World Applied Sciences Journal 5 (6): 686-691, 2008
ISSN 1818-4952

A Novel Parallel Approach for Automatic Database Normalization

[1]Jalal A. Nasiri, [2]Amir H. Bahmani, [3]Mahmoud Naghibzadeh and [3]Hossein Deldari

[1]Communication and Computer Research Center,
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran
[2]Young Researchers Club, Department of Computer Engineering,
Islamic Azad University of Mashhad, Mashhad, Iran
[3]Department of Computer Engineering, Ferdowsi University of Mashhad, Iran

Abstract: As processing power becomes cheaper and more available by using cluster of computers, the needs for parallel algorithms, which can harness these computing potentials, are increasing. Automatic database normalization is an application of parallel algorithms. Normalization is the most exercised technique for the analysis of relational databases. It aims at creating a set of relational tables with minimum data redundancy that preserve consistency and facilitate correct insertion, deletion and modification. Moreover, existing algorithms are usually much time consuming. In this paper, we have proposed a parallel algorithm in EREW PRAM model for Automatic Database Normalization. The proposed algorithm has been examined with MPI and its implementation results on EDM showed that parallel approach reduces the time, efficiently. Exploiting p processors has reduced the time of Automatic Database Normalization to $((n^2+m)/p)+c$ which c is the communication overhead between the processors, m is the number of simple keys and n is the number of determinant keys.

Key words: Parallel algorithm . Automatic normalization . Relational database . Functional dependency

INTRODUCTION

Normalization as a method of producing good relational database designs is a well-understood topic in the relational database field [1]. The goal of normalization is to create a set of relational tables with minimum amount of redundant data that can be consistently and correctly modified. The main goal of any normalization technique is to design a database that avoids redundant information and update anomalies [2]. Parallel database systems are being used nowadays applications to decision support systems [3, 4]. Automatic database normalization is an application of parallel algorithms. The process of normalization was first formalized by E.F.Codd. Normalization is often performed as a series of tests on a relation to determine whether it satisfies or violates the requirements of a given normal form. Three normal forms called first (1NF), second (2NF) and third (3NF) normal forms were initially proposed. An amendment was later added to the third normal form by R. Boyce and E.F. Codd called Boyce–Codd Normal Form (BCNF). The trend of defining other normal forms continued up to eighth normal form. In practice, however, databases are normalized up to and including BCNF. In this paper, we explain minimal functional dependency, 2NF and 3NF parallel algorithm. The first normal form states that every attribute value must be atomic, in the sense that it should not be able to be broken into more than one singleton value. As a result, it is not allowed to have arrays, lists and as such data structures for an attribute value. Each normal form is defined on top of the previous normal form. That is, a table is said to be in 2NF if and only if it is in 1NF and it satisfies further conditions. Except for the 1NF, the other normal forms of our interest rely on Functional Dependencies (FD) among the attributes of a relation. Functional Dependency is a fundamental notion of the Relational Model [5]. Functional dependency is a constraint between two sets of attributes in a relation of a database. Given a relation *R*, a set of attributes *X,* in *R,* is said to functionally determine another attribute *Y*, also in *R*, (written as $X \rightarrow Y$) if and only if each *X* value is associated with at most one *Y* value. That is, given a tuple and the values of the attributes in *X*, one can unequally determine the corresponding value of the *Y* attribute. It is customarily to call *X* the *determinant set* and *Y* the *dependent attribute*.

Given that *X*, *Y* and *Z* are sets of attributes in a relation *R*, one can derive several properties of

Corresponding Author: Jalal A. Nasisi, Communication and Computer Research Center, Department of Computer Engineering, Ferdowsi University of Mashhad, Iran

functional dependencies. Among the most important ones are Armstrong's axioms. These axioms are used in database normalization:

Subset property (Axiom of Reflexivity): If Y is a subset of X, then $X \rightarrow Y$

Augmentation (Axiom of Augmentation): If $X \rightarrow Y$, then $XZ \rightarrow YZ$

Transitivity (Axiom of Transitivity): If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

By repeated application of Armstrong's rules all functional dependencies can be generated. These functional dependencies provide the bases for database normalization [6].

Normalization is a major task in the design of relational databases [7]. Mechanization of the normalization process saves tremendous amount of time and money. Despite its importance, very few algorithms have been developed to be used in the design of commercial automatic normalization tools.

