Sindh Univ. Res. Jour. (Sci. Ser.) Vol. 50 (3D) 143-147 (2018)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

A Tool for Query Normalization and Elimination of Redundancy

M. S. VIGHIO[++], T. J. KHANZADA, M. KUMAR

Department of Information Technology, Quaid-e-Awam University of Engineering, Science & Technology, Nawabshah, Sindh, Pakistan

Received 10th June 2018 and Revised 15th September 2018

Abstract: Redundancy and complexity in user queries reduce the performance of database software. A remedy is to simplify queries before their processing. This paper presents a tool that automatically simplifies queries in the initial phase of their processing. Simplification process is implemented using idempotent and equivalence rules for simplification. Also, the tool automatically performs normalization of complex and redundant queries, and as a result provides a simplified query along with the cost incurred on simplification process. The paper also presents a detailed cost comparison of executing redundant and non-redundant queries. Experimental results show that the queries involving redundancy and complexity take more time and consume more resources as compared to executing non-redundant queries.

Keywords: Relational DBMS, Normalization, Structured Query Language.

1. INTRODUCTION

Database (DBMS) are complex programs which allow users to enter queries, translate these queries in a format required for data access, and produce meaningful results efficiently i.e., results which take less processing time and consume fewer resources (Connolly *et al.,* 2004). Developing efficient DBMS software has always been a challenge for software developers because there are many factors which influence on the performance of DBMS software. These factors include design approaches, optimization strategies, redundancy, response time and throughput. The DBMS developers have to make intelligent decision making in order to develop better software. This decision making involves which design approaches to use for designing the DBMS software, which optimization strategies to implement, how to control the redundancy in data and queries, how to decrease the response time, and how to increase throughput. Among all these factors, redundancy and complexity is the direct result of users' interaction with the DBMS software. Redundancy refers to the unnecessary or duplicate information. Redundancy is a source of risk and creates unpredictable problems for the DBMS software at the time of executing queries (Chaudhuri *et al.*, 2004). Redundancy may occur at the level of data and queries. Redundancy in data is a much researched topic in the literature, see for example (Darwen *et al.,* 2012, Vincent, 1995, Date, 2003). On the other hand, this paper focuses on redundancy in queries and also investigates its effects on the performance of relational database systems. As a result of redundancy, queries become more complex and put extra burden of simplification on DBMS software. Thus, the overall performance of the system is reduced (Vardi, 1982, Papadimitriou *et al.,* 1997). Redundancy and complexity result in the following:

- Wastage of space
- Increase in response time
- Maximum resource consumption
- Search may fail due to complexity
- Overall performance may be reduced.

This indicates that redundancy and complexity are big problems for database systems. Moreover, eliminating them may even be a bigger problem which requires additional time and resources for simplification (Jarke *et al.*, 1984). However, this problem can be resolved by developing better algorithms and techniques. The main objective of this paper is to highlight the effects of redundancy and complexity in user queries on the performance of relational database systems. For this purpose, we have developed a tool that automatically detects and eliminates redundancy from input SQL queries and produces a simplified query along with the statistics of cost incurred on simplification process. The tool implements an algorithm and mathematical rules (idempotent and equivalence rules) to normalize queries in the initial phase of processing. As a result, an equivalent query is achieved that can be useful for further processing and optimization more efficiently. The paper also presents a detailed comparison of execution time and consumption of resources when redundant and complex queries are executed with and without the tool support.

