Optimization of SuperSQL Execution by Query Decomposition

Ria Mae Borromeo
Faculty of Information and Communication Studies,
University of the Philippines Open University
Los Baños, Laguna, Philippines
riamae.borromeo@upou.edu.ph

Motomichi Toyama
Department of Information and Computer Science
Faculty of Science and Technology, Keio University
Yokohama, Japan
toyama@ics.keio.ac.jp

***Abstract*— SuperSQL is an extension of SQL that allows formatting and publishing of database contents into various kinds of application data directly as a result of a query. Possible application data output formats include, but are not limited to, HTML, PDF, XML, XLS, and Ajax-driven pages. Originally, the SuperSQL query is directly converted into a single SQL query. In some query cases, this procedure returns a large intermediate table, which typically require a long execution time and consume a lot of memory. To minimize the execution time and memory consumption, query decomposition, the process of dividing a query into sub queries whose result sets' union is equivalent to the result set of the original query, was applied to SuperSQL query processing. Experiments show that for several query cases, there is significant reduction in SuperSQL query execution time and memory consumption.**

Keywords- database applications; database publishing; query processing; query optimization.

I. INTRODUCTION

Relational databases are here to stay. Although new database technologies continue to arise and gain popularity, relational databases are far from being obsolete [2]. SQL, the standard query language used for managing and querying relational databases, returns query results to the user in the form of a flat table. To translate this output into a specific application, report writers have been used. However, there is no standard language that covers the specification of such translations into various types of application data [10].

SuperSQL is an extension of SQL that has the capability of generating various kinds of application data directly as a result of a query. Its syntax is similar to SQL with additional formatting capabilities. Currently, it is mainly used to easily create data-driven web pages and applications. Figure 1 is a sample SuperSQL query and Figure 2 is a sample SuperSQL query output. This is a sample page of a bookstore website that lists from left-to-right its records of all books, authors and publishers.

A SuperSQL query is converted into a single SQL query, which is processed by the Database Management System (DBMS). Consequently, one intermediate table is returned and processed by SuperSQL. Depending on the number and size of tables to be accessed, the size of intermediate table can become large even though the actual number of tuples in the desired output is small.

```
GENERATE HTML [
      {"Books"![b.title]!
      {"Authors"![a.name]!},
      {"Publishers"![p.publisher]!}
  ]!
FROM books b, publishers p, authors a
```

Figure 1. Sample SuperSQL Query

Figure 2. Sample SuperSQL Query Output

The SuperSQL query in Figure 1 is converted into the SQL query in Table I a). If the books, authors and publishers tables have 550, 25 and 20 tuples respectively, the resulting SQL query takes the Cartesian product of the three tables. The intermediate table size would have 275,000 tuples. However, in the sample query, there is no relationship between the tables. The desired output only consists of a list of the contents of each table, displayed from left-to-right. Therefore, the desired number of tuples is only 595, which is the sum of the tuples in each table.

Initial experiments using the current SuperSQL version showed that as the intermediate table size increases, the execution time and memory consumption of a SuperSQL query also increase. Thus we aim to reduce the intermediate table returned by the DBMS to reduce execution time and memory consumption of executing SuperSQL queries.

In this study, the concept of query decomposition was

 ISBN: 978-1-61208-247-9

applied to SuperSQL queries. Query decomposition is the process of finding several queries wherein the union of the result sets is equivalent to the original result set. Having several queries instead of one eliminates unnecessary Cartesian product and join operations thus reducing the intermediate table size and consequently reducing execution time and memory consumption.

TABLE I. COMPARISON OF RESULTING SQL QUERIES WITH AND WITHOUT QUERY DECOMPOSITION

a) without Decomposition	b) with Decomposition
SELECT DISTINCT b.title, a.name, p.publisher FROM books b, authors a, publishers p	1) SELECT title FROM books; 2) SELECT name FROM authors; 3) SELECT publisher FROM publishers;

If query decomposition is applied to the example, instead of being converted to the SuperSQL query in Table I a), the query is converted into Table I b). The results of the individual queries are combined later on to produce the desired output.

