CHAPTER-6

IMPLEMENTATION

6.0 Introduction

A working prototype of SIQS has been developed that has the feature discussed in the previous chapters. We have used the client/server architecture to implement the above system because of its scalability, parallelism, loosely coupled and multi-tier characteristics. Since Client/Server systems can be scaled vertically and horizontally, the clients can be added easily and the performance can be improved by adding faster server hardware. This in turn allows the system to grow in size. Also, client server systems allow several users to submit requests independently at the same time. These requests are processed in parallel, which in turn leads to better performance due to faster generation of results. Another important reason for using the client-server system is that it allows for dividing the application into separate processes operating either on the client machine or on the server machine. The only mode of communication between the client and the server is through message passing. This feature results in a loosely coupled system. The multi-tier feature of the client/server system allows the components in each tier to be added and removed
based on the requirements. Also this aspect allows a client to act as server and server
to act as a client in different contexts. The implementation architecture is discussed
next.

6.1 Implementation Architecture

The commonly used client-server architectures in system design are either the two-
tier architecture or the n-tier architecture. In a two-tier system one of the tiers
provides interface to the user whereas the other tier manages the data. Here the
application logic is distributed between the client and server which results in a
tightly coupled client and server i.e. any change in the storage representation of data
necessitates change in the client program. On the other hand, an n-tier system
divides the system into individual parts, where each part is responsible for
performing a certain task. The only mode of communication between these parts is
through message passing. In the n-tier architecture the user interface layer, the data
storage layer and the application logic layer constitute different tiers. They
themselves can be divided into independent sub parts forming a separate tier. The n-
tier architecture results in a loosely coupled system. Since in SIQS, the system is
divided into four parts namely the Graphical User Interface (GUI), the Application
Server, the Database Server and the Component Databases, the four-tier client/server
architecture is used for implementing the system. The implementation architecture is
shown in Figure-1.

The Database Server provides access to the component databases registered with the
global system. The Application Server consists of the Query Generator and the
Result Integrator. The GUI enables the user to construct the query and remains independent of the manner in which the system responds to the query. The Query Generator is responsible for generating multiple results corresponding to the user query. The Query Generator uses the Database Server to interact with the component databases. The Result Integrator is responsible of integrating relevant results chosen by the user.

![Implementation Architecture of SIQS](image)

We have used JAVA\cite{Hortsmann1999} as the programming language for writing Client and Server programs and JAVA RMI\cite{Horstmann2000} for communication between distributed JAVA applications. The system uses ORACLE \cite{Durbin1997} as back-end RDBMS and connection to it is established using JDBC\cite{Reese2000}. The Microsoft Windows \cite{Solomon2000} is used as the operating system.

The implementation details about the Query Generator are discussed in the next section.
6.2 Query Generator

The Query Generator, as discussed in Chapter-4, is responsible for generating multiple non-redundant generated queries based on the user query. The algorithm[VijayKumar2002] given below generates the query resolutions in the desired order using the principles of density and relationship as discussed in Chapter-4. Further, the redundant query resolutions are not generated.

Let the user query be

```sql
SELECT name1, name2, ....... name_i
WHERE name_{i+1} = ' ' OR name_{i+2} = ' ' OR .... OR name_n = '
```

The names name1, name2 ... name_n are the SMNs used by the global system. These SMNs represent the relations and/or attributes present in the component databases registered with the global system. The relation name together with schema to which the relation belongs is referred to as a reference. In general, there can be more than one reference for a given name in the user query. A reference set for each name that contains all the references for that name is created.

Let RefSet_1, RefSet_2 ... RefSet_n be reference sets for the names name_1, name_2 ... name_n in the user query. A relation called RefTable, which stores information about the references in the reference sets, is defined by the system. The RefTable has 3+n attributes where n is the number of names in the query.