Mathematical normalization algorithm is implemented in [8]. In [9] a comparison of related students' perceptions of different database normalization approaches and the effects on their performance is studied. In [10] a set of stereotypes and tagged values are used to extend the UML metamode. A graph rewrite rule is then obtained to transfer the data model from one normal form to a higher normal form. In [6] a new complete automated relational database normalization method has been presented. It produces the dependency matrix and the directed graph matrix, first. It then proceeds with generating the 2NF, 3NF and BCNF normal forms. All tables are also generated as the procedure proceeds. One more side product of this research is to automatically distinguish one primary key for every final table, which is generated. Depends on [4]'s sequential algorithms, this paper presents parallel algorithms for automatic database normalization except BCNF.

PROPOSED PARALLEL ALGORITHM

In this section, we discuss the details of our proposed parallel algorithm. We first assumed that we have an initialized Dependency Matrix (DM), which is used for representing dependencies [6]. Secondly, we will produce Directed Graph matrix (DG) for

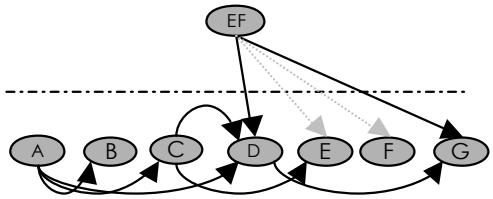

Fig. 1: Graphical representation of dependencies

	A	B	C	D	E	F	G
A	2	1	1	1	0	0	0
C	0	0	2	1	1	0	0
D	0	0	0	2	0	0	1
EF	0	0	0	1	2	2	1

Fig. 2: Initial dependency matrix

Determinant keys by a parallel algorithm. Finally, through pass finding algorithm all possible dependencies will find. Therefore, the dependency matrix will be updated by related parallel algorithm. Figure 1 and 2 shows The DM for Example 1.

Example 1: FDs = {A $\rightarrow$ BCD, C $\rightarrow$ DE, EF $\rightarrow$ DG, D $\rightarrow$ G}

Figure 1 is the graphical representation of the dependencies.

From a dependency graph, the corresponding DM is generated as follows [6]:

- Define matrix DM [n] [m], where
  n = Number of *Determinant Keys*.
  m = Number of *Simple Keys*.
- Suppose that $\beta \subseteq \alpha, \gamma \not\subset \alpha$ and
  $\beta, \gamma \in$ {Simple key set}
  $\alpha, \lambda \in$ {Determinant key set}
- Establish DM elements as follows:
  If $\alpha \rightarrow \beta \Rightarrow$ DM [α] [β] = 2
  If $\alpha \rightarrow \gamma \Rightarrow$ DM [α] [γ] = 1
  Otherwise $\Rightarrow$ DM [α] [γ] = 0,

The DM for Example 1 is shown in Fig. 2 [6].

Producing directed graph matrix for determinant keys: In this phase, we analysis two matrices, one of them is DM and the other one is DG. The sequential algorithm for producing the DG graph follows:

For parallel purpose, we assume that the DM matrix was produced and each slave processor has this matrix that is called Local DM or LDM. The algorithm for producing the DG graph in Fig. 3 shows that the DM does not change during the algorithm, so we send it

```
Directed- Graph-Matrix()
{
 for (i=0; i<n; i++)
 for (k= each attribute that composed determinant key i)
 for (j=0; j<n ; j++) {
       if (DM[j][k]!=0 && DG[j][i]!=-1)
            DG[j][i]=1;
       else DG[j][i]=-1; }
}
```

Fig. 3: Pseudo code of producing the DG graph

```
if rank = 0
for i=1 to p do
send(DG [n][(n/p*(i-1)], processor i)
else
receive (Local DG, processor 0)

for all processors
{
for (z=n/p* (ranki -1) ; z<n/p* ranki; z++)
 for (k=0; k<m; k++)
 if (LDM[z][k]==2)
 for (j=0; j<n ; j++) {
 if (LDM[j][k]!=0 && DG[j][z]!=-1)
 DG[j][z]=1;
 else DG[j][z]=-1; }
}
```

Fig. 4: Pseudocode of parallel producing the DG graph

```
Dependency-closure ()
{
for (i=0; i<n ; i++)
 for(j=0; j<n ; j++)
 if(i!=j && Path[i][j]!=-1) {
 for (k=0; k<m ; k++)
 if(DM[j][k]!=0 && DM[j][k]!=2)
 DM[i][k]=j; }
}
```

Fig. 5: Recognition of dependency closure

to all the slave processors. In addition, each column of the DG matrix is independent from others. Therefore, we can send n/p columns for each slave processor. The parallel algorithm for producing the DG matrix is presented in Fig. 4.

