CHAPTER-1

INTRODUCTION

Database systems, which were proposed to overcome the difficulties faced by the file systems, were initially centralized in nature as they ran on a single computer system with no interaction with other systems. The decentralized nature of various organizations led to the existence of many databases on computers spanning a wide geographical area. This decentralized nature was due to several factors like information being too large to be kept and managed in one centralized database, several databases being developed separately for different departments of an organization as also the advent of PC’s that could store small databases. This required the user to perform certain tasks to access these distributed databases [Dwyer1987]. These tasks included determining the computers that contained the data to be accessed, formulating queries to be executed on different computers, copying or transferring the results to a single computer for merging, combining and merging the results and finally extracting the answer to the original request from the result. These tasks being error-prone, tedious and time consuming required developing systems that would automate them i.e developing systems that would enable querying of data present in multiple databases. [Sheth1990] classifies...
multiple database systems along three orthogonal dimensions: Distribution, Heterogeneity and Autonomy. They are discussed next.

1.1 Distribution

Distribution deals with data being distributed among multiple databases instead of being stored at one central location. These multiple database systems may reside on a single or multiple computers interconnected by a communication system. The data is distributed by either partitioning it or by replicating it. Partitioning involves dividing the data into fragments each of which is stored at a different site. The fragmentation may be vertical or horizontal depending upon the requirement. Replicating involves storing each fragment at more than one site. [Ozsu1999] discusses different modes of distribution namely processing logic, function, data and control. All these modes are important and necessary for distribution. Distribution of data results in higher reliability, higher availability and improved access times [Ceri1984].

There are two ways in which database management systems can be distributed namely peer-to-peer distribution and client-server distribution [Ozsu1999]. The peer-to-peer distribution allows each machine to have full database management system functionality. Each of these machines communicates with other machines for querying and transaction purposes.

In client-server distribution, the server machine manages the data and the client provides an environment for writing application programs. The communication
responsibilities are shared between the client and the server machine. These client-server systems result in distribution of functionality between the client machine and the server machine. The client server system has the following characteristics [Orfali1993]

- It provides transparency by hiding the location of the server from the client.
- It is independent of the hardware and operating system.
- It uses messages for sending requests and receiving replies to the requests.
- It provides horizontal scalability by allowing addition or removal of client and vertical scalability by allowing a server to migrate to a faster server machine.

The Client-Server architecture makes the system loosely coupled as it divides the application into separate processes in separate machines connected by a communication network [Sinha92]. There are three building blocks of a client-server architecture: the server, the client and the middleware [Chaudhary1997] [Lewandowski1998].

Server: The server is a computer that stores data and procedures required to support end-user interaction. It provides data management services. The server applications (SQL database servers, transaction processing monitors and groupware servers) run on the server operating system. The server interfaces with the middleware through its operating system. The server can even act as a client by sending requests to other servers.

Client: The client is a consumer of services provided by the server. These services are provided to the user through a GUI. The GUI is arrived at with the help of the
operating system in order to maintain consistency across applications. The client side includes a GUI and a client operating system along with business specific applications. A client can offer services to other clients as also access services from any number of servers.

Middleware: The middleware is a distributed software that facilitates client-server interaction. It provides a framework for client-server communication. The middleware runs on both the client side and the server side of the application. Middleware helps to manage the complexity and heterogeneity that is present in distributed systems. The middleware facilitates distributed application component portability and interoperability. It is defined as a layer of software above the operating system but below the application program that provides a common programming interface across a distributed system. It provides a higher-level building block for programmers than the Application Programming Interfaces (APIs) such as sockets that are provided by the operating system. It significantly reduces the burden on application programmers by relieving them from the tedious and error-prone socket programming. The common middleware platforms that are in use are CORBA[1997], DCOM[1998], JAVA RMI[1999].