Related Work:

Query processing and optimization has been discussed in much detail in the literature (Antoshenkov *et. al*, 1996, Kossmann, 2000, Ceri *et al.*, 1985, Molina

[++] Corresponding Author: Email: saleem.vighio@quest.edu.pk *et al.,* 2008). However, query normalization and elimination of redundancy still requires much attention as it directly effects on the performance of the database systems. In (Chan *et al.*, 1999), the importance of query complexity as a determinant of user performance when retrieving information from a database has been examined. Using the classification of simple versus complex queries, the authors in (Chan *et al.*, 1999) found that the complexity in user queries significantly affects the performance of the database. (Bendre, 2015) has developed a tool that translates relational algebraic statements to Structured Query Language (SQL). By extending the work presented in (Bendre, 2015), the authors in (Memon *et al.*, 2015) have implemented few optimization strategies in their tool. The implementation ideas presented in (Bendre, 2015) and (Memon *et al.*, 2015) remained helpful for developing our query simplification tool. (Ozsu *et al.*, 2007) and (Elmasri *et al.*, 1999) have provided a detailed introduction of query normalization and elimination of redundancy using idempotent and equivalence rules. However, cost of simplification and effects of redundancy and complexity on the performance of DBMS have not been discussed. To fill in that gap, the tool presented in this paper implements the techniques presented in (Ozsu *et al.,* 2007) and (Elmasri *et al.,* 1999) in order to highlight the adverse effects of redundancy and complexity on the performance of DBMS software. The tool is expected to help in understanding query processing concepts and elimination of redundancy in a better way. Further, the tool may also be helpful for the development of efficient DBMS software with a particular focus on query simplification.

The rest of the paper is organized as follows: Section 2 presents methodology followed for the development of the tool. Section 3 gives tool details and elaborates the actual procedure for performing simplification of redundant and complex queries. Section 4 gives detailed comparison of performance results achieved after executing redundant and non-redundant queries with and without the tool support. Section 5 gives conclusive remarks and suggestions for the future work.

2. METHODOLOGY

The tool is implemented using Visual Studio C# language and allows connectivity with SQL Server databases. As it is evident from the literature that the WHERE clause of any SQL query is considered as the most complex part of the query as it may involve redundant predicates (Ozsu *et al.*, 2007). Therefore, we have also focused on the WHERE clause of the input SQL query for simplification purpose. Query simplification has been performed by applying both equivalence and idempotent rules together, because only solely combination of these rules allows to detect and eliminate redundancy and complexity of queries spontaneously.

It is worth mentioning that these standard rules are implemented with few minor simplification because empirical implementation in programming language is entirely tricky. The complete implementation of the tool is available at (Vighio *et al*, 2017a).

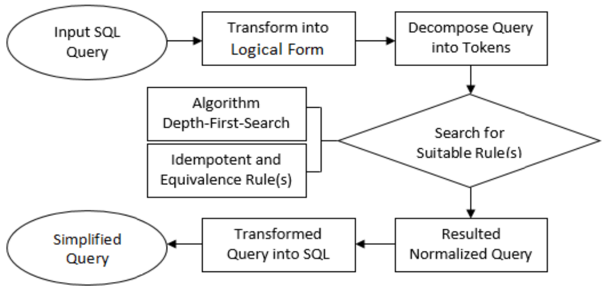

Input SQL Query
Transform into Logical Form
Decompose Query into Tokens
Algorithm Depth-First-Search
Idempotent and Equivalence Rule(s)
Search for Suitable Rule(s)
Simplified Query
Transformed Query into SQL
Resulted Normalized Query

Fig. 1. System Model

(Fig 1), the tool works in the following seven steps:

Step 1: Input query in SQL from the user,
Step 2: Transformation of SQL query in internal representation i.e., normal (logical) form suitable for applying simplification rules,
Step 3: Decomposing query into tokens,
Step 4: Replacing tokens with predicates (e.g. POSITION = PLAYER as P1),
Step 5: Applying Depth-First-Search (DFS) technique for searching suitable simplification rule(s),
Step 7: Producing simplified query in SQL format along with the statistics of cost incurred on the simplification process.

The idea of implementing DFS algorithm for finding suitable simplification rule(s) is based on its property of linear memory requirement with respect to the search space (Cormen *et al.*, 2001). Under this strategy, the search starts from the root and explores each rule along each branch before backtracking, see **(Fig 2).** The rules implemented are idempotent and equivalence for simplification as given in **(Table 1 and 2)** below and provided in (Ozsu *et al.*, 2007).

