It is certified that the work embodied in this dissertation entitled, "**Query Processing in Distributed Database Systems**" submitted by B.D. Badgaiyan, an M.Tech. student of School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, is original and has not been submitted in any other university or institute for any degree or diploma.

Date: JANUARY 1988

DR. Y.K.SHARMA
[ADDITIONAL DIRECTOR,
NATIONAL INFORMATICS CENTRE,
DEPARTMENT OF ELECTRONICS,
GOVT. OF INDIA, NEW DELHI]

DR. G.V. SINGH
[ASSOC. PROFESSOR,
SCHOOL OF COMPUTER AND
SYSTEMS SCIENCES,
J.N.U., NEW DELHI]

PROFESSOR KARMESHU
[DEAN
SCHOOL OF COMPUTER AND SYSTEMS SCIENCES,
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI]

ACKNOWLEDGEMENT

I express my deep gratitude to Dr.G.V. Singh, Associate Professor ,School of Computer and Systems Sciences, for providing me constant encouragement and guidance.

I am deeply indebted to Dr. Y.K. Sharma, Additional Director, National Informatics Centre, Department of Electronics, Govt. of India for kindly allowing me to work in the Systems Software Group, N.I.C. for this project.

I thank all the members of the S.S.G. for their help. I thank specially to Mr. Sujit Verma, Mr. Joshy Joseph and Mr. Santosh Kumar.

My special thanks are also to Prof. Karmeshu, Dean, School of Computer and System Sciences, J.N.U. New Delhi for his keen interest in my work.

B D Badgaiyan

(B.D. BADGAIYAN)

CONTENTS

CHAPTER - I

INTRODUCTION

In recent years, distributed databases have become an important area of information processing. The reason for this are both organisational and technological. On the organisational side the motivations for having a distributed database system are many - first of all distributed database fits more naturally with the decentralised character of most organizations, where the data is dispersed geographically across the organization. Secondly, distributed databases (DDB) are also natural solutions when several databases are already existing in the organization and the necessity of performing a global application arises. Thirdly, as compared to a centralised database (DDB) also reduces communication cost as most applications can be satisfied locally. Fourthly, it is also relatively easier to add new organizational unit (with new database) to the existing units then in the case of a centralised database. The technological reasons are first, the emergence of low cost, relatively small mini and microcomputers in 70s, which makes it possible and feasible to distribute large number of such

computers across the organization. Secondly, the development of computer network technology has facilitated the growth of distributed systems. The other advantages of a distributed system are increase of performance through a high degree of parallelism, a higher degree of reliability against system failures as there are many processors.

Formally, we can define a distributed database as [1]:- "A distributed database system is a collection of data which are distributed over different computers of a computer network. Each site of the network has autonomous processing capability and can perform local applications. Each site also participates in the execution of atleast one global application, which requires accessing data at several sites using a communication system. "

The entire data residing in the DDB system can be manipulated using a simple, userfriendly Data Manipulation Language (DML). Thus to retrieve data from the system a user simply frames his query in terms of easy to understand constructs of the underlying DML/query language. Also the user may frame his queries as if the database were not distributed at all. In general a user's query may involve

data stored at several sites and supported by different DBMSs. Thus to satisfy a query we must have an appropriate strategy so that we can carry out the above mentioned task in a cost effective manner. It is the objective of the present study to discuss such an optimal strategy for query processing for a heterogenous DDB system with special attention to data retrieving queries. At the backdrop of our study is the proposed National Informatics Centre Network (NICNET) which is to cover all the 438 districts of our country. The typology of the network is based on a site-to-site model. The distributed environment supported by the network is heterogenous i.e there are several databses supported by different DBMSs located at different sites. Also a single relation is allowed to be fragmented over several sites. But the fragmentation is only horizontal, and it is based on a distribution criterion (fragmentation predicate)- which may be a condition on the key field of a relation. Further, there is no duplication of fragments i.e. a fragment exists at only one site.

The query originates at any site and is formulated in terms of the global query language - SQL with global referents. The query may be satisfied locally (for which it

is checked first) or may require access to other sites. Broadly speaking any particular strategy for query processing must address itself to the following problems:- Firstly suitable sites where processing needs to be done should be selected. Secondly, appropriate subqueries should be generated and be distributed to the respective sites. Thirdly, partial results generated at the processing sites should be collected and be transferred to the site of origin of request.

Now, since to satisfy a given query in general there can be several different ways we should choose the one which is best with respect to the optimising criterion selected. In literature several methods of query processing have been suggested [2,3,4]. Most of them seek to minimise them transmission cost. But they differ mainly in their evaluation strategy, particularly with respect to the evaluation of joins. Wong [2] has described an algorithm which is used by SDD-1 (a system for distributed data processing - Roth [5]). It optimizes only the transmission cost and is true only for a site-to-site model of network. It also assumes that relations do not span more than one site. In this method after performing all one variable

restrictions the fragments are moved to the site with most data where the remaining processing (join etc.) takes place. After processing the result is moved to the destination site.

Hevner and Yao [3] have also proposed an algorithm which is based on Wong's work and is valid only for a site-to-site model and assumes no fragmentation of relations. It considers two different optimization:- (1) minimising transmission cost and (2) minimising network delay. The algorithm begins like that of Wong's algorithm by performing all local processing first but in looking for an optimum solution it examines more alternatives than Wong's work. Thus it finds an equal or better solution than Wong's solution. Wong's algorithm is a "greedy" algorithm that optimises for the current processing step without regard to the "global" optimization. On the other hand Hevner and Yao's algorithm does an exhaustive search for solution. But it is not clear how the algorithm can be extended to allow multiple fragments of a relation on multiple sites [6].

Stonebroker [4] has suggested an algorithm which is used in distributed version of INGRESS. This algorithm

is essentially an extension of "query decomposition technique" suggested by Wong [7] for optimization in centralised databases to the case of a distributed environment. But again this algorithm does only "local" optimization as it is primarlily oriented towards finding a sequence of subqueries which if run will "advantageous" as compared to the given query.

We shall discuss in detail the algorithm proposed by Epstein [6]. This algorithm is valid for both site-to-site and broadcost type of network. Since the underlying network (NICNET) is based on site-to-site model we shall confine ourselves only to this model. The algorithm allows for a possible fragmentation of relations. It can be extended to perform exhaustive search of possible alternatives of partitioning the given query into subqueries.

The present work is organised as follows:-

In chapter 2 we give a description of the query language SQL in terms of which queries are formulated.

In chapter 3 we discuss the algorithm mentioned above for generating the optimal sequence of subqueries.

In chapter 4 we discuss the design and implementation of Parser for the language SQL. As we shall see later the Parser is an essential input for the implementation of a query processing strategy. Particularly in heterogenous DDB system Parser is needed also to translate SQL queries into the DML of local DBMS.

Appendix A contains a listing of the program for the Parser which has been written in 'C' language on a NEC - S/1000 system.

Appendix B contains illustrations which explain the output of Parser.

Appendix C contains the list of keywords, symbols and constants in the language SQL.

CHAPTER II

The Query Language - SQL

In this chapter we shall discuss the UNIFY implementation of SQL [9] - as this version is going to be supported on the proposed distributed processing network (NICNET). SQL was developed at the IBM Research Centre as a relational inquiry and data manipulation language based on an English keyword syntax. Its structure was refined through extensive testing to produce a language easy enough for nonspecialists to use, yet powerful enough for data processing professionals. SQL is fast becoming the standard relational query language on all sizes of computers.

This implementation of SQL is based on the language description given by D.D. Chamberlin et.al. [10]. In order to adapt it to supermicros and the operating system environment, some changes have been made to the syntax.

SQL Query Facilities

A query consists of "phrases" (also called clauses), each of which is preceded by a keyword. These keywords have special meaning to SQL, and so cannot be used for record type or field names.

Some of the phrases are optional, and some of them are required. The required phrases are:

Select some data (a list of field names)

From some place (a list of record types)

The optional phrases are:

Where a condition (a true/false statement)

Group by some data (a list of field names)

Having a group condition (a true/false statement)

Into a file

The following sections take each of the phrases and show some possible queries. The example given are based on the following database :-

emp (Number, Name, Dept_No, job, manager, salary, commission)

dept (Number, Name, Location).

Select From Clause

The simplest kind of SQL query includes both a select clause and a from clause. The select clause lists the fields to printed, while the from clause tells which

record type (or types) the fields are to come from. The fields to be selected must be in the record types listed in the from clause.

Example:Select all the fields for each record of the emp record type. This will show the entire contents of the emp records. The "*" is shorthand for all the fields.

```
select *

from emp/
```

Example 2 List the employee number, job, name, and salary for every employee.

```
select number. Job, Name, Salary

from emp/
```

Where Clause

Since we rarely want to list the entire contents of a specific record type, the where clause is provided to specify selection criteria. The where clause compares a field with a constant, expression, or the results of another select clause. These nested queries will be described in more detail later. The where clause can also contain a complex boolean expression composed of selection criteria connected by and and or operators.

Example (3): List the name and location of department number 70. This illustrates comparing a field with a numeric constant.

```
select Name, Location

from dept.

where Number = 70/
```

Example (4):List the name, job, salary and commission for employees whose commission exceeds their salary.

```
select Name, Job, Salary, Commission

from emp

where Commission > salary/
```

The standard boolean operators and and or can be used to connect simple comparisions to form complex expressions.

Example (5): List the name, job, salary and department number for the employees who work in department 10 and are either clerks or make less than or equal to $1200.

```
select Name, Job, Salary, Dept_No

from emp

where Dept_ No = 10 and
```

```
(job = 'clerk' or salary <= 1200)/
```

Boolean expression can be negated in whole or in part to select those records that do not match a specified criterio.

Example (6): List the name, job and salary of all employees who are not salesmen or who make less than $ 2000. This uses the not operator to negate an entire expression.

```
select Name, job, Salary
from emp
where not (job = 'salesman' or salary >= 2000)/
```

Set Inclusion

In many queries we may want to compare a field to a list of values, not just a single value. For example, let us consider selecting all the employees who are in departments 10, 20, 30 or 40. With the standard operators, this becomes a sequence of equalities connected by ORs, such as

```
Dept_No = 10 or Dept _ No =20 or Dept _ No = 30 ....
```

SQL provides a set inclusion notation to make this kind of query easier.

Example (7): List the name, job and department number for employees in departments 20, 30 or 40.

```
select Name, Job, Dept _ No
from emp
where Dept _No in ≤ 20,30,40 >/
```

Unique Operator

If a query doesn't select a primary key field from one of the record types, it is possible for that query to produce rows that are exact duplicates of each other. This is because only the primary key is required to be unique. Sometimes, these duplicates are not desired. The unique operator is provided to suppress duplicate information in a query result.

Example (8): List the different job titles in the company.

```
select unique job
from emp/
```

Order by Clause

All the previous queries returned their results in an order determined by SQL. Even though the unique operator sorts its output, we are still not able to direct the order of output. The order by clause lets us explicitly specify

the sequence of the rows that result from a query. The default sort sequence is ascending (ase), with STRING fields sorted in alphabetic order from A to Z.

Example (9): List every employees number, name and job, sorted by employee number.

```
Select Number, Name, Job
from emp
order by Number/
```

One can sort also by more than one field, and specify the direction, whether ascending (asc) or descending (desc), for each field in the sort.

Example (10): List every employee's department number, name and job, by ascending name within descending department number.

```
select Dept _ No, Name, Job
from emp
order by Dept_No desc, Name asc/
```

Aggregate Functions

SQL provides 5 different built-in aggregate functions to allow calculation of aggregate items in a query result. The functions are count (*), min, max, sum and avg. Aggregate functions are only valid when used in select or

having clauses. One cannot use an aggregate function directly in a where clause, although we usually achieve the same effect by using a nested query. An aggregate function only applies to a group of records with a common characteristics for example the average salary of employees in department 10.

Aggregate functions are most commonly used in conjunction with the group by clause. A group by clause explicitly paritions the selected records into groups for which the indicated aggregate functions are computed. If there is no explicit group by, than any aggregate functions are assumed to apply to the entire set of selected records.

Example (11): Compute in list the total number of employees in department 10.

```
select count (*)
from emp
where Dept _ No =10/
```

Example (12): List the job and average of salary plus commission for all salesmen. The job can be listed in this example because all the selected records have the same job - salesman. select Job, avg (Salary + Commission)

```
from emp
```

```
where job ='salesman'    /
```

Group By Clause

The group by clause is provided to allow computation of aggregate functions on groups of records that have common characteristics. Thus using a group by clause without an aggregate function has no meaning. The effect of a group by is to sort the selected rows by the indicated fields, and then perform the aggregate functions at each level break. This results in the output being sorted also.

Example (13): For each department, list the department number, count of employees and sum of salary plus commission.

```
select Dept_No, count(*), sum(salary + Commission)
from emp
group by Dept _No/
```

Aggregate functions can also be applied to the result of other aggregate functions. This lets us compute such items as the maximum average or the averagae count. When used in this way, aggregate functions require a group by clause. The result is computed as follows. First, all qualifying records are selected using the where clause, if any. Then, they are sorted according to the fields in the

group by clause, and the inner aggregate function is computed. The outer function is then applied to these results. Since this second level of computation removes all identity from the groups, a nested query is required to list fields other than the function result.

Example (14): List the maximum average monthly salary for all jobs, except for the job of "President". It should be noted that if we wanted to see what job this was, we would have to perform a nested query.

```
select max(avg(Salary)
from emp
where Job ^= 'president'
group by job/
```

Nested Queries

Nested queries allow us to answer a whole new set of questions that cannot be answered using the capabilities of SQL presented so far. Nesting lets us use the results of one query as input to another, so we can use the results of one question in answering another one.

Example (15):Find the name and job iof the employee who makes the maximum salary plus commission.

```
select Name, Job
from emp
```

```
where (salary + commission) =
        select  max (salary + commission)
        from emp/
```

This query works by first evaluating the inner query to get a value for the maximum salary plus commission. This value is then used as a constant to the outer query which finds the employees (there could be more than one) who make that amount.

Queries can be nested to any level. The following four-level query finds the person with the second highest compensation. The method is to find the name of the person with the maximum compensation, and the find the maximum compensation among those left. This also shows that we can compare an expression with the results of a nested query.