1.2 Heterogeneity

Heterogeneity in multiple database occurs due to their diverse nature. This diversity may be due to differences in DBMS or due to the difference in semantic representation of data. Heterogeneity due to differences in DBMS results from the
difference in type of data models used by the component databases or due to differences at the system level. The data model differences may be due to difference in structure, constraints or query languages of the data models used by the component databases. The system level heterogeneity is due to differences in transaction management techniques, hardware and system software requirements and communication capabilities.

The heterogeneity due to difference in semantic representation of data in multiple component databases is termed as semantic heterogeneity. This arises during the integration of various component databases. There are two types of integration in databases: Schema Integration and Data Integration.

Schema Integration is the activity of integrating schemas of existing and proposed databases [Batini1984] [Batini1986]. This Integration may occur in two contexts: View Integration and Database Integration. View Integration produces a global conceptual description of the proposed database whereas Database Integration produces the global schema of a collection of databases.

The second type of integration, Data Integration, refers to the creation of an integrated view over incompatible data collected from different component databases [Deen1987]. Integrating data instances involves entity identification, resolving attribute value conflicts and handling data conflicts related to field types, sizes, precision and scaling factors [Lim1993][Lim1994][Lim1996][Gadia1993][Sciore1994][Reddy1994].

The schemas that are to be integrated are diverse in nature. The factors that are responsible for this schema diversity are discussed in [Batini1986]. These are
• Different user perspectives or viewpoints in modeling the same objects in the application domain

• Equivalence among constructs of the model as several combinations of constructs can model the same application domain equivalently.

• Incompatible design specification due to error in choosing name, types and integrity constraints.

• Same concepts in the application domain may have different representations although they may be identical, equivalent, compatible or incompatible.

• Different concepts in different schemas are mutually related by some semantic properties.

Due to the above-mentioned factors, conflicts may exist while integrating schemas of various component database systems. The conflicts that are usually encountered are the naming conflicts and the structural conflicts. They are discussed below.

Naming Conflict: This is due to inconsistency among names in different component schemas. The inconsistency may be due to

• Same name being used for two different concepts (Homonyms)

• Same concept being described by two or more names (Synonyms).

The homonym problem can be handled by comparing concepts with the same name but the synonym problem cannot be handled until explicitly specified.

Structural Conflict: This is due to difference in choice of modeling constructs and integrity constraints. They may arise due to
• Same concept being represented by different modeling constructs in different schemas (Type Conflict).

• Groups of concepts are related among themselves with different dependencies in different schemas (Dependency Conflict).

• Different keys are assigned to same concept in different schemas (Key Conflict)

• Same class of objects uses different insertion/deletion policies in distinct schemas (Behavioral Conflict).

Resolving these conflicts whether naming or structural will give a more accurate and consistent view of various component databases.


1.3 Autonomy

Autonomy indicates the degree to which a component database can be independent in its decisions and operations. Autonomy is important, as each component database would like to have control over their data and operations even while cooperating with other component database systems [Sheth1990][Bukhres1996][Colomb1995].

The autonomy of the component databases may be of the following types [Veijalainen1988][Sheth1990]
Design Autonomy: It refers to the ability of the component database system to make a choice of its design i.e. the component database system is independent, from others, in choosing its design. Also the component database systems having such autonomy can change their design at any time.

Execution Autonomy: It refers to the ability of a component database system to decide the way in which operations are executed locally i.e. the component databases are independent in execution and scheduling of incoming requests.

Communication Autonomy: It refers to the ability of a component database system to decide whether or not to communicate with other component database systems i.e. the component database system has the freedom to choose the component database systems it wishes to communicate with.

Association Autonomy: It refers to the ability of the component database to decide whether and how much data and operations are to be shared with other systems i.e. the component database has the freedom to decide the amount of data it wants to share with the other systems.

[Du1990][Sheth1990] defines two types of autonomy namely operation and service. The operation autonomy is equivalent to the design and the execution autonomy whereas the service autonomy is similar to the association and the communication autonomy.

Preserving the above mentioned autonomy for component database systems is advantageous as it avoids the costly and tedious process of modifying component
database systems. It also facilitates in the addition or removal of component database from the global system.