After generating the DG matrix, we turn our attention towards finding all possible paths between all pairs. This new matrix will show all transitive dependencies between determinant keys. There are many such path finding algorithms like Prim, Kruskal and Warshal. Parallel Path finding algorithm works like the algorithm in Fig. 4.

Recognition of dependency closure: After generating the path matrix, the dependency-closure algorithm uses this matrix for updating the dependency matrix. Depending on the dependency-closure algorithm, it is concluded, that path matrix is constant and each DM's row is independent from other rows. Therefore, for parallel purpose, like the pervious parallel algorithm, the master processor sends path matrix and n/p rows of the DM for each slave processor (Fig. 5).

```
if rank = 0
for i=1 to p do
send(global_DM [(n/p*(i-1)][m], processor i)
else
receive (global_DM, processor 0)

for all processors
{
for (z=n/p* (ranki -1) ; z<n/p* ranki; z++)
 for(j=0; j<n ; j++)
 if(z!=j && Path[z][j]!=-1) {
 for (k=0; k<m ; k++)
 if(LDM[j][k]!=0 &&L DM[j][k]!=2)
 global_DM[z][k]=j; }
}
```

Fig. 6: Parallel algorithm for Recognition of dependency closure

```
Circular-Dependency ()
{
 for (i=0; i<n; i++)
 for(j=0; j<m; j++)
 if(global_DM[i][j]!= {0,1,2})
 if(FindOne (i, j, j, n)_&& DM[i][j]==1)
 global_DM[i][j]=1;
}
```

Fig. 7: Replacing transitive dependency with original direct dependency

```
int FindOne (int i, element j, int k, int n)
{
 if(global_DM[j][k]==1 && n>=1) return 0;
 elseif (n<1) return 1;
 else return FindOne (i,global_ DM[i][k], k, n-1);
}
```

Fig. 8: Replacing transitive dependency with original direct dependency

Figure 6 shows Parallel algorithm for recognition of dependency closure. Because of using DM in the next algorithms, a copy is created as global_DM.

Replacing transitive dependency with original direct dependency: To tackle circular dependency, the following c*ircular-dependency* algorithm is designed. This algorithm internally uses the FindOne recursive algorithm.

From Fig. 7 and 8 it can be concluded, the DM represents the initial dependency matrix, so in the parallel algorithm each slave processor has a LDM. Also, we can understand that each DM's column is

```
if rank = 0
 for i=1 to p do
 send(global_DM, global_DM [n][(m/p*(i-1)], processor i)

else
 receive (global_DM, processor 0)
```

Fig. 9: Parallel Algorithm for replacing transitive dependency with original direct dependency

```
for all processors
{
 for (z=n/p* (ranki -1) ; z<n/p* ranki; z++)
 for(j=0; j<m; j++)
 if(global_DM[z][j]!= {0,1,2})
 if(FindOne (z, j, j, n)_&& LDM[z][j]==1)
 global_DM[z][j]=1;
}

int FindOne (int i, element j, int k, int n)
{
 if(global_DM[j][k]==1 && n>=1) return 0;
 elseif (n<1) return 1;
 else return FindOne (i, global_DM[i][k], k, n-1);
}
```

Fig. 10: Parallel algorithm for replacing transitive dependency with original direct dependency

independent from other columns, so the master processor sends m/p columns of global_DM to each slave processor.

Figure 9 and 10 show the parallel Algorithm for replacing transitive dependency with original direct dependency.

The proposed normalization process: It is assumed that the reader is familiar with the definitions of different normal forms. On the other hand, tables of a relational database are assumed to be in 1NF form, to begin with. Our proposed 2NF and 3NF normalization process makes use of both dependency and determinant key transitive dependencies.

Second normal form (2NF): The goal is to discover all partial dependencies, to produce the 2NF form. To do this, the DM is scanned row-by-row (ignoring the primary key row), starting from the first row. If all values of the simple keys that make up the determinant key of the row being scanned are equal to 2 and the values of the corresponding columns of the candidate key are equal to 2, then a partial dependency is found [6]. PKR in Fig. 11 indicates the row of the primary key of global_DM.

