

**A NOVEL APPROACH IN DESIGN AND DEVELOPMENT
OF OBJECT-ORIENTED HETEROGENEOUS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM**

A THESIS

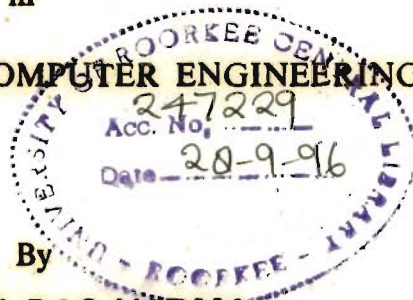
Submitted in Fulfilment of the
requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

ELECTRONICS AND COMPUTER ENGINEERING



By

GUR SARAN DAS VARMA



**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE-247 667 (INDIA)**

APRIL, 1994

Gratis

A NOVEL APPROACH IN DESIGN AND DEVELOPMENT
OF OBJECT-ORIENTED HETEROGENEOUS
DISTRIBUTED DATABASE MANAGEMENT SYSTEM

A THESIS

Submitted in Fulfillment of the
requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

ELECTRONICS AND COMPUTER ENGINEERING

BY

GUN SARAN DAS VARMA



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE-247 667 (INDIA)

APRIL, 1994

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **A Novel Approach In Design and Development of Object-Oriented Heterogeneous Distributed Database Management System** in fulfillment of the requirement for the award of the Degree of Doctor of Philosophy and submitted in the Department of **Electronics and Computer Engineering** of the University is an authentic record of my own work carried out during a period from January 9, 1992 to April 25, 1994 under the supervision of Dr. R. C. Joshi and Dr. K. Singh.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other University.



25/4/94

(Gur Saran Das Varma)


This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

Date:



25/4/94

(Dr. R. C. Joshi)
Professor
Electronics and Computer
Engineering Dept., U.O.R.



26/5/94

(Dr. K. Singh)
Prof. and Director
Continuing Education
Dept., U.O.R.

The Ph.D. Viva-Voce examination of Mr. Gur Saran Das Varma, Research Scholar, has been held on 10-4-1995



Dr. R. C. Joshi



Dr. K. Singh



Signature of External Examiner



(Dr. R. P. Agrawal)
Professor and Head
Electronics and Computer
Engineering, U. O. R.

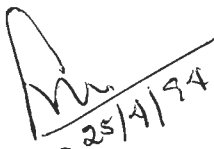
ACKNOWLEDGEMENT

I am deeply indebted to my guides Dr. R. C. Joshi, Professor, Department of Electronics and Computer Engineering and Dr. K. Singh, Prof. and Director, Continuing Education Department, University of Roorkee, Roorkee, for their invaluable guidance and support in pursuing this research work.

I would also like to express my sincere gratitude to Prof. P. S. Satsangi, Director, D.E.I. (Deemed) University, Dayalbagh, Agra, for his continuous encouragement and motivation to carry out further research in the field of Distributed Processing.

I wish to thank Dr. R. P. Agrawal, Dr. S. C. Gupta, Dr. A. K. Sarje, Dr. (Mrs.) K. Garg, Dr. J. P. Gupta, Professors, Department of Electronics and Computer Engineering and Dr. R. Thaper, Reader, Department of Electronics and Computer Engineering, University of Roorkee, Roorkee for providing encouragement and support for the research work.

Thanks are due to all my friends and relatives who have directly or indirectly contributed to the implementation of GURU, Object-Oriented, Heterogeneous, Distributed Database Management System. I would specially like to thank my Grandparents and Parents for their constant encouragement and interest throughout my research work.


25/4/94
(Gur Saran Das Varma)

ABSTRACT

Object-Oriented Heterogeneous Distributed Database Management Systems (OOHDDDBMS) are used to handle databases using the object-oriented programming (OOP) philosophy in a wide range of application domain, such as, running with different kinds of Database Management Systems, and using different programming languages. Object-Oriented Query languages (OOQLs) are more powerful than the relational query languages which are used widely.

Here a new OOHDDDBMS named, GURU is designed and implemented to supplement the various problems faced with OOQLs, which are available with research prototypes only. To tackle this problem, a new OOQL named, GSQL is designed and implemented which provides userfriendliness and runs the complete popular ANSI standard SQL (Structured Query Language) designed for relational database systems. Apart from this, GSQL supports its own powerful Object-Oriented Programming Language with semantic constructs. GSQL compiler uses its own (used with GURU) translator/mapper to support heterogeneous programming languages through GURU's shell.

The existing systems do not have the intelligent local query mechanism with object-oriented support which can work with different indexes automatically. We have presented an approach to handle data definition language part with intelligence to support queries in heterogeneous environment for the quick replies. This scheme helps in maintaining the perfect consistency of the database with the intelligent design of schema. The local schema is derived from the global schema and the site autonomy is preserved accordingly. The heterogeneous database fragments are

designed to provide the maximum access efficiency at the local sites.

An approach is suggested which integrates the heterogeneous schemas and provide the environment design accordingly, for handling an application at different remote terminal nodes with different database management systems. A design of translator/mapper is presented which helps in a heterogeneous global domain. A process known as user application registration is illustrated which is very helpful in the integration of the schemas. Further, a concept of cooperative system is provided, which helps network terminal nodes for the maintenance of applications on the global network with a modified master/slave concept.

An intelligent server design is presented which uses the knowledgebase to provide the different fragment allocation schemes with the available knowledge using trigger methods. The server helps the different other blocks like Object Manager, Data Manager etc. to coordinate the proper fragment handling operations. A concept of general object is presented which uses no files and no messages in the system. The different complexity measures are suggested. Later, the different ways are suggested, using them the complexity can be reduced considerably.

A concept of message-task is explained, which is used to provide great power to handle objects. The messages are scheduled like normal tasks by GURU and the burden on Operating System, to create separate processes and execute message-task is reduced. The three different layers are explained which are associated with GURU. These layers have the individual assigned tasks to maintain the isolation among the system, network and the

application tasks for the better privacy, security and operational efficiency.

A dynamically changing environment with changing object structures is proposed with no limits for the changes. The support of metaclass concept is also illustrated. There need not be any prior proposed ways in which the structure of the objects can be changed. The different thread and trigger mechanisms are suggested to trace the active objects and maintain the required consistency in the system. The concept of intensional and extensional notions are explained with GURU and the both notions are used with the proposed object structure in GURU. Further, the designs of object manager and data managers are explained which maintain the different versions of the objects and the database fragments used as objects. A concept of virtual object and object migration is explained with GURU.

The different privacy and security arrangements are explained with the dynamically changing object structures. The tight security with a new concept of extended object structure is provided which also helps in the protection of the object's view. The suggested methods use the object's privacy locks with embedded data in the object's structure. Some other information can also be used to relax locking dynamically, which extends a wide range of userfriendliness in the system.

GURU has been developed in C language, which consists of about 60,000 lines source code. A comparative study of GURU with the other existing DBMSs has been made.

LIST OF ABBREVIATIONS

2PC	2 Phase Commitment
2PL	2 Phase Locking
ADT	Abstract-Data Type
AT	Acceptance Test
CAD	Computer-Aided Design
CAI	Computer-Aided Instructions
CAM	Computer-Aided Manufacturing
CPU	Central Processing Unit
DBMS	Data Base Management System
DDBMS	Distributed Data Base Management System
DMTS	Distributed-Message Task Scheduling
DSMS	Distributed Shared Memory System
DVSMS	Distributed Virtual Shared Memory System
GSQL	GURU Structured Query Language
I/O	Input-Output
ISO	International Systems Organization
LAN	Local Area Network
MBCS	Message-Based Communication System
MM	Main-Memory
NLRB	Named-Linked Recovery Block
OID	Object-Identifier
OODB	Object-Oriented Database
OOHDBMS	Object-Oriented Distributed Database Management System
OOP	Object-Oriented Programming
OOPL	Object-Oriented Programming Language
OOQL	Object-Oriented Query Language
OSI	Open System Interconnection
SQL	Structured Query Language

TABLE OF CONTENTS

Pages

ABSTRACT	i
ACKNOWLEDGEMENT	
TABLE OF CONTENTS	
LIST OF ABBREVIATIONS	
CHAPTER 1 Introduction and Statement of the Problem	1
1.1 Introduction	1
1.2 Motivation for the Current Work	5
1.3 Statement of the Problem	12
1.4 Organization of the Thesis	13
CHAPTER 2 Review and General Considerations	17
2.1 Introduction	17
2.2 Network Scaling Problems	21
2.3 Distribution Design Methodology	22
2.4 Distributed Query Processing	25
2.5 Distributed Transaction Processing	28
2.6 Integration with Distributed Operating Systems	31
2.7 Distributed Multidatabase Systems	34
2.8 Object-Oriented Data Model	38
2.8.1 Objects and Object Identifiers	40
2.8.2 Aggregation	43
2.8.3 Methods	46
2.8.4 Metaclasses	54
2.8.5 Inheritance	55
2.8.6 Operational Aspects	60
2.8.7 Versions	60

CHAPTER 3 Query Automation	64
3.1 Introduction	64
3.2 Query Handling	66
3.3 Distributed Schema Handling	67
3.3.1 Global Schema Handling	69
3.3.2 Local Schema Handling	72
3.3.3 Applications of Schema for Local Processing	74
3.3.4 Server Design to Access Local and Global Schemas	75
3.4 Approach	78
3.4.1 Constraints	82
3.5 Design View	83
3.6 Fragment Design	88
3.6.1 Horizontal Fragment Design	88
3.6.2 Vertical Fragment Design	89
3.6.3 Mixed Fragment Design	90
3.6.4 Fragment Allocation	91
3.6.5 Versioning	92
3.7 Query Maintenance	93
3.7.1 Dependency Check	93
3.7.2 Index Organization	96
3.7.3 Query Transformation	101
3.7.4 Backlog Handling	103
3.7.5 Log Management	104
3.7.6 Server Management	105
3.7.7 Catalog Management	107
3.7.6 Data Dictionary Management	108
3.8 Conclusion	115
CHAPTER 4 Schema Design	118
4.1 Introduction	118
4.2 Approach	119

4.3	Schema Design Process	125
4.4	Distributed Schema Design	130
4.5	Local Schema Management	144
4.6	Conclusion	153
CHAPTER 5	Schema Integration	156
5.1	Introduction	156
5.2	Object Model's Properties	157
5.3	Environment Design for Schema Integration	159
5.3.1	Design Objectives	160
5.3.2	Design View	161
5.4	Intelligent Translator/Mapper Design	183
5.5	Heterogeneous DDBMS's Objectives Affected With Schema Design	190
5.6	Conclusion	193
CHAPTER 6	Complexity Measures	195
6.1	Introduction	195
6.2	Intensional and Extensional Notions	196
6.2.1	Applications	199
6.3	Brief Architecture of GURU	200
6.4	Complexity in Distributed Schema Handling	202
6.4.1	Query Mapping	203
6.4.2	Query Rerouting	203
6.5	Complexity With Transaction Handling Tools	204
6.6	Complexity Measures in Fragment Handling	204
6.6.1	Optimal Fragment Allocation	206
6.7	GSOL instructions	209
6.8	Conclusion	225
CHAPTER 7	Message-Task Scheduling	227

7.1 Introduction	227
7.2 Objects and Messages	228
7.3 Heterogeneous Message-Task Scheduling Policies	230
7.3.1 Objects	230
7.3.2 Task Handling	236
7.3.3 Message-Task Conversation	239
7.4 Conversation Policies	239
7.4.1 Conversation Structure	239
7.4.2 Exit Procedures	245
7.4.3 Acceptance Test	250
7.4.4 NLRB Scheme	252
7.4.5 Abstract Data Type Scheme	252
7.4.6 Heterogeneous Systems	253
7.5 Thread Management and Concurrency Control	253
7.6 Problem Implementation	255
7.7 Conclusion	260
CHAPTER 8 Objects and Dynamic Environments	262
8.1 Introduction	262
8.2 Structural Tools	263
8.3 Class and Object Structures	264
8.4 Extended User Defined Area	277
8.5 Class and Object Creation Methods	278
8.6 Object Management	286
8.6.1 Object Manager	287
8.7 Problem Implementation	287
8.7.1 Scenario	288
8.7.2 Implementation Strategies	289
8.8 Conclusion	292
CHAPTER 9 Object Protection and System Administration	293

9.1 Introduction	293
9.2 Kinds of Protections	295
9.3 System Level Protection	298
9.4 Extended User Defined Procedures	305
9.4.1 Triggers	310
9.5 System Administrator and User Level Protection	313
9.6 Message Protection	318
9.6.1 Ciphering	319
9.6.2 Port Arbitration	319
9.7 Problem Handling	320
9.8 Conclusion	320
CHAPTER 10 Conclusion and Future Scope of Work	323
10.1 Conclusion	323
10.2 Future Scope of Work	327
APPENDIX-A	329
APPENDIX-B	331
APPENDIX-C	368
APPENDIX-D	379
APPENDIX-E	384
APPENDIX-F	401
BIBLIOGRAPHY	406

ERRATA

Page No.	Para. No.	Line No.	Present Words	Should be read/modified as
95	3	2	<i>continue</i>	<i>continuous</i>
110	2	11	<i>appointment</i>	<i>appoint</i>
139	2	6	<i>can</i>	<i>can be</i>
140	2	2	<i>but,</i>	<i>and also</i>
144	1	6	<i>form</i>	<i>form of</i>
150	2	1	<i>creates</i>	<i>create</i>
154	3	11	<i>provide</i>	<i>provides</i>
155	1	1	<i>provide</i>	<i>provides</i>
170	2	6	<i>both of</i>	<i>both</i>
176	1	2	<i>assume</i>	<i>assumed</i>
176	1	4	<i>send</i>	<i>sent</i>
182	2	3	<i>DBMS</i>	<i>DBMSs</i>
200	2	6	<i>a</i>	<i>an</i>
203	3	8	<i>terminal</i>	<i>terminals</i>
206	2	11	<i>operation</i>	<i>operations</i>
214	2	3	<i>.not.</i>	<i>.not. are</i>
214	2	6	<i>give</i>	<i>given</i>
217	3	6	<i>belong</i>	<i>belongs</i>
225	4	4	<i>fragment</i>	<i>fragments</i>
313.B	3	9	<i>becomes</i>	<i>become</i>
315	2	9	<i>committed. The</i>	<i>committed, the</i>
318	2	1	<i>kind</i>	<i>kinds</i>

CHAPTER - 1

Introduction and Statement of the Problem

1.1 INTRODUCTION

A need of getting data and information from the remote computer terminal nodes is increasing rapidly. In the present day environment the latest data and knowledge play a major significant role. The lack of the latest data and information could create various adverse effects in the wide range of human and machine environments and the sufferings may not be repairable. Distributed database technology is one of the most important computing developments of the past few years. During this period a number of first generation commercial products have been evolved. It is expected that the distributed database technology will be effective for data processing in the same way as centralized systems did about a decade ago.

A distributed database is a collection of different logically interrelated databases, distributed over a computer network, connecting the different computer terminals, kept geographically apart. A distributed database management system (DDBMS) is a software system that manages the distributed database and makes the distribution transparent to users. The term distributed database management system is used to refer the combination of the distributed management component and the distributed databases.

The most of the current systems are built in the local area network environments, consisting of the each site a single computer system. The database is distributed so that each site

manages a single local database. The design of next generation DDBMSs will be different, however, as a result of technological developments - especially the emergence of affordable multiprocessors and high-speed networks with the upcoming object-oriented design philosophy. System design will also be affected by the increasing use of database technology in application domains that are more complex than business data processing and by the wider adoption of client-server mode of computing, accompanied by the standardization of the client-server interface. Thus the distributed DBMS environment will include multiprocessor database servers connected to high speed networks linking them and the other repositories to client machines that run application code and participate in executing database requests. Distributed relational DBMSs of this type are already appearing, and several existing object-oriented systems also fit this description.

One of the common feature of the next generation database systems is that they will need a data model more powerful than the relational model, without compromising its advantages (data independence and high-level query languages). When applied to more complex applications such as CAD/CAM, software design, office information systems, and expert systems, the relational model exhibits limitations in terms of complex object support, type system, and rule management. To address these issues two important technologies - knowledge bases and Object-oriented databases - are being investigated. Another major issue is going to be system performance as more functionality is added. Exploiting the parallelism available in multiprocessing computers in one promising way to provide high performance. Techniques designed for distributed databases can be useful but need to be significantly extended to implement parallel database

systems.

Object-oriented database management systems (OODBMSs) combine object-oriented programming and database technologies to provide greater modeling power and flexibility to programmers of data intensive applications [200]. Over the last five years, OODBMSs have been the subject of intensive research and experimentation, which led to an impressive number of prototypes and commercial products. But the theory and practice of developing distributed OODBMSs have yet to be fully developed [182], [142]. Distributed environments will make the problems even more difficult. In addition, the issues of data dictionary management and distributed object management have to be dealt with. However, distribution is an essential requirement, since applications that require OODBMS technology typically arise in networked workstation environments. The earlier commercial OODBMSs (for example, Gemstone) use a client-server architecture in which multiple workstations can access the database centralized on a server. However, distributing an object-oriented database within a network workstations (and servers) is becoming very attractive. In fact, some OODBMSs already support some form of data distribution transparency (Ontos and Distributed Orion, for example).

Knowledge base management systems try to make database management more intelligent by managing knowledge in addition to data. Capturing knowledge in the form of rules has been extensively studied in a particular type of knowledge base system called deductive database. Deductive database systems manage and process rules specified over large amount of data within the DBMS rather than in separate subsystem. Rules can be deductive (assertions) or imperative (triggers). Rule management is

essential, since it provides a uniform paradigm to deal with semantic integrity control, views, protection, deduction, and triggers. Much of the work in deductive databases has concentrated on the semantics of rule programs and on processing the deductive queries, particularly in the presence of recursive and negated predicates. But much work is needed to combine the rule support with object-oriented capabilities. For reasons similar to those for OODBMS applications, knowledge base applications are likely to arise in networked workstation environments. These applications can also arise in parallel computing environments when the database is managed by a multiprocessor database server. In any case we can simplify a number of issues by replaying on distributed relational database technology. Unlike most OODBMS approaches, which try to extend an object-oriented programming language, this similarity is a strong advantage for implementing knowledge bases in distributed environments. Therefore the new issues have more to do with distributed knowledge management and processing and debugging of distributed knowledge base queries than with distributed data management.

Parallel distributed systems are designed to exploit recent multiprocessor computer architectures to build high-performance and fault-tolerant database servers[13]. For example, by fragmenting the database across multiple nodes, we can obtain more interquery and intraquery parallelism. For obvious reasons such set-oriented processing and application portability, most of the work in this area has focused on supporting SQL.

The design problems of parallel database systems, such as operating system support, data placement, parallel algorithms, and parallelizing compilation, are common to both kinds of

architectures. If parallel data servers become prevalent, we can foresee an environment in which several of them are placed on a backbone network, giving rise to distributed systems consisting of processor clusters [81]. An interesting concern in such an environment is networking. Specifically, executing database commands that span multiple, and possibly heterogeneous, clusters creates at least the problems that are seen under distributed multidatabase systems. In addition, the queries must be optimized not only for execution in parallel on a cluster of servers, but also for execution across a network.

Here a new object-oriented heterogeneous distributed database management system, GURU is designed and implemented which runs its own structured query language known as GSQL (GURU Structured Query Language). GSQL supports the complete SQL along with its own object-oriented powerful query language with semantic constructs. The basic aim in the design of GURU is to remove the many more existing problems in the existing OOHDBMSs apart from a few discussed above.

1.2 MOTIVATION FOR THE CURRENT WORK

The working environment for the design can be defined in the three dimensions:

Autonomy refers to control distribution and indicated the degree to which individual DDBMSs can operate independently. It involves a number of factors including whether the component systems exchanges information. Whether, they can independently execute transactions, and whether one is allowed to modify them. Three types of autonomy are tight integration, semiautonomy and full autonomy. In tightly integrated systems a single image of

entire database is available to users who want to share the information in multiple databases. Semiautonomous systems consist of DBMSs that can operate independently but have been designed to operate in a federation to make their local data shareable. In fully autonomous systems, however, the individual components are stand-alone DBMSs, which know neither of the existence of the other DBMSs nor of how to communicate with them.

Distribution deals with data. The two cases can be considered: Either data are physically distributed over multiple sites that communicate with each other over some form of communication medium or they are stored only at one site.

Heterogeneity occurs in various forms, ranging from hardware heterogeneity and differences in networking protocols to variations in different data managers. The important forms of the heterogeneity from the perspective database systems are differences in data models, query languages, interfaces and transaction management protocols. The taxonomy classifies DBMSs as the homogeneous or heterogeneous.

Centralized database management system have taken us from a data processing paradigm in which data definition and maintenance were embedded in each application to one in which these functions are abstracted from the applications and placed under the control of a server called the DBMS. This new orientation results in data independence - the immunity of application programs to changes in the logical and physical organization of the data and vice versa. Distributed database technology intended to extend the concept of data independence to environments in which data is distributed and replicated over a number of machines connected by a network. Data independence

provided by several forms of transparency: network (and therefore, distribution) transparency, replication transparency and fragmentation transparency. Transparent access to data separates a system's higher level semantics from lower level implementation issues. Thus, database users would see a logically integrated single-image database, even though it was physically distributed, enabling them to access the distributed database as if it was a centralized one. In its ideal form, full transparency would imply a query language interface to a distributed DBMS no different from that to a centralized DBMS.

Most commercially distributed DBMSs do not provide a sufficient level of transparency. Part of the problem is the lack of support for replicated-data management. Some systems do not permit the data replication across multiple databases; systems that do permit it require that the user be physically logged on to one database at a given time. Some DDBMSs attempt to establish their own transparent naming schemes, usually with unsatisfactory results, requiring the users either to specify the full path to data or to build aliases to avoid long path names. An important part of the problem is the lack of proper operating system support for transparency. Network transparency can easily be supported by a transparent naming mechanism in the operating system. The operating system can also assist in replication transparency, leaving the task of fragmentation transparency to the DDBMS.

It is agreeable that the management of distributed data is more difficult transparent access is provided to users, and that the client server architecture with RPC-based communication is the right approach. In fact some commercially distributed DDBMSs are organized in this fashion (Sysbase, for example).

However, the original goal of providing the transparent access to distributed and replicated data should not be abandoned because of the difficulties. The issue is, what should take over the responsibility of managing distributed replicated data-the DDBMS or the application? It should be distributed DBMS, whose components can be organized in a client-server fashion. The related technical problems are among the remaining research issues that must be added.

Distributed DBMSs are intended to improve reliability, since they have replicated components and thereby eliminate single point of failure. The failure of the single site, or a communication link failure that make one or more sites unreachable is not enough to bring down the entire system. In distributed database this means that the some of the data may be unreachable, but, with proper care users may be permitted to access other parts of the database. This proper care comes in the form of support for distributed transactions.

A transaction consists of a sequence of database operations, executed as an atomic action that transforms a consistent database state to another consistent state, even when a number of such transactions are executed concurrently (some time called concurrency transparency), and even when the failures occur (called failure atomicity). Therefore, a DBMS supports full transaction support guarantees that current execution of user transactions will not violate the database consistency in the face of system failures as long as each transaction is correct-that is, obeys the integrity rules specified for the database.

Distributed transactions execute at multiple sites, where they access the local database. With full support for

distributed transactions, user applications can access a single logical image of the database and rely on the distributed DBMS to ensure that their requests will be executed correctly no matter what happens in the system. Correctly means that the user applications need not be concerned with coordinating their access to individual local databases, nor need they worry about the possibility of site or communication link failures during execution of their transactions. There is a link between a distributed transactions and transparency, since both involve distributed naming and directory management.

Data replication increases the data availability; copies of the data store at a failed or unreachable site exist at other operational sites. However, replica support requires the implementation of the control protocols that enforce a specified replica access semantics. The most straightforward semantics is one-copy equivalence, which can be enforced by the ROWA (read one, write all) protocol. In ROWA, a logical read operation on a replicated data item is converted to one physical read operation on any one of its copies, but a logical write operation is translated to physical writes on all copies. More complicated, less restricted replica control protocols based on deferring the rights on some copies, have been studied but are not implemented in any known system.

Better performance, The case for distributed DBMSs superior performance is usually based on two points. First, a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its point of use. This feature is called data localization, has two potential advantages: (1) Since, each site handles only a portion of the database, contention for CPU and I/O is not as severe as

centralize databases, and (2) Localization reduces remote access delays, which usually occur in wide area networks (for example the minimum round trip message propagation delay in satellite-based systems is about one second). Most DDBMSs are structured to gain maximum benefit from data localization. Full benefits of reduced contention and reduced communication overhead can be obtained only through a proper fragmentation and distribution of the database.

The second point in the favor of DDBMS's performance advantage is that the inherent parallelism of distributed systems can be exploited for interquery and intraquery parallelism results from the execution of multiple queries at the same time. Intraquery parallelism is achieved by breaking up a single query into a number of subqueries, each executed at different site, accessing a different part of a distributed database. If the user access to distributed database consisted only of querying (read-only access), then provision of interquery and intraquery parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires the implementation of the elaborate concurrency control and commit protocols.

Today, commercial systems employ two alternative execution models (other than the implementation of full distributed transaction support) to realize improved performance. The first alternative is to have the database open only for queries (read-only access) during regular operating hours, while updates are batched. The database is then closed to query activity during off-hours, when the batched updates are run sequentially. This run time-multiplexing between read activity

and update activity. The second alternative is based on multiplexing the database: Two copies of the database are maintained, one for ad hoc querying (called the query database) and the other for updates by application programs (called the production database). At regular intervals, the production database is copied to query database. The alternative does not eliminate the need to implement concurrency control and reliability protocols for the production database, since these are necessary to synchronize the write operations on the same data: however it improves the performance of queries, since they can be executed without transaction manipulation overhead.

The performance characteristics of the distributed database systems are not well understood. There are not enough true distributed database applications to provide a sound basis of practical judgments. In addition, performance models are not sufficiently developed. The database community has developed a number of benchmarks to test the performance of transaction-processing applications, but it is not clear whether they can be used to measure the performance of distributed transaction management. The performance results of commercial DBMS products, even with respect to these benchmarks, generally are not openly published. Nonstop SQL is one product for which performance figures, as well as experimental setup used in obtaining them, have been published.

Easier, more economical system expansion. In a distributed environment, accommodating increasing database sizes should be easier. Major system overhauls are seldom necessary; expansion can be usually handled by adding processing and storage power to the system. This is known as database size scaling, as opposed to network scaling. It may not be possible to obtain a

linear increase in power, since this also depends on the distribution overhead, but significant improvements are still possible.

1.3 STATEMENT OF THE PROBLEM

The main objective of this work is to design and develop a new Object-Oriented Heterogeneous Distributed Database Management System (OOHDDBMS), which has the following features apart from the existing features in the modern OOHDDBMS:

(i) To design and implement a Structured Query Language (SQL) with semantic constructs supporting Object-Oriented features. The SQL with the system, provides the support to heterogeneous environment management with a powerful compiler support for the concurrency control.

(ii) Some efforts will be made to design and implement an intelligent server, to process the different queries efficiently and to provide better user friendliness in the overall heterogeneous environment handling all kinds of fragments.

(iii) To introduce a concept of intelligent schema design and a new technique of schema integration with on line application registration in heterogeneous environment.

(iv) To propose a concept of object structures with triggers for complexity and performance measurement of the heterogeneous system.

(v) To modify the existing master/slave concept in Object-Oriented system, supporting the interaction among the active

objects for the global heterogeneous network environment.

(vi) To implement the concept of metaclasses in the Structured Query Language.

(vii) To provide a concept of virtual memory support with the object managers and the concept of message-task in a large database system environment.

(viii) To introduce the concept of unlimited support for dynamically changing object structures maintaining the different required versions in global heterogeneous environment.

(ix) To maintain the consistency, privacy and security in dynamically changing object structures and privilege levels.

1.4 ORGANIZATION OF THE THESIS

In Chapter 2, the review and general consideration for the existing database management systems are presented. Which discusses the various centralized and distributed database management systems. Further, the different existing homogeneous and heterogeneous distributed database management systems are compared with their specific features and later, the object-oriented distributed database management systems are compared.

An intelligent method to handle heterogeneous distributed queries is explained with the index mechanism in Chapter 3. A method is discussed which is used to handle schema in GURU OOHDBMS uses the intelligent techniques. The several existing approaches are discussed to explain the data structures used in memory to reply queries locally. It discusses the various

kinds of heterogeneous fragments and their design. Later, the fragment allocation is discussed. The server design and management to handle the intelligent techniques is also explained. Some important tools which are used for query handling and recovery are also explained. These tools include backlog, log, catalog and data dictionary for the distributed database management systems. A technique of maintaining the different versions of the database fragments is also explained.

The basic mechanism known as heterogeneous schema, used in GURU to handle the heterogeneous distributed database, through which all other remote terminal nodes can access the global database is presented in Chapter 4. An intelligent database view, which helps in providing good site autonomy is explained. The different ways are suggested which could be used for the object-oriented schema design approach to fetch the local schema.

A process of heterogeneous schema integration is illustrated in Chapter 5. The various kinds of schemas are explained which could be linked to provide global schema. A procedure known as user's schema registration is explained for the three kinds of registration procedures. Further, the design of server is discussed for the efficient translation process. The intelligent translator/mapper design is also presented. Later, the effects of heterogeneous database are discussed for the overall performance of the system. And the effect of schema integration on DDBMS objectives is illustrated.

Chapter 6, discusses the various complexity issues in the design of a heterogeneous distributed database management system. The intensional and extensional notions are explained for the object design from the complexity view point. Later, a brief

architecture of GURU is explained. Further, the allocation of heterogeneous fragments is discussed which affects the complexity for replying queries.

A concept of message-task in GURU is explained in Chapter 7, which provides a wide flexibility to control objects and handle the communication among objects efficiently. The procedure of object interaction for replying queries is discussed in detail. The mechanism of synchronously and asynchronously exited active objects is illustrated with a proper comparison. Some other communication protocols are also illustrated for the communication purposes among the different distributed objects. The mechanism used in GURU to protect the system through the different layers is also explained.

The changing object structures in dynamically changing environments are discussed in Chapter 8. The different kinds of procedures are illustrated which are used to change the object structures and maintain the database consistency. The changes of database view and the trigger procedures to watch changes and control the required objects is discussed. The mechanism to invoke objects using threads is explained. The design of object manager and the data manager with the object version support is also illustrated.

An arrangement to protect view of the objects is explained in Chapter 9. The existing protection schemes are discussed which are used to protect databases in various ways. The object-oriented design philosophy and the protection of object view in dynamically changing environment are illustrated. The various arrangements used with user level, system level and the process level protection are discussed. Some trigger schemes

are also suggested for guarding the various views of the objects.

In Chapter 10, the conclusion of the whole work is explained and the comparison of GURU with the existing distributed DBMSs is presented. Later, the future scope of the work and new research areas are illustrated.

Review and General Considerations

2.1 INTRODUCTION

The previous chapter discusses the current state of commercial DDBMSs and how well they meet the original objectives set for the technology. Obviously, the commercial state of art still has a way to go. The situation is not only that commercial systems have to catch up with and implement research results, but that significant research problems remain to be solved. The following issues still require more work:

(1) performance models, methodologies and benchmarks to better measure the sensitivity of the proposed algorithms to the underlying technology;

(2) distributed query processing to handle queries more complex than select-project-join, multiple query processing to save on common work, and optimization cost models to determine when such multiple query processing is beneficial;

(3) advanced transaction methods that differ from those defined for business data processing, better reflecting the processing mode (cooperative sharing, user interaction, long duration) common in the applications distributed database technology is going to support;

(4) Analysis of replication and its impact on distributed database system architecture and algorithms, and development of efficient replica control protocols that improve system availability;

(5) distributed DBMS implementation strategies that emphasize a better interface and cooperation with distributed

operating systems;

(6) theoretically complete, correct, and practical design methodologies for distributed databases; and

(7) the full set of problems related to the interconnection of autonomous information-processing systems.

The distributed database management systems can be classified into two categories, homogeneous and heterogeneous. The homogeneity and heterogeneity can be considered at the different levels in the distributed database. They are mainly the hardware, operating system and the local DBMSs. The differences at the various levels can be considered by observing the behavior with the communication software. The homogeneous DDBMS refers to a DDBMS with the same DBMS at the different sites on the computer network even if the computer systems and the operating systems are not same. A heterogeneous DDBMS at least uses two different DBMSs. Heterogeneous DDBMS adds the translation procedure to the complexity of homogeneous DDBMSs. Although the problems with heterogeneous DDBMSs are very hard and therefore, still the heterogeneous DDBMSs exist only as the research prototypes.

As seen earlier, many important technical problems await solution, and new ones will arise as distributed database management systems develop. These problems should keep distributed DBMS researchers and implementers busy for some time to come.

The first and most relevant application of database management systems technology were for business and administration. This influenced data organization and usage in DBMSs.

Recently, however, innovations have opened a market to new applications that require adequate software tools[33]. Typical examples include computer-aided design, office information systems, hypermedia systems, documentation of complex mechanical systems, knowledgebases, and scientific applications.

These applications require effective support for management of complex, possibly multimedia, objects. For example, hypermedia applications require handling of text, graphics and bit map pictures, while design applications might require support for geometric objects. Other crucial requirements derive from the evolutionary nature of applications and include multiple versions of the same data and long-lived transactions.

The support of complex objects imposes several requirements on both the object data model and object management. The data model should support the modeling of object structures and interrelationships in a natural way. The model should support not only an object structural definition, but also the modeling of object behaviors and dynamic constraints. In addition, in the intended application environments, the object structures, behavior and interrelationships evolve over time.

From the object management view point, the application characteristics, the object dimensions, and the length of operations require extending or completely modifying traditional DBMS architectures, techniques and algorithms to deal with a number of issues. For instance,

* Two modalities of access to objects should be provided (that is, access to a single object) based on some object identifier or name, and access to sets of objects, based

on declarative queries.

- * Object versioning mechanism should be provided to take into account different object evolution states, validity times, and information about alternatives.

- * Transaction can extend in time and involve large amounts of data. This requires revisiting the recovery and concurrency control mechanisms.

- * The evolutionary nature of applications makes schema modifications the norm rather than the exception. Therefore, execution of schema modification operations should be supported without requiring system slowdown and shutdown.

- * Protection mechanisms should be based on the object, which is the natural access unit.

To satisfy these requirements, one direction undertaken in the database is the design and development of object-oriented database management systems (OODBMSs). An object oriented data model lets the user model every conceptual entity by using a single modeling concept: the object. In addition mechanism such as aggregation and generalization let the user represent relationships among objects, and among object collections.

The object-oriented paradigm was primarily introduced in the design of advanced programming languages and environments such as smalltalk. A first attempt to apply this paradigm to data management has result in building of object-oriented interfaces on top of relational DBMSs. Mainly for performance reasons, this solution is generally unsatisfactory.

More advanced systems such as Avenge[28], Encore[200], Gemstone[36], Iris[192], O₂[67], Orion[96], and Vbase[12], have taken the approach of designing an architecture to directly

support the object-oriented paradigm.

The problem can be discussed more precisely in the following sections scalability, design methodology, query processing, transaction processing, integration, autonomy, multidatabase, and object-oriented data models etc.

2.2 Network Scaling Problems

The database community does not have a full understanding of performance implications of all the DDBMSs design alternatives. Specially, questions have been raised about the scalability of some protocols and algorithms as systems become geographically distributed or as the number of system components increases[77]. One concern is the suitability of the distributed transaction-processing mechanisms(the 2PL and, particularly, the 2PC protocols) in distributed database systems based on wide area networks. A significant overhead is associated with these protocols and implementing over a slow wide area network may pose difficulties[172].

Scaling issues are only a part of more general problem: There is no good handle on the role of network architectures and protocols in the DDBMS performance. Almost all the performance studies assume a very simple cost model, some time so unrealistic as to use a fixed communication delay independent of all network characteristics such as load, message size, network size, and so on[5],[22]. The inappropriateness of these models can be demonstrated easily. Consider, the DDBMSs that run on an Ethernet-type local area network. Message delays in Ethernet increases as the network load increases and generally cannot be bounded. Therefore, realistic performance models of an Ethernet-based DDBMS

cannot realistically use a constant network delay or even a delay function that does not consider the network load. In general, the performance of the proposed algorithm and protocols in different local area network architectures is not well understood, let alone their comparative behavior in moving from local area networks to wide area networks.

The proper way to deal with scalability to deal with general, sufficiently powerful performance models, measurement tools, and measurement methodologies. Such work has been going on for some time for centralized DBMSs but has not been yet sufficiently extended to DDBMSs. The questions raised regarding the suitability of the existing benchmarks, detailed and comprehensive situation studied do not exist either. Although there are plenty of DDBMS performance studies, they usually employ simplistic models, artificial work loads, or conflicting assumptions, or they consider only a few special algorithms. Making generalization on the basis of existing performance studies require a giant leap of faith. This does not mean that there is no understanding of trade-offs. In fact certain trade-offs have long been recognized, or even the designers of the earlier systems considered them. For example, the query processor of SDD-1 system was design to execute distributed operations most efficiently on slow wide area networks[122]. Later, studies considered the optimization of query processor in faster, broadcasting local area networks. But these trade-offs can mostly spelled out only in qualitative terms; their quantification requires more research on performance models.

2.3 Distribution Design Methodology

Distributed database design methodology varies

according to system architecture. For tightly integrated distributed databases, design proceeds top-down from requirements analysis and logical design of the global database to physical design of each local database. For distributed multidatabase systems, the design process is bottom-up and involves the integration of existing databases.

The step of interest to us in the top-down process is distribution design, which involves designing local conceptual schemas by distributing global entities over the sites of the distributed system. The global entities are specified within the global conceptual schema. In the relational model, both the global and local entities are relations, and distribution design maps global relations to local ones[49],[50]. The important research issue that requires attention is the development of a practical design methodology and its integration into the general data-modeling process. The two aspects of distribution design are fragmentation and allocation. Fragmentation is the partitioning of each global relation into a set of fragment relations. Allocation concentrates on the (possibility replicated) distribution of these local relations across the distributed system's sites. Research on fragmentation has concentrated on horizontal (or selecting) or vertical (or projecting) fragmentation of global relations. Numerous allocation algorithms based on mathematical optimization formulations have also been proposed[37],[99],[113],[183].

No underlying design methodology combines the horizontal and vertical partitioning techniques; horizontal and vertical partitioning algorithms have been developed completely independently. If one starts with a global relation, there are algorithms to decompose it horizontally and algorithms to

decompose it vertically into a set of fragment relations. However, there are no algorithms that fragment a global relation into a set of fragment relations of which some are decomposed horizontally and others vertically. Researchers always point out that most real-life fragmentations would be mixed, that is, would involve both horizontal and vertical partitioning of a relation; but research on how to accomplish this is lacking[182],[3],[40]. A distribution design methodology is needed that encompasses the horizontal and vertical fragmentation algorithms and uses them as part of a more general strategy. Such a methodology should take a global relation and a set of design criteria and come up with a set of fragments, some from horizontal and others from vertical fragmentation.

Allocation that the second part of distribution design, is typically treated independently of fragmentation. The process, therefore, is linear; the output of fragmentation is input to allocation. At first site isolation of the fragmentation and the allocation steps appears to simplify the formulation of the problem by reducing the decision space. However the closer examination reveals that isolating the two steps actually contributes to the complexity of the allocation models. Both steps have similar inputs, differing only in that fragmentation works on global relations whereas allocation considers fragment relations. They both require the information about the user applications (such as how often they access data and what the interrelationship of individual data object is), but each ignores how the other uses these inputs.

The end result is that the fragmentation algorithms decide how to partition a relation partly on the basis of how applications access it, but the allocation models ignore the part

this input plays in fragmentation[40],[65]. Therefore, the allocation models have to include all over again a detailed specification of the relationship among the fragment relations and how user applications access them. A more promising approach is to extend the methodology discussed above so that it reflects the interdependence of the fragmentation and the allocation decisions. The approach requires the extensions of existing distributed design strategies[115],[116].

The integrated methodologies such as the one proposed here may be complex. But combining these two steps may have synergistic effects enabling the development of acceptable heuristic solution methods. Some studies give us hope that such integrated methodologies and proper solution mechanism can be developed. In these studies, researchers build a simulation model of the DDBMS, taking as input a specific database design, and measure its effectiveness. Using such methods to develop tools that aid human designers rather than attempting to replace them is probably the appropriate approach to the design problem.

2.4 Distributed Query Processing

Distributed query processors automatically translate a high-level query on a distributed database, which is seen as a single database by users in an efficient low level execution plan expressed on the local databases. Such translation has two important requirements[182],[180]. First, the translation must be a correct transformation of the input query so that the execution plan actually produces the expected result. The formal basis for this task is the equivalence between the relational calculus and relational algebra, and the transformation rules associated with relational algebra. Second, the execution plan

must be optimal - that is, it must minimize the cost function that incorporates resource consumption. Thus, the query processor must investigate equivalent alternative plans to select the best one.

Because of the difficulty of addressing each two aspects together, they are typically isolated in two sequential steps; data localization and global optimization[141]. These steps are generally preceded by query decomposition, which simplifies the input query and rewrites it in the relational algebra. Data localization transforms and input algebraic query expressed on the distributed database into an equivalent fragment query (a query expressed on database fragments stored at different sites), which can be further simplified by algebraic transformations. Global query optimization generates an optimal execution plan for the input fragment query by making decisions regarding operation ordering, data movement between sites, and the choice of both distributed and local algorithms for database operations. Global operation gives rise to a number of problems. These have to do with the restrictions imposed on the cost model, the focus on a subset of the query language the trade-off between optimization costs and execution cost, and the optimization-reoptimization interval.

The cost model is central to global query optimization because it provides the necessary abstraction on the DDBMS execution system in terms of access methods and it provides the abstraction of the database interims of the physical schema information and related statistics[180],[200]. The cost model is used to predict the execution cost of alternative execution plans of a query. Important restrictions are often associated with the cost model, limiting the effectiveness of optimization in

improving throughput. Work in extensible query optimization, is useful in parametrizing the cost model, which can then be refined through experimentation.

Although query languages are becoming increasingly powerful, (for example, new version of SQL), global query optimization typically focuses on a subset of the query language, namely select-project-join queries with conjunctive predicates[178],[179]. This is an important class of queries for which good optimization opportunities exist. As a result of a good deal of theory has been developed for join and semijoin ordering. However other important queries warrant optimization, such as queries with disjunctions, unions, fixpoints, aggregation or sorting[1],[201]. A promising solution is to separate the language understanding from the optimization itself, which can be dedicated to several optimization modules.

A trade-off is necessary between optimization cost and the quality of the generated execution plans. Higher optimization costs are probably acceptable to produce better plans for repetitive queries, since these plans would reduce the query execution cost and amortize optimization cost over many executions. However, higher costs are unacceptable for ad hoc queries executed only once. Optimization cost is incurred mainly by searching the solution space for alternative execution plans. In a distributed system the solution space could be quite large because of wide range of distributed execution strategies[185]. Therefore, developing efficient strategies that avoid the exhaustive search approach is critical.

Global query optimization is typically performed prior to the execution of the query (hence, it is called static

optimization). A major problem with this approach is that the optimization cost model may become inaccurate because of changes in the fragment size or database reorganization, which is important for load balancing. The problem therefore, is to determine the optimal interval of query recompilation and reoptimization, taking into account the trade-off between optimization and execution cost.

2.5 Distributed Transaction Processing

It may be hard to believe in an area as widely researched as distributed transaction processing there are still important topics to investigate, but it is true[56],[49]. The scaling problems of transaction management algorithms are already discussed. Additional topics include replica control protocols, more sophisticated transaction models, and nonserializable correctness criteria.

In replicated DDBMSs, database operations are specified on logical data objects. The replica control protocols are responsible for mapping an operation on a logical data object to an operation on multiple physical copies of this data object. In so doing, they ensure the mutual consistency of the replicated database. The ROWA rule discussed earlier is the most straightforward method of enforcing mutual consistency. Accordingly, a replicated database is in a mutually consistent state if all copies of every data object have identical values.

The field of data replication needs further research and experimentation on replication methods for computation and communication and the systematic exploitation of application-specific properties. Experimentation is needed to evaluate the

claims made by algorithm and system designers, and lacks a consistent framework for comparing competing techniques. One of the difficulties in quantitatively evaluating replication techniques is the absence of commonly accepted failure incidence models. For example, Markov models, some times used to analyze the availability achieved by replication protocols, assume the statistical independence of individual failure events and the rarity of network partitions relative to site failures[57]. It is not currently known that either of these assumptions are tenable, nor it is known how sensitive Markov models are to these assumptions. Validation of the Markov models by simulation cannot be trusted in the absence of empirical measurements, since simulations often embody the same assumptions that underlie the Markov analysis. Thus, empirical studies are needed to monitor failure patterns in real life production systems, with the purpose of constructing a simple model of typical failure loads.

To achieve the twin goals of data replication - availability and performance - there is a need to provide integrated systems in which replication of data goes hand in hand with replication of computation and communication (including I/O). Only data replication has been studied intensively; relatively little has been done in the replication of computation and communication[9],[185]. Computation replication has been studied in several settings, including the execution of synchronous duplicate process as "hot standbys," and the implementation of different versions of the same software to guard against human design errors. Replication of communication messages, primarily by retry, has been studied in the context of providing reliable message delivery, and a few papers report on the replication of I/O messages to enhance the availability of transactional systems. However, more study is needed of how these

tools can be integrated with data replication to support such applications as real-time control systems, which may benefit from all three kinds of replication. This work would be invaluable for guiding operating system and programming language designers in developing the proper set of tools to support fault-tolerant systems.

In addition to replication, but related to it, more elaborate transaction models are required, especially models that exploit application semantics to achieve higher availability and performance, as well as concurrency[73],[74],[102]. As the database technology enters new application domains, such as engineering design, software development, and office information systems, the nature and requirements of transaction change. Thus work also is needed on correctness criteria other than serializability.

As a first approximation, transaction models can be classified along two dimensions: the structure of transactions and the structure of objects that transactions operate on. Along the transaction structure dimension, it recognizes flat transactions, closed-nested transactions such as sagas, and structures that include both open and closed nesting, in increasing order of complexity. Along the object structure dimension, it identifies simple objects (such as pages), objects as instances of abstract data types, and complex objects, again in increasing complexity. The last two are distinguished to indicate that objects as instance of abstract data types support encapsulation (and therefore are harder to run transactions on than simple objects) but do not have a complex structure (do not contain other objects), and their types do not participate in an inheritance hierarchy.

Within this framework, most of the transaction model work in distributed systems has concentrated on the execution of flat transactions on simple objects. This point in the design space is well understood. While some work has been done in the application of nested transactions on simple objects, much remains to be done, especially in distributed databases. Specially the semantics of these transaction models are still being worked out[110],[3]. Similarly, work has been done on applying simple transactions to objects as instances of abstract data types and to complex objects. These are initial attempts that should be followed up to specify their full semantics, their incorporation into a DBMS, their interaction with recovery managers, and their distribution properties.

Complex transaction models are important in distributed systems for several reasons. First, transaction processing in distributed multidatabase systems can benefit from the relaxed semantics of these models. Second, the new applications that distributed DBMSs will support in the future (such as engineering design, office information systems, and computer assisted cooperative work) require transaction models incorporating more-abstract operations that execute on complex data. Furthermore, these applications have a sharing paradigm different from typical database access we are accustomed to. For example, computer assisted cooperative work environment require participants to cooperate in accessing the shared resources rather than to compete for them as is usual in database applications[81],[103]. These changing requirements necessitate the development of new transaction models and accompanying correctness criteria.

2.6 Integration with Distributed Operating Systems

The undesirability of running a centralized or distributed DBMS as an ordinary user application on top of a host operating system has long been recognized[81],[173]. There is a mismatch between the requirements of the DBMS and the functions of current operating systems. This is even more true in the case of distributed DBMSs, which require functions that existing distributed operating systems do not provide - for example, distributed transaction support with concurrency control and recovery, efficient management of distributed persistent data, and more complicated access methods. Furthermore, DDBMSs necessitate modifications in how the distributed operating systems perform their traditional functions (task scheduling naming, buffer management). Here, the various issues are highlighted in distributed DBMS - distributed operating system integration: system architecture, transparent naming of resources, persistent data management, remote communication, and transaction support[159],[191],[200].

An important architectural consideration is that the coupling of distributed DBMSs and distributed operating systems is not a binary integration issue. The communication network protocols also must be considered, adding to the complexity of the problem. Thus the architectural paradigm must be flexible enough to accommodate the distributed DBMS functions, distributed operating system services, and communication protocols standards such as the ISO/OSI (Open System Interconnection) model or the IEEE 802, in the context efforts that include too many database functions inside the operating system kernel or those that modify tightly-closed operating systems are likely to prove unsuccessful[63],[67],[74],[114]. In our view the operating system, should implement only essential operating system services

and only those DBMS functions that it can implement efficiently: then it should get out the way. The model that best fits this requirements seems to be the client server-architecture with a small kernel that provides the database functions that can be provided efficiently and does not hinder the DBMS in efficiently implementing other services at the user level (examples are Mach and Amoeba). Object orientation may also have a lot to offer as a system structure to facilitate this integration.

Naming is the fundamental mechanism available to operating system for providing transparent access to system resources. Whether or not access to distributed objects should be transparent at the operating system level is a contentious issue involving the trade-off between the data management flexibility and ease of use on the one hand and the system overhead on the other[137],[148]. For a distributed DBMS, transparency is important. As is indicated earlier, many distributed DBMSs attempt to establish their own transparent naming scheme without significant access. Further investigation of the naming issue and the relationship between distributed directories and operating system name servers is needed. A worthwhile naming construct that deserves some attention in this context is the capability concept, which was used in older systems such as Hydra is being used in more modern operating systems such as Amoeba.

Storage and management of persistent data - data that survives past the execution of the program that manipulates it - is the primary function of the database management systems. Operating systems have traditional dealt with persistent data by means of files systems. If a successful cooperation paradigm can be found, it may be possible to use the DBMS as the operating system file system. At a more general level, cooperation among

the programming languages, the DBMS, and the operating system manage persistent data requires further research. Distributed file systems do not address distributed DBMS concerns because either they do not provide for concurrent access to data or granularity of sharing is too large.

Two communication paradigms that have widely studied in distributed operating systems are message passing and the remote procedure call[201],[160]. The relative merits of these approaches have long been debated, but the simple RPC semantics (blocking one-time execution) have been appealing to distributed system designers. As discussed earlier an RPC-based access to the distributed data at the user level is some times proposed in place of fully transparent access[81]. But implementing an RPC mechanism for heterogeneous computing environment is not easy. The problem is that the different vendors' RPC systems do not interoperate. It may be necessary to view communication at higher levels of abstraction to overcome heterogeneity or at lower levels of abstraction (message passing), to achieve more parallelism. This trade-off needs further study.

In current DBMSs, the transaction manager is implemented as part of the DBMS[189],[187],[111]. Whether transactions should and can be implemented as part of standard operating system services has long been discussed. There are strong arguments on both sides, but the clear resolution of the issue requires more research as well as more experience with the various general-purpose (that is non-DBMS) transaction management services.

2.7 Distributed Multidatabase Systems

Multidatabase system organization is an alternative to logically integrated distributed databases. The fundamental difference between the two approaches is the level of autonomy afforded the component data managers at each site. While integrated DBMSs' components are designed to work together, the multidatabase management system consist of components that may have no notion of cooperation. Specially, these components are independent DBMSs, which means, for example, that although they may have facilities to execute transactions, they are incapable of executing distributed transactions that span multiple components.

The autonomy and the potential heterogeneity of component systems create problems in query processing and especially in query optimization. The basic problem is the difficulty of global optimization when local cost functions are not known and the local cost values cannot be communicated to multi-DBMS [183]. It has been suggested that semantic optimization based on qualitative information may be the best we can do, but semantic query processing is not fully understood either. It may be possible to develop the hierarchical query optimizers that perform some amount of global query optimization and then let each local system perform further optimization on localized subquery. This may not provide an optimal solution but may enable some optimization. The emerging standards, also may make sharing some cost information easier.

The autonomy of the underlying DBMSs makes transaction processing in autonomous multidatabase systems more difficult. Since they are autonomous, they have their own transaction processing services (transaction manager, scheduler, recovery manager) and are capable of accepting local transactions and

running them in completion. The multi-DBMS layer has its own transaction-processing components, in charge of accepting and coordinating global transactions that access multiple databases. A global transaction is divided into subtransaction, each of which is submitted to one of the component DBMSs[187]. However, since the multiDBMS is not aware the local transactions, it cannot control the local conflicts, nor can it control indirect conflicts between global transactions caused by the interference of local transactions.

Various solutions have been proposed to deal with concurrent multidatabase transaction processing. Some use global serializability of transactions as their correctness criterion; others relax serializability. Most of this work should be treated as preliminary attempts at understanding and formalizing the issues. Many issues remain to be investigated[147]. One area of investigation is revisions in the transaction model and correctness criteria. Initial attempts have been made to recast the transaction model assumptions, and this work should continue. The nested-transaction model looks particularly promising for multidatabase systems, and its semantics, based on knowledge about the transaction's behavior, need to be formalized. In this context the meaning of consistency in multidatabase systems should be considered. A good starting point is the four degrees of consistency defined by Gray[81].