The query decomposition algorithm models the SuperSQL query as an undirected graph where the query's attributes are represented as nodes and the attributes' relationship with each other are represented as edges. The number of connected components in the resulting graph represents the number of possible divisions for the query. The original query is divided into the number of connected components and the resulting queries are executed individually. The results are combined later on to produce the desired output.

II. RELATED WORKS

Query Decomposition is the process of dividing a query into several queries wherein the union of the result sets of the divided queries is equivalent to the result set of the original query. It is widely used in systems wherein a query contains data from different database servers. In such applications, a query is decomposed based on the mapping of data attributes to its sources.

In 2008, Le applied query decomposition to access data from various data repositories [4]. However, since creation of mappings is an expensive process, an input query is automatically decomposed into sub queries without pre-defined mappings. The algorithm traverses from the bottom to the top of a schema tree depending on the structure of local schemas. Compared to top-down approaches, the algorithm can reduce the time for creating the divided queries for local schemas.

Also in 2008, Bonchi applied query decomposition to a document retrieval system [3]. A query is decomposed into a small set of queries whose union of resulting documents corresponds approximately to that of the original query. The goal is to assist users in finding the information they are looking for, by providing them a suitable set of queries as part of the results of their queries.

The problem was instantiated a specific variant of a set cover problem where an efficient greedy algorithm and a clustering algorithm were designed.

In this study, we applied query decomposition to SuperSQL queries. Instead of converting a SuperSQL query into one SQL query, when possible, it is converted into several queries in order to minimize the intermediate table output and reduce SuperSQL execution time and memory consumption.

III. SUPERSQL

SuperSQL extends the functionality of an SQL query by using the Target Form Expression (TFE) processing system. TFE was formerly developed to generate ordinary reports from the contents of relational databases [9]. Currently, it has been extended to generate application data directly from database queries.

A. Syntax

TABLE II. COMPARISON OF SQL AND SUPERSQL SYNTAX

SQL Syntax	SuperSQL Syntax
SELECT <attribute list>	GENERATE <media> <Target Form Expression (TFE)>
FROM <tables>	FROM <tables>
WHERE <condition>	WHERE <condition>

The syntax of SuperSQL is similar to SQL. The major difference is the introduction of the GENERATE keyword, which allows the user to specify the target application data output. Table II shows the syntax comparison of an SQL and a SuperSQL query. Instead of the SELECT keyword, a SuperSQL query starts with the GENERATE keyword. Moreover, the attribute list in SQL is specified in the TFE. As of the latest version of SuperSQL, the following are the possible target media: Actiview [5], Excel, Flash, HTML, HTML5, LaTeX, LDAP, PDF, VRML (X3D) and XML.

B. Target Form Expression

While an ordinary target list in SQL is a comma-separated list of attributes, TFE uses new operators called connectors and repeaters to specify the structure of the document to be generated by the query. For example, in a table element of a webpage, the columns of the table are associated to the first dimension while the rows are associated to the second dimension and the hyperlinks are associated to the third dimension. Binary operators represented by a comma (,), an exclamation point (!) and a percent (%) symbol are used as the connectors of the first three dimensions. They connect objects generated by their operands horizontally, vertically and in the depth direction, respectively [10]. This is illustrated in Figure 3.