Sequential algorithm in Fig. 11 shows that each global_DM's row is independent from other rows, so the master processor sends n/p rows of the global DM and related row to the primary key for each slave processor.

```
for (z=1 ; z<n; z++ && z!=PKR)
{
 Partial=true;
 for(j=0; j<n && global_DM[z][j]==2 ; j++)
 {
 if(global_ DM[PKR][j]==2 && partial!=false){
 Partial=true;
 Else
 { Partial=false; break; }
 }
If(Partial)
 return z;
else
 return -1;
}
```

Fig. 11: Sequential algorithm for producing finding partial dependencies

```
if rank = 0
 for i=1 to p do
 send(global_DM [(n/p*(i-1)][m], processor i)
 send(PKR,DM[PKR], processor i)
else
 receive (global_DM, processor 0)
 receive (PKR, DM[PKR], processor 0)
```

Fig. 12: Parallel algorithm for finding partial dependencies

```
for all processors
{
 for (z=n/p* (ranki -1) ; z<n/p* ranki; z++ & z!=PKR)
 {
 Partial=true;
 for(j=0; j<n && DM[z][j]==2 ; j++)
 if(DM[PKR][j]==2 && X!=false) {
 Partial=true;
 else
 { Partial=false; break; }
 If(Partial && z!=PKR)
 send(z, processor0)
 }
}
```

Fig. 13: Parallel algorithm for finding partial dependencies

Figure 12 and 13 show the Parallel Algorithm for finding partial dependencies.

If any partial dependency was reported, to produce the 2NF form, we have to create new tables (DM) for these partial dependencies [6].

Third normal form (3NF): In order to transform the relations into 3NF, each DM is scanned row-by-row starting from the first row. If a determinant key is encountered, whose dependency is neither partial nor it is wholly dependent on part of the primary key [11] a separate table has to be formed. Of course, if a table is previously formed a duplicate is not generated. This

```
For each DM do:
 for (z=0; z<n; z++)
 {
 Transitivity=true;
 for(j=0; j<n && DM[z][j]==2 ; j++)
 if(DM[PKR][j]==2)
 { Transitivity=false; break;}
 If (Transitivity)
 send(Transitivity,z, processor0)
 }
}
```

Fig. 14: Sequential algorithm for producing 3NF

```
For each DM do:

if rank = 0
 for i=1 to p do
 send(global_DM [(n/p*(i-1)][m], processor i)
 send(PKR,DM[PKR], processor i)
else
 receive (global_DM, processor 0)
 receive (PKR, DM[PKR], processor 0)
```

Fig. 15: Parallel algorithm for producing 3NF

```
for all processors
{
 for (z=n/p* (ranki -1) ; z<n/p* ranki; z++)
 {
 Transitivity=true;
 for(j=0; j<n && DM[z][j]==2 ; j++)
 if(DM[PKR][j]==2)
 { Transitivity=false; break;}
 If (Transitivity)
 send(Transitivity,z, processor0)
 }
}
```

Fig. 16: Parallel algorithm for producing 3NF

new table will include the determinant key and all other attributes, which are transitively, depend on this key. The sequential algorithm for producing 3NF follows (Fig. 14):

In Fig.15 and 16, database is normalized up to 3NF with parallel algorithm.

RESULTS

In this section, we show data related to efficiency of proposed parallel algorithm for automatic database normalization on EDM. We have used a Beowulf cluster with 8 nodes and Ethernet 10/100 network infrastructure. The algorithm has been implemented with LAM/MPI ver. 7.1.14. We will examine the efficiency of our MPI program and the parallel speed of proposed algorithm

EDM: EDM is a dependency matrix with 100 rows and 9000 columns. We use this DM for sequential and our parallel algorithms and we will evaluate our parallel approach by using it.