Another difficult issue that requires further investigation is the reliability and recovery aspects of multidatabase systems. The autonomy of individual DBMSs makes it difficult to incorporate 2PC into global transaction processing, thus making it difficult to enforce distributed transaction atomicity. Although researchers have addressed the topic in

recent studies, their approaches are initial engineering solutions. Reliability and recovery protocols for multidatabase systems still need to be developed and integrated with concurrency control mechanisms[147].

Atomicity is the major contributor to additional complexity of multidatabase systems. One of the main impediments to further development of these systems is the lack of understanding of the nature of autonomy. The autonomy is itself probably composed of several factors. Thus, the nature of autonomy must be clearly and precisely characterized. Furthermore, most researchers treat autonomy as if it were an all-or-nothing feature. Even the taxonomy considered here, indicated only three points along this dimension. But the spectrum between no autonomy and full autonomy probably contains many distinct points[110],[111]. It is essential in our opinion, to (1) precisely define what is meant by "no autonomy" and "full autonomy," (2) precisely delineate and define the many different levels of autonomy, and (3) identify the degree of database consistency possible for each level. At that point, it will make sense to discuss the different transaction models and execution semantics appropriate at each level. In addition this process should enable the identification of the layered structure, similar to the ISO/OSI model, for the interoperability of autonomous and possibly heterogeneous database systems. Such a structure would allow us to specify interfacing standards at different levels. Some work is already under way on the remote data access (RDA) standard, and this line of work will make practical solutions to the interoperability problem possible.

The initial promises of distributed database management systems - transparent management of distributed and replicated

data, improved system reliability by means of distributed transactions, improved system performance by means of interquery or intraquery parallelism and easier and more economical system expansion - are met to varying degrees by today's commercial products. Full realization of these promises depends not only on commercial technology's catching up with research results, but also on our solving a number of problems.

The changing nature of technology underlying distributed DBMSs will make parallel database servers feasible[31]. This will affect the database systems in two ways. First, implementing distributed DBMSs on these parallel database servers will require revision of most of the existing algorithms and protocols to operate on the parallel machines. Second, the parallel database servers will be connected as servers to networks, requiring the development of distributed DBMSs that will have to deal with a hierarchy of data managers. Further more, as distributed database technology infiltrates nonbusiness data processing domains, the capabilities required of these systems will change, forcing a shift in emphasis from relational systems to more powerful data models. Current research along these lines concentrates on object-oriented and knowledgebase systems.

2.8 Object-Oriented Data Model

The object-oriented paradigm is based on five fundamental concepts[164], [200]:

(1) Each real-world entity is modeled by an object. Each object is associated with a unique identifier.

(2) Each object has a set of instance attributes

(instance variables) and methods; the value of an attribute can be an object or a set of objects. This characteristic permits arbitrary complex objects to be defined as an aggregation of other objects. The set of attributes of an object and the set of methods present the object structure and behavior respectively.

(3) The attribute values represent the object's status. This status is accessed or modified by sending messages to the object to invoke the corresponding methods.

(4) Objects sharing the name structure and behavior are grouped into classes. A class represent a template for a set of similar objects. Each object is an instance of some class.

(5) A class can be defined as a specialization of one or more classes. A class defined as a specialization is called a subclass and inherits attributes and methods from its superclass(s).

However, there are many variations with respect to these five concepts. In fact they are used mainly as a way to organize the discussion, rather than as definition of the object-oriented paradigm.

An OODBMS can be defined as DBMS that directly supports a model based on the object-oriented paradigm. Like any DBMS, it must provide persistent storage for objects and their descriptors (schema). The system must also provide a language for schema definition and for manipulation of objects and their schema. In addition to these basic characteristics, an OODBMS usually includes a query language and the necessary database mechanisms for access optimization, such as index and clustering, concurrency control and authorization mechanisms for multiuser accesses, and recovery.

2.8.1 Objects and Object Identifiers

In object-oriented systems, each real world entity is uniformly represented by an object. Each object is uniquely identified by an object identifier. The identity of an object has an existence independent of its value. Using OIDs lets objects share subobjects and makes possible the construction of general object networks.

The notion of an object identifier is different from the concept of key in the relational data model[94],[156]. A key is defined by the value of one or more attributes and therefore, can undergo modifications. But, two objects are different if they have different object identifiers, even if their all attributes have the same values. Note that sharing objects in models where identity is based on value leaves the applications to manage key values and the associated normalization problems.

However, there are models in which both objects and values are allowed; in these models not all entities are objects. Informally a value is self-identifying and has no associated OID. In some models, all the primitive entities, such as integers or characters are values, while all other entities are objects.

Other models, notably Avance[28], and O_2 . provide the possibility of defining complex (or structured) values. Complex values cannot be shared among objects. They are built using the same constructors as those provided for objects.

In general complex values are useful in situations where aggregates (or sets) must be defined to be used as components of other objects but will never be used alone. If

complex values are not allowed, these aggregates must be defined by using a class and must have an OID associated with them. Therefore, some performance penalty is incurred.

An example is dates. Suppose a date is defined as a tuple of three components, respectively, representing month, day and year. Dates are likely to be used as components of other objects. However it is unlikely that user will issue a query on the class of all dates. Therefore, it appears more convenient to define dates as complex values rather than as objects.

The notion of object identity introduces at least two different notions of equality among objects. The first denoted here by $=$, is the identity equality. Two objects are identity-equal, or identical if they have the same OID.

The second, denoted here by $==$, is the value equality. Two objects are value-equal if all their attributes that are values are equal and all their attributes that are objects are recursively value equal. That is, the two objects have the same content, even if they have two different identifiers. Two identical objects are also value equal, but two value equal objects are necessarily identical.

Different approaches for building OIDs can be devised. For example, in the approach used in Orion system[96], an OID consist of the pair \langle class identifier, instance identifier \rangle . The first element is the identifier of the class to which the object belongs, and the second identifies the object within the class.

The complete definition of attributes and methods for all instances of a class is factored and kept in an object

representing the class itself (called a class-object). When a message is sent to an object, the system extract the class identifier from the object identifier and access the class object to determine the message validity and fetch the corresponding method. This approach has the major disadvantage of making object migration from one class to another (for example, in case of object reclassification) difficult or even impossible, since this would require the modification of all object identifiers[81]. Therefore, all references to migrated objects would be invalidated.

In another approach, used for example in the Iris system, the OID does not contain the class identifier. The identifier of the class to which an object belongs is generally kept as control information stored in the object itself. To determine whether a message is valid for a given object, the system must first fetch the object and then retrieves it from the class identifier. Therefore, nonvalid messages cause useless object accesses, and type checking is rather expensive.

In both previous approaches the OID is logical; that is it does not contain any information about the object location and secondary storage. Therefore, a correspondence table exists for mapping OIDs onto physical addresses.

Based on physical identifiers, O₂ uses a different approach, in which each object is stored in a Wisconsin Storage Subsystem record and the OID is the record identifier. The OID does no change even if the record is moved to a new page, for example, when the record grows too big for the page it resided on. A forward marking technique (as in the relational systems System R and Ingres) handles cases of records that are moved to

different pages.

The approach used in O₂ has the main advantage that the persistent OIDs are provided that support fast access to objects, since there is no need to map the OID on the physical location. The major disadvantage is that the temporary OID must be assigned to an object created on a site different from the object store site (for example, on a workstation).

To avoid excessive message exchange, the permanent OID is assigned only when the object is stored on the corresponding Wiss record, at a transaction commit time[187],[188]. This implies that all reference generated during the transaction to newly created objects must be updated at transaction commit time.

The OID can also contain the object location for distributed databases. For example, in the distributed version of the Orion system[96], the OID contains the identifier of the site where the object was created. When an object migrates to a different site, its OID does not change. The object creation site keeps information concerning the new storage site of the object, so messages can be appropriately forwarded to the object.

2.8.2 Aggregation

The values of an object's attributes can be other objects, both primitive and nonprimitive[191]. When the value of an attribute of an object O is nonprimitive object O', the system stores the OID of O' in O. When complex values are supported by the model, the system usually stores in the object attribute the entire complex value.

Using complex values as components are more efficient than using objects. In the first case once the object O is fetched, all components that are complex values are usually fetched as well. When objects are used as components, the object O contains only the OIDs of its components.

Additional access messages might be needed to retrieve the component objects. Therefore, the data model might provide the possibility of defining complex values. As pointed out in Bjonerstedt and Hulten[28], depending on these differences, the database designer can implement an entity as an object or as a complex value, if the data model supports both. More sophisticated strategies, however, are used when (complex) values are large (typically larger than the page size), such as in the case of multimedia data types. In this situation, the value is not stored with the object of which it is a component. Deux describes an example of storage strategy for large (complex) values[67].

In defining complex objects and values, different constructors can be used. A minimal set of constructors that should be provided by a model includes set, list, and tuple[13]. In particular the set constructor, allows multivalued attributes and set objects to be defined. The list is similar to the set, but it imposes an order on the elements. Finally, the tuple constructor is important because it provides a natural way to model properties of an object.

As discussed in Atkinson et al.[13], the object constructors should be orthogonal; that is any constructor should be applicable to any object, including, of course, objects constructed using any constructor whatsoever. This is noted that

some models impose the constraint that the tuple be the first-level constructor. This implies, for example, that when defining a set object, the object must be defined as a tuple of a single attribute, which is in turn defined as a set.

The notion of a composite objects is found in some object models. As stated, a complex object can recursively reference any number of other objects. The references, however don't imply any special semantics that can be interest to different classes and applications.

One important relationship that could be superimposed on the complex object is the part-of relationship, that is, the concept that an object is part of another object. A set of component objects forming a single entity is a composite object.

A similar concept is found in Atkinson et. al.[13], where two different types of references are defined: general and is-part-of. The pair of relationship among objects has some consequence on object operations. For example, if the root of the composite object is removed, all component objects are deleted.

Moreover, in some models of composite objects, an object can be part of only one object; that is, the part-of relationship imposes an exclusivity constraint. In some systems, a lock on the root of a composite object is propagated to all the components.

Some extended relational models and object-oriented programming languages (for example, the loops language) also provide the notion of composite objects[80]. However in some models and papers, the term complex object means the composite

object.

2.8.3 Methods

Objects in an object-oriented database are manipulated using methods. In general, a method definition consists of two components. The first is the method signature, specifying the method name, the name and classes of the arguments, and the class of the result, if there is one.

Some systems like Orion[96], don't require the class of arguments and the results to be declared. This happens when the type checking is executed at runtime and there is therefore no need to know this information in advance.

The second component is the method implementation, consisting of code written in some programming language. Different OODBMSs use the different languages for method implementation. For example, both Vbase and O₂ use C language, while Orion uses Lisp. Gemstone uses Opal, which is nearly identical to Smalltalk.

In addition to the method signature and implementation, other components can be present in the method definition. For example, in Vbase the method definition can specify some trigger methods in addition to the base method and exceptions that can be raised by the method execution. Trigger methods are often used to augment the semantics of inherited methods and system defined methods - for example, the creation and deletion methods.

The language used to write methods is also used to write applications. In addition, some systems provide access to

the database from conventional languages. Gemstone, for example, supports access from C and Pascal.

Often, an object's attributes cannot be directly accessed in object-oriented programming languages. The only way to access attributes is to invoke the methods available at the object interface (strict encapsulation).

In databases, a lot of applications read or write attribute values. Queries are often expressed as Boolean combinations of predicates on attribute values. Therefore, most OODBMSs provide direct access to attributes by means of system-defined methods. Examples of these methods are "get" and "set" of Vbase[12], used to respectively read and write a given attribute.

These methods, provide the part of the system, have an efficient implementation and save the user from having to write a large amount of trivial code. The major drawback of providing these system-defined methods is that an object attribute must sometimes be manipulated only through some user defined methods. If, however, these system defined methods are available, nothing prevents a user from using the system-defined method rather than the user-defined method (unless the user employs authorization or they're disciplined).

Therefore, some systems (for example, Vbase[12] and the system described in Bertino et al[26].) let the user redefine the implementation of these methods for a given attribute. Each time the attribute is accessed the user defined method implementation is invoked, instead of system defined implementation. This allows the right semantics to be imposed, when needed on the system-defined methods. Moreover, as discussed in Andrews and

Harris[12], this capability is useful when importing data from external databases.

In O_2 , it is possible to make attributes visible from outside objects on user demand. Two special clauses of the class definition language, `public read` and `public write`, provide the function of making attributes directly available for reading and writing. If the `public read` (and, respectively `public write`) clause is associated with an attribute of name `Aname` appearing in the definition of class `C`, the system makes the attribute public for reading (and, respectively, writing).

In OODBMSs characterized by distributed or client-server architectures, an important architectural issue concerns the site where an invoked method is executed. In Gemstone, for example, the application designer has the option of moving an object on which a method has been invoked to the workstation (and then executing the method locally) or executing the method remotely on the server.

A simple option is provided in O_2 system. In general, the choice concerning the method execution site can be rather complex, since different factors must be taken into account, such as complexity of manipulations executed on the object, the references made to other objects during method execution, the network bandwidth and the competition for the network and for the server.

Classes and instantiation mechanisms. Instantiation is the first reusability mechanism (the second is inheritance) in that instantiation makes it possible to reuse the some definition to generate objects with the same behavior and structure. Object-

oriented data models provide the concept of class as the instantiation basis. A class is an object that acts as a template. As such a class specifies the intended purpose of its instances by defining

- * a structure, that is, a set of instance attributes (or instance variables);
- * a set of messages that define the external interface; and
- * a set of methods that are invoked by messages.

In this sense a class can be viewed as a specification for its instances. Since the class factors the definitions of set of objects, it is also an abstraction mechanism.

Given a class it is possible to generate through the instantiation mechanism objects that answer all messages defined in the class. The system keeps the attribute values for each object separately. Replicating the message and method definitions is not necessary. These are kept in the class object, since they are the same for all the class instances.

All instances of a given class have the same structure and behave similarly. The O_2 system however, lets exceptions be defined at instance level. In O_2 , an instance can have additional attributes and methods. Additional methods characterize the exceptional behavior of an instance. If an additional method has the same name as a method defined at class level, the instance method definition overrides the method definition provided at class level.

An alternative approach to instantiation is using

prototypical objects. This involves generating a new object starting from another existing object by modifying its attributes and/or its behavior. Therefore, a prototype is an individual object containing its own description; a prototype can also be used as a model for creating other objects.

This approach is useful when objects evolve (that is, modify their structure and behavior) quickly and are characterized more by their differences than their similarities. The approach is also useful when there are few instances for each class. In this way, the proliferation of many classes, each with few instances, is avoided.

In general, the approach based on the instantiation mechanism is more appropriate for mature application environments in which object properties and behaviors are consolidated. In fact, the approach based on instantiation makes it more complicated to experiment with alternative object structures and behaviors.

The prototype-based approach appears more appropriate based on the initial phase of application design, in application environments that evolve quickly with fewer consolidated objects and, in applications in which classes have few instances.

OODBMSs have usually adopted the instantiation approach since, in most cases, database applications have a lot of instances of each class for which an efficient storage organization must be provided. However, expect the broadening of the scope of database applications to also result in the use of prototypical objects or similar mechanisms.



So far, it is implicitly assumed that an object is an instance of only one class. However in some models, the instances of a class C are also members of the superclasses of C. As Moon shows[128], to distinguish between the notions of instance of a class and member of a class.

An object is an instance of a class C if C is the most specialized class associated with the object in a given inheritance hierarchy. An object is a member of a class C if it is an instance of C or some subclass of C.

Most object-oriented data models restrict each object to being instance of only one class, even though they let an object be a member of several classes through inheritance. However, object-oriented data models[200] can be found and let an object be an instance of several classes.

As an example, consider a class Person with subclass Student and Pilot and the case of person P being a student and a pilot. This situation can be easily modeled by associating both classes with P. Therefore, P will be an instance of both Student and Pilot and will also be a member of Person through the inheritance hierarchy.

These models provide the name classification mechanisms so to solve ambiguities deriving from attributes and methods, with the same name being used in the class of which an object is an instance[157]. Note, however, that even if the data model imposes the restriction that each object is an instance of only one class, multiple inheritance (discussed later in this article) can be used to handle situations like one discussed above. For example, a subclass could be defined as Student-Pilot, having a

superclass both Student and Pilot, and make instance of this subclass.

In all object-oriented database models, each attribute has associated a domain specifying the class of the possible objects that can be assigned to it as values. This differs from certain programming languages, such as Smalltalk, in which instance variables don't have an associated type. For data management applications requiring efficient management of persistent data to allocate the appropriate storage and access structures, the system must know the types of the possible values taken by attributes. Thus, even the Gemstone[36] system, derived from Smalltalk, requires the declaration of the instance variable domains in certain cases.

The fact that an attribute of a class C has a class C' as domain implies that each instance of C takes an instance of class C', or any subclass of C', as the value of the attribute. This establish an aggregation relationship between the two classes.

An aggregation relationship from class C to class C' specifies that C is defined in terms of C'. Since C' in tern defined in terms of other classes, the definition of class C results in aggregation hierarchy. An aggregation hierarchy can contain cycles, since classes can be recursively defined.

An important question concerning instances and classes is whether an object can change class. The ability to change the class of objects provides support for object evaluation. It lets an object change its structure and behavior and still retain its identity. The Encore[200], Gemstone[36], and Iris[192] systems

provide this capability, while most of the OODBMSs don't.

A domain constraint problem arises when objects are allowed to change class. As discussed earlier the value of an attribute A of an object O can be another object O'. If the O' changes class and its new class is not compatible with the class domain of A, O will contain an incorrect object as the value of A. A possible solution, illustrated by Zodnik[200], consist of placing a tombstone in O', to indicate that the object has changed class. The major disadvantages of this solution is that the applications must contain code to handle the exception of referenced object being an instance of a class other than the expected one.

In addition to acting as a template, the class in some systems also denotes the collection of all its instances, that is, its extension. This is important because the class becomes the base on which queries are formulated.

The concept of query has meaning only if applied to sets of objects. In systems where the class does not have this extensional function, the model provide set constructors for object grouping. Queries are then issued on the sets defined by these constructors. In this respect, there are differences among the various systems. For instance,

* In some systems, for example Gemstone[36], the class has only the specification meaning. A collection constructor groups objects of the same class. It is also possible to define several collections all containing instances of same class. Queries are issued on object collections. Moreover, indexes are defined on collections and not on classes.

* In other systems, for example Orion[96], the class has a meaning of both specification and extension.

* Finally, other systems, for example O_2 , provide the notions of type and class. In O_2 instances of type are values and therefore don't have OIDs (that is, type generate complex values), while instances of class are objects. Moreover, in O_2 , a class has associated its extension only if explicitly required by the user in the class definition (through the special keyword with extension). Therefore in O_2 the class has extensional function only if required by the user.

In general, the notion of decoupling specification from the notion of extension is found correct. The major draw back is that the data model becomes more complex compared to a simpler model in which the class acts both as object template and object extent.

2.8.4 Metaclasses

If each object is an instance of a class, and a class is an object, the model should provide a notion of metaclass. A metaclass is a class of a class[93],[94]. Metaclasses are crucial to support expert systems and AI applications. However, most OODBMSs don't provide metaclasses.

Finally, some data models provide the possibility of defining attributes and methods that characterize the classes as objects. Such attributes and methods, called class-attributes and class-methods, specify the properties and behavior of the class and not of its instances of a class. An example of a class-

attribute would be an attribute containing the average of an attribute value, evaluated on all instances of a class.

2.8.5 Inheritance

The concept of inheritance is the second reusability mechanism. It lets a class, called a subclass, be defined starting from the definition of the another class called the superclass. The subclass inherits the superclass attributes, methods, and messages. In addition subclass can have specific attributes, methods and messages that are not inherited. Moreover, the subclass can override the definition of the superclass attributes and methods.

Therefore, the inheritance mechanism lets a class specialize another class by additions and substitutions. Inheritance represents an important form of abstraction, since the detailed differences of several class descriptions are abstracted away and the commonalities factored out as a more general superclass.

A class can have several subclasses. Some systems let a class have several superclasses (multiple inheritance), while the other impose a restriction of a single superclass (single inheritance). Based on inheritance, the set of classes in the schema can be organized in an inheritance graph (orthogonal to the aggregation hierarchy). The inheritance graph is a tree when the model does not provide multiple inheritance. Unlike the aggregation hierarchy, an inheritance graph might not have cycles.

The possibility of defining a class from other classes

simplifies the task of class definition[128]. However, it can cause conflicts, especially in the case of multiple inheritance. If the name of an attribute (or method) explicitly defined in a class is the same as the attribute of a superclass, the attribute from a superclass is not inherited; that is, the definition in the subclass overrides the superclass definition.

If the model provides multiple inheritance, other types of conflicts can arise. For example, two or more superclasses might have an attribute with the same name but different domains.

Usually, rules are defined for solving conflicts. If the domains in the superclasses are related by inheritance relationships, the most specific domain is chosen for the subclass. If the domains are not related by inheritance, the user can specify from which superclass the attribute must be inherited.

In O_2 , for example, the name of an attribute in a subclass can be followed by the from clause containing the superclass from which the attribute definition must be inherited. If the user does not specify the inheritance paths, the solutions used in most models is to inherit the attributes (or methods) based on an order of precedence among superclasses.

In the last two cases questions might arise on the validity of inherited methods. An inherited method can contain in its implementation a reference to an attribute A. This attribute, however has not been inherited in a given subclass C, since an attribute with the same name but different domain has been inherited in the subclass from a different superclass.

When an inherited method is invoked on an instance of the subclass, inconsistencies can arise when the semantics and properties of A are not those expected in the method.

As mentioned earlier, the inheritance mechanism lets the implementation of an inherited method be overridden in the subclass. This is accomplished by defining in the subclass a method with the same name and a different implementation.

Each time a message is sent to an instance of the subclass, the implementation local to the subclass will be used to execute the method. This results in a single name denoting different method of implementations. However, this unit of change (that is the entire method) can be too coarse, since in some situations it might be desirable to refine the object behavior rather than completely change it.

Mechanisms to accomplish this have been proposed in the framework of object-oriented programming languages. For example, Smalltalk supports procedural combination of new and inherited behavior through pseudovariable `send super` (denoted as `<- Super`). When `send super` is used during the execution of a method invoked by a message `m`, the superclass method answering message `m` is invoked.

The CLOS language[128] provides mechanisms supporting declarative method combinations, based on the notion of `before method`, `primary method`, and `after method`. All `before methods` are invoked before the primary method, whereas all `after methods` are invoked after primary method.

Before and after methods are subject to some

limitations in that they cannot modify the control structure or the results of the primary method. In addition, CLOS provides "around" methods supporting procedural methods combinations. An around method, if applicable, takes precedence over all other methods and controls whether and when all other methods are called. Moreover, an around method can modify the input parameters and results of the methods it invokes.

Vbase[12] is an OODBMS that provides procedural method combination and a pseudovisible '\$\$' for this purpose. As discussed earlier, a method definition in this system can contain a base method and an arbitrary number of trigger methods.

When a method is invoked, the first trigger method is actually invoked. This trigger method then transfers the control to the next method by using '\$\$' syntax. The next invoked method is the next trigger method, or the base method, if there are no more trigger methods.

Once the base method is executed, the superclass method is invoked. Therefore, in Vbase there is a fixed invocation order: first all trigger methods, on the basis of their declaration order in the method definition, then the base method, and finally the superclass method.

When there are no trigger methods, the '\$\$' pseudovisible behaves like the `<- Super` of Smalltalk. This method combination is procedural, since the control to the next method must be explicitly passed by using the '\$\$' syntax. Therefore, the first invoked method might decide, on the basis of certain conditions, to return from the execution without invoking the other methods.

Often, the notion of subtyping is also found in OODBMSs. It is important however, not to confuse inheritance with subtyping, even if there is often a unique mechanism providing both functions.

For this discussion, a brief characterization is presented for the difference between these two concepts as follows. Inheritance is a reusability mechanism allowing a class to be defined from another class, possibly by extending and/or modifying the superclass definition. A type T, on the hand, is a subtype of a type T' if an instance of T can be used in place of an instance of T'. Therefore, subtyping is characterized by a set of rules ensuring that no type violations occur when the instance of a type T replaces an instance of supertype of T.

The fact that a class C is a subclass of a class C' does not necessarily imply that C is also a subtype of C'. Subtyping, however, influences inheritance, since it can restrict the overriding and can impose conditions on multiple inheritance so that the subtyping rules are not violated. An example of restriction on overriding is the requirement that when the domain of an attribute is redefined in a subclass, this domain must be a subclass of the domain associated with the attribute in the superclass. Wenger discusses inheritance and subtyping[191].

For this article, the behavioral and structural (or inclusion) subtyping are distinguished. In behavioral subtyping, a type T is a subtype of T' if T provides methods with the same name and the same (or compatible) arguments as T' (and possibly additional ones).

The criteria for behavioral subtyping are often based on the notion of conformity[29]. Conformity can only be used when the method signatures include the type of arguments and the result. In structural subtyping, a type T is a subtype of T' if T provides the same attributes as T' or attributes compatible with those of T' (and possibly additional ones).

Usually, most OODBMSs enforce only structural subtyping, even if subclasses inherit both attributes and methods from the superclasses. For example, the O₂ system[67] uses structural subtyping, while methods use a condition different from conformity. This condition leads to a less restrictive type system that does not guarantee that an instance of a subclass can always be safely used in place of an instance of a superclass. Instead, the Vbase system[12] enforces both structural and behavioral subtyping, using the notion of conformity.

2.8.6 Operational Aspects

Effective support of an object-oriented data model requires the techniques and algorithms traditionally used for data management to be extended and/or modified. Moreover, applications that are expected to use OODBMSs require additional functionalities, such as version mechanisms or long transactions, that are not usually provided by traditional DBMSs.

2.8.7 Versions

In traditional DBMSs, once transaction updated have been committed and permanently installed, the previous values of data usually are discarded. However, advanced applications, especially design applications, require facilities to maintain

object versions, that is, to keep different states of the same object. This requirement is inherent in applications that are exploratory and evolutionary.

Versions are useful mechanisms for maintaining object histories, that is, to keep track of object evolutions during time. Moreover, versions can be used to provide different alternatives of the same object. This is particularly useful in design applications, where different designers might need to explore different design choices in parallel. OODBMSs providing versions include Avance[28], Iris[192], and Orion[96].

In an OODBMSs, a versioned object O is a collection of objects that derived directly or indirectly from O . The fact that a version V_i is derived from a version V_j establishes a derivation relationship between the two version objects. The set of all versions related by derivation relationships is a version hierarchy[96].

V_i and V_j are first class objects. Therefore, they have their own OIDs and can be directly accessed and modified. Each version is a snapshot of an object state. The version management mechanism maintains all the information necessary to connect all versions of an object.

A common way to represent the version hierarchy is to store it in a special object, called the root object. This object has its own OID, like any other object. All version mechanisms provide operations for the management and inspection of version hierarchies, such as determining the predecessor or successors of a given version. Bjornerstedt and Hulten present a taxonomy of operations[28].

A versioned object O can be referenced by another object by either a specific reference (also called static), or a generic reference (called dynamic by Kim[96]).

In the first case, the reference is to a specific version of O , while in the second case, the reference is to the root object of O . When using a generic reference, the system has a task of determining the version to be returned. This is the default version; it is usually the last generated.

The Iris[192] system provides a more sophisticated mechanism to solve generic reference to versioned objects. The mechanism is declarative and is based on the notion of context. A context is a set of user-defined rules triggered on the invocation of methods. Using those rules, a user can specify that if certain conditions are true, each time a generic reference to an object is found during the execution of a method, a given version is to be used instead of the default version.

Object versions are often classified as stable or nonstable. Stable versions are considered consolidated enough to be used as a basis for branching alternatives or important enough to be saved as a reference point in object histories. In general, some restrictions are placed on modifications that can be executed on a stable version. For example, it is possible to delete but not modify a version if other versions have been derived from it.

Updating a stable version would require update propagation algorithms to ensure that versions derived from it are properly updated. As Kim noted[96], if a stable version needs

modification, a version can be derived from it containing the desired changes.

Nonstable versions are not yet consolidated and therefore can undergo modifications. A nonstable version can change into a stable version on explicit user request (through operations such as freeze[28] or promote[6]) or automatically by the system. For example, in Orion[96], a nonstable version is automatically promoted by the system if a version is derived from it.

In general, most version mechanism don't support version merging. That is, a new version is constrained to have only one direct predecessor.

The Avance system[28] provides some limited support for version merging. Whenever a new version must be derived as a merge of several existing versions, the user has to indicate one more version. Only a main version is copied into the new version, while it is left to the user to access the other versions to be merged and extract relevant information from them. The system connects a direct successor the new version with the merged versions. Therefore, the version hierarchy becomes a general graph.

Some version mechanisms support versions of classes. A major application of this is in the support of schema evolution, since it is possible to first define a class and then modify it without losing the previous class definition and the instances generated from it. The design features of GURU are discussed in the subsequent chapters. In the next chapter, discussions about the query automation are presented.

QUERY AUTOMATION

3.1. INTRODUCTION

In the present available database management systems this has been found that the process of structuring the fragments on different query structures is complicated [57],[6],[143],[47]. Most of the DDBMSs handle only the horizontal and vertical fragments. A very few DDBMSs deal with the mixed kind of fragments [142]. The effectiveness, the efficiency and the response time are mostly dependent on the locality and the optimum size of the fragments and their distribution [180],[197].

Whenever any interquery or intraquery arrives to be processed on a terminal node, first it is required to be properly analyzed and tuned as per the available database and index structure. This is carried out by the server deployed for accessing the right structure.

In general, this has been a practice with distributed database management systems that, while programming the applications on a computer system using a data definition language and a data manipulation language, the programmers have to be totally dependent on the database structure and the query structure to process queries. The query processing efficiency could be increased if a good structuring between the two is established [152]. That shows, if the existing scheme works with the available query languages, that the programmer needs to know the individual details of the fragments and their respective

maintenance done by the distributed database management system, would lead to the difficulties and inefficiency in processing queries. The database being maintained by a DDBMS is configured in the way required by the owner. Therefore, the task of maintaining the database fragments requires deep understanding about the system [50],[90]. During the system maintenance there has to be a perfect understanding about the system, so that, the programming is done efficiently in the required application domain [3]. Any wrong information furnished in the system leads to wrong results. Further, this has been observed practically that, maintaining indexes manually with large databases is a tough task and prone to errors[52].

Therefore, this is desirable, if the entire loading operations are made in built and automatic with the system so that the required tasks can be carried out efficiently. This removes the possibility of wrong specifications and maintaining the other specifications externally, hence solving a great application programming task. To automate these tasks, a required intelligent support structure, not available with the local schema and the terminal node can be automatically maintained by the DDBMS. Therefore, users are not required to be bothered about the understanding of the global and local schemas also.

This is necessary that the concept of class should be maintained during the all kinds of transactions performed among the different terminal nodes of the network intended to change the schema structure [126],[26],[180]. The classes have subclasses and superclasses and the different classes can be coupled inside an object and the different objects can be placed inside a class [108]. Therefore, the very nature of the classes and their inheritance should be maintained in the subclasses and

the superclasses. This maintenance with OOP is a tough task and leads to confusion, specially with large number of objects and classes. Therefore, an attempt has been made to design and develop an intelligent indexing mechanism to provide fast query support.

3.2 QUERY HANDLING

To process a query, the first step is to check the available local schema whether, all required domains in the query satisfy the local schema or not. The query is required to satisfy the attributes for vertical fragments and the domain structure or tuples for the horizontal fragments. Besides this, there could be some mixed fragments also. The mixed fragments have some key associated with them like the vertical fragments and the key lies with the fragment support structure. The key has a specific relation with the class of objects lying in the local schema of the current terminal node. The individual fragments are linked with some basic pointers, stored with the individual attributes (objects). The pointers address the specific terminal nodes with the physical address of the fragments. This helps in locating the right attributes at the specified keys but it is different for horizontal fragments. The horizontal fragments maintain the fragment qualifier in the catalog, and also in the global and local schemas.

The consistency of the data is maintained after watching the object-oriented view that is, the concept of subclasses and superclasses being maintained. The benefit of resolved hierarchical structure is to get inheritance of the relationships at the higher and related nodes. There is a concept of parent, sibling and child, which makes the all other nodes informed about

the search procedure. Therefore, the concepts of supernodes, which belong to the parent part or the subnodes, which belong to the child part, are provided. The required relations are traced by following a sequence from the root node in the hierarchy. The different nodes could only be searched, if the software accessing them has the permission for the search. This is done because the different groups of the users have the different privilege levels, assigned by the system administrator and they would get access over the nodes accordingly. The server checks if the appropriate object requiring to access a class is authorized for the same or not. If the user is performing some unauthorized operations with the database then he or she is informed accordingly with some messages.

There are two ways to process the queries, either by seeing the destination to which the query is referred and the results are subsequently processed from it, or shifting the required fragments, which are transferred to the current node. Later, the new fragments are updated at the appropriate levels in the local schema. Finally, the query is processed at the local terminal node increasing the future locality of the system, but at the same time it is also increasing the redundancy. This transfer of the fragments also increases the overhead of maintaining the new fragments in the catalogs. The catalog maintains the information, like version numbers and the different addresses of the fragment's destinations etc. on the different terminal nodes.

3.3 DISTRIBUTED SCHEMA HANDLING

In GURU DDBMS there are the two kinds of basic schemas, which are global and local schemas.

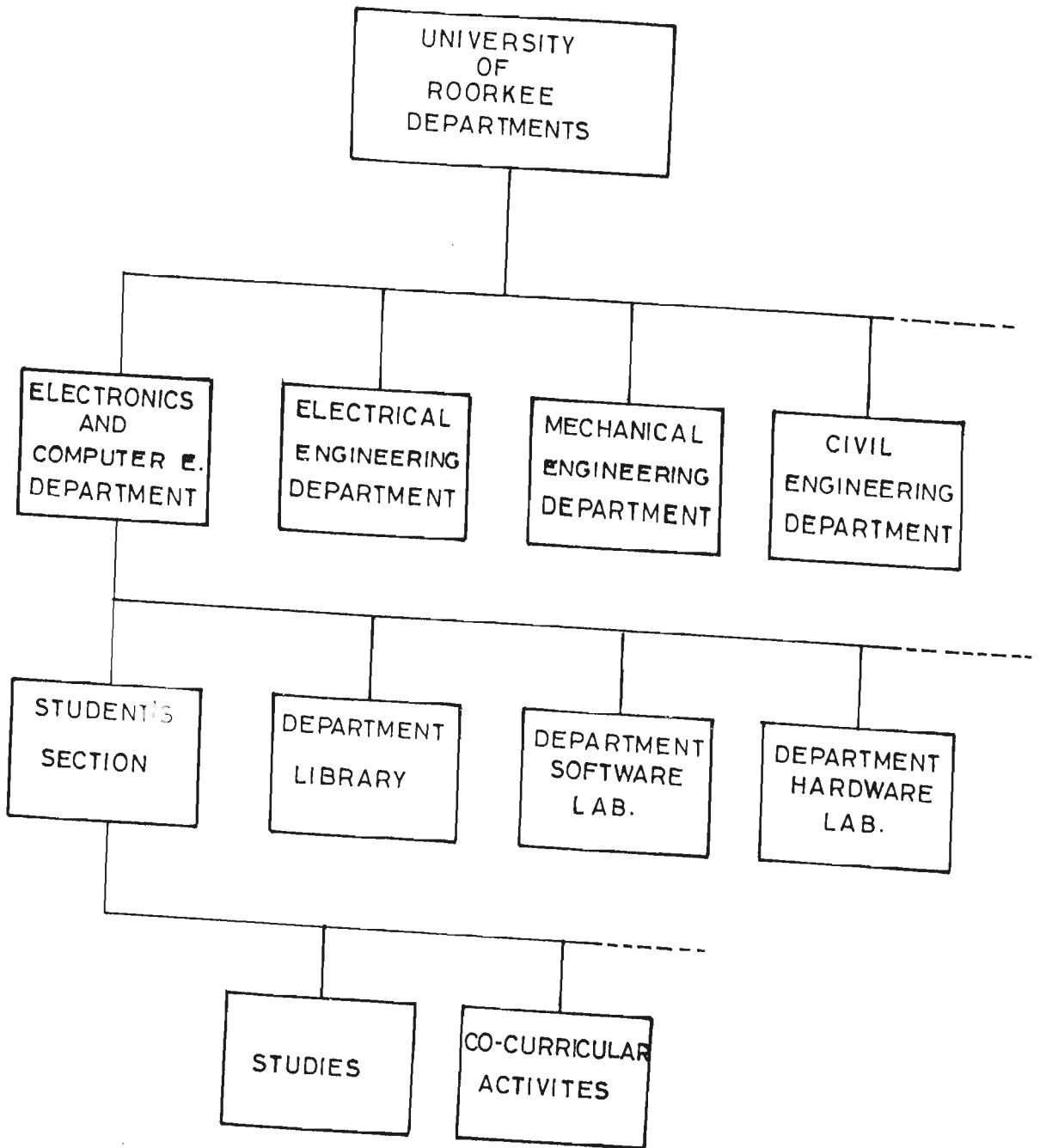


FIG. 3.1 SCHEMATIC HIERARCHICAL STRUCTURE OF UNIVERSITY OF ROORKEE DEPARTMENTS

3.3.1 Global Schema Handling

The global heterogeneous schema maintains the entire details about the relationships among various objects and classes at an instant of time. Every class can have so many other objects and classes. Thus, providing the high scalability to the database. The objects consist of the basic entities or attributes to have better cohesiveness or integration for the purpose of providing perfect privacy and security and the better operational efficiency. These objects and classes also maintain the linked information about the various kinds of fragments, which can be given access to the right users operating the system. The global schema is common for the entire application as a whole and can be kept at more than one terminal nodes of the distributed system. This increases the availability of the distributed system. The more replicated copies in the system enhance the availability of the system.

The distributed schema design can be illustrated with some example pertaining to University of Roorkee and the maintenance of the data for the same. Roorkee University is organized with so many Managerial Heads, which come under its administration. The university tree is represented in the form of a block diagram in Fig. 3.1. The several departments of the university are under its direct administration with a head, University of Roorkee Departments, which is divided under so many heads like Electronics and Computer Engg. Deptt., Electrical Engg. Deptt., Mechanical Engg. Deptt., and Civil Engg. Deptt. etc. Further the Electronics and Computer Engg. Deptt. is sub-divided in the following heads like Student's Section, Deptt. Library, Deptt. Software Lab., and Deptt. Hardware Lab. etc. Then, the head

Student's Section is sub-divided in the heads like Studies, and Co-curricular Activities etc. And again, there is a sub-division possible and so on and so forth.

The schema showing the structural relationships, which can be prepared from the symbolic notations is given bellow:

NODE-1.

U-O-R-D(E-C-E-D, E-E-D, M-E-D, C-E-D).

E-C-E-D(STUD, D-L, D-S-L, D-H-L).

STUD(STUDIES, C-C-A).

All nodes of the tree represent the head names, and the child nodes represent the child node's head name and its relationship with the parent node.

The following are the GSQL instructions used to construct the above schema and process queries:

```
define structure stud
datafiles frg_1
fields studies, c_c_a
procedure stud;
```

```
define structure e_c_e_d
include structure stud, d_l, d_s_l, d_h_l
procedure e_c_e_d;
```

```
define structure u_o_r_d
include structure e_c_e_d, e_e_d, m_e_d, c_e_d
procedure u_o_r_d;
```

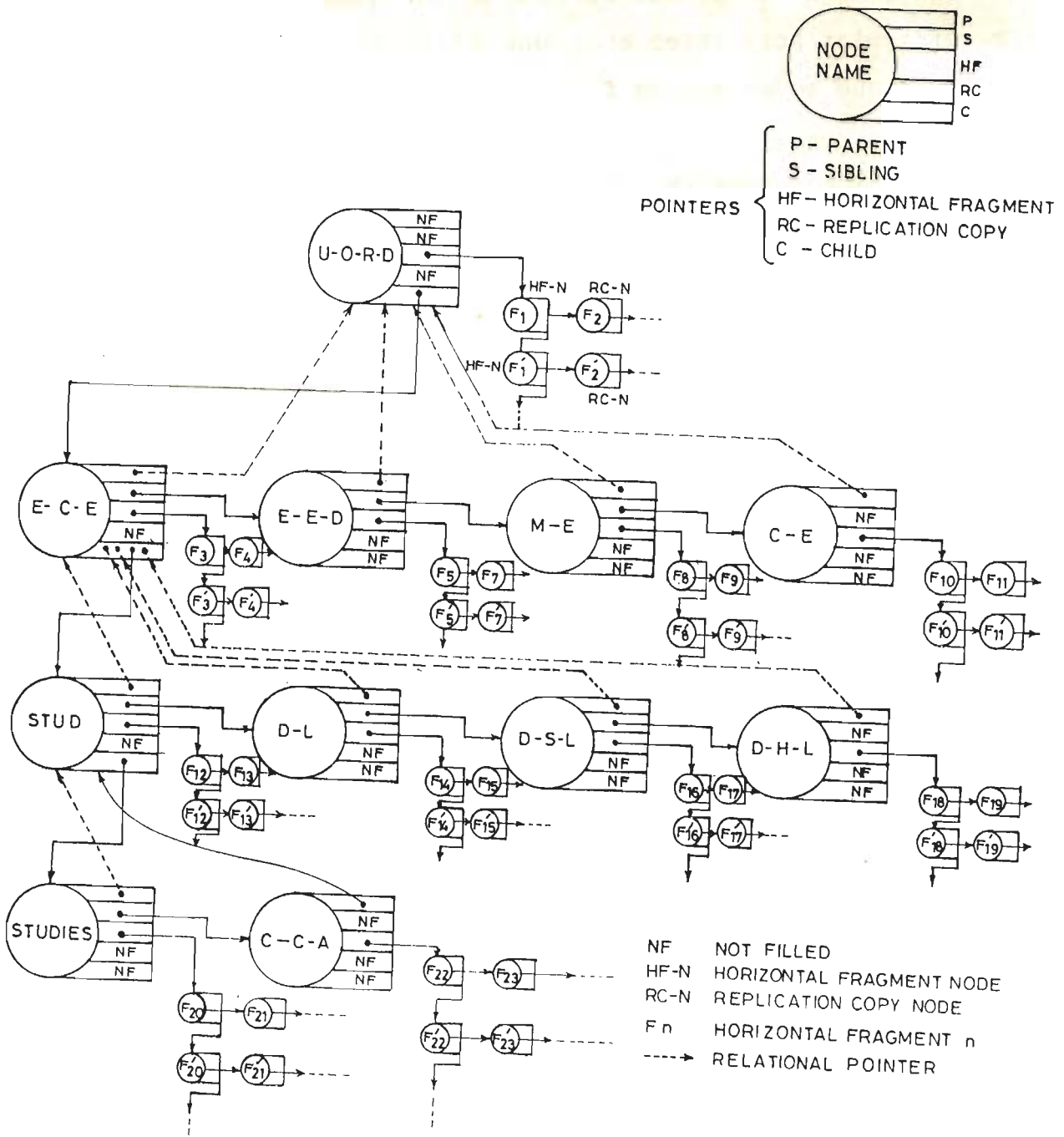


FIG. 3.2 GLOBAL SCHEMA STRUCTURE

```
create university with u_o_r_d;  
list studies of university;
```

Different objects are invoked with the messages sent to them. In brief, the following are the activities associated with GURU:

- * GURU system is composed of named address spaces called objects. Objects provide data storage, data manipulation, data sharing, concurrency control, and synchronization.

- * Control flow is achieved by threads invoking objects.

- * Data flow is achieved by parameter passing.

3.3.2 Local Schema Handling

The local autonomy is provided by maintaining the local schema with the individual terminal nodes, which is maintained at the different individual sites. The local schema depends on the requirements of the site for the data updates to be made in the local terminal node exclusively [76],[57]. Therefore, the local schemas are designed to see the site requirements at an instant of time. These classes and the objects can further have the different classes and the objects into them, as discussed in case of the global schema. Encapsulation of the various objects and the classes is done to ensure the privacy, security and the tight integration among the various entities available in the classes and the objects. In the local schema, the details of the various fragments are recorded at every node as illustrated in Fig. 3.2. The local catalog is also maintained at the local terminal node to record this information. The local database

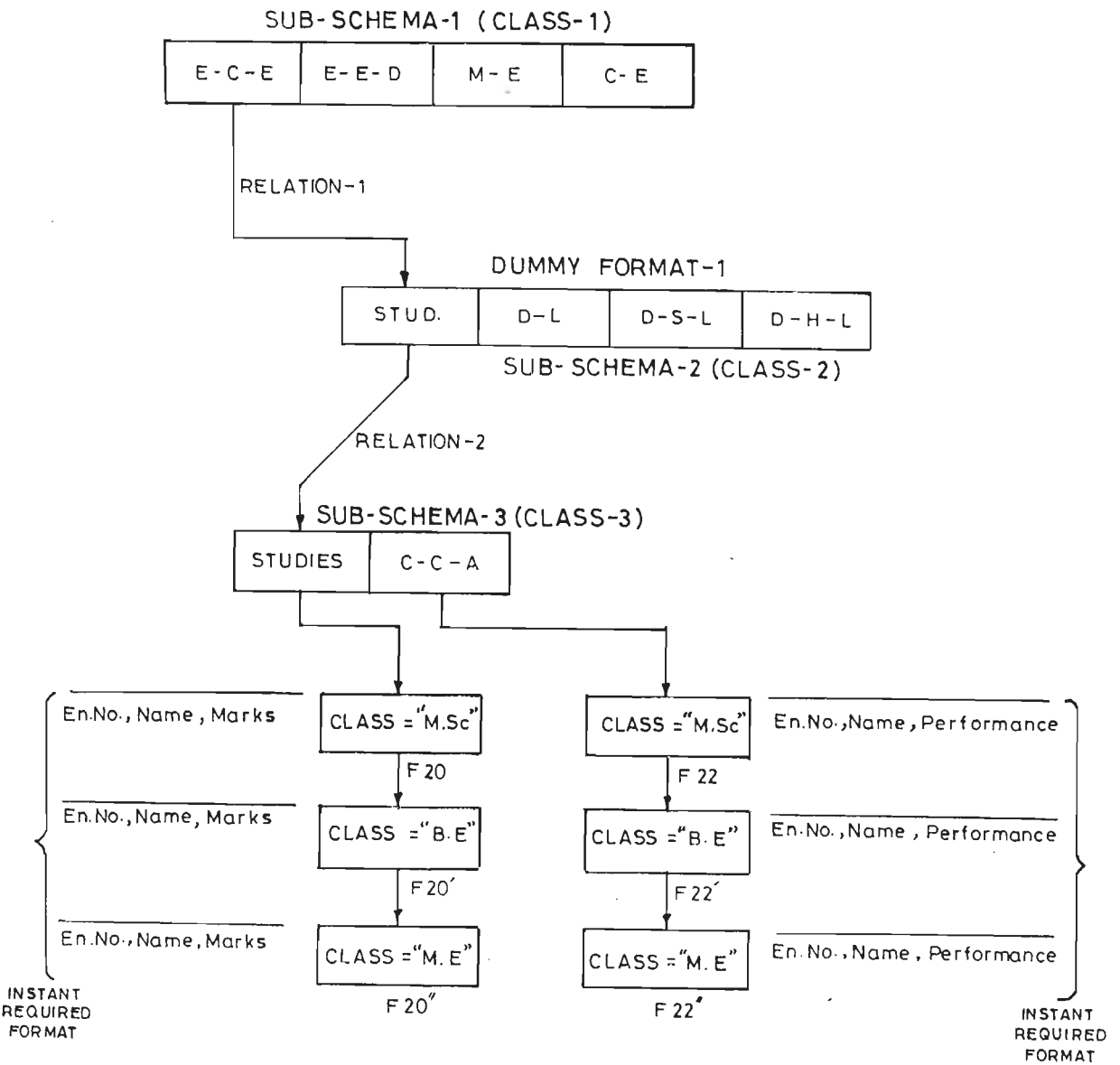


FIG. 3.3 LOCAL SCHEMA

maintained at the different local sites is having the entire information regarding the different entities with the database and the kind of fragment relationships as shown in the global and local schemas of Fig. 3.2 and Fig. 3.3 respectively. In this way, it helps in maintaining the different indexes, which are recorded with the different relationships of the fragments. For the different entities, which are common between the two fragments, maintaining a relationship, the different indexes are created. These indexes take the view of any order of the entities required by the queries otherwise, one concatenated index could easily be maintained for the two fragments. The information regarding the various common entities is marked in the global and local schemas. And the indexes are recorded in the catalog. The information about the indexes is also being indirectly maintained in the global and local schemas by placing the address of the catalog entry, where the information about index files is lying.

3.3.3 Applications of Schema for Local Processing

In GURU DDBMS the global and the local schema both, record the information regarding the various fragments, about their physical placements and the type of the fragments. In this way, the system locates quickly the required information from the local or global schema. The limits of the fragments are also recorded, which further help in mapping the query to the right limits lying in the area of the fragments. If the local schema does not find a required fragment then the intraquery is routed to the required closest cost effective terminal node through the coordinator. The information is furnished from the catalog to provide transparency. The results are subsequently received by the server and finally the global computations are performed and the results are routed to the requester.

3.3.4 Server Design to Access the Local and Global Schemas

To process queries, the fragments are placed with the proper indexing mechanism depending upon the query attributes and the indexes available. In case, when the high frequency of the intraqueries is available on certain key areas of the fragments then, there is a mechanism introduced in the system which automatically creates those index tables on the specified key values. The indexes are placed in such a way to configure the system with duplicate key values, if found in the database. And if the duplicate key values are present, then the system looks for some more common attributes within the two or more classes, and the server loads the different required indexes simultaneously to acquire the right sequence. The indexes created by the system on its own through the frequency of arrival of certain queries are marked separately and remain transparent to the users. Depending upon the necessity, the fragments are created and if the frequency of access goes down, the fragments are removed automatically from the system. The detailed statistical analysis and maintenance of the fragments is discussed in chapter 6.

If the required attributes are missing from the local schema, the global schema is traced out for the purpose. The attributes are traced for the optimal fragment sizes and the control is handed over to the terminal node consisting the selected fragments, through the global coordinator. These operations are performed by the coordinator through the current catalog, holding the information regarding the current alive systems on the network. Finally, the terminal node is requested to perform the actions through its requester, which performs the tasks as if they are being required by the local terminal node. Server

performs the operations and hands over the results to the requester which sends them back to the query originator terminal node through the coordinator. The whole operations are implemented transparently.

Intelligent local query processing is shown in Fig. 3.4. Whenever a query is required to be processed at any terminal node, it is routed through the requester, which takes it to the server after converting it into the required format of the GURU (local) DDBMS. The server requires the catalog to form the local schema into the memory. The different required attributes are traced out and the other mechanism like indexing scheme etc. is looked into the server for the analysis to be carried out through the local schema. Server, then places the data to its query processor, and sends the directives to the fragment tracer. Which sends commands to the intelligent database controller to give the final shape to the databases with the proper indexes loaded in the memory. Query is processed by placing the required data back to the server. These operations are recorded in the log at the same time and verified. In the last the results are sent to the requester. After getting the results converted into the user's format the results are sent to the required output unit.

The above discussion is purely for the queries based on the local terminal database. If the query is having the entity faults, i.e., the required attributes are missing from the local schema then the server sends the information to the requester for sending the global schema (in case its copy is not available with the local terminal node). The requester places the request for global schema to the local coordinator, which directs the message to the addressed terminal node. That terminal node sends the catalog fragment consisting the global schema to the local coor-

dinator which directs to the sever through the requester. The global schema is loaded in the memory and the required classes are traced. The query is distributed to those terminal nodes through the local coordinator which are carrying the fragments. The resultant fragments are sent to the current terminal node and they are synthesized maintaining the serializability of the intraqueries at the current terminal node. Rest of the operations are performed as were in the previous case of the local database query processing. Finally, the results are directed to the required output unit.

A local area network (LAN) is shown in Fig. 3.4, connecting other terminal nodes with the local coordinator. To process queries in the system, the user does not require to provide any internal details manually for the fragments but they are arranged automatically based on the query structure required by the user. A complete data transparency is provided with a required degree of site autonomy to the users, for which LAN controller plays an important role.

3.4 Approach

With the present object-oriented programming (OOP) philosophy used with database query languages, it is difficult to access the different required attributes unless one adopts the right approach to activate right object's procedures [76]. These procedures if found correct, provide the attributes through the relationships among the different database fragments linked with each other, otherwise a message like, 'Trying to invoke wrong procedure' will be resulted.

The details about the various fragment types are re-

corded in the global schema, which consists of the following fragment types:

1. Horizontal fragments
2. Vertical fragments
3. Mixed fragments
4. Replicated fragments of the above kinds
5. Overlapped fragments

The above listed fragments can reside on any terminal nodes kept geographically apart but connected with a network. The approach helps in maintaining the entire details about the various kinds of the fragments, which can be accessed transparently by the available servers at the various terminal nodes of the computer network. This approach is very helpful in handling the heterogeneous data. This is done by converting the various data types required to be fitted in the local database configuration and hence the converted fragments can be tackled in the same way as if the homogeneous data is brought in. The server helps in maintaining the data by putting it to form the proper relationships with the existing local or global databases.