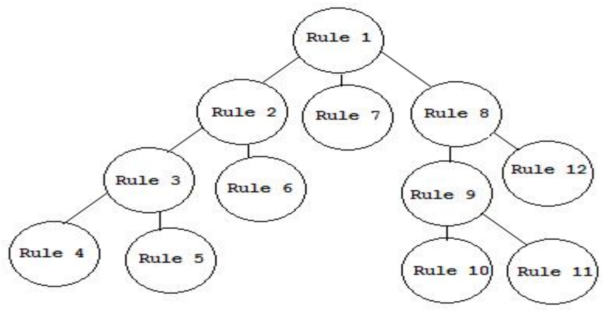


Fig. 2. Order of Visiting Rules Using DFS Algorithm

Table–1: Equivalence rules for simplification

1	$P1 \wedge P2 \leftrightarrow P1 \wedge P2$
2	$P1 \vee P2 \leftrightarrow P1 \vee P2$
3	$P1 \wedge (P2 \wedge P3) \leftrightarrow (P1 \wedge P2) \wedge P3$
4	$P1 \vee (P2 \vee P3) \leftrightarrow (P1 \vee P2) \vee P3$
5	$P1 \wedge (P2 \vee P3) \leftrightarrow (P1 \wedge P2) \vee (P1 \wedge P2)$
6	$P1 \vee (P2 \wedge P3) \leftrightarrow (P1 \vee P2) \wedge (P1 \vee P2)$
7	$\neg(P1 \wedge P2) \leftrightarrow \neg P1 \vee \neg P2$
8	$\neg(P1 \vee P2) \leftrightarrow \neg P1 \wedge \neg P2$
9	$\neg(\neg P) \leftrightarrow P$

Table–2: Idempotent rules for simplification

1	$p \wedge p \leftrightarrow p$
2	$p \vee p \leftrightarrow p$
3	$p \wedge \text{true} \leftrightarrow p$
4	$p \vee \text{false} \leftrightarrow p$
5	$p \wedge \text{false} \leftrightarrow \text{false}$
6	$p \vee \text{true} \leftrightarrow \text{true}$
7	$p \wedge \neg p \leftrightarrow \text{false}$
8	$p \vee \neg p \leftrightarrow \text{true}$
9	$p \wedge (p_1 \vee p_2) \leftrightarrow p_1$
10	$p \vee (p_1 \wedge p_2) \leftrightarrow p_1$

3. TOOL DETAILS

The tool is implemented in Visual Studio C# language. The key features of the tool include:

- allowing connectivity with SQL server databases,
- correctly verifying syntax and semantics of each input query,
- generating complete trace of applying simplification rules, and
- producing simplified SQL query along with statistics of time and resources used in the simplification process.

Implementation details are provided as an extension of our previous paper (Vighio *et al.,* 2017b) for clear understanding of the working of the tool. Simplification rules are implemented using an array string rules []= newstring[] {"!(!(A))=A","AVA=A","A^A=A","A^FALSE=FALSE",…};

Since users are allowed to type queries in all formats, in order to maintain the consistency of the text user query is converted to lower case

```
query = query.ToLower();
```

where "query" is a string variable used to hold user query typed in rich text box

```
string query = richTextBox1.Text;
```

The entered query is checked to contain the WHERE clause and if it is found, the query is transformed into normal (logical) form and stored in **Tra_query** variable; otherwise, the same input query is printed as final output as it does not requires simplification. if (query. Contains("where"))

```
    {
  //query is transformed into
    relational algebra and stored
   in Tra_query variable.
    }
else
    {
      //print query
    }
```

Once the query is transformed into normal (logical) form, simplification rules are applied on query using a loop. The loop continues till all the rules are applied and the query is simplified.

```
for(int i = 0; i<Rules.Length;i++)
{
 string lhs =
   rules[r].Substring(0,rules[r].IndexOf("="));
 string rhs = rules[r].Substring(rules[r].IndexOf("=") +
  1);
if (Tra_query.Contains(lhs))
   {
     Tra_query=Tra_query.Replace(lhs,rhs);
   }
}
```

