * temp;
  temp=Head;
  while (temp → ptrTonext != NULL)
    if (determiner_id[temp]=determiner_id[temp]) then counter ++;
    if ((counter == lengthof(temp→determiner_id[]))
    & (counter==lengthof(temp→determiner_id[]))
    then add temp → attributeName in dependantArray[]
  return(dependantArray[])
END

D. Algorithms For Computing Third Normal Form Of Given Relation

ThirdNormalForm(listptr) : This algorithm is used to convert a 2NF relation into the 3NF. listptr is a single linked list which represents a relation in 2NF and all FD’s hold on it. It uses function PrimeKeyCount(listptr), which returns total no of prime attributes used in a relation. It also uses OtherDependent (listptr,Trav) function to find out all dependents of same determiner. This algorithm uses two variable pointer trav and temp and one integer variable counter. It also uses two arrays to store attributes of newly created tables. This algorithm is below and Output of this algorithm is tables generated in 3NF

BEGIN
  1) Find total no of prime attributes in linked list
     Call PrimeKeyCount(listptr)
  2) Node * trav
     trav = Head;
     counter =0;
     while (trav → ptrTonext != NULL)
       if (trav → keyAttribute==0)
         /* it means attribute is non key attributes */
         then find the determiner id[] of trav
       if (determiner_id[ ] of trav == primary key)
         then counter++ :
       else if (counter==keyCount)
         /*means full dependancy exists */
         then Call otherDependent(listptr, trav )
       elseif ((counter < keyCount) & &
                 (counter==sizeof(trav→determiner_id[]))
                 /*means partial dependancy exist*/
                 then Call otherDependent(listptr, trav )
       elseif ((counter > 0 ) ) & &
                 (counter< sizeof(trav→determiner_id[]))
                 then Call otherDependent(listptr, trav )
     Find on which prime arribute both determiner and dependent of trav (current node) depends also add both determiner and dependent in same table of that prime attribute.
     if (counter==0)
       /* means transitive dependancy exist */
       then Call otherDependent(listptr, trav )
     3) Once all dependent of same determiner is found create their table in oracle. Mark table name and table attributes in separate array.
        trav = trav → ptrTonext
     4) Repeat step 1 to step 3
        for unmark table name and attributes.
END

IV. EXPERIMENTATION AND RESULT

To test the performance of DBNorma using these algorithms we have collected 10 examples of relation normalization up to 3NF from various research papers. Table 1 shows description of few of these relations where NOA is number of attributes and NOFD is number of functional dependancies. Table2 shows the decomposition of relations shown in Table 1 into 2NF and 3NF. In Table 1 and 2 FD are separated by semicolon. Table 2 is used for testing the output of our tool DBNorma. These relations can also be helpful to the readers as a reference.

A. Actual Result:

Table 2 shows the expected output of proposed algorithms collected from the above research papers. We have compared output of DBNorma using above algorithms with the expected output from Table 2 and can say that output of
DBNorma using above algorithms is valid and it works in expected manner.

B. Time Analysis:
Table 3 shows the performance i.e. time required to normalize the given relation, where T2NF is time required to bring a relation in 2NF measured in mili-seconds. Similarly T3NF is time required to bring a relation in 3NF.

Plot of number of attributes and time required to bring relation in 2NF and 3NF using DBNorma is shown in Fig.6 and Fig.7 similarly Plot of number of functional dependencies and time required to bring relation in 2NF and 3NF using DBNorma is shown in Fig.8 and Fig.9. From the graphs shown in Fig. 6 to 9, it is observed that the time required to convert a given relation into 2NF is around linear but for 3NF it is observed non-linear as compared to 2NF due to transitive dependency. It can also be concluded that the time required to bring the given relation into 2NF and 3NF does not depend only on number of Attributes or only on number of FDs but depends on both no of FDs and no of Attributes.
### Table I
**Details of Standard Relation Used for Testing**