This has been a practice that the appropriate fragments which are required by a user, are known to the user in advance. In this way, there is no fragment transparency available to the users. The queries of any kind, which are processed on a computer terminal node belonging to an application, need the required data fragment configuration in the memory. The data fragment configuration for the application is loaded in the memory by the server. one does not require to load fragments manually, the query is simply assigned to the system and it becomes the task of the system to load whatever becomes necessary for it to be loaded

in a view to process the assigned query. This task is carried out by the intelligent server of the distributed database management system. First, the server tries to trace the required parameters from the local schema of the terminal node. If an attribute is missing then an attribute fault is generated and the global schema is looked for the attribute. If the attribute is found, then the address and the fragment name is traced through the global schema, which itself could be on some remote terminal. In this way, the server places the request about the query to be processed at the remote terminal through the coordinator. The intraquery could be processed and the derived fragment can be brought to the current place where the global query can be processed.

In this case the knowledgebase is explored which traces the local schema to process the query. There could be a possibility that the query requires more than one fragment, which lie under the different objects and at the current terminal node. Then, the fragments could be joined on some recorded key, by indexing the fragments on the common key value and a semi join operation is applied. The semi join operation together with clustering is performed so that the efficient way in retrieving the data from the fragments is adopted. This method is used for those queries, which are pertaining the high arrival rate and the other is for the queries not having high arrival rate. The last method does not disturb the existing fragments in the configuration or does not create any other fragments. In this way, redundancy is also reduced but on the cost of the processing speed. These types of queries are required to maintain a data fragment configuration in the memory. Which decides the different kinds of fragments in the memory in such a way that they properly maintain relationships, which help in keeping the different tuple pointers

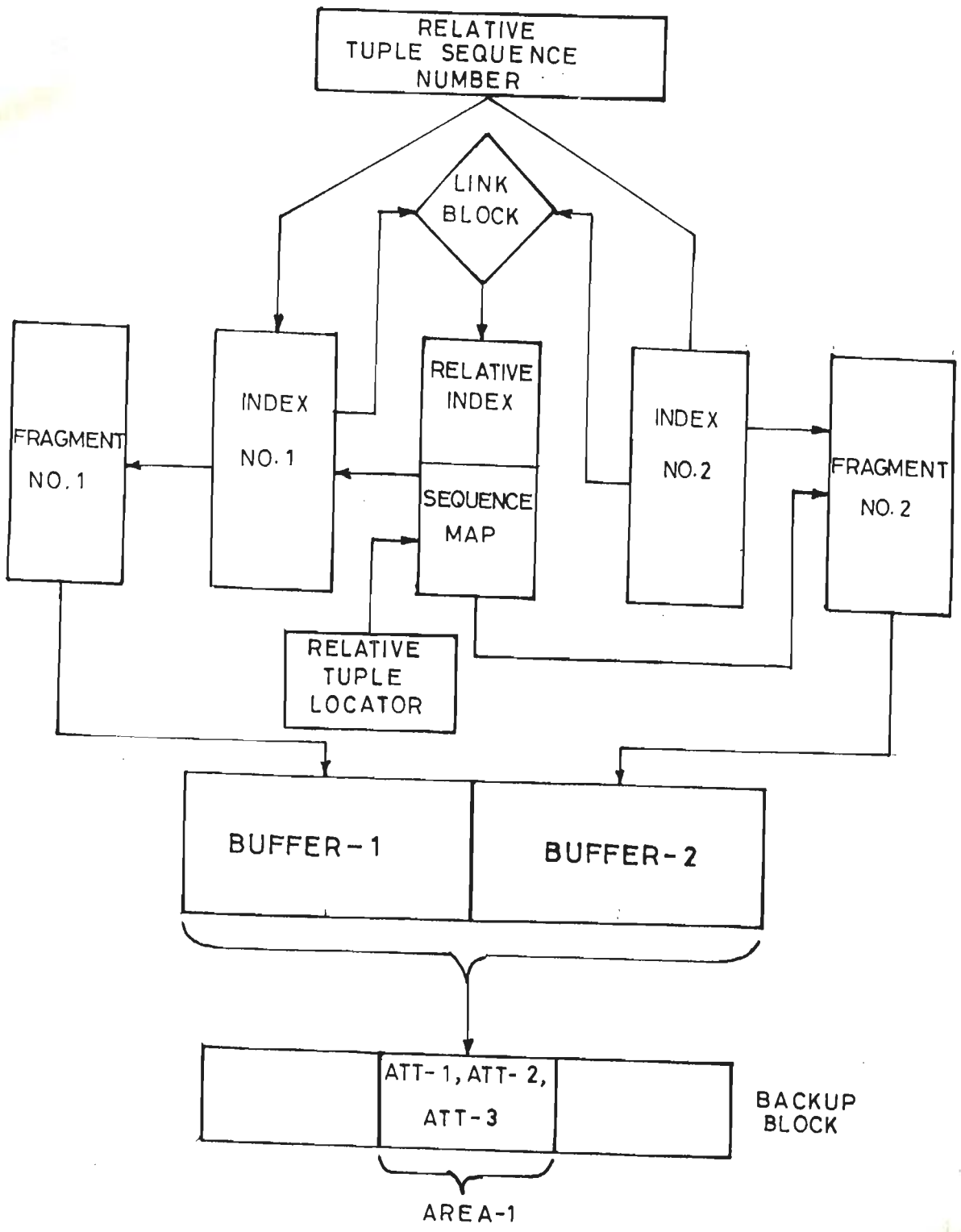


FIG. 3.5 FRAGMENT LINK MECHANISM

to the right places in the fragment configuration, as shown in Fig. 3.5. Similarly, so many pointers start moving in different fragments, such that, one pointer remains with one fragment, and all these fragments are organized on some relation maintaining the required indexes. Hence, the required data can be read or written to the specified fragments.

The different fragments maintain different kind of keys, and also maintain the relationships among them as shown in Fig. 3.5. This approach is very much applicable in maintaining the heterogeneous keys. Some indexes based on certain key values are shown with index, no_1 and index, no_2, which are connected to the fragment, no_1 and fragment, no_2 respectively. The fragments shown are vertical or the mixed type, of the original fragments. Which are based on the key attribute values, for later processing on the key expression. The given two indexes, no_1 and no_2 by linking with the database fragments, no_1 and no_2, provide a relative index to the database fragment, no_2. To process the query based on any key value, the locator finally searches the tuple sequence no. through the sequence map. This directs the database fragment, no_2 and the index, no_1 to trace the right tuples and place them in the buffers. These values are later transferred to the backup memory.

3.4.1 Constraints

The constraints which affect the local processing efficiency are the type of terminal node, the network used, the live terminal nodes carrying the precious information like global schema, various fragments, redundancy, atomicity, serializability, current network average throughput, deadlocks, fragment size, timeouts, site autonomy, heterogeneity of the data, available

index structures, the clustered ready fragments and statistical intelligence available with the server to have forecasting of the type of queries going to be arrived in future etc.

3.5 DESIGN VIEW

The search mechanism used with local and global schemas is illustrated with the help of an example. The available intelligent data in the catalog directs it to configure and load the required configuration of the data fragments, with the index mechanism, so that, the query is processed efficiently. To provide an efficient access, the data tuning is done before the data is accessed. This task is taken up by the server, and whenever a query is processed, it gets registered in the catalog in the appropriate categories, after watching the synthesis part of the query and the corresponding tables. Which record the different kinds of the queries analyzed periodically, considering the various kinds of remote fragments and maintaining the logs. In this way, periodically, the frequency of queries is calculated with respect to the network congestion. If required a few copies can be maintained individually of some fragments, lying at the remote terminal nodes, which can be brought and recorded at the local terminal node and the local schema is updated automatically. Similarly, if network traffic is not high and safe transmission with reachable performance is obtained then, certain replicated fragments are removed by the server and hence updating the local and global schemas accordingly.

At any time if a terminal node is not alive due to some reason, which was processing the global schema, the other dependent terminal nodes place the request to some other registered terminal nodes for the application, following the 2PC and 2PL

communication protocols, to provide the current state of the global schema. Later the local catalog and its global schemas are updated. The changes made during certain time interval are recorded in a log, and the log lets the changes finally made, or removes the changes with some atomic instructions, if commit or abort signals are received respectively.

Using the above discussed methodology, decisions are taken to maintain the required fast access mechanism for the data to be processed. Certain indexes based on some keys are created and by linking those indexes to the basic database, some new clusters are created, and the data is recorded in them. Using this method, the queries, with higher frequencies are processed. But, in certain cases, which are found enough fast, this procedure of getting the local fragments in a different replicated form is not maintained. Its decision is taken considering the another fact of increasing the local redundancy, which obviously could not be increased to the higher values due to mainly two reasons. One is not to let so much of the disk space and the other is the local maintenance cost for every update, which increases proportionately with the replicated copies. Therefore, an interactive approach is suggested, which is found very suitable. Using this approach the different database fragments are connected with the different pointers which make use of the specified index table mechanism hence making the server, retrieve or manipulate the data efficiently.

A local schema is discussed in Fig. 3.3. For example, In this case the query is issued as follows:

List En_no, Name, Marks, Performance for Class = "B.E." of University

AREA NAME	LOGICAL FRAGMENT NAME	AREA CODE	VERSION NO.
RAM-1	K-D-S	57	32
RAM-2	G-H-S	29	27
	⋮		
NIL	NIL	NIL	NIL

INDEX CODES

IC 11	IC 12		IC 18
IC 21	IC 22		IC 20
IC 31	IC 32		IC 38
		⋮	
IC NIL	IC NIL		IC NIL

FIG. 3.6 BACKLOG FRAGMENTS AND INDEX MAINTENANCE

The "of" construct above loads the local schema (as an Object not with strict encapsulation) with the name, Uor_st. Suppose, that the one class in the local schema, University consists of the attributes as En_no, Name, class and Marks with the fragment name, K_D_S, and the other class consists of the attributes as En_no, Name, Class, Performance and Address, with the fragment name, G_H_S. The processing of these fragments is illustrated in Fig. 3.3. The actual internal process is shown in Fig. 3.6, and Fig. 3.5. The fragment names as fragment, no_1 and the fragment, no_2 are used for the logical replicated derived fragments, K_D_S and G_H_S (as shown in Fig. 3.6) respectively. These fragments remain transparent to the users. The working details are explained later. Finally, the query after getting processed, supplies the list of the resultant fragment with the attributes, En_no, Name, Marks, Performance to the screen of the local terminal.

In GURU DDBMS, the maximum 26 fragments (each fragment can have any number of linked horizontal fragments) can be loaded together to form one logical fragment, this limit can be extended as and when necessary as shown in Fig. 3.6. Each fragment can consist of maximum up to 254 attributes and can be associated with 8 index files to give proper sequences to the individual database fragments. Therefore, total 208 files (fragments) can be opened simultaneously. The indexes, which are active can be changed as per the requirements of the system as shown. Any time, the scheme activated inside the computer active memory can be changed with a new assignment over the database. This could be done with the help of a method known as backtracking. Where every possibility is explored from the available set of knowledgebase to find that the query is satisfactorily replied with the available scheme. Therefore, the distributed database management

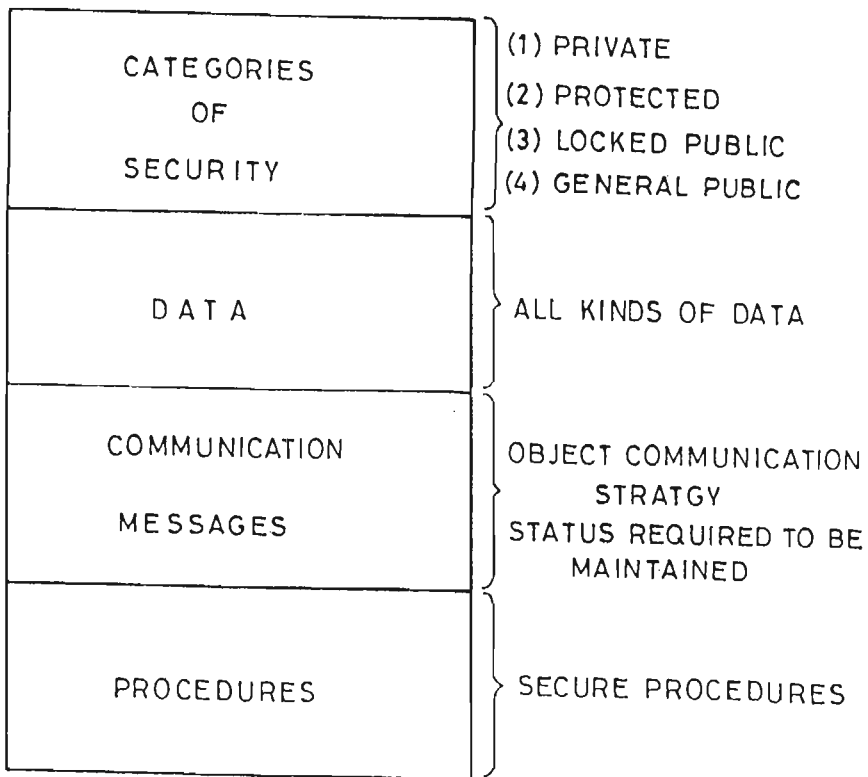


FIG. 3.7 GENERAL OBJECT

system uses its own supervisor to find that, if any time a fragment does not meet the required condition, then, the alternative condition will be required to be checked. This is done to have another view of the active memory scheme. Hence, altering the present linked structure and getting a new structure set up in the memory by placing a new index order. This method provides the good solution with reasonable access efficiency.

In Fig. 3.7, an encapsulation is illustrated of the data and procedures. The communication is done with the messages, which consist of certain codes to get the identity and required features, which are send to get the data from the object (capsule).

3.6 FRAGMENT DESIGN

3.6.1 Horizontal Fragment Design

These fragments are based on certain key expression, and the all tuples with all attributes present in the class or the object are divided with the tuples satisfying the key expression. There is a condition that, there should be at least one tuple in the fragment which satisfies the key expression. These tuples forming a new fragment can now be placed any where throughout the network in any terminal node. This fragmentation is done on the requirement of the database on some of the sites where the maintenance of the fragment would be done to increase the overall access efficiency and the local autonomy. These horizontal fragments are also divided into two kinds: primary fragments and the derived fragments. Primary fragments are those, which are of nonoverlapping type i.e., the two primary fragments cannot overlap with each other. The tuples belonging to them are two disjoint sets. Whereas, the derived fragments can overlap.

That means that, some tuples can belong to two or more fragments. Derived fragments are also designed to provide the fast access over the database. The overlapping of the fragments is designed to keep in mind that the common tuples of the two or more fragments should have mostly the read operations to maintain the good access efficiency. But, at the same time if the update operations are performed over the replicated tuples then, it may lower down the overall efficiency. Since, the update operations to the replicated tuples belonging to the other fragments and sites are also required to be performed at the same time. Therefore, the network is accessed for the additional operations.

The size of the fragments is chosen on the requirements of the various sites. If some query expressions require certain database to be accessed more frequently, at some of the sites then those attributes (the all fragment attributes for the horizontal fragmentation) required with that query, to be processed would be fragmented with the required fragment qualifiers and kept on the site having the highest required frequency. This method is known as the best fit method. The selection of the site is done with the help of cost calculations. If the cost of some other site is lower for those query operations then this fragment could be allocated to that site.

3.6.2 Vertical Fragment Design

Vertical fragments are designed on the requirements of some of the sites, needing some attributes of one or more classes. There could be some common attributes among some vertical fragments, which would be considered for the all tuples present in the fragment. These common attributes are again kept, keeping in mind that these attributes would mostly be used for

the read operations to be performed. Otherwise, the similar things would matter as in the case of horizontal derived fragments discussed above. The fragmentation criterion is chosen on some of the attributes as required by the various sites.

3.6.3 Mixed Fragment Design

The mixed fragments are designed by combining the two approaches of the horizontal and vertical fragmentation schemes. These fragments are also required by the various sites hence their placements are done in the similar manner, as was done with the previous two cases of the horizontal and vertical fragment allocations.

All kinds of fragments can be created with GURU DDBMS, which are presented in the earlier discussions. The classes consisting of various kinds of attributes are illustrated in one layer and are connected with sibling pointers as shown in Fig. 3.2. Therefore, so many classes can be maintained in global and local schemas, where each class can consist of one or more attributes. Each and every attribute belonging to some class is connected with some child nodes. The vertical nodes connected with the horizontal attribute nodes represent the horizontal fragments. These horizontal fragments can be of any number and size, and all horizontal fragments are connected vertically. These fragments have certain replicated copies which are shown with the horizontal RC (Replication Copy) of the horizontal fragments. The other kind of the fragments are the overlapped fragments and can be subsequently some subsets of the existing horizontal fragments. Therefore, they are marked with a special flag kept in the replicated fragment node, which denotes that the fragment is the replicated fragment but, it is a subset of the

primary horizontal fragment or is a derived fragment of the primary horizontal fragment.

Each and every fragment node consist of the following information:

1. Terminal code
2. Owner's directory
3. Class (Object) name (Logical name)
4. Size field (flag field)
5. Total number of tuples (Cardinality)
6. Key expression
7. Time stamp (modification time)
8. Catalog address (for reference)

The above first three points, locate the right class and its replicated copies through the links. The fourth point is kept specially for overlapped fragment adjustments, considering it as the partial fragment of the derived horizontal fragment, or it is the replicated copy as a whole. The above fifth point tells the total tuples for size optimization criterion selected by the server. Sixth point tells the fragmentation criterion selected. Seventh point tells about the latest updates made in the fragment and the last one is further to refer about the other details recorded in the catalog like the maintained indexes etc.

3.6.4 Fragment Allocation

The fragment allocation criterion is basically decided by the cost factor as discussed in the fragment design section above. The basic criterion for the calculation of cost of the fragments is depending upon its requirements over some sites in

an application. The requirements of the fragment could be of the two kinds, one is only for reading that fragment, the other is to update that. Therefore, the total local references to that fragment would be the sum of the products of the frequency of query arrival for the fragment on the site, with the total individual fragment references for that application. This is multiplied by the cost, gives the total fragment reference cost. There are some other factors like the type of the terminal node and the memory speed etc., which are also considered for the cost calculation criterion.

The system is designed to have the fragments allocated in such a way, that the fragments can be accessed quickly and the fragments should also take the minimum memory space. Therefore, the fragments should also be provided with the minimum local replication. Further, the fragments should be maintained in an order to have the proper relationships among the various attributes through the classes. Therefore, there is a technique called common clustering, which is used to provide fast backup access[169]. All related domains are maintained on the given key orders, and are placed together forming a common physical record structure on some specified keys. This is done after watching the characteristics from the log. Which results in quicker replies to the queries, since all attributes are common clustered, as if all belong to the one class and are also recorded at the same place. The other kind of the fragments, which are not tuned as in the earlier case can be placed ordinarily. If the common clustered scheme is not used for processing queries, so many pointers are formed which move synchronously to the corresponding individual fragments, hence taking more time.

3.6.5 Versioning

To establish the perfect relations among the various objects in the local schema, certain time and date stamps are maintained on each and every fragment brought from the other node [198]. Not only this but, a version number decided by the server depending upon the arrival of the new fragments is also marked in the local schema. Which helps in maintaining the various versions of the fragments as shown in Fig. 3.6. These versions help in the following ways:

1. For concurrency control, which maintains the users accessing the information simultaneously, and keeping the data fragments protected from the simultaneous changes made by others, who had a control over that fragment.

2. For recovery of the damaged versions due to certain crashes occurred in the system or transaction failures.

3. To enhance the performance of the distributed system. More are the copies on the network, better would be the locality, hence the response time would be short.

4. For implementing the databases without updates. This is done to maintain the separate copies of all the modifications done in the database. This helps in maintaining the various versions of the database, which may be required in future.

3.7 QUERY MAINTENANCE

3.7.1 Dependency Check

The query expression is expressed and elaborated in

such a way to load the required indexes in the memory which are used to access the required data from the database efficiently. These indexes are loaded following the intelligent scheme. A query is analyzed for the comparison operators (>, <, >=, <=, = etc.) used with queries, which could be seen for the dependency checks as bellow:

$$\text{SNO} > 5$$

The attribute, SNO is compared with a constant, therefore, the only index on the key SNO is loaded which is used to check the required condition in the database. If another expression, $\text{SNO} > 5 + X$ is expressed, which comprises of an identifier, X, not as an attribute in the database, therefore, it does not lead to complication. But, if X is an attribute in the database, this keeps changing its value for every fresh tuple, which would require to deload the other index, SNO from the main memory. The other expression for the similar discussion could be seen as below:

$$\text{SNO} * \text{EN_NO} > 5 + X$$

The SNO and EN_NO both are the attributes, both tend to have different values for the tuples. Therefore, in such an expression, it is difficult to establish any index in the memory which provides the quick access with least searching the tuples. A procedure used to watch such activities closely in the database maintains a separate activity chart in the relevant objects. This chart with the help of the catalog and data dictionary in the system maintains the entire information related to the dependency of the different attributes in the database. If an index for such a database fragment is prepared then at the same time after

finding from the dependency chart (through the programming in GSQL) a separate dependent index is created, which records the relative orders in the dependent attributes after finding the characteristic pattern from the memory. The information regarding the characteristic pattern is obtained by watching the type of changes in the data belonging to some attribute over the other in the normal index behavior. This information is analyzed from a knowledgebase and knowledge is updated for the different patterns of the data.

Some arithmetic operators are used between two attributes, operated for the generation of characteristics of the data in the database. The process could be seen for the basic arithmetic operators as for the expression, $SNO - EN_NO$, assuming the both attributes numeric. If the data pattern for the corresponding tuples is increasing (ascending order), it is concluded that the two attributes have correspondence in their values. This makes the index with one of the attributes working successfully.

The different tuples are getting their values changed with respect to time in the database. Therefore, a continuous monitoring regarding the changes in values is done and the drift from the original pattern of the data is checked. Some tuples may have the different sequence than the sequence observed among others. Therefore, these tuples are separated out in the index. If the drift is among a few tuples, they are marked in the relative index created separately for the purpose. The separate index provides the negative values in the original index. This indication helps in not scanning those tuples specially marked. These would be scanned subsequently either sequentially or using a separate index made for them. The procedure depends on the number of tuples. If the tuples are very few, they would be

scanned sequentially otherwise, a separate index depending upon the previous guide lines would be prepared. The attributes are tried accordingly for the different other arithmetic operators. For these operators, the different patterns are observed for the data in the database, used for the later processing. The different patterns are recorded in the catalogs and the object extended structures, which are activated on the later requirements.

3.7.2 Index Organization

The indexes can be created based on the different keys. A key can consist of any number of attributes partially or fully. The object requiring to mention a key is depending upon the defined key with the current object and the other linked objects and classes. The environment existing for the indexes being created may not be present in future when a query needs to be processed. Therefore, the indexes are defined in such a way that they also work in the dynamically changing environments. The index created consists of the pointers to the different tuples and the value of the keys required for the individual tuples. This method facilitates in providing the right sequence to the different tuples in the dynamically changing object environments, which could affect the different tuple sequences due to the changing environmental scenario. The indexes consist of the following structure:

1. Fragment Name
2. Fragment Qualification
3. Index Formula
4. Order
5. Pointers
6. Key Values

To create an index for a database fragment lying with an object, the following instructions are used:

```
Index on <key expression> to {<index name>| <tag name> [of  
<multiple group index file>]} [unique] [descending]
```

The *key expression* above specifies the value of the key with the specified objects on which index is to be prepared on the specified *index name* or the specified *tag name* of the *multiple group index file* carrying the multiple indexes of the different (remote) database fragments. The specified key can have the attributes present partially or fully from the specified fragments declared in the current object or any other inherited or related objects. The size of the index key should not increase beyond 950 characters. The key can consist of any kind of the data defined as the passive objects. The *unique* clause above specifies that there should not be any duplicate key values. If the duplicates are found then a message is displayed to the user and the index is not created. The order of the index created above is ascending otherwise if mentioned as *descending* separately. The following instructions are also used to create indexes:

```
create [unique] index <index name> on <database fragment name>  
<column name> [{asc | desc}] [, <column name> [{asc | desc}...]];
```

The *column name* is the name of the attribute defined in the database fragment with *ascending* or *descending* order. The other things are applicable as were in the last instruction.

Once an index is created, the index can be loaded

externally by the programmer and the following are the commands used to load the index along with the fragment to reply the required kind of queries:

```
use {fragment name | ? [in <work area>]} [index <index names>]
[order [tag] <index name> [of <multiple group index file>]]
[alias <alias name>]
[exclusive]
[noupdate]
[again]
```

The *fragment name* above is the name of the database fragment, which would be loaded in the current work area or in the specified work area with *in* clause. If the name of the current fragment is required then a question mark (?) is used; if the no database fragment lies in the current work area then the all database fragments related to the current user are displayed from the catalog. Which displays the name of the database and the indexes loaded in the current work area. The *index names* specifies the list of the indexes to be loaded with the current database fragment. The sequence of the indexes starts from the primary index to the next relative priority secondary indexes. With one specified fragment the maximum eight indexes can be loaded simultaneously to provide a sequence to the tuples. This limit can also be extended if required, by allocating more space for the indexes. With a *multiple group index file* the different indexes can be loaded and maintained automatically as and when required through the GSQL instructions. The *tag* names specify the tags, monitored in the *multiple group index file*. Their orders are also maintained in the current working *multiple group index file*.

The *order* clause names an index or a tag from a *multiple group index file* as the controlling index. The optional *of* clause could be used to identify the name of the *multiple group index file* that contains the selected *index tag*; this might be necessary to avoid ambiguity when an open *index file* and an *index tag* have the same name.

In the optional *alias* clause, an alternate name can be provided that identifies the database fragment while it is open. If the alias name is omitted, it takes the fragment name as the alias or the prior assigned name for the work area could be considered. The alias name could be given in maximum eight characters; the first character must be an alphabet or an underscore character (*_*); the other characters may be included as the alphabets or the numeric digits.

If the optional *exclusive* clause is provided then the database fragment is only accessible to the owner of the database or only to the current user. A database fragment opened for *exclusive* use cannot be shared by the other users on the network. The *noupdate* database fragment is opened for only *read* operations and not for updating.

The maximum 26 database fragments are opened simultaneously on the network for one user. The *again* clause makes the same database fragment open for more than one work area, for some special purpose operations to be performed among the remote objects.

The instruction that defines the different *work areas* which could be used by the user for placing the different database fragments on the network is following:

```
select alias <work area name>
```

The *select* clause selects the different work area for a new database fragment to be made ready for processing. *Alias* specifies above the work area name as discussed earlier. The other *select* clauses are used with the query launching instructions, which would be discussed subsequently.

The following instruction rebuilds the all indexes or the tags from the *multiple group index file* in the current active work area:

```
reindex
```

The process of rebuilding the indexes in the backup memory is to update the indexes with time. The order of the indexes could be altered by using an instruction, which changes the primary index; the newly activated secondary index becomes the primary index and the indexes are rotated in the order selected towards left, starting from the new primary index. The following instruction performs this operation:

```
set order to [{identifier | numeric literal | [tag] <tag name>
[of <multiple group index file>] | <index name>}]
```

The value of the identifier can vary from 1 to N (where N is the number of maximum indexes used in current work area, which can go maximum up to 8) for the current set index mechanism, which could be enhanced for the later requirements. The numeric literal's value also lies in the above range. The optional *tag* clause includes the tag name of the index from the

multiple grouped index file. The optional *index name* includes the name of the index activated for the current use. If no clause is found after *set order to*, the all active indexes are deactivated; this works in the same way as the *set order to 0* works.

The instruction discussed below directly searches the indexes and no further search is required.

seek {expression | literal}

The expression consists of a value after getting it resolved which would be searched from the current indexes. The literal's value can be any thing defined as a passive object. The other instruction listed below searches a numeric value only:

find <numeric number>

It works in the same way as the *seek* works but it searches only *numeric numbers*.

The following instruction takes the pointer to the next tuple through the current active index.

next

The other instructions used to manipulate the indexes with different applications involved directly or indirectly are provided in *Appendix - B*.

3.7.3 Query Transformation

Optimal equivalent query transformation to the avail-

able fragments is the required objective to process query efficiently. With available structure of the local and the global schema, the distribution of intraqueries in the canonical form is the most desirable feature. To transform the queries, the relational algebra with some set of rules is used. These rules govern the various equivalent operations for the unary and binary operators. The equivalent and related operators are recorded to form the required query transformation. When the query is requested to be processed the unary and binary operators are so operated to form the query structure from the class or the object structures (for the attributes) available preferably locally or otherwise, based on the global schema.

Intelligent query processing is the another approach which fetches the all attributes or identifiers around the unary and the binary operators and put them into the knowledgebase. These are further synthesized based on certain operator rules which further govern the query tree and the query comes out from them as the transformed query.

The optimal selection of query processing strategy involves:

1. The query needs materialization for its efficient processing, but with the available fragment structure, it is difficult.

2. The execution order of the query is important for the query to go through the minimum cardinality of the fragments.

3. The selection of the method of execution of the individual procedures is important. The unary and binary

operators and the required efficient way of their grouping play an important role.

The basic objective in query optimization is the performance criterion of the overall system with optimal cost of processing. The system must perform the operations in the way so that the fast execution of the different intraqueries could be performed. The basic four parameters are considered for the performance evaluation of the overall global system, which are: the use of local I/O, the use of CPU, the use of the network and the degree of heterogeneity with required links on the network. The detailed discussions about the issue of complexity and performance are considered in chapter 6.

3.7.4 Backlog handling

Backlogs are to record the information of various intermediate operations during the query transformation stage, index manipulation stage, data access and the manipulation stages etc. [90]. These backlogs can be maintained preferably in the cache memory to provide fast access to the relevant operations, as shown in Fig. 3.6, with the recorded details of some fragments, which would be accessed for regular and fast access. These details are recorded in the backup for safe operations to be performed. The updates to these logs can be done in two modes of operations, which are quick and lazy. Quick mode of operation is used for immediate operations to be recorded in the backlog. But, the lazy operation is beneficial from the view point of overall speed. The data goes in a lazy queue, which gets updated when system has relatively free time.

A backlog, B_R , for a relation, R , is a relation that

contains the complete history of change requests to relation R.

3.7.5 Log Management

The log is given importance to record the all transaction's Ready states, the Commit states and the Abort states. Any time when a transaction is started, the name of the transaction with the data is recorded. Since, the transactions are performed to process queries, the individual queries are broken in the intraqueries and they are processed over several terminal nodes of the network, which carry the related application. Any time, any failures on some terminal nodes may require the all other intraqueries previously operated to be undone and require to explore the other possibilities of the query, which would be broken up (transformed) subsequently for the new values of the intraqueries. To undo these things, the log stores the all values into it. The previous values would be taken from the log and they would be stored to the places where they were before this processing of the last ready state (the last Checkpoint). In this way, some of the intraqueries might get the same shape for some other possibility of the query, which had faced the failure on some terminal node previously. In this case, the previous operations have to be applied through the log to redo the last operations which were done previously.

Log maintains the complete past history of transactions, which were dumped on tape. So that in future, if any past transactions are required to be used then, they could be fetched back through the log dump. The log approach is also helpful in retrieving the past data if some hazards affect the volatile memory of a terminal node.

Log is also helping in maintaining the serializability of the concurrent operations. The atomicity of the operations is maintained using the log. The all kinds of transactions performed among the different terminal nodes communicated to the local terminal node are recorded in the local log. If an operation is committed then this operation is seen through the log and the later operations can go in synchronization. Thus, maintaining the serializability of the concurrent operations. The log record contains:

1. The transaction identifier
2. The record identifier
3. The type of action being performed
4. The old record value which is being replaced
5. The new record value to be put into
6. The pointer to the previous value of record for the recovery

3.7.6 Server Management

Basically, servers are implemented to perform, the database access and computations for the database servers. have a close coordination with requesters. Requesters have the proper interface with local coordinator and act as interface to it, which performs field validation, convert data to the required data format based on the format of the current database management system and controls the results given by the server for communication to the terminal user.

To access and manipulate the data, the proper fragments are loaded in the memory which form some standard configuration which helps in fast access to the data. Therefore, servers are

ATTRIBUTE LOGICAL NAME-1	FRAGMENT LOGICAL NAME- 1	ATTRIBUTE LOGICAL NAME-2	FRAGMENT LOGICAL NAME-2	RELATION	LOGICAL INDEX NAME
S- R-NAME	RAHIM-1	S- R-NAME	RAHIM-2	S-R-NAME	R- D- S

FIG. 3.8 CATALOG RECORD FRAGMENT

implemented to study the situation and do the needful and maintain efficiency, and autonomy as per the requirements. To achieve this, certain in built intelligence is necessary which helps it to acquire information from the catalog, the data dictionary and the local log. As per the recorded information, controls are generated to perform the transfer of the required fragments from remote terminal node to the current terminal node. The replicated fragments could also be maintained at the current terminal node, which would not be needed at all or would be required with lower access frequency. Therefore, those fragments are transferred to some other terminal nodes as per their requirements, or they may be deleted from the current terminal nodes.

3.7.7 Catalog Management

Catalogs record the information about the database and its access procedures. Catalog itself is a fragment which is required to be distributed on the need, as shown in Fig. 3.8 with some example. Therefore, these fragments are required to be allocated efficiently. The catalogs are of three types: centralized catalogs, fully replicated catalogs and local catalogs. Centralized catalogs are used normally in the case of centralized database management systems but are also used with distributed database management systems having no site autonomy. Fully-replicated catalogs are recorded on all sites. The modifications made on some of the sites, are also required to be made on all other related sites on the network. The local catalogs are kept only at the local sites where the data is handled locally. Several intermediate ways are there, one of them is partially replicated catalog which is used in GURU DDBMS. In this case, the replicated copies of the catalog are not lying with all the terminal

nodes but they lie with a few of them. Therefore, updates will not be a problem.

In another way, catalogs are basically of two types local and global catalogs. They consist of the following information:

1. Local schema details: It consist of the name of the fragment and the attributes lying with them maintaining the relationships.

2. Fragment details: Since, the separate classification is done with each attribute, the details of the qualification of the horizontal fragments with replication or overlapping criterion are recorded.

3. Allocation description: It provides the details about how the fragments are mapped to physical images.

4. Access procedure description: The details of access procedures with the types of index files locally available are recorded. This is different with different individual machines storing the local database.

5. Profile description: This includes the details of the frequency of access of the database and the time stamps of the access left for statistical analysis of the network response time.

6. Privacy and security description: Various attribute level access to be provided to various level of the users for which their authorization is watched for read, update, insert, delete and move operations performed on the database.

3.7.8 Data Dictionary Management

Data dictionary records the information about the

synonyms in the database with their access procedures and so many other different kinds of the data, which are needed to enhance the knowledge and performance of the system. Data dictionary is referred by translator/mapper and the server kept in the system to work in the heterogeneous environment in GURU DDBMS. The data and the knowledge are recorded in the data dictionary at appropriate places for which records are maintained in the catalog, and also in the global and local schemas. The data dictionary plays a very important role for any kinds of updates required in the system. Any new incorporations on the LAN, which operate on some different kinds of the DBMSs, need some information to be fed in the system, to update its knowledge. Further, any improvements made on the current system also need regular updates on the corresponding systems. The contents of the data dictionary regarding the synonyms for the example of Fig. 3.1 and Fig. 3.7 are shown in Fig. 3.5.

The organization of the different types of the indexes during processing of a query is provided bellow:

Scenario

Here, an example with a university is considered. A university is consisting of a Central Administrative Block, a Hospital, Student Hostels, Staff Hostels, a Central Library, a Central Computer Centre, and Teaching Departments etc. The different students getting admission to the various departments are enrolled at various places in the university. One student seeking admission to a course B.E. in Computer Engineering is enrolled with the following sections: Department of Computer Engineering, a under graduate hostel, a hostel mess, Central Library, the Central Computer Centre, the Hospital etc. To

evaluate the overall performance of the student, the different sections of the university are considered. The actual time the student has spent at the various places could be calculated with various aspects such as the academic achievements, the general fitness etc. This helps in the judgment of a right kind of a student with the abilities in the various required arenas for the final selection.

Problem Implementation

The problem is implemented by recording the various activities of the students in the computer system by providing a master card to each and every student; who gets the permission to enter on placing the master card in the computer's master card reader. This method maintains the student's record on the computer network. The student's name and the activities are available in the various database fragments, which could subsequently be accessed through a wide area network to the common global student's database. Therefore, the different agencies requiring to recruit the right kind of persons, can send their willingness to appointment the students through the network. Further, the agencies can finance the right students during their study schedule.

The different activities of the students can be encapsulated in different sections. Each section consists of the record of all students in an object. The different sections are forming the different objects identified by the unique object identifiers. These objects are related in nature. The following instructions prepare a logical view of the problem:

```
procedure main  
define structure univ_dept
```

```

datafiles students
fields en_no, name, address, class, spec, year, marks
memory stud, s_add, rec_no
procedures univ_dept, ~univ_dept
functions marks, position
devices printer, keyboard, vdu;

define structure univ_hosp
datafiles patients
fields en_no, name, address, class, b_g, disease, health
memory std, s_add, h_c
procedure univ_hosp, ~univ_hosp
functions disease, cronic, history;

define structure univ_lib
datafiles ug_st, pg_st, r_s, staff, others
fields en_no, name, address, class, journals, books, date,
        time_ent, time_left, overdue, excep, grade
memory std, st_add, efficiency, overdue, others
procedures univ_lib, ~univ_lib
functions overdues, exceptions, grade
device printer;

define structure univ_cao
include structure univ_dept, univ_hosp, univ_lib as union
datafiles ug_rec, pg_rec, r_s
fields en_no, name, address, class, performance, fellowships,
        scholarships, excep, awards, punish
memory st, award
procedures univ_cao, ~univ_cao
functions exceptions, awards, punishments
device printer, vdu, keyboard;

```

```
create object o_dept with structures univ_dept;
create object o_hosp with structures univ_hosp;
create object o_lib with structures univ_lib;
create object o_cao with structures univ_cao;
```

```
x=o_dept->marks()
y=o_dept->position()
x1=o_hosp->disease()
y1=o_hosp->cronic()
z1=o_hosp->health()
x2=o_lib->overdues()
y2=o_lib->exceptions()
z2=o_lib->grade()
x3=o_cao->exceptions()
y3=o_cao->awards()
z3=o_cao->punishments()
```

```
procedure univ_dept of o_dept
select alias dept
use students
index on name to named
index on address to addressd
use students index named, addressd
```

```
procedure univ_hosp of o_hosp
select alias hosp
use patients
index on name to nameh
index on address to addressh
use patients index nameh, addressh
```

```
procedure univ_lib of o_lib
select alias lib
use ug_st
index on name to namel
index on address to addressl
use ug_st index namel, addressl
```

```
procedure univ_cao of o_cao
set echo off
set talk off
select alias cao
use ugc_rec
index on name to namec
index on address to addressc
use ugc_rec index namec, addressc
set relation to name into dept
set relation to name into hosp
set relation to name into lib
```

```
procedure ~univ_dept of o_dept
select alias dept
use
```

```
procedure ~univ_hosp of o_hosp
select alias hosp
use
```

```
procedure ~univ_lib of o_lib
select alias lib
use
```

```
procedure ~univ_cao of o_cao
```

```
select alias cao  
use
```

```
function marks of o_dept  
list name, adress, marks  
return .t.
```

```
function position of o_dept  
select alias dept  
index on marks to marksd  
use students index marksd  
list name, address, recno()  
select alias cao  
return .t.
```

```
function disease of o_hosp  
select alias hosp  
list name, address, disease  
return .t.
```

```
function cronic of o_hosp  
list name, address for disease $ "cronic"  
return .t.
```

```
function health of o_hosp  
list name, address, health  
return .t.
```

```
function overdues of o_lib  
list name, address for overdue  
return .t.
```


function exceptions of o_lib
list name, address, excepep
return .t.

function grade of o_lib
list name, address, excepep
return .t.

function exceptions of o_cao
list name, address, excepep
return .t.

function awards of o_cao
list name, address, awards
return .t.

function punishments of o_cao
list name, address, punish
return .t.

3.8 CONCLUSION

The view of query automation has been presented with automatic generation of indexes and loading them for the launched query. When the schema is defined, the required order of the data in the fragments is prescribed. But, the queries never stick to the prescribed order, this is due to a wide range of probability in supplying the query structures. Part of this is met by the sequences prescribed in the objects regarding the attributes from the instruction *define structure* and later it is supported by dynamically created indexes when the query frequency exceeds a limit. This method is very helpful in avoiding the unnecessary

indexes hence reducing the database overhead. The dynamically created indexes are marked in the catalog and are controlled by the system. Further, these indexes remain transparent in the system and do not display in the directory seen by the user. The user's view of the data remains protected.

The suggested index structure mainly carries two things the pointers and the index values therefore, the large main memory size is not required and the index values do not remain in main memory for a long time. In the specific cases, the values stay in memory, when certain instructions need them. Based on the query structure the different indexes are resolved and by making use of the knowledgebase, indexes are loaded with the database. The search procedure is loaded in the order of indexes hence reducing the query response time. The composite indexes are designed and implemented on the requirement of the available pattern for the fragments. These indexes are watched strictly; because they involve relatively more dependence. The composite keys play an important role in providing better integrity in the system.

The neat description of indexes helps in locating the right indexes with appropriate index expression, attributes and fragment qualifiers etc. Therefore, users can design their programming environment better useful and efficient. The different indexes on remote keys can be linked and enhance the consistency check and future dependability over the system. The system works efficiently in the heterogeneous environment with the suggested schema structure and makes the database fully invertable with the provided intelligent scheme.

The approach of intelligent server design to handle

various databases efficiently is very effective. Which provides a complete transparency of various details about fragments and attributes and their placements. A lot of burden of remembering the techniques of efficient data access by the programmer and the programming efforts are reduced and system itself takes the guarantee of efficient, effective and transparent operations required by the programmer.

A system sample program to handle indexes is given in Appendix - E

SCHEMA DESIGN

4.1 INTRODUCTION

To support the wide range of application domains, different software are being provided by the various software companies. The software consist the different kinds of distributed database management systems. Since, the applications belong to the different domains, the access over the data could be provided through a database management system, which could be integrated to the heterogeneous environment to share the diversified data types. To record the data and information in the heterogeneous environment, the available systems use the poor techniques of data representation which lead to poor schema design. The lack of intelligence in the system creates the problem of poor data maintenance, which requires a great attention to support the heterogeneous environment.

A good database design should provide the quick and reliable accessibility of the database available at the remote terminal nodes with proper transparency, serializability, heterogeneity, site autonomy, and the perfect atomicity. Several researchers have suggested the different data models to handle the databases [49],[50]. The relational database model is more widely used in the distributed databases [141]. The relatively new approach of object-oriented database model is becoming popular in the fast changing environment. This model needs to be improved in many ways to provide a better schema design, which could be used in the existing heterogeneous database environments. The intelligent schema design is an effort in the same direction.

This has been found that the object-oriented scheme is successful in giving the least data description, that is the unnecessary data description is avoided. This is particularly very much useful for object-oriented designs of the schemas [25],[127]. The object-oriented scheme provides the three main features, namely data abstraction, structural mechanism and behavioral mechanism. The data abstraction provides the abstracted data to be scanned required by the query access procedures, the structural mechanism establishes the basic relationships among the different class and object structures comprising the different attributes, and lastly, the behavioral mechanism provides the behavior of the data in the database. The behavioral formalism is the unique feature, which provides the inheritance of the different classes and the objects with their properties. The existing object-oriented databases do not provide proper site autonomy due to the complexity in the system.

The common existing ways to maintain the relationships are the maintenance of the flat file structures, which are related in nature [49],[50]. The flat files are kept, maintaining the relations among them. The system maintaining the flat files (also known as database fragments) distributed at different terminal nodes is known as distributed database management system [141]. The following sections discuss the design of heterogeneous schema:

4.2 APPROACH

The complete logical structure design of the database essentially requires the clear declarations of the entities and attributes with the existing relationships among themselves. This

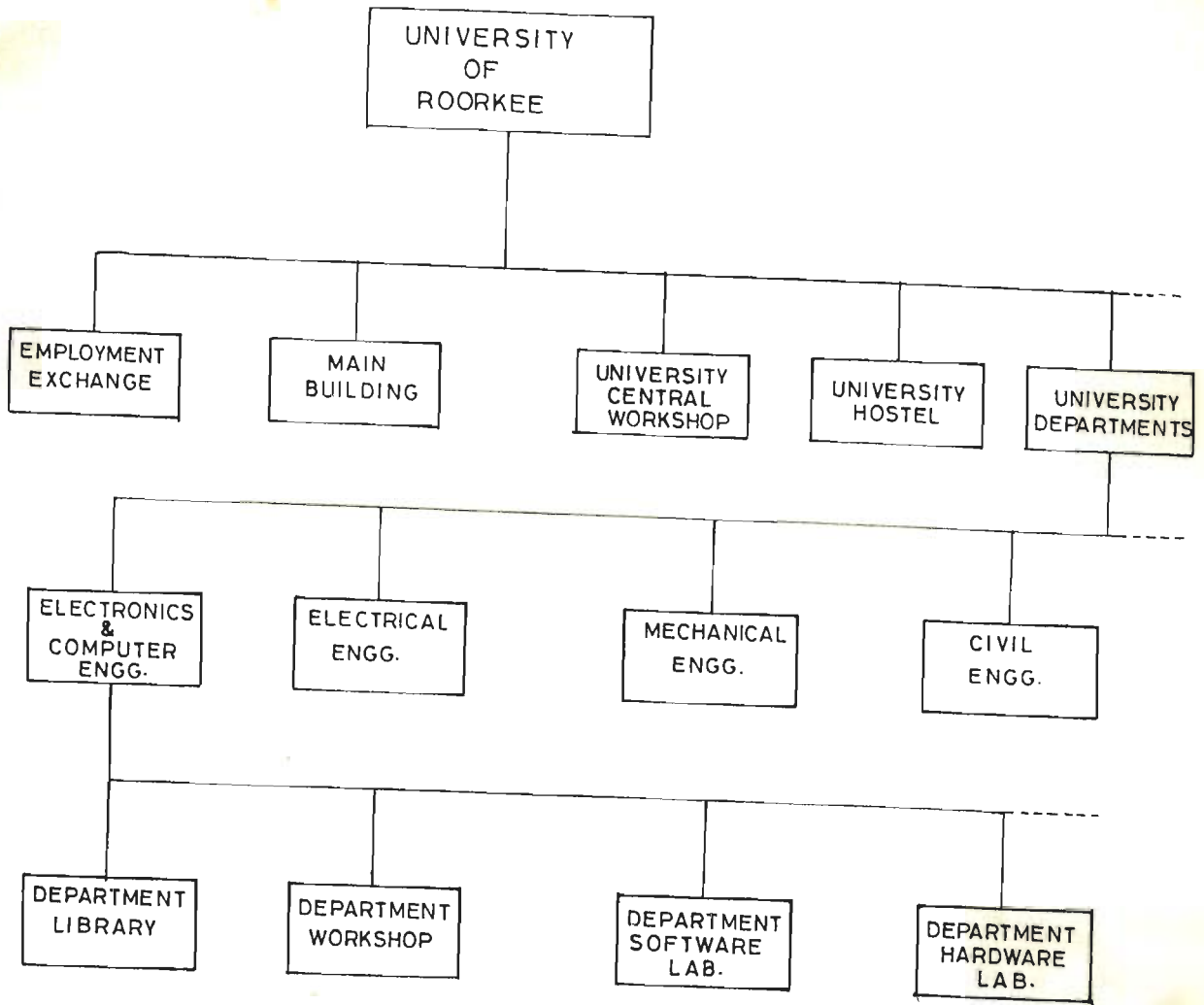


FIG. 4.1 SCHEMATIC HIERARCHICAL STRUCTURE OF UNIVERSITY

is possible if they are connected to provide a tree like structure, where the relations are so kept that one - to - many and one - to - one relationships can only exist, which forms a hierarchical model of the database. The other possibility is that, the relationships are complex like many - to - many relationships. This needs to restabilize the relationships in one to many format, to provide a format of hierarchical structure, otherwise, the resultant structure forms a network structure as illustrated in Fig. 4.1. This kind of structure is difficult to handle by most of the existing database management systems.

To define a schema, one needs to clearly define the attributes, entities and the database fragments. Further, the relationships and the access methods are also required to be indicated among all the attributes, entities and the database fragments. The object-oriented approach makes the integration with proper consistency among all attributes, entities and fragments, defined as objects. Each and every defined object is identified with an OID (Object's Identifier) and has the local object memory variables, attributes, entities, fragments and the various procedures and functions, known as methods in Object-Oriented Programming (OOP). The memory variables belonging to the objects are accessed by the local methods of the objects. Further, they may also be accessed by the outside procedures depending upon the access rights and the access environments declared in the objects regarding them. The memory variables, which are not allowed to be accessed by the outside environment, provide the image of strict encapsulation. GSQL provides the feature of accessing the memory and other variables defined in objects, through some outside procedures, which make the system with enhanced strength.

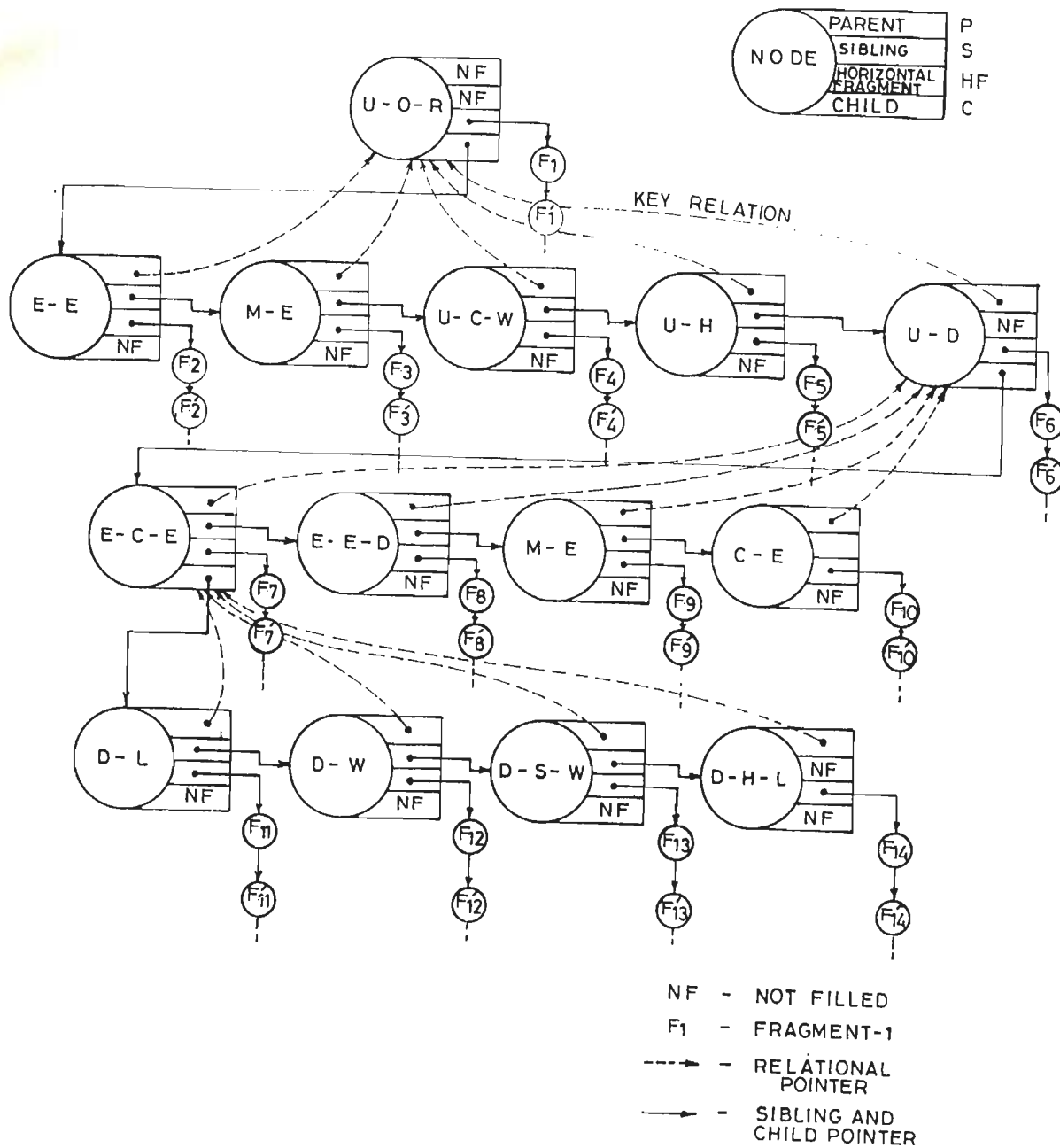
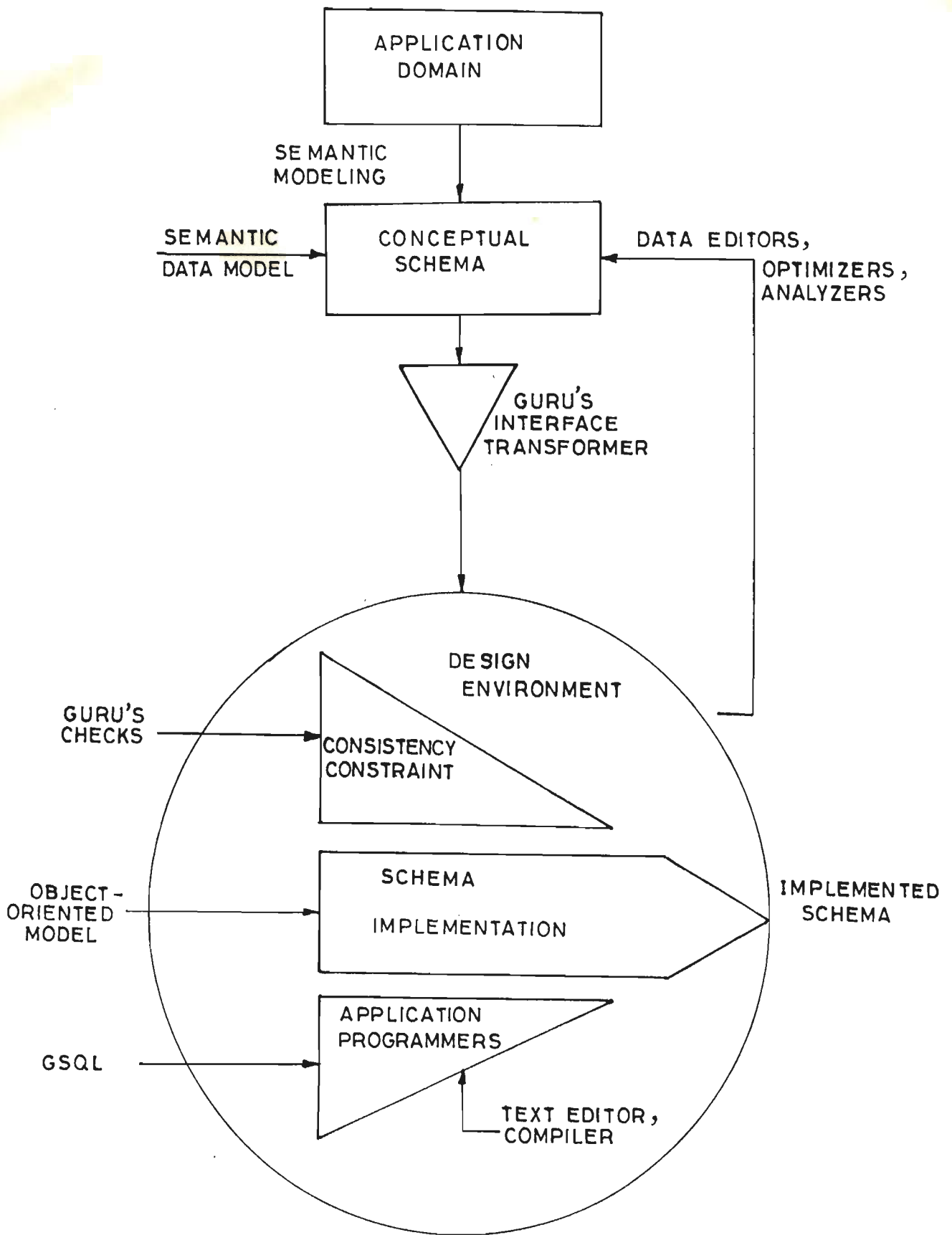


FIG. 4.2 GLOBAL SCHEMA STRUCTURE

The different fragments having the different names of their entities and attributes are kept in the form of the nodes of a tree, and the branches connecting them are the relations existing among them hence forming a class or object structure. A fragment is organized over certain keys either with a single or multiple attributes to form a single concatenated key for a fragment. Secondly, one fragment can be organized on any number of keys, and the keys can also be defined as partial values of the attributes. This is done with a proper indexing scheme, which is organized on the different local terminal nodes. A method is used to define synonyms of the various objects, taking part in the global heterogeneous environment. Which also helps in stabilizing site autonomy to some extent. Therefore, relieving the global schema without modifications for the local database processing, and the manipulations are done to fix up the local schema, seeing the local requirements.