1.4 Distributed Database Systems

The aim is to develop a system that can provide access to multiple component databases based on the three aspects: distribution, heterogeneity and autonomy discussed above. One approach that is used to provide access the multiple component databases is the Distributed Database System approach. Distributed Database Systems manages these multiple databases through a single integrated schema called the global unified schema. The design of the global schema in the earlier distributed database systems SDD-1[Rothnie1980], POREL[Neuhold1982] resulted in the entire distributed database to be designed from scratch[Litwin1987]. The distribution of data was achieved by splitting the global data into data for each site and the global system had the entire control over the local data and operations. The concept of global schema when applied to existing databases required integration of schemas of the component databases [Batini1986] [Parent1998] [Larson1989] [Czejdo1987][Sheth1988] [Rajnikanth1989]. This required resolving various conflicts that may exist between the schemas of these component databases. These conflicts, mainly due to the fact that reality can be perceived in different ways, made creation of the global schema a difficult task even for small number of component databases [Krishnamurthy1991][Dayal1984] [Litwin1985] [Lyngbaek1983] [Litwin1987] [Ceri1984] [Heimbigner1985].
Also in distributed database systems, the global system has complete control over local data and processing and the processing is optimized as per global requirements. This makes distributed database system to perform well at the global level but at the cost of lack of autonomy of the component database systems. Need for autonomy of the component databases led to the development of multidatabase systems [Litwin1990]. The multidatabase system is discussed in the next section.

1.4.1 Multidatabase Systems

Multidatabase system has been defined differently by different researchers. [Litwin1990] defines multidatabase system as a system that manages multiple autonomous databases simultaneously without a global schema. [Dayal1984] defines multidatabase system as providing a global view of the data stored in local databases without the databases being physically integrated. [Breitbart1988a] defines multidatabase system as a collection of several databases that provide uniform access to pre-existing databases without requiring the users to know either the location or the characteristics of different databases and their corresponding DBMS. [Sheth1990] defines multidatabase systems as systems that support operations on multiple component databases.

Based on the above definitions, a multidatabase system can be defined as a system that provides access to multiple homogeneous or heterogeneous, autonomous component databases where each of the component databases may either be centralized or distributed.
Since the component databases in the multidatabase system are independently designed, they may use different data models to represent their data. This may become a major obstacle for interoperability of these component databases. One way to overcome this problem is to have a common interface to each component database so that they can interact in a uniform manner. The common interface is arrived at by using a common data model. The relational model is the most widely used common data model for the multidatabase systems because of the simplicity of the relational language and features in expressing and mapping user queries into the languages of component databases. MERMAID[Templeton1987a][Templeton1987b], CALIDA[Jakobson1989][Rajnikanth1989], DATAPLEX[Chung1987][Chung1990], ADDS[Breitbart1988b], MYRIAD[Clements1993], SIRIUS-DELTA[Litwin1982] use the relational model as the common data model.

Also, in multidatabase systems, the global schema may or may not exist depending upon the way it is designed. [Bright1994] describes two approaches for designing a multidatabase system i.e. a global-schema approach and a multidatabase language approach. Global-schema approach to multidatabase system design is similar to Distributed Databases design except that the global system has no control over the local systems data and processing. Global schema is arrived at, from local schema, on resolving conflicts that may occur between the various component databases. Global database administrator is responsible for defining the global schema. Many systems are based on this approach namely SIRIUS-DELTA[Litwin1982], DDTS[Dwyer1987], MYRIAD[Clements1993], IMDAS [Krishnamurthy1987][Krishnamurthy1988], UniSQL/M[Kelley1995][Kim1995]
The global database administrator, in order to define a global schema, needs to have an extensive knowledge about the local schemas as also the user requirements of the global system. Further in the global schema design approach, the global database administrator is also responsible for maintaining the global schema with any change in the local schema. This task, being complex for a small number of databases, becomes nearly impossible for large number of component databases. Therefore the amount of knowledge required of each component database for integration and the complexity involved in updating the global schema with the change in the local schema have been the major drawbacks of this approach.