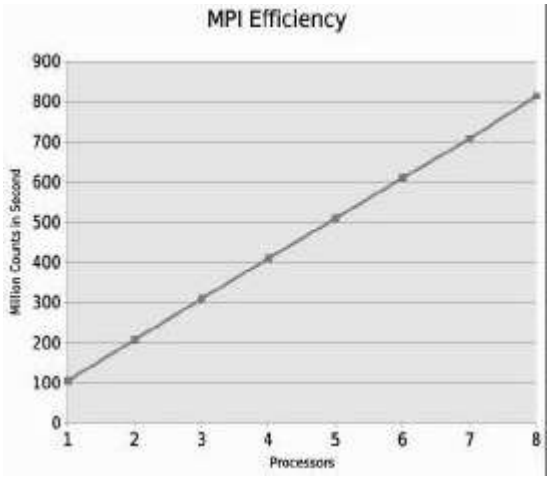

Fig. 17: MPI Efficiency of a simple arithmetic program

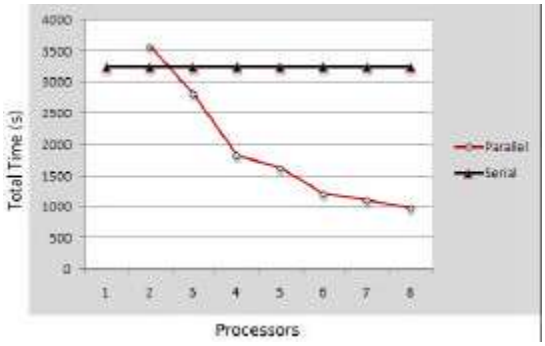

Fig. 18: Speed of parallel algorithm against serial algorithm

MPI efficiency: In order to examine the efficiency of a MPI program, a small arithmetic operations program is written. The overall speed of all processors is determined by taking the amount tasks which should be done and dividing the total execution time. This is illustrated in Fig. 17. This figure shows that the ratio of the growth in number of processors to the growth in speed is constant. This indicates that the MPI process is efficient.

Speed of parallel algorithm: One of the best performance evaluations in parallel programming is the speed comparison of parallel and serial implementations. In Fig. 18, the speed of parallel and serial implementations of recognition of dependency closure algorithm is depicted on EDM. We have

changed the number of processors from 1 to 8 increasingly. The total time for the serial algorithm is constant for any value of thr number of processors.

Surprisingly, when we increase the number of processors to 2 processors, the time gets worth. This is because of communication costs. As the number of processors increases, the total time of parallel implementation decreases.

CONCLUSION

In this study we have proposed a parallel algorithm for automatic database normalization. The process is based on the generation of dependency matrix, directed graph matrix and determinant key transitive dependency matrix. The details of the methods for 2NF, 3NF are discussed. Implementation results MPI and a cluster eight processors indicate a considerable reduction in time for automatic database normalization.

ACKNOWLEDGEMENTS

This work was supported by Communications and Computer Research Center Ferdowsi University, Ministry of Information Technology, Mashhad, Iran.

REFERENCES

1. Arenas, M. and L. Libkin, 2005. An Information-Theoretic Approach to Normal Forms for Relational and XML Data. Journal of the ACM (JACM), 52 (2): 246-283.
2. Kolahi, S., 2007. Dependency-Preserving Normalization of Relational and XML Data. J. Computer Sys. Sci., 73 (4): 636-647.
3. Ameet, S. Talwadker, 2003. Survey of performance issues in parallel database systems. J. Computer Syst. Sci. Colleges, 18 (6): 5-9.
4. David Taniar and J. Wenny Rahayu, 2002. Parallel database sorting, Information science-Application: An International Journal, 146 (1-4): 171-219.
5. Mora, A., M. Enciso, P. Cordero and IP de Guzman, 2003. An efficient preprocessing transformation for functional dependencies sets based on the substitution paradigm, CAEPIA2003, pp: 136-146.
6. Amir H. Bahmani, M Naghibzadeh and B Bahmani, 2008. Automatic Database Normalization and Primary Key Generation. IEEE CCECE/CCGEI, pp: 11-16.
7. Du, H. and L. Wery, 1999. A normalization tool for relational database designers. J. Network Computer Appl., 22 (4): 215-232.
8. Yazici, A. and Z. Karakaya, 2006. Normalizing Relational Database Schemas Using Mathematica, LNCS, Springer- Verlag, 3992: 375-382.
9. Kung, H. and T. Case, Traditional and alternative database normalization techniques: Their impacts on IS/IT students' perceptions and performance. Intl. J. Inform. Technol. Edu., 1 (1): 53-76.
10. Akehurst, D.H., B. Bordbar, P.J. Rodgers and N.T.G. Dalgliesh, 2002. Automatic Normalization via Metamodelling, ASE 2002 Workshop on Declarative Meta Programming to Support Software Development.
11. Connoly, Thomas, 2005. Carolyn Begg: Database Systems. A Practical Approach to Design, 3rd Edn., Implementation and Management, Pearson Education.