Example (16): List the department number, name, job, and salary plus commission of the person with the second highest compensation.

```
select Dept_No, Name, Job, Salary + Commission
from emp
where salary + commision =
     select max (Salary + Commission)
     from emp
     where Name ^=
```

```
select Name
from emp
where Salary + Commission =
Select max(salary+ commission)
from emp/
```

Nested queries can also be used as part of more complex expression. In this case, the inner query must be ended with a semicolon (;), so SQL can figure out where the nested query ends and the rest of the boolean expression begins.

Example (17):List the department number, name, job, total compensation and commission for the salesman with the maximum total compensation, and for all the salesmen in department 20. Sort the result by salary within commission in descending sequenve, so the high earners come first.

```
select Dept_No, Name, Salary + Commission, Commission
from emp
where Salary + Commission=
     select max (Salary + Commission)
     from emp
     where Job = 'salesman';
or (job = 'salesman' and Dept_No =20)
order by Commission desc, Salary desc/
```

Having Clause

The having clause lets us select some of the groups formed by a previous group by clause and reject others, based on the results of another selction using one or more aggregate functions. This gives us capability equivalent to using an aggregate function in a where clause, which is not allowed.

Example (18):List the department number and average salary for departments having an average salary over $2000.

```
select Dept_No, avg(Salary)
from emp
group by Dept_No
having avg(Salary) > 2000/
```

When a query contains both a having and a where cluase, the query is evaluated as follows: First the where clause is applied to select qualifying records, then the groups indicated by the group by clausé are formed; then the having clause is applied to select qualifying groups.

The having clause can also contain nested queries. A query nested in a having clause is evaluated in the same way as a query nested in a where clause.

Nested queries can be used in both the where and having clauses at the same time.

Join Queries

Up until now, all the queries we have been doing involved only a single record type. However, an SQL statement may list fields from any number of record types in a single query. Queries that list fields from several record types are called join queries, because they combine, or "join", the different record types together. The different record types to be involved in the query are listed in the from clause, in any convenient order. SQL then determines what is the most efficient method of performing the selection and qualification.

The fundamental concept underlying join queries is that of the Cartesian product. Conceptually, a join query first forms the cartesian product of the record types, and then "filters" the results by the conditions in the where clause. Thus a join query without a where clause does in fact list the Cartesian product of the record types.

Example (18): List the employee name and all the fields from the department that the employee works in.

```
select emp.Name, dept.
```

```
from emp, dept
where Dept_No = dept, Number/
```

The where clause in a join query can contain expressions that use fields from any of the record types involved in the join. The expression is not limited to being an equality.

Self Join

Sometimes it is necessary to join a record type with itself. In our sample data base, the emp record type contains a field that indicates who the employee's manager is. This is simply the number of another employee. We could therefore join the emp record type to itself, using the employee's number and number of his manager.

The best way to think of queries like this is to imagine that there are two copies of the emp record type - one that contains employees, and one that contains managers. SQL doesn't really make a copy, but achieves the same effect by letting us give record type a temporary name. We are free to join these record types just like "real" ones. The following query uses the record types emp and mgr. The name mgr is merely a temporary name given to the emp record type.

Example (19):List the employees' name and salary, and their manager's name and salary, for employees who make more than their manager.

```
select emp.Name, emp.Salary, mgr.Name, mgr.Salary
from emp, emp mgr
where emp.Salary > = mgr.Salary and
      emp.Manager  = mgr.Number/
```

CHAPTER III

Optimisation of query decomposition

Before we proceed with the actual strategy, let us first examine the architecture of a typical distributed database system. The following figure (fig.1) shows the elements of a typical DDB system[1].

At the top level is the **global schema**. It defines all the data which are contained in the DDB as if the database were not distributed at all. Using the relation model [8], the global schema consists of a set of global relations.

At the next level is the **fragmentation schema**. This defines the mapping between the relations and their fragments. This mapping is one to many i.e. several fragments correspond to one global relation. but only one global relation corresponds to one fragment.

The **allocation schema** defines at which site(s) a fragment is located. If we have a single fragment available at more than one site, then we have what is called a

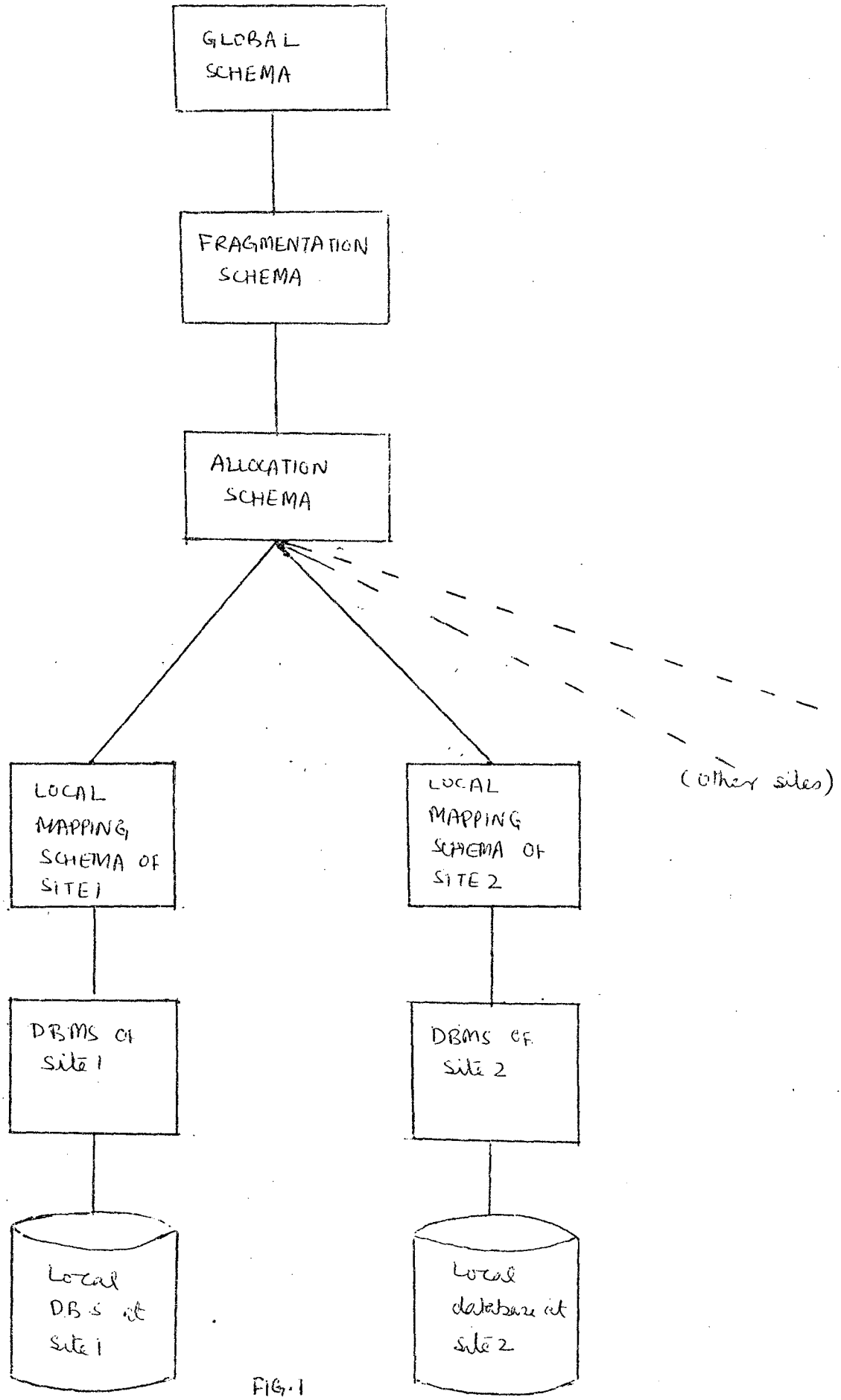

GLOBAL SCHEMA
FRAGMENTATION SCHEMA
ALLOCATION SCHEMA
LOCAL MAPPING SCHEMA OF SITE 1
LOCAL MAPPING SCHEMA OF SITE 2
(other sites)
DBMS of site 1
DBMS of site 2
Local DB's at site 1
Local database at site 2

FIG. 1

redundant DDB, otherwise the DDB system is set to be non-redundant.

The **local mapping schema** serves to map the fragments at a particular site to the object in local DBMS, and it depends on the local DBMS - thus for a heterogenous system we have different types of local mapping at different sites.

At the next level of the architecture we have the local DBMSs with their associated local databases.

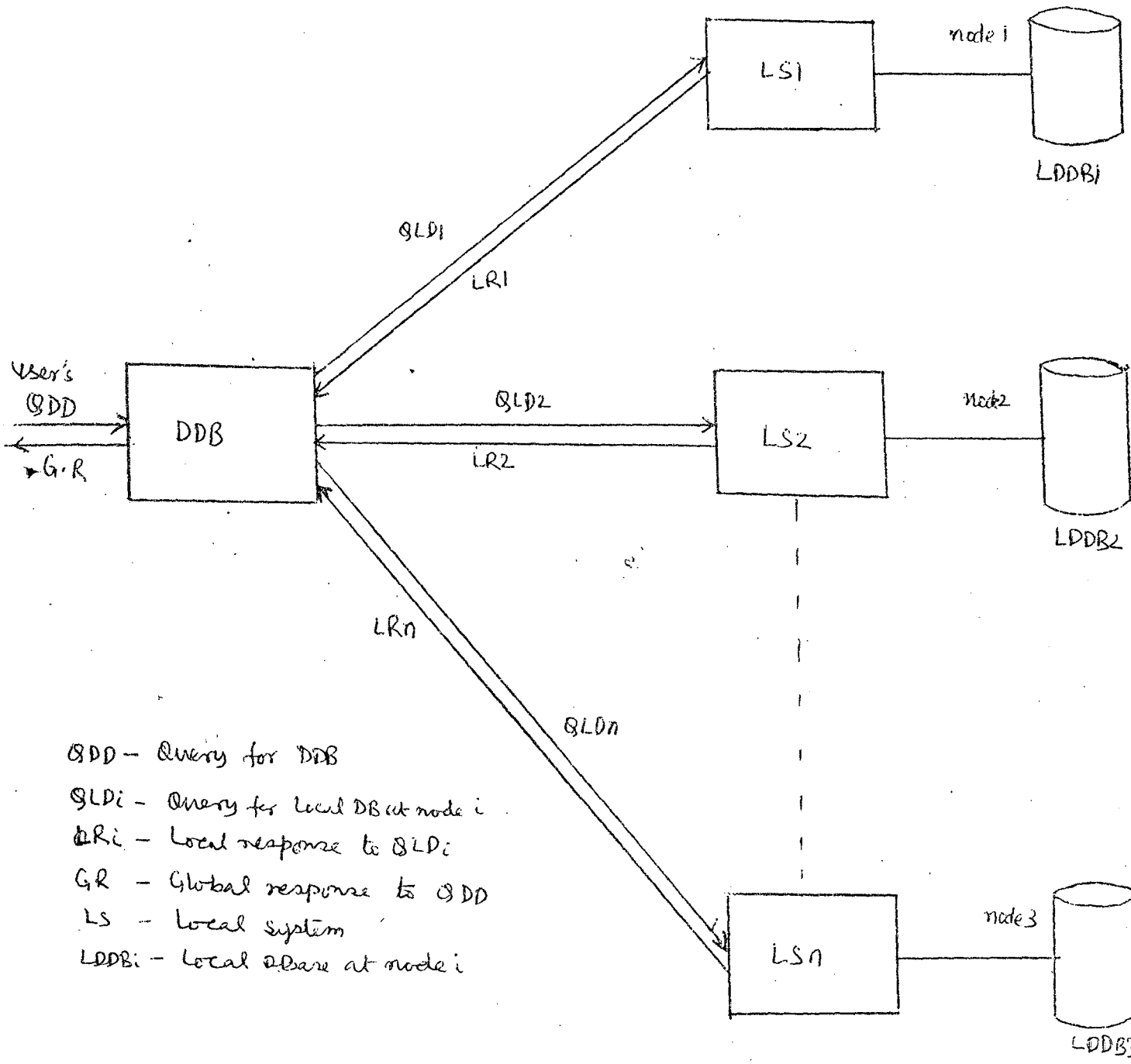
Only the top three layers are site independent and they serve to define the distributed database much in the same way that a global directory does for a centralised database.

The abovementioned architecture has the advantage that it provides for location as well as fragmentation transparency i.e. a user works only on global relations without being concerned about with the location and fragmentation aspects.

In the light of the above architecture we shall now give an outline of the steps involved in query processing in a DDB system.

A user formulates his query at any node of the network in a global query language. A query may possibly involve a set of data stored over several nodes. To process such a query the local system at access node has to be analysed the distribution of the requested data and consequently decompose the query into a set of subqueries; each of them is a query for local data. The local system perform the subqueries and send back the requested data, if any, to the requesting node. (It is possible that for performing subqueries some data may need to be transferred from other sites). The access node finally synthesises all local results to provide the user with global response to the query. The overall mechanism is illustrated in the following figure (fig.2)

Thus query processing in a DDB corresponds to the translation of requests, formulated in a high level language on one computer of the network, into a sequence of elementary instructions which retrieve data stored in the distributed database. The following figure (fig.3) shows



QDD – Query for DDB
QLDi – Query for local DB at node i
LRi – Local response to QLDi
GR – Global response to QDD
LS – Local system
LDDBi – Local DBase at node i

FIG 2

the main software module which implement this evolution of the query [B].