 ISBN: 978-1-61208-247-9

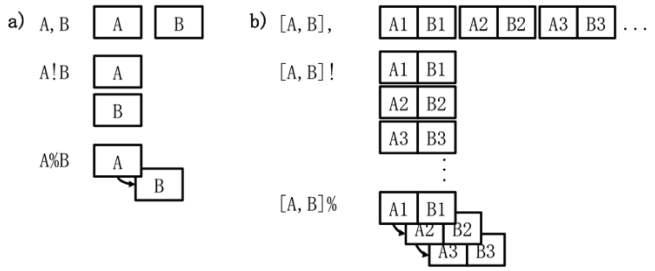

Figure 3. a) Connectors and b) Repeaters

A pair of square brackets ([ ]) followed by any of the above connectors is a repeater for that dimension. It will connect multiple instances in its associated dimension. For example, *[publishers.publisher, books.title, authors.name]!* will connect a publisher, a book title and an author into horizontal direction and connect them vertically as long as there are tuples in the query result.

C. SuperSQL Architecture

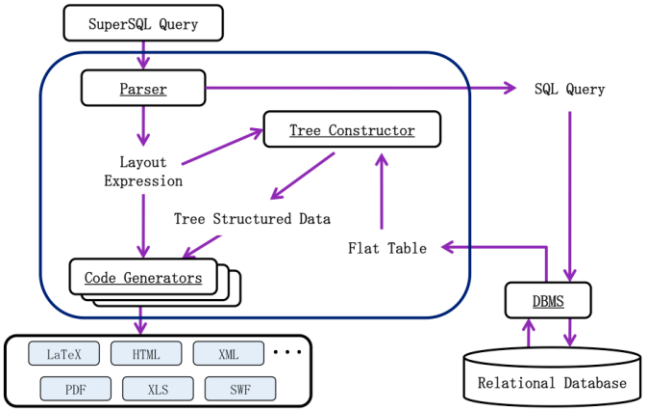

Figure 4. SuperSQL System Architecture

As can be seen in Figure 4, the SuperSQL system has four major components: the Parser, the Tree Constructor, the DBMS and the Code Generators. The Parser is responsible for detecting syntax errors. It extracts the underlying SQL syntax components such as the SELECT, FROM and WHERE clauses. Then an SQL query is created and sent to the DBMS.

The Parser also extracts the layout expression. The layout expression is composed of two parts. The first part is called the schema, which is a tree-structured representation of the layout of the attributes in the query. The second part is called the data formatting, which contains layout information specific to the application data. For example, if the desired application data output is HTML, the data formatting would contain information about HTML formatting such as background color, font size, page title, etc.

The DBMS, which is currently either PostgreSQL or MySQL, executes the SQL query and returns a flat table. The Tree Constructor combines the schema with the flat table containing the SQL query result and outputs a tree-structured data. The Code Generator takes as inputs the data formatting and the tree-structured data and produces the application data output specified in the SuperSQL query.

IV. QUERY OPTIMIZER

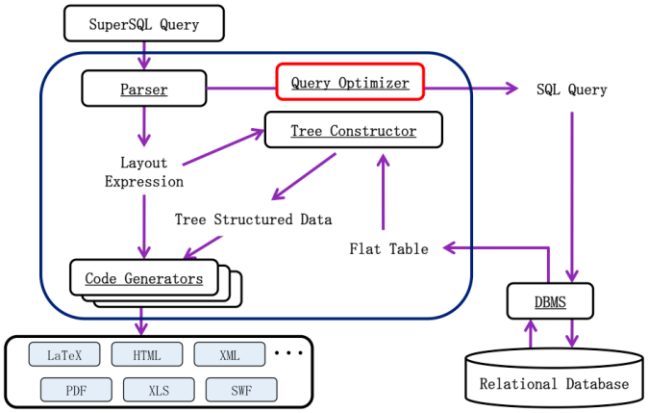

Figure 5. Proposed SuperSQL System Architecture

To implement query decomposition, a Query Optimizer component was added to the system as can be seen in Figure 5. The Query Optimizer is responsible for checking the divisibility of a query and creating single or multiple SQL statements.

The divisibility of a query is determined by modeling the query as an undirected graph. All the attributes found in the schema and the WHERE clause are represented as vertices. The relationships that exist between the attributes are represented as edges.