Apart from the prescribed relations maintained in the form of tree structure, the relations can also form the plex or network structures therefore, their relations are maintained in the form of linked fragments either available at the local terminal node or are present on some other remote terminal nodes. The detailed node addresses are maintained for these linked nodes in the schema with linear linked structures, as shown in Fig. 4.2. The different classes maintain their structures as they move from the global schema to form the local schema in the form of transaction objects. This maintains the integrity of the database during dynamic transactions.

Making the separate independent global and local schemas improve the site autonomy and provides the fast reply to the queries with better reliability and efficiency of access.



SCHEMA DESIGN IN GURU DDBMS

FIG. 4.7

Providing the better site autonomy increases the availability of the system hence reducing the system performance time. Therefore, a system has been designed having all the qualities to provide the fast transparent access to the data by providing the global and local schemas accordingly. The detailed global view of the data and the fragments of the three kinds discussed earlier, are associated with the global schema. The other details, defined by the users are maintained in the local and global schemas. The description about the various access methods used with the different database fragments like, providing the index methods with different assigned indexes and any other methods are recorded in the form of a problem template.

The data structures used in maintaining the templates are defined by the users. The user defined data structures are used to mechanize the system for the efficient access and the methods used to access or link some attributes, entities and the fragments required in the problem.

4.3 SCHEMA DESIGN PROCESS

The schema design steps are illustrated in Fig. 4.7. The *application domain* provides information about an application using *schematic modeling* (in natural language). The conditions are focused on the available DDBMS (GURU), which provides the facilities of data abstraction. The various efforts (in semantic model) are made to categorize the problem and provide the format which would be fitted with the available tools. System provides *data editor* and *optimizer* to give a required shape to the data. *Analyzers* are used to analyze a situation and provide information about the various types and kinds of formats for the efficient

and consistent implementation of the required data. The features from the *design environment* with the feedback policies refine the data. Later, the format is transformed by the *GURU's transformer interface* for the GURU shell. The consistency checks are applied to verify the data consistency with the systems and a *text editor* is provided to prepare the complete schema. A database programming language (GSQL) is used with the *object-oriented data model*. Finally, the consistency is checked with *consistency check procedures* and the schema is finally implemented.

Here we have proposed a scheme, which is used to prepare a schema with the help of an interactive command instruction language. The language uses all kind of constructs needing for the definition of objects and the maintenance of their description used for the schema integration etc.

The following GSQL instruction defines a class (structure):

```
define structure class-name
[include object object-name-list
[as {subset | superset | union | intersect}]]
[include structures class-name-list
[as {subset | superset | union | intersect}]]
[datafiles data-file-name-list
[fields field-name-list]]
[memory memory-variable-list]
[procedures procedure-name-list]
[functions function-name-list]
[privilege privilege-code]
[device device-name-list];
```

The *define structure* clause defines a class, with the

other classes and objects as *subset*, *superset*, *union*, or *intersect*. The other objects are inherited for their basic structures as classes. The following procedures create the different objects using classes.

```
create object with structures class-name-list;
```

```
create object object-name  
[include object object-name-list  
[as {subset | superset | union | intersect}]]  
[include structures class-name-list  
[as {subset | superset | union | intersect}]]  
[datafiles data-file-name-list  
[fields field-name-list]]  
[memory memory-variable-list]  
[procedures procedure-name-list]  
[functions function-name-list]  
[privilege privilege-code]  
[device device-name-list];
```

The following are the rules used to determine class (structure) hierarchy and property (values) inheritance, where S1, S2, and S3 represent classes (structures).

1. If S1 is a subclass of S2 and S2 is a subclass of S3, S1 is a subclass of S3.

2. Subset: If S1 is defined as a subset of S2, an object OBJ-1 in S1 is recognized to be a member of not only S1 but also S2. The object OBJ-1 inherits properties from S2 and superclasses of S2. S1 becomes a subset of S2.

3. Superset: If S1 is defined as a subset of S2, all objects in S2 belong to S1. Properties of S2 which are not specified as generalized properties are inherited by S1. This means S1 also has component objects defined in S2 except the component objects referred to by the reference name defined in generalized properties. And methods and constraints in S2 which are not specified by means of the reference names in the generalized properties are acceptable to S1. S1 becomes the superclass of S2 and a subclass of the classes which have been superclasses.

4. Intersection: If S3 is defined as an intersection of S1 and S2. S3 is considered to be a subclass of both S1 and S2. Therefore S3, inherits all properties defined in both S1 and S2. An object which belongs to both S1 and S2 also becomes a member of S3. S3 then becomes a superclass of the greatest subclass of S1 and S2. Here, the greatest subclass is the class located at the highest level in the class hierarchy being organized by only subclasses of both S1 and S2.

5. Union: If S3 is defined as the union of S1 and S2, every object which belongs to either S1 or S2 becomes a member of S3. S3 inherits properties, which are commonly defined in S1 and S2. Then S3 becomes a superclass of both S1 and S2 and a subclass of the least superclass of S1 and S2. Here, the least superclass is the class which is located at the lowest level in the class hierarchy being organized by only superclasses of both S1 and S2.

6. If subclasses of a class newly defined are not determined using rules 2-5, The class NULL becomes a subclass of the class.

In a few cases, *intersection* operation generates a new

class with no members, and union operation generates a new class with no properties for objects. Here, the conflict of properties to be inherited causes some problems. The rules for conflict resolution of property inheritance are given as below:

7. If the name of a property defined in S1 is the same as that of a property in S2 and S2 is a superclass of S1, S1 inherits the property of S2 by changing its name to the combination of its class (structure) name and the property (data) name, such as "S1->prop_name".

8. If two or more superclasses of S1 have properties with the same name, S1 inherits every property by changing its name to a combination of its class name and the property name.

There is a provision of centralized schema, which maintains the record of the entire database with all tuples and domains in the system. The global schema is maintained in the different terminal nodes in the form of replicated copies. Which helps in providing the information about the various details of the attributes, entities and the fragments. This information is necessary to process queries at the different terminal nodes connected through a network. The local schema is also maintained at the terminal node, on which the user is working. The local schema maintains effectively the view of the local database which is being maintained by the users for their own, required on the same terminal node only. Since, the data recorded by a user is also available to other users at the different terminal nodes concurrently, the owner has to show his or her willingness to allow other users to access the data. On the wishes of the owner the data can be connected in the global database environment. But, the owner may need a complete site autonomy so that the view

recoded by him should not be disturbed by the other users. Therefore, a kind of transparency is required in the system, so that nothing should be disturbed on the way, user is working.

For example, an inventory system where many different terminals are put in the different individual departments, which record the different attributes with the maintained classes from the global schema view point. The purchase department purchases different goods from the different suppliers after seeing their quotations or tenders, which are finally recorded in the database. The issue of different items from the stock is done to manufacture so many other items. Therefore, the inventory is recorded about all of them.

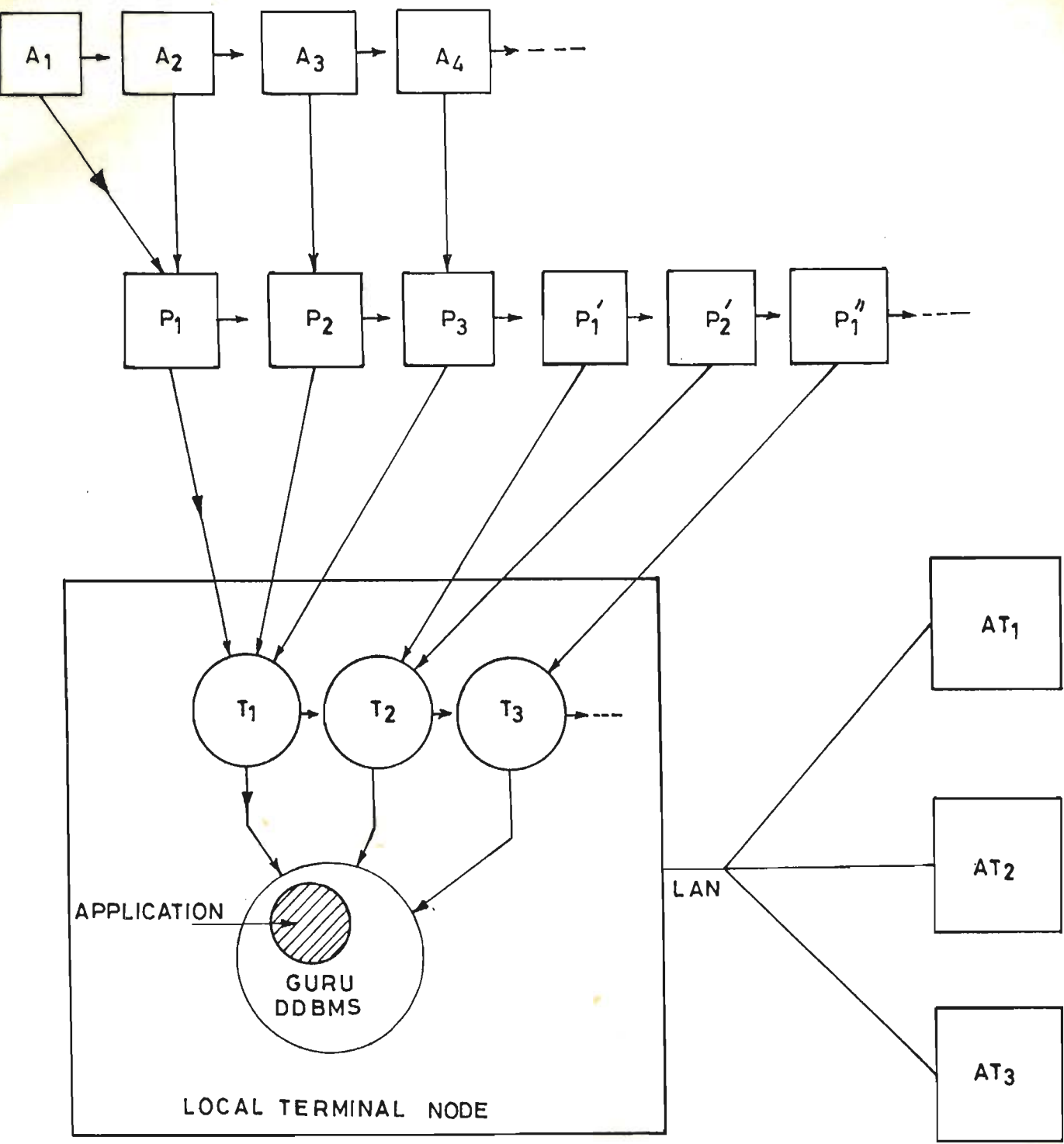
The design of the schema is performed for global and local operations and is discussed below:

4.4 DISTRIBUTED SCHEMA DESIGN

The system comprises the basic three layers at the system level, through them communications are made in the system hierarchy:

1. Terminal node level
2. Programmer level
3. Application environment level

The network consists of several terminal nodes, on them the different programmers work for the different applications. These three levels are identified and processed in the required application domains. The distributed environment provides the several connections to the different applications connected



VIRTUAL CONCEPTS IN GURU
 FIG. 4.6

through the network. If data are required from a terminal node from a particular programmer's application environment, the virtual logging is performed through the network coordinator on the local terminal node. The virtual terminal node gets connected to the local terminal node and the two programmers (local and remote) are linked for the required application environment. The two environments are connected in the form of two objects and have a mutual transfer for the required data, known as the intersection data. Similarly, the different environments are interfaced in the form of virtual programmers and the terminal node planes. These planes communicate among themselves through messages (remote procedure calls). The global schema maintains the information for the different fragments and are referred through the virtual planes. The virtual planes are illustrated in Fig. 4.6. T_1, T_2, T_3, \dots are the terminal nodes, connected to a local application environment. P_1, P_2, P_3, \dots are the programmers connected to the terminal node T_1 , and P_1', P_2', \dots are the programmers registered for the different applications on terminal node T_2 , and so on so forth. A_1, A_2, \dots are the applications belonging to the terminal node T_1 for the programmer P_1 . Similarly, A_1', A_2', \dots are the applications belonging to the programmer P_2 on terminal node T_1 . A local application forms an active link marked with arrows with the local applications. Similarly, the active links are prepared among other terminal nodes for the different programmers and applications. The global coordinator places the control of the other terminal nodes into the local system managed by the local coordinator. The local coordinator disconnects a virtual link on the requirement from some application. AT_1, AT_2, AT_3, \dots are the physical (actual) terminal nodes connected through the network.

The virtual terminals, programmers and the applications

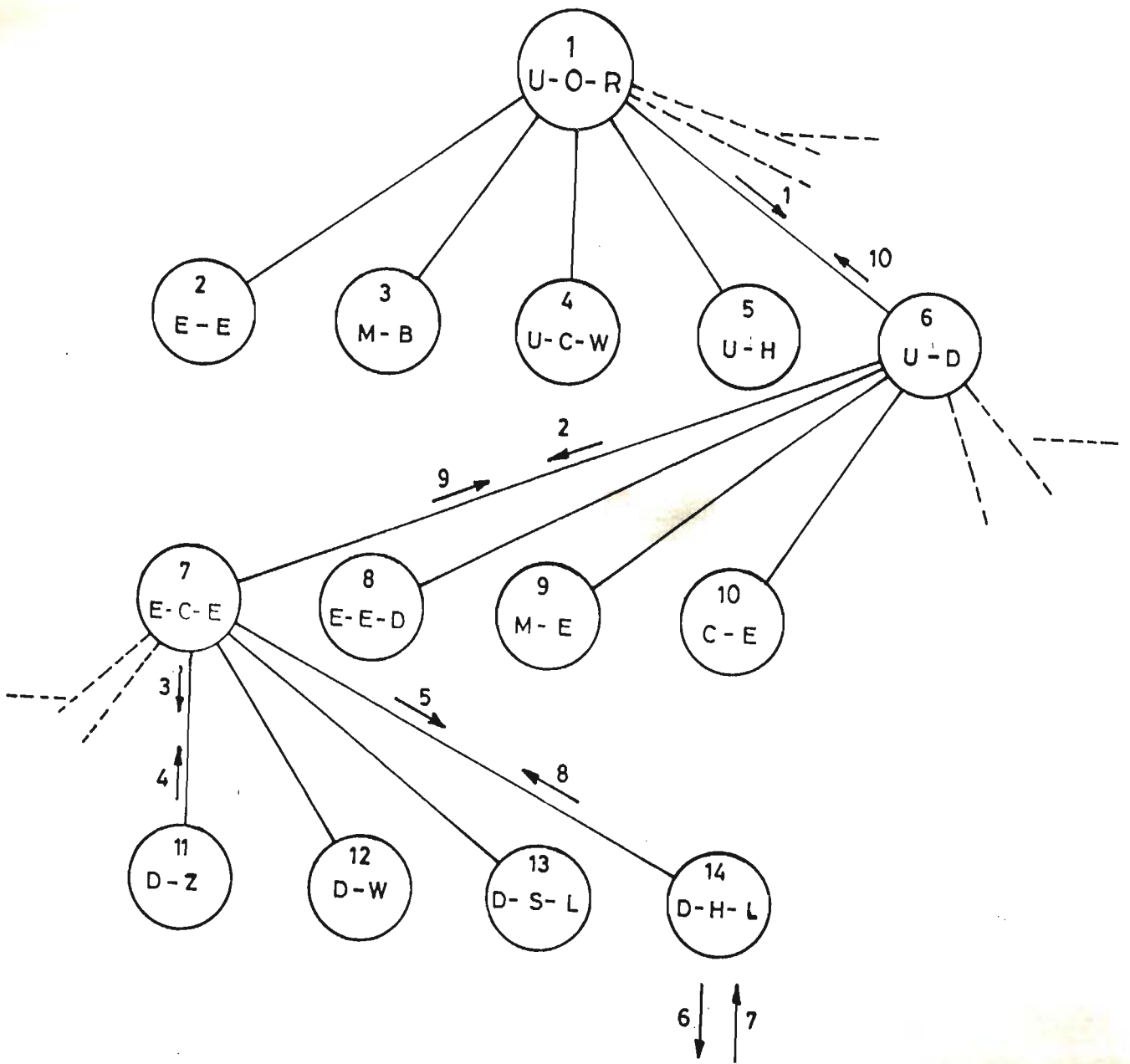


FIG. 4.3 SEARCH PROCEDURE

occupy memory space in the local terminal node. This space is used for the required (intersection) data. Which is made available on requirement from a local application. The transfer procedures maintain the current record of the various addresses (pointers) in the local schema. If the local schema is a static schema, the virtual memory required in the system is occupied for a longer time. This time is minimum in case of a dynamic schema.

Distributed schema is nothing but a logical representation of some related data distributed and linked on some relations at an instant of time. The distributed schema design is illustrated with the help of a problem. Here, it has been illustrated with a problem of University of Roorkee and the maintenance of data for the same. Roorkee University is organized with so many managerial heads, which come under its administration. The university tree is represented in the form of a block diagram in Fig. 4.1. The university has some blocks which come under its direct administration, they are like Employment Exchange, Main Building, University Central Work-shop, University Telephone Exchange, University Hostel Control Block, University Departments etc. Further, a head, University Departments is sub-divided under so many heads like Electronics and Computer Engg. Deptt., Electrical Engg. Deptt., Mechanical Engg. Deptt., Civil Engg. Deptt. etc. Further the Electronics and Computer Engg. Deptt. is also sub-divided in the following heads like Deptt. Library, Deptt. Workshop, Deptt. Software Lab., Deptt. Hardware Lab. etc. Further there is a sub-division possible and so on and so forth. The coded Roorkee University tree is shown in the Fig. 4.3.

The schema used to represent structural relationships, prepared from the symbolic notations is given below:

NODE-1.

U-O-R(E-E, M-B, U-C-W, U-T-E, U-H-C-B, U-D).

U-D(E-C-E-D, E-E-D, M-E-D, C-E-D).

E-C-E-D(D-L, D-W, D-S-L, D-H-L).

D-H-L(FURNITURE, APPARATUS, STAFF).

APPARATUS(C-R-O, D-A-S, M-C-C).

All the nodes of the tree represent head names, and the child nodes represent the child's node name and its relationship with the parent node.

The following are the GSQL instructions used to draw schema and process queries:

```
define structure apparatus
datafiles frg_1
fields c_r_o, d_a_s, m_c_c
procedure apparatus;
```

```
define structure d_h_l
include structure apparatus, furniture, staff
procedure d_h_l;
```

```
define structure e_c_e_d
include structure d_l, d_w, d_s_l, d_h_l
procedure e_c_e_d;
```

```
define structure u_d
include structure e_c_e_d, e_e_d, m_l_d, c_e_d
procedure u_d;
```

```
define structure u_o_r
include structure e_e, m_b, u_c_w, u_t_e, u_h_c_b, u_d
procedure u_o_r;

create university with u_o_r;
list c_r_o of university;
```

The following instruction creates a flat file (database fragment):

```
create frag-name;
```

The *create* instruction puts the user in a full-screen mode for designing the structure of the new database fragment. In this mode from 1 through 255 attributes can be defined. The five items can be selected for the attribute type by pressing the space bar key, which are discussed below:

1. Field name: The field name (attribute name) can be up to 15 characters long and can contain first character as a letter or underscore and the remaining characters could be underscore, letter or a digit.

2. Field Type: The field type is one of the five data types: Character, Numeric, Logical, Date, or Memo. To choose one of these types spacebar is pressed for next option. The selection is made by pressing a return key.

3. Width: The width of the field can be up to 950 bytes for character fields and 20 digits for numeric fields. The date logical and memo fields have fixed width of 8, 1, and 10 bytes, respectively. These width are entered at the time when the data

type is selected.

4. Dec: The decimal item specifies the number of decimal places for the numeric field types.

5. Index: This is a field which is reserved for the indexes to be linked with the required fragment and carries the information about the index tag.

When a user completes the structure definition and exit from the create mode, system provides an option of beginning the data entry immediately or exit. If the user chooses the begin entering the data, the system moves to the full-screen append mode. The default extension to a database fragment is *dbf* and if memo field is selected, a new memo file has the extension *mmo* which is opened with the same name prescribed for the database fragment as *alias*.

```
create database database-name;
```

The above instruction opens a fresh directory in the current system for a new database and build a catalog file in the directory to initialize the database. GURU uses these catalog tables to keep track of objects that a user creates subsequently inside the database, including fragments, indexes and views. System also creates an other file for data dictionary, where system records the details of synonyms created for the database. The extensions of the files for catalog and the data dictionary are *cat* and *ddt* respectively.

The following instruction joins the two database fragments on a specified key and the condition and produces a new

database fragment:

```
join with alias-name to result-frag-name  
for spec-cond [fields attribute-name-list];
```

The *alias-name* is the area name or the fragment name of the target fragment which will be joined with the current active fragment. The *result-frag-name* used with *to* clause is the name of the resultant fragment. The resultant fragment copies the information from the two database fragments for the specified condition, *spec-cond* with *for* clause and the optional specified attributes, *attribute-name-list* with *fields* clause. Any required attributes are selected in the following format:

```
alias->attribute
```

If the *field* clause is omitted all unique attributes from both the database fragments would be copied in the resultant fragment. But, the maximum attributes that are copied would be 255. The structure of the new database fragment is determined by the attribute names with *fields* clause. Potentially, *join* has combined every tuple from the first fragment with every tuple from the second fragment, creating one new tuple for every possible combination. In most cases, however, only a subset is required of these combinations; the required *for* clause lets a selection be made by specifying a condition.

A complete internal schema is maintained in the computer which helps in providing the quick reply to various queries. This schema is a logical representation and is further divided in two ways for the distributed database management system. Which are the global schema and the local schema. The

global schema is one which is common for the whole application and further kept at more than one terminal nodes of the distributed system. This is done to increase the availability of the distributed system. More are its replicated copies present in the system more will be its availability.

The second kind of schema is local schema, which may also be called as the subschema. That is maintained at the different individual sites and depends on the requirements of the site for the later data updates to be made. Therefore, the local schemas are designed to see the site requirements. For the above example, the data can be recorded at the terminal nodes lying in the university hostels, which would be more likely of the student's university enrollment no., student's name, student's class, student's mess dues, student's room rent, student's electricity consumption etc. They all can be recorded in a local hostel schema or distributed hostel subschema. The hostel authorities are nothing to do with the student's academic activities and the student's general fitness etc. Therefore, local schema is the one which actually records the current most relevant data required at that terminal node. Any other information of the student can also be recorded, which would become necessary in future. But, this is a rare case when this information is required. Therefore, local schema keeps the information of those items which are related and most likely to be called at that terminal node.

With the data description to maintain the information at various terminal nodes of the schema tree, the leaf nodes have to specify the actual format of the data to be prescribed for them. Therefore, here a special data description language is designed and implemented which helps the users to define their own global and local schemas neatly and easily.

The distributed database management system has to involve the intelligence to design the local schema automatically based on the requirements from a subschema. This has been done to choose a method of backtracking and unifying to find the appropriate condition. When this is found, then the local schema has a link to be connected. This way it approaches a standard Artificial Intelligence (A.I.) greedy search technique, also known as heuristics, to find the correct related node. The full approach of connecting a link and finding solution for the distributed link is based on A.I. When a database is searched on a terminal node, procedures are tried to get the results successfully obtained from that node. If the search fails then the next link is tried and so on so forth. This backtracking after every trial forces it to follow the greedy approach for a closest node from the priority list. The system uses heuristics for the automatic efficient data resources required in the different applications.

By keeping the replicated copies of the global schema and local schema, increases the system availability but, increases the network congestion and hence delaying the throughput. By keeping more fragments in the local system, increases the requirement of maintaining more backup memory, which also increases the access efficiency due to better availability of the data but, at the same time, it is difficult to comment over the network congestion. Because, maintenance of the data would increase the network congestion.

The data belonging to a local schema is maintained with certain subschemas applicable with the current problem, where the details have been recorded, maintaining the integrity of the classes and objects, and the consistency with the inheritance.

To handle queries at the different terminal nodes of the network, the query would be given in the form of some attributes required from a terminal node. Hence, all the attributes would be brought in a way to reply this query's attribute structure. Some of the attributes are found right at the terminal node itself but others have to be brought from the other places (terminal nodes) or the query is required to be broken in such a way so that all the attributes present at a terminal node will get the partial query replied, and connect the attributes present with that node itself. The process is continued till the full query command is satisfied with all the required attributes. After going through this process, there should not be any chance, when a query needs an attribute and that attribute is not available on the terminal node. Hence, this type of query processing performs parallel operations, which have to be operated at the same time maintaining the serializability.

A perfect query analysis is repeatedly required so that in spite of all the kinds of deadlocks and timeouts the system works with full efficiency to reply the query. There are some techniques that are applied to find considerably suitable answers. First is using the A.I. as discussed earlier, to find the correct way to divide or break the query into sub-queries. The mixed kind of fragments make the situation worst, getting the repeated cancellations and making the system inefficient, with the improper record of the fragments. Therefore, a modified approach is suggested to go into the depth, and to find the correct query distribution of the different kind of fragments. An approach is used, which records the complete details of the fragments distributed at the different terminal nodes on the global network. The information is also recorded regarding the

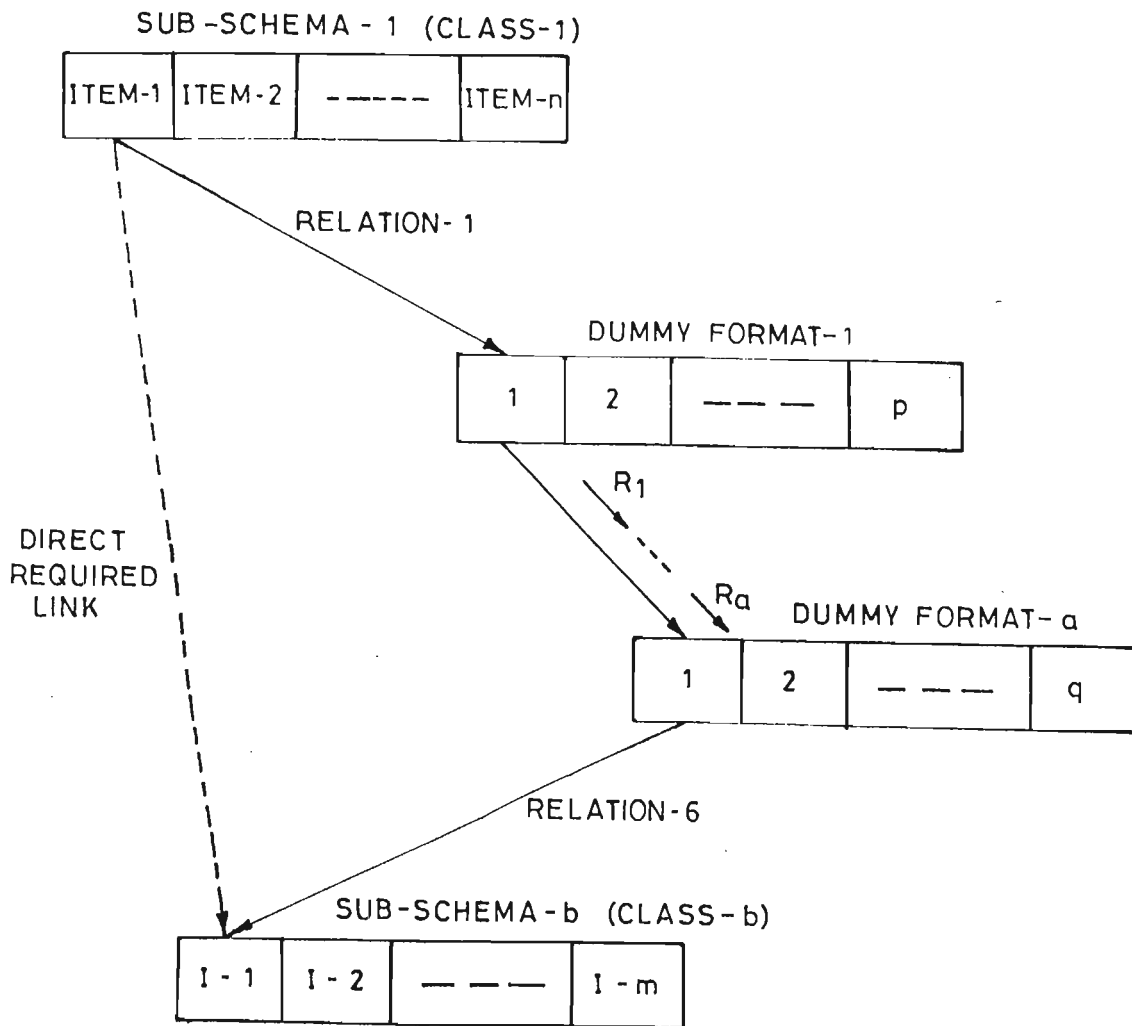


FIG. 4.4 LOCAL SCHEMA

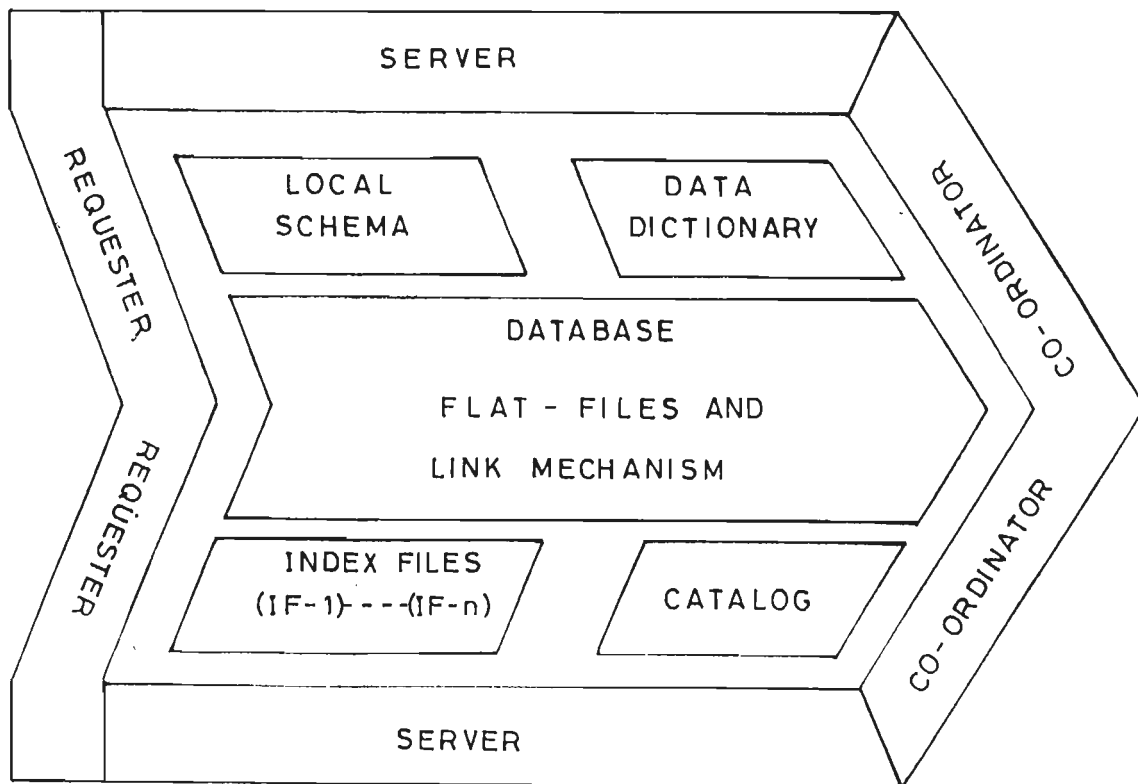


FIG. 4.5 DEMONSTRATION OF INDEX MECHANISM WITH LOCAL SERVERS

replicated and overlapped fragments and the fragment qualifiers. This technique helps users to get a proper transparent system with the fast response. The distributed database management system provides a requester, server and a coordinator to work together to process queries. In Fig. 4.2, a global schema is represented in the form a derived tree from the Fig. 4.1 and Fig. 4.3.

4.5 LOCAL SCHEMA MANAGEMENT

In case, when the fragmented partial query has arrived at a fragment then, it puts the query after loading the system in a specific manner to complete the transaction. The system uses a new object-oriented scheme to respond queries quickly. This technique organizes the fast search operations. The different indexes are organized, which sit to tie files in such a way that a correct relation of indexes is observed. The required records or tuples are accessed from the predefined configuration, organized with the prescribed keys in a prescribed manner.

To fast access these fragments, different indexes are organized, which do not change the original sequence of records in the fragments physically. This also avoids the unnecessary replication of database fragments on the same terminal node. But, some index files which record the pointers over these fragments on some specified keys are also maintained. In overall way, the system looks as organizing the one database fragment. In reality so many fragments form a linked view to maintain a scheme so that, it looks as one fragment is being accessed at a time.

The Fig. 4.4 shows the linking procedures of local schema and Fig. 4.5 shows a fragment lying with the system to be

managed by it. The local schema shown in Fig. 4.4, consists of the traced subschemas or the classes. The dummy subschemas are not recorded but, are shown for the consistency to be maintained in the form of relationships among various objects of a class. In Fig. 4.5, the methodology is shown to reply the query based on multiple keys. This is shown with index organization mechanism. The requester, server, and the coordinator work together with data dictionary and catalog to reply the subclass attribute values. During the search operations, the different kinds of fragments are found. Therefore, this has been proposed that the fragments should be so designed to be maintained completely indigenous in one system as a whole. Fragments not satisfying this condition are required to be handled by a different processor either simultaneously or putting the same job in batches over the same terminal. In this way, the operation is smoothed to provide quick response to the queries.

The regular updates of data do not take much time in changing the active primary key to a new key value. This key places the required query structure (partial query structure) in the memory and meet the desired subquery structure accordingly.

The design of local schema is totally based on the user's efforts to represent the data. On placing a query the user is provided some attributes, which are totally arbitrary. During the design of schema more knowledge is required than what is available normally. Therefore, the subschema is obtained in different ways. First is the new domain configuration should strictly follow the configuration of the query, the other is with the available structure of the attribute groups. How to obtain this structure? Former is impractical because, this may require many groups i.e. factorial of n , where n is the total number of

attributes in the global schema. Therefore, the later is taken into practice. Which leads to fragmented relationships to be maintained. Thus, whenever a query arrives, the fragments and their relations are checked. If a relation does not exist, then either the query is further broken or the fragment is copied from the directed terminal node of the global schema. Later, it is copied to the local terminal node by appending the subschema. The dynamic shuffling of the fragments makes the local schema continuously appended. Thus, the information is needed from the global schema, and the schema is also forced to come temporarily on the local (current) terminal node. The other possibility is that the global schema is also present with the local terminal node in replicated form. Later, the global schema would be consulted to modify the local schema of the terminal node.

The local schema is formed through the global schema by tracing the required attribute from all parent and child nodes on the common keys for the fragments. This is a process of a backward search in the tree structure for the global schema. Whenever, the available fragment's keys match, the operation stops and the relationships are recorded. This operation continues until all the relationships are getting exhaust. This procedure helps in replying queries with long range.

The queries start processing from the local schema, and the data structures are scanned either from the beginning or some other stage in the middle, which is decided by the server. No body has rights to change the global schema except with the proper authorization. The owner of the database is recognized by a separate secret code.

Different local schemas are adjusted with the available

fragments at their own terminals. The local schemas are recorded as flat files belonging to the individual fragments and their organization is maintained with respect to the global schema entries.

After defining a class which belongs to some other classes and objects, the instantiation procedure establishes the correct relationships for the component objects. The relationships are defined with the different fragments on the basis of a key. The following instructions define relationships with other inherited fragments:

```
set relation to [{key-attribute | recno() | mem-var |  
numeric-no} into fragment-name];
```

The tuple pointers in all declared fragments with the objects move independently of one another. This instruction makes the other target fragment's pointer move dependently to maintain the correct tuple relations between the two fragments. The current fragment acts as a parent and the linked fragment acts as a child. The target fragment is identified by its alias name.

The *key-attribute* denotes the attribute name in the target fragment. The *key-attribute's* name could be defined as a synonym, if it is different from the one in the current fragment. The *recno()* is defined as a library function and is available with the library of the system. Which works to establish a relationship depending upon the relative record numbers, obtained through indexing the two fragments on a specified attribute on a composite key, already loaded in the system. The different sequences can be obtained using the different indexes, which control the pointers in a required manner.

The *mem-var* and *numeric-no* represent the destination relative tuple numbers, which becomes fixed for the current active fragment. This acts as if the whole tuples in the current fragment belong to one tuple in the target fragment. If the *set relation to* clause is specified it removes all the relations established with the current fragment acted as parent into other fragments. The usage of this instruction is presented in chapter -3.

The following instruction locates the multiple matching tuples in a child fragment for each tuple in the parent fragment:

```
set skip to [fragment-alias-list];
```

In related database fragments the child database (specified by the *into* clause of the previous discussed *set relation* instruction) contains exactly one tuple for each matching value in the *key-attribute* of the parent database fragment (the current active fragment when *set relation* instruction was issued. In other words, many more tuples exist in the parent database fragment than in the child fragment. In contrast the *set skip* instruction allows users to work successfully with a relation in which this scheme is reversed. The child fragment contains multiple matching tuples for each value in the *key-attribute* of the parent database fragment.

When *set skip* is activated for the child fragment, the all matching tuples are allocated in the child fragment. Each time the tuple pointer advances by one tuple in the parent database fragment, the pointer in the child database fragment progresses through all matching tuples. For a successful *skip*

condition child database fragment must be indexed on the linked *key-attributes*.

In the *fragment-alias-list* clause one or more child database fragment names, linked with the current database fragment are specified. An instruction issued subsequently by a user from the current active environment of the parent database fragment potentially displays all data from all matching tuples in the linked child fragment - providing the scope of the instruction encompasses more than one tuple in the parent database fragment.

The *set skip to* instruction issued without parameters, deactivates the condition for all child database fragments.

The following instruction sets the interval for updating distributed records used with *append/change/edit/browse* instruction, if those same records are also being modified by some other users.

```
set refresh to {numeric-no | mem-var};
```

In a distributed environment one user might be working with a database fragment in any of the *append/change/edit/browse* modes, while another user is changing the database fragment. In this case, the *set refresh* instruction determines the frequency at which system will update the information on the screen of the user who is using editing instructions (listed earlier).

The default value for *set refresh* is 0. Under this condition the refresh operation is not executed. Any value in seconds can be assigned for the refresh time.

The following instruction creates synonyms:

```
creates synonym syn-name for curr-name;
```

Any names for the objects, classes, primitive objects, or complex objects could be defined as synonyms. These names are recorded in the data dictionary. Synonyms are very useful for providing the short names in the system.

The following instruction sets the maximum number by which a fragment locked by another user in a distributed environment could be tried for access:

```
set reprocess to {numeric-no | mem-var};
```

The default value for the *set reprocess* instruction is 0, the system continually attempts to access a fragment or tuple that is currently locked by another user. The attempts are stopped by pressing the Escape key. The status can be set to any value in integers; or a value -1 can be set for infinite tries, which cannot be stopped on pressing the Escape key. The read-only operations can be checked by placing the locks. During a read operation performed by one user on a tuple, the other user cannot change the tuple. The following is the instruction used for this purpose:

```
set lock {on | off};
```

The default value for the lock is *on*. The lock can be made to *off* by selecting *off* value. At this time another user can change the value of a tuple, while other is reading it.

The following instruction displays the records, marked for the deletion in a database, if the *on* clause is selected.

```
set deleted {on | off};
```

The records marked for deletion are not displayed, if *off* clause is selected. The default value is *on*. The following instruction controls the block size of the indexes memos used in the system:

```
set blocksize to {mem-var | numeric-no};
```

The block size for a memo attribute represent the unit of space allocated in the memo file for each memo entry. The default setting for the block size is 1, which corresponds to a normal block size of 512 bytes. A value from 1 through 64 as a block size parameter could be used. The block size in bytes is equal to the current *set blocksize* value times 512. A new block size value applies to the database that is created subsequently under this setting, not the database already exist.

The other instructions used in the design of schema are illustrated in Appendix - B.

A local schema is shown in Fig. 4.4. The following instruction is processed in the below listed points:

```
LIST CLASS, SCORE, G-F-M FOR SNO=5 OF UNIVERSITY;
```

The above instruction is processed in the following steps:

1. Establish relation among all the flat-files (fragments) keeping the proper information in the local schema. The information consist of the relation between the two fragments with their specified keys. The relations of the index files available are also recorded in the data dictionary. This includes the complete keys on which the index file is organized.

2. When all index files along with the database fragments are loaded in the memory, establish the relation among them with the proper keys. This method helps in creating the one logical fragment of the database. That is done for the minimum overhead in the system.

The server supervises over the different tuples and the fragments, places them in sequence with the required attribute names in which they arrived through the user. Later, a required sequence is given to the tuples and are accesses in the required order with fast speed.

The terminal nodes maintaining the global schema keep the reference of those leaf nodes, which are stored at the remote terminal nodes. Therefore, a communication is established by explaining the DDBMS invoking strategy giving the terminal node name and the reference of the global schema title.

The following is the information recorded in each leaf node:

1. DDBMS code
2. Terminal code
3. Local schema name

4. Flat-file or the recorded fragment name.
5. Fragment code qualifiers

4.6 CONCLUSION

The suggested schema design provides data transparency by providing the concept of virtual terminal nodes, virtual programmers and virtual application domains. The intelligent server provides the correct routing of the information through the schema hence it takes the minimum time of processing. Schema helps in providing the right details about the fragments and their qualifiers; this helps in maintaining serializability and query materialization. The tuple search time is minimum by providing the proper mechanism of indexes and their links to the database fragments. Site autonomy is maximized by recording more fragments at the local terminal nodes, which have higher access frequency. The proper privacy and security locks prevent the unauthorized access from the database, and the data cannot be modified. The higher site autonomy increases the replicated fragments. This provides quick response to queries but increases the fragment updation time. This has been tackled by providing optimal site autonomy by the intelligent servers (global coordinators). The information regarding the frequency of access at appropriate locations helps in resolving this issue.

Atomicity of the data transactions is maintained by providing the correct serializability and the information about the commits and aborts. The global commits and aborts are specially watched and recorded in the log. The schema provides information neatly by providing the details of all attributes, about their placements and their order, which is connected through the information on the objects regarding the databases,

frequency of access and formats. The information regarding the linked attributes is quickly provided hence reducing the query response time and present the clear view of the information for various environments integrated with the current application domain. The data structures regarding the various attributes are stored in the form of linked memory elements through the extended area, which provide a perfect view of the data using triggers.

The presented instruction *set relation to* provides good dynamic links among the fragments for temporary joins with different specified conditions through indexes. This makes available the good cohesion in the database. The other instructions like *join* works with different indexes to provide new fragments in different orders, attributes and tuples. These fragments take a new place in the schema to be used in the system.

The object-based operations *subset, superset, union and intersection* provide a facility of connecting the different objects, database fragments, attributes and tuples on the different required relationships. Therefore, it helps in reducing the redundancy, improving performance, and connecting the various database fragments to provide the greater power with the database for a wider application domain. This facility also provides quick response to the queries by the optimal attribute settings. Further the suggested object-oriented model takes the advantages from all the previous data models and fits the data structures for the optimal usage. The developed approach provide the two modes of operations the static and dynamic modes. The both modes are advantageous in the different situations. The static mode is more efficient when the database is used by successive instructions therefore, it saves time in reloading the database structure. The dynamic structure is used for relatively different

application domains for the successive queries, and provide good access efficiency.

SCHEMA INTEGRATION

5.1 INTRODUCTION

A heterogeneous schema is an integrated view of some schemas with different DBMSs. Which are lying on some terminal nodes of the network and provide the complete definition of all the data in the distributed database as if, the entire data is available at the local terminal node [17],[18],[80]. These schemas may belong to the different kinds of database models. The complete heterogeneous view of the database requested to be linked on a common network, gives rise to a heterogeneous schema. If the native database organization is based on an object-oriented database model then, this schema is called as heterogeneous object-oriented schema. In this schema all the attributes of the various database organizations are mapped to the objects in the native database, having the global naming scheme [108]. Each and every object is given a separate identifier (synonym) for the native database. When these objects are put to work to their own environments to which they belong originally, their names will be given back after conversion to the individual naming schemes. Therefore, each and every object in the native heterogeneous schema remains with a unique name. This helps in maintaining the integrity of the database when the changes occur in the heterogeneous global schema view [15]. The different terminal users access the different database fragments transparently through the heterogeneous object-oriented global schema.

The heterogeneous schema is the most important tool, around which the whole heterogeneous database is built. If the

schema is perfectly designed and implemented then, lot of problems phased by the users during the programming on a system could easily be sorted out. Since, the heterogeneous schema is the integrated view of some local schemas on some terminal nodes connected with the network therefore, it consists of so many limitations. Like, two persons perfect in speaking two different languages want to communicate with each other but, they lack to communicate properly. Similarly, the two different database management systems cannot be connected in an efficient manner to share the two databases belonging to them. This arrangement would have been better efficient if the same joined database were to belong one of them only. In the earlier case, it is only possible if an intelligent translator is available between the persons. Similarly, by improving the design process of heterogeneous schema with intelligent translators, more efficient model can be prepared. Which needs further investigations of the parameters affecting the overall system for the various reasons [80].

Indispensable to any full-scale automation database design, like design processes in general, is the availability of a computer-based design environment. Indeed, considerable effort has gone into the development of database design environments[4]. All of them seem to suffer from the same weakness as design environments in other areas: they emphasize the mechanization of tedious routine work like drawing and book keeping, but give little immediate support to the creative and decision making processes. Hence, to place the automation of the schema integration process into context a detailed view has been presented into the following sections.

5.2 OBJECT MODEL'S PROPERTIES

In object-oriented model, various objects are grouped in different objects and classes, which maintain the relationships among the different objects and classes. Since, the different objects carry the different names, the clear relationships can be defined and maintained giving better structural integration to the database. The following are the properties associated with our object-oriented model:

Encapsulation, all the data and the functions inside an object or class are grouped together to provide the better cohesiveness and tight integration among them. All the local identifiers belonging to the object are only accessed by the member functions (or also known as the methods). But, outside the objects those identifiers are accessed, which are declared as the protected, general or locked public type declaration.

Inheritance, objects can consist of some other objects or classes into them. Similarly, an object can be a subset of some objects and the super set of some other objects. All super sets can access all protected identifiers of their subsets but, cannot access the private declared identifiers.

Polymorphism, this is also known as overloading of the functions. The functions and procedures inside an object or class can be declared with same identifier names but, with different number and types of the arguments. Certain operators could also be defined carrying multiple meanings, depending upon their environments. This is known as operator overloading.

Structural object-orientation emphasizes the agreement of objects into clusters, with establishing relationships between the objects. Behavioral object-orientation focuses on objects as

computation agents which consists of a set of methods (procedures or/and functions) that describe its external behavior, and a private memory. Both categories classifies object classes into generalization hierarchies and provide for inheritance mechanisms which exploit these hierarchies. In behavioral object-orientation much of the application semantics is hidden in the private memory and methods of objects. In structural object-orientation these semantics remain visible in the form of structure that can be manipulated.

5.3 ENVIRONMENT DESIGN FOR SCHEMA INTEGRATION

The conceptual schema is the result of modeling an application domain under the constraints of semantic data model chosen. Construction has been done internally via editors. Intermediate stages of the final design are analyzed for certain formal properties such as logical consistency, nonredundancy, minimality. If these properties are not satisfied, an explanation is given to the designer, and/or automatic correction by means of optimizers be attempted. The adequacy of the schema as a description of the application domain is checked via validation tools, e.g., by some kind of prototyping.

Once the conceptual schema is deemed satisfactory, automatic transformation into an implementation schema for a database system with a given data model start under the constraint that as much of the semantics as possible is being passed on the implementation schema. It requires close interaction between transformation tools and the human designer. Intelligent explanation components should support the dialogue. Properties that are not be mapped, are identified and either automatically converted into consistency constraints, or presented to the

designer for inclusion within application programs. Finally, the implementation schema is subjected to some kind of formal analysis and/or manual post-editing.

Usually the design results in a set of local schemas that cover small sections of the application domain. Schema integration takes place at the conceptual level, subjecting then the total conceptual schema to transformation. Alternatively, the schemas are individually transformed so that integration is done on local implemented schemas. Under the hypothesis that richer expressiveness of a data model contributes toward more automation of the schema integration process, one should choose the level which provides a reasonably rich set of modeling concepts. In general this is the conceptual level.

5.3.1 Design Objectives

The following are the main objectives considered to implement the heterogeneous object-oriented distributed schema:

1. The schema should correctly represent the relationships among various attributes of the heterogeneous database as if the system is a homogeneous system.

2. All the different databases having the difference in the units, they are recorded in, should be correctly translated with the right values.

3. All the relationships existing among the different attributes should be correctly transformed.

4. There should not be any types of conflicts in the attributes with the correct representation of different attributes belonging to the different databases and linked together. If so conflicts should be resolved before getting integrated

with the heterogeneous schema.

5. The different fragment qualifiers should be correctly transformed.

5.3.2 Design View

The schema integration includes the following activities:

1. Comparison: This phase consists of checking all conflicts in the representation of the some objects in the different schemas. The basic two types of conflicts are distinguished: The former arises in connection with concept classes and attributes. The problems encountered here are that synonyms occur when classes or attributes with different names represent same concept or attribute, respectively, and homonyms occur when the names are the same but different concepts are represented. Homonyms can relatively easily be detected, e.g., by examining names derived automatically by string comparison. Where as synonyms are often difficult to determine even under intellectual inspection and should be specified in the form of assertions. Structural conflicts arise as a result of different choice of modeling constructs. Several kinds of conflicts can be distinguished: type conflicts; dependency conflicts; key conflicts.

2. Conformation: The goal of this activity to conform or align schemas to make them compatible for integration. Achieving this goal accounts to resolving the conflicts, which in turn requires that schema transformations be performed. In order to resolve a conflict, the designer must understand the semantic relationship among the concepts involved in the conflict. Simple renaming operations can be used for naming conflicts. For struc-

tural conflicts a wide variety of transformations have been suggested that depend on the particular data model used. Transformations are required to be information preserving.