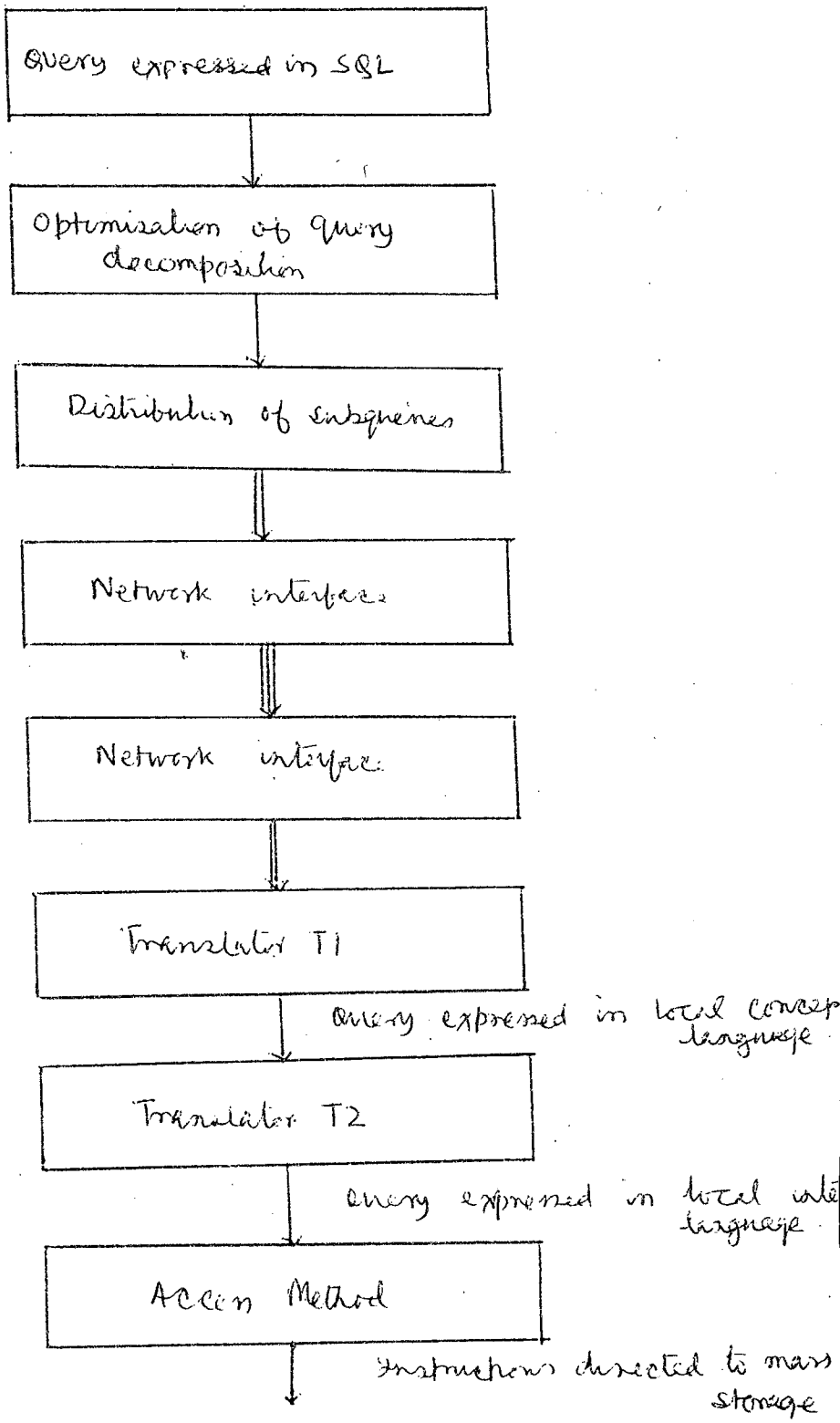


FIG 3

The optimisation process produces as output a sequence of operational commands to the local databases, which are optimal according to the optimisation strategy. Accordingly subqueries are generated and are distributed by the network. The translator T1 translates the subqueries into the language (DML) of the local DBMS. Each command is then further analysed by the local DBMS of the remote computer and the optional local data retrieval strategy is performed. The execution of these commands may however be postponed if there is a need for receiving data from other computers.

We shall now discuss the specific problems of query processing optimisation.

Optimisation

The optimisation of query processing consists of decomposing the query into subqueries and determining a sequencing of the subqueries so that the obtained strategy is optimal with respect to the selected **cost function.**

The cost function:- It reflects the requirements of the system and may be obtained by weighting the following factors:-

(a) Total response time

(b) Total usage of local resources (CPU utilisation, I/O operations)

(c) Network traffic

(d) Parallel load distribution on the computers of the DDB.

Of course these factors are not independent (in fact, some of them are contradicting) therefore, it is not possible to subdivide the problem and to optimise each factor separately. The optimisation algorithm that have been created for a centralised environment usually weight in the cost function the elapsed CPU time and the I/O operations. The problems in distributed environment are more complex - e.g. increase in parallelism in processing leads to increase in network traffic while increase in parallelism helps to cut down the response time. But the major bottleneck in DDB system has been found to be the inter-computer communications - as the network transmission speed is very slow as compared to local processing speeds, hence most of the algorithm in literature take only communication cost in their optimisation algorithm. Let us now discuss the specific optimisation method mentioned earlier.

An algorithm for Optimisation of query decomposition:- We will confine ourselves mainly to minimisation of the amount of data transferred between nodes. In the end we will relax our optimisation a bit using a heuristic to allow for increase in parallelism - without unduly increasing the amount of data transferred. Also we shall work in an overall relational model of DDB system. It is assumed that a copy of global schema, fragmentation schema, and allocation schema is available at every site. Also every site has a copy of fragmentation criterion. Further, the transmission cost between each node has been taken to be equal.

Before applying any specific optimisation technique the first step that we must perform is to parse the query. After parsing we know what all are the clauses present in the query, and also what relations are involved in it. We can then refer to the schemas mentioned above and find out what sites and what fragments are involved in the query.

A simple query like :

```
select name
from emp
```

which does not involve any condition (no where clause) can be directly sent to the sites involved in the form of appropriate subqueries.

But if the query contains the where clause we first convert the conditions into a Conjunctive Normal Form. This is done so that we can eliminate portion of data which do not satisfy even one of the conditions from further consideration. We then look for one variable restrictions on field(s) which serve to define the fragmentation. A site for which any of these conditions is false is ruled out from further consideration.

The main optimisation is required in the evaluation of joins since the relations involved in the join may be distributed and to evaluate the join the relation fragments must be transferred from one site to another.

The strategy that we are considering - called the fragmented processing technique [6], reduces both transmission cost as well as response time. Response time is reduced by increasing the degree of parallelism and it can be shown that this method results in less transfer of data because of judicious choice of processing sites.

We shall assume that our query involves join between n relations such that all the joins are connected i.e. we don't have disconnected joins like R1 joined with R2 and R3 joined with R4. We shall confine ourselves to the site-to-site model of network.

The technique essentially consist of choosing one relation (Rp) which is not moved (i.e. it remains fragmented) and choosing k processing sites. The remaining relations , Ri, $i \neq p$, are moved to the k processing sites. Processing then begins on all k sites and the result is the result of union of results on the K sites. The basic question here is how to choose Rp and K. This requires an analysis of communication costs involved. Evidently following transfers of data must be made in our strategy:-

(i) for each processing site Sj, $R^j_{i,i\neq p}$ must be moved to all other K-1 processing sites.

(ii) for each nonprocessing site Sj, $R^j_{i,i\neq p}$ must be moved to the K processing sites and R^j_p must be moved to one processing site.

For simplicity, we will number the processing sites to be S1, S2 ----Sk. The formula for the number of bytes which must be moved is then:-

$$\text{Comm} = \sum_{j=1}^{K} C_{K-1} \left[\sum_{i \neq p} |R_i^j| \right] + \sum_{j=K+1}^{N} C_K \left[\sum_{i \neq p} |R_i^j| \right] + \sum_{j=K+1}^{N} C_1 \left[|R_p^j| \right]$$

where C_K is the cost to send data to K sites

The first term comes from (1) above. The (ii) and (iii) come from (2) above.

Epstein [6] has shown that for a site-to-site model of network the above communication cost is minimised by choosing Rp and K in the following manner:-

(i) choosing Rp to be max($|R_i|$) (i.e. the relation with the highest cardinality) and choosing every site Sj to be a processing site for which

$$\sum_{i \neq p} |R_i| < \sum_{i} |R_i^j|$$

(i.e. for the site j the amount of data required to be brought in when it is made a processing site is less .than the data to be moved out if it were to be a non-processing site).

or

(ii) if the above condition is found to be not true for all sites, then, choose only one site Sj as a processing site which has

$$\max_j [\Sigma_i |R_i^j|]$$

(i.e. the site containing the largest amount of data).

We can illustrate the method of fragmented processing technique by means of the following example:-

Suppose we have two relations:-

Supplier (sno, sname, city)

Supply (sno, jno, amount)

Let the relations be fragmented (horizontally) as follows:-

	site 1	site 2	site 3	site 4
No. of tuples of relation supplier	200	25	300	-
No.of tuples of relation supply	100	100	-	50

Let there be a query :- "find the names of all suppliers which figure in the supply list ". In SQL this query would be

```
select sname
from supplier, supply
where supplier. sno = supply. sno ≠
```

We shall evaluate this equijoin using the technique mentioned above.

We shall first decide on relation Rp. Clearly cardinality of supplier (550) is greater than cardinality of supply (250), so supplier is Rp.

Now to choose processing sites we used a criterion mentioned above. We have $\sum_{i \neq p} |R_i| = 250$

For site 1 $\sum_i |R_i^j|$ = 300

since $\sum_{i \neq p} |R_i| < \sum_i |R_i^j|$ site 1 is a processing site.

For site 2 $\sum_i |R_i^j|$ = 125

This is not a processing site.

For site 3 $\sum_i |R_i^j|$ = 300

This also is a processing site.

Site 4 is not considered as Relation Rp is not present there.

Let us now compute the cost of evaluation of the join:-

For site 1, cost = 100+50+25 (we have moved contents of Rp at site 2 to site 1)

For site 2, cost = 100 + 100 = 200

Therefore total cost = 375.

If we now evaluate the cost by transferring all the data to one site (site with maximum data - site 3 in our case) we get

cost = 300 + 125 = 425.

Clearly our method is less costly. Of course, in our case the result of join is fragmented at site 1 and site 3 so an additional transfer is required . But that will be the case even for the crude method if the query originates at site 1 or site 2. It is possible that the join may blow up the resultant relation.

Yet another level of optimisation in communication cost is possible for the case of queries involving joins between more than two relations. This is done through a

method called query splitting tactic [6]. The essential idea is to look for an intermediate result which is such that if this result were evaluated first and used in subsequent evaluations, the overall communication cost will be least. In this method the query q is split into two parts q' and q". q" uses the outcome of q'. q' successively contains all the combinations (from i=2 to i = n -1) of the relations involved in query q. For each case we estimate the size of resultant relation and the cost of evaluating q' using fragmented processing technique. Also we use the estimate of resultant relation and calculate the cost of evaluating q" using FPT. The total communication cost is the sum of the two costs. Out of all possible total costs thus calculated we pick up the lowest one and do the corresponding splitting of the query and proceed with further processing.

A possible modification of the tactics is to evaluate the q" cost using the query splitting tactics again. This would be an exhaustive search.

It should be noted that correct estimation of result-size plays an important role in this tactics. But

while an extensive statistics about the data can not be stored, it is nevertheless necessary to store some information about the data. Usually a one bit of information is stored in the global schema which tells whether a domain is nearly a primary key or not.

Yet another way would be to decide on further processing strategy after the evaluation of every step so that an accurate knowledge about the result-size is available. But this method increases processing time. Also it is not possible to revert the decisions if it turns out that the strategy has been expensive enough.

Having identified as to how the query is to be split (if at all) and also the relations to be moved, we now concentrate on the actual sites where data is to be picked up for transfer. Our objective here is to pick up only relevant data. For this we perform following operations at each site before transfer:-

(i) Apply all one variable conditions applicable for the fragments on that site.

(ii) Project only those fields from the relation fragments, which:-

(a) either are in the target list (i.e. in select clause)

(b) or are involved as joining fields in the joins in the query.

Processing of Aggregates

Some optimisation specific to aggregate functions can be done. For example, the aggregates that range over only one relation are processed on individual sites and the aggregated results are transmitted back to the requesting site and they are combined to produce the final result.

Aggregates which involve more than one relation can be performed by first retrieving the values to be aggregated into a distributed temporary relation and then aggregating on that temporary relation.

Minimising response time

Increase in parallelism can decrease processing time but it increases the communication cost. But still we can use some heuristic to improve response time by increasing

parallelism. Thus for the model presented we can change the equation.

$$\sum_{i \neq p} |R_i| < \sum_{i} |R_i^j|$$

$$T \sum_{i \neq p} |R_i| < \sum_{i} |R_i^j|$$

Where T is a heuristic value between 0 and 1 . When T = 1 communication costs are minimised. When T = 0 all sites become processing sites. We can choose a suitable value of T after some experimentation.

CHAPTER - IV

Design and implementation of Parser for the SQL

The parser we have developed supports almost all the features of the language SQL as discussed in chapter II (it does not include set inclusion). The implementation of a Parser requires the execution of following two phases:-

(i) the lexical analyser

(ii) the syntax analyser

The lexical analyser reads the input character by character (from left to right) and generates tokens as soon as a valid construct has been encountered.

The syntax analyser checks for the syntactical correctness of the sequence of tokens generated and if correct, it builds the corresponding parsed tree i.e. it stores the input in an appropriate strucutre.

We have combined the above two phases into one pass. The syntax analyser calls the lexical analyser as and when it needs a taken. Also our parser works in a top-to-bottom fashion using a recursive descent procedure.

The output of the parser is a pointer to the root of the tree.

We will now discuss each phase separately.

(1) **The lexical analyser**:- A valid token returned by the lexical analyser is one of the following:

(i) a keyword

(ii) an identifier

(iii) a constant

(iv) a symbol

The list of keywords, symbols and constants is given in appendix C.

An identifier is a string of alphabets and digits, starting with an alphabet. Also the identifier may contain a dot (.) (to represent constructs like emp. name - where emp denotes the relation which has attirbute name)

For recognising the various tokens the adjacent transition diagram has been used.