Edges are added to the graph based on three conditions: first, two attributes are from the same table; second, two attributes are equated in a WHERE condition; and lastly, two attributes are grouped together in the schema.

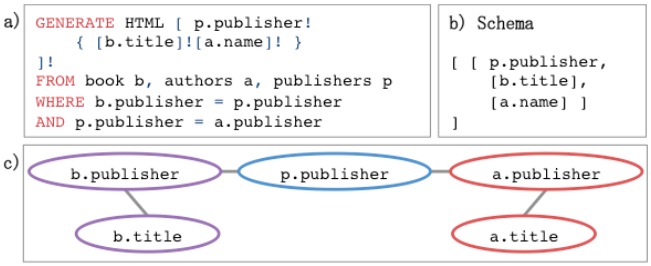

Figure 6. Example: Query graph with Connector Node

In Figure 6 a), we have a SuperSQL query with Figure 6 b) as the schema. From the schema, we added the attributes, *b.title*, *p.publisher* and *a.name* were added as vertices in our graph. From the WHERE clause, *b.publisher* and *a.publisher* were also added as vertices in the graph. Figure 6 c) shows the resulting graph.

Connector nodes are nodes that connect at least two attributes from different tables. A connector node must not

be in the same table as any of the attributes it connects. In Figure 6 c), *p.publisher* is a connector node.

After creating the graph, connected components were identified using Depth-First Search. A connected component is a sub graph that contains a path between all pairs of vertices in the graph. Depth-First Search is a search algorithm that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path. Each node can only be visited once except for connector nodes, which can be visited as many times as the number of nodes it connects. The Depth-First Search was repeated until all nodes have been visited. The resulting number of paths in the path list is equal to the number of connected components of the graph, which is subsequently equal to the number of possible divisions for the query.

In the example in Figure 6, we get two connected components: *{b.title, b.publisher, p.publisher}* and *{p.publisher, a.publisher, a.name}*. Therefore, the query can be divided into two.

A. Tree-Structured Data Construction

The Tree Constructor was modified to handle the results of multiple queries generated by the query optimizer. To create the tree structure, the schema must be combined with SQL query results from different tables. The results of SQL queries are stored in a structure, which can be referenced by the schema. The schema is traversed to create the tree structure. A node in the schema becomes a parent node and the values from the query results are added to the tree structure as its children. Attribute-value pairs are used to ensure the correctness of the parent-children mapping.

B. Cases Handled

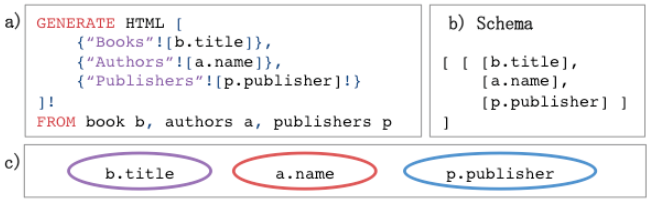

Figure 7. Example: Trivial Case

1) Trivial Case: Figure 7 illustrates the trivial case. In this example, *p.publisher*, the list of publishers, *b.title*, the list of books and *a.name*, the list of authors, are retrieved. These attributes are from different tables and are not related by any conditions thus they are independent of each other. Originally, the resulting query will generate an intermediate table equal to the Cartesian product of the three tables. However, based on the proposed query decomposition algorithm, the query can be divided into the following sub queries and retrieve smaller intermediate tables.

- *SELECT p.publisher FROM publishers p;*
- *SELECT b.title FROM books b;*
- *SELECT a.name FROM authors a;*

2) Grouped Columns: In this case, independent columns are grouped together by a common attribute. This is illustrated in Figure 6. In the given example, *b.title* and *a.name* are grouped together by *p.publisher*. The original query generates an intermediate table which takes the Cartesian product of the *books* and *authors* tables joined with the *publishers* table. However, based on the query decomposition algorithm, this query can be divided into the following sub queries.