The Multidatabase Language approach overcomes most of the problems faced by the global-schema approach as most of the responsibility regarding integration of component databases rests with the user instead of the global system. It results in the shift of responsibility from the global system to the end user and the local system. The multidatabase language systems that are designed using this approach offer a uniform multidatabase query language, which provides functionality that allows querying several component databases at the same time. The multidatabase language is used to integrate information from several component databases. Since no global schema is maintained, many of the problems associated with global schema creation and maintenance are resolved [Bright1994]. It realizes a loosely coupled approach to integration and is termed as a loosely coupled federated
database system in [Sheth1990]. These systems are suitable when the number of component databases is large and the only operations that need to be performed are read-only operations for retrieving information from these component databases.

Although the multidatabase language systems have no constraint on the system size, the user is expected to have more knowledge about location of data and its representation while using the multidatabase language. Many multidatabase languages have already been proposed in the literature namely MDSL[Litwin1985][Litwin1986] [Litwin1987] [Wong1984][Wong1985], MSQL[Litwin1989] [Grant1993], SemQL[Lee1999], SchemaSQL [Lakshmanan1996] [Lakshmanan1999] [Lakshmanan2001]. Multidatabase language is discussed next.

1.5 Multidatabase Language

A multidatabase language allows a user to define and manipulate a collection of autonomous component databases. The language should enable the component databases to retain their autonomy while they are cooperating as part of the global system. One of the important aspects of a multidatabase language is the type of interface it provides to the user. The multidatabase query language interface is discussed next.

1.5.1 Query Interface

The multidatabase language should be able to provide a query interface that allows the user to pose queries on multiple component databases. The user query should
contain all the necessary information for retrieving data from various component databases, as it is responsible for integrating the data in multidatabase language systems. Since most of the multidatabase languages are relational, they provide a non-procedural interface to query multiple autonomous databases. The query interface of some of the multidatabase languages is discussed next.

MDSL is a manipulation language of multidatabase system MRDSM. It extends the classical DML of MRDS called DSL. MDSL is specific to MRDS System but has syntax similar to relational language somewhere between QUEL and SQL. The general form of the MDSL query is

\[
\text{OPEN name1[mode 1] name2[mode2]...}
\]
\[
\text{RANGE (tuple_variable relation) ...}
\]
\[
\text{SELECT <target list>}
\]
\[
\text{WHERE <predicates>}
\]

The OPEN command in it is used to specify the component databases that need to be accessed. The RANGE command specifies the relations to be accessed whereas the SELECT and the WHERE commands are similar to SQL commands.

MSQL is a multidatabase language that can express queries over multiple databases in a single statement. MSQL, termed as Multidatabase SQL, extends SQL by adding new functions for non-procedural manipulation of data in different and mutually non-integrated SQL databases. The overall design of MSQL is based on that of MDSL with adaptations to SQL syntax and semantics. The general form of an MSQL data manipulation statement is
<USE statement>

<SELECT statement>

USE statement specifies the component databases that are to be used and defines the scope of the query whereas the SELECT statement is similar to the SELECT statement of SQL adapted to be used for multidatabase queries.

SchemaSQL, an extension of SQL, offers the capability of uniform manipulation of data and meta-data in a relational multidatabase system. SchemaSQL query expression has the following Syntax

```
SELECT <var>.<attName>
FROM <range> <var>
WHERE predicate expression
```

The SELECT and FROM clause refers to attributes where var being the variable declared in the FROM clause and attName refers to the name of attribute of the relation over which var ranges. SchemaSQL permits the declaration of variables that can range over any of the following five sets.

a) names of databases in a federation
b) names of relations in a database
c) names of the attributes in the schema of a relation
d) tuples in a given relation in a database
e) values appearing in a column corresponding to a given attribute in a relation
In the Multidatabase System discussed above the user frames the query by giving database name as part of the OPEN command in MDSL, USE in MSQL and FROM in SchemaSQL. The relation name is expressed as part of the RANGE statement in MDSL, FROM clause in MSQL and SchemaSQL. If there is a conflict in the relation name it is preceded by the database name. The attributes are mentioned as part of the SELECT and WHERE clause and in case of conflict are preceded by the relation name or sometimes by the database name as well. In the above systems, the user is aware of, the names of the databases, the names of the relations and the names of the attributes he/she needs to access. He/She specifies them as part of the query.