3. Integration: The conforming schemas are merged by means of super imposition of common concepts, and subsequently restructuring operations are performed on the integrated schema obtained by such merging. The objectives of restructuring are completeness, minimality, and understandability. Completeness is achieved by introducing class hierarchies, additional relationships, or discriminating attributes. Minimality seeks to discover and eliminate redundancies. Understandability obviously is a subjective goal where alternatives must be presented to some sort of arbiter because no objection measures exists as yet for it.

One of the foremost challenges to integration is the discovery of concepts in different schemas that can be considered the same but are presented differently and may even give rise to conflicts, and of concepts that, though not being the same, are however are related.

The following are the design phases to be followed for the integrated schema design:

(1) Object's integrity

The existing relational data model is not good enough to maintain the proper relationships among the different attributes to maintain the integrity of the data. Any one can change the existing relationships. Therefore, some kind of integrity check is required to be maintained with the structure of the database. Any ambiguous changes should not be allowed to be made,

so that the entire view of the data is not affected adversely. The solution to this problem is given by providing the concept of objects and class. The concept of an object provides an inherited link to the various objects and classes hence, ensuring the data integrity.

The another problem is that since, the data and procedures are two separate entities, any wrong procedure can be applied to some data for the purpose of changing the existing relationships etc. Similarly, any wrong data can be applied to a procedure. In this way, the properties of the data cannot be recorded in the database properly. Therefore, a scheme, known as encapsulation is made available to the objects. Objects comprise of some data and procedures which are also known as methods in GURU SQL (GSQL). These methods in the objects can access the data pertaining to them. The internal data is protected from the unauthorized accesses by the external world as is shown in Fig. 3.7. There are four kinds of protection schemes available with the GURU's objects, which are discussed below:

- i. Private: This type of declared identifiers can be accessed inside the methods only but, cannot be inherited in the subclasses or the superclasses.
- ii. Protected: This type of declared identifiers can be accessed inside the methods and are inherited with the concept the subclasses or the superclasses.
- iii. Locked Public: This type of declared identifiers can be accessed in outside environment provided the privilege level is up to mark.

iv. General Public: This type of declared identifiers can be accessed by all categories of the users.

The above methods provide a tight security among the objects. Which cannot be broken up by any wrong procedure applied to access the objects or the identifiers hence, maintaining the required integrity in the database and protecting the schema, from the unprivileged processes.

All the attributes required to be transferred to the integrated heterogeneous schema are required to be mapped into the objects of appropriate kinds. Since, the attributes can have some common names with the heterogeneous schema, which might lead to confusion with the schema object identifiers. Therefore, a naming scheme is implemented, which maintains the synonyms created by the heterogeneous schema manager, on the heterogeneous schema. These, synonyms can later be referred for their local identifier names from the data dictionary, which is maintained regularly.

(2) The Design Structure

To design the integrated heterogeneous schema, the two schemas which need integration should be made available to the local server. To make them available at the different terminal nodes, a process known as registration is invoked. Through this process, any other terminal node wanted to take part in a specific global application being carried out on the current network can take part as an active participant. The database which would be linked subsequently is controlled in the integrated environment, enabling the whole application to work with more data and knowledge and thus, enhancing the growth rate of application for

the overall incremental application growth. The other alternative is that for a specific application working for a goal, can distribute it in many independent subgoals. These subgoals would be performed independently at the different terminal nodes situated geographically apart but connected with a network. Therefore, by fulfilling the subgoals, serves the main objective of achieving the overall goal. In this way the complete global tasks can be divided into some dependent and independent tasks which can work for a common goal hence, contributing in the entire upliftment of the global regime. There are vast amount of applications for this area, where knowledge grows with much faster speed due to so many parallel operations independently performed to meet a common goal.

(3) Registration Process

The following are the schemes working for the process of registration:

- i. *User requested scheme*: Any user can place a request to GURU DDBMS to provide a list of heterogeneous global schemas for some applications through its own DDBMS on which the user is working. Then the requester would be provided a list of available global schemas for the required application. The list consists of the details about the various objects and their relations set in the global schema through a schematic. Further, the amount of the objects is decided by the system depending upon the privilege level of the requester. The requester with the top privilege level is given the entire global view of the objects and the relationships being maintained among them. Others can get the objects depending upon the value of the privilege they have. In Fig. 3.7, one object is illustrated with the four types of

restricted access levels for its identifiers and the methods. This restricted level access is imbibed in the objects. Therefore, enabling the automatic in built procedures to be worked out for the various granules of the data for providing access to its own encapsulated identifiers.

Later the user fills a form representing the entire relationships and the detailed entities in the format specified in the form. This form carries the complete logical relationships of the data, which are required to be linked in the global schema along with the limits of the fragments or the database recording the information on specific key values. The following information is required to be furnished with the form:

1. The name of the application of the heterogeneous schema with which requester wants integration.
2. Structure of the local schema in the required format.
3. The key expressions of the various individual data base fragments required to be linked.
4. Place in the heterogeneous schema, where requester is requesting to have integration, with a fragment qualifier.
5. Fragments scheme.

The requested local schema to be linked with the system is analyzed for the various synonyms, specified in the requested schema and the integrity as a whole is studied through the mapper, which uses the other prescribed information provided by the

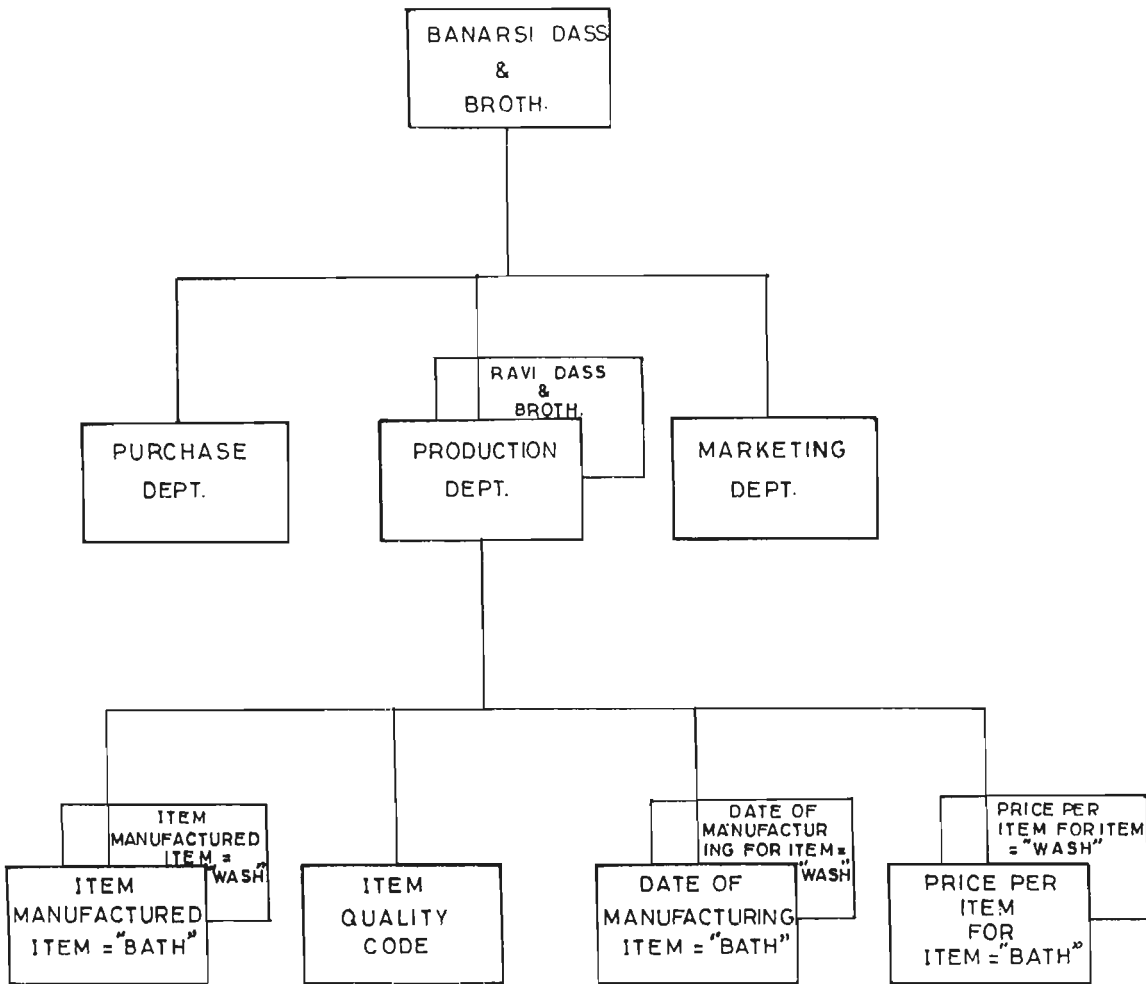


FIG. 5.5

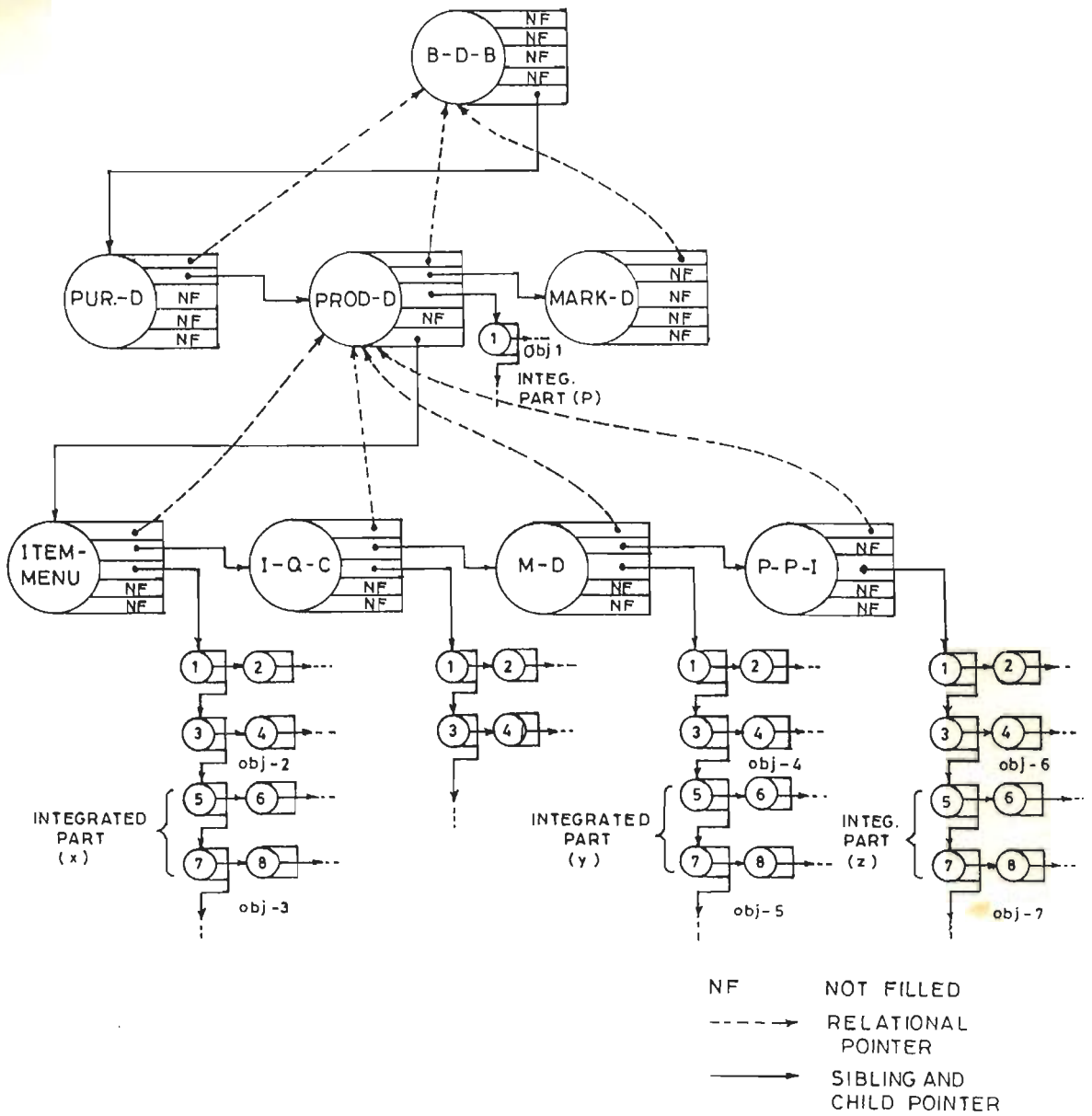


FIG. 5.6

user regarding the various units used to record the data and their validity with respect to time. If the whole information provided by the user is found consistent and satisfactory then, the integrity check is applied and if found successful the mapper transforms the entire names specified by the user through a different naming scheme, used to maintain the synonyms. These synonyms are recorded in the data dictionary. Finally, the integration is performed by converting the specified form format to the object format or transforming the different entities in separate objects with object identifiers. These object identifiers are new with the naming scheme in the heterogeneous global schema. The integration scheme is shown in Fig. 5.5 and Fig. 5.6. All objects in the heterogeneous global schema have certain information which is stored with them to be retrieved in the specific queries. This lets the fragments accessed based on the individual requirements of the queries, by retrieving information stored in the various objects. The following is the information that is stored with the objects, to locate the right requested fragment:

- * Terminal code on the network.
- * DBMS code.
- * Application code.
- * Fragment code.

ii. *User requested automatic scheme:* Under this scheme a user specifies the application name to which the required database is to be linked with. Then requester obtains a form (in the prescribed format) from the GURU DDBMS which would be later filled by the requester. The form comprises the details of the local schemas required to be integrated. The form needs the details of the data and their relationships, which are to be given with the qualifiers and the identifiers of the data

aggregates. The form is later received by the GURU DDBMS for further processing. If all the entries are found satisfactory after verification then the local schema is integrated using heuristics. This approach is the most efficient approach for the schema integration.

iii. *User requested manual scheme*: Under this scheme a user can personally approach the owner of the global application to get his database also integrated with the heterogeneous database for the purpose of incremental growth of the application. If owner agrees then the heterogeneous schema is modified by linking the requested schema through manual programming. Thus, both of the persons (owner and requester) are benefited. Rest of the operations remain same, which are available in the earlier schemes.

(4) Conflicts in Integration and Their Handling

Heterogeneous schema is constructed using bottom up approach from the existing remote database models. It is possible that the similar views are seen from more than one DBMS's and it is also possible that the two descriptions do not represent the way to be integrated with the heterogeneous schema. Differences in the data definition of the similar objects are called conflicts. Conflicts can be classified as below:

i. *Conflicts due to names*: These are the conflicts due to the different facts denoted by same name and also are known as the homonyms. The other is the different names for the same fact and are also known as synonyms. This problem is rectified by giving a global naming scheme, which transforms all the names into some new object identifier names and recording this informa-

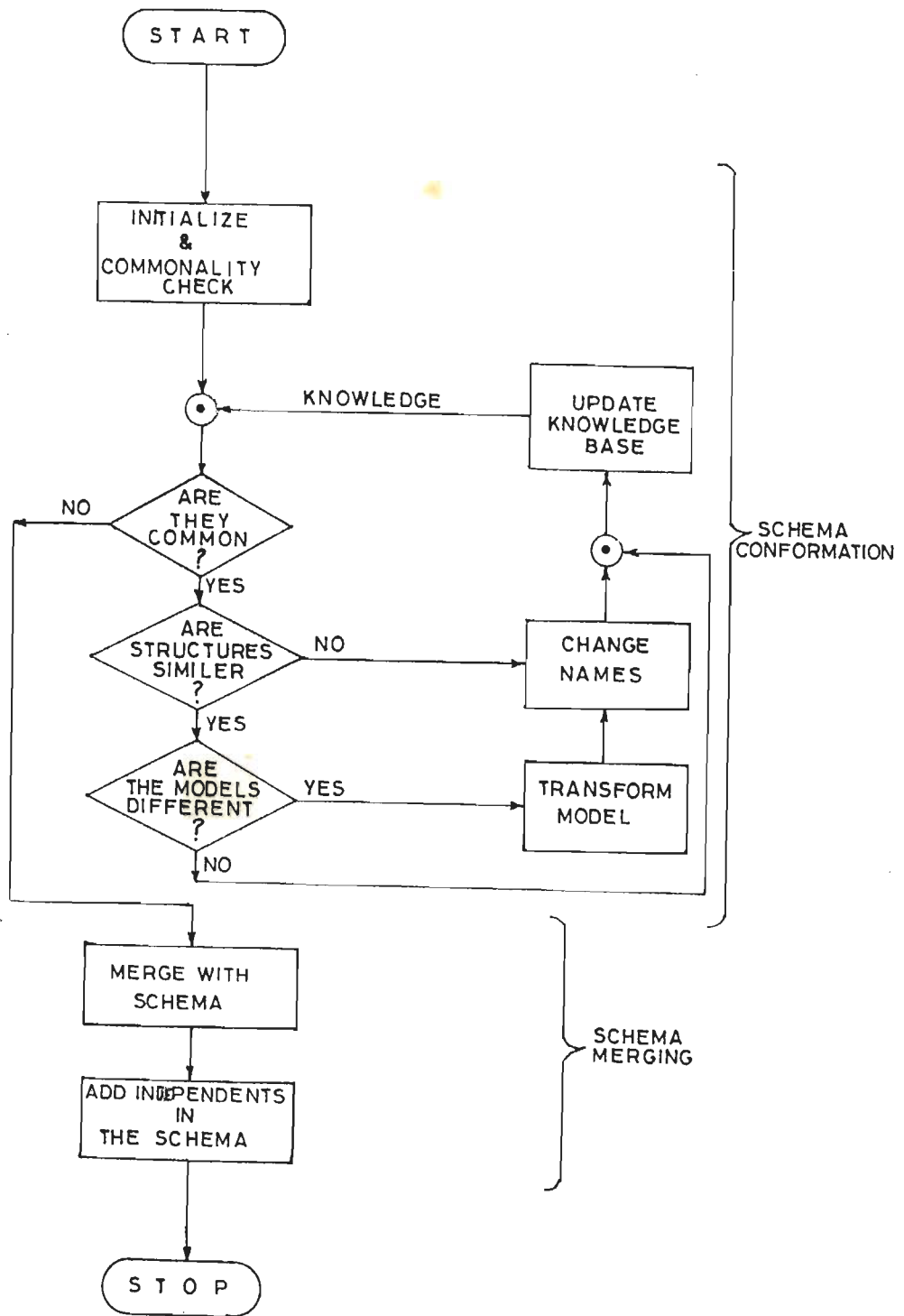
tion in data dictionary for later references.

ii. *Conflicts due to different scales*: In this case different units are used to measure in the two places of heterogeneous schema before integration and the schema required to be integrated with the heterogeneous schema. This can be further tackled by recording the information of the interconversion factor of the different units in the data dictionary. The latest information is regularly updated in the data dictionary about the conversion factors of the chosen scales. For example, the money can be recorded in rupee, pound and dollar. Since, the corresponding conversion values do change with time therefore, they need to be updated periodical.

iii. *Conflicts due to structure*: The structural differences come at those places where same facts are reported in two schemas by using the different elements of the data model. For an example, with a common model including the entities and attributes, consists of the same fact as an autonomous entity in one schema and as the attribute of the different entity in the other schema. This can be controlled using the two different structures maintained separately with different names.

iv. *Conflicts due to different levels of abstractions*: In this one schema contains more detailed information than the other. That is more attributes in one schema than the other. This can be checked by transforming the correct type attributes (or objects) into the new synonym objects at the heterogeneous schema site and the missing object types at the heterogeneous site can be created new in the same class with the other objects.

(5) Integration Process Design



HETEROGENEOUS SCHEMA INTEGRATION PROCESS
 FIG. 5.8

The integration process is the step by step procedure to be followed to link the two schemas in the heterogeneous environment. To organize the integration process the two basic phases of schema integration are chosen, which are the schema comparison and the schema confirmation. Lastly a phase is chosen to merge the schemas.

First the similarities of the two schemas are located. If two structured object types have some common types, then they are assumed to be similar. The environments of the two structures within their respective schemas are compared. This comparison is made for the respective predicates, selecting the ranges and types. If the conditions satisfy then, the modeling conflicts are checked, if conflicts are found then renaming scheme is applied as discussed previously. The structural models are tested to incorporate the knowledgebase to resolve structures using the available knowledge. This knowledgebase is recorded in the data dictionary provided with the DDBMS. Later the conflicts due to abstractions are removed by introducing the new objects in system. Finally, the two schemas are integrated.

The steps for schema integration are explained with the help of a flow diagram (refer Fig. 5.8). The integration steps have been organized into the two interleaving and iterative phases of schema comparison and confirmation and a succeeding phase of schema merging. To perform overall steps, a knowledge about the individual schemas as well as heterogeneous schema has to be managed. The iterative phase basically manages the comparison operations between two schemas to state the similarities. The similarities can be arranged into two groups conformed and suspected. The process of *initialization* starts by initializing all the conformities. The two structures (small group of attributes)

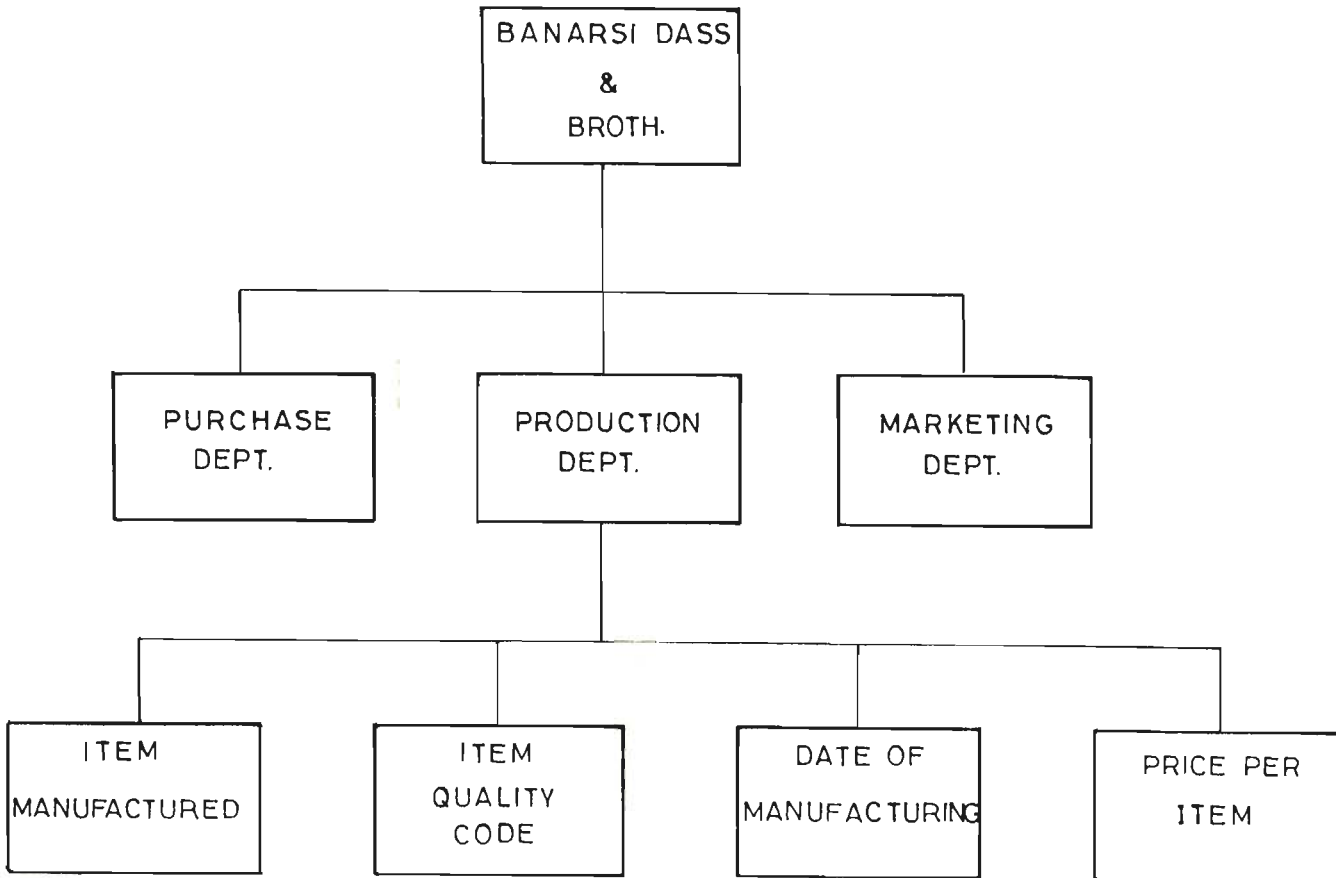


FIG. 5.1

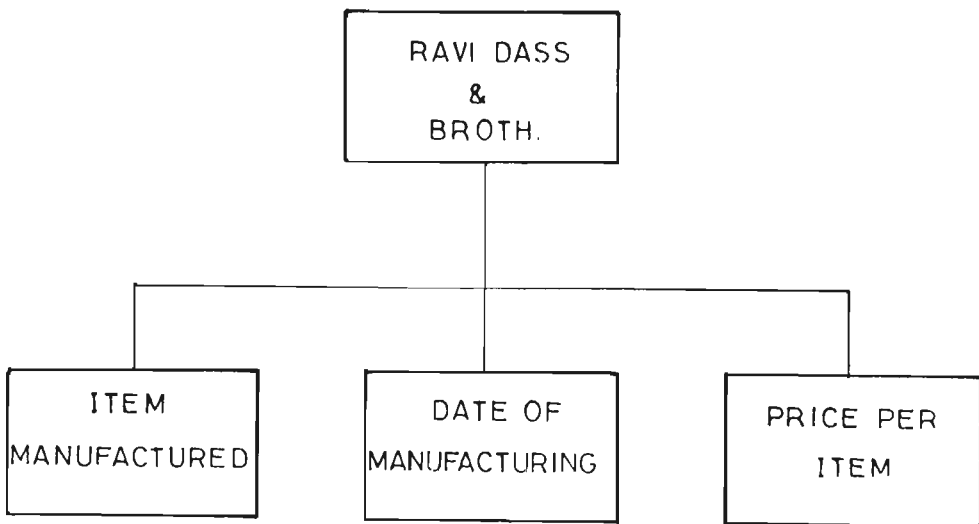


FIG. 5.2

are taken from two schemas and compared. If two structured objects have same component types in common, they are assume to be similar. The next step asks about the similarities for the last selected group. If they are not found common, the groups are send to the merging operations otherwise the similarities in the structure would be checked. If the structural similarities are not found, the names are changed and the knowledgebase is updated for the new group, and the knowledge is circulated. If the structures were found similar, the data model is checked, if the two data models are same then the knowledgebase is updated and recirculated. If the data models were different, they are transformed and the names are changed to support in the data model. Later, the knowledge is updated and the operations are performed till the all attributes of the two schemas are exhausted. The merging of the independents is done in the schema. The whole process is implemented in terms of check procedures (predicates). The structures are finally resolved into a heterogeneous schema.

The distributed global schema design can be illustrated with the help of an example easily. Here, it has been illustrated with the help of some problem of two companies Banarsi Dass & Broth. and Ravi Dass & Broth. both are connected through a LAN and are manufacturing all kinds and varieties of soaps. The company Ravi Dass & Broth. is a new company in comparison to Banarsi Dass & Broth. company. The Ravi Dass & Broth. company is working in ORACLE for its database and wanted to link their own database with the database of Banarsi Dass & Broth., which is working in GURU DDBMS in heterogeneous environment with some other companies. The Individual schematics are represented in the Fig. 5.1 and the Fig. 5.2 for the Banarsi Dass & Broth. and the Ravi Dass & Broth. respectively. The internal schemas of the two companies, Banarsi Dass & Broth. and Ravi Dass & Broth. are shown

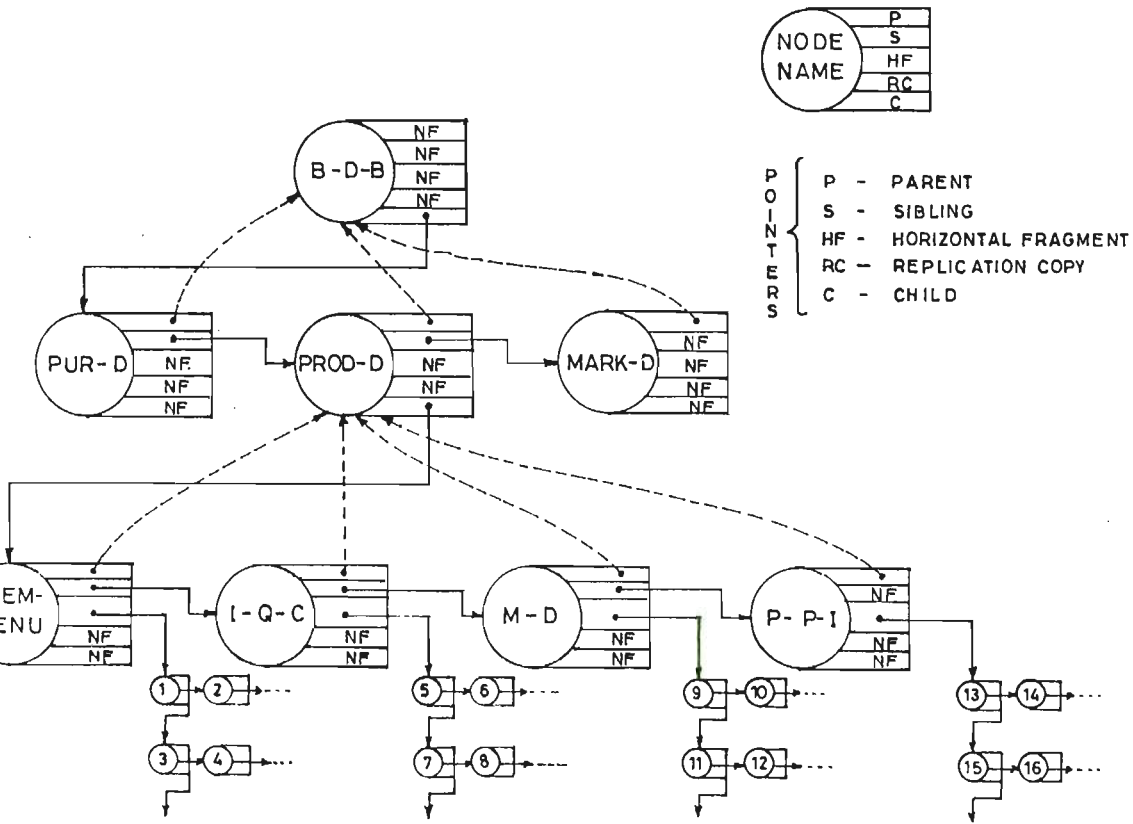


FIG. 5.3

FIG. 5.3

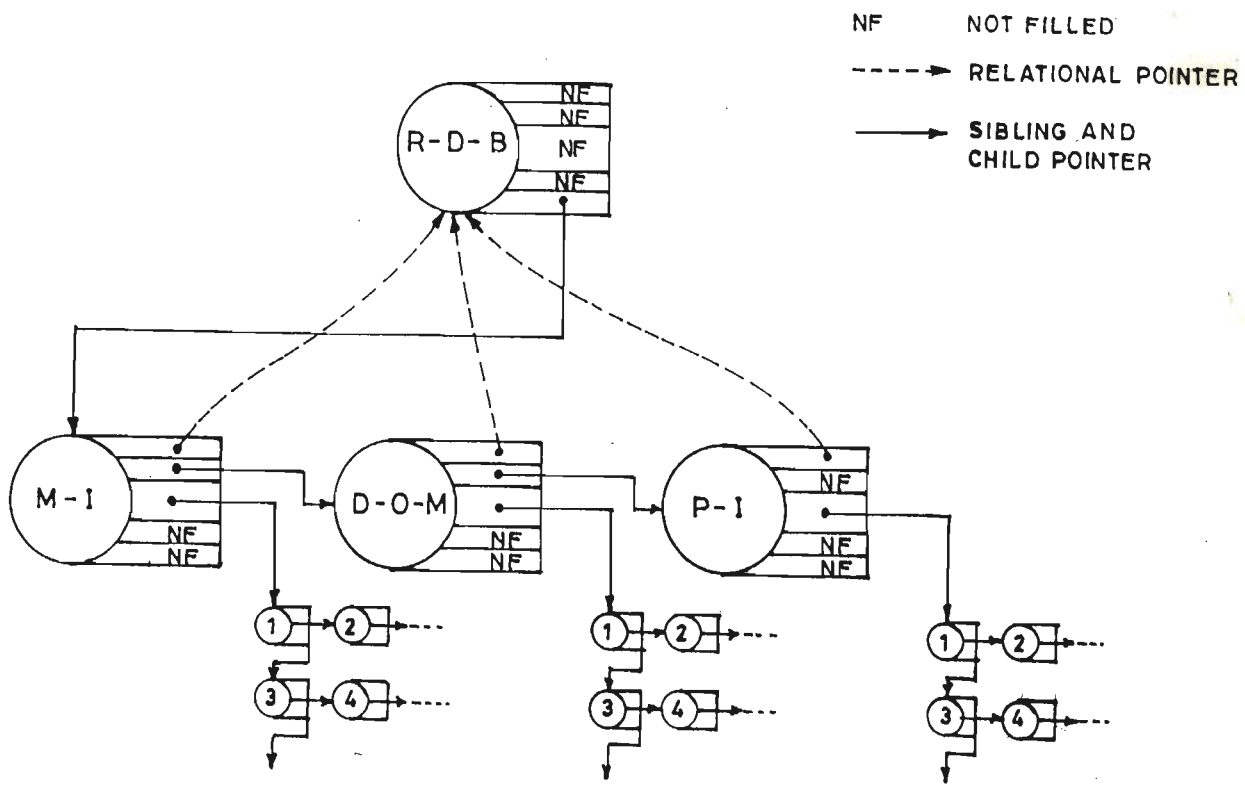


FIG. 5.4

NAME	SYNONYM	DBMS CODE
R - D - B	OBJ - 1	56 - 27
M - I	OBJ - 2	56 - 27
D - O - M	OBJ - 3	56 - 27
P - I	OBJ - 4	56 - 27

FIG. 5.7 SYNONYM TABLE IN DATA DICTIONARY

in Fig. 5.3 and Fig. 5.4 respectively. The integrated schematic is shown in Fig. 5.5 for the two companies. The heterogeneous view for the integrated schemas is shown in Fig. 5.6. In Fig. 5.5, the intelligent integration is shown through the search process and in Fig. 5.6, the synonym Obj_1 is shown for the block abbreviated name R_D_B. and other synonym Obj_2 is shown for ITEM_MANU etc. These synonyms are recorded in data dictionary as shown in Fig. 5.7. The schemas prepared with symbolic notations are given bellow:

The structural schema prepared for Banarsi Dass & Broth. from Fig. 5.1:

```
B_D_B(PUR_D, PROD_D, MARK_D).  
PROD_D(ITEM_MANU, I_Q_C, M_D, P_P_I).
```

The structural schema prepared for Ravi Dass & Broth. from Fig. 5.2:

```
R_D_B(M_I, D_O_M, P_I).
```

All the nodes of the tree in Fig. 5.3 consist of five basic pointers parent pointer, sibling pointer, horizontal fragment pointer, replication copy pointer and child pointer, by which these nodes can maintain the various kinds of relationships to the various other named nodes in the tree. In Fig. 5.3 all the nodes in one horizontal layer are belonging to one class and every node is an object.

GURU Data Models:

The data models in GURU DDBMS are classified as below:

- i. Type: Type declarations have unique names and can also represent the group of objects that share the common characteristics. Types are organized in a directed cyclic graph that supports generalization and multiple inheritance. Objects that are instances of a type are also instances of its super-types.

- ii. Objects: They are uniquely identified by their object identifiers. Some objects like numeric, character, logical, date and memo are self identifying characters in GURU DDBMS. Objects may gain and loose types dynamically.

- iii. Methods: They are the manifestations of operations and provide mapping among objects. To represent relationships among objects, giving their properties and computations on objects are expressed as methods. Arguments of the methods and the results given by them are typed.

The integration process can be illustrated as one student working as a student in one schema model and the same student can work as a teacher in the other schema model, they can be integrated for the common results with a name gentle-man, which shares both the properties of a student and as a teacher and is represented by the instructions in GSQL as below:

```
CREATE OBJECT RAM WITH STRUCTURES GENTLE_MAN;  
INSERT OBJECT STUDENT, TEACHER INTO RAM;
```

The above first instruction creates an object with identifier RAM of type GENTLE_MAN. By the second instruction shown above The two objects STUDENT and TEACHER are joined (inserted) in RAM. The name RAM is the superset of the two objects STUDENT and the TEACHER above.

(6) Role of Translator/Mapper

Translator or Mapper in a heterogeneous system, translates the various queries which are going to be processed at the terminal nodes having the different DBMS. It takes the help from the data dictionary carrying the all information. The information may consist of some scale factors of some of the objects to be transformed. For an example, at the terminal node 1 some objects are processed in object-oriented database management with the value rupees and at the other terminal node 2, where the intraquery is going to be processed, the database management system is a relational model based system and is handling the data in dollars. Then the translator first converts the synonyms from the synonym table of data dictionary and then, converts the object into simple identifier with the name collected from the data dictionary. Next, the identifier with the attribute name is traced through the local database manager in the terminal node 2. The values are read out from the specified fragment located from the information stored in the heterogeneous schema. These values are now tested with the help of the predicate intraquery which is transformed using the scale factor, already described.

In the above case, result fragments are required from the different terminal nodes connected to the network. In that case, the scale factors are multiplied to the values lying in the fragments then, these fragments are transferred to the terminal

node 1. Therefore, translators or mappers play an important role. To test the correctness of the operations performed by the mapper, in integration of the two schemas into heterogeneous schema, some queries are required to be supplied on the local terminal. If the same queries are supplied in transformed conditions on some other terminal nodes then the same results should have come. In this way the testing can be performed of the integrated schema.

Mapper is designed to work in extreme cases to resolve conflicts of all the kinds as discussed previously. The mapper uses the intelligent schemes of fast and correct transformations for the various schema structures, the predicates and the fragment qualifiers. This is found that the transfer of environment takes lot of problems in transforming the queries. Since, queries may contain plenty of functions treated differently in various database management system's query languages therefore, a complete understanding about the various structuring etc. is needed. To process a query, the different functions and operators have to be translated in the heterogeneous environment. Further, the different DBMSs have their individual efficiency of working. This all makes the performance evaluation of the heterogeneous system a trivial task. But, the problem can be simplified to some extent by incorporating the various knowledgebases into the system, evaluating the delay in different intraqueries, and periodically updating the knowledgebases with proper information about the various fragment allocations. In translator, the knowledgebase is accessed and updated periodically which is stored in the data dictionary.

5.4 INTELLIGENT TRANSLATOR/MAPPER DESIGN

The translator/mapper is designed to help in integrating the two schemas to form one heterogeneous global schema. This helps further in resolving the various conflicts in the two schema structures and decide one optimal configuration which is in any way equivalent to the completely integrated schemas. The integrated schema gives the same results to any query launched from any of the terminal nodes on the network. In Fig. 3.4, the intelligent design of the translator/mapper is shown. The following are the functional blocks of the translator/mapper:

1. *Operator Equator*

Different queries and the fragment qualifiers in the global and local schemas are based on certain predicates. These predicates form the different syntaxes on the different DBMSs. For example, the logical operator ".AND." in GSQL is equivalent to "AND" in SQL, and the logical operator ".OR." in GSQL is equivalent to "OR" in SQL etc. Therefore, when a query is launched at any terminal node for an application, this tries to locate the all identifiers from the local schema. If the local schema fails to locate some of them then, the heterogeneous global schema is consulted by sending the intraquery to a node consisting of heterogeneous global schema. The intraquery does not have the syntax of the query language of the DBMS to which it is required to be routed. Therefore, the translator is invoked for the purpose. This along with the knowledge operator syntax equator converts the intraquery operators to destination DBMS syntax equivalence operators. The intraquery after complete equivalent transformations belonging to the destination DBMS formats, is sent to the destination terminal for later processing.

2. *Function Equator*

As the discussions held in the Operator Syntax Equator paragraph, apart from the operators the different function programs are also available in the intraquery. These function programs are also required to be converted into equivalent function programs to be operated at the different terminal node with different DBMSs. For example, the function "int(x)" in GSQL is equivalent to the function "integer(x)" in some other language of some DDBMS. The transformation into equivalent functions is a difficult process because, it is quite possible that the equivalent function does not exist at all at the DBMS where the intraquery or interquery is suppose to be executed. Secondly, if the similar situation occurs for certain fragment qualifier predicates consisting of the different function names. In such situations the alternative condition must be thought about that, the intraquery or interquery should be sent to some other terminal node from where this can be processed. In the case of fragments also, the fragment should not be replicated at the place where such things happen. But, the alternative to this problem is to develop some equivalent functions with respect to those in the destination DBMS. This is possible that the knowledgebase could be used for such purposes and using heuristics equivalent functions can be prepared. Finally, the equivalent function programs are substituted and the interquery or intraquery can be dispatched.

3. *Object Equator*

When the interquery or intraquery is suppose to be launched at the different terminal node then the synonyms are also required to be put back in the format of the destination

DBMS. If the query is being routed from the object-oriented DBMS to the relational DBMS, the all object identifiers belonging to the query are converted into synonym identifiers from the data dictionary and then the intraquery or interquery can be launched.

4. *Name Equator*

When the intraquery or interquery is suppose to be launched at the different terminal node then the synonyms of the identifiers are also required to be put back in the format of the destination DBMS, then the intraquery or interquery can be launched.

5. *Scale Equator*

The different scales are available at the different local schemas about some objects. Then, before launching the query to the destination terminal node with a different DBMS, the query object should get the scale of the destination DBMS's object. This is done by rescaling that object in the query with the corresponding scale factor stored in the data dictionary.

6. *Structure Equator*

In this scheme two structures are compared to be resolved in the heterogeneous schema. There are various ways of launching a query. The same query can be launched carrying the same meaning but the different operators. Hence, the fragment qualifier predicates and the intraqueries can take different shapes syntactically but having the same meaning. Therefore, to check those fragment qualifier predicates and the intraqueries the relational algebra is used. In this way some operators can be

equated by substituting the equivalent values of the operators. For example, the operator ">=" is equivalent to the operator "not <", and the operator ">" is equivalent to the operator "not <=" etc. This technique is applied using heuristic approach and the solution corresponding to the intraquery equivalent format is achieved. In this way the equivalent structure is obtained and the relation is found in the required format. Later, the two structures can be integrated using the earlier discussed approach.

7. Abstraction Equator

In this case there is a conflict of more number of objects inside a class than the attributes of some local schema to be integrated. Since, the GURU DDBMS uses the single attribute transform as an object and this object belongs to some class. But, still this object is used as a separate identifier with some privilege level. Therefore, in GURU DDBMS this does not create any problems and the local schema can easily be transformed into heterogeneous schema with synonyms.

8. Direct Format Resolver

This is the best technique available in GURU DDBMS to hook up speed by inferencing the knowledgebase and finding the exact equivalent expressions most likely used by some queries. This makes the system quickly responded for general category queries. This is done with continuous observations and applying statistics for such queries. This process updates the knowledgebase periodically to increase the efficiency of the system.

9. Translation time evaluation

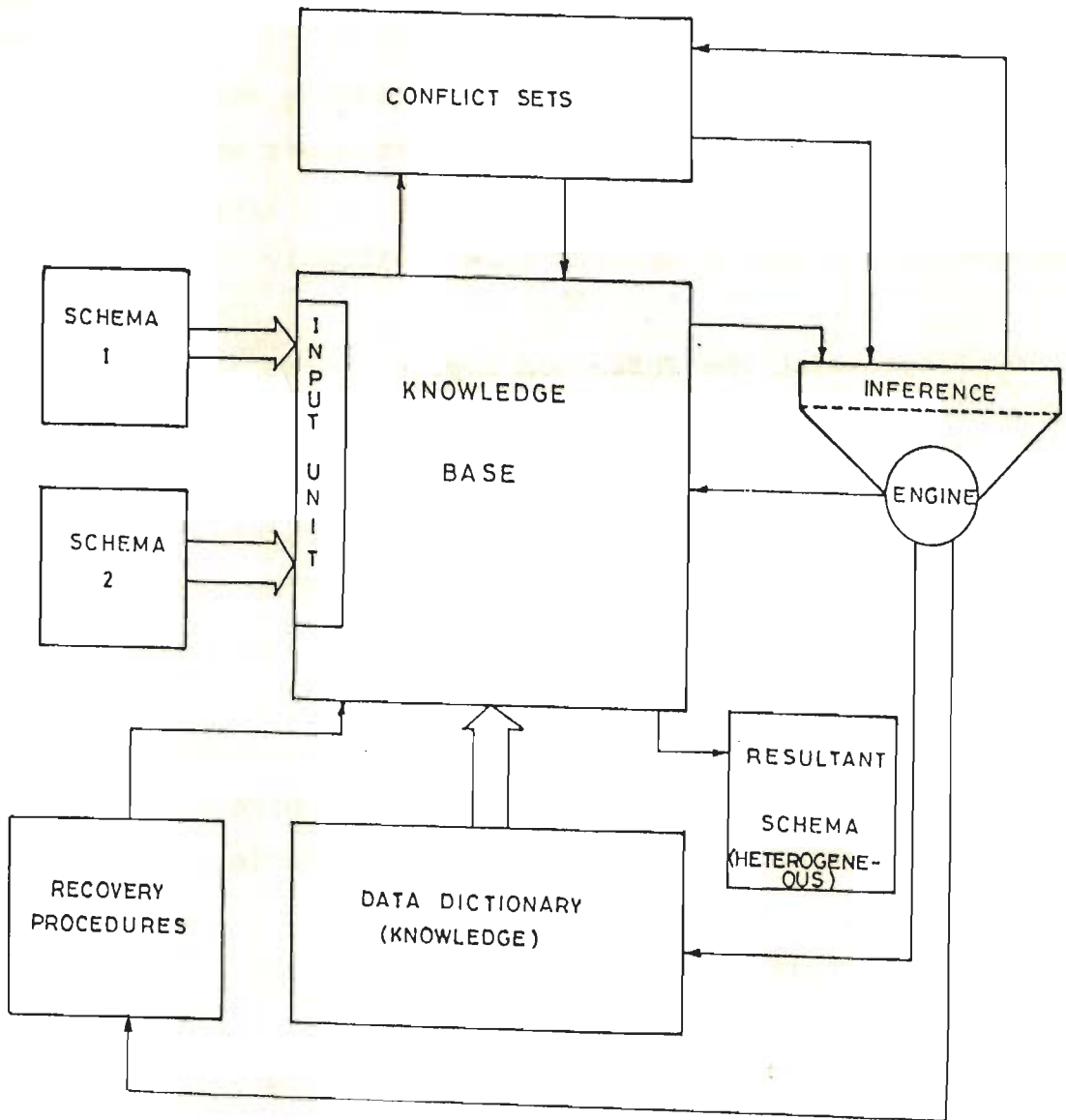


FIG.5.9 INTELLIGENT TRANSLATOR MAPPER BLOCK DIAGRAM

The system continuously records information about the various response times provided to the different queries by some other systems. These systems are finally evaluated periodically for certain queries with the remote values of the cardinalities. Finally, the system records these performance measures of the other systems with some kind of query messages which have the higher frequencies of their arrivals. In this way, the fragment allocation policy can be applied more fruitfully.

10. *Programming the Inference Engine*

The inference engine is designed for some kind of applications. These applications have the procedural programming to the inferencing operation to be performed by the inference engine for some applications. These applications are normally resolving conflicts among certain predicates to provide consistency. For example, the structural equator uses this approach mostly. In other kinds of applications like, direct format resolver, the inferencing is done to maintain consistency among the knowledgebase and the predicate rules. When the conflict set is resolved, it provides a unique solution.

In Fig. 5.9, Inference engine is programmed to be applied to some applications like, structure equator, the all rules of relational algebra are loaded into the memory and the inferencing is done after loading the required predicate into the knowledgebase. Soon as the consistency is obtained among the rules and the predicate, that rule is selected and the equivalent format is obtained. These rules are stored in the data dictionary. The recovery procedures are the procedures recovering the previous consistent set for the query recovery.

5.5 HETEROGENEOUS DDBMS'S OBJECTIVES AFFECTED WITH SCHEMA DESIGN

The following objectives are maintained while performing the integration of two schemas:

1. **Transparency:** This is the first and foremost requirement in GURU DDBMS to implement the complete transparent system having transparent fragmentation, fragment allocation and the global database etc. In this way, the user is not required to indulge in complexities of the database unnecessarily.

2. **Site autonomy:** For better site autonomy, it is desirable that most of the data needed to support queries should be available at the local terminal node without using the network. This provides quick response to the queries irrespective of the network conditions. In this way high site autonomy can be achieved by independent operations on individual sites.

3. **Atomicity of distributed transactions:** A transaction is an atomic unit. The atomicity is required to be maintained in heterogeneous transactions. During the transaction execution a transaction passes through various phases, and if at any phase it does not succeed, all the previous operations performed have to be undone to maintain the atomicity of the distributed transaction. If the heterogeneous schema is well designed, which records the various probabilities of the failures then the transaction would not be routed to that terminal node hence, improving the performance.

4. **Quick response:** The schema has been designed in such a manner that optimal fragment replication is done and the allo-

cation is done at the sites where higher query frequencies are found for the fragments. The replication increases the time of updates, but reduces the read only time of the transactions. Therefore, the optimal number of fragments are allocated after watching the frequency of updates and the frequency of read only queries for the application.

5. **Concurrency control:** Concurrency control is done to maintain the parallel operations performed by the DDBMS. There are basically two kinds of the parallel operations, one is the independent operation the other is dependent operation. Therefore, the independent operations need not to be worried about, as compare to the dependent operations [198]. The reason is that independent operations need not follow serializability criterion i.e., after the operations are performed in parallel the results are given. But, in case of dependent operations, the second operation would only take start when first is finished execution. Later, the results are provided to the second operation. Hence, taking more time as compare to the independent operations. This is known as serializability of the operations. If the serializability criterion is not maintained, obviously wrong results would be obtained.

6. **Quick recovery from failures:** The failures occur due to various reasons. Some are due to the network link failure, memory failure, terminal node failure due to deadlocks, and failure due to timeouts etc. If such kinds of failures occur, the recovery process to the data has to be started. If other copy of the data is lying on the network then it could be recovered. In case of deadlocks, an alternative terminal node with the replicated copy of the fragment could be approached. Thus, the data replication provides the fast possible recovery.

7. *Reliability*: More are the replicated copies of the fragments on the network more would be the reliability. Therefore, the schema design should be done in a manner to have more fragmentation.

8. *Deadlocks prevention and timeouts*: Deadlocks occur due to many reasons, here one reason could be a poor schema design. If conflicts are found in access procedures of the database then deadlocks are bound to occur. Poor written software normally cause deadlocking. In GURU DDBMS a timeout signal is given during a deadlock. The other reason of timeout is the network congestion due to which more delay occurs in the transactions. If certain transactions take more time due to preoccupancy of the system, timeouts are generated. This makes the system follow an alternative path to get quick response.

9. *Minimum cost of operations*: The replicated fragment's allocation is done at those terminal nodes, where their demand is the highest. The network congestion is reduced hence, making the system respond quickly. This also reduces the use of the network if the query could be replied locally (with the local database of the current terminal node) hence, minimizing the cost criterion.

10. *Privacy and Security*: The privacy and security control of the system is provided by providing the various privilege levels to the various kinds of users [83]. The system has its built in security by providing the restricted access to the processes that can access the system.

11. *Integrity*: The integrity of the system is main-

tained by providing the complete integration to the database. Here the integration is provided by providing the object-oriented design, which helps in maintaining the integrity by its own properties and the external privilege levels.

12. *User friendliness*: This is the another feature which makes the learning of the database quick and fast. Lot of restrictions in programming make the users nervous and the disliking for the system start building up. Therefore, this is the very bad symptom for a healthy design of the software, which lowers down the productivity. The object-oriented features make the system easily programmable with less efforts thus, providing good user friendliness.

5.6 CONCLUSION

We have successfully integrated two heterogeneous schemas. The intelligent translator/mapper have translated the query and fragment structures with the machine dependent (hardware) transformations. For the real life applications, a system running ORACLE interfaced with GURU has supported the heterogeneous schema. Good data transparency is obtained with the data dictionary and intelligent sever. A method of schema integration with knowledgebase is illustrated to provide consistency in the database. The various kinds of conflicts are resolved by the comparison of the two schema models with the suggested thumb rules. The *operator equator*, *function equator* and *direct format resolvers* resolve the structures in a perfect manner and help in the schema integration.

The different methods are suggested to improve the integrated schema by preserving the laid out objectives for the

integration.

The intelligent approach gives rise the complete transparency of various details about fragments and attributes and their placements. A lot of burden of remembering the techniques of efficient data access by the programmer and the programming efforts are reduced and the system itself takes the guarantee of efficient, effective and transparent operations required by the programmer.

COMPLEXITY MEASURES

6.1 INTRODUCTION

The heterogeneous distributed database environments are complex compared to homogeneous environments. The reason is that whenever the global database environment is used, the different kinds of database environments are faced, to solve the problem submitted to the local terminal node [13],[180]. The solution of the problem submitted in the heterogeneous environment is based on several factors. The most dominant factor which affects the performance to the worst condition, is the speed of the slowest terminal node, participating in the process.