- *SELECT b.title, p.publisher*
  FROM books b, publishers
  WHERE b.publisher = p.publisher;

- *SELECT a.name, p.publisher*
  FROM authors a, publishers p
  WHERE a.publisher = p.publisher;

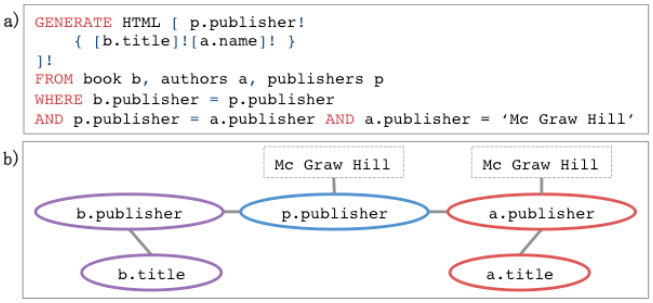

Figure 8. Example: Query with String Literal

3) Queries with String or Literal Conditions: In Figure 8, *a.publisher* is equated to the string literal, *"McGraw Hill."* Since *a.publisher* is connected to the connector node, *p.publisher*, the string literal, *"McGraw Hill"* is copied to the *p.publisher*. This is done so that when the query is divided, the sub queries will contain the same condition and thus the number of tuples retrieved would be minimized. The resulting queries for this example are:

- *SELECT a.name, p.publisher*
  FROM authors a, publishers p
  WHERE a.publisher = p.publisher
  AND (a.publisher = 'McGraw Hill');

- *SELECT b.title, p.publisher*
  FROM books b, publishers p
  WHERE b.publisher = p.publisher
  AND (p.publisher = 'McGraw Hill');

V. EVALUATION

This study aims to reduce the execution time and memory consumption of SuperSQL queries by reducing the size of the intermediate table returned by the DBMS. This was done by adding a query optimizer, that implements query decomposition, to the SuperSQL system. The resulting SuperSQL system is called the optimizer version.

A. Input Queries

To evaluate the effectiveness of the query optimizer, two query cases were used. Query case 1 refers to the query in

 ISBN: 978-1-61208-247-9

Figure 7, which illustrates the trivial case. Query case 2 refers to the query in Figure 6, which illustrates the grouped case. Table III compares the expected number of tuples with and without query decomposition.

TABLE III. COMPARISON OF EXPECTED NUMBER OF TUPLES WITH AND WITHOUT QUERY DECOMPOSITION

Query Case	Expected no. of Tuples w/o Decomposition	Expected no. of Tuples w/ Decomposition
1	$\|b.title\| \times \|p.publisher\| \times \|a.name\|$	$\|p.publisher\| + \|b.title\| + \|a.name\|$
2	$\|(b.title \times a.name) \bowtie p.publisher\|$	$\|p.publisher \bowtie b.title\| + \|p.publisher \bowtie a.name\|$.

Theoretically, the reduction of the intermediate table size results to the following: faster execution of the DBMS, smaller data structure size used to store the intermediate table, and faster selection of tuples to be included in the application data tree structure. In other words, reduction of intermediate table size results to faster execution and less memory consumption, which were proven by the experiments discussed in this section.

B. Experimental Environment

The test queries were executed by a machine running Mac OS X version 10.6.8 with 2.3GHz Inter Core i5 processor and 4GB 1333 MHZ DDR3 main memory. The intermediate table size is the independent variable. It is defined as the number of expected tuples without query division. Three intermediate table sizes were used. Small – with tuples ranging from 10 to 100; medium – with tuples ranging from 100 to 1000; and large with tuples ranging from 10000 to 100000. Results were compared to the performance of the SuperSQL src08 package, which was used as the baseline in the experiments.