The three fixed attributes are

```
SchemaName, RelationName, RefCount
```
The remaining \( n \) attributes are

\[
\text{InRejSet}_1, \text{InRejSet}_2, \ldots, \text{InRejSet}_n
\]

The \text{RefTable} will, therefore, have the following attributes

\[
(\text{SchemaName}, \text{RelationName}, \text{RefCount}, \text{InRejSet}_1, \text{InRejSet}_2, \ldots, \text{InRejSet}_n)
\]

Distinct references are identified from all the reference sets. A tuple is inserted in the \text{RefTable} for each of these distinct references. The tuple is generated as follows:

The \text{SchemaName} and the \text{RelationName} are derived from the reference. A reference can belong to more than one reference set. This arises if more than one name in the user query refers to the same relation of the same schema.

\text{RefCount} is the number of reference sets to which the reference belongs.

In the remaining attributes 1 is inserted if the reference appears in the corresponding reference set and 0 is inserted otherwise.

The \text{RefTable} is used to find a resolution for the user query. The \text{RefTable} is treated as a relation and queries are executed using this table. These queries are referred to as meta queries. The result of executing a meta query is a query resolution.

The meta query has two aspects to it which together account for the Distance and Weighted Counts of the result queries. The first aspect is the number of range variables which range over the \text{RefTable}. The second aspect is the Condition. The meta query is formulated in the following way:
If there are \( m \) range variables over RefTable then the list of attributes in the SELECT clause will be

\[
R_1.\text{SchemaName}, R_1.\text{RelationName} \ldots R_m.\text{SchemaName}, R_m.\text{RelationName}
\]  

(1)

and the FROM clause will have the following relation names

\[
\text{RefTable} \ R_1, \text{RefTable} \ R_2 \ldots \text{RefTable} \ R_m
\]  

(2)

The condition for the meta query has to take into consideration that all the names in the user query are accounted for. This can be achieved by having \( \text{lnRefSet} \) value as 1 for at least one of the range variables. The corresponding condition statement looks as follows

\[
( R_1.\text{lnRefSet}_1 = 1 \ OR \ R_2.\text{lnRefSet}_1 = 1 \ldots \ OR \ R_m.\text{lnRefSet}_1 = 1 ) \ AND
\]

\[
( R_1.\text{lnRefSet}_2 = 1 \ OR \ R_2.\text{lnRefSet}_2 = 1 \ldots \ OR \ R_m.\text{lnRefSet}_2 = 1 ) \ AND
\]

\[
\vdots \quad \vdots \quad \vdots
\]

\[
( R_1.\text{lnRefSet}_n = 1 \ OR \ R_2.\text{lnRefSet}_n = 1 \ldots \ OR \ R_m.\text{lnRefSet}_n = 1 )
\]  

(3)

Consider, now, the Distance which is defined as

\[
\text{Distance} = \text{number of joins} \times \text{number of schemas}
\]

For the same number of joins, the distance will be more if the number of schemas are more. Therefore, if there are \( n \) range variables, then the meta query with the condition
\[ R_1.SchemaName = R_2.SchemaName \]
\[ R_1.RelationName \neq R_2.RelationName \]
\[ \text{AND} \]
\[ ( \]
\[ R_1.SchemaName \neq R_3.SchemaName \text{ AND } \]
\[ R_2.SchemaName \neq R_3.SchemaName \]
\[ ) \text{ AND} \]
\[ \text{AND} \]
\[ \text{AND} \]
\[ ( \]
\[ R_1.SchemaName \neq R_m.SchemaName \text{ AND } \]
\[ R_2.SchemaName \neq R_m.SchemaName \text{ AND } \]
\[ \vdots \]
\[ R_{m-1}.SchemaName \neq R_m.SchemaName \]
\[ ) \]

This generates a query resolution with less distance than a meta query with the condition
\[ R_1.SchemaName \neq R_2.SchemaName \]
\[ \text{AND} \]
\[ ( \]
\[ R_1.SchemaName \neq R_3.SchemaName \text{ AND } \]
\[ R_2.SchemaName \neq R_3.SchemaName \]
\[ ) \text{ AND} \]
\[ \text{AND} \]
\[ \text{AND} \]
\[ ( \]
\[ R_1.SchemaName \neq R_m.SchemaName \text{ AND } \]
\[ R_2.SchemaName \neq R_m.SchemaName \text{ AND } \]
\[ \vdots \]
\[ R_{m-1}.SchemaName \neq R_m.SchemaName \]
\[ ) \]
This fact is used while generating query resolutions in the desired order.