Seeing the problems with the heterogeneous DDBMSs, it is very difficult to measure complexity and performance. Thus, a very few successful efforts are made in this area of research [142]. Since, the whole problem, distributed among the different terminal nodes have to maintain the serializability of the operations. Therefore, the actual time taken to process the whole problem would add up with the maximum time, taken by the slowest terminal node. The problem of heterogeneous environments is simple to explain with an example, the two experts in one field but, knowing the two different languages and want to communicate with each other, to produce some fruitful results in the field. But, they fail to communicate properly. Because, there are not enough similarities found, between the two languages so that the clear understanding can be established. So, this becomes an exercise in futility. The degree by which the two languages are different with each other is known as the degree of heterogeneity.

ty.

A new incoming DBMS can be tested in the heterogeneous integrated environment with some application. The application should run on the system from any of the terminal nodes and should provide the same results. The testing can be performed with different test sets, written in different languages to be run on different DBMSs [98]. In heterogeneous integrated environment the different DBMSs linked through a network play around the different kinds of interfaces. The DBMS is interfaced with the operating system and the operating system is interfaced with the network manager, which launches the data and instructions on the network. And on the other side of the network, the order is reverse of the earlier discussed interfaces. In this way, the two terminal nodes are communicating through many kinds of operating systems and database management systems, which are designed to work in different environments. The complexity and execution time increases due to the different environments thus, reducing the global task efficiently. Because, on a network a variety of different kinds of environments are available locally, it is very difficult to predict the performance of the system. To trace the criterion of complexity and performance, the different domain objects are mapped after transforming to the other domains. The process uses an inefficient way of accessing the data from the database which do not belong to the DBMS, originally.

To depict the clear understanding of complexity and performance measures the discussions are based on certain facts of internal assignments to the heterogeneous object-oriented GURU DDBMS.

6.2 INTENSIONAL AND EXTENSIONAL NOTIONS

To handle queries using object-oriented query processing approach, the procedures and functions (methods) are categorized under different kinds, which when executed define appropriate required actions. These methods are executed through a call from the other methods like, defining a method for read operations to be performed on a special kind of fragment etc. The methods are typed automatically in some categories. Which are defined when the classes or objects are defined using the object-oriented philosophy. Therefore, whenever certain methods are required in the system, they are invoked automatically from the appropriate objects. This method is very useful in the cases, where the different procedures are specified to operate the defined objects in the different application domains.

The above method is also helpful in managing a large number of methods defined for different classes and objects. The applications are categorized in different domains. These application domains are loaded dynamically to deal with specific applications. Further, the different kinds of environments belonging to some application domains are linked in different ways, thus, providing a heterogeneous application platform. The behavior with the heterogeneous platform becomes very important to handle, for the applications requiring the unique results with the interrogation of the other results.

The defined classes and the objects are placed in racks. A rack basically is an object, which helps in providing a good logical concept, that is very useful with the different application programming tasks. These racks are helpful in defining the basic elementary operations.

Some basic methods are classified with the object or class configuration, which are, constructor and destructor used for the user specified procedures to be invoked at the time of the object instantiation and the deletion of object from the memory respectively. The another category of methods are defined in such a way, that a user has an option to perform any operations at an appropriate time. These methods could also be invoked automatically without the intervention of the user. Whereas the other methods defined in GSQL need the user's procedure call to invoke them. GSQL provides a facility where, the different objects are passed as arguments and they play an important role in defining the different procedures. This process is known as polymorphism for the defined procedures.

For handling the specific applications, GSQL provides a wide range of instructions for activating the specific properties required for some specific behavioral model. The database in GSQL can be handled using all kinds of procedures mentioned by the users. Some of the methods use *intensional notions*, which are the specifications made in the programming language (here GSQL) and uses the existing object module or the object structure. But, GSQL provides a facility of *extensional notions* also, used for the enhanced scheme in the object structure, designed with GURU. This scheme is very helpful in specific applications like, the complexity and performance measures etc. The extensions suggested in the object structure provide a wide range of data and knowledge, which could be imbibed at the specific places in the object structure and executed using trigger procedures, at any required instant during the application execution. These methods could be very helpful in maintaining the secret records for some applications, such as specific defense applications etc. Thereby providing a complete monopoly with the specified applications for

general purposes and for the general public.

For example, the following triggers are used to measure complexity at schema and subschema levels:

- * *Attribute trigger*
- * *Fragment trigger*
- * *Database trigger*
- * *Index trigger*
- * *Sort trigger*
- * *View trigger*
- * *Form trigger*
- * *Print trigger*

At appropriate levels these triggers are activated. An *attribute trigger* is activated when any attribute is accessed, a *fragment trigger* is activated when a fragment is required, and so on. The different triggers are activated for some specific purposes and activating the required procedures to take necessary actions required in the system.

6.2.1 Applications

Some of the basic primitives defined for the general applications are like, the different methods defined in the objects for reading and writing a specific database, from a tape drive of number *my-89032*, using the procedure names *tape-read-my-89032*, and *tape-write-my-89032* respectively.

In the object-oriented approach the existing computer philosophy with the existing hardware and software is not recommended for the general applications considering the efficiency

view point. But since, they are already in use with standard available infrastructure for which already a lot of investment is done in past therefore, the existing computer infrastructure is used with the new implemented concepts. One of them is the use of enormous available main memory (MM), which is very cheap and easily supported in large sizes with the recent operating systems, running on the general purpose computers. With these concepts there is a system which works without any files and messages, and runs very efficiently. Because, the files need the serial conversion of their data unnecessarily, when they are saved on the backup and the reverse conversion is required, when they are read into the main memory from the backup. Similarly, the messages are not needed as the concept of distributed shared memory works out to be very efficient. Using these concepts, the different efficient procedures are written to access main memory and provide very fast and efficient mechanisms for the general purpose applications.

6.3 BRIEF ARCHITECTURE OF GURU

GURU is composed of three major components a local database management system, a data communication component which provides message transmission with a message-task concept, a server, also known as a transaction manager which coordinates the implementation of multisite transactions. The local database management system can be further divided into three components: a object manager which controls objects and classes through the object and class tables respectively and a data manager required to handle the read and write operations on the data, and a database language processor which translates high-level GSQL instructions into operations on the storage system.

An application program places requests to GURU through its local site. All intersite communications are between GURU and the other DDBMSs, here for our discussions GURU is mainly responsible for locating the distributed database for its own users.

The server at the site of the application considers the first GSQL statement which is not within explicitly defined transactions as the start of a transaction, and implicitly performs a *begin transaction*. When the user completes a session, an implicit *end transaction* is assumed and all the work done is committed.

Explicit *commit* and *abort* instructions can enclose several GSQL instructions, when the application programmer wants them to be considered as an atomic unit by the server; after any explicit *abort* instruction, GURU will assume that the next instruction begins a new unit of work, and implicitly issues a *begin transaction*.

To each transaction, the server assigns a unique *transaction identifier*, made with a local transaction counter and the identifier of the site. GURU uses the local process model for computations. Instead of allocating a fresh process to each database access request. Thus, the number of processes assigned to an application is limited and the cost of process creation is reduced. User identification is performed only once at the time of creation of a process and all the active objects required to process the application are loaded into the *process memory*.

An *active object* activated at a remote site can, intern request the activation of other objects at another sites. This

process provides a tree of objects activated for an application. The problems which are due to the concurrent execution of several active objects on behalf of the same transaction at the same site are avoided by strictly serializing their access.

Objects communicate using *sessions* which are established when a remote process is created and are retained for the whole processing. Thus, *active objects* and *sessions* constitute a stable computational structure for an application. Sessions are particularly useful for the detection of processor and communication failures; communication failures are reported to the communicating active objects, taking the advantage of low-level protocols. When the session between two communicating objects fails, the current transaction is aborted; objects which cannot communicate with parents are deactivated and also destroy the sessions communicating with their children, while other sessions are retained for subsequent attempts to continue the application with the different access strategy.

6.4 COMPLEXITY IN DISTRIBUTED SCHEMA HANDLING

In GURU DDBMS there are the two kinds of basic schemas, which are global and local schemas. The complexity can be seen in the heterogeneous schemas as discussed below:

The process of searching the various objects from the schema using the supporting tools like, backlogs, logs, catalogs and data dictionary take considerable computer time therefore, the schema handling is a complex operation creating lot of complexity of operations. The efficient methods to access and maintain schemas are very important.

6.4.1 Query mapping

When the query approaches the terminal node requested to process the query, its structure is mapped into its local schema at the local terminal node by the local server and the required database along with the fast access procedures is loaded into the memory. These fast access procedures are certain index mechanisms, if the index mechanisms are found on that query structure. Otherwise, the query is processed with what best is available at the terminal node. This method involves certain complexities since, the terminal node may not be having the efficient index mechanism required in the query structure. The other aspect is the poor access time due to the poor data tuning in the database in case of queries not having fast access mechanisms.

6.4.2 Query Rerouting

In certain cases, due to certain reasons of not replying the queries within the assigned time slot, the timeouts are generated and the query is required to be routed at some other alternative terminal node. This takes more time and increases the complexity.

After processing the query at the assigned terminal nodes, the results are sent back to the terminal node requested the query processing. But, if the results are carrying bigger size of the fragments then, the routing them through the network increases network congestion. Therefore, a strategy can be adopted before routing the results to the requester terminal node, the requester terminal node is informed about the cardinality of the resultant fragment. This is done by all the terminal processing

the different intraqueries. The query originator terminal node sends the intermediate intraquery to be followed subsequently to a node, to which the other processed smaller fragments can be routed for the operations like, semi-join etc., by sending the intermediate intraquery to the terminal node. These operations reduce the fragment size considerably. The process is repeated till the whole query is processed. Hence, passing the minimum data through the network which improves the speed of operations and reduces the complexity.

6.5 COMPLEXITY WITH TRANSACTION HANDLING TOOLS

The procedures used to maintain the data in the required format with the available methods create complexity due to the access requirements of backup storage. The applications not requiring access of the backup memory run much faster than those requiring access to the backup memory. Most of the transaction tools with general purpose computer systems require backup access hence, this area is important from the view point to investigate complexity.

6.6 COMPLEXITY MEASURES IN FRAGMENT HANDLING

Fragmentation is done to fulfill the requirements of the various users of the database. The data are grouped in different classes and the objects. These classes have different properties on the behavior model of the attributes. All attributes required in some application are maintained in some class where the behavior model of these attributes can be represented. These attributes are named as the *molecular objects* in GURU DDBMS. The different attributes can consist of some type of data. This type declaration is done to maintain structural ato-

micity of the attributes. In GURU SQL (GSQL) the five types are defined as the basic types which are Character, Numeric, Logical, Date and Memo. Any user can choose one among the basic five types for preliminary declaration. Later, the different groupings can be done to handle the basic five types, to form some new types, by putting them in a capsule known as an object.

Any such groups (classes) maintain the structural model to record the data and form the proper groups, which can be transferred from one terminal node to the other, on the requirements. The various relationships among the different fragments are maintained in the schema, which represent the complete relationships among the various attributes and the fragments. The schema in GURU DDBMS directly helps in locating any kind of relationships and the fragments. The fragments can be replicated, partially replicated or can be relocatable to any terminal node and DBMS. The size of these fragments depends mostly on the requirements given by the user. These requirements may be area based or on the basis of some other criterion selected by the user. The replicated and the partially replicated fragments are selected by the system required by the different terminal node users. The complexity in providing the access to these fragments on the demand of the users increases if, the fragment is not lying on the current terminal node. Secondly, if the fragment lies on the current terminal node but its portion is lying on some other terminal node. This increases the complexity since, the system follows a longer path to reply the query. Therefore, the system provides tuning periodically to reduce the complexity. Certain fragments which have the higher frequency of access can be replicated and the copy is brought to the central place from where most of the queries are routed, requiring to access the fragment.

Distributed fragments are designed to cater the needs of the distributed database processing, to increase the site autonomy, access efficiency, system availability, concurrency of the operations and reliability of the system. The following is the design methodology adopted for the various kinds of the fragments design:

6.6.1 Optimal Fragment Allocation

In GURU DDBMS, the extra cost calculation and to handle the subtle conditions of varying frequencies of access requirements, an intelligent system is designed. Which provides the cost in addition to the minimum cost calculated previously. This criterion is mostly based on the past experience with the system, for the more frequently used queries. An intelligent system using heuristics is shown in Fig. 5.9, which provides the additional cost criterion of the system based on the different DBMS's performance. Further, fragment access schemes are divided into two categories, one is time multiplexing, in this scheme the fragment for read and update operation is the same fragment but, the read and update operations are time multiplexed. The other scheme is the fragment multiplexing, in this case there are separate fragments for read and update operations. The updated fragments are copied back periodically, to provide latest data. The second scheme is more popular and is applicable to GURU DDBMS.

The cost calculations are worked out periodically, for the optimal fragment allocation. Before, looking to record the important parameters for the fragment allocation criterion, the situation is analyzed. There has to be some record maintained periodically, about the terminal nodes, taking part in the cur-

TERMINAL
CODE GLOBAL CHART

TERMINAL 1	
TERMINAL 2	---
TERMINAL 3	--- ---

SCHEMA NO.		OBJECT ATTRIBUTE	FRAGMENT QUALIFIER	FREQUENCY
SCHEMA-1	ATTRIBUTE-1	QUALIFIER-1		5
		QUALIFIER-2		7
SCHEMA-1	ATTRIBUTE-2	QUALIFIER-1		30
		QUALIFIER-2		100
		QUALIFIER-3		201
SCHEMA-1	ATTRIBUTE-3	QUALIFIER-1		105
		QUALIFIER-2		1001
SCHEMA-1	ATTRIBUTE-4	QUALIFIER-1		11
SCHEMA-2	---	---		---

FRAG-1		FRAG-2	
T ₁	1	T ₁	1
T ₂	3	T ₂	0
T ₃	3	T ₃	1
T ₄	0	T ₄	3
T _n	0	T _n	0

FIG. 6.1 GLOBAL CATALOG (DATA BASE)

rent application are alive or not. This is recorded in the catalog maintained in the system. Each and every terminal node connected with the network and taking active part in the heterogeneous environment are requested to send their identity to the terminal nodes, requiring the terminal node for some applications. This is done to maintain the record of alive nodes. In the situation when some of the intraqueries are routed to these nodes, if these nodes are not alive or are in the hangup state due to some reasons, the intraquery would not be allowed to be routed to such terminal nodes. Some times it happens that some of the terminal nodes are busy with some important work being carried out with them and the owner of the terminal node does not want any disturbance in the performance of the terminal node, that terminal node will not send the alive message. Therefore, the work would not be distributed to that terminal node by the other terminal nodes.

A continuous evaluation record is shown in Fig. 6.1, which has been maintained at each and every terminal node having information about the global schemas, for some applications on the current network. This record is maintained in some special heterogeneous database like, the other heterogeneous databases are maintained in the global system. This database can also be distributed, when required on the network. The chart shown consists of the details about the terminal node, application name, global schema name, object/attribute name, fragment qualifier to which an object or an attribute belongs to, and the record of all the terminal nodes having accessed the fragments in the past fixed time slot decided by the global systems, to which the object/attribute belongs to, and the total accesses required over a fragment. This table is analyzed by the statistical unit connected with the system which is in direct control of the server.

The statistical unit tries to locate the hot zones, where the maximum queries require a fragment. The performance of the system can be improved if a copy of the fragment can be located to the terminal centrally placed in the hot zone. If the maximum queries are read only queries, the performance would be increased significantly. If the update frequencies are high, the performance after placing a copy to that terminal may not be improved, due to more updates required to be done in all the copies, of that fragment as is discussed under ROWA scheme.

6.7. GSQL INSTRUCTIONS

The GSQL instructions used to reduce complexity and recover the data from the accidental losses are discussed subsequently.

The following instruction restores one or more database fragments (flat files):

```
rollback [{frag-name | alias-name}];
```

The rollback instruction is used within the *begin transaction...end transaction* structure to restore a database to its original state prior to the *begin transaction* instruction. *Begin transaction* starts a transaction; after this instruction - and until an user issues a corresponding *end transaction* instruction - system keeps track of database changes in a special log file, stored on the disk as *gtlog.log* (GURU Transaction Log).

The tuples can be added and updated using instructions such as *append*, *edit*, *change*, *browse*, *replace* and *update*. The tuples can be marked as deleted using the instruction *delete*. All

these operations are recorded in the log file. A subsequent *rollback* instruction instruct the system to use log file to restore the contents and the status of the original database. While a transaction is still in effect - that is, before the *end transaction* instruction has been issued. The *rollback* can be executed only as a single word instruction without a parameter.

At the time of some accidental crash in the system, it takes restart and when a new session is started, the system displays the following warning message with a beep:

Uncompleted Transaction found in frag-name

The user subsequently can issue an instruction *rollback* with the parameter of the database fragment name, and the system will restart the database to its pretransaction contents. The *rollback* instruction could be issued without a parameter to restore all database changes that were recorded during the last transaction. The *frag-name* is the name of the database fragment and *alias-name* is the name of the work area where fragment is lying.

The following instruction commits the changes made during the last transaction:

```
commit;
```

The changes are performed after the last *begin transaction* or last *end transaction* issued are written finally in the database. The changes made are by the instruction *edit*, *change*, *append*, *insert*, *delete*, and *update* etc. After using this instruction the changes made are irreversible.

The following instruction always commits an instruction or make it reversible:

```
set autocommit {on | off};
```

The changes made are irreversible, if *on* clause is used otherwise, it can be governed with *rollback* instruction as discussed previously, with the *off* clause.

The following instructions prepare a transaction block:

```
begin transaction [path-name];  
[GSQL instructions]  
[rollback];  
[GSQL instructions]  
end transaction;
```

When *begin transaction* and *end transaction* with a *rollback* are used, this sets up a transaction. This provides a option of changing a database and then restoring a database to its original state.

The *begin transaction* operation starts the operation. After this instruction is issued - and until a corresponding instruction *end transaction* is found - system keeps track of database changes in the system log file. The tuples can be edited using instructions *edit*, *browse*, *change*, *update* and *replace*. The tuples can be appended and by using instruction *delete*, they can be invoked for the deletion. All these operations will be searched in the log file.

If the *path-name* is given the log file would be traced into the give path directory. A subsequent instruction *rollback* instructs the system to use the log file to restore the contents and status of the original database. The instruction *end transaction* finishes the transaction, makes the changes permanently and log file is marked accordingly.

The following instruction places a query along with the required format for dynamic schema:

```
select [all | distinct] {expression-list | *}  
[into mem-var-list]  
from frag-name-list  
[where cond-statement]  
[group by expression-list having cond-statement]  
[{union | intersection | minus } select-clause]  
[order by attribute [asc | desc], attribute [asc | desc]...]  
[for update of expression-list]  
[save to temp frag-name [expression-list] [keep]];
```

Select is a powerful GSQL instruction that extracts, combines, and/or calculates data from one or more database fragments or views. The output from select, called a result fragment, can be displayed on the screen, or it can be a source to some other GSQL instruction. Several GSQL instructions take select instructions as subset clauses, including create view, insert, declare and select instruction itself.

In complexity and scope of purpose, the select instruction is a kind of sublanguage in itself. The select clause provides a list of expressions that will be included in the result-

ant fragment. The default keyword *all* specifies that all selected records will appear in the fragment; in contrast, the *distinct* keyword results in a fragment that contains no duplicate tuples for the listed attributes. The asterisk symbol (*) is used to select all attributes from a given fragment or view. The various GSQL functions such as *avg()*, *count()*, *min()*, *max()*, and *sum()* could also be used to compute statistics from the values in the specified attributes. In combination with *group by* clause, the same functions return statistics about group of tuples.

The Optional *into* clause lists memory variables in which the values of the resultant fragment would be stored. The clause is available only *select* instructions that are embedded in GSQL programs, and it is typically used when the resultant fragment consists of only one tuple. (If the fragment consists of multiple tuples then only the first tuple is stored in the memory variables) Only the *select*, *from* and *where* clauses are used with *select* instruction that includes an *into* clause.

The required *from* clause lists one or more fragments and views from which data is extracted for the result fragment. In this clause the temporary variable can also be assigned as *alias* names for each fragment and view;

frag-name-1 alias1, frag-name-2 alias1,...

These alias names can then be used in subsequent clauses of the *select* instruction. A *from* clause that lists more than one fragment results in a *join* operation that combines data from the fragments listed. Potentially, a *join* operation combines each tuple of each fragment with each tuple of second table, resulting in an abnormally large result fragment. The *where*

clause is used typically to select a subset of tuples from among all these combinations.

The optional *where* clause expresses a condition for selecting the tuples of data that will make the result fragment. The GSQL logical operators *.and.*, *.or.*, and *.not.* for building compound conditional expressions. In addition GSQL predicates *between*, *in*, and *like* are available for special condition. *Between* tests to see if a give value is between two expressed values. *In* tests a value for membership in a list of values. *Like* provides an skeleton string with wildcard characters for testing string values.

The optional *group by* clause produces a representative tuple for each group of tuples. A group consists of all tuples with identical data entries for a specified attribute. Each attribute name in the *select* clause must also be listed in *group by* clause; the *select* clause, however, can also contain aggregate functions that execute statistical calculations on the data on each group. In addition, the optional *having* clause could be included with the *group by* clause; *having* expresses a condition that each representative group tuple must meet to be included in the result fragment.

The optional *union*, *intersection*, and *minus* clauses are always followed by a *subselect* clause and results into union, intersection and minus operations on two fragments. For the successful operation each component fragment must have the same number of attributes - and corresponding attributes in the component fragments must match in type and width. The result fragment consists only unique tuples; the duplicates are eliminated.

The optional *order by* clause specifies the order of tuples in the result fragment. The one or more attribute names can be specified to fix up the order - the attribute is the primary key in the resulting sort, the second attribute is the secondary key, and so on. An attribute can also be identified by an integer that represents the position of the column in the result fragment. Finally, each column name and integer can be followed by the keyword *asc* (ascending order, the default condition) or *desc* (for descending order).

The optional *for update of* clause is used in the subselect statement of *declare cursor* instruction. Thus clause lists cursor columns that can subsequently be updated. *For update of* excludes the use of *into*, *order by*, and *save to temp* clauses.

The optional *save to temp* clause saves the result fragment as a temporary fragment that can be recorded during the current GSQL session. In addition, the optional keyword *keep* saves the result fragment on disk as a GURU database file. *Save to temp* excludes the use of the *for update or* or *into* clauses.

The following instruction sorts a database fragment:

```
sort [scope] on attribute-list-order
    [ascending | descending]
    [for cond-expression] [while cond-expression]
    to frag-name;
```

The *sort* instruction creates a sorted copy of the current database fragment and stores the result on the disk as a new database fragment. The *on* clause specifies a list of key attributes (or a single attribute) upon which the sort is execut-

ed. Character, numeric, logical and date attributes can serve as keys to the sort. The first attribute in the list is the primary key, the second is the secondary key, and so on.

Each attribute listed on the *on* clause can include an optional code to specify the sorting order, in this format:

```
attribute-name[/order]
```

The *orders* are represented by letters. The sorting *orders* are:

/a	Ascending
/d	Descending
/c	Alphabetic case ignored
/ac or /ca	Ascending, case ignored
/dc or /cd	Descending, case ignored

The optional keywords *ascending* and *descending* apply to any attributes that do not have their own orders. By default sort arranges the tuples in ascending order, uppercase first (ASCII collating sequence).

The *to* clause specifies the name of the new fragment (flat-file) that is to be created on disk. The GURU assigns the default extension name *dbf* (database fragment). The optional *scope* and *cond-expression* clauses to specify a subset of tuples to be included in the sorted version of the database fragment. Without these clauses the entire database fragment is sorted. At the same time when a new fragment is opened it is recorded in the catalog.

The following instructions define methods (procedures and functions) used with GSQL:

```
procedure proc-name [of object-class-name];
```

A *procedure* can be declared inside and outside of objects or classes. A *procedure* is a structured block of statements that execute a defined task. The *procedure* instruction identifies the block as a procedure and provides a name. The optional *of* clause follows with a name *object-class-name* of the class or object to which the procedure belong to. A procedure is executed by *do* instruction, which calls the procedure by name. After the procedure instruction, a procedure consists of:

1. An optional *parameters* instruction specifying the number of parameters that will be required for an execution of the procedure. The optional *with* clause in the *do* statement sends parameter values to the procedures.
2. Any number of instructions that define the actions of a procedure.
3. A *return* instruction that marks the end of the procedure and sends control back to the calling program.

The names of procedures can contain up to 30 characters, always beginning with a letter. Subsequent characters can be letters, digits or underscore(_) characters. A procedure that is part of an open procedure file can be called from the GURU shell or from within a program. Several external procedures and functions can be included in the procedure using a *include* instruction. These are called directly from the GURU shell and

included in the procedure. A procedure can also appear within a large program file, in which case the program can call the procedure any time during the execution. The procedures can also be called recursively.

Functions are defined as bellow:

```
function func-name [parameter mem-var-list]
[of object-class-name];
GSQL instructions
return val-returned;
```

A user-defined function is a routine that can accept a defined number of arguments and returns a single value as its result. The *function* identifies a routine as a user-defined *function* and identifies the name of the function. After the *function* instruction, a *function* definition consists of:

1. An optional *parameters* instruction, specifying the number and types of arguments that will be required in a call to the function.
2. Any number of instructions that define the action of the function.
3. A *return* instruction that provides the function's return value.

The optional clause *of* follows with a name *object-class-name* of the class or object to which the function belong to. A call to user defined *function* always appears as part of an expression or statement that uses the value that function

returns. Such a call appears in the same format as calls to any of the built-in GSQL functions: The name of a function is followed by a list of argument values, enclosed in parentheses. For a user defined function that does not require arguments, the function name must be followed by a pair of empty parentheses.

The name of the user-defined *function* can contain up to 30 characters, always beginning with a letter. Subsequent characters can be letters, digits or underscore characters. A function that is a part of an open procedure file can also be called from the GURU shell or from a program. A function can appear within a larger program file, in which case the program can call the function any time during execution.

The following instruction defines the global memory variables:

```
public mem-var-list
```

The *public* declares one or more memory variables or array-variables as global. Any procedure or function in a program can access or changed the value stored in a global variable.

Variables declared as *public* are also retained in memory after a program's execution is complete. Later, the values of these variables can be examined from the GURU shell.

The following instruction defines local memory variables:

```
private {all | mem-var-list | all like skeleton |  
all except skeleton}
```

The *private* instruction defines one or more *memory variables* as local to a given procedure, even when the same procedure names would normally be available to a procedure from another level of the program. The *memory variables* listed in the *private* instruction can therefore have the same names as *global memory variables* used elsewhere in the program, without interfacing with the values of those *global memory variables*.

Private values are available only to the procedure in which they are declared and are released from memory as soon as that procedure relinquishes control. The names are selected in the following ways:

1. A list of variable names
2. An *all* clause, specifying all currently defined memory variables.
3. An *all like* clause, using the * and ? wildcard characters to specify memory variable names with common elements.
4. An *all except* clause, also using wild card characters, but this time it specifies variable names that are to be excluded from the *private* list.

The following instruction saves the memory variables and arrays to a disk file:

```
save to file-name [{all like skeleton | all except  
skeleton}]
```

The *save* instruction stores defined variables and arrays - and their current values in a memory file on a disk. The values stored in the memory file then be loaded back into memory using

the `restore` instruction. Unless one specifies otherwise, `save` create a file with the extension name `mem`.

By default, `save` stores all currently defined variables in the memory file. However, a user can define an optional `all like` or `all except` clause to select only certain group of variables for inclusion in, or exclusion from the file. These clauses can use any combination of the two wildcard characters, `*` and `?`. The asterisk stands for any number of characters, and the question mark stands for a single character.

The following instructions send text to the screen or the printer:

```
text
[lines of text]
endtext
```

The `text...endtext` block is a convenient tool for outputting several sequential lines of text from a program to the screen or the printer. It takes the place of the series of `?` instructions. All the lines of text located between `text` and `endtext` are sent to the current output device.

The following instructions control the printing operations:

```
printjob
instructions
endprintjob
```

The `printjob` and `endprintjob` keywords enclose a block

of instructions that represent a particular printing operation. The behavior of this structure depends specially on the settings assigned by the user - in advance - to several of GURU environment variables. Most importantly, the *printjob* structure reacts to these values as bellow:

1. The *_num_cop* variable contains an integer specifying the number of printed copies that the *printjob* structure will produce. In other words, the instructions located between *printjob* and *endprintjob* are executed *_num_cop* times.

2. The *_cond_eject* variable indicates when an eject should be executed during the *printjob*. This variable consists of code for the following values:

- before*: Eject will be executed before each copy of the *printjob*.

- after*: Eject will be executed after each copy of the *printjob*.

- both*: Eject will be executed both before and after each copy of the *printjob*.

- none*: No eject will be executed.

3. The *_con_before* provides special codes that the *printjob* instruction will send to the printer before each page of the print job.

4. The *_con_after* provides codes that the *endprintjob* will send to the printer after each page of the print job.

The following instruction displays a list of objects currently stored in memory:

```
display memory [ to {printer | file file-name}]
```

The *display memory* instruction provides information about the following definitions stored in memory:

1. Memory variable, both private and public
2. declare arrays
3. Windows
4. Pop-up menus, bar memos and pads
5. System memory variables

If the display takes up more than one screen, system pauses the operation at the end of each screen until a key is pressed to continue. The *to printer* option sends the information to the printer. The *to file* clause stores the list in a text file on disk, with a default extension name *txt*.

The following instruction displays the information about the users on the network with the current system:

```
display users
```

The *display users* instruction displayed all the users logged in virtually in the system for the global application domains. This instruction provides the complete information regarding the users and the applications used by them.

The following instruction displays the complete database structure loaded in current area of the memory:

```
display structure [in work-area] [to {printer | file  
file-name}]
```

The *display structure* displays the fragments structures with respect the currently selected database fragment or of the database fragment with related fragments selected by a user with *in* clause.

If the *display* takes up more than one screen, the *display structure* pauses at the end of each screen until a user hits a key to continue. The *to printer* option sends the information to the printer. The *to file* clause stores the information in a text file on disk, with a default extension name of *txt*.

The following instruction processes the control structure to process tuples of a database fragments:

```
scan[scope][for cond-expression][while cond-expression]  
GSQL instructions  
[loop]  
[exit]  
endscan
```

Scan...endscan is a structure that loops through the records of the current database. By default, *scan* begins its activities with the first tuple in the database (or the first tuple in the indexed order) and proceeds tuple by tuple through the entire database. All the instructions located between the *scan* and *endscan* are executed for each tuple.

If the *scan* and *endscan* is not required to be processed for the entire database, a *scope* or *condition* clause or both can be included to restrict the action to a subset of tuples. The *all scope* clause is the default. Next selects a specified number of

tuples, starting from the current one. *Rest* selects the remaining tuples, moving forward from the current one. *Record* specifies a single tuple by number (an unlikely scope for the *scan* structure, which is normally used to process multiple tuples).

The *for* condition clause works through the database from the beginning to end; a given tuple is included in the processing only if the *for* condition evaluates to true for the tuple. In contrast, the *while* clause starts with the current tuple and processes each record that follows until it encounters a tuple for which the specified condition evaluates to false.

6.8 CONCLUSION

The heterogeneous environment is complex in the way that, many DBMSs which belong to different areas and working environments are connected in a global integrated environment and work for different kinds of applications. The applications may not be included in the domain for which the DBMSs are designed. Therefore, they meet most of the kinds of conflicting languages, data and the structural and behavioral constructs, which becomes a tedious task for any system, to handle it properly. A few methods suggested using the intensional and extensional notions with triggers are effective to check the complexity and help in reducing it.

The different schemes suggested to implement better fragmentation criterion are based on the requirements of the particular application, which help in improving the performance considerably. The fragment are allocated with the criterion of hot points and the involvement of the complexity of different DDBMSs. This helps in boosting the performance and reducing the

complexity considerably. The method includes the coding scheme for the current alive terminal nodes and the speed code corresponding to the application efficiency. This scheme appeals very much in the selection of the right terminal.

The complexity can be reduced by intelligent procedures using triggers. The instruction *select* helps in reducing the complexity with the proper planning of attribute placements. Proper index loading mechanism in the system with appropriate grouping of fragments makes the system better operated. The intelligent schema and proper fragment loading schemes with clearly defined fragment qualifiers are making the system, materialize queries efficiently. The proper instructions to form transaction blocks, such as *begin transaction* and *end transaction with rollback* and *commit* instructions save data from the accidental damages. Proper allocation of the fragments at the different terminal nodes reduces the access time hence reducing complexity.

The all discussed methods help in reducing the complexity to a great extent and rendering the system to achieve good efficiency.

A sample program using GSQL is shown in Appendix - A.

The data structures for the GSQL instructions are given in Appendix - B. The first and the next reserved words are given Appendix - C and Appendix - D respectively.

MESSAGE-TASK SCHEDULING

7.1 INTRODUCTION

In Message Based Communication System (MBCS), the messages are communicated in the form of separate packets, designed for the purpose of the communication network being used for the distributed environment. The packets are later assembled and form the complete message. In this system, maximum communication efficiency for the packets is maintained from the network view point. In Distributed Shared Memory System (DSMS), the memory being distributed is converted in some fixed size of pages and these pages are sent to the destination. Later, these pages are translated in the local environment and are accessed as data for the local application. This method is inefficient over MBCS due to the limitation of fixed page size and the handling of the DSMS needs a separate layer between application and a message passing system.

The message-task is sent in the form of packets. The size of the packets depends on the local communication network used to connect the distributed environment. These packets are later assembled to form the complete message-task. The assembled packet is having the shape of the distributed virtual memory segment comprising a new task to be assigned to the Task-scheduler. If the task is a heterogeneous one, it is translated in the local system's language, which is recognized from the type field in the message packet. Later, this task is executed as a virtual shared memory segment at the local terminal node, to perform the actual functions. Application programmer understands well about

the abstraction, since, the access protocol is such to access data sequentially. This communication process remains totally transparent to the application programmer, working on the system. Therefore, application programmers need not be conscious about the data movement between processes and the complex data structures, which can be passed by reference.

There is a wide need of passing messages to the various objects in the object based systems [11],[26],[60]. These messages play an important role for handling objects in a specific manner. The objects can access data for the procedures inside the object and a kind of the messages can alter or change the structure of the existing objects in a required manner [180]. The messages changing the structure of the object are known as structural messages. The structural changes of the objects could be brought to a limited level, as per the plans laid out earlier in the object handling system [129],[168]. Since, the limitations of the system are that the objects can only be changed as per the prior assigned schedule, none of the later changes in the objects could be performed by the application programmers in the way other than the prescribed ones. This limitation exists in the system due the fact that the changes made in the system could adversely affect the integrity of the objects. The remaining sections discuss about the message-tasks and their shedding.

7.2 OBJECTS AND MESSAGES

Object-oriented database systems are superior to conventional tuple or record-oriented database systems in their ability to handle the database with unique way of their data representation and maintenance of the privacy and security of the data. The conventional systems consisting of tuples or record-

oriented database, represent the unit of information. This record and entity based system poses a great difficulty to the application programmers, due to the fact of understanding all the aspects at the same time. This leads to more time and insecure operations performed by the application programmer. Object-oriented databases consist of only a set of objects and classes, which ultimately correspond to some entity in the real world. The ability to maintain hierarchies in the object-oriented databases makes the development of the system more systematic and simple.

Objects can be categorized in the different categories and can be linked in a manner to form an object hierarchy. Different objects connected in the hierarchy have certain properties. All these properties of the above nodes are inherited in the nodes bellow in the hierarchy. These objects are given different object identifiers, due to which they can be identified in the system. The data and the procedures consisting in an object can be accessed at an appropriate level in the object hierarchy. The privacy and security of the data and procedures can also be maintained in the objects based on the assigned codes by the application programmers, to the different entities present in the object. Message-task can be adopted to tune the objects, change the structure of the objects and reschedule the object in the object hierarchy. In the object based system, the number of the objects grows to an extent that the main memory cannot record all of them. Therefore, a virtual object management scheme is suggested to record the objects in the virtual memory. Some of these objects are assigned by the application programmer specially to be recorded in the virtual memory. Therefore, the large databases and the large number of objects can easily be handled by the system. The system can also choose some of the objects to be directed to the virtual memory. But, the amount of thrashing

required to be controlled, which should be of a limited value, in the overall design provided by the application programmer.

The overall criterion is to have the fastest communication speed, to go through a rout, which takes the minimum cost. This cost factor depends upon the network structure used, which is the topology of the network, the heterogeneity of the medium and lastly the communication protocols used. The message-task is sent in the form of packets. The cost of synchronization for the packets being received and acknowledged should be minimum. At the time of acceptance test (AT) for the process as a task, it has to maintain the synchronization to have the serializability of the operations. After the acceptance test is successful, the exit procedure adopted takes time in the global environment and follows the specific procedure for the commitments to be done synchronously. In case, if the test is failed, all processes are abandoned and the previous states have to be reestablished through the rollback procedure. The another time taken is the cost of acceptance test, which is nothing but execution of the acceptance test procedure.

7.3 HETEROGENEOUS MESSAGE-TASK HANDLING POLICIES

7.3.1 Objects

A GURU database is a set of objects. Each object represents a certain entity in the real world. Each and every object is a different object than the other objects, if the names are different. Therefore, the different objects have the separate object identifiers. Objects in a database are divided into primitive and complex objects.

A primitive object keeps a particular value in a database like some prior defined value in GURU. Primitive objects belong to the prior defined classes or subclasses, known as structures in GURU. Functions and procedures are also defined as the primitive objects in GURU apart from the regular defined values. The defined procedures or functions, also known as methods are executable. These are defined to fetch and store the values, establish the relationships, presenting the view etc. for the objects.

A complex object is defined as the combination of some objects, defined earlier. The objects form a hierarchy, when defined at the different places using the other classes or/and the objects as the sub-objects. The objects form the complex object by associations in different ways with the component objects.

A structure also known as a class, is regarded as a set of objects. Which is used to share same data and properties defined to the other structures and objects. Each and every object belongs to one or more class (structure in GURU). A structure description maintains the definitions of other objects bound to the structure and the hierarchy of objects, methods for objects and the relationships to the other structures.

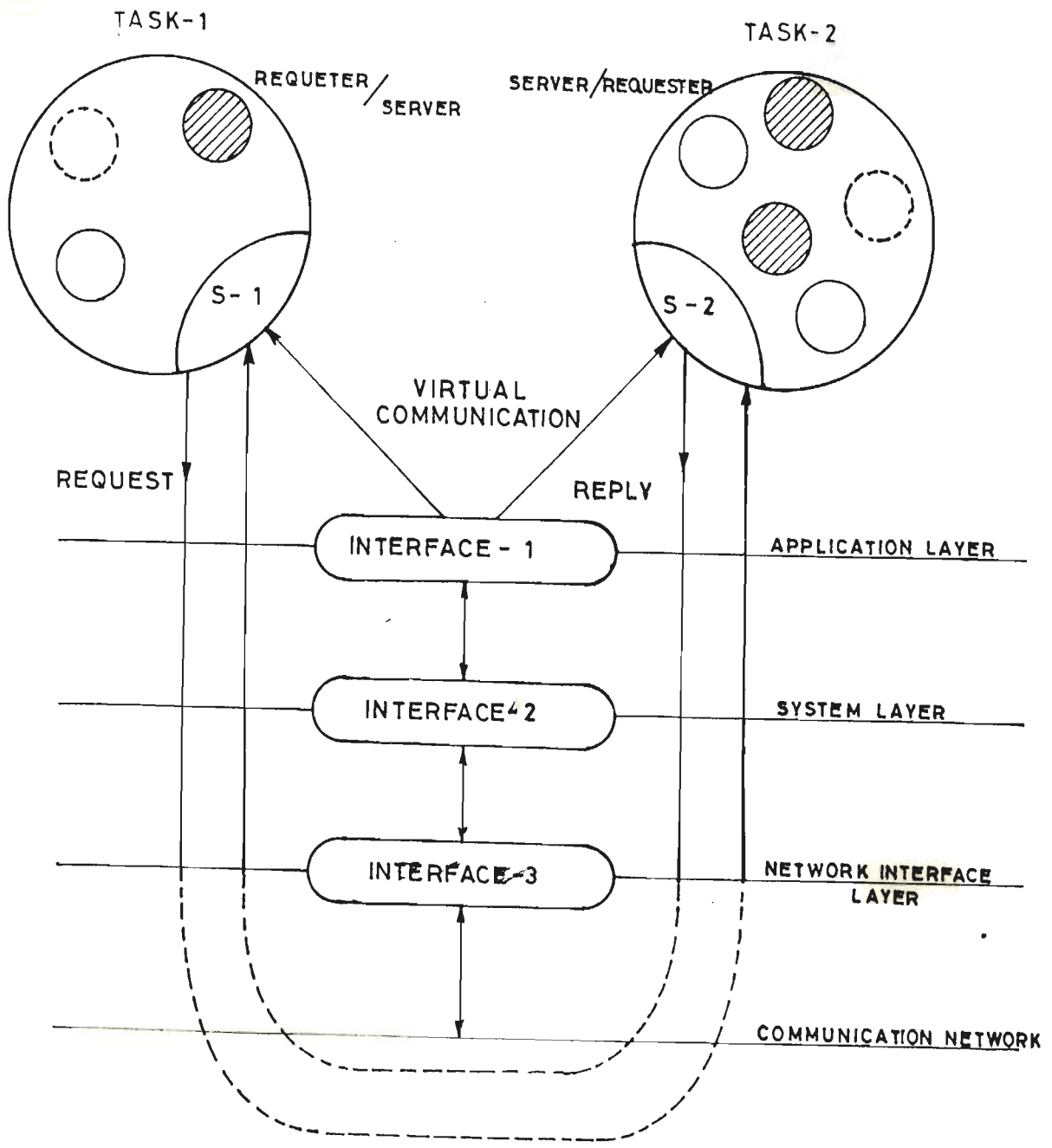
An object hierarchy is defined by complex and component object relationships. It defines the conceptual schema to manage the objects in a database. The object hierarchy is a union set of possible object structures, which objects in the class can take. The structure of each object can be changed by passing messages. The message which changes the structure of the object is called a structural message, and the derived object structure is known as

the aggregation hierarchy of the object.

At the time of creation of an object, one object supervisor is bound to the object. Which supervises all the activities of the object, the activities include the checking of access to objects, method (also known as procedure/function in GURU) calling, mapping the other objects, and the message-task scheduling activities etc. in GURU. The supervisor takes care of executing the object tasks by proper mapped functions, which include the memory variables, databases, procedures, and functions, of local and inherited classes and objects. Apart from this, supervisor contains the information of the state descriptions for all the objects in the object and the class hierarchy, the task synchronization constraints to be enforced, the regular scheduling to be performed, and the managerial policies regarding the local resources.

The supervisor spawn processes (active objects) after getting attached to an object to provide the service on request and to execute some procedure or function. The supervisors tied up with the objects, remain active till the life of the active objects (task). To process the outside requests, the supervisors remain at the front end. On request, the supervisors invoke the different procedures or functions and guarantee the integrity of the conflicting objects. The supervisors provide the message-task scheduling to the different messages coming, to be processed. The basic message-task structure in GURU is divided into following three categories:

- i. Application task
- ii. Network task
- iii. System task



GURU FUNCTIONAL LAYERS
FIG. 7.1

The tasks (shown in Fig. 7.1) required to process some category of application are known as application tasks. The application task is a complex task, where the different objects are linked and form a global data structure with relations. For example, these relations form a schema for some application for a database to be handled. The different entities belonging to different fragments are connected with some relation among them, hence, establishing some data structures to relate various tuples among them. The tuples belonging to the different fragments are linked in some specific order. These can be accessed later by the different users. The procedures accessing them can be associated separately by the application programmer. The other aspect is that, the procedure to access a given data item can be provided on request by the dedicated supervisor, made to provide such kind of services. Therefore, users are provided a facility to write their own procedures to certain operations and make them available to the object as a procedure or function, to perform the required operations. For example, some fast access procedures may be written by the users to access the database. These procedures could be some indexing methods, or some hashing mechanisms etc. for the database to be accessed.

The network task is the task provided to handle some applications not available locally with the terminal node. Therefore, such tasks are shipped to different terminal nodes for further processing. The shipping operation is performed through a network, connecting the current terminal node with the remote terminal node, to which the task is shipped for further execution at its end. This service provides the user tasks the illusion of all hardware and software resources residing on current terminal site, whereas, in reality they could be stored anywhere. The

network task managers at this level will create tasks to manage the interaction on the two sites for the user and server so that they need not know or be aware of the actual processes that provide the service. The detailed discussion about this is done later in this chapter.

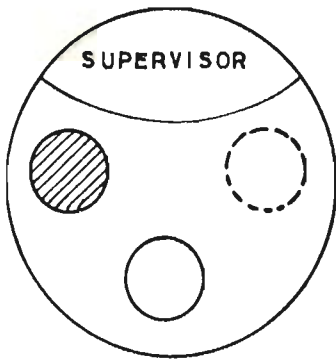
The last task is the System task, which makes use of the GURU system routines to handle the tasks. This task resides on each physical site of the computer terminal node or the individual computer, which is required to process it. This task is basically handled by the GURU system for the management of the rest of the processes to be coordinated to run on it. This task acquires the GURU's supervisor, which ultimately is responsible for the overall activities in the task handling mechanism. GURU is responsible for scheduling the tasks, resource management, checking the integrity of the tasks, further allowing the access procedures after verifying the access rights for the requesters, requesting the resources etc., but it must interface and react to the rest of the system just as the another task would react to its requests. The systems architecture is structured as a hierarchy, it consists of three layers: virtual machine layer, system support layer, communication layer. Virtual layer consists of the high level tasks available through the GURU's language GSQL. All the tasks can reside in any of the three layers. To provide the proper transparency to the users. All supervisors are given the unique names, through which they can be invoked.

Since, GURU works in a distributed environment, therefore, there is a need to provide the interaction with the inter-computer and intra-computer mechanisms to provide processes with control and information. The message passing mechanism is used for the inter-task communications. The procedures are called

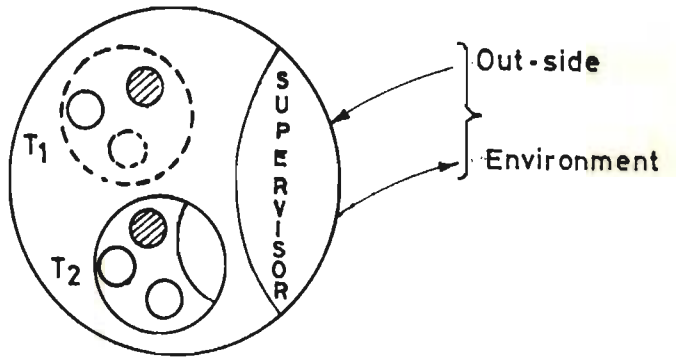
to invoke a task using the conventional protocols. All communications between the tasks are through messages. GURU provides the scheduling to these message-tasks and could be independently executed. The messages coming, form a shape of the process to be executed. This process is scheduled by the scheduler and the required actions are obtained. This facility helps in programming the objects intelligently in the entirely dynamic environment. The messages are formatted in a specific format, which helps in detecting the kind of the message. Later, the message could be detected as the pure data format, the control format or lastly the procedure format. The last procedure format tells the message-task needs scheduling. After the message-task gets properly scheduled, the supervisor takes up the responsibility of executing the operations performed by the control task. The control task carries the high level GSQL instructions made to provide certain directions to modify the object structure in the required format (the high level instructions of GSQL). This facility helps in keeping the GURU, better suitable in the heterogeneous environment. Which needs the structure of the required objects working with the other heterogeneous DDBMS to modify as per the requirements lying with the local DDBMS. This facility upgrades the requirements to cope up the basic deficiencies of other environments to match the existing local DDBMS environment.

7.3.2 Task Handling

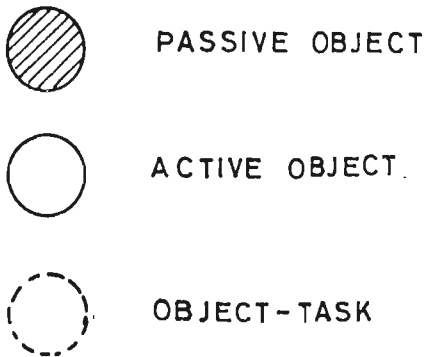
Tasks are defined as active objects in GURU. The message is also handled as an active object to be executed independently. All objects consist of some mailbox with each and every object, where the requests can be stored, when the objects are busy. Soon after, the object becomes free from its last activi-



(a)



(b)



TASKS IN GURU
FIG. 7. 2

ties, requests are processed from the mailbox. All requests consist of certain identity codes, which help in replying the requests with process identification and the class hierarchy to which an object belongs to. A requester wishes to perform some operations on another objects, which may require certain access to a database fragment bound with an object. The server object will provide the data from the database if the requester has the rights to access the data from the database. Moreover, if the granularity of operations is high, better would be the efficiency to process the overall task.

For handling the remote objects, the conversation from the remote objects will be required. This conversation needs messages to be exchanged. The mechanism to support this is the system support service called the messenger. A messenger is the task that provides the control of communications within the network for requesting and serving tasks. Messengers are available with each layer within the GURU architecture. These messengers provide the interface between adjacent layers and to the network communication subsystem. The messengers are themselves active objects, which take participation in message communication to the adjacent layers and these messages are stored in the corresponding mailboxes of the objects.

The unauthorized operations performed by the remote messages, over the objects can be controlled by the privacy and security codes, which are recorded in the object structures for the individual operations. The all messages are checked for the authorization codes, if some message does not have the right for the requested operation, then reply is sent accordingly.

An object structure is shown in Fig. 7.2, which comprises a local object supervisor, a few active and the passive

objects, and object tasks. Each object can have a number of sub-objects, which can act as separate tasks. The tasks maintain a class hierarchy.

7.3.3 Message-Task Conversation

The process of message conversation is shown in Fig. 7.2 (b). The various objects are sending the different messages, which could be executed as separate message-tasks. The conversation among the various objects can access the object's resources, these resources could be a few memory variables, databases, hardware equipment, software etc. Further, the messages may invoke some procedures or functions tied up with the objects. The front end processor of the object is the local object supervisor, which remains active all the time for the object.

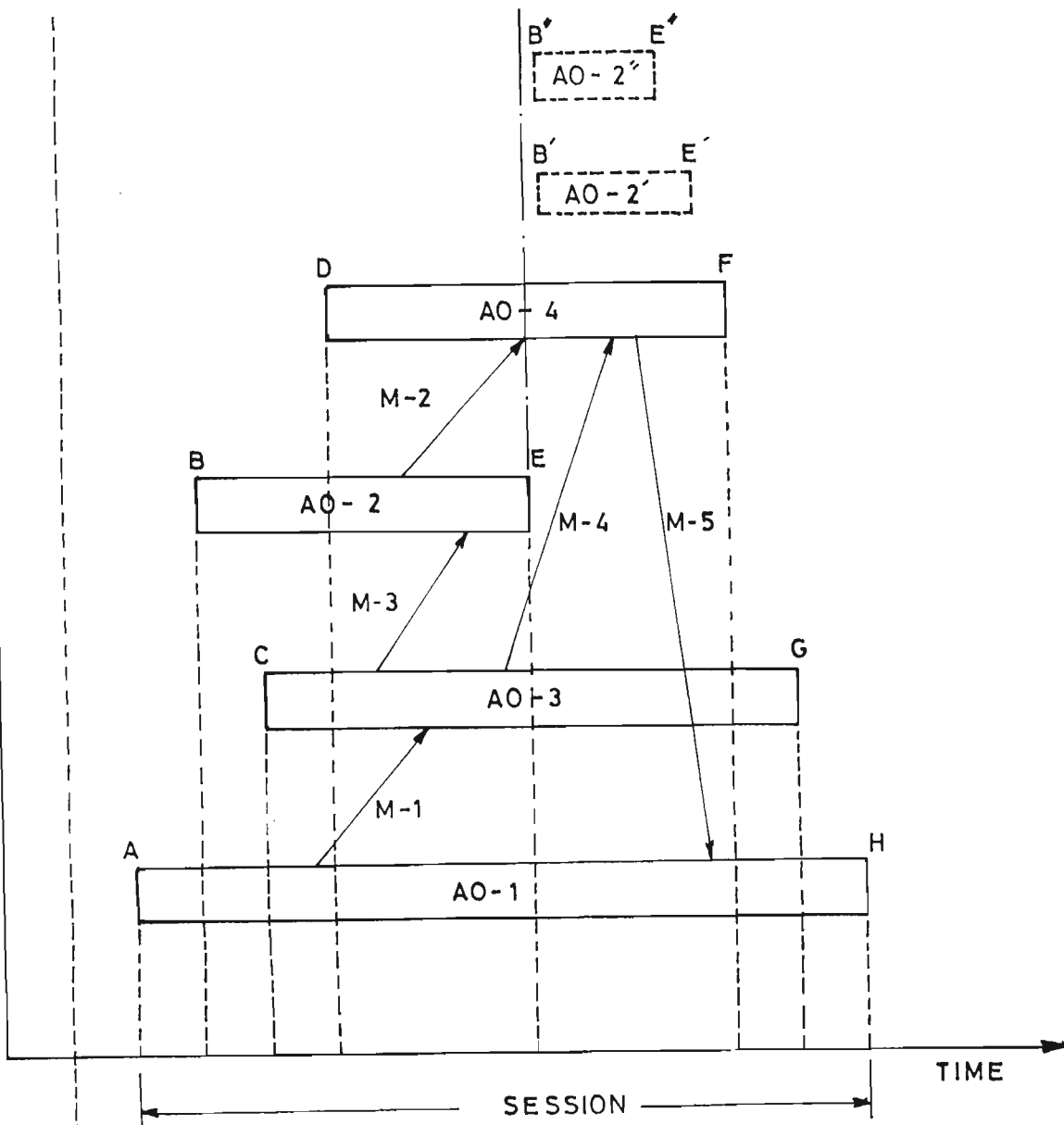
7.4 CONVERSATION POLICIES

In this section some brief description of conversation policies is presented. The two basic conversation methods synchronously exited and asynchronously exited with mailbox technique are discussed. The mailbox serves the basic inter-message communication task to reduce the overall service time of the processes. These two approaches differ in the way to provide the different system performances. A brief comparison of the two is presented in the end of this section.