C. Data Construction Time

For the three different size ranges, the data construction time of the optimizer version was compared to the baseline. In the following graphs, the legend Base1 refers to the baseline Query Case 1 (trivial case) and Opt-1 refers to the optimized Query Case 1. Base-2 refers to the baseline Query Case 2 (grouped case) and Opt-2 refers to the optimized Query Case 2.

In the small intermediate table size range, it can be observed in Figure 9 a) that the data construction time of all the cases are almost the same. Since the original query yields only a small intermediate table, its processing time is not significantly different from the total processing time of checking the divisibility of the query, executing and combining results of individual sub queries. Nevertheless, it can be seen that Opt-1 and Opt-2 performed a little faster than their baseline counterparts.

In the medium intermediate table size range, the optimizer version performed significantly faster than the baseline. This can be observed in Figure 9 b). It can also be seen that after processing 800 tuples, a rapid growth was seen in baseline algorithm's data construction time. However, for both trivial and grouped case, the data construction time growth of the optimizer version was linear.

In the large intermediate table size range, the optimizer version also performed significantly faster than the baseline. In Figure 9 c), it can be observed that the growth of the baseline in the grouped case is exponential. The growth of the baseline in the trivial case is also exponential but at a slower level. In the optimizer version, the data construction time of Opt-1 and Opt-2 were almost the same and their growths were both linear.

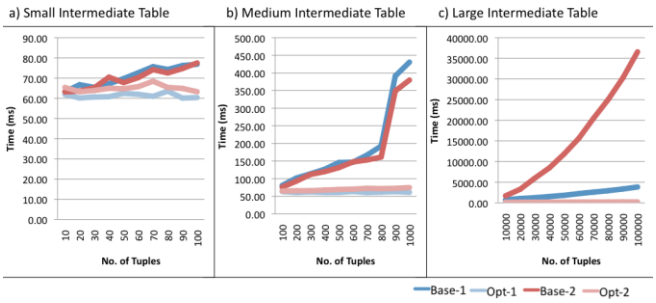

Figure 9. Data Construction Time of Queries

It can be inferred from these results that for the medium and large intermediate table size ranges, the percentage of data construction time reduced is significantly larger than the overhead cost of checking the divisibility of the query and executing and combining results of the sub queries.

D. Memory Consumption

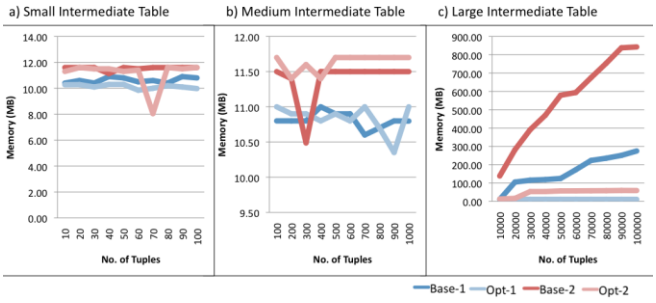

Figure 10. Memory Consumption of Queries

Query Case 1 and Query Case 2 were also used to observe the memory consumption.

For small and medium intermediate table size ranges, the amount of memory used to process both query cases in both the baseline and optimizer version range from 8-12 MB. This can be observed in Figure 10 a) and Figure 10 b). This is because for such intermediate table size ranges, the amount of memory saved from the reduction of the

 ISBN: 978-1-61208-247-9

intermediate table size and the overhead memory consumption cost of the graph structure used for checking divisibility are not significantly different.

It can also be observed that the Query Case 2 or the grouped attribute case consumed a little more than the trivial case. This is because the height of its schema tree is higher and thus more memory is needed to represent the tree.

It can be observed in Figure 10 c) that the optimizer version consumes significantly less amount of memory than the baseline for both the trivial case and grouped attribute case. This means that for this size range, the amount of memory saved from the reduction of the intermediate table size is significantly greater than the overhead memory consumption cost of the graph structure used in checking the divisibility of the query.