Consider, now, the Weighted Count. The generated queries are to be generated in the decreasing order of Weighted Count. This can be expressed as

\[
\text{SELECT } R_1.\text{RefCount}^n + R_2.\text{RefCount}^{n-1} + \ldots + R_m.\text{RefCount}^{n-m+1} \\
\text{FROM } \text{RefTable } R_1, \text{RefTable } R_2 \ldots \text{RefTable } R_m \\
\text{WHERE} \\
R_1.\text{RefCount} > R_2.\text{RefCount} \\
R_2.\text{RefCount} > R_3.\text{RefCount} \\
\ldots \\
R_{m-1}.\text{RefCount} > R_m.\text{RefCount} \\
\text{ORDER BY } 1 \text{ DESC}
\]

The condition for the meta query also takes into account that the redundant queries are not generated. This is ensured by saving the resolutions and using this information in subsequent meta queries. Any subsequent resolution, which has in it all the references of the saved query resolution, is considered redundant.

Towards this, (m-1) tables are created named Join_0, Join_1 ... Join_m-2. Each table represents the number of joins in a given query resolution. The number of attributes, therefore, is determined by the number of joins. Thus, the table Join_0 created for query resolutions without join will have the following attributes
and the table $Join_{m-2}$ created for resolution with $m-2$ joins will have the following attributes

$$(SchemaName_1, RelationName_1), \ldots, (SchemaName_{m-1}, RelationName_{m-1})$$

A tuple is inserted in the table $join_j$ if a resolution has $j$ joins. This tuple contains the schema names and relation names that take part in the join.

The meta query's condition, now, should reflect the fact that a query resolution should not be the same as any of the tuples in the tables $join_0, join_1, \ldots, join_{m-2}$.

This condition is specified as shown in Figure-2.

In addition GROUP BY clause is used to avoid permutation of query resolutions.

GROUP BY

$R_1.SchemaName, R_1.RelationName \ldots R_m.SchemaName, R_m.RelationName$ \hspace{1cm} (8)

Combining (1),(2),(3),(4),(6),(7) and (8) we get the complete meta query.

6.2.1 An Example

Let us consider the following user query ($Q$)

SELECT $Company, Department$

CONDITION $Cloc = 'Bangalore' or Dname = 'Computer'
NOT EXISTS

( SELECT * FROM Join_0
WHERE
  ( Join_0.SchemaName = R1.SchemaName OR
    Join_0.SchemaName = R2.SchemaName )
  AND
  ( Join_0.RelationName = R1.RelationName OR
    Join_0.RelationName = R2.RelationName )
)
AND

( SELECT * FROM Join_m-2
WHERE
  ( Join_m-2.SchemaName = R1.SchemaName OR
    \ldots
    Join_m-2.SchemaName = Rm.SchemaName )
  AND
  ( Join_m-2.RelationName = R1.RelationName OR
    \ldots
    Join_m-2.RelationName = Rm.RelationName )
  AND
  \ldots
  AND
  ( Join_m-2.SchemaName_{m-1} = R1.SchemaName OR
    \ldots
    Join_m-2.SchemaName_{m-1} = Rm.SchemaName )
  AND
  ( Join_m-2.RelationName_{m-1} = R1.RelationName OR
    \ldots
    Join_m-2.RelationName_{m-1} = Rm.RelationName )
)

Figure-2 – Condition in the Meta Query for Non-Redundant Query Resolution
The names in the user query are Company, Department, Cloc, Dname and therefore \( n=4 \).

Using the schemas, shown in Figure-14 of Chapter-2, the reference sets for each name in the query Q is shown in Figure-3.