The above transition diagram has been implemented in the form of a transition table, with input symbols - after mapping them to integer values - representing coloumns and states representing rows. Each table entry represents

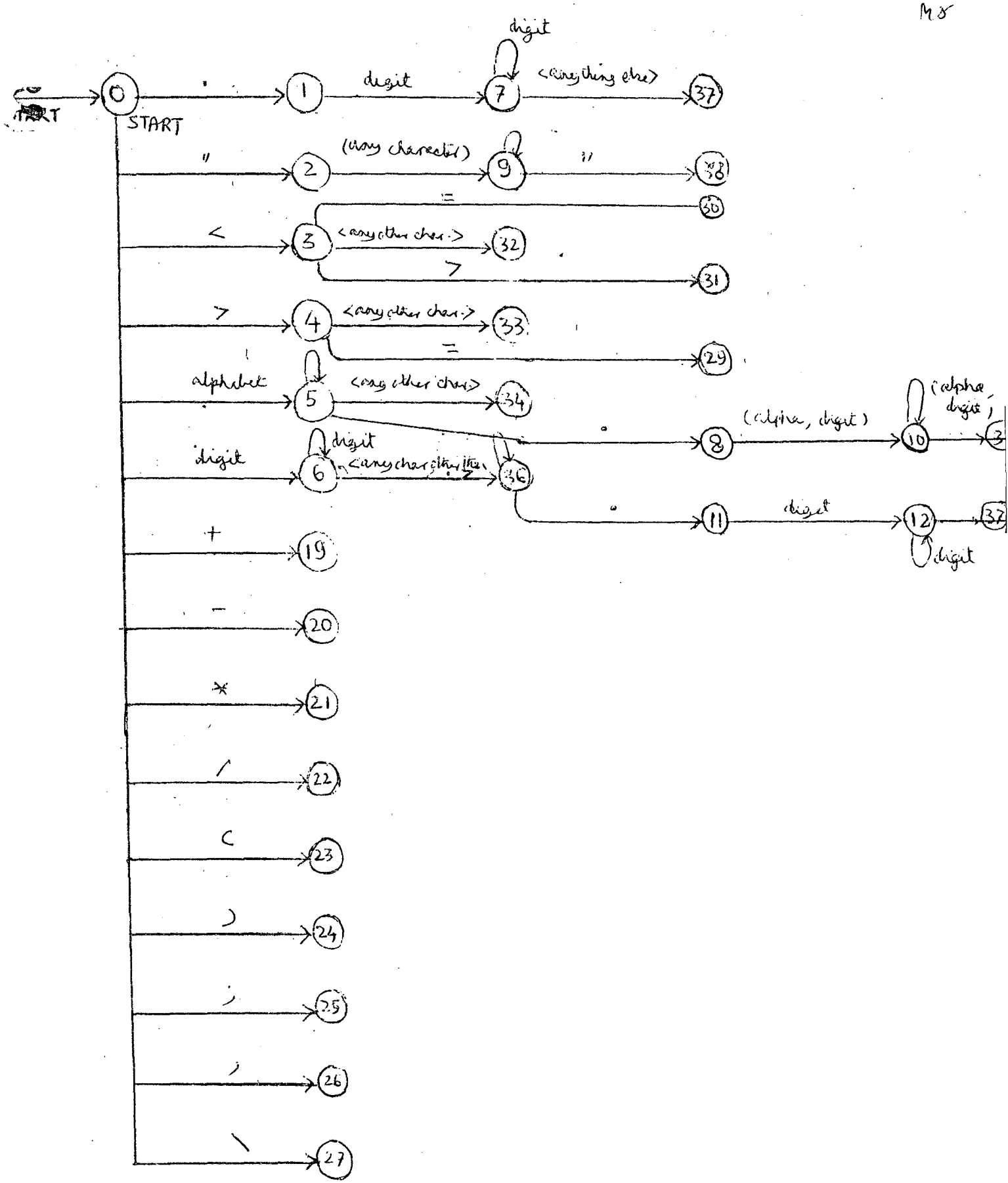

START
0
1
digit
7
digit
<anything else>
37
"
2
(any character)
9
"
38
=
30
<
3
<any other char>
32
>
31
>
4
<any other char>
33
=
29
alphabet
5
<any other char>
34
digit
6
digit
<any char other than 2>
36
8
(alpha, digit)
10
(alpha, digit)
11
digit
12
digit
32
+
19
-
20
*
21
/
22
(
23
)
24
;
25
,
26
\
27

FIG 4

the next state of transition with the given inital state represented by the row number and with given input symbol-represented by the coloumn number.

We start with the state 0 and as characters are read we move from one state to the next. At the same time we keep storing the characters in a string. On reaching a final state the transition process stops. We use this final state for mapping the given string of characters to a token.

Every token returned by the lexical analyser has three attributes - the name of the token i.e. the actual input, the token number and the precedence number associated with the token (the precedence number is needed as we shall see later for building the parsed tree.

The precedence associated with various tokens are given in Appendix (C). What is actually returned by the lexical analyser to the syntax analyser is the address of the location where the given token with all its attributes is stored. For this purpose we maintain four separate tables for keywords, symbols, identifiers and constants.

All the keywords are a priori stored in the keyword table. A string of alphabets is first checked for a keyword. If it is a keyword we return the address of the corresponding keyword table entry. If the given string is not a keyword, then like for any identifier we check for its presence in the identifier table, if present, we return the corresponding identifier table entry address, otherwise we install the new identifier in the table and return its address.

A constant is similarly checked for its presence and installed in the constant table, if necessary.

A symbol is matched against the symbol table entries (which has all symbols installed in it) and the address of its match is returned.

The following program (in pseudo code) sketches the working of lexical analyser (it is not exhaustive).

Program Lexical Analyser

```
Begin
    S = ' ';  [Initialise S to an empty string]
    state = 0 ; [Initial state]
```

```
C = GETCHAR; [Get one character from input]

S = S CAT C; [concatinate C with S]
CASE C OF

     Alphabet: While (state ≠ final state)

               Begin
                   C = GETCHAR;
                   S = S CAT C;
                    state  = nextstate; [nextstate is
                         the state resulting from a
                         transition to a new state, on
                         input]
               end;
               CASE STATE OF
                   F1   :     Lookup (Keytab); [F1
                              corresponds to a string
                              of alphabets]
                        If (found)
                        return (& keytab (entry));
                        else
                        begin
                        lookup (idtab);
                        if (found) return (&idtab (entry));
                        else
```

```
                    begin
                    instal (idtab);
                    return (& idtab (entry));
                    end;
               end;

          F2   :   lookup    (idtab); [ F2
                              corresponds to
                              a string of
                              alphabets and
                              digits]
               If (found) return (& idtab (entry));
               else
               begin
               install (idtab);
               return (& idtab (entry));
               end;

Digit:    While (state ≠ final state)
          Begin
               C = GETCHAR;
               S = S CAT C;
          end;
          CASE STATE OF

          F3 : Const-type = integer;
```

```
                    lookup (constab);
                    If (found) return(&constab (entry));
                    else
                    begin
                    install (constab);
                    return (& constab (entry));
                    end;

               F4 : Const-type = real;
                    [then do the same as above]

     Symbols:  While (state ≠ final state)
               Begin
                    C = GETCHAR;
                    S = S CAT C;
               end;
               lookup (symtab);
               return (& symtab (entry))
End.
```

We shall now discuss the working of syntax analyser.

(2) **Syntax Analyser**: For checking the grammatical correctness of the query and for building the corresponding parsed tree (if gramatically correct) we follow a left to right recursive descent procedure.

In this method, we have one recursive procedure for each nonterminal U, in the grammar which parses phrases for U. We begin by looking for a phrase for some starting nonterminal (query in our case). The procedure finds its phrase by comparing the input at the point indicated with right part of rules for U, calling other procedures to recognise subgoals when necessary [12].

Thus with reference to the grammar given (below), we begin by looking for a phrase for query. For this, we first look for presence of the item SELECT in the input, if present we now try to satisfy the expression subgoal. For this we compare the input portion of program (after SELECT0 with the right parts of rules for expression. Satisfaction of this subsubgoal may require comparision with the right hand side of the rules for other nonterminals). If this subgoal is satisfied we look for the presence of FROM in the query, if present, we look for the presence of a list of relation names in the query and so on.

It should be noted that as a particular subgoal is satisfied the routines called also build the corresponding parsed structure.

Before going into the details of the actual implementation of the syntax analyser let us formally write down the grammar for the SQL . With reference to the the language defined in chapter II we have the following production rules:

<query> → SELECT [unique] <E1> FROM <rellist>
[label] [WHERE <E2>] [GROUP BY <field list> HAVING <E3>] [ORDER BY <field list>] [ASC or DESC] [INTO <filename>]/

<query> → SELECT [unique]<E1>FROM<rel - list>
[label] [WHERE <E2>] [GROUP BY <field list>] [ORDER BY<field list>] [ASC or ESSC] [INTO<filename>]/

Where [ ] indicates that the enclosed item is optional and may be left out. This rule says that a query starts with a select clause continues with a form clause and then includes none, one or more optional clauses. The clauses must occur in the order given, even though some of them may be omitted in a specific query. Also the query must end with a /.

We now consider rules for each of the nonterminal present in the rule 1.

<E1> → <E1> <OP1> <E1>

<E1> → field | rel.field | constant | agg.funname(<E1>)

<E1> → (<E1>)

<E1> → (-<E1>)

<OP1> → +|-|*|/

<E2> → <E1'> <OP2> <E1''>

<E2> → <E2> <OP3> <E2>

<E2> → NOT <E2>

<E2> → (<E2>)

<OP2> → >|<|=|≤|≥|<>

<OP3> → AND | OR

<E1'> has the same rules as for <E1> except in rule(2) (of <E1>) aggregate function is not allowed for <E1'>

<E1''> → <querry>

All other rules of <E1''> are same as for <E1'>

<field list> → field, <field list>

<field list> → field

<field list> → relation.field, <fieldlist>

<field list> → relation field

<rel list> → relation, <rel.list>

<rel list> → relation

We now give an outline of the implementation of the syntax analyser (in pseudo code). It is only a sketch of the actual routine and is not exhaustive.

```
Program Syntax Analyser

Begin

    LEX; [LEX is a call to the lexical analyser for getting
         the next token]

    If  nexttoken = "select"; [nexttoken stores the token
                               returned by LEX]
    then EXPR(1);   [EXPR(1) parses arithmatic expressions
                     valid for the select clause].

    LEX;
    If nexttoken = "from"
    then REL-LIST; [The REL-LIST checks for presence of
                    relations list and stores them if found]

    LEX;
    If nexttoken = "where"
    then WH-CLAUSE; [WH-CLAUSE parses the remaining query
                     once the item where has been found]

   Else if nexttoken = "Group By"
   then GP-CLAUSE; [GP-CLAUSE parses the remaining query
```

```
                    once the item Group By has been found]
     Else if nexttoken = "Order By"
     then ORD-CLAUSE; [ORD-CLAUSE parses the remaining  query
                    once the item Order By has been found]
     Else if nexttoken = "Into"
     then INTO-CLAUSE; [INTO-CLAUSE parses the remaining query
                    once the item Into has been found]

     Else if nexttoken = "/"

     then exit; ["/" marks the end of the query].

     Else error;

End.
```

Let us consider the procedures called above, one by one.

Procedure WH-CLAUSE

```
Begin
     EXPR(2);  [EXPR(2) parses boolean expressions, which do
               not contain agg.function as agg.functions are
               not allowed in WHERE clause]

     If nexttoken = "Group By";

     then GP-CLAUSE;
     Else if nexttoken = "Order By"
```

```
        then ORD-CLAUSE;
        Else if nexttoken = "Into"
        Then INTO-CLAUSE;
        Else if nexttoken = "/"
        Then exit.
        Else error;
End.

Procedure GP-CLAUSE

Begin
        Field-List;     [Field-list checks for the presence of
                        list of fields and if present it stores
                        them]

        If nexttoken = "Having"
        then HAVING-CLAUSE; [Having-Clause parses the remaining
                            query once the item having has occurred]

        Else if nexttoken = "Order By"

        Then ORD-CLAUSE;

        Else if nexttoken = "Into"

        Then INTO-CLAUSE;

        Else if nextoken = "/"
```

```
        Then exit;
        Else error;
End.
Procedure HAVING-CLAUSE
Begin
        EXPR(3);  [EXPR(3) parses boolean expressions which may
                  contain  aggregate function as  agg.functions
                  are allowed in Having Clause]

        If nexttoken = "Order By"
        then ORD-CLAUSE;

        Else if nexttoken = "Into"

        then INTO-CLAUSE;

        Else if nexttoken = "/"

        then exit;
        Else error;
End.

Procedure ORD-CLAUSE
Begin
        Field list;
        If nexttoken =  "Into"
        then INTO-CLAUSE;
        Else if nexttoken = "/"
```

```
      then exit;
      Else error;
End.
Procedure INTO-CLAUSE

Begin
      filename; [filename checks for a valid filename and
                stores it if present]

      if nexttoken ≠ "/"
      then error;
      Else exit;
End.
```