VI. CONCLUSIONS AND FUTURE WORK

The study proposed a query optimizer for the SuperSQL system based on query decomposition. The main goals were to reduce SuperSQL execution time and memory consumption by reducing the intermediate table size. A SuperSQL query was modeled as a graph wherein vertices are the attributes in the query and edges are the relationships that exist between the attributes. The relationships between the attributes were based on the desired layout of the attributes in the query and the schema, the relational operations in the input query's WHERE clause, and the tables where each attribute belongs. The connected components of the resulting graph were computed by using the Depth-First Search algorithm. The number of connected components is equal to the number of possible divisions for the query. The connected components were converted into SQL queries and executed individually. As a result, for some query cases, the combined size of the intermediate tables of the sub queries was significantly smaller than the size of the intermediate table without query division.

The data construction time and memory consumption of the SuperSQL with the query optimizer were compared to the original SuperSQL version. The comparison was done for three intermediate table size ranges. For queries with intermediate table size of 10 to 100 tuples, the optimizer version did not significantly differ from the original in terms of execution time and memory consumption. For queries with a medium intermediate table size of 100 to 1000 tuples, the optimizer version executed significantly faster, but the memory consumption was not significantly lower. For queries with a large intermediate table size of 10000 to 100000 tuples, the optimizer version executed significantly faster and consumed significantly less memory. Based on these experiments, it can be concluded that the proposed optimizer is effective in reducing SuperSQL execution time and memory consumption for the query cases that it can handle.

The proposed optimizer has already been integrated to the currently working SuperSQL system as a command line parameter for the standalone SuperSQL JAR executable file and as preference setting in the SuperSQL Eclipse plugin.

For future work, the processing of more query cases is deemed necessary to increase the optimization level of the proposed optimizer. The proposed optimizer is still not able to handle all possible query cases. However, it was designed it to fail safely and execute the original process when unhandled cases are encountered.

ACKNOWLEDGMENT

The authors would like to thank the members of the SuperSQL group of Toyama Laboratory for their support and Asian Development Bank – Japan Scholarship Program for the graduate scholarship grant.

REFERENCES

[1] SuperSQL Website: http://ssql.db.ics.keio.ac.jp/en, [retrieved: November, 2012].

[2] T. Bain: "Are Relational Databases Doomed?", ReadWrite Enterprise, http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php (2009), [retrieved: November, 2012].

[3] F. Bonchi, C. Castillo, D. Donato, and A. Gionis: "Topical Query Decomposition", In Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2008), pp. 52-60.

[4] T. T. T. Le, D. D. Doan, V. C. Bhavsar, and H. Boley: "A Bottom-up Algorithm for Query Decomposition", International Journal of Innovative Computing and Applications Volume 1, Issue 3 (July 2008), pp. 185-193.

[5] Y. Maeda and M. Toyama: "ACTIVIEW: Adaptive data presentation using SuperSQL", In Proceedings of the 27th International Conference on Very Large Data Bases (2001), pp. 695-696.

[6] S. G. Shin: "The Integration of Media Generators in SuperSQL Query Processor", Master's Thesis: Keio University School of Science for Open and Environmental Systems (2002).

[7] A. Silberchatz, H. F. Korth, and S. Sudarshan: "Database System Concepts (6th Edition)", McGraw-Hill, (2010).

[8] S. S. Skiena: "The Algorithm Design Manual", Springer, (2008).

[9] M. Toyama: "Three dimensional generalization of Target List for simple database publishing and browsing", Research and Practical Issues in Database (Proc. 3rd Australian Database Conference), World Scientific Pub. Co. (1992), pp. 139-153.

[10] M. Toyama: "SuperSQL: an extended SQL for database publishing and presentation", In Proceedings of ACM SIGMOD International Conference on Management of Data, (June 1998), pp. 584-586.

 ISBN: 978-1-61208-247-9