<table>
<thead>
<tr>
<th>Name</th>
<th>RefSet(_1)</th>
<th>Name</th>
<th>RefSet(_2)</th>
</tr>
</thead>
</table>

Figure-3 – Reference Sets for Names in the user query

The RefTable will have the following attributes

\[(\text{SchemaName}, \text{RelationName}, \text{RefCount, InRefSet}_1, \text{InRefSet}_2, \text{InRefSet}_3, \text{InRefSet}_4)\]

There are in all 7 distinct references that are identified from the four reference sets.

A tuple for each of these distinct references is inserted into the RefTable. The RefTable is shown in Figure-4.
<table>
<thead>
<tr>
<th>Schema Name</th>
<th>Relation Name</th>
<th>RefCount</th>
<th>InRefSet₁</th>
<th>InRefSet₂</th>
<th>InRefSet₃</th>
<th>InRefSet₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema_1</td>
<td>Department</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Schema_3</td>
<td>Industry</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Schema_2</td>
<td>Comp</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Schema_2</td>
<td>Dept</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Schema_3</td>
<td>Branch</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Schema_4</td>
<td>Company</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Schema_4</td>
<td>Emp</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure-4 – Reference Table for the user query**

The first meta query that is generated is arrived at by putting m as 1 in the meta query algorithm. The meta query is

```
SELECT SchemaName, RelationName
FROM   RefTable
WHERE InRefSet₁ = 1 AND InRefSet₂ = 1 AND InRefSet₃ = 1 AND InRefSet₄ = 1
```

The query resolution generated after executing the above meta query is
The information about this query resolution is stored in the Join_0 table which is used in the subsequent query. The table Join_0 is shown in Figure-5.

<table>
<thead>
<tr>
<th>SchemaName₁</th>
<th>RelationName₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema₁</td>
<td>Department</td>
</tr>
</tbody>
</table>

Figure-5 – Table Join_0

This query resolution is transformed into the following generated query by the system:

```
SELECT Schema₁.Department.D#, Schema₁.Department.Name, 
FROM Schema₁.Department 
WHERE ( 
    Schema₁.Department.Cloc = 'Bangalore' OR 
    Schema₁.Department.Name = 'Computer' 
);
```

The next meta query that is generated for m=2 is shown in Figure-6. The two query resolutions generated after executing the meta query shown in Figure-6 are:

- Schema₃.Industry
- Schema₃.Branch
- Schema₂.Comp
- Schema₂.Dept
SELECT (R1.RefCount*4+ R2.RefCount*3), R1.SchemaName, R1.RelationName, R2.SchemaName, R2.RelationName
FROM RejTable RJ, RejTable R2
WHERE
  (R1.InRejSet1 = 1 OR R2.InRejSet1 = 1) AND
  (R1.InRejSet2 = 1 OR R2.InRejSet2 = 1) AND
  (R1.InRejSet3 = 1 OR R2.InRejSet3 = 1) AND
  (R1.InRejSet4 = 1 OR R2.InRejSet4 = 1)
AND
  R1.SchemaName = R2.SchemaName AND R1.RelationName <> R2.RelationName
AND
  R1.RefCount > R2.RefCount
AND
NOT EXISTS
(SELECT *
FROM Join
WHERE
  (Join.O.SchemaName1 = R1.SchemaName OR
   Join.O.SchemaName1 = R2.SchemaName )
  AND
  (Join.O.RelationName1 = R1.RelationName OR
   Join.O.RelationName1 = R2.RelationName )
)
GROUP BY R1.SchemaName, R1.RelationName, R2.SchemaName, R2.RelationName
ORDER BY 1 DESC

Figure-6 – Meta Query Generated for m=2 for the user query
The information about this query resolution is stored in the Join_1 table which is used in the subsequent query. The table Join_1 is shown in Figure-7.