Now we consider the routine for parsing expressions.

```
Procedure EXPR(n)

Begin
      LEX;
      if (n=1)

      then if ((nexttoken = "OPR") or (nexttoken = "OPL"))

      then error; [The select clause cannot have relational
                (OPR) or logical (OPL) operators]
      if (n=2)
      then if (nexttoken = "agg.function name")
```

```
then error;   [The where clause cannot contain an
              aggregate function]

if (flag = 0)
then if (nexttoken = "an identifier or a constant or an
              agg.function name or "(" ")
then begin    [flag is a global variable indicating
              the start of an expression. Instially it
              is set to 0 and is used for checking the
              valid tokens at the begining of an
              expression. Once the expression has
              started it is set to 1.
postfix;      Postfix is a procedure which converts
              the input stream into a postfix notation.
EXPR(n);
flag = 1;
previous token = nexttoken;  [previous token stores the
                             current token]

Else if (flag=1)
begin
if (nexttoken = "/" or "Group By" or "Order By" or
"Into") then break; [if any of these tokens occur we
                    exit from the EXPR(n) routine]
```

```
[the following lines do syntactical checking by
comparing the previous token with the nexttoken].
if (previous token = "Identifier or a constant")

then if (nexttoken ≠ "Operater" or ")" )
then error;
else begin
     postfix;
     previous token = nexttoken;
     EXPR(n);
     end;

else if (previous token = "Operater")
     then  if (nexttoken ≠ an identifier or a  constant
     or an agg.function name or "(" ) then error;
     else begin
          postfix;
          previous token=nexttoken;
          EXPR(n);
          end;
else if (previous token = ")" )
     then if (nexttoken ≠ "operater" or ")" )
     then error;
     else begin
          postfix;
          previous token = nexttoken;
```

```
                EXPR(n);
                then end;
        else if (previous token = "(" )
        then if  (nexttoken ≠  "("  or "an  identifier"  or  a
             "constant" or an "agg.function name") then error;
             else begin
                  postfix;
                  previous token = nexttoken;
                  EXPR(n);
                  end.

        else if (previous token = "an aggregate function name")
        then if (nexttoken ≠ "(" ) then error;
        else begin
             postfix;
             previous token = nexttoken;
             EXPR(n);
             end;

 Emptystack (stack); [this  procedure empties the contents of
                     stack onto an output stream]

 Build_expression (output stream); [after  converting  the
                     input  sequence (if correct) to  postfix
                     we call the procedure "build expression"
                     which builds the tree for the expression
                     using the postfix form which is stored in output
                     stream.

End.
```

```
Procedure Postfix

Begin
     O = " ";   S = " "; [O is the output stream which will
                          finally have the postfix form of
                          expression.   S is the Stack.
                          Initially both are empty]

     i = 1; j = 1;

     If (nexttoken = "identifier or constant");
     then begin
          O [i] = nexttoken;
          i = i+1;
     else if (nexttoken = "operator")
     then begin
          if (S = empty)
          then begin
          S[j] = nexttoken;
          j = j+1;
          end;
          else begin
               while (nexttoken.precedence < S[j-1].precedence)
               O[i++] = S[j--];
               S[++j] = nexttoken;
               end;
```

```
end.
Procedure Emptystack(s)

Begin
     while (j ≦ 1);
     O[i++] = S[j--];
end.

Procedure Build_expression (O)

Begin
     t = 0;  l=0;        [we keep storing the tokens of the
                         output stream, O in locations
                         exp(l) till we get an operator. On
                         getting an operator, we store it
                         and link it with the preceeding two
                         locations - exp(l-1) and exp(l-2)
                         as respectively the right and left
                         link. We repeat the above process
                         till the output stream is
                         exhausted.]
    while (t ≦ k)
    begin
          while (O[t] ≠ "Operator")
          begin
          exp(l++) = O(t++);
```

```
        exp(l) -  left = NULL;
        exp(l) -  right = NULL;
        end

        exp(l) = O[t];
        exp(l) -  right = exp(l-1);
        exp(l) -  left  = exp(l-2);
        l = l-2;
    end.
```

Semantic check: The parser that we have developed here does only syntactical checking and builds the corresponding parsed tree if the input query is grammatically correct. Semantic checking is done by another routine which takes the output of Parser as the input and has an access to the global schema. It checks for the validity of fields and relations referred in the query and also looks into the problem of type matching.

Within the framework of distributed processing strategy that we discussed earlier the parser plays an important role. It is used at two levels. First at the top level where the given query is pased so that it could be properly partitioned. Secondly, once the partitioning of the query has taken place and subqueries have been distributed to local sites parser is used for translating

the query which is in SQL to the DML of the local DBMS. In this process parser is an essential intermediate step. The output of the parser is taken by the respective translating routines. These routines then generate the appropriate code in the local DML.