Inside the loop body each rule is divide into two parts i.e., before and after the **"="** sign and stored in **lhs** and **rhs** variables respectively. The query expression is searched for **lhs** expression and if it is found it is replaced with equivalent **rhs** expression as provided in the list of rules. This process continues till no further rules are applicable. In that case, the final query is returned as a simplified query along with the statistics of CPU execution time and memory usage. For that a built-in **Performance Counter()** function is used with the following commands:

```
Performance Counter cpu Counter = new Performance
Counter();
Performance Counter ram Counter = new Performance
Counter();
```

The initial and final CPU execution time and RAM values are set as delimiters. After the simplification process completes, the final values are subtracted from the initial values in order to obtain the cost incurred on the simplification process. As provided in (Vighio *et al.,* 2017), the procedure for translating SQL query in normal form and eliminating redundancy is explained with the help of the following query example that finds locations of students whose name is HARRIS.

```
   Select Location
   From Student
Where (NOT (Location like 'New York') and (Location
like 'New York' or Location like 'Texas')
   and not (Location like 'Texas')) or
   Name like 'Harris'
```

Redundancy and complexity can be seen in the WHERE clause of the query (repeating same statements with the use of multiple AND, OR, and NOT operators). Once the query has been entered and simplification process is started, the tool finds predicates from the WHERE clause of the query and converts query in logical form connected with AND, OR, or NOT operators as provided in the query.

Based on the above example query, following predicates are found:

P1 Location like 'New York'

P2 Location like 'Texas'

P3 Name like 'Harris'

Further, based on the predicates found, following query qualification is achieved:

(¬ p1 ∧ (p1 ∨ p2) ∧ ¬ p2) ∨ p3

The tool scans the qualification expression, finds and applies suitable simplification rule(s) as provided in Table 1 and 2 using DFS search strategy, and produces a simplified query such that no further simplification is possible.

The simplification proceeds as follows: By applying rule 5 of Table 1 the new qualification achieved is:

(¬p1∧ ((p1 ∧ ¬p2) ∨ (p2 ∧ ¬p2))) ∨ p3

Further, the new expression is scanned again and simplification rule 3 of Table 1 is applied to obtain:

(¬p1 ∧ p1 ∧ ¬p2) ∨ (¬p1∧ p2 ∧ ¬p2) ∨ p3

By Applying rule 7 of Table 2, the new expression is:

(false ∧ ¬p2) ∨ (¬p1 ∧ false) ∨ p3

By Applying rule 5 of Table 2, we obtain:

(false ∨ false) ∨ p3

By applying rule 4 of Table 2, the expression produced is:

p3

Since no further simplification rule is found applicable, the tool converts the final expression to its equivalent SQL form and produces the simplified query as given below:

Select Location
From Student
Where Name like 'Harris'

Fig. 3 shows the user interface of the tool along with input query in SQL form, trace of applying rules, and generating final query in SQL form. Furthermore, the tool also shows the cost of simplification process.

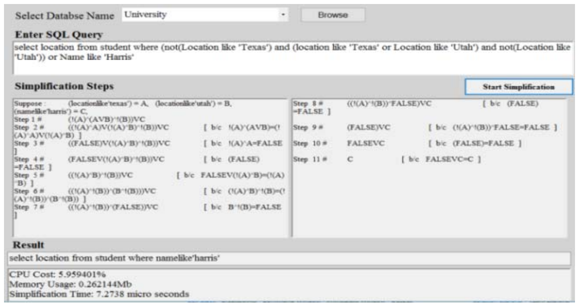
4. <u>EXPERIMENTAL RESULT</u>

Using this tool support, the cost of executing redundant and non-redundant queries can be measured separately in terms of CPU time and memory usage. Furthermore, the time it takes to eliminate redundancy has also been calculated. The key objective is to highlight the effects of redundancy and complexity in user queries on the performance of database. The tests are performed on Windows 10 Home Single Language, 64-Bit Operating System, Intel (R) Core(TM)i5-3230M CPU, 2.60 GHz. Microsoft Visual Studio 2012 version, and connectivity has been provided with Microsoft SQL Server 2012. For experimental results, we have created University database containing STUDENT relation as shown in **(Table 3).**