<table>
<thead>
<tr>
<th>SchemaName1</th>
<th>RelationName1</th>
<th>SchemaName2</th>
<th>RelationName2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema_3</td>
<td>Industry</td>
<td>Schema_3</td>
<td>Branch</td>
</tr>
<tr>
<td>Schema_2</td>
<td>Comp</td>
<td>Schema_2</td>
<td>Dept</td>
</tr>
</tbody>
</table>

Figure-7 – Table Join_1

The generated query for the above two query resolutions are

```sql
  AND

and

to_char(Schema_2.Dept.Dno), Schema_2.Dept.Dname
FROM Schema_2.Comp, Schema_2.Dept
WHERE Schema_2.Comp.C#=Schema_2.Dept.C#
  AND
  ( Schema_2.Comp.Cloc='Bangalore' OR Schema_2.Dept.Dname='Computer' )
```

Continuing in this way we find that the query Q results in 7 non-redundant query resolutions in the desired order. The query resolutions, with their Distance and Weighted Counts for the user query Q, are shown in Figure-3 of Chapter-4. The generated queries corresponding to the query resolutions are shown in Figure-4 of Chapter-4.
6.3 Result Integrator

The Result Integrator, as discussed in Chapter-5, is responsible for integrating the relevant results for the user based on the user query. The user while viewing the results, one at a time, is allowed to save them. These results are stored as tables in the database server. The SIQS stores the result in the following way.

Let the generated query corresponding to the result to be saved by the user be

```
SELECT S1.R1.A1, S2.R2.A2, S3.R3.A3, ... S_m.R_m.A_m
FROM S1.R1, S2.R2, S3.R3, ..., S_m.R_m
WHERE
  ...
  ...
  S_m-1.R_m-1.A_m-1 = S_m.R_m.A_m
AND
  (S1.R1.A_1 = ' ' AND
   S2.R2.A_2 = ' ' AND
   S3.R3.A_3 = ' ' AND
   ...
   ...
   S_m.R_m.A_m = ' ')
```

Where \( S_i \) denotes the schema name, \( R_i \) denotes the relation name and \( A_i \) denotes the attribute name.

In order for the result to be saved in the database server the following CREATE TABLE statement is generated
CREATE TABLE table_name_1
AS
(
FROM S1.R1, S2.R2, S3.R3, ..., Sm.Rm
WHERE
   ...
   Sm-1.Rm-1.Am-1=Sm.Rm.Am
AND
   (S1.R1.A1=' ' AND
    S2.R2.A2=' ' AND
    S3.R3.A3=' ' AND
    ...
    Sm.Rm.Am=' ')
)

Executing the above statement, saves the result in the database server. The table_name_1 is the name of the table by which the result is saved. This name is generated by SIQS for storing the result.

Once all the results for the user query are generated, the results saved are shown to the user for integration. Suppose the user saves n results and let table_name_1, table_name_2, table_name_3 ... table_name_n be the table names by which these n results are saved. The results are then fetched from the database server in order to be shown to the user. The query used to fetch the results from the database server is the following
These queries are executed and the results of these queries are shown to the user. The user is given an option to choose results, from these saved results, for integration. The result integrator integrates the chosen results based on the 'connectedness' criterion for integration, as discussed in Chapter-5. Suppose the user chooses all the results saved by him/her for integration. These results are then integrated in the following manner.

Let $C_{ij}$ denote the commonality between results $R_i$ and $R_j$ and $V_{ij}$ denote the variance between results $R_i$ and $R_j$. Figure-8 shows the commonality and variance between the pair of results saved by the user.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>...</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td>$C_{12}$</td>
<td>$V_{12}$</td>
<td>$C_{13}$</td>
<td>$V_{13}$</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td>$C_{23}$</td>
<td>$V_{23}$</td>
<td>...</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>$C_{3n}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Rn-1</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>$C_{n-1n}$</td>
</tr>
</tbody>
</table>

**Figure-8 – Commonality and Variance between the pair of results**

Let Max denote the maximum commonality and Min denote the minimum variance then the algorithm shown in Figure-9 is used to identify the pair of results that have maximum commonality and minimum variance. $R$ and $S$ will store this pair of results.
For $i = 1$ to $N-1$
Begin
  For $j = i + 1$ to $N$
  Begin
    If $(C_{ij} > \text{Max})$ then
    Begin
      Max = $C_{ij}$
      Min = $V_{ij}$
      R = $R_i$
      S = $R_j$
    End
    Else if $(C_{ij} = \text{Max})$ then
    Begin
      If $(V_{ij} < \text{Min})$ then
      Begin
        Min = $V_{ij}$
        R = $R_i$
        S = $R_j$
      End
    End
End