<table>
<thead>
<tr>
<th>Sr No</th>
<th>Relation Name</th>
<th>Relation Description</th>
<th>NOA</th>
<th>NOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Beer</td>
<td><code>{beer, warehouse, city, brewery, strength, region, quantity}</code></td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDs = <code>{beer → brewery, beer → strength, brewery → city, city → region, beer, warehouse → quantity}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDs = <code>{A → BC, E → AD, G → AEJK, GH → FI, K → AL, J → K}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Property</td>
<td><code>(A, B, C, D, E, F, G, H, I)</code></td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDs = <code>{AB → CD, A → E, B → FGH, H → I, A.C → BFGDHI, B.C → ED}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AB</td>
<td><code>{A, B, C, D, E, F, G, H}</code></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDs = <code>{AB → CEGFH, A → D, F → G, BF → H, BC → ADEFG, BCF → ADE}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Emp</td>
<td><code>(emp_id, emp_name, emp_phone, dept_name, dept_phone, dept_mgrname, skill_id, skill_name, skill_date, skill_lvl)</code></td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FDs = <code>{emp_id → emp_name, emp_id → emp_phone, emp_id → dept_name, dept_name → dept_phone, dept_name → dept_mgrname, skill_id → skill_name, emp_id, skill_id → skill_date, emp_id, skill_id → skill_lvl}</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table II
**Standard Solution**

<table>
<thead>
<tr>
<th>Sr No</th>
<th>Relation Name</th>
<th>2NF</th>
<th>3NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Beer</td>
<td>beer <code>{beer, brewery, strength, city, region}</code>&lt;br&gt;<code>beerwarehouse </code>{beer, warehouse, quantity}<code>&lt;br&gt;</code>city<code> </code>{city, region}<code>&lt;br&gt;</code>beerwarehouse<code> </code>{beer, warehouse, quantity}<code>&lt;br&gt;</code>brewery <code>{brewery, city}</code>&lt;br&gt;<code>city</code> <code>{city, region}</code>&lt;br&gt;<code>beerwarehouse</code> <code>{beer, warehouse, quantity}</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>GH</td>
<td><code>{G, H, F, I}</code>&lt;br&gt;<code>G</code> <code>{G, E, J}</code>&lt;br&gt;<code>J</code> <code>{J, K}</code>&lt;br&gt;<code>K</code> <code>{K, A, L}</code>&lt;br&gt;<code>E</code> <code>{E, A, D}</code>&lt;br&gt;<code>A</code> <code>{A, B, C}</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Property</td>
<td><code>{A, B, C, D, E, F, G, H, I}</code>&lt;br&gt;<code>A</code> <code>{A, E}</code>&lt;br&gt;<code>B</code> <code>{B, F, G, H, I}</code>&lt;br&gt;<code>AB</code> <code>{A, B, C, D}</code>&lt;br&gt;<code>A</code> <code>{A, D}</code>&lt;br&gt;<code>AB</code> <code>{A, B, C, E, F, G, H}</code>&lt;br&gt;<code>A</code> <code>{A, D}</code>&lt;br&gt;<code>AB</code> <code>{A, B, C, E, F, G, H}</code>&lt;br&gt;<code>A</code> <code>{A, D}</code></td>
<td></td>
</tr>
</tbody>
</table>
CONCLUSIONS

This paper represents a new semi-automated approach for relational database normalization. It proves that a relation can be represented with only one singly linked list along with its set of FD’s. As the understanding of linked list is easy, the representation is easy to understand. The definition of 2NF and 3NF algorithms on such a representation becomes efficient since linked list structure can be manipulated efficiently. Algorithms suggested in this paper are very efficient and can be used as an alternative algorithm for DBNorma. However, we will compare our algorithms with other similar algorithms, in the future.

ACKNOWLEDGMENT

We are thankful to management of Dnyanganga College of Engineerimg and research and Vishwakarma Institute of Technology, Pune for their encouragement and whole hearted cooperation during this work.

REFERENCES