7.4.1 Conversation Structure

A conversation is defined as the session established between two or more interacting processes (active objects), which can be recovered to their previous states, if required. To have

ESSES



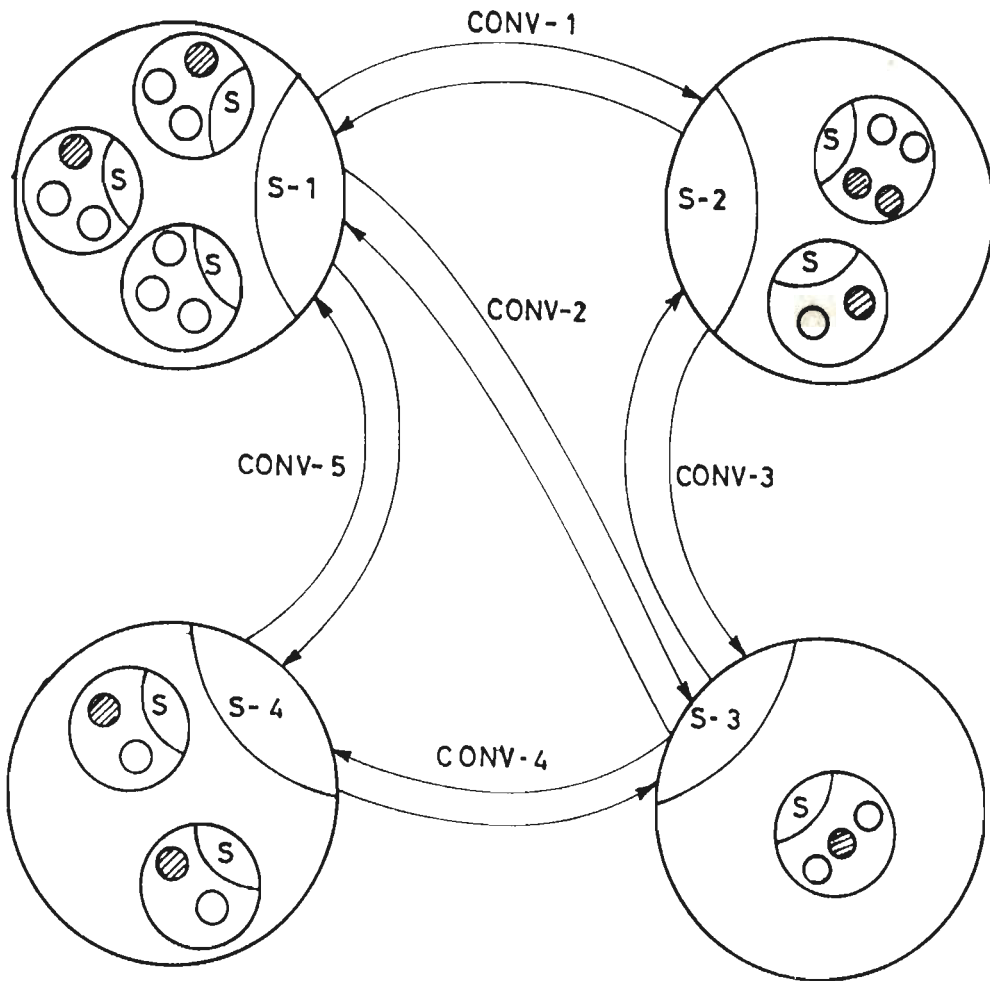
PROCESS
CHECK
POINT

A, B, C, D, B', B'' — PROCESS RECOVERY POINTS

E, F, G, H, E', E'' — POINTS OF ACCEPTANCE TEST

INTERACTING ACTIVE-OBJECTS (PROCESSES)

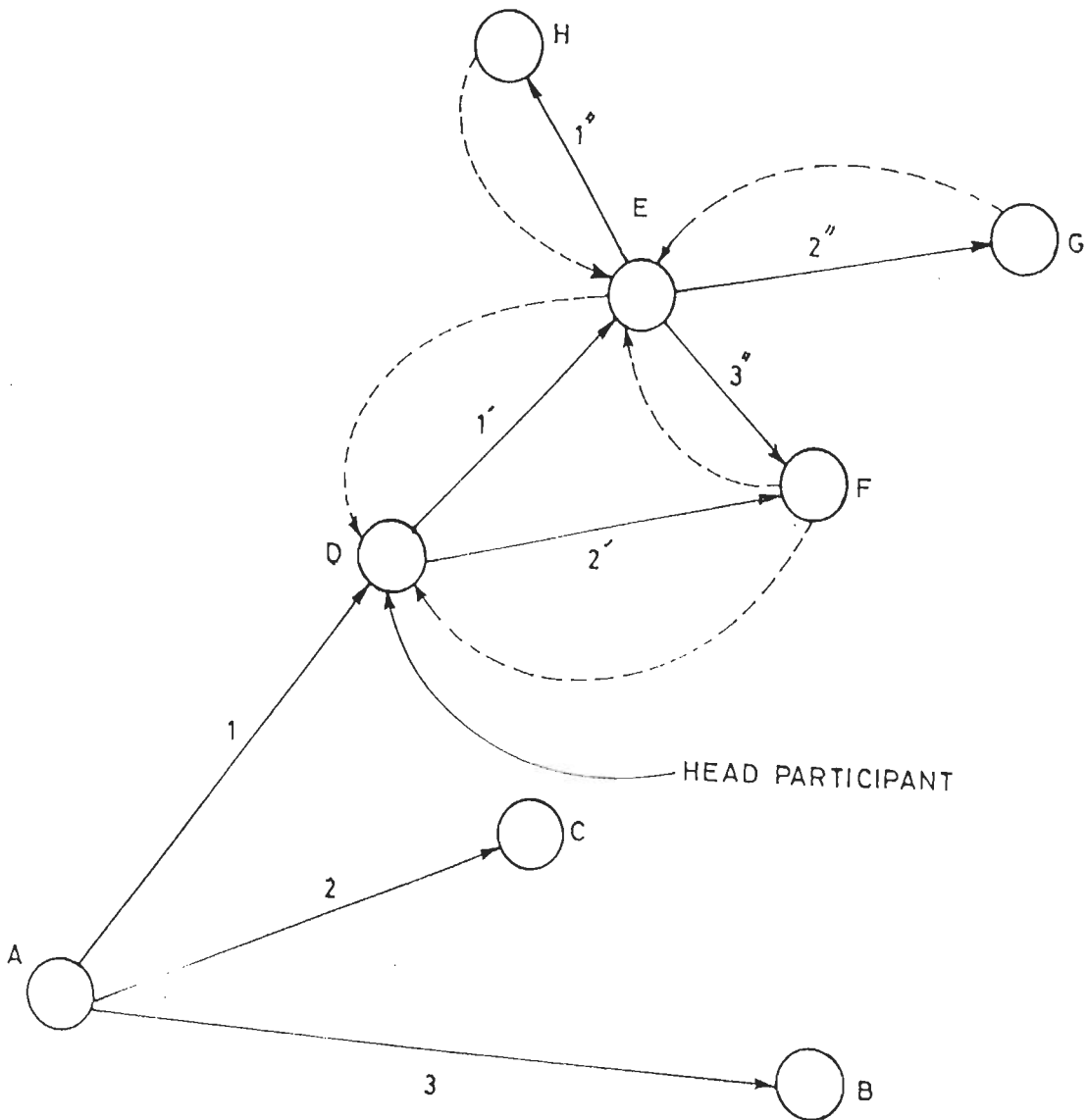
FIG. 7.3



OBJECT CONVERSATION
FIG.7.4

the conversation, some well defined rules are followed. None of the processes can violate these rules. A two dimensional sketch is shown in Fig. 7.3, consisting of a few processes interacting in time space. The X direction represents the process interaction time and the Y direction represents the distinct processes interacting among each other. The objects interacting with active objects (processes) are shown in Fig. 7.4. The active objects can sprout any number of active sub-objects. The sub-objects later called as active objects, can further take part in conversation. The process recovery points, A, B, C, D, B', B" are shown in Fig. 7.3, these points are maintained in Log, so that the recovery of any active object can be performed on the rollback operation. The process checkpoint, is also recorded in the Log, which helps in maintaining the complete previous state of all the operations performed after the check-point is established. Therefore, any dead-locks or system failures will not cause the loss of data. There are four active processes shown, which are taking part in a session. These four processes are marked as AO-1, AO-2, AO-3 and AO-4. The active object (object task) AO-2 finishes its execution and passes its acceptance test and exits at point E from the synchronous state and sprout in two other tasks using lookahead scheme as shown with the names AO-2' and AO-2". These two tasks establish synchronism with the previous sub-tasks AO-1, AO-3, AO-4, which belong to the same one task, AO originally. The conversation is established by passing messages among the active objects. The messages passed are recorded in the mailboxes of the objects.

The four tasks using the message-task scheme are shown in Fig. 7.4. The various messages communicated among them are CONV-1, CONV-2, CONV-3, CONV-4, and CONV-5. The messages sent and received by the different local supervisors s-1, s-2, s-3, and s-



DISTRIBUTED ACTIVE-OBJECT TASK GRAPH
 FIG. 7.5

4 of the active objects are shown. When an object is under the running state, it remains locked for all the outside requests made by the other active objects. The outside requests are stored in the mailbox of the object. The process (active object) needing the service of the other active object, waits till an another message is received regarding the required results of the request, placed earlier. The active object finishing the task checks the mailbox and services the requests from it. Each participant process consists of one or more try blocks or program blocks designed to produce the same or similar results, as well as an acceptance test which is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks.

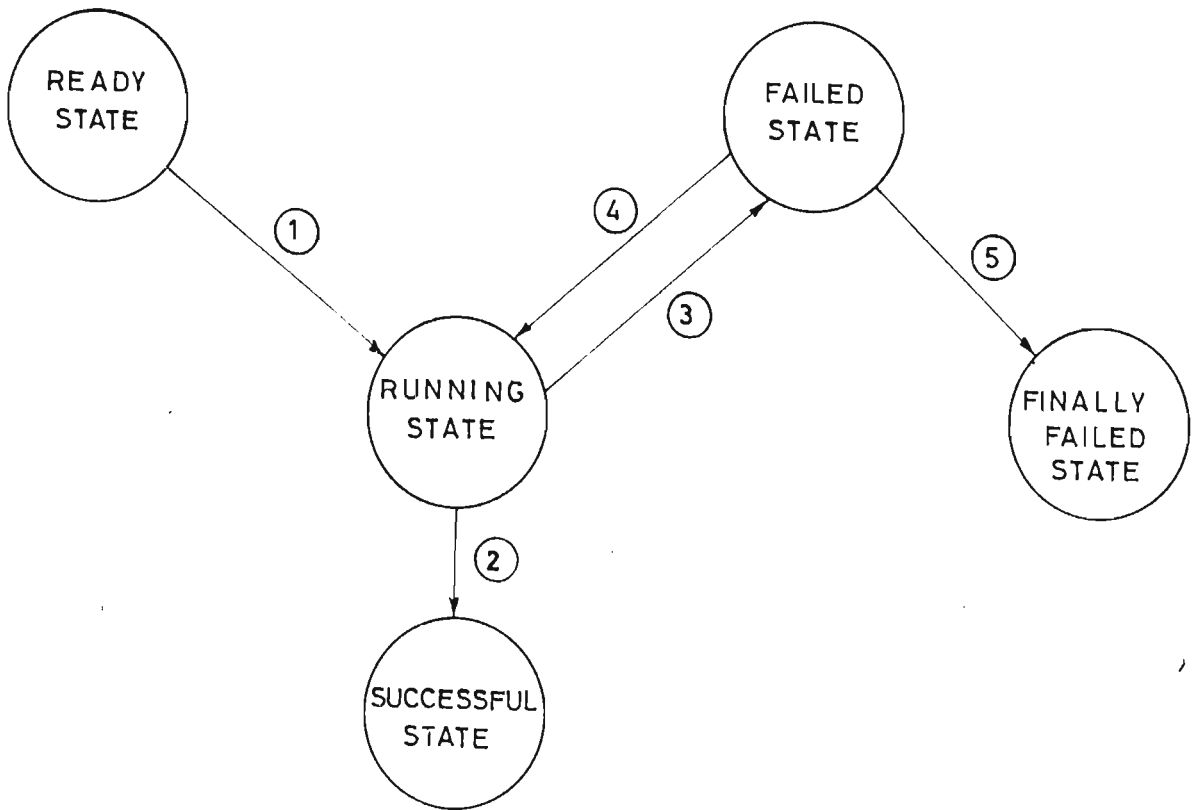
For example, refer Fig. 7.5, a query, A being executed at a local terminal node, needs the three fragments, B, C, D of a globally distributed database, a few fragments may be the remote fragments, therefore, query needs to be broken and formed in a shape of different sub-queries out of which the remote queries are to be distributed at the remote terminal nodes following the routs 1, 2 and 3 respectively. The query at fragment, D needs further two different fragments, E and F following the routs, 1' and 2' respectively. The fragment, E further needs the two fragments G and H following the routs 2" and 1" respectively. The results are processed and follow up the rout back to the fragment, E. The fragments E and F are processed and follow back to fragment, D, later, fragments B, C and D are processed and return back to requester terminal node for the query, A, where they can be processed to provide the required results, which are further communicated to the user through the local terminal node. These individual results are finally combined to provide the final result. The splitting of the query can be performed in a

number of ways therefore, the different sets (try blocks) of the main query can be generated on the requirement. The process recovery points are the individual recovery points for the sub-processes to be rollback to their previous states when any acceptance test of the sub-processes fails. The recovery points are mark of the processes before the processes started interaction. Any process taking part in conversation goes through an acceptance test, which succeeds or fails. On a success the other linked processes continue to execute. But, on failure another alternate set of processes is tried out. The alternate try blocks define an alternate interacting session.

A conversation succeeds only if all the sub-processes succeed in their individual acceptance tests. Therefore, the participants are allowed to leave the conversation session if all the participants have passed their acceptance tests. A set of processes finish their conversation session on successful acceptance test. The conversation occurs among only active processes.

7.4.2 Exit Procedures

All the processes enter the conversation session asynchronously but get synchronized themselves on their entry to the conversation session. The synchronization can add considerably to the time and the cost of the conversation scheme. The basic approach required to reduce time of synchronization is known as lookahead. This approach makes the exit of a participant on success of its acceptance test. This has the understanding that the other participants can fail in their acceptance test in future. Therefore, the rollback would be needed. This increases the overall cost of the recovery procedures but saves the syn-



SYNCHRONOUS EXIT STATE - DIAGRAM
FIG. 7.6

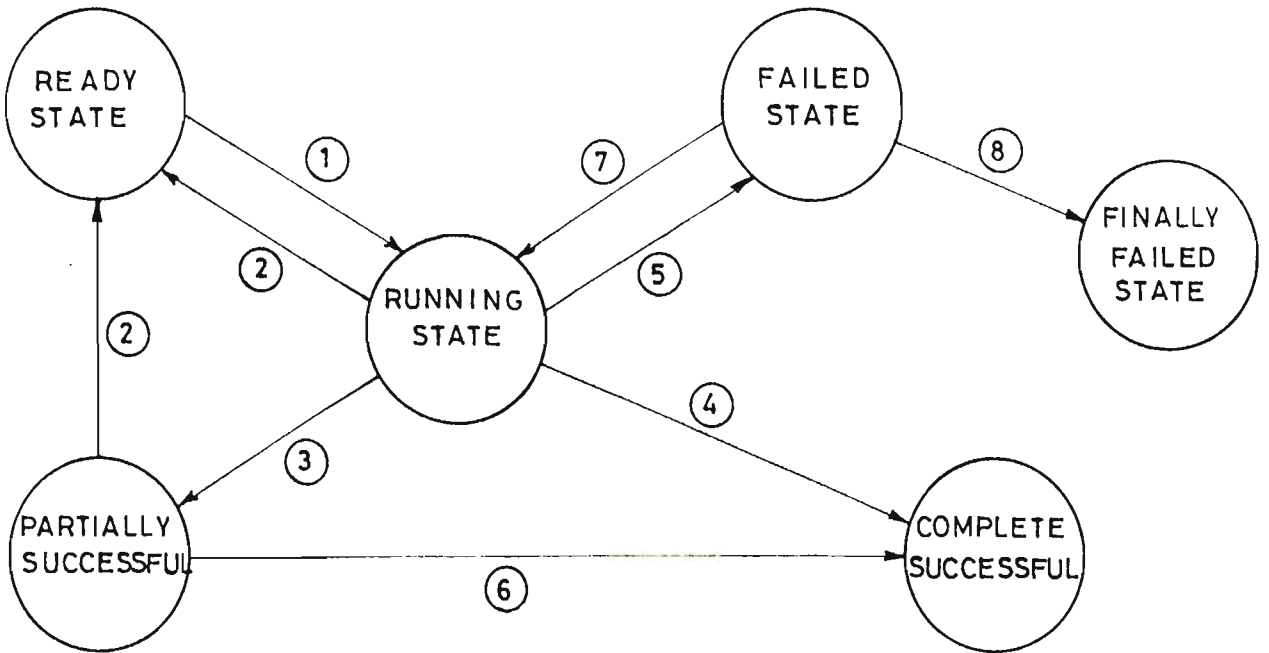
chronization overhead.

There is a choice of making the conversations beyond the unfinished conversation without any limits, but, one should limit the extent of lookahead conversation with respect to controlling the implementation complexity. Some limits of the lookahead conversations, where the data recovery is not possible should not be crossed. These limits are the critical irreversible limits. The conversations which belong to the lookahead category are known as asynchronous exited conversations and where lookahead is not allowed are known as synchronous exited conversations. A synchronous exited conversation is shown in Fig. 7.6, which passes through the following states:

- i. Blocked state
- ii. Running state
- iii. Successful state
- iv. Failed state
- v. Unsuccessful state

The blocked state represents the no conversation, running state provides at least one conversation, successful state provides the all participants have passed the acceptance test, the failed state provides that at least one participant is failed in the acceptance test, lastly the unsuccessful state provides that all alternate sets (try blocks) are failed in acceptance test.

The asynchronous exited conversation, a process exiting from a conversation session enters into an another conversation. If the second conversation however does not depend on the first one, the conversation should not exit finally, because in case of



ASYNCHRONOUS EXIT STATE-DIAGRAM
 FIG. 7.7

failure rollback operation is required to be started.

An asynchronous exited conversation is shown in Fig. 7.7. this has the following six states of operation:

- i. Blocked state
- ii. Running state
- iii. Partially successful
- iv. Successful state
- v. Failed state
- vi. Unsuccessful state

In the above list only one new state has been introduced, which is not there in synchronously exited conversation. This state is just to represent at least partial success of some participant, which exited and continued an another conversation. The system performance would increase with the asynchronous exits of the participants. The performance improvements would be particularly conspicuous, when the acceptance test failure probability is very low and the number of participants are large.

With the asynchronous exit conversation approach, the complete information needed for rollback (try block entry points) prior values for the external variables which have been changed after the process entered the conversation, should be kept until the conversation is completely successful. To maintain this information GURU provides a facility of local log, which maintains the complete information of all the participants in the session. All conversations are being maintained in the local log, this log maintains the dynamically created main memory object images. The object images are finally linked, till the

conversation is over or enters to the successful state. After the conversation, the removed active objects make the images also removed from the local log. This activity is performed with proper checkpoints in the local log. These checkpoints help in the data recovery. The conversation records are linked hierarchically to maintain the structure of the conversation process. The main memory variables being processed at the remote terminal nodes maintain the values to be sent for the remote terminal nodes. These values are recorded in the form of variable list and the values to be supplied to them on the log. The values are finally extracted from the logs along with the final results to be communicated to the remote terminal nodes and are supplied to the network layer manager of GURU. These values are sent to the remote terminal nodes in the form of message-task, where they were required.

7.4.3 Acceptance Test

The acceptance tests for the conversation is divided into basic three categories centralized, decentralized and semi-centralized.

Centralized acceptance test, provides the test procedure to only one participant known as head. And all other participants communicate with this participant by sending the results in the form of message-tasks. These results are processed at this node and the rollback or commit indications are sent to the other participants. The declaration of head can be done in two ways, static head or the dynamic head. The static head is prior defined head, which always execute the acceptance test. Where as a copy can be provided to some participants and the process executing over the last participant node can execute this procedure or any

other node out of the participant nodes can execute the acceptance test procedure. With the synchronous exit procedure, this is one of the same thing that any node can execute the acceptance test procedure. Since, the all participants are running with synchronism. The all participants are exited at the same time, and hence there is no advantage of getting the concurrency of the acceptance test procedure for the component participants. However, in the case of asynchronous exit procedure the dynamic placement of the head designation is efficient than the static head approach. This is due to the requirement of communicating some extra message in static head placement than the dynamic head placement policy.

Decentralized acceptance test, provides individually all the participants, to perform their individual acceptance tests. Further, the results are exchanged by the participants as are needed by the requester. In this case the communication cost is very high due to sharing of all results among all the participants and analyzing them. Secondly, the acceptance test procedure has to be divided to be executed at the remote terminal nodes.

Semi-centralized acceptance test, provides the compromise over the previous two methods. In this case all local acceptance tests are done as usual, as in the case of decentralized acceptance test procedure, but the head is only one to which the final results are communicated. This makes the communication burden little reduced. But, still, the acceptance test procedure requires to be divided as in the case of decentralized acceptance test procedure.

The above discussed all the three schemes have certain advantages and disadvantages among them. But, the semi-

centralized acceptance test procedure is the best among the three. Some other schemes declared by the users for reducing the overall complexity for the division of the acceptance test procedure and also the head placement policies can be programmed externally by the users. The one of the scheme is known as the Name-Linked Recovery Block (NLRB) scheme and the Abstract Data Type (ADT) conversation scheme. In case of NLRB scheme a set of procedure is illustrated, which is supplied to all the participant terminal nodes. After executing the given procedure for the different try blocks, provided by the external user, the terminal node will be found which runs the acceptance test for the participant. Therefore, there is no need to divide the acceptance test procedure and the performance could be improved.

7.4.4 NLRB Scheme

This scheme is the extension of the previous discussed schemes, one conversation identifier is associated in the recovery block, to identify the conversation. The different try blocks are tried out for the acceptance test. The set of name-linked RB's, each executed by a different process but having the same conversation identifier, compose a conversation construct. This scheme is advantageous in the large distributed environment.

7.4.5 Abstract Data Type (ADT) Scheme

This scheme was proposed as a remedy the shortcomings of the earlier described NLRB scheme. This is due to the nonuniform way of constituent recovery blocks in this scheme, blocks are organized in a structured form of abstract data type. The different processes are attached the conversation identifier. Which helps in identifying the conversations and presenting a

uniform way. This scheme adds to the data format a few more bytes. But, is quite helpful in locating the processes belonging to one conversation.

7.4.6 Heterogeneous Systems

Before taking part in the process of sharing the data from a network, the process of registration in the global network environment is a must, This heterogeneous pool of resources is maintained with the coordinating processes running at the different geographical places and connected with a network. These processes belong to the different computer systems and also being run on different database management systems and the different operating systems. Such an environment for handling the shared database is known as heterogeneous database environment. To communicate through such an environment, the messages have to be translated in the language so that the two sites communicating with each other can successfully and exactly understand each other. Therefore, the messages being sent from the requester to the server needs proper translation at the different stages. The messages cannot be delivered without the proper translation procedure. Hence, the messages are the task, executed for the proper translation. This process of translation can be performed in various ways. Since, the translators and their availability again matters for the distribution. Therefore, the message task is considered for the scheduling to the most suitable terminal node, which can be helpful in the required task with proper local and the global efficiency of message conversation task.

7.5 THREAD MANAGEMENT AND CONCURRENCY CONTROL

The different activities in GURU can be invoked using

threads. A thread is a logical path of execution of different processes known as active object processes. To communicate among the different objects, a message is sent, the message invokes certain procedure or function in GURU. Which gets executed and forms a thread to link with other procedure calls. The thread can be invoked by a programmer with the help of some program. When a thread executes a logical space of an object, it accesses or updates the persistent data in the database of the object or the primitive objects inside the object. Further, the procedures and the functions may invoke the other objects. In such an event thread can leave temporarily the calling object, and enters to the called object. After processing the results it goes back to the calling procedure with the results. The object invocations can be nested or recursive. When a process finishes execution after providing the results to the requester the thread is terminated. Several threads can enter an object and execute concurrently. They can share the common primitive objects inside an object using the standard procedures known as locks and semaphores to work concurrently. The basic system consists of objects and threads. Inter-object interfaces are procedural. The objects are invoked using procedure calls, which work in the boundary of GURU.

Consistency of the operations is maintained through the users atomic instructions. This scheme is known as user based consistency control. The scheme is used in locking the primitive objects inside the object. These objects can be clustered in the different segments and these segments can also be locked through the user's instructions. Since, the segments are user defined, therefore, the granularity of the locking scheme is directly controlled by the users.

7.6 PROBLEM IMPLEMENTATION

The following procedure illustrates the message-task communication made in GSQL (GURU SQL) for some example protection scheme using object-threads. The users writing the system level programs are also required to provide the different protection schemes so that the unauthenticated applications cannot be entertained by the designed application procedures. Such kinds of implementations are carried out in GURU, which lets the application programmers define the different levels of protection to their own primitive and complex objects. The following instruction in GSQL shows the different levels of protection provided to its primitive objects:

```
define structure marks_sheet
privilege protect
memory sno, name, class, marks
privilege print
device printer_3, printer_7
privilege read
datafiles mohan
privilege general public
procedures read_record, read_marks;
```

The above instruction in GURU defines a class named marks_sheet, which has the memory variables sno, name, class and marks defined with privilege protect. The protect privilege lets the memory variables known in its own class environment and the different environments inheriting this class, marks_sheet. All other objects or classes inheriting the class, marks_sheet can refer all the memory variables but from outside this environment, they cannot be accessed. The next declaration in the in-

struction is device with privilege declared as print. This shows that the devices declared in the list of the device names i.e. printer_3, printer_7 can only be referred inside the class, marks_sheet with the privilege, print only. No other operation is allowed on these devices. This scheme protects the devices declared from the other processes. The other clause in the last instruction is datafiles with the declared privilege, read only. The declaration, datafiles is the declaration for the directory names of the database fragments. Therefore, the database fragment, mohan has only privilege to read its own tuples. The procedure declaration in the above instruction is for the procedures defined by the application programmer to handle the assigned tasks. The procedures, read_record and read_marks are separately written in GSQL and have the privilege to be called from any where including the outside environment.

Similar operations as defined above are implemented in the system software, which run on the system to support various system tasks. Further, the different objects can be created using the classes and enforcing the new instructions in the same object to support additional tasks. The following instruction creates an object:

```
create object m_s
privilege protect
include structures marks_sheet
privilege private
memory ch_1
privilege general public
functions test_ty;
```

The above instruction creates an object, m_s, which

PRIVILEGE IN BASE	PRIVILEGE AT NEXT LEVEL	RESULTANT PRIVILEGE IN BASE
1	1	0
1	2	0
1	3	0
1	4	0
2	1	1
2	2	2
2	3	2
2	4	4
3	1	1
3	2	2
3	3	3
3	4	4
4	1	4
4	2	4
4	3	4
4	4	4

PRIVATE - 1
PROJECT - 2
GEN-PUB - 3
LOCK-PUB - 4
NOT ACCEPTABLE - 0

CLASS INHERITANCE TRUTH TABLE WITH METACLASS SUPPORT
FIG. 7.8

inherits class `marks_sheet`. The privilege of inherited class is declared as `protect`. Therefore, the primitive objects derived from the structure (class), `marks_sheet` are only known to the object `m_s` and the other objects inheriting this object. The additional memory variable, `ch_1` is only known inside the object, `m_s`. The function `test_ty` is available to all the environments. The table shown in Fig. 7.8, discusses the final inherited privileges of the new classes and objects. Which become finally applicable to the various classes and objects after the inheritance. The another instruction which creates an object from the class inheritance procedure is given below:

```
create object d_r with structures marks_sheet;
```

The object, `d_r` is created with the class, `marks_sheet`. The complete set of primitive objects in the class `marks_sheet` is obtained and the privileges hold the same values as are present in the structures (classes) declaration. Let us discuss the privilege obtained with the inheritance of an object in more than one objects as shown in the Fig. 9.3. The primitive objects of the sub-object are known to the different objects inheriting the sub-object, are different. Because, the privilege levels associated with various objects are different. The following are the instructions used to construct the scheme.

```
create object registrar  
privilege private  
include objects m_s  
privilege general public  
memory r_marks  
procedures give_marks;
```



```
create object student
privilege general public
include objects registrar
memory s_marks
procedures display_marks;
```

```
create object teacher
privilege protect
include objects m_s
privilege protect
memory t_marks
procedures enter_marks;
```

The first instruction creates an object, registrar. This object inherits the other object m_s. The object m_s is created earlier. The only primitive objects declared in the object m_s having the privilege, protect and above will be accessible maximum to the object registrar. Further, the object, student inheriting the object, registrar for the purpose of knowing marks, will be available. The another object teacher is inheriting the object, m_s. This procedure is illustrated to maintain the marks of the students. The teacher enters the marks for the object, m_s through the procedure, enter_marks of the object, teacher. These marks are accessible to the object, registrar directly and the object, registrar checks these marks through the procedure, give_marks. Later the modified marks or the correct format marks are available to the object, student by the procedure, display_marks. Object, student here cannot know, who has given the marks and actual marks given etc. Therefore, this scheme is successful in providing the marks of the students with proper protection.

To invoke a procedure the following are the steps which help through an object-thread procedure to invoke a thread:

```
do procedure enter_marks of teacher;
```

The above instruction calls the procedure, `enter_marks` of the object, `teacher`. The procedure, `enter_marks` can further invoke some other object, `registrar` for the procedure inside that object, `registrar` being invoked. Therefore, the thread moves from the object, `teacher` to the object, `registrar`. The procedure in the object, `registrar` is invoked with some arguments to be passed as message. The arguments would be the marks of the students and the other reference for the database, where the marks would be entered. Finally, the procedure in `registrar` finishes execution and gives back control to the object, `teacher` hence, returning the thread to the object `teacher`. The method shows to control the procedure calls through a mechanism of thread, which helps in maintaining the atomicity and concurrency of operations.

7.7 CONCLUSION

The methods discussed for handling messages as tasks in the object-oriented heterogeneous environment are very effective from the view point of inter-object communication. The separate allocation of task is done in the form of the active objects, which are invoked using the thread procedure. This procedure helps in many ways such as, for the recovery of the data in cases of rollbacks in concurrency control mechanism, tracing the active object paths in heterogeneous distributed environment, performance evaluation of block operations, concurrency control etc. The asynchronous exited protocol with path assigning scheme provides a better concurrency in the system over other methods.

For some existing applications, we have found that message-task can result in superior performance. This is possible for two reasons, first with memory DSM algorithm, data is moved between hosts in large blocks. Therefore, if the application exhibits a reasonable degree of locality in its data accesses, reducing overall communication requirements. Second, many (distributed) parallel applications execute in phases, where each compute phase is preceded by a data exchange phase. The time needed for the data exchange phase is often dictated by the throughput of existing bottlenecks. In contrast, DSM algorithm typically move data on demand as they are being accessed eliminating the data exchange phase, spreading the communication load over a large period of time, all allowing for a greater degree of concurrency.

The different layers separating the application, network and system tasks are properly coordinating and the message-task is efficiently handled by GURU shell.

A few Library functions running in GSQL and GURU are provided in Appendix - F

OBJECTS AND DYNAMIC ENVIRONMENTS

8.1 INTRODUCTION

Often, during the object-oriented database handling, it is necessary to modify the object and class structures due to the changing object behavior [108],[180]. Most of the time the objects keep changing their environment and therefore, their structure needs modification [80],[83]. The modifications in their structures can be performed by adding or deleting some classes or objects from the current structure of the object, or some primitive objects can be added or removed in the object or class structure. Later, the initial structure can also be recovered from the backlogs if needed. In some applications, it is desired to maintain the previous structure of the object or class, which existed in past, at a given time [69],[198].

The changes made in the objects are reflected to the overall environment connected with the object. This environmental change may cause inconsistency, which can adversely affect the integrity of the system as a whole and leads to chaos. Due to the changes made dynamically in the global schema, the consistency of the distributed environment tends to change and hence it needs to be preserved. The control messages are passed to the different objects maintaining the database fragments to change the schema. These messages are executed to incorporate the required changes in the schema maintaining the consistency.

With the existing database management systems, the facility of changing object structures is very rarely available

[4],[13]. Only a few prototype systems such as ORION and MORE have the facility of changing structures but, that is up to the prior defined instructions, which are mostly menu driven [25].

The available communication networks with workstations need fast device control to meet the fast transparent distributed database requirements. Due to many limitations on the networks, the data cannot be provided with fast speed. Therefore, the object environments need variable kind of support, which can be handled with dynamically changing environments based on the available network conditions.

In the following sections, efforts are made to support the demands for changing object environments with the proper design of object structures.

8.2 STRUCTURAL TOOLS

A dynamically changing environment is presented by an example, the activities of a person can be clubbed in the different groups. Each group has some activities which belong to a category of a task. Like, a person can be a good player of tennis, a good teacher in the area of Computer Science and Engineering, and also a good father. These distinct activities are assigned to different classes, which are sports activities, professional activities and the household activities. By clubbing these activities the behavior of a person can be identified. Further, the classes can also be broken into sub-classes like, the sports activity can be further classified in indoor and outdoor games activities etc. By linking these classes the task becomes simplified, and well organized in a hierarchy. The top node of the hierarchy is known as root node and the lower nodes

in the hierarchy are known as sub-nodes. A root node in the hierarchy can access all primitive and complex objects of its sub-nodes. The different component objects belonging to primitive and complex objects can be protected with locks in the hierarchy. Which provide the tight privacy and security to the component objects. Locking is made better userfriendly by providing the different privilege levels, which can be used to identify the right user. Therefore, the users do not require to assign any external keys unnecessarily. But, whenever external locks are required for some confidential data, the external keys are assigned, only once in a while or as is relevant with an application. Which may become sufficient to retrieve the data from objects with complete privacy and security. The unauthorized access over the objects can be prevented by providing the proper locking scheme.

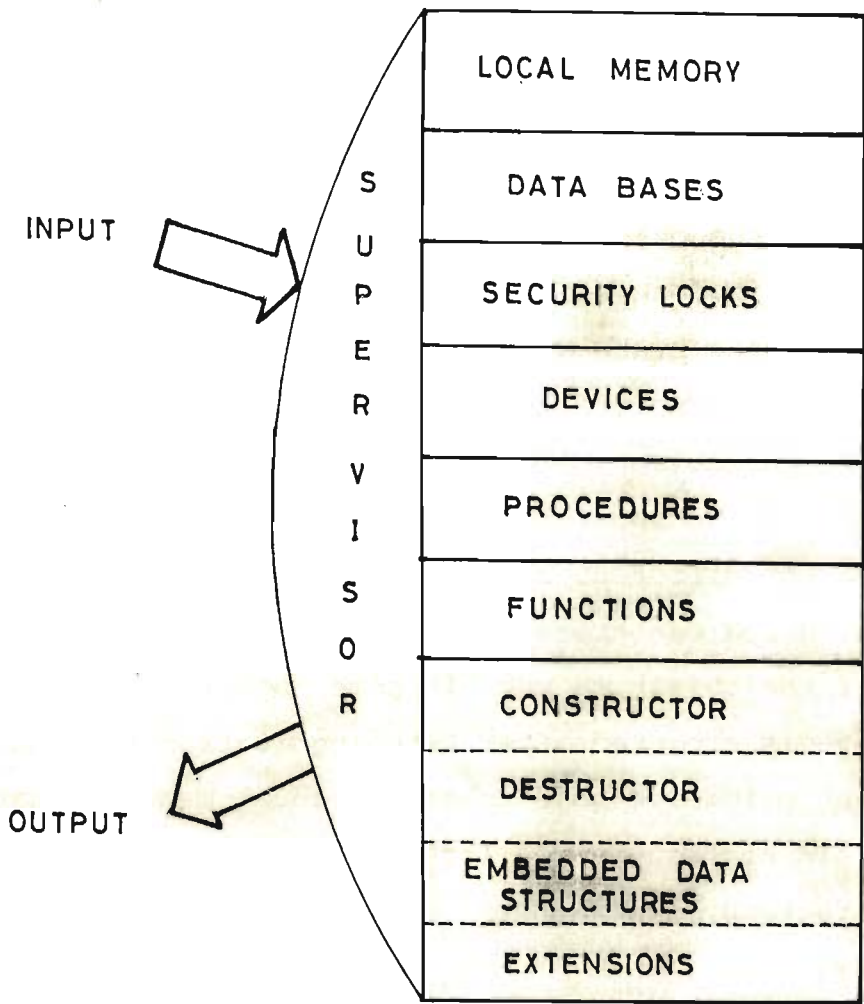
The changes in the object structure may affect the database adversely, in the sense, if the wrong manipulation is done on certain data and the data structures, then the possibilities of loosing data would increase. Which may provide the wrong results on querying the database from the different terminal nodes in the distributed environment. To check the consistency effect of the last operation, the intermediate locking arrangement proves to be a very useful scheme. Further, the facility of maintaining backlog can prove to be an asset for the fast recovery operations. The backlogs provide the different previous states of the operations from the last checkpoint. Which are required for the rollback instruction invoked due the failure of the acceptance test.

8.3 CLASS AND OBJECT STRUCTURES

A class can be defined as a category of the related objects in some problem domain, which is identified based on certain properties. For example, in the earlier discussed example, Profession is a class. A class can inherit different other classes and objects. The different classes are assigned different class identifiers, which help in locating a particular class.

An object can be defined as the collection of a few small entities of the real world belonging to a class. An object can inherit one or more classes and objects, in other way representing the overall view as a class. At any instant of time contents of an object provide the real status of the object. The process of deriving an object from some other objects and classes is known as instantiation of the object. GURU database is a set of objects, where each object represents certain entity in the real world. The objects can be differentiated by the unique object identifiers and the objects can be categorized as the active and passive objects. The passive object is an element in an object which comprises of some values like memory variable, the field variable etc. A passive object cannot be executed on its own to process values. Where as an active object can be executed and can process values on its own.

Further, the objects can be classified as primitive objects and complex objects. The primitive object is the smallest entity of the object defined in terms of some active or passive element of the object like, some memory variables, database field names, device name or some procedure/function of the object. A complex object can be defined as the combination of a few objects linked together. The complex objects can be linked in the form of hierarchies, where different objects encapsulated in an object



OBJECT IN GURU

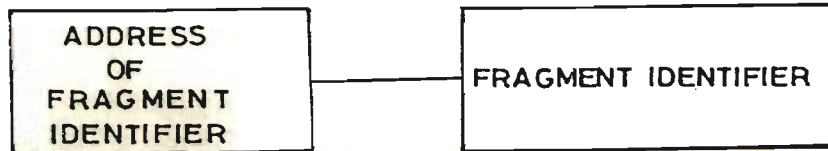
FIG.8.1

are known as the sub-objects of the object. The object, which encapsulates the other objects is known as the super-object. The data structure connecting the objects can be obtained by linking the various pointers to the various other objects to form some relation. The objects organizing the complex objects are known as the component objects. The component objects manage with other component objects and form a hierarchy with ordered set as predecessor and successor of a component object as determined individually in a complex object.

An object is shown in Fig. 8.1, which comprises the individual components. Each and every object consists of a supervisor, which is also known as a guardian. The supervisor manages all the activities of the object. The management of the individual primitive objects is done with the help of a supervisor. The supervisor, is something transparent as far as the structure of the object is concerned. Therefore, the supervisor belongs to the third dimension of the object. All kinds of transactions within an object are done via the supervisor of the object. The transactions may include the different messages to be delivered to the primitive objects. Which can store these messages in the mailbox of the object, if the object is busy in some other activities. Later, the message is executed after fetching it from the mailbox. The different primitive objects are as local memory, databases in the form of fragments, locks, devices, procedures, functions, constructors, destructors, embedded data structures, and the extensions.

Local memory in the objects comprises the memory variables of different kinds, in GURU they can belong to some basic classes or derived classes. The basic class memory variables are of type character strings, numerics, Boolean, date and a general type to record any kind of the data in any size,

OBJECT NAME	DATABASE FRAGMENT ALIAS	DATABASE REFERENCE NAME	FRAGMENT DESCRIPTION CODE	INDEX DESCRIPTION CODE	-----
----------------	-------------------------------	-------------------------------	---------------------------------	------------------------------	-------



FRAGMENT HANDLING DATA STRUCTURES
FIG. 8. 2

which is known as a memo. The character string can have any mixed characters which can be of maximum size as 950 characters wide. Numerics can be of type integer, real or the floating point data. Boolean variables can be recorded as logically true or false, or yes or no type. Dates can be recorded in a standard format of the data requested by the user. The various countries have their own different formats in specifying dates. Therefore, an appropriate format can be selected and stored. The memo can store any kind and size of data. Derived classes, can define the individual name to the passive object and in their own formats. Which can be used later to define any kind of data with the objects as the passive object memory variables.

Databases can be defined with their fragment names (refer Fig. 8.2), which are recorded in the memory and as the object is being referred, the fragments are loaded in the main memory to be manipulated as required. The fragments are governed in accordance with their qualifiers. Before accessing a fragment its qualifier is checked for its selection. If the query satisfies the fragment qualifier, then the query is processed by loading the fragment in the main memory. Otherwise, the query needs to be checked on other fragment qualifiers. If the required fragment does not exist with the current terminal node, then the query is routed to the terminal node where the fragment exists. Later, the query is processed and the results are sent to the current terminal node. This operation is carried out transparently. The user thinks as if the fragment lies with the current terminal node only. Finally, the results are obtained by the terminal node, where the user is working. The fragments can also be maintained from the local user's perspective. Where the complete local autonomy is established. To establish the local autonomy the different fragments maintain their synonyms and are

locked with the directives of the programmer. Locks are basically divided into four kinds, which are private, protect, locked public and general public. The private declared database can be accessed in the object, where this is defined and cannot be accessed by any other class or object inheriting this class or by the other classes inherited into this class. Protected databases defined with an instruction protect can be accessed in the inherited classes or objects but not outside. The locked public kind of objects can be accessed any where, provided the key of the lock is provided. Lastly, the general public declared databases can be accessed any where in the database environment. These locks can be used for any declared primitive object inside a class or an object.

The individual fields can be protected by specifying the appropriate lock, therefore, making a view in an object or a class to be protected as required by the application programmer. If some fields are not specified separately, which means that the fields have the same lock as is specified with the fragment name of the database. Otherwise, the field names are required to be mentioned along with the kind of locking needed. The key of the lock as discussed previously can be based on the privileges assigned to the various users of the system. These privileges can change with time and also can vary with the environment of the problem.

The devices can also be made available to the object environment, for example, hardware and software resources can be defined as the devices, which are printer, plotter, video display unit, keyboard, some area of hard disk, some interface software etc. The reason in defining the various devices in the object is to protect them with the unprivileged processes and have proper

control over them. For example, some data is needed to be displayed on the user's terminal, then this data will only be displayed to the user, provided the user has the privilege of accessing the display unit in the current object's environment. This makes the task very suitable from the privacy and security view point. The provision of fine grain control applied on the various defined devices provides the better integrity and consistency.

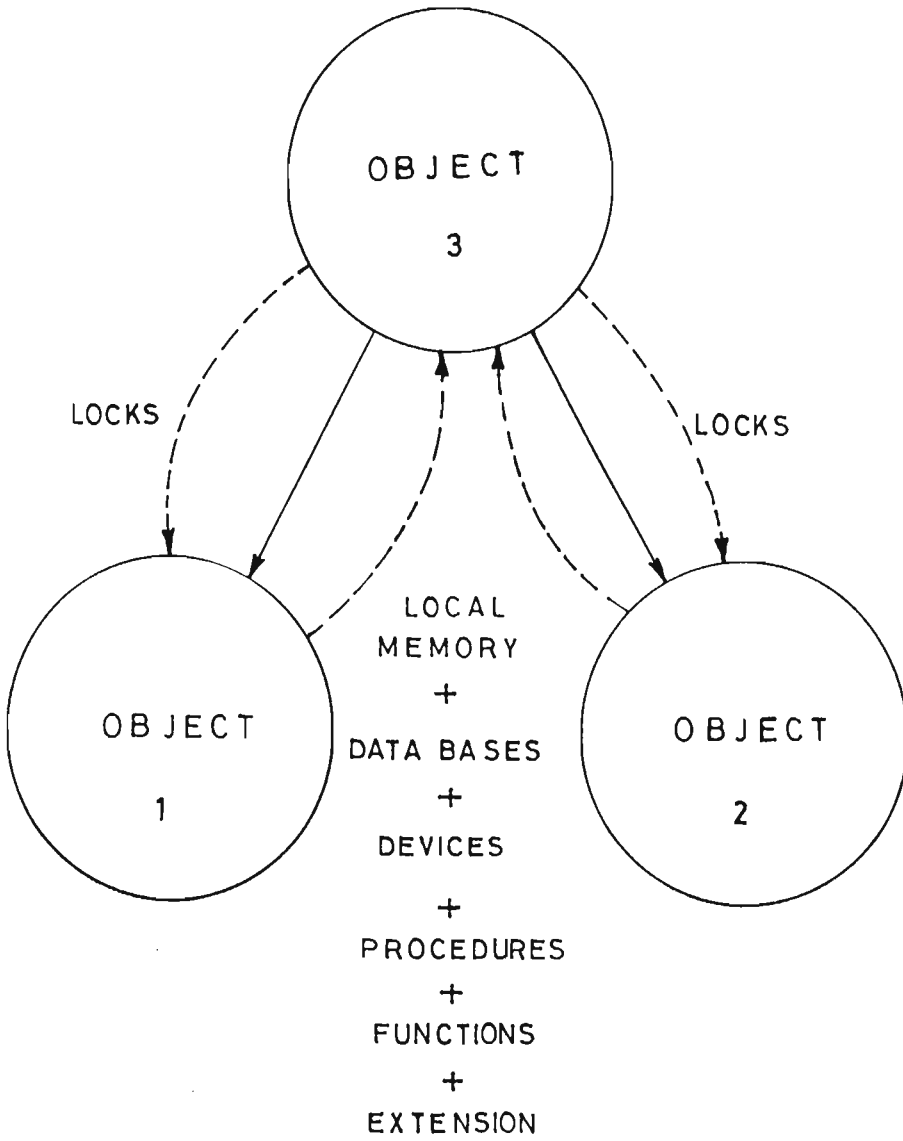
Procedures and functions are basically methods in GURU, these methods are separately defined by the users to process some of the activities in the local object or class environment. These activities are the user defined activities and may or may not be existing with the internal configuration of the supervisor belonging to the object. For example, a few database fragments are required to be accessed in the way programmer wants, or some of the data structures defined by the programmer are required to be handled in the ways the programmer requires. Therefore, apart from the power available with the supervisor, the user defined activities are also executed. To support these activities, GURU provides the two kinds of methods: Procedures, which are separately called by a programmer. The procedures have the power to pass many arguments, and the results are returned back in the same arguments, after the procedure is executed. This method of getting the returned values in the corresponding arguments, is known as call by reference method and provides more than one results. Where as the other method known as Function, provides the only one result and the arguments passed to the function remain intact with the values given to them, after the function is executed. Functions can be used in general expressions, where they can play an important role to return the values in the expression after getting executed, which are subsequently

processed in the expressions.

Constructors are the procedures which are executed to construct the block of the object in the memory and initialize the different values to the object block. These values are assigned to the memory variables or some prior defined status of the object. In GURU, constructors are optional procedures, to be declared by the programmers. If the constructors are not included by the programmer, the supervisor initializes the values to all the elements present in the object automatically with the default values, when the object is instantiated.

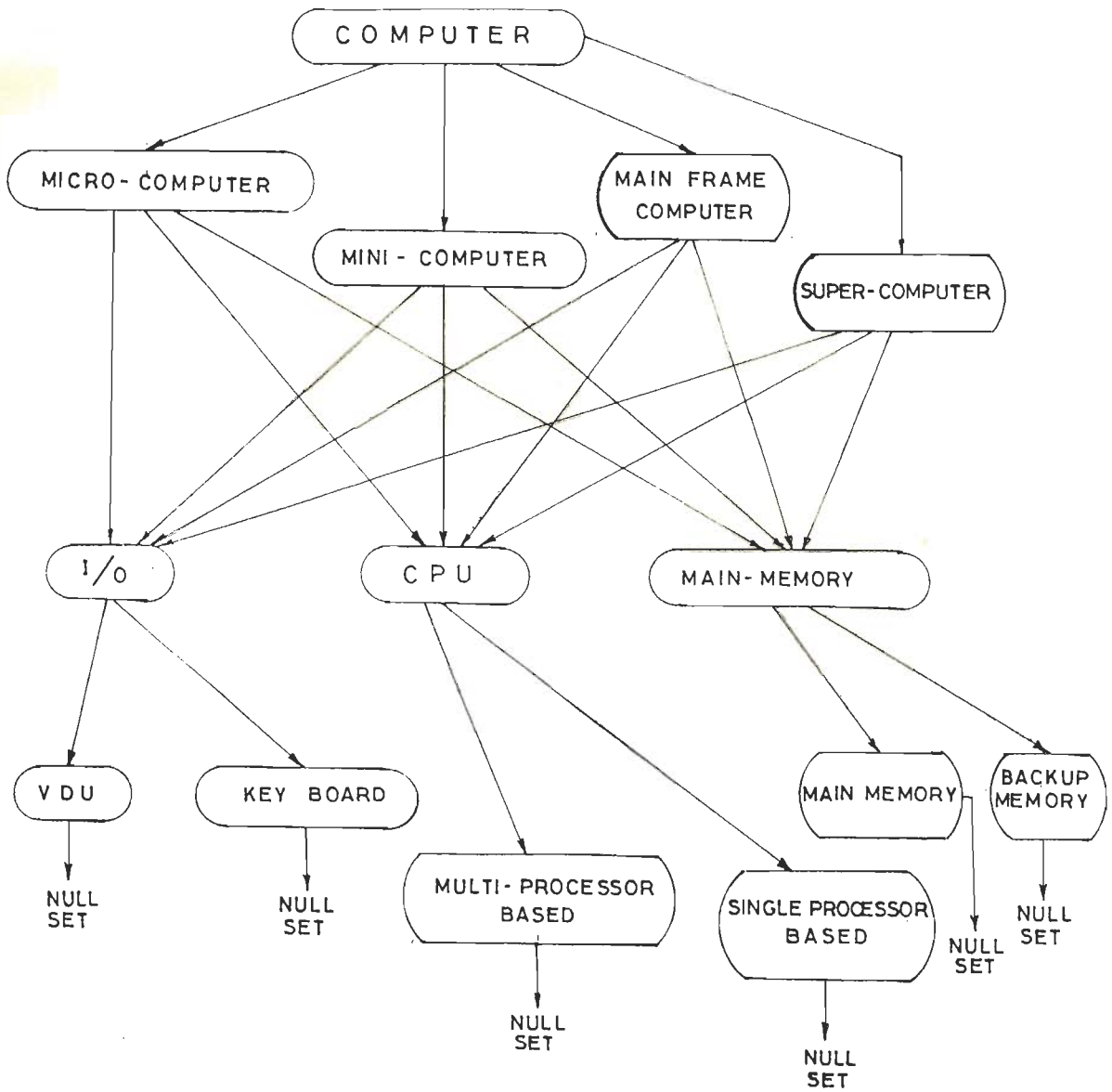
Destructors are the procedures which are executed at the time when the object is required to be removed. Destructors make the memory free, which was occupied by the object. The destructors are also the optional procedures, declared by the application programmer. The user assigns a destructor program separately, which is intended to remove all the area occupied by the object as well as, the application programmer can perform many more operations, to compensate the deletion of the object. The method could be used to inform the other objects related to the object being removed, regarding the policies to be adopted after the current object is removed. This operation updates the data structures dependent on the current object.

Embedded data structures are the mechanisms through which the different fragments of the database can be controlled and the extended structures help in preparing the various views and many more operations could be performed with them. The status of the object can be defined, the procedures and functions are linked with proper relations, the inheritance of classes and objects can be recorded and lastly making a room for the changes



COMPLEX OBJECT

FIG. 8.3



CLASS HIERARCHY
FIG. 8.4

to be incorporated in the data types and the structure as a whole of the object. The extensions mentioned make a room for the new kind of primitive objects which could be declared by the programmers.