Figure-9 – Algorithm for finding maximum commonality and minimum variance

If the pair of results $R$ and $S$ are union compatible then the following statement will be used to integrate this pair

```
SELECT * FROM R
UNION
SELECT * FROM S
```

In the case, when these pair of results are not union compatible, the following Natural Full Outer Join operation supported by ORACLE [Freeman2002] is used to integrate them.

```
SELECT * FROM R NATURAL FULL OUTER JOIN S
```
The result after execution of the above query is stored as a table in the database server by using the following statement

CREATE TABLE table_name_rs AS
(SELECT * FROM R
UNION
SELECT * FROM S)

or

CREATE TABLE table_name_RS AS
SELECT * FROM R NATURAL FULL OUTER JOIN S

where table_name_RS represents the name of the table storing the result that is arrived at after integrating the results R and S.

The number of results to be integrated after integration of a pair of results is reduced by one. This process continues till all the results are integrated. The final integrated result is then shown to the user.

6.3.1 An Example

Consider the user query Q of example in section 6.2.1. Suppose the user saves and chooses all the results for integration. The results, chosen by the user, are shown in Figure-10.

<table>
<thead>
<tr>
<th>Results</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>D# Dname Company Cloc</td>
</tr>
<tr>
<td>R2</td>
<td>Company Department</td>
</tr>
<tr>
<td>R3</td>
<td>C# Cname Cloc D# Dname</td>
</tr>
<tr>
<td>R4</td>
<td>C# Cname Cloc D# Dname</td>
</tr>
<tr>
<td>R5</td>
<td>C# Cname Cloc Department</td>
</tr>
<tr>
<td>R6</td>
<td>Company Department</td>
</tr>
<tr>
<td>R7</td>
<td>C# Cname Cloc Department</td>
</tr>
</tbody>
</table>

Figure-10 – Results chosen by the user
Figure-11 shows the commonality and variance between these seven results.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>R4</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Figure-11 – Commonality and Variance between the seven results chosen by the user

After using the algorithm shown in Figure-9, the values arrived at are

Max = 5
Min = 0
R = R3
S = R4

This shows that the pair of results R3 and R4 are considered the most connected and hence should be integrated first. The query used for integrating R3 and R4 is

```sql
SELECT * FROM R3 NATURAL FULL OUTER JOIN R4
```

The integrated result is then stored in the database server by executing the following statement

```sql
CREATE TABLE IR34 AS
SELECT * FROM R3 NATURAL FULL OUTER JOIN R4
```

The results left for integration after integrating R3 and R4 are shown in Figure-12.

<table>
<thead>
<tr>
<th>Results</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>D# Dname Company Cloc</td>
</tr>
<tr>
<td>R2</td>
<td>Company Department</td>
</tr>
<tr>
<td>R5</td>
<td>C# Cname Cloc Department</td>
</tr>
<tr>
<td>R6</td>
<td>Company Department</td>
</tr>
<tr>
<td>R7</td>
<td>C# Cname Cloc Department</td>
</tr>
<tr>
<td>IR34</td>
<td>C# Cname Cloc D# Dname</td>
</tr>
</tbody>
</table>

Figure-12 – Results after Integrating R3 & R4
In a similar manner, the remaining results are integrated to give the final integrated result. The final integrated result is shown in Figure-13. This integrated result is shown to the user.

<table>
<thead>
<tr>
<th>Result</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR1234567</td>
<td>C### Cname Cloc Company D## Dname Department</td>
</tr>
</tbody>
</table>

Figure-13 – The Final Integrated Result

6.4 Performance

In order to measure the performance of the system, we consider two different aspects:

(i) The number of generated queries that are generated by the system vis-a-vis the number of generated queries that can potentially be generated.

(ii) The performance of the system with the increase in the number of names as well as the number of schemas.