The local component databases, which are independently designed, have different conventions for naming relations and attributes. This leads to a problem due to synonyms and homonyms. Also the attributes with same names in them may have different data types and scaling measures, which may lead to conflict due to difference in data type and scaling. The query language discussed above provides semantics that are used to resolve these conflicts to a certain extent. The user is expected to resolve these conflicts while constructing the query. In MDSL and MSQL, the user uses semantic variables as part of the user query to broadcast his/her intention over differently named data. MDSL uses the RANGE_S statement and MSQL uses LET < > BE < > statement to explicitly specify these semantic variables. These semantic variables are explicitly specified using character strings with generic characters % and * in RANGE and FROM statement in MDSL and MSQL query respectively. In SchemaSQL, the user associates the context information to relation
names as well as attribute names in addition to the values in the database. Also the semantic values are specified as part of the user query in order to include type information as part its context information.

The multidatabase language systems, based on the user query, should fetch information that is most desirable to the user. This is achieved by using an appropriate query processing strategy. The query processing in multidatabase language systems is discussed next.

1.5.2 Query Processing

The query processing in a multidatabase language system is performed in the following way.

The multidatabase system decomposes the user query into one or more subqueries, one each for the referred component databases. These individual subqueries are then sent to their respective component database sites for execution. The local component database system, if required, transforms this query to query for the native data model. The query is then applied to the interface of the local component databases. The local component database system, which treats these queries as any other local query, executes these queries and sends the result to the global system. The multidatabase system collects the results for each of these subqueries from their corresponding component database sites and integrates them. The criteria used for integration is defined by the multidatabase system. The integrated result is first filtered, based on the access rights of the user, and then presented to the user.
In MDSL and MSQL, the user is allowed to pose two types of queries namely an elementary query and multiple queries. In an elementary query, the relations may refer to one or more component databases with the WHERE clause specifying the inter-database join. This elementary query is then executed to give a single result in the form of a table to the user. The multiple queries simplify the task of the user when various component databases model the same reality. The user instead of giving individual elementary queries for each component database may give a single multiple query for it. This multiple query is then decomposed into multiple elementary queries, which are then executed to give multiple results. Each of these multiple results is a table. In the case when the user uses semantic variables, explicitly or implicitly, as part of the user query, these semantic variables are substituted by the names in their domain for execution.

Query processing in SchemaSQL is divided into two phases. In the first phase, the queries for local component databases are prepared. In the case when context information is associated with the relation name and the attribute name in the user query, the processing for comparison in query processing is modified. The comparison is performed after finding the type information with its associated context information and applying the appropriate conversion functions. The prepared queries are then executed at their respective component database sites. The multiple results returned, after execution of the prepared queries, are collected and
stored as tables. The second phase involves writing the original SchemaSQL query as a sequence of SQL queries over the stored tables of results.

The user query in the multidatabase language discussed above may generate multiple results. These generated results are stored as tables. These tables are termed as working tables in MDSL and MSQL. SchemaSQL allows a sequence of SQL queries, similar to the original query in SchemaSQL, to be written using these tables. The following problems exist with the multidatabase language systems discussed above.

1. Multidatabase language systems that are discussed above expect the user to have knowledge about the structure of the database i.e. the name of the database, the relations in it and the attributes that are present in these relations. This knowledge is used to frame a user query. The user has to explicitly state the relation from which the data is to be retrieved and in case of conflict in the relation, the relation has to be qualified with the database name. [Quass1995] discusses a query language where the FROM clause may be omitted but the path expression is present in the Select as well as the Where clause. The graphical user interface in [Tan1996] assists the user in inserting the requisite path information in the user query. In [Rezende1999] entry points are defined for each database and the corresponding E/R diagram is presented, through the GUI, which is used by the user to construct the query.