As shown in **(Fig. 3),** query for finding student locations whose name is Harris has been simplified at CPU cost of 5.96%, memory cost of 0.26Mb, and total simplification time of 7.27 micro seconds. In total 11 steps are used including also translating WHERE clause to normal form and using the simplification rules.

Table – 3: Student relation

StId	Name	Class	Location
17CS25	Harris	2	Texas
17CS21	Julian	1	Texas
17CS22	Robert	2	Utah
17CS23	Rehana	1	Utah
17CS24	Joseph	2	Indiana
17CS34	Harris	2	New York



```
select location from student where (not(Location like 'Texas') and (location like 'Texas' or Location like 'Utah') and not(Location like 'Utah')) or Name like 'Harris'
```

```
Suppose :   (locationlike'texas') = A,  (locationlike'utah') = B,
(namelike'harris') = C,
Step 1 #    (!(A)^(AVB)^!(B))VC
Step 2 #    ((!(A)^A)V(!(A)^B)^!(B))VC         [ b/c  !(A)^(AVB)=(!(A)^A)V(!(A)^B) ]
Step 3 #    ((FALSE)V(!(A)^B)^!(B))VC          [ b/c  !(A)^A=FALSE ]
Step 4 #    (FALSEV(!(A)^B)^!(B))VC            [ b/c  (FALSE)=FALSE ]
Step 5 #    ((!(A)^B)^!(B))VC                  [ b/c  FALSEV(!(A)^B)=(!(A)^B) ]
Step 6 #    ((!(A)^!(B))^(B^!(B)))VC           [ b/c  (!(A)^B)^!(B)=(!(A)^!(B))^(B^!(B)) ]
Step 7 #    ((!(A)^!(B))^(FALSE))VC            [ b/c  B^!(B)=FALSE ]
Step 8 #    ((!(A)^!(B))^FALSE)VC              [ b/c  (FALSE)=FALSE ]
Step 9 #    (FALSE)VC                          [ b/c  (!(A)^!(B))^FALSE=FALSE ]
Step 10 #   FALSEVC                            [ b/c  (FALSE)=FALSE ]
Step 11 #   C                                  [ b/c  FALSEVC=C ]
```

```
select location from student where namelike'harris'
```

```
CPU Cost: 5.959401%
Memory Usage: 0.262144Mb
Simplification Time: 7.2738 micro seconds
```

Fig. 3. Tool Interface with Query Execution

The non-redundant version of the same query has also been experimented and in that case, the cost of CPU was 3.06%, memory usage remained 0.07Mb, and 0.28 micro seconds were used for translating SQL query to normal form and translating it back to SQL form after no simplification was found applicable.

Furthermore, the experiments have also been performed in SQL server environment directly both for redundant and non-redundant queries; the results of which are shown in **(Table 4).**

Parameter	**Average Cost**		**Improvement in %**
	Redundant Query	**Simplified Query**	
Query profile statistics			
Number of SELECT statements	1	1	0
Rows returned by SELECT statements	2	2	0
Network statistics			
Bytes sent from client	566	192	374
Bytes received from server	1214	1031	138
Time statistics			
Client processing time	23	13	10
Total execution time	67	14	53
Wait time on server replies	44	1	43

As it can be seen from Table 4, there is a clear difference in average cost of executing redundant and non-redundant queries in terms of processing time and resources. The cost of executing redundant queries can be even more higher if complexity and redundancy is increased.