APPENDIX A

The following pages contain a listing of the program (in 'C') for parser for SQL.

```
/* IN THE MAIN ROUTINE WE GET THE INPUT QUERY INTO A FILE AND CALL THE
   ROUTINE PARSE() WHICH IS A TOP LEVEL ROUTINE AND IT SUBSEQUENTLY CALLS
   OTHER ROUTINES AS IT READS THE INPUT AND BUILDS THE CORRESPONDING
   STRUCTURE. THE STRUCTURES REFERRED TO HERE HAVE BEEN DEFINED IN THE
   FILE /DHAR$DEEP/EXTRA                                          */

#INCLUDE <STDIO.H>
#INCLUDE "/DHAR$DEEP/EXTRA";
 STRUCT SEL_S *B,*A;
 STRUCT FROM_S  *R,*V;
 STRUCT GPBY_S *M,*U;
 STRUCT ORDBY_S *E, *W;
 STRUCT TOKEN *WHERE1(),*GROUP1(),*HAVING1(),*ORDER1(),*INTO1();
 STRUCT QRY_S *QSR,*QQQ;

  MAIN()
     *  SMARKER = 0; TAG = 0;MARKER = 0; SEL = 0; CT = 0; UNI = 0;
               PRINTF("ENTER YOUR QUERY :\N");
               FP = FOPEN("BUF","W");
               DO
                * CF = GETCHAR();
                  FPUTC(CF,FP);
                * WHILE (CF != '\\');
               FCLOSE(FP);
               IF ((FP = FOPEN("BUF","R")) == NULL)
                  * PRINTF("BUF NOT OPENED \N"); EXIT(0);  *
               QSR = PARSE();
               RETURN(QSR);
            *
           STRUCT QRY_S  *PARSE() /* THIS IS THE MAIN ROUTINE */
           STRUCT QRY_S *Q;
           INT GRP,ORR,HAV,INO;
           GRP = 0;ORR = 0; HAV = 0; INO = 0;
           IF ((C->TKN) != 1)
           D = LEX1();
        IF ((D->TKN)==1 ** (C->TKN)==1)
        *   SEL = 1;
           /* IF THE FIRST ITEM IS SELECT WE START BUILDING THE
            STRUCTURE                                          */

            Q = (STRUCT QRY_S *) MALLOC(SIZEOF(STRUCT QRY_S));
            Q->S = NULL;Q->F = NULL;Q->W = NULL;Q->H = NULL;
            Q->G = NULL;  Q->O = NULL;Q->UNIQUE = 0;
            STRCPY(Q->INTO,"$");
            B = (STRUCT SEL_S *) MALLOC(SIZEOF(STRUCT SEL_S));
            Q->S = B;
            B->S_EXP = EXP(1);

            /* EXP(1) PARSES ARITHEMATIC EXPRESSIONS - WE EXPECT ONLY
               ARITHEMATIC EXP. IN SELECT CLAUSE.THE VARIOUS DIFFERENT
               ITEMS IN THE LIST ARE STORED IN A LINKED LIST */
            IF (UNI == 1) /* THIS IS FOR THE CASE WHEN UNIQUE IS
                             PRESENT                           */
           * Q->UNIQUE = C->TKN;
             B->S_EXP = EXP(1); UNI = 0;
```

```
*
 WHILE ((C->TKN) != 3)
 *     A = (STRUCT SEL_S *) MALLOC(SIZEOF(STRUCT SEL_S));
       B->SLINK = A;
       A->S_EXP = EXP(1);
       B = A;
 *
       B->SLINK = NULL;

  /* ONCE THE ITEM FROM HAS OCCURED WE STORE THE RELATION
     NAMES OCCURING IN A LINKED LIST                    */
  R = (STRUCT FROM_S *) MALLOC(SIZEOF(STRUCT FROM_S));
  Q->F = R;
  D = LEX1();
  IF ((D->TKN)==34)  STRCPY(R->REL_NM,D->NAME);
ELSE * PRINTF("RELATION NAME EXPECTED IN FROM CLAUSE\N");EXIT();*
  D = LEX1();
  IF ((D->TKN)==34)
  * STRCPY(R->LABEL,D->NAME);D = LEX1();*
  ELSE STRCPY(R->LABEL,"EMPTY");
  WHILE ((D->TKN) == 26)
*   D = LEX1();
    IF ((D->TKN) == 34)
    * V = (STRUCT FROM_S *) MALLOC(SIZEOF(STRUCT FROM_S));
      R->FLINK = V;
      STRCPY(V->REL_NM,D->NAME);
    *
    ELSE * PRINTF("REL_NAME EXPECTED IN FROM CLAUSE\N");EXIT();*
     D = LEX1();
     IF (D->TKN == 34)
     *STRCPY(V->LABEL,D->NAME); D = LEX1();*
      ELSE STRCPY(V->LABEL,"EMPTY");
     R = V ;
*
   R->FLINK = NULL;
     C = D ;

/* IF THE END OF THE QUERY  I.E. "\" HAS NOT OCCURED WE INVOKE
   THE APPROPRIATE ROUTINE DEPENDING ON THE NEXT ITEM IN THE
   QUERY.THIS PROCEDURE IS REPEATED AFTER  PARSING OF THE
   NEXT CLAUSE IN THE QUERY.FOR EACH CLAUSE THERE IS A MARKER
   WHICH ENSURES THAT THE SAME CLAUSE DOES NOT OCCUR TWICE (AT
   THE SAME LEVEL) IN THE QUERY.                          */

/* MARKER AND SMARKER HAVE BEEN USED FOR TAGGING A NESTED QUERY */


IF ((C->TKN)!= 27 && (MARKER < 1) && (C->TKN)!= 25)

  * SWITCH(C->TKN)*
                  CASE 4: WHERE1(Q);BREAK;
                  CASE 5: GRP = 1; GROUP1(Q);BREAK;
                  CASE 7: ORR = 1; ORDER1(Q);BREAK;
                  CASE 8: IF (GRP <= 0)
                      * PRINTF("HAVING NOT PREC.BY GROUP BY\N");EXIT();
```

```
                    ELSE * HAV = 1; HAVING1(Q);*BREAK;
                CASE 9: INO = 1; INTO1(Q);BREAK;
                DEFAULT: PRINTF("ERROR\N"); BREAK;
               *

      IF ((C->TKN)!=27 && (MARKER <1) && (C->TKN)!= 25)
      *    SWITCH(C->TKN)*
CASE 5: IF (GRP==1) PRINTF("ERR-2 GRP BY\N");ELSE *GRP=1; GROUP1(Q);*BREAK;
CASE 7: IF (ORR==1) PRINTF("ERR-2 ORD BY\N");ELSE *ORR=1; ORDER1(Q);*BREAK;
CASE 8: IF (HAV==1) PRINTF("ERR-2 HAVING\N");
        ELSE IF (GRP <= 0)
        * PRINTF("HAVING NOT PRECEEDED BY GROUP BY\N");EXIT(); *
        ELSE * HAV = 1;HAVING1(Q); * BREAK;
CASE 9: IF (INO==1) PRINTF("ERR-2 INTO\N");ELSE *INO=1;INTO1(Q);*BREAK;
                    DEFAULT:PRINTF("ERROR\N");EXIT();BREAK;
                       *

      IF ((C->TKN)!=27 && (MARKER < 1) && (C->TKN)!= 25)

      *    SWITCH(C->TKN)*
CASE 7: IF (ORR==1) PRINTF("ERR-2 ORD BY\N");ELSE *ORR=1;ORDER1(Q);*BREAK;
CASE 8: IF (HAV==1) PRINTF("ERR-2 HAVING\N");
        ELSE IF (GRP <= 0)
        * PRINTF("HAVING NOT PRECEEDED BY GROUP BY\N");EXIT();*
        ELSE * HAV = 1;HAVING1(Q);* BREAK;
                    DEFAULT:PRINTF("ERROR\N");EXIT();BREAK;
                       *
      IF ((C->TKN)!= 27 && (MARKER < 1)  && (C->TKN)!= 25)

      *    SWITCH(C->TKN)*
CASE 7:IF (ORR==1) PRINTF("ERR-2 ORD BY\N");ELSE *ORR=1;ORDER1(Q);*BREAK;
CASE 9:IF (INO==1) PRINTF("ERR-2INTO\N");ELSE *INO=1;INTO1(Q);*BREAK;
                    DEFAULT:PRINTF("ERROR\N");EXIT();BREAK;
                       *
      IF ((C->TKN)!=27  && (MARKER < 1)  && (C->TKN)!=25)

        * SWITCH(C->TKN)*
CASE 9:IF (INO==1) PRINTF("ERR-2 INTO\N");ELSE *INO=1;INTO1(Q);*BREAK;
                    DEFAULT:PRINTF("ERROR\N");EXIT();BREAK;
                       *

        IF ((C->TKN)!=27 && (MARKER < 1) && (C->TKN)!=25)
                    PRINTF("\\ IS MISSING\N");
  ELSE *IF ((C->TKN)== 27) MARKER = 2;ELSE IF ((C->TKN)==25) SMARKER = 2;*
      *
ELSE * IF ((C->TKN) == 27) MARKER = 2;ELSE IF ((C->TKN)== 25) SMARKER =2;*
      *
ELSE * IF ((C->TKN) == 27) MARKER = 2;ELSE IF ((C->TKN) == 25) SMARKER =2;
      *
ELSE * IF ((C->TKN) == 27) MARKER = 2;ELSE IF ((C->TKN) == 25) SMARKER= 2;
      *
ELSE * IF ((C->TKN) == 27) MARKER = 2;ELSE IF ((C->TKN) == 25) SMARKER=2;*
   *
ELSE * IF ((C->TKN) == 27) MARKER = 2;ELSE IF ((C->TKN) == 25) SMARKER =2;
```

```
ELSE    * PRINTF("SELECT IS MISSING\N");EXIT();     *
        RETURN(Q); /* IT RETURNS THE POINTER TO THR ROOT OF THE TREE */


          STRUCT TOKEN *WHERE1(Q)
          /* THIS ROUTINE PARSES THE  WHERE CLAUSE */
          STRUCT QRY_S *Q;
            *   Q->W = EXP(2);
                RETURN(C);
            *

          STRUCT TOKEN *HAVING1(Q)
         /* THIS ROUTINE PARSES THE HAVING CLAUSE */
          STRUCT QRY_S *Q;
           *
               Q->H = EXP(3);
                RETURN(C);
            *

         STRUCT TOKEN *INTO1(Q)
        /* THIS ROUTINE STORES THE FILE NAME  IN INTO CLAUSE IF THE
           FILE NAME IS A VALID ONE                                */
         STRUCT QRY_S *Q;
            * D = LEX1();
              IF ((D->TKN) == 34)
              STRCPY(Q->INTO,D->NAME);
              ELSE PRINTF("INVALID FILE NAME \N");
              D = LEX1();
              C = D;
              RETURN(C);
            *

          STRUCT TOKEN *GROUP1(Q)
          /* THIS ROUTINE STORES THE ITEMS IN GROUP BY CLAUSE IN A
             LINKED LIST                                          */
          STRUCT QRY_S *Q;
            * CHAR REE[13],ATT[13];
                D = LEX1();
                IF ((D->TKN) != 6)
        *    PRINTF("BY IS MISSING IN GROUP BY CLAUSE\N"); EXIT(); *
                ELSE
            *   D = LEX1();
                IF ((D->TKN) == 34   ** (D->TKN) ==35)
            *   M = (STRUCT GPBY_S *) MALLOC(SIZEOF(STRUCT GPBY_S));
                Q->G = M;
                IF ((D->TKN) == 35)
             *   STRCPY(REE,CONREL(D->NAME));
                 STRCPY(M->GP.REL,REE);
                 STRCPY(ATT,CONATR(D->NAME));
                 STRCPY(M->GP.ATR,ATT);
              *
              ELSE
             * STRCPY(M->GP.REL,"$");
               STRCPY(M->GP.ATR,D->NAME);
              *
```

```
    D = LEX1();
   WHILE ((D->TKN) == 34 ** (D->TKN) == 35 **(D->TKN) == 26)
*   IF ((D->TKN)!=26)
*   U = (STRUCT GPBY_S *) MALLOC(SIZEOF(STRUCT GPBY_S));
     M->GLINK = U;
   IF ((D->TKN) == 35)
* STRCPY(U->GP_REL,CONREL(D->NAME));
  STRCPY(U->GP_ATR,CONATR(D->NAME));
 *
 ELSE
 * STRCPY(U->GP_REL,"$");
    STRCPY(U->GP.ATR,D->NAME);
 *
    M = U;
*
    D = LEX1();
*
     M->GLINK = NULL;
*
*
     C = D;
     RETURN(C);
*


  STRUCT TOKEN *ORDER1(Q)
/* THIS ROUTINE STORES THE ITEMS IN ORDER BY CLAUSE IN A
   LINKED LIST.IT ALSO CHECKS AND STORES ANY EXPLICIT
   ORDERING (LIKE ASC OR DESC ) PRESENT.
  STRUCT QRY_S *Q;
*  D = LEX1();
   IF ((D->TKN)!=6)
*  PRINTF("BY IS MISSING IN ORDER BY CLAUSE\N");EXIT(); *
    ELSE
*  D = LEX1();
   IF ((D->TKN)==34 ** (D->TKN) ==35)
*  E = (STRUCT ORDBY_S *) MALLOC(SIZEOF(STRUCT ORDBY_S));
   Q->O = E;
   IF ((D->TKN) == 35)
* STRCPY(E->ORD_REL,CONREL(D->NAME));
  STRCPY(E->ORD_ATR,CONATR(D->NAME));
*
 ELSE
*  STRCPY(E->ORD_REL,"$");
   STRCPY(E->ORD_ATR,D->NAME);
*
   D = LEX1();
  IF ((D->TKN)== 39 ** (D->TKN)== 40)
    *     E->OKEY = D->TKN - 39; D = LEX1();    *
      ELSE
    E->OKEY = 0;
   WHILE ((D->TKN)==34 ** (D->TKN)==35 ** (D->TKN) == 26)
* IF ((D->TKN) != 26)
*  W = (STRUCT ORDBY_S *) MALLOC(SIZEOF(STRUCT ORDBY_S));
   E->OLINK = W;
```

```
              IF ((D->TKN) == 35)
            * STRCPY(W->ORD.REL,CONREL(D->NAME));
              STRCPY(W->ORD.ATR,CONATR(D->NAME));
            *
             ELSE
             * STRCPY(W->ORD.REL,"$");
               STRCPY(W->ORD.ATR,D->NAME);
             *
               D = LEX1();
IF ((D->TKN) == 27 ** (D->TKN) == 9) * W->OKEY = 39;E = W;BREAK; *
               IF ((D->TKN)== 39 ** (D->TKN)== 40)
               * W->OKEY = D->TKN - 39; *
               ELSE
               W->OKEY = 0;
               E = W;
             *
               D = LEX1();
             *
                E->OLINK = NULL;
             *
             *
               C = D;
               RETURN(C);
             *
```

```
 STRUCT TOKEN *C,*D;
 STRUCT QRY_S *PARSE();
 INT TAG,FLAG,MARKER,CT,SEL,UNI,SMARKER;

/* THE FOLLOWING ROUTINES CONREL AND CONATR EXTRACT THE RELATION AND
   ATTRIBUTE PART OF AN INPUT LIKE EMP.NAME                    */

 CHAR *CONREL(S)
 CHAR S[2*NAME_SIZE];
 * INT I,J;
   CHAR CREL[NAME_SIZE];
   I = 0; J = 0;
   WHILE (S[I] != '.')
    CREL[J++] = S[I++];
    CREL[J] ='\0';
    RETURN(CREL);
  *

 CHAR *CONATR(S)
 CHAR S[2*NAME_SIZE];
 * INT I,J;
   CHAR CATR[NAME_SIZE];
   I = 0; J= 0;
   WHILE (S[I] != '.') I++;
   ++I;
   WHILE (S[I])
   CATR[J++] = S[I++];
   CATR[J] = '\0';
   RETURN(CATR);
 *

/* THE FOLLOWING ROUTINE PARSES AN EXPRESSION      */

  STRUCT EXP_S *EXP(N)
  INT N;
*
      STRUCT TOKEN O[50],S[50];
      STRUCT EXP_S *Z,*X[50];
      STRUCT AGREN_S  *PTR;
      STRUCT QRY_S *PSR;
      STRUCT QRY_S *PPP;
      CHAR RE[NAME_SIZE],ATTNAME_SIZE];
   INT P[50];
   INT J,K,I,F,T,A,LP,RP,LOGOP,RELOP,SEC,LMARKER;

J=1;K=0;I=0;F=0;LP = 0; RP = 0; S[0].PREC = -1;RELOP = 0;LOGOP = 0;SEC=1;
 LMARKER = 0;

/* LEX1() ROUTINE IS IS CALLED TILL AN EXPRESSION DELIMITER IS ENCOUNTERED
   IF THERE IS SYNTACTICAL ERROR WE EXIT FROM THE ROUTINE */

WHILE ((C=LEX1())!= NULL  && (C->TKN)!=3 && (C->TKN)!=5 &&
 (C->TKN)!=7 && (C->TKN)!= 25 && (C->TKN)!= 27 && (C->TKN)!= 26 &&
 (C->TKN)!= 9 && (C->TKN)!= 8)
```

```
/* N = 1 CORRESPONDS TO THE EXPRESSIONS IN SELECT CLAUSE */

* IF (N == 1)
  IF (((C->TKN)>=28 && (C->TKN)<=33) ** ((C->TKN)>=16 && (C->TKN)<=18))
  PRINTF("ONLY ALG.EXP. ALLOWED IN SELECT CLAUSE AND IN FUN. ARG"\N");

/* N = 2 CORRESPONDS TO THE EXPRESSIONS IN WHERE CLAUSE  */

  IF (N == 2)
  IF ((C->TKN) >= 10 && (C->TKN) <= 14)
  PRINTF("AGG.FUN. NOT ALLOWED IN WHERE CL USE\N");

/* THE TOKENS ARE STORED IN AN ARRAY AS THEY ARE GENERATED     */

  P[++I] = (C->TKN);

/* NOW WE LOOK FOR TOKENS WITH WHICH AN EXPRESSION CAN BEGIN (F=0) AND
  TAKE APPROPRIATE ACTIONS IS THE STACK WHICH IS USED FOR CONVERTING
  INFIX TO POSTFIX FORM.O IS AN ARRAY WHICH FINALLY CONTAINS THE EXP.
  IN POSTFIX FORM                                              */

  IF (F<=0)
*IF (C->TKN >=34 && C->TKN <=38)*O[K++]= *C;PRINTF("%S",O[--K].NAME);K++;*
 ELSE IF (C->TKN ==18) *S[J++]= *C;*
 ELSE IF (C->TKN==23) *  S[J++] = *C;IF (T G >0 ) ++LP; *
 ELSE IF (C->TKN==20)*STRCPY(S[J].NAME,"(");S[J].TKN=20;S[J].PRE=9:J++;*
*ELSE IF ((C->TKN) >= 10  && (C->TKN) <= 14)
 * O[K++]= *C; PRINTF("%S\N",O[--K].NAME);++K;++TAG;
   PTR = (STRUCT AGRFN_S *) MALLOC(SIZEOF(STRUCT AGRFN_S));
   PTR->FN-KEY = AGT[C->TKN];
   IF ((C->TKN) == 13)  CT = 1;
   PTR->ARG = EXP(1);
 *
 ELSE IF ((C->TKN)== 21 && SEL==1) * O[K++] = *C;*
 ELSE IF((C->TKN)==2) * UNI = 1;  BREAK; *
 ELSE *PRINTF(" ERROR\N"); EXIT();  *
 F=1;
*
/* NOW DEPENDING ON THE PREVIOUS TOKEN WE TAKE APPROPRIATE ACTION AS
 THE NEXT TOKEN IS RECEIVED FROM THE LEXICAL ANALYSER.IN CASE OF A
SYNTAX ERROR WE EXIT FROM THE ROUTINE          */

 ELSE
 IF ((P[--I] >=34 && P[I] <=38) ** (P[I]>=10 && P[I]<=14) ** P[I]==24)
 * IF ((C->TKN)>=28 && (C->TKN)<=33) ++RELOP;
   IF ((C->TKN) == 16 ** (C->TKN) == 17) ++LOGOP;
 SWITCH(C->TKN)*
   CASE 24: --J;WHILE (STRCMP(S[J].NAME,"("))
 * O[K++] = S[J--];PRINTF("%S",O[--K].NAME);K++;* IF (TAG >0) ++P;BREAK;
   CASE 25: EXIT();BREAK;
   CASE 18:;
   CASE 20:;
   CASE 21:;
   CASE 22:;
   CASE 29:;
```

```
  CASE 30:;
  CASE 31:;
  CASE 33:;
  CASE 32:;
  CASE 28:;
  CASE 19:;
  CASE 17:;
  CASE 16:
          IF ((C->PREC) > (S[--J].PREC))
          * S[++J] = *C;J++;*
          ELSE *WHILE (S[J].PREC > C->PREC) *O[K++]=S[J--];
          PRINTF("%S",O[--K].NAME);K++;* S[++J]= *C; J++;*BREAK;
  DEFAULT: PRINTF("ERROR\N");EXIT();BREAK;
 *
 I++;
*
ELSE
IF (P[I]==23)
 *SWITCH(C->TKN)*
   CASE 24:IF (TAG==2) *--J;WHILE (STRCMP(S[J].NAME,"(")
          *O[K++] = S[J--];PRINTF("%S",O[--K].NAME);K++;*
          *
           ELSE PRINTF("ERROR\N");EXIT();BREAK;
   CASE 21:IF(CT==1) S[J++] = *C;ELSE PRINTF("ERROR\N");BREAK;
   CASE 10 :;
   CASE 11 :;
   CASE 12 :;
   CASE 13 :;
   CASE 14 : ++TAG;IF(TAG <= 2)
             * O[K++] = *C;PRINTF("%S\N",O[--K].NAME);K++;
               PTR =  (STRUCT AGRFN_S *) MALLOC(SIZEOF(STRUCT AGRFN_S));
             PTR->FN_KEY = AGT[C->TKN];
             PTR->ARG = EXP(1);
             * BREAK;
   CASE 23 : S[J++]= *C;IF (TAG >0) ++IB; BREAK;
   CASE 34 :;
   CASE 35 :;
   CASE 37 :;
   CASE 38 :;
   CASE 36 :; O[K++] = *C;PRINTF("%S",O[--K].NAME);K++;BREAK;
   CASE 20: *STRCPY(S[J].NAME,"-");S[J].TYPE=20;S[J].PREC=9;J++;
   CASE 18 :IF ((C->PREC) > (S[--J].PREC))
            *S[++J]= *C;J++;*
            ELSE *O[K++]=S[J];S[J]= *C;I++;*   BREAK;
   DEFAULT  : PRINTF("ERROR\N"); EXIT();BREAK;
   *
   I++;
 *
 ELSE IF (P[I]==16 **P[I]==17)
 *SWITCH(C->TKN)*
   CASE 10:;
   CASE 11:;
   CASE 12:;
   CASE 13:;
   CASE 14: ++TAG;IF (TAG <= 2)
```

```
            * O[K++] = *C;
            PTR = (STRUCT AGRFN_S *) MALLOC(SIZEOF(STRUCT AGRFN_S));
            PTR->FN_KEY = C->TKN;
            PTR->ARG = EXP(1); * BREAK;
   CASE 34 :;
   CASE 35 :;
   CASE 36:;
   CASE 38:;
   CASE 37: O[K++]= *C;PRINTF("%S",O[--K].NAME);K++;BREAK;
   CASE 23: S[J++] = *C;BREAK;
   CASE 18 :IF ((C->PREC) > (S[--J].PREC))
            * S[++J]= *C; J++;*
    ELSE *O[K++]=S[J];S[J]= *C;PRINTF("%S",O[--K].NAME);J++;K++;*BREAK;
   DEFAULT : PRINTF("ERROR\N");EXIT();BREAK;
   *
   I++;
 *
 ELSE IF ( P[I] >= 28 && P[I] <= 33)
 * SWITCH(C->TKN) *
   CASE 10:;
   CASE 11:;
   CASE 12:;
   CASE 13:;
   CASE 14: ++TAG; IF (TAG <= 2)
               * O[K++] = *C;
                 PTR = (STRUCT AGRFN_S *) MALLOC(SIZEOF(STRUCT AGRFN_S));