Recall that, the algorithm explained in Section 6.2 eliminates redundant queries while generating the non-redundant queries in the user desired order. The graph shown in Figure-14 depicts two curves - one for the queries generated by our system and the other for all potential queries. It can be seen that, as we increase the number of schemas, while the former is close to a linear curve the latter shows an exponential growth.

133
In order to measure the performance with the increase in the number of schemas and names in the user query, we have plotted four line graphs based on the number of names in the user query. These line graphs Graph-1, Graph-2, Graph-3 and Graph-4 are arrived at by executing the system for one name, two name, three name and four name queries respectively. The x-axis denotes the number of generated queries that are generated and the y-axis represents the time taken (in milliseconds) to generate these queries. In order to be able to compare the performance of the system as the number of schemas increase, the curves for four, five, six, seven and eight schemas are plotted in the same graph.

The graphs show that as we increase the number of schemas from four to eight there is no significant increase in time. This shows that the system performance does not degenerate as the number of schemas increases.
Graph-1 – One Name Query

Graph-2 – Two Name Query
In this thesis, we have proposed a system, Structure Independent Query System (SIQS), which provides access to multiple component databases. The system has the following features

SIQS resolves the semantic heterogeneity that may exist between the component databases registered with the global system. For this a globally consistent naming methodology is described that is used to assign consistent and meaningful names to the relations and attributes present in the component databases registered with the global system. These names, called the semantically meaningful names (SMNs), are assigned by the database administrator of the participating component databases on interaction with the global system. The various conflicts in names like synonyms, homonyms, data type, scaling are resolved using this methodology.

SIQS provides the user with a simplified query interface to pose a query. Unlike the present multidatabase language systems, the user is not expected to know the structure of the component databases while posing a query. These structures are
inferred at run time. This reduces the unnecessary burden on the user. For this, a query language, Structure Independent Query Language (SIQL), is proposed that enables the user to pose a query without any explicit reference to the schema. The query is expressed using the SELECT and the WHERE clause. A Graphical interface is provided to the user, which assists him/her to construct a query in SIQL by providing him/her with a list of SMNs used by the global system.

Unlike most of the present multidatabase language systems, the responsibility of handling naming conflicts that exists between the data present in multiple component databases lies with the system and not with the user. Since the user uses the SMNs in the user query, he/she need not have to worry about the conflicts that may exist due to names. This makes the system easy to use.

SIQS determines the structures, from which the data is to be fetched, at run time corresponding to the SMNs present in the user query. Using these structures the path expressions are inserted by SIQS in the user query. There can be many sets of structures for a user query and therefore multiple queries are generated by the system. A ‘closeness’ criterion is defined by the system to order these multiple queries generated with the ones more ‘close’ higher in the order. This criterion results in the more desired result, from the user’s point of view, getting generated earlier in the querying process.

SIQS allows the user to integrate these multiple generated results, for the user query, in order to make these results more desirable and meaningful from the user’s point
of view. The system enables the user to play an active role during the integration of these results by allowing him/her to choose the ones that are relevant to him/her. For integrating these results, an ordering criterion based on ‘connectedness’ of results is proposed. This criterion defines the order in which the results need to be integrated so that the integrated results are desirable and meaningful from the user’s point of view. This ordering of results for integration, based on the defined criterion, leads to full disjunction if the set of results, for integration, form a connected γ-acyclic hypergraph.

SIQS allows the user to pose multiple interrelated queries in addition to a single query in SIQL. The system allows the user to save the results of each of these interrelated queries. The results for a user query may motivate the user to seek more information and hence the user may feel inclined to pose another query. At the same time the user would like to save the current result for future reference. SIQS provides this facility to the user. The results of each of these individual queries in SIQL can be integrated in order to be presented to the user.

We have measured the performance of the system by measuring the increase in the time taken to generate and execute the generated query. Graphs were plotted by varying the number of schemas and names in the user query. The graphs show that as we increase the number of schemas from four to eight there is no significant increase in time. This shows that the system performance does not degenerate as the number of schemas increases.
7.1 Future Directions

In SIQS, the component databases, which are registered with the global system, are relational. The system can be enhanced in future to support access to data present in object oriented databases and web databases.