5. CONCLUSION

In this paper, we presented a tool that checks redundancy and complexity in SQL queries and automatically provides simplified query along with the statistics of CPU and memory incurred on the simplification process. The tool has been tested to provide:

- connectivity with SQL server databases,
- correctly verifying syntax and semantics of each input query,
- generating complete trace of applying simplification rules, and
- producing non-redundant SQL query along with statistics of CPU and memory used in the simplification process.

Based on the experimental results, it is concluded that the execution of redundant and complex queries require more processing time and resources as compared to simple and non-redundant queries. Furthermore, redundancy and complexity in SQL queries puts extra burden of simplification on DBMS software and ultimately effects on the overall performance of the DBMS. In order to develop better DBMS software, it is suggested and complexity in user queries must be eliminated in initial phase.

REFERENCES:

Antoshenkov G. and M. Ziauddin (1996). “Query processing and optimization in oracle rdb,” The VLDB Journal, vol. 5, no. 4, pp. 229–237, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1007/s007780050026

Bendre M. (2015). “Relational algebra translator (rat) [accessed: Oct 10, 2015],” http://www.slinfo.una.ac.cr/rat/rat.html.

Ceri S. and G. Gottlob (1985) “Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries,” *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 324–345, Apr. 1985. [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.232223

Chan H. C., C. Tan B. and K.-K. Wei (1999). “Three important determinants of user performance for database retrieval,” *Int. J. Hum.-Comput. Stud.*, vol. 51, no. 5, pp. 895–918, Nov. 1999. [Online]. Available: http://dx.doi.org/10.1006/ijhc.1999.0272

Darwen Hugh, Data C. J. and R. Fagin (2012), A Normal Form for Preventing Redundant Tuples in Relational Databases, *In Proceedings of the 15th International Conference on Database Theory (ICDT'12)*, is bn = 978-1-4503-0791-8, 114-126, Publisher ACM, USA.

Date, C. J. (2003), An Introduction to Database Systems, isbn = 0321197844, edition = 8, publisher = Addison-Wesley Longman Publishing Co., Inc., address = Boston, MA, USA.

Elmasri R. A. and S. B. Navathe (1999). Fundamentals of Database Systems, 3rd ed., C. Shanklin, Ed. Boston, MA, USA: Addison-Wesley Longman .

Jarke M. and J. Koch (1984). “Query optimization in database systems,” ACM Comput. Surv., vol. 16, no. 2, pp. 111–152, June. 1984. [Online]. Available: http://doi.acm.org/10.1145/356924.356928.

Kossmann D. (2000). “The state of the art in distributed query processing,” ACM Comput. Survey. vol. 32, no. 4, 422–469, [Online]. Available: http://doi.acm.org/10.1145/371578.371598.

Memon N., M. S. Vighio S. Nizamani N. A. Memon A. R. Memon and U. R. Shaikh (2015). “Analysis of query processing and optimization,” *Bahria University Journal of Information & Communications Technology (BU-JICT)*, vol. 8, no. 1, 14–20.

Molina H. G., J. D. Ullman and J. Widom (2008), Database Systems: The Complete Book, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press.

Ozsu M. T. (2007). Principles of Distributed Database Systems, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press,

Papadimitriou C. H. and M. Yannakakis (1997). “On the complexity of database queries,” in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ser. PODS ’97*. New York, NY, USA: ACM, 1997, pp. 12– [Online]. Available: http://doi.acm.org/10.1145/263661.263664

Vincent M. W. (1995), "Redundancy Elimination and a New Normal Form for Relational Database Design", *In proceeding of Semantics in Databases*, Selected Papers from a Workshop, Prague, Czech Republic, 1995, 247-264.

Vighio M. S., T. J, Khanzada M. Kumar (2017a). “Query Simplification Tool. [Online]. Available: https://sites.google.com/a/quest.edu.pk/saleem/tool

Vighio M. S., M. Kumar (2017b). “Analysis of the Effects of Redundancy on the Performance of Relational Database Systems, In proceedings of the 3rd International Conference on Engineering, Technologies and Social Sciences (ICETSS'17), held AIT, Bangkok.