                 PTR->FN_KEY = C->TKN;
                 PTR->ARG = EXP(1);*  BREAK;
   CASE 34:;
   CASE 35:;
   CASE 36:;
   CASE 38:;
   CASE 37: O[K++] = *C;PRINTF("%S",O[--K].NAME);K++;BREAK;
   CASE 23: S[J++] = *C;BREAK;
   CASE 20: STRCPY(S[J].NAME,"-");S[J].PREC = 9;S[J].TKN =20;J++;BREAK;
   CASE 1:O[K++]= *C;PRINTF("%S",O[--K].NAME);K++;
          PSR = PARSE(); /* CASE OF NESTED QUERY  */
          BREAK;
   DEFAULT : PRINTF("ERROR\N");EXIT(); BREAK;
   *
   I++;
 *
ELSE IF (P[I]== 1)
* SWITCH(C->TKN)*
   CASE 16:;
   CASE 17: O[K++] = S[--J];S[J++] = *C;PRINTF("%S",O[--K].NAME);K++;
            ++LOGOP;BREAK;
   DEFAULT: PRINTF("ERROR\N"); BREAK;
   *
 I++;
 *
 ELSE IF (P[I] == 21 && CT == 1)
  * SWITCH(C->TKN)*
     CASE 24: --J;WHILE (STRCMP(S[J].NAME,"(")
            * O[K++] = S[J--];PRINTF("S\N",O[--K].NAME); K++;* --RP;BREA
```

```
      DEFAULT :PRINTF("ERROR\N"); EXIT();BREAK;
    *
   I++;
   *
  ELSE
  *SWITCH(C->TKN)*
    CASE 10:;
    CASE 11:;
    CASE 12:;
    CASE 13:;
    CASE 14: ++TAG; IF (TAG <= 2)
              * O[K++] = *C;
             PTR = (STRUCT AGRFN_S *) MALLOC (SIZEOF(STRUCT AGRFN_S));
             PTR->FN_KEY = C->TKN;
             PTR->ARG = EXP(1); * BREAK;
    CASE 34 :;
    CASE 35:;
    CASE 36:;
    CASE 38:;
    CASE 37: O[K++]= *C;PRINTF("%S",OF--K].NAME);K++;BREAK;
    CASE 23 : S[J++] = *C; IF (TAG >0 ) ++LP;BREAK;
    CASE 20 :STRCPY(S[J].NAME,"-");S[J].PREC=0;S[J].TKN=20;J++;BREAK;
              DEFAULT  : PRINTF("ERROR\N") EXIT();BREAK;
    *
  I++;
  *

   IF (LP>0 &&  RP>0)
      IF (LP == RP) * --TAG;BREAK;*

   IF (MARKER > 1) BREAK;
   ELSE IF (SMARKER >1)
   * PPP = PSR;LMARKER = SMARKER;SMARKER = 0;
   *
*

  IF (( P[I]>=16 && P[I]<=22) ** ( P[I]>=28 && P[I]<= 33))
  IF (P[I]==21 && (CT==1  ** SEL == 1)) *P INTF("\N");*
  ELSE PRINTF("INVALID EXPRESSION\N");

   CT = 0; SEL = 0;

 IF ((N==2) ** (N==3))
 IF (RELOP <= 0)PRINTF("ERR-NO REL.OPERATO  IN HAVING OR WHERE CLAUSE\N");;

  WHILE (J > 1) * O[K] =S[--J];PRINTF("%c",O[K].NAME);K++;*
  O[K].TKN = -2;
 T = 0; A = 0 ;

/* THE FOLLOWING PART BUILDS THE STRUCTURE FROM THE EXP. IN POSTFIX
FORM I.E. IT READS FROM THE ARRAY O[] AND  UILDS THE TREE */


 WHILE (T < K)
 * WHILE ((O[T].TKN) > 33 ** ((O[T].TKN) >= 10 &&  (O[T].TKN) <=14) **
```

```
(O[T].TKN) ==1)
*X[A] = (STRUCT EXP_S *) MALLOC(SIZEOF(STRUCT EXP_S));
 IF ((O[T].TKN)==1)          /* NESTED QUERY      */
 *  X[A]->KEY = 2;
   IF (LMARKER > 1)
   * IF (SEC <= 1)
   * X[A]->DATA.QLINK = PPP; SEC = 2;
   *
   ELSE
   X[A]->DATA.QLINK = PSR;
   *
   ELSE
   X[A]->DATA.QLINK = PSR;
   *
   ELSE IF ((O[T].TKN)>=10 && (O[T].TKN)<=14)
   * X[A]->KEY = 5;
     X[A]->DATA.AGRGN = *PTR;
   *
   ELSE IF((O[T].TKN)== 34 ** (O[T].TKN)== 35)
   * X[A]->KEY = 4;
     IF ((O[T].TKN)== 35)
     * STRCPY(RE,CONREL(O[T].NAME));
    STRCPY(X[A]->DATA.FIELD.REL,RE);
    STRCPY(AT,CONATR(O[T].NAME));
    STRCPY(X[A]->DATA.FIELD.ATR,AT);
   *
   ELSE
   * STRCPY(X[A]->DATA.FIELD.REL,"$");
     STRCPY(X[A]->DATA.FIELD.ATR,O[T].NAME);
   *

 *
   ELSE IF((O[T].TKN)>= 36 && (O[T].TKN)<=38)
   * X[A]->KEY = 3;
     X[A]->DATA.CONST.KEY = O[T].TKN - 35;
     IF ((O[T].TKN) == 36)
     X[A]->DATA.CONST.VAR.I = ATOI(O[T].NAME);
     IF ((O[T].TKN) == 38)
     STRCPY(X[A]->DATA.CONST.VAR.S,O[T].NAME);
   *
   X[A]->LEFT = NULL;
   X[A]->RIGHT = NULL;
     A++;
     T++;
 *
  IF ((O[T].PREC >= 9) ** (O[T].TKN == 18)) /* UNARY MINUS OR A NOT */
  *    Z = (STRUCT EXP_S *) MALLOC(SIZEOF(STRUCT EXP_S));
       Z->KEY = 1;
       Z->DATA.KEY_OP = MPT[O[T].TKN];
       Z->RIGHT = X[--A];
       Z->LEFT = NULL;
       T++;
  X[A] = Z; A++;
   *
   ELSE IF ((K==1) && (O[T].TKN) == 21)
   * Z = (STRUCT EXP_S *) MALLOC(SIZEOF(STRUCT EXP_S));
```

```
    Z->KEY = 4;
    STRCPY(Z->DATA_FIELD.ATR,OPT[T].NAME);
    Z->RIGHT = NULL;
    Z->LEFT = NULL;
    T++;
 *
 ELSE IF (K==1)
 * Z = X[--A]; T++;
 *
 ELSE
 *  Z = (STRUCT EXP_S *) MALLOC(SIZEOF(STRUCT EXP_S));
     Z->KEY = 1;
     Z->DATA_KEY_OP = OPT[OPT].TKN;
     Z->RIGHT = X[--A];
     Z->LEFT = X[--A];
     T++;
     X[A] = Z;    A++;
  *
*

 IF (LOGOP > 0)
* ++LOGOP; IF (REL_OP != LOGOP) PRINTF("INVALID BOOLEAN EXP.\N"); *

RETURN(Z);  /* Z IS THE ROOT OF THE TREE   */
```

```
/* THIS PART OF THE PROGRAM IMPLEMENTS THE LEXICAL ANALYSER.
   THE NAME OF THE FILE CONTAINING THIS PART IS /DHARSDEEP/FEYNMAN.
   BEFORE DEFINING THE ROUTINE LEX1() WHICH GENERATES TOKENS
   NECESSARY INITIALISATION OF KEYWORD-TABLE,SYMBOL-TABLE,
   AND TRANSITION TABLE HAVE BEEN DONE */


#DEFINE NAME_SIZE 13
#DEFINE TAB_SIZE 100
STRUCT TOKEN *                          /* TOKEN IS A STRUCTURE WITH FIELDS:
             CHAR NAME[2*NAME_SIZE];     NAME:STORES THE CONTENT OF THE TOKEN
             INT TKN;                    TKN: STORES THE TOKE NUMBER
             INT PREC;                  PREC:STORES  PRECEDENCE NO.OF TOKEN */
             *IDTAB[TAB_SIZE],CONSTAB[TAB_SIZE];

 /* IDTAB,CONSTAB ARE IDENTIFIER AND CONSTANT TABLE RESPECTIVELY */


/* WE NOW FORM THE TRANSITION TABLE.THE ROWS REPRESENT STATES WHILE
   THE COLOUMNS REPRESENT THE INPUT CHARACTERS AFTER BEING MAPPED TO
   INTEGERS  */


              INT TRANSTAB[13][19] = *
          *19,20,21,22,23,24,25,26,27,1,2,28,3,4,5,6,-1,-2,-1*,
         * -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-2 -1*,
         *9,9,9,9,9,9,9,9,9,9,-2,9,9,9,9,9,9,-2,9*,
         *32,32,32,32,32,32,32,32,32,32,32,30,-1,31,32,32,32,-2,32,*,
         *33,33,33,33,33,33,33,33,33,10,33,29,33,33,33,33,33,-2,33*,
         *34,34,34,34,34,34,34,34,34,8,34,34,34,34,5,5,5,-2,34*,
         *36,36,36,36,36,36,36,36,36,11,36,36,36,36,36,6,36,-2,36*,
         *37,37,37,37,37,37,37,37,37,37,37,37,37,37,37,7,37,-2,37*,
         *-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,10,10,-2,-1*,
         *9,9,9,9,9,9,9,9,9,9,38,9,9,9,9,9,9,-2,9*,
         *35,35,35,35,35,35,35,35,35,35,35,35,35,35,10,10,10,-2,35*,
         *-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,-1,-2,-1*,
         *37,37,37,37,37,37,37,37,37,37,37,37,37,37,37,12,37,-2,37,**;
     INT L = 0; Y = 0;
FILE *FP,*FOPEN();/* FP IS POINTER TO THE FILE CONTAINING THE INPUT
                     QUERY */
STRUCT TOKEN KEYTAB[20] = *"SELECT",1,0,
                          "UNIQUE",2,0,
                          "FROM",3,0,
                          "WHERE",4,0,    /* THIS IS THE
                          "GROUP",5,0,
                          "BY",6,0,           INSTALLATION
                          "ORDER",7,0,
                          "HAVING",8,0,           OF
                          "INTO",9,0,
                          "MAX",10,0,          KEYWORD-TABLE
                          "MIN",11,0,                        */
                          "AVG",12,0,
                          "COUNT",13,0,
                          "SUM",14,0,
                          "IN",15,0,
```

```
                        "AND",16,1,
                        "OR",17,1,
                        "NOT",18,2,
                        "ASC",39,0,
                        "DESC",40,0
                       *;
STRUCT TOKEN SYMTABL15] = *"+",19,4,
                           "-",20,4,
                           "*",21,5,    /* THIS IS THE
                           "/",22,5,
                           "(",23,0,       INSTALLATION
                           ")",24,0,
                           ":",25,0,          OF
                           ",",26,0,
                           "\",27,0,       SYMBOL-TABLE
                           "=",28,3,
                           ">=",29,3,                    */
                           "<=",30,3,
                           "<>",31,3,
                           "<",32,3,
                           ">",33,3,
                          *;
/* THE FOLLOWING ROUTINE LEX1() READS THE INPUT CHARACTER BY CHARACTER
   AND RETURNS THE ADDRESS OF THE APPROPRIATE TOKEN   */
  STRUCT TOKEN *LEX1()
*
  INT STATE,X,K,J,T,I,P,Q,B,E,C;
  CHAR STR2*NAME_SIZE];
                        T = 19;
                   WHILE ((C = FGETC(FP)) == 32 ** C == '\N');
               /* SKIPPING BLANKS AND NEWLINE CHARACTER */
                        UNGETC(C,FP);
                        STATE = 0;  /* INITIAL STATE = 0 */
                        B = 0;

              /* FINAL STATES ARE GREATER THAN 18 */
                   WHILE (STATE >= 0 && STATE < 19)
                    *       C =FGETC(FP);
                            STR++T = C ;
              /* S STORES THE INPUT STRING */

             /* WE NOW MAP THE INPUT CHARACTERS TO INTEGER VALUES */

                            IF (ISALPHA(C))
                             I = 14;
                            ELSE IF (ISDIGIT(C))
                             I = 15;
                            ELSE
                            * SWITCH(C)  *
                                 CASE '+' : I=0;BREAK;
                                 CASE '-' : I =1;BREAK;
                                 CASE '*' : I=2;BREAK;
                                 CASE '/' : I=3;BREAK;
                                 CASE '(' : I=4;BREAK;
                                 CASE ')' : I =5;BREAK;
```

```
                        CASE ';' : I=6;BREAK;
                        CASE ',' : I=7;BREAK;
                        CASE '\\' : I=8;BREAK;
                        CASE '\.' : I=9;BREAK;
                        CASE '"' : I=10 ;BREAK;
                        CASE '=' : I =11 ;BREAK;
                        CASE '<' : I=12 ;BREAK;
                        CASE '>' : I=13 ;BREAK;
                        CASE '_' : I=16 ;BREAK;
                        CASE EOF : I=17 ;BREAK;
                        DEFAULT : I =18;BREAK;
                        *
          *
        STATE = TRANSTAB[STATE][I];
      /* THIS STATEMENT DEFINES THE TRANSITION FROM ONE STATE TO
        THE NEXT */
       *
       IF (STATE >= 32  && STATE < 38)
           UNGETC(C,FP);
       IF (STATE >= 19 && STATE <= 33)
           RETURN(&SYMTAB[STATE-19]);
        /* THIS RETURNS THE ADDRESS OF THE APPROPRIATE SYMBOL-TABLE
           ENTRY */
       IF  (STATE == 38)
           S[B] = '\0';
       ELSE S[--B] = '\0';

      /* A CANDIDATE FOR AN IDENTIFIER IS FIRST CHECKED TO SEE
        IF IT IS A KEYWORD .IF YES THEN CORRESPONDING KEYWORD-
        TABLE ENTRY'S ADDRESS IS RETURNED,ELSE IT IS COMPARED
        TO SEE WHETHER IT HAS BEEN ALREADY INSTALLED IN IDTAB.
        IF YES, THEN CORRESPONDIG ADDRESS IS RETURNED ELSE THE
        INPUT STRING IS INSALLED IN IDTAB AND THE ADDRESS
        RETURNED.                                              */

       IF (STATE == 34 ** STATE == 35)
*       P = 0;
        WHILE ((STRCMP(S,KEYTAB[P].NAME)) !=0 && P <= T)
        P++;
        IF (P <= T)
        RETURN(&KEYTAB[P]);
        ELSE
      * IF (L<= 0)
      * STRCPY(IDTAB[L].NAME,S);
        IDTAB[L].TKN = STATE;
        IDTAB[L].PREC = 0;
        RETURN(&IDTAB[L++]);
      *
        ELSE
      * Q = 0;  --I;
        WHILE (Q <= I)
       * IF (STRCMP(S,IDTAB[Q].NAME)) Q++;
         ELSE  BREAK;
       *                                          );
        IF (Q <= L)
```

```
    *    ++L;RETURN(&IDTAB[Q]);     *
         ELSE
       * STRCPY(IDTAB[++L].NAME,S);
         IDTAB[L].TKN = STATE;
         IDTAB[L].PREC = 0;
         RETURN(&IDTAB[L++]);
       *
       *
       *
*
 /* A CONSTANT IS FIRST CHECKED TO SEE IF IT HAS BEEN ALREDY
    INSTALLED IN CONSTAB. IF YES,THE CORRESPONDING TABLE-ENTRY'S
    ADDRESS IS RETURNED ELSE IT IS INSTALLED AND THE ADDRESS
    RETURNED                                              */
       IF (STATE >=36 && STATE <= 38)
       IF (Y <= 0)
       * STRCPY(CONSTAB[Y].NAME,S);
         CONSTAB[Y].TKN = STATE;
         CONSTAB[Y].PREC = 0;
         RETURN(&CONSTAB[Y++]);
       *
         ELSE
       * X = 0; --Y;
         WHILE ( X <= Y)
         * IF (STRCMP(S,CONSTAB[X].NAME)) X++;
           ELSE BREAK;
         *
         IF (X <= Y)
         *  ++Y;  RETURN(&CONSTAB[X]);  *
         ELSE
       * STRCPY(CONSTAB[++Y].NAME,S);
         CONSTAB[Y].TKN = STATE;
         CONSTAB[Y].PREC = 0;
         RETURN(&CONSTAB[Y++]);
       *
       *
*
       IF ( STATE == -1)
       * PRINTF("INVALID STRING\N"); RETURN(NULL); *
      /* THE INPUT STRING IS NOT ALLOWED BY THE LANGUAGE */
```

```
/* THIS PART OF THE PROGRAM BUILDS THE STUCTURE FOR EXPRESSIONS.
   THE NAME OF THE FILE CONTAINING THIS IS /DHAR$DEEP/EXTRA.
   THIS FILE ALSO CONTAINS THE DECLARATION OF VARIOUS STRUCTURES
   USED                */


#INCLUDE "/DHAR$DEEP/FEYNMAN"
#DEFINE NAME_SIZE 13
#DEFINE FN_LEN 6
#DEFINE STR_LEN 256
UNION CONST_U
            * INT I;
              FLOAT R;
              CHAR S[STR_LEN];
            *;
/* THIS STORES THE CONSTANT IN APPROPRIATE FIELD  */


STRUCT CONST_S
            * INT KEY;
              UNION CONST_U VAR;
            *;
/* KEY IS CODE FOR CONSTANTS.IT IS 1 FOR INTEGERS,2 FOR REAL AND 3 FOR
   STRING CONSTANTS                                          */

STRUCT REL_S
            * CHAR REL[NAME_SIZE];
              CHAR ATR[NAME_SIZE];
            *;
/* THIS IS STORES THE RELATION AND ATTRIBUTE NAME   */

STRUCT AGRFN_S
            * INT FN_KEY;
              STRUCT EXP_S *ARG;
            *;
/* THIS STORES THE AGGREGATE FUNCTION.
   FN_KEY IS THE CODE FOR  AGGREGATE FUNCTION.*/

 UNION DATA_U
            * INT KEY_OP;
              STRUCT QRY_S *QLINK;
              STRUCT CONST_S CONST;
              STRUCT REL_S FIELD;
              STRUCT AGRFN_S  AGRFN;
            *;
/* DATA_U IS THE DATA CONTAINED IN THE NOD. OF THE TREE.IT IS ONE OF
   THE FOLLOWING:AN OPERATOR OR A POINTER TO A QUERRY (FOR NESTED QUERYS)
   OR A CONSTANT OR A FIELD NAME  OR AN AG R.FUNCTION */

 STRUCT EXP_S
            *  INT KEY;
               UNION DATA_U DATA;
               STRUCT EXP_S *LEFT;
               STRUCT EXP_S *RIGHT;
            * ;
```

```
/* IT STORES AN EXPRESSION.KEY IS AN INEGER WHICH INDICATES WHAT IS
   STORED .DATA IS THE ACTUAL DATA STORED.LEFT AND RIGHT ARE RESPECTIVELY
   LEFT AND RIGHT LINKS */
   STRUCT SEL_S
          *  STRUCT EXP_S  *S_EXP;
             STRUCT SEL_S  *SLINK;
          *;
/* IT STORES THE CONTENTS OF THE SELECT CLAUSE IN THE FORM OF A LIST.
   THE LIST CONSISTS OF EXPRESSIONS */

 STRUCT FROM_S
          *  CHAR REL_NM[NAME_SIZE];
             CHAR LABEL[10];
             STRUCT FROM_S *FLINK;
          *;
/* IT STORES THE CONTENTS OF THE FROM CLAUSE IN THE FORM OF A LIST.
   THE LIST CONSISTS OF RELATION-NAMES .THERE MAY BE A LABEL FOR A
   RELATION (FOR SELF-JOIN)  */

  STRUCT GPBY_S
          *  STRUCT REL_S  GP;
             STRUCT GPBY_S *GLINK;
          *;
/* IT STORES THE CONTENTS OF GROUP BY CLAUSE IN THE FORM OF A LIST.
   THE LIST CONSISTS OF FIELD NAMES. */

 STRUCT ORDBY_S
          * STRUCT REL_S  ORD;
            INT     OKEY;
            STRUCT ORDBY_S  *OLINK;
          *;
/* IT STORES THE CONTENTS OF ORDER BY CLAUSE IN THE FORM OF A LIST.
  THE LIST CONSISTS OF FIELD NAMES.OKEY STORES THE ASC. OR DESC.
  QUALILIFICATION                                                 */

  STRUCT QRY_S
         *  STRUCT SEL_S *S;
            INT    UNIQUE;
            STRUCT FROM_S *F;
            STRUCT EXP_S *W;
            STRUCT GPBY_S *G;
            STRUCT EXP_S  *H;
            STRUCT  ORDBY_S *O;
            CHAR INTO[NAME_SIZE];
         *;
/* QRY_S STORES THE POINTR TO ALL THE CLAUSES EXCEPT UNIQUE AND INTO.
   IF UNIQUE IS PRESENT ITS FLAG IS SET.INTO STORES THE FILE NAME IN
   IN THE INTO CLAUSE                                              */

        INT MPT[34]= *0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,1,14,15,12,11,0,
                      0,0,0,0,4,7,9,5,8,6*;
/* IT MAPS THE OPERATORS TO THEIR RESPECTIVE CODE NOS.  */

        INT AGT[15] = *0,0,0,0,0,0,0,0,0,0,5,4,1,2,3*;
/* IT MAPS THE AGG.FUNCTIONS TO THEIR RESPECTIVE CODE NOS.  */
```