More often it is seen that the user is unaware of the name or even the existence of the relevant databases. Even if the user is aware of the relevant databases, the
user probably is not aware of the relations in them. This may be due to the fact that for
small numbers of databases, it is reasonable to expect a user to remember
the names but for large number of component databases remembering names
becomes impossible. So, in order to pose a query, the user needs to learn about
each of the component databases and their constituents. This is a burden to the
user. Although, in SemQL[Lee1999], the user is allowed to pose queries without
having prior knowledge of the component database schemas, the user specifies
concepts instead of the relations or attribute names in the query.

2. The user is expected to resolve the semantic heterogeneity while specifying the
user query in the multidatabase language systems discussed above. This makes
the task of the user more complex. To construct such a query, the user requires
more information about the component databases in addition to the naming
information. Since the conflicts are mainly due to inconsistency in choosing
names to represent relations and attributes in various component databases, the
information needs to be explicitly specified by the component databases in order
to be used by the user to frame a query. SemQL[Lee1999] defines a semantic
network to handle semantic heterogeneity between various component databases.
Since the concepts are specified by the user and not provided by the system, the
concepts may be vague or imprecise and therefore may or may not find a correct
match in the component databases.

3. The user of the multidatabase language systems, discussed above, is presented
with multiple results based on the user query. It may be possible that the
information of interest to the user may be present in more than one result. The user then may find these results difficult to interpret and, therefore, may want to view these results in a way that makes them easy to interpret. One way to make these results interpretable is by integrating them. A single result arrived at after integrating these multiple results would be more desirable as related information about a particular concept will be present in a single tuple instead of being split across multiple tuples in multiple results. This will make the results easier to interpret.

1.6 Aim

In this thesis, we propose a Structure Independent Query System (SIQS) to overcome the above-mentioned problems in the following way:

1. The system aims to provide a simplified query interface that will assist the user in framing a query. This interface would enable the user to frame a query without requiring to know about the schemas of the component databases registered with the global system. This reduces the unnecessary burden to the user while framing the query. For this, we propose a new query language, Structure Independent Query Language. This query language has no reference to the structure of the component databases. The structures that are required to answer the user query are inferred at run time.

2. A methodology to achieve semantic interoperability between the component databases registered with the global system is proposed. The methodology
enables the system to resolve the various naming conflicts that may exist between the relations and attributes in the participating component databases. This will result in a system that is easy to use as the entire responsibility for handling semantic conflicts will lie with the system and not with the user.

3. In SIQS, multiple results are generated for the user query. In order to make these multiple results more desirable from the user point of view, we propose a strategy to integrate these multiple results. The strategy aims to define an order in which these multiple results should be integrated so that the final integrated result is meaningful to the user. In SIQS, the user is also allowed to pose multiple user queries in SIQL in addition to a single query in SIQL. The inter-query result integration is also supported by the system.

1.7 Organization of the thesis

The research work carried out in this thesis has been presented in the following six chapters.

In Chapter-2, the architecture of SIQS is discussed. The chapter discusses the components of the system architecture and the way they interact to answer the user query.

A methodology to handle semantic heterogeneity that may exist between multiple component databases is discussed in Chapter-3. A consistent naming methodology
is defined to assign semantically meaningful names to the relations and attributes present in the component database that wishes to participate in the global system.

Chapter-4 presents a new query language, Structure Independent Query Language, that has no reference to the structure of the component databases. A 'closeness' criterion is defined to order the multiple queries that can be generated for the user query.

The integration of multiple results based on the user query is discussed in Chapter-5. A result ordering criterion that defines the order in which the multiple results need to be integrated is discussed in this chapter.

The implementation details of SIQS are dealt with in Chapter-6. The performance of the system is also discussed in this chapter.

Chapter-7 is the concluding chapter.