APPENDIX B

The following pages contain illustrations which show the output of parser. We traverse the parsed tree in the order of occurance of the clauses in the query. The tree for expressions is traversed in the preorder.

```
SELECT NAME,JOB,SALARY,DEPTNO
FROM EMP
WHERE (DEPTNO = 10 AND SALARY <= 1200) \
TRAVERSING THE TREE
SELECT CLAUSE
ATTR_NAME  NAME
ATTR_NAME  JOB
ATTR_NAME  DEPTNO
FROM CLAUSE
 RELATION EMP
WHERE CLAUSE
OPERATOR CODE = 2
OPERATOR CODE = 4
ATTR_NAME  DEPTNO
CONSTANT 10
OPERATOR CODE = 9
ATTR_NAME  SALARY
CONSTANT 1200
```

NATIONAL INFORMATICS CENTRE

NATIONAL INFORMATICS CENTRE

```
SELECT NUMBER,NAME,JOB
FROM EMP
ORDER BY NUMBER \
TRAVERSING THE TREE
SELECT CLAUSE
ATTR.NAME  NUMBER
ATTR.NAME  NAME
ATTR.NAME  JOB
FROM CLAUSE
 RELATION EMP
ORDER BY CLAUSE
ATR  NUMBER
```

```
SELECT DEPTNO,AVG(SALARY)
FROM EMP
GROUP BY DEPTNO
HAVING AVG(SALARY) > 2000 ;

TRAVERSING THE TREE
SELECT CLAUSE
ATTR.NAME  DEPTNO
AGGR.FN KEY = 1
ATTR.NAME  SALARY
FROM CLAUSE
 RELATION EMP
GROUP BY CLAUSE
ATR  DEPTNO
HAVING CLAUSE
OPERATOR CODE = 6
AGGR.FN KEY = 1
ATTR.NAME  SALARY
CONSTANT 2000
```

```
SELECT DEPTNO,AVG(SALARY)
FROM EMP
GROUP BY DEPTNO
HAVING AVG(SALARY) <
        SELECT AVG(SALARY)
        FROM  EMP \

TRAVERSING THE TREE
SELECT CLAUSE
ATTR.NAME  DEPTNO
AGGR.FN KEY = 1
ATTR.NAME  SALARY
FROM CLAUSE
 RELATION EMP
GROUP BY CLAUSE
ATR  DEPTNO
HAVING CLAUSE
OPERATOR CODE = 8
AGGR.FN KEY = 1
ATTR.NAME  SALARY
NESTING
SELECT CLAUSE
AGGR.FN KEY = 1
ATTR.NAME  SALARY
FROM CLAUSE
 RELATION EMP
```

```
SELECT NAME,SALARY
FROM EMP
WHERE DEPTNO =
        SELECT DEPTNO
        FROM . EMP
        GROUP BY DEPTNO
        HAVING AVG(SALARY) =
                SELECT MAX(AVG(SALARY))
                FROM EMP
                GROUP BY DEPTNO )

TRAVERSING THE TREE
SELECT CLAUSE
ATTR.NAME   NAME
ATTR.NAME   SALARY
FROM CLAUSE
RELATION EMP
WHERE CLAUSE
OPERATOR CODE = 4
ATTR.NAME   DEPTNO
NESTING
SELECT CLAUSE
ATTR.NAME   DEPTNO
FROM CLAUSE
RELATION EMP
GROUP BY CLAUSE
ATR   DEPTNO
HAVING CLAUSE
OPERATOR CODE = 4
AGGR.FN KEY = 1
ATTR.NAME   SALARY
NESTING
SELECT CLAUSE
AGGR.FN KEY = 5
AGGR.FN KEY = 1
ATTR.NAME   SALARY
FROM CLAUSE
RELATION EMP
GROUP BY CLAUSE
ATR   DEP
```

```
SELECT NAME
FROM EMP
WHERE SALARY + COMM.=
        SELECT MAX(SAL + COMM.)
        FROM  EMP
        WHERE JOB = 'SALES';
    OR ( JOB = 'SALES' AND DEPTNO = 30 ) \

TRAVERSING THE TREE
SELECT CLAUSE
ATTR.NAME  NAME
FROM CLAUSE
RELATION EMP
WHERE CLAUSE
OPERATOR CODE = 3
OPERATOR CODE = 4
OPERATOR CODE = 14
ATTR.NAME  SALARY
ATTR.NAME  COMM
NESTING
SELECT CLAUSE
AGGR.FN KEY = 5
OPERATOR CODE = 14
ATTR.NAME  SALARY
ATTR.NAME  COMM
FROM CLAUSE
RELATION EMP
WHERE CLAUSE
OPERATOR CODE = 4
ATTR.NAME  JOB
CONSTANT "SALES"
OPERATOR CODE = 2
OPERATOR CODE = 4
ATTR.NAME  JOB
CONSTANT "SALES"
OPERATOR CODE = 4
ATTR.NAME  DEPTNO
CONSTANT 30
```

APPENDIX C

LIST OF KEYWORDS IN SQL

SELECT, UNIQUE, FROM, WHERE, GROUP BY, HAVING, ORDER BY, INTO, AND, OR, NOT, ASC, DESC, MAX, MIN, AVG, SUM, COUNT.

LIST OF SYMBOLS

(,), +, -, *, /, ., ;, ",<,>,\

CONSTANTS

We have Integer, Real and String constants.

The precedence number associated with various tokens are as follows :

)	0
(	0
+	4
-	4
*	5
/	5
AND	1
OR	1

NOT 2

RELATIONAL OPERATORS 3

All other tokens have precedence number zero.

When a unary minus is detected, we increase the precedence of (-) to 9.

REFERENCES

1. S. Ceri and G.Pelagatti, Distributed Database Principles and Systems, McGrawHill Book Company 1984.

2. Wong. "Retrieving Dispersed Data from SDD1", Proc. o the second Berkley workshop on Distributed Dat Management and Computer Networks.

3. Hevner and Yao, "Query Processing in Distributed Dat Base Systems", IEEE Trasactions on Softwar Engineering.

4. Stonbroker et. al., "A Distributed Version of INGRESS", Berkley workshop on Distributed Data Management and Computer Networks, 1977.

5. Roth et. al., "Anoveriew of the Preliminary Design of SDD-1-A system for Distributed Database", Berkley workshop on distributed Data Management and computer network, Larence Berkley Laboratory, May 1977.

6. Epstein R., "Query processing techniques for Distributed, Relational Database System", UMI, Research Press.

7. Wong E. & Youssefi K. "Decomposition a strategy fo query processing", ACMTODS, Vol. No. 3, 1976.

8. Date C.J. "An introduction to Database Systems" Addison - Wesley, 1981

9. Unify Relational Database Management System, Tutoria Manual.

10. Chamberlin et. al., "Sequel2; A unified approach t Data Defination, Manipulation and control", IBM Journa of Research and Development, 1976.

11. I.W. Draffan, and F. Poole, Distributed Data Base (edited), Cambridge University press.

12. D.Gries, "Compiler Construction for Digital Computers" John Wiley and Sons, Inc. 1971