A complex object is shown in Fig. 8.3, which comprises the two sub-objects. The complex object is also known as super-object of the two sub-objects. Similarly, the different objects can be connected to form a hierarchy of objects. A super-object can access all the resources of its sub-objects. The resources can be made available to the higher objects in the hierarchy, provided they have the privilege to do so. The locks are shown to lock the access of the lower objects. Therefore, to perform operations with the lower level objects in the hierarchy, the keys should be made available with the super-objects. The keys defined in GURU are applicable automatically in some cases of the objects, which belong to the higher privilege level. This makes the objects relieved from the burden of specifying keys to communicate with other objects. The primitive objects of the lower objects in the hierarchy are controlled from the higher level in the object hierarchy. The primitive objects are local memory, database fragments, devices, procedures, functions and the extensions. The extensions are the declared class or object structures which modify the structure in a non-standard prescribed way. The type of passive objects can be associated to declare the format of the data, for which the extended data structures can be used to have the proper primitive object linking. The further discussions are done later regarding the class and object table maintenance for the extended structures. For example, a class name, computer is classified in Fig. 8.4, with the sub-objects as micro-computer, mini-computer, main-frame-computer and super-computer. Next level can further be achieved which consists of

I/O, CPU, and memory, the new level of that could be further derived as VDU, key-board, parallel processor, single processor, main-memory and back-up memory. In the end all the last level components can further be emerging in a common null pointer. That shows the leaf nodes are all connected to a null pointer. In this way a class or object hierarchy is organized. The behavior of any object node in the hierarchy can be seen through the other nodes linked as sub-object nodes to the object node in the hierarchy.

1.4.1.1

In distributed environment the structure of the remote objects can be changed by sending messages to the object. The message which requires to change the structure of some object is known as the structural message in GURU. The format of a message is shown in Fig. 8.5. To start with the message packet, the length of the message is given in terms of bytes. The size of the message packet comes first. Next is the number of terminal nodes to which the message is required to be directed and after this the addresses of the terminal nodes lying on the heterogeneous network, to be communicated. Along with each of the terminal node the address of the DBMS is also enclosed, which tells about the DBMS to which the message belongs to. Next the user specifies its own identification in terms of User's name and the password. Which makes the system to accept the message, if the user's name and the password are correct. Later, the user specifies the name of the application the user wants to interact and along with this user also specifies the name of the schema in that application, to which user wishes to interact. After getting the last procedure accepted, the type of operation requested by the user is being indicated through a message code. The structural messages given for changing the structure of the class or object have a specific indicated code. The message data is given,

through the type of message byte. The message is accepted as structural message and there after the message is translated in the actual language code with the help of an intelligent translator. The last item in the message format is the end flag, which indicates the end of the message packet.

The translator discussed earlier comprises all registered DBMSs translation codes on the network. If any DBMS is registered to participate in the distributed environment and share the common database among the different terminal nodes running the different DBMSs, then the translator is a must to support the global environment. Which translates the different languages in exactly equivalent instructions to the other languages in the global environment. If the exact translator is not available with the terminal node to which the message is directed, then the message is directed to the terminal node which translates the message and the translated message is sent to the place where it is to be processed. A procedure known as registration is executed to identify any new terminal node, which wanted to participate in the activities of the global data sharing scheme.

8.4 EXTENDED USER DEFINED AREA

This area has been playing an important role in the dynamically changing environment, by maintaining the different details of the trigger functions discussed in the next chapter. The different tasks can be performed with the help of triggers which become beneficial from the view point of checking performance and other details with the system. The other objectives which are solved from this extended area are the intelligent functions which can be built with the system to help

in various ways like efficient query handling, with the least provided information the system performs, as is required by the user etc.

The following GSQL instructions use the *extended user defined area* of object structure:

copy object s-object-name to extended structure of object d-object-name;

copy extended structure of object s-object-name to object d-object-name;

The above first instruction write the *properties* (contents) of the object *s-object-name* to the programmable extended structure area of the destination object *d-object-name*. Any properties of *s-object-name* can be copied to the *d-object-name* and later can be retrieved by the same object or the object inherited from the class used by the *s-object-name* to maintain consistency. This instruction is mostly used to store the data required for triggers.

The instruction *move extended...*, copies the extended structure data as properties of an object. The extended structure data of the object *s-object-name* is copied to the object *d-object-name*, the *d-object-name* starts having the new properties after the instruction is executed. For maintaining consistency, the directions of the first instruction should be followed.

8.5 CLASS AND OBJECT CREATION METHODS

A class can be created by specifying the nature of the primitive objects to be defined. Most of the kinds of passive

MESSAGE LENGTH	DESTINATION TERMINAL CODE	SOURCE TERMINAL CODE	DESTINATION DBMS CODE	SOURCE DBMS CODE	USER NAME & PASSWORD	APPLICATION NAME	SCHEMA NAME
-------------------	---------------------------------	----------------------------	-----------------------------	------------------------	----------------------------	---------------------	----------------

TYPE OF MESSAGE	M E S S A G E D A T A	END FLAG
--------------------	-------------------------	-------------

MESSAGE FORMAT
FIG. 8.5

objects are supported in a class. These structures can be recorded in the memory to inherit the properties of the defined objects. A class in itself is an environment, which supports a complete behavior of the involved objects. The following instructions create a class in GSQL:

```
define structure class-name
[include object object-name-list
[as {subset | superset | union | intersect}]]
[include structures class-name-list
[as {subset | superset | union | intersect}]]
[datafiles data-file-name-list
[fields field-name-list]]
[memory memory-variable-list]
[procedures procedure-name-list]
[functions function-name-list]
[privilege privilege-code]
[device device-name-list];
```

The above instructions, create a class with the name, class-name. The statement, structure is called a class in GSQL. Therefore, the class-name is a must to be defined, to declare a unique class-identifier. A dynamic class table is shown in Fig. 8.6. The table records the class identifier as a class name. A class can inherit all kinds of objects and the classes. The include clause is an optional instruction, which provides the different classes and the different objects to be inherited in the defined class. Constructors can be defined by the name of the class in the class structure under the procedure statement. The defined constructors play an important role to establish the various data structures and maintain the right position of the class in the class hierarchy. Constructors also maintain the data structures to place the different relationships of the other

classes and objects with the database fragments maintained in terms of datafiles. The database fragments of a class are related with certain relationships maintained as the key fields organized with required indexes and the fragment qualifiers. Therefore, by declaring the passive objects in the class (structure), it cannot be suffice but, there has to be certain relationships to be declared which would be maintained with the data structures. The established data structures, help in accessing the data from the database and checking the different privileges to prevent unauthorized access on the database. The data structures also provide a path to access procedures, which access data efficiently or in the required manner.

The fields clause in the above instruction is also an optional instruction. Which provides certain locks to the fields prescribed with the field names and the fragment name to identify the right field. This provides a view management to the data fields and provides the privileged access only. The privilege clause can be used before any primitive object declaration to provide the privilege to a primitive object. The memory clause declares the local object memory variables, which can be used to record the local object environment parameters. These memory variables can also be accessed as discussed in the previous section. The procedures and functions are also used to define the user's procedures for certain tasks. These can simply be called by calling their names with the request made to the supervisor of the object. The following is the instruction used to create objects in the memory:

```
create object object-name
[include object object-name-list
[as {subset | superset | union | intersect}]]
```

```

[include structures class-name-list
[as {subset | superset | union | intersect}]]
[datafiles data-file-name-list
[fields field-name-list]]
[memory memory-variable-list]
[procedures procedure-name-list]
[functions function-name-list]
[privilege privilege-code]
[device device-name-list];

```

The above instruction declares all the primitive objects as usual except it has a facility of maintaining the complex objects with union and intersect data relations. The other feature is that, the declared object does not essentially require a class structure in GSQL. Later, the classes can be inherited by other instructions. Therefore, in the beginning the skeleton of the object can be a bared structure. The another format which directly derive an object from a class is illustrated as below:

```

create object object-name with structures class-name-list;

```

The above procedure inherits the complete class structure of class-name-list in the object declared with object-name. All primitive objects can be accessed by the object name first and then with an arrow symbol '->' to the primitive object name. For example, the memory variable, mem_var of an object obj1 is written as obj1->mem_var. The instructions, which change the structure of the object are as following:

```

move object object-name-list of obj-class-name-1 to
obj-class-name-2;
move structure class-name-list of obj-class-name-1 to

```



```
obj-class-name-2;  
move procedures procedure-name-list of obj-class-name-1 to  
obj-class-name-2;  
move functions function-name-list of obj-class-name-1 to  
obj-class-name-2;  
move memory mem-var-list of obj-class-name-1 to  
obj-class-name-2;  
move datafiles data-frg-list of obj-class-name-1 to  
obj-class-name-2;
```

The above instructions move a list of objects from a class or from an object to the other class or an object. In GURU, objects are also treated as the classes for the inheritance property. A concept of metaclass is maintained by maintaining the class inheritance hierarchy as the object inheritance hierarchy is maintained. Similarly, all other instructions are used to move the primitive objects from one object or class to another object or class. This procedure helps in changing the structure of the class or the object as per the requirement. The other instructions which remove the links from the inherited objects or classes and with the other primitive objects are discussed as below:

```
remove object object-name-list from obj-class-name;  
remove structure class-name-list from obj-class-name;  
remove procedures procedure-name-list from obj-class-name;  
remove functions function-name-list from obj-class-name;  
remove memory mem-var-list from obj-class-name;  
remove datafiles data-frg-list from obj-class-name;
```

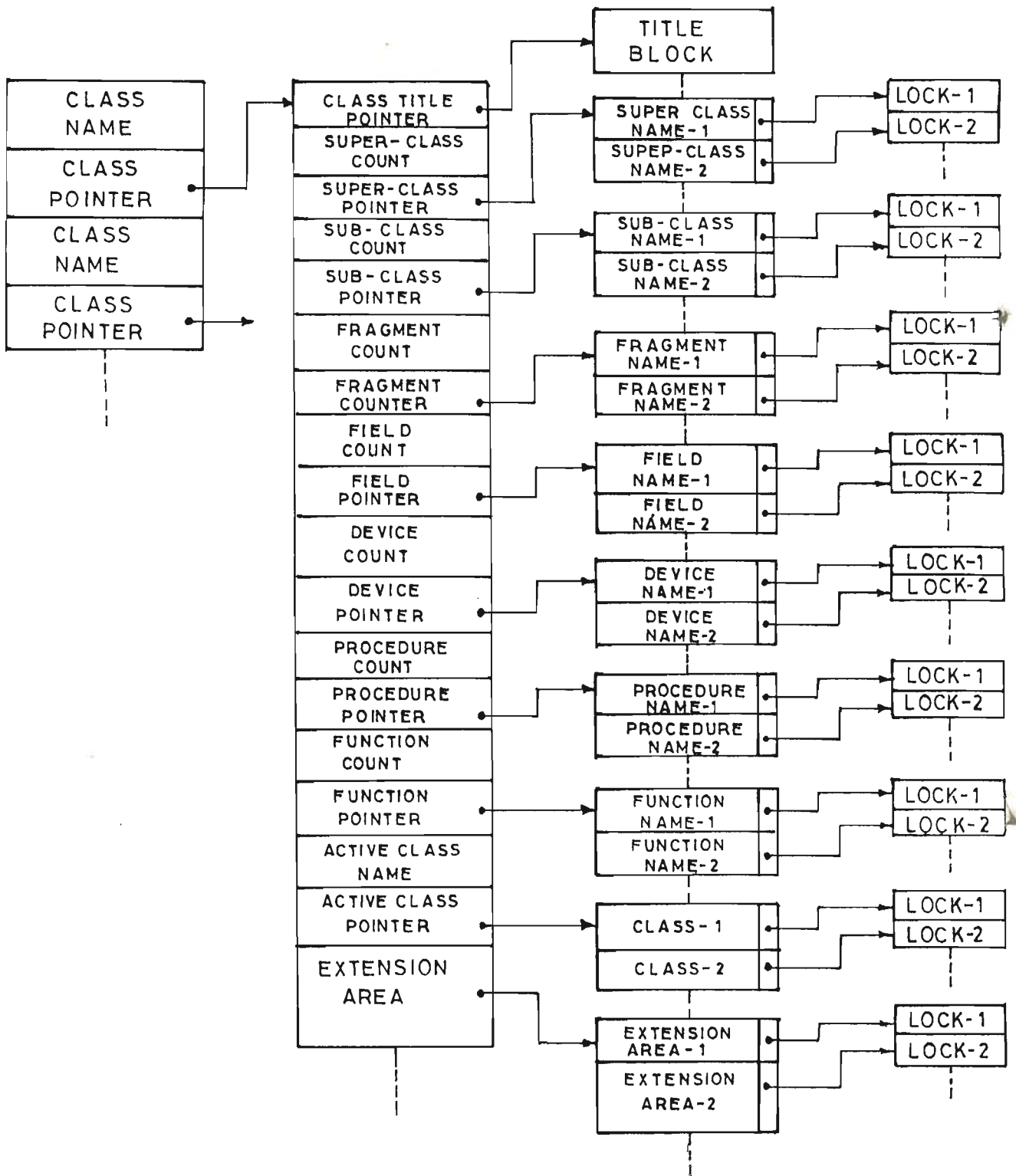
The above first two instructions remove the list of objects or classes from a given class or an object. The other instructions remove the list of the given primitive objects from

a given class or an object. Similarly, the instructions discussed below insert certain objects or classes into the given class or the object. The other instructions insert primitive objects into the given object or the class with the specified privilege.

```
insert object object-name-list into obj-class-name  
[with privilege privilege-name];  
insert structure class-name-list into obj-class-name  
[with privilege privilege-name];  
insert procedures procedure-name-list into obj-class-name  
[with privilege privilege-name];  
insert functions function-name-list into obj-class-name  
[with privilege privilege-name];  
insert memory mem-var-list into obj-class-name  
[with privilege privilege-name];  
insert datafiles data-frg-list into obj-class-name  
[with privilege privilege-name];
```

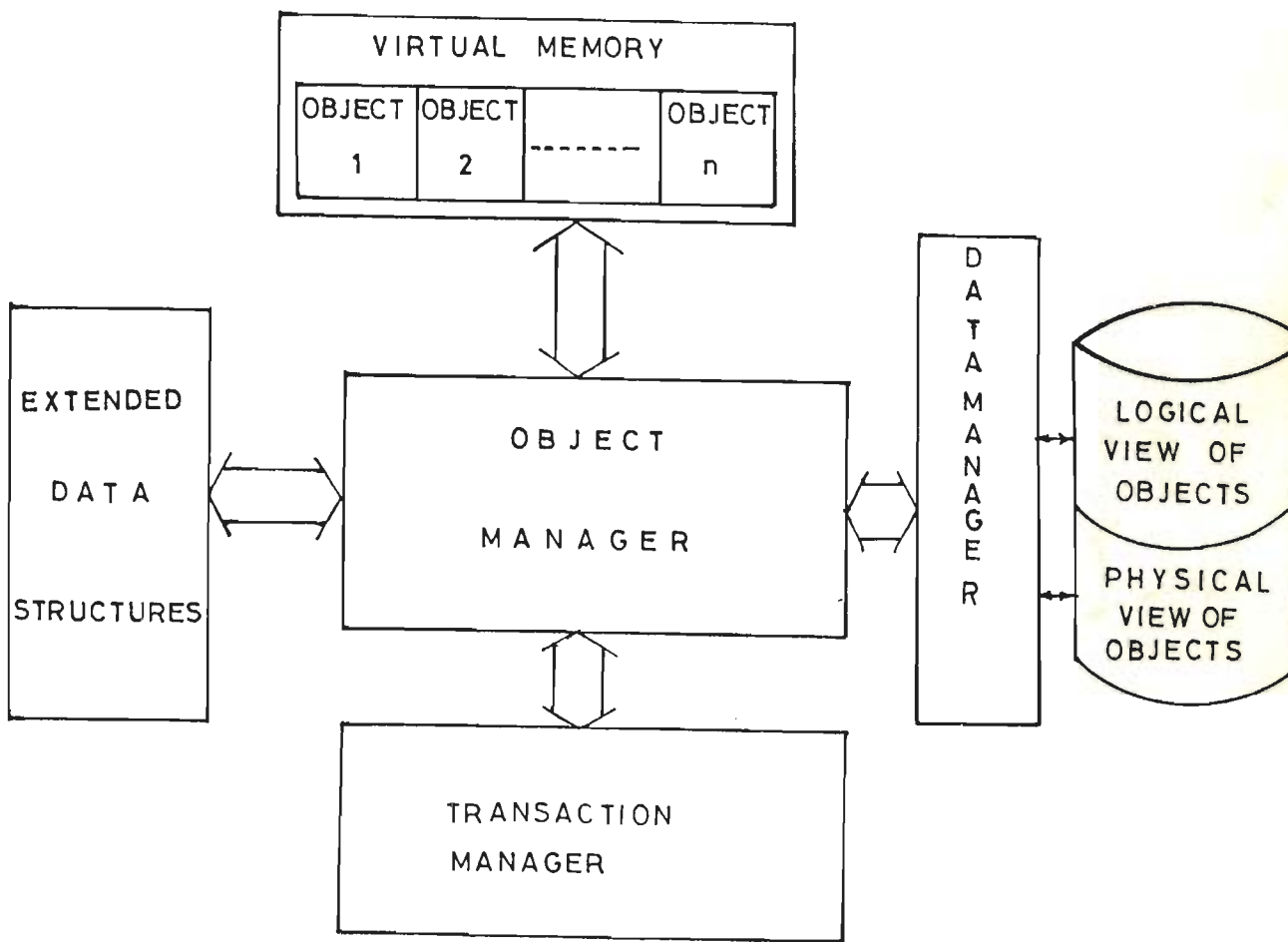
The object and class handling instructions can be seen as below:

```
display object object-name structure;  
display structure class-name;  
display memory of object object-name;  
display memory of structure class-name;  
display procedures of object object-name;  
display procedures of structure class-name;  
display functions of object object-name;  
display functions of structure class-name;  
display devices of object object-name;  
display devices of structure class-name;  
display privilege of object object-name in structure
```



CLASS TABLE

FIG. 8.6



OBJECT HANDLING MECHANISM

FIG. 8.7

class-name;
display privilege of structure class-name in structure
class-name;
display privilege of object object-name in object
object-name;
display privilege of structure class-name in object
object-name;
display privilege of memory mem-var in object
object-name;

The above instructions display the primitive objects from the object or class structure. The privileges of the primitive objects in the declared classes or objects can also be displayed.

A dynamic class table keeps the record of the various declared classes. A similar table is constructed which maintains the data structures of the defined objects. The different objects created in the application are recorded in the object table. The created objects consist of the different classes which are referred from the class table for the future reference and linkages. The objects comprise of different sub-objects which are listed in the table of the objects. The objects can also hold relationship with another object as a super-object. The dynamic table for the super objects is also constructed.

8.6 OBJECT MANAGEMENT

Objects are managed separately with the help of an object manager. In Fig. 8.7, the transaction manager has a direct link with the object manager, which manages the objects dynamically. Objects can be managed in the main memory using the virtu-

al memory concept. Any objects which are not referred frequently are mapped in the backup memory using the virtual page scheme. This procedure saves space in the memory. This also makes room for more number of processes to be activated in the main memory simultaneously.

8.6.1 Object Manager

The task of handling the objects is performed by the object manager. Objects are treated as separate entities. Objects are placed in the memory exactly alike the processes are recorded and handled by the process scheduler or task handler. This mechanism is implemented in the same fashion as the page tables are referred. The object tables are referred for the objects to be located in the memory, if an object is missing from the main memory, an object fault is generated and the object is brought from the backup memory into the main memory. The transfer of object is carried out with the help of object and data managers. The data manager traces the logical view of the data and finally the physical data is retrieved from the object tables. The change in the structure of the objects and classes can also be communicated to data manager by the object manager. The logical and physical views of the data are appended accordingly. Further, the object and class tables are also appended. The data manager manipulates the data in the form requested by the object manager and any other unit of the GURU requiring the service. The extended data structures are recorded in the memory to establish the various links and relationships of the future referred data types and the data referring schemes for the extensions of the object structures.

8.7 PROBLEM IMPLEMENTATION

A problem can be defined and finally implemented in GSQL to alter the object and class structures.

8.7.1 Scenario

Let us consider an example of some university running only two departments of engineering and one Computer Centre. The two departments are Computer Engineering Department and the other is the Civil Engineering Department. Both the departments have their own computer cell and the computer personnel working with them. Since, the two departments are working in two different problem domains, therefore, the specialization of their computer personnel are also different. The Computer Centre is using the computer personnel in the mixed domain. Since, both the departments are taking their problems to the computer centre. In this example, for the whole university, the following are the operations suggested, where the change of the structure of the objects would be needed:

i. For solving the problem of some mixed domain in the department of Computer Engineering, some knowledge is required to be shared for some project being run under the Department of Computer Engineering. Therefore, some computer personnel are transferred from the Civil Engineering Department to Computer Engineering Department. This causes the structure of the object environment dealing with Computer Engineering applications as well as the object environment dealing with Civil Engineering applications are changed.

ii. Let us take the other side, where some of the computers are removed, because they were outdated in Computer Engineering Department. This also changes the structure of the object.

iii. A new department, Electrical Engineering is introduced in the university. Which at present does not have computer facilities of its own, but, it takes the help from the Computer Centre.

The above discussed points make use of the different GSQL primitives to be formulated into the final changing object environments dynamically. The dynamically changing environment is available in the heterogeneous distributed environment, so that it helps to the other users to know about the details of the computer facilities in the different departments of the university and the computer centre.

8.7.2 Implementation Strategies

The problem is formulated to handle the environment discussed above. The two basic classes are defined in the beginning. These classes consist of the details of Computer Equipment at various places in the university and the other is the computer personnel or the computer manpower. the following are the definitions of the two classes:

Class declaration for Computers-

```
define structure computer  
memory hp, dec, wipro, novel  
datafiles hpf, decf, wiprof, novelf  
procedures equipment, price  
functions test_equip;
```

Class declaration for computer manpower-


```

define structure c_manpower
memory maths_exp, data_struct_exp, hidro_exp
datafiles mef, dsef, hef
procedures skill, wages, efficiency
functions skill_test;

```

```

create object computer_o with computer

```

```

create object c_manpower_o with c_manpower

```

The above statements define the two classes one with name computer and the other with name c_manpower in GSQL. These two classes can be used in creating two departments and a Computer Centre. Further, the two objects computer_o and c_manpower_o are created. The following instructions create the three different objects:

```

create object comp_e_d
include objects computer_o, c_manpower_o
procedures set_comp_rel;

```

```

create object civil_e_d
include objects computer_o, c_manpower_o
functions test_civil_rel;

```

```

create object comp_c
include objects computer_o, c_manpower_o
functions test_comp_c_rel;

```

The university record consists of the two departments and the computer centre. Therefore, the object with the name of the university can be created in the following way:

```
create object university_x
include objects comp_e_d, civil_e_d
functions test_university;
```

The above instruction creates an university object with the name university_x. Now the earlier operations are tried on the university_x hierarchy.

i. Computer personnel are transferred from Civil Engineering Department to Computer Engineering Department:

```
move memory hidro_exp of civil_e_d to comp_e_d;
```

The above instruction transfers memory variable with name hidro_exp to the object comp_e_d. After this operation the memory variable hidro_exp is remove from the object of civil_e_d and is added in the object of the comp_e_d.

ii. Some computers which are out dated are removed from Computer Engineering Department as follows:

```
remove memory dec from comp_e_d;
```

The above instruction removes the memory variable dec from the object comp_e_d.

iii. A new department with name Electrical Engineering is being inserted in the university as follows:

```
create object e_e_d
include objects comp_c
```

```
functions test_civil_rel;
```

```
insert object e_e_d into object university_x;
```

The above instruction inserts the department e_e_d into the object university_x.

8.8 CONCLUSION

The various schemes of defining the objects and classes are discussed. The objects have a wide range of the primitive objects. Which are found exclusively better for their structure, available with other object-oriented distributed database management systems. The metaclass approach has been implemented. The existing object model is the best for the high speed computer terminal nodes having the large main memory size and also handling the large databases in the dynamically changing heterogeneous environment. The objects maintain the extended data structures, which can be used for various purposes. The methods used with various triggers, maintain the various finest level transactions going on with GURU. This approach helps in maintaining the atomicity and the serializability of the concurrent operations. The other approaches used with the extended data structures provide the maintenance of the database fragments and the virtual objects. A statistical approach is used for the purpose of allocating the various virtual objects and the fragments at the different terminal nodes. Further, the methods used with extended structures are helpful in providing the strict privacy and security to the views of the object in various environments.

OBJECT PROTECTION AND SYSTEM ADMINISTRATION

9.1 INTRODUCTION

In the modern time of advance distributed processing, the available protection schemes for the database and objects are not properly established, which are leaving behind the possibility of getting the access and power to modify, the confidential precious data in the database by the unauthorized processes. Therefore, there is a great need to device certain foolproof systems, which make impossible to temper or access the data by the unauthorized processes. In this direction so many efforts are made [193],[196], but with the current trend of getting the massive multiprocessor environment with distributed support running multiple processes concurrently, some times in thousands, make the system weak, from the view point of privacy and security. Unnecessarily putting the security locks in the system also make the system slow, inefficient, and non-user-friendly. Therefore, there is a great requirement of providing the optimal locks, which make the system with tight security and on the minimum compromising the efficiency issue of the system.

A system which changes the priority of its own objects and the privilege levels dynamically, depending upon the requirements, puts forward a great challenge to the privacy and security of the different components on behalf of the other parameters to provide support. The object-oriented distributed database management system is the one which needs more attention to guard its objects from unauthorized processes, running in the dynamically changing environment. Since, the processing is done on object

level, the objects inter-link the different processes by the thread control mechanism [60]. One active object can deal with multiple threads to support the current process. Therefore, the different active objects interact in the way to access and modify the structure of the object and the privilege levels of the primitive objects based on the requirements. The message based communication among the different objects threatens the privacy and security to the object access mechanism [156]. A message itself is an object, which is transmitted through the communication network to remote terminal nodes. Where, the message object is reached to the required object supervisor through a message object processing channel. This processing channel is the one where the message object could be tempered or accessed. Therefore, vast arrangements are made by the communication network servers to guard the objects. There are many communication networks available to communicate with the global network environment. Out of them a few are popular due to higher communication band width and easy availability. Ethernet is the one, which is more extensively adopted by a wide category of the people. This provides speed of communication of about 10 megabytes per second [196]. Using ethernet, which does not provides security to its data but, one can have suitable fast communication among the terminal nodes on the global network.

The communication link takes the information packet to the terminal node to which the message is required to be directed, the message is accepted by the terminal node from some internal port of the node [129]. Since, the message enters to a particular terminal node, the results may also be expected to come via this port, hence, causing an easy threat to the results. Therefore, a different procedure is adopted to alter the route of the results, by changing the ports randomly, making difficult to

know which port is transmitting the results. This is illustrated in the next section.

9.2 KINDS OF PROTECTIONS

Basically, three kinds of protections are used in a computer system. First is system controlled protection, this kind of protection is used by the system of its own, to protect the different processes and system resources, software and hardware, for the proper mechanization of the system. Second is controlled by the system administrator, this kind of protection is provided by the system administrator for proper resource utilization, like, hardware and software utilization. System administrator allocates the different memory pages to the different users based on their requirement. System administrator makes available the different system software, databases and application software required by the users based on the actual necessity to the right kind of user. The last kind is the user controlled protection, this kind of protection is done by the users to their own resources and making their resources available to other users. Before providing the control to others, the authorization of the users is checked. All the three kinds of protection mechanisms play their individual important role to guard the computer resources from unauthorized processes.

All the above three kinds of protection mechanisms are implemented in GURU. System protection is implemented of its own, which is done by GURU with the help of a shell, which protects and maintains the various access levels in the system software from the outside processes. The internal processes of GURU are protected by providing the different consistency measures and the privilege levels. The Second kind of protection is made available

	DOM-1	DOM-2	DOM-3	DOM-4	DOM-5
OBJ-1	P-1 P-2	P-3	P-1 P-2 P-3	P-1 P-2	
OBJ-2	NIL	P-1	P-1 P-4	P-1	
OBJ-3	P-1 P-2	P-2	NIL	NIL	
OBJ-4	P-1 P-4	P-2	NIL	NIL	
OBJ-5	P-4	NIL	NIL	NIL	
OBJ-6	NIL	P-1	P-3	NIL	
OBJ-7					

OBJ - n : OBJECT - n
 T - n : TIME - n
 DOM - n : DOMAIN - n
 P - n : PRIVILEGE - n
 NIL : NO PRIVILEGE

LIMITED PRIVILEGE CONTROL IN DYNAMICALLY CHANGING ENVIRONMENT

FIG. 9.1

by the system administrator. In GURU, the system administrator defines the basic access rights of the persons logging in the GURU DDBMS environment. The resource access and manipulation rights are granted by the administrator, which fixes the user under some resource sharing domain. The type of the domain and the available facilities, which can be shared by the person depend on the type of the person and the category under which the person is a registered user. The last category provides the access grant rights to the different domain users by the resource owner. In this kind of protection, the users themselves define the other users who can access owner's resources in the way owner has assigned. The all the three kinds of protection schemes are discussed with the new features in GURU using the object-oriented approach in dynamically changing environments.

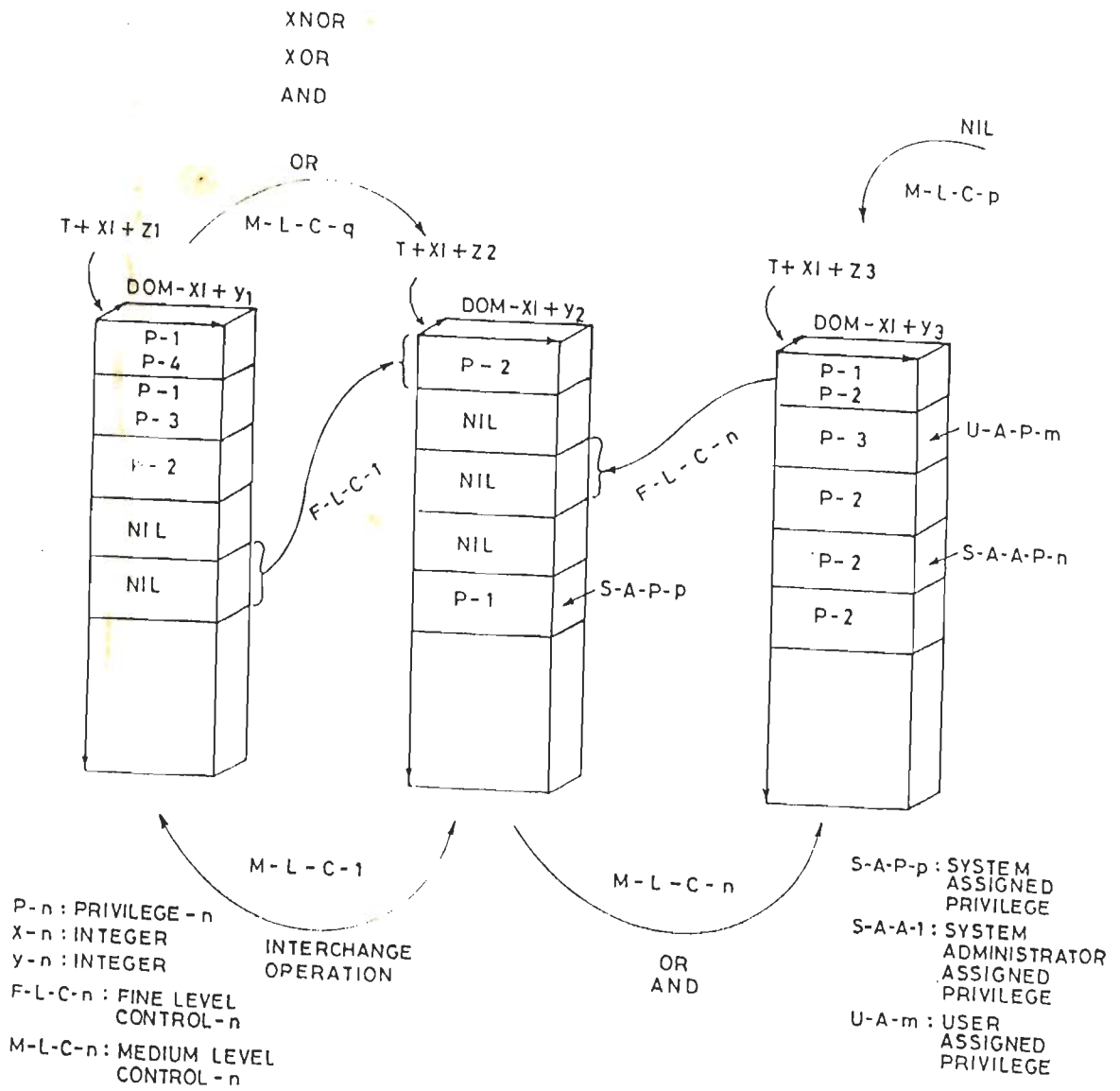
The different considered objects are designed and implemented for a few domains only. The protection scheme for the view is implemented in the form of access matrix. In case of changing environments the few domains are interchanged or they are added to some domains etc. as shown in Fig. 9.1. This provides a set of access matrices for changing domains. This method is only applicable where, the domains vary with the given limits. These limits essentially are very few. Otherwise, it increases the burden of handling a large number of access matrices. And, the different approaches like sparse matrix handling and some other kinds of matrix handling techniques are used to solve the problem of protection. These methods cannot be used to protect data of a heterogeneous database in the object changing environment. Therefore, we have suggested a scheme which requires the structures to be implemented in the different problem domain and can be solved using the concept of object-oriented design philosophy. For this, GURU provides the enormous extended power of

changing structures and the privileges, so that the other changing structures are automatically implemented in the structural object data structures designed and implemented in the changing environment.

9.3 SYSTEM LEVEL PROTECTION

System level protection is implemented with most of the systems working to protect their environment in which the system processes are running. Without this protection, the whole system cannot run smoothly, because, the other application processes can disrupt or over write system memory where the system programs are running and finally the system is crashed. To guard such chaos most of the systems are equipped with the hardware which protects the memory. The latest processors have the in-built facility to protect the memory areas, where the system programs are running. Some special instructions are implemented with the latest processor chips, which distinct the operations being executed in the different regions of the memory and generate interrupts if any process is intended to write in the protected memory zone, where the system programs are running. Besides this, the different privileges and priorities are assigned to the different system processes, which are executed in accordance with the privileges assigned to them.

A limited privilege control in dynamically changing environment is shown in Fig. 9.1. On X-axis, various domains, DOM-1, DOM-2, DOM-3, DOM-4, and DOM-5 are shown. On Y-axis, various objects, OBJ-1, OBJ-2, OBJ-3, OBJ-4, OBJ-5, OBJ-6, and OBJ-7 are shown. The different objects can be allocated to different domains with the provided privileges. At an instant of time, T-1 domain, DOM-1 is assigned an object, OBJ-1 with the



SAMPLE OPERATIONS ON PRIVILEGE
 FIG.9.2

privileges P-1 and P-2 and no other privileges are assigned. Therefore, DOM-1 can operate the OBJ-1 for the privileges P-1 and P-2 only. Similarly, the other domains are given access to the different objects with the privileges assigned. When the time is also considered as a parameter, the different domains are given access to the objects in the time frame specified with the minimum slots as T-1, T-2, T-3 etc. These slots are known as timestamps of the different domains with different objects and the privileges vary accordingly. The three dimensional matrix shown in Fig. 9.1, is very difficult to store, due to its large memory requirement. Therefore, a method is suggested as shown in Fig. 9.2. The vertical columns represent the X-axis and the Y-axis elements as usual from the Fig. 9.1. The different domains and objects at a time slot, known as the timestamp are produced.

The changes in the privileges at the different time slots can be derived as shown in Fig. 9.2. The applicable conditions at the different time slots can be governed with some Boolean expressions, which can be applied and the resultant timestamp privileges can be acquired. These privileges are assigned to the resultant timestamp domains. The three kinds of privileges can be assigned by the different processes running in the system. The first is the system assigned privileges. These privileges can be assigned by the different system processes running in the heterogeneous environment with different authentications. The privileges can be assigned based on certain prior defined rules for the system operation at the various levels in the global environment and are represented with the symbol S-A-P. The next kind of privileges are the system administrator assigned privileges and are represented with symbol S-A-A. These privileges can be assigned by the system administrator at any time to maintain the heterogeneous global

environment. All the resources in the system have to go through the S-A-A assigned privileges. The last kind of the privilege assignment is the user assigned privilege and is represented by a symbol U-A. All users having the system level privilege can assign their resources to any other domain in the system. Therefore, the different users can assign their privileges to any other users working in the heterogeneous environment. This scheme is represented in Fig. 9.2. and the direct arrows illustrate the different time level privilege assignment to the different domains for some privileges over the other objects, with the forced values. The different expressions used in assigning the privileges at the different intervals of time are shown.

The users writing the system level programs are also required to provide the different protection schemes so that the unauthenticated applications cannot be entertained by the designed application procedures. Such kinds of implementations are carried out in GURU, which lets the application programmers define the different levels of protection to their own primitive and complex objects. The following instruction in GSQL (GURU SQL) shows the different levels of protection provided to its primitive objects:

```
define structure marks_sheet
privilege protect
memory sno, name, class, marks
privilege print
device printer_3, printer_7
privilege read
datafiles mohan
privilege general public
procedures read_record, read_marks;
```

The above instruction in GURU defines a class named marks_sheet, which has the memory variables sno, name, class and marks defined with privilege protect. The protect privilege lets the memory variables known in its own class environment and the different environments inheriting this class, marks_sheet. All other objects or classes inheriting the class, marks_sheet can refer all the memory variables but from the other out side of this environment, these memory variables cannot be accessed. The next declaration in the instruction is device with privilege declared as print. This shows that the devices declared in the list of the device names i.e. printer_3, printer_7 can only be referred inside the class, marks_sheet with the privilege, print only. No other operation is allowed on these devices. This scheme protects the devices declared from the other processes. The other clauses in the last instruction are datafiles with privilege declared as read only. The declaration, datafiles is the declaration for directory names of the database fragments. Therefore, the database fragment mohan has only privilege to read its tuples. The procedure declaration in the above instruction is for the procedures defined by the application programmer to handle certain assigned tasks. The procedures read_record and read_marks are separately written in GSQL and have the privilege to be called from any where including the outside environment.

Similar operations as defined above are implemented in the system software, which run on the system to support various system tasks. Further, the different objects can be created using the classes and enforcing the new instructions in the same object to support additional tasks. The following instruction creates an object:

```

create object m_s
privilege protect
include structures marks_sheet
privilege private
memory ch_1
privilege general public
functions test_ty;

```

The above instructions create an object, `m_s`, which inherits the class, `marks_sheet`. The privilege of inherited class is declared as `protect`. Therefore, the primitive objects derived from the structure (class), `marks_sheet` are only known to the object `m_s` and the other objects inheriting this object. The additional memory variable, `ch_1` is only known inside the object, `m_s`. The function `test_ty` is available to all the environments. The table shown in Fig. 7.8, discusses the final inherited privileges of the new classes and objects. Which become finally applicable to the various classes and objects after the inheritance. The another instruction which creates an object from the class inheritance procedure is given below:

```

create object d_r with structures marks_sheet;

```

The object, `d_r` is created with the class, `marks_sheet`. The complete set of primitive objects in the class `marks_sheet` is obtained and the privileges hold the same values as are present in the structures (classes) declaration. Let us discuss the privilege obtained with the inheritance of an object in more than one objects as shown in the Fig. 9.3. The primitive objects of the sub-object are known to the different objects inheriting the sub-object, are different. Because, the privilege levels

associated with the different objects are different. The following are the instructions used to construct the scheme.

```
create object registrar
privilege private
include objects m_s
privilege general public
memory r_marks
procedures give_marks;
```

```
create object student
privilege general public
include objects registrar
memory s_marks
procedures display_marks;
```

```
create object teacher
privilege protect
include objects m_s
privilege protect
memory t_marks
procedures enter_marks;
```

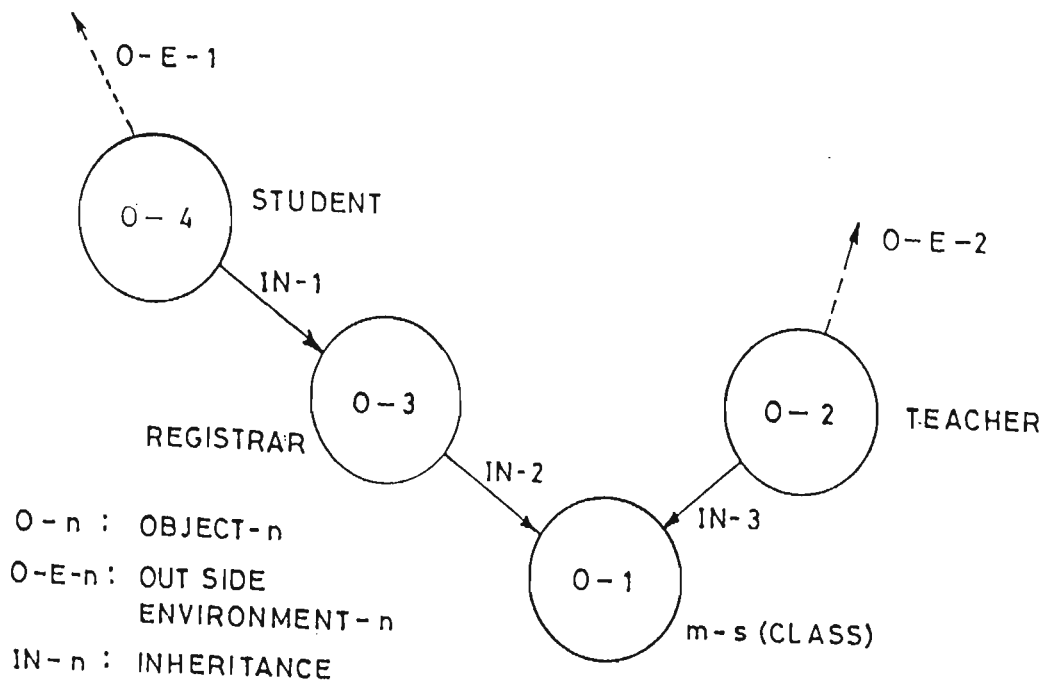
The first instruction creates an object, registrar. This object inherits the other object m_s. The object m_s is created earlier. The only primitive objects declared in the object m_s having the privilege, protect and above will be accessible maximum to the object registrar. Further, the object, student inheriting the object, registrar for the purpose of knowing marks, will be available. The another object teacher is inheriting the object, m_s. This procedure is illustrated to maintain the marks of the students. The teacher enters the marks

for the object, m_s through the procedure, enter_marks of the object, teacher. These marks are accessible to the object, registrar directly and the object, registrar checks these marks through the procedure, give_marks after processing. Later the modified marks or the correct format marks are available to the object, student by the procedure, display_marks. Object, student here cannot know, who has given the marks and actual marks given etc. Therefore, this scheme is successful in providing the marks of the students with proper protection.

9.4 EXTENDED USER DEFINED PROCEDURES

So far, none of the existing distributed database management systems have provided a scheme for the protection of the large number objects in an infinite domain, where the large number of privileges are dynamically assigned to the objects. These privileges are not defined earlier in the system. To handle such kind of environment, a facility known as extended structure facility is available for the dynamically changing object and class structures. This facility provides the scope for the implementation of allocation space for the data structures being maintained and provide a systematic control over them to maintain the consistency in the system. The Fig. 8.6, shows the extended structure facility in GURU.

The changing object structures are implemented in the system by reallocating the space for the required class and the object. The affect on the net system is analyzed from the view point of maintaining consistency in the overall global system. In case, if the consistency of the system disturbs then, the system correction procedure is invoked after checking the privileges of the user requesting to implement such changes into the system.



EXAMPLE INHERITANCE

FIG.9.3

The different privileges can be altered dynamically in the system for supporting such applications.

In Fig. 8.1, the basic structure of an object is shown, which has the facility to change structures, maintaining the strict privacy and the security control on the smallest data item present in the object and the database fragments assigned in the object. The data view is maintained completely with the data structures maintaining the individual components to be protected in the dynamic environment. The objects can be recorded in the virtual memory by the object manager. Therefore, to achieve good performance a complete view can be built with the object. The different inherited objects show view inside a view and so on. These views are protected with proper privacy codes. The class table shown in Fig. 8.6, can be used to obtain the object table, just by replacing the class name with object name. The table consists of a control block with the pointer to the object identifier. This block carries the information regarding the items present in the identifier table. Which identifies the correct sequence of the items present in the table. The different flags shown in Fig. 9.4, present in the table represent the presence of some primitive object declaration. The names, SUP-CC represents Super-Class Count, provides the information of this field presence, similarly, SUB-CC for Sub-Class Count presence, T-F-C for Total Fragment Count presence, F-C for Fields Count presence, D-C for Devices Count presence, P-C for Procedures Count presence, F-C for Functions Count presence, and E-F Extension Field presence are maintained. The Status Flag Register of class name block helps in saving the memory and also speeds up the operations of reading the class structure. The class identifier shows the classes present in the sub-class or the super-class format which further carries a lock entry. This entry

represents the two kinds of locks with the system. One is the normal privilege lock, which finds the various users with the normal privilege levels in the system. The other lock is the key information regarding the external lock. In case any external lock is found at a place, this requests the lock procedure built with the supervisor to check the key of the lock, present in the message format. If the key is found correct then the access to the primitive object is provided otherwise, a resultant message is sent to supply the proper key regarding the object being referred.

At the bottom of the identifier table an extended structure pointer is shown. This structure pointer helps in various ways to maintain the different control blocks in the system. These are given as bellow:

- i. Trigger entries
- ii. External interrupt entries
- iii. Fragment extension privacy codes and the fragment lock qualification
- iv. Field-names, statistical map for the declared fragments
- v. The location table for the same field-names for replication

The trigger entries are recorded to have the various trigger procedures for specific applications. Which are executed on the requirements. Some of these are the requirements for changing the environment. The various environments can be changed dynamically by executing the triggers. These environments can be changed by setting certain given values to the specific objects. Further, it can also invoke some translation procedures to translate or express a query on a given object in the language of

the destination terminal node to which the query would be routed. Some other uses are, to translate the heterogeneous object structure in a given object-oriented DBMS format. So that the object can be processed at the required terminal node with a required DBMS.

Triggers also help in accessing the statistical information from the statistical data recorded regarding the object placement policies at different geographical places on the computer network. This information is supplied to the local central server, which takes steps to communicate with transaction manager to reschedule the various fragments and the objects. The different users are free to make use of the system resources by invoking certain interrupt procedures. For this, an interrupt number is recorded in the interrupt index table. Which is referred to invoke a procedure, if the requester, making an interrupt call, is having a proper privilege level.

The different qualifiers for the tuples are recorded in the extended area with the proper privilege and privacy locks. If a fragment needs to be accessed and it matches a tuple, which has the tuple security lock, the tuple is allowed only if the requester is having a proper privilege to access the tuple. Similarly, the fragment qualifier and the field, both may be locked. The similar, operations are performed to provide access to the requester. There is no limit prescribed in the extended data structures regarding the size and the number of fragment qualifiers for the locking arrangements. The location tables are also maintained, which provide the complete information regarding a field-name and the fragment, which is required by some requester. The terminal node code, DBMS code, and the application names are provided to the server, which manages to

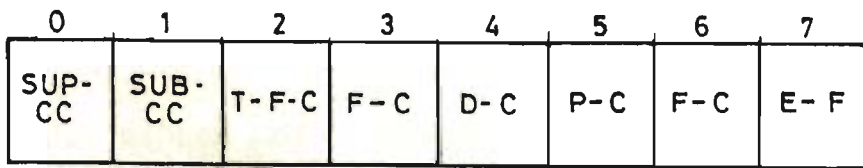
send the whole query after getting it translated, if the translator is found on the current terminal node.

9.4.1 Triggers

Triggers are very helpful in organizing the activities of the objects. Objects can be considered to play in different activities, like providing the facility to support a well formatted database results on the screen, making the privilege changes to affect the environment of the formatting scheme etc. The following are a few examples where triggers can be used, which are recorded in the extended object structures, used for changing the environments dynamically, with the privileges of the objects:

(1) Key Triggers

These triggers help the users to work in various activities to divert the results in the way user requires on the terminal node. Suppose, while getting the results from some database on the launched query, the results may be required to be displayed in an specific format, like giving the one line space between the two records, the title may not be required for the moment, the names are required in capital letters only etc. For such applications, some macros are defined in GURU, these macros can be invoked by the key control procedures. The effect is realized and is taking the all kinds of privileges into account, that is a trigger function may not have the privilege to do so, which may be requested in an unauthorized way. Therefore, the messages are displayed on the control panel for the user to check the keys, which are required to have the proper privileges.



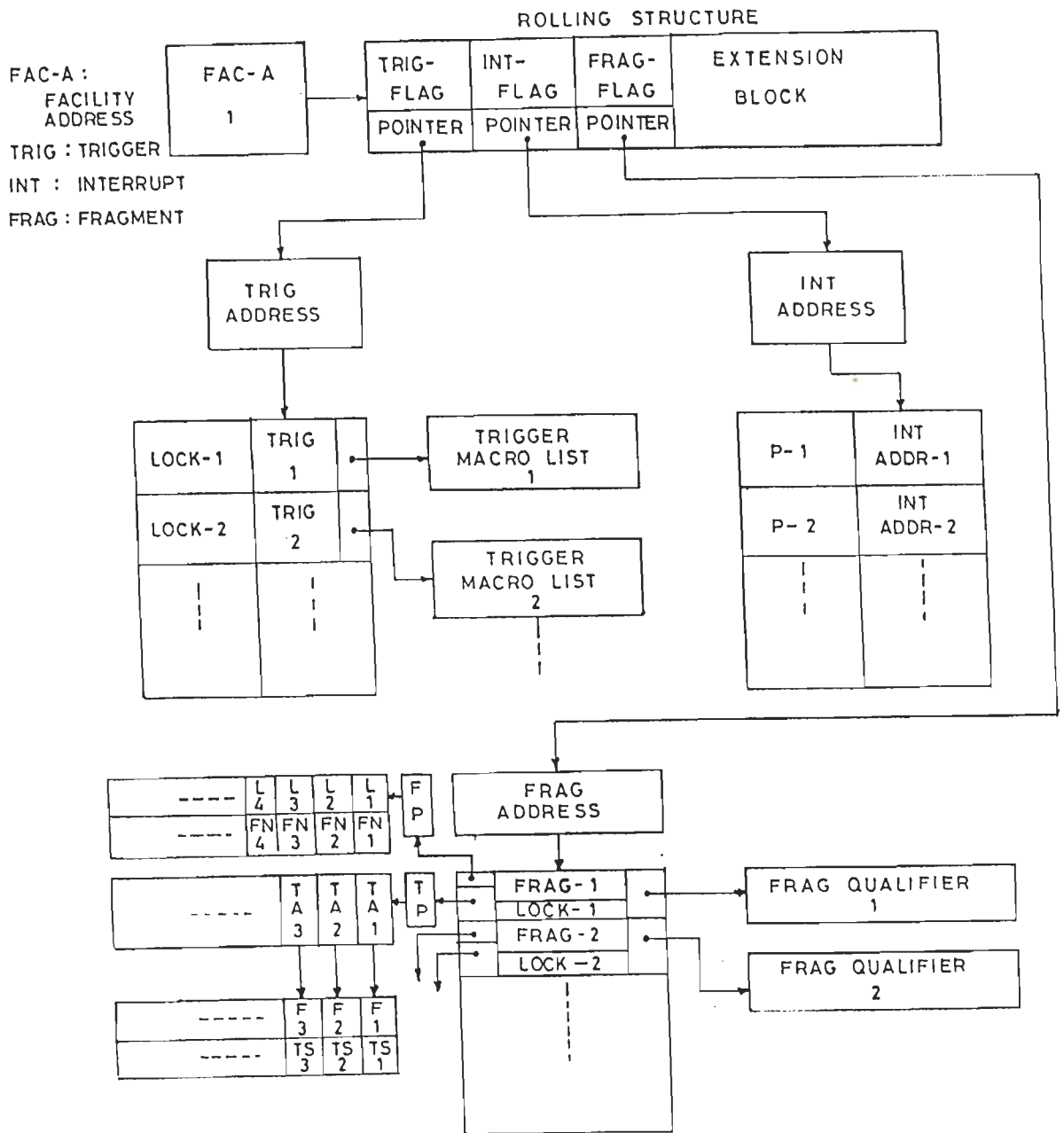
STATUS FLAG REGISTER OF CLASS NAME BLOCK
FIG.9.4

(2) Field Triggers

These triggers invoke the trigger procedures, whenever any fields are referred, which checks the privileges automatically and in the dynamically changing privilege levels, the privileges are set accordingly. These procedures are important for the applications to study the performance of the system on certain access and modification procedures. Individual fields can be chosen to check the performance in some required applications. Therefore, a field known as a tag field is established in the object extended structure, which is further helpful in recording the access or modification time, taken for a particular field. This makes the system well accepted for such kinds of performance analysis and deadlock detection at an appropriate stage.

(3) Block Triggers

These triggers are designed to handle the block activities and protect the actual block operations being performed. The block operations can be considered as some prior defined areas, which are required to be protected and maintained as per the system requirements. These areas can be considered as query area, commit and rollback area, catalog area, log and backlog area, data dictionary area etc. Therefore, the triggers (procedures) can be implemented to handle the most important part of the distributed database management system. The area tools are helpful in maintaining the various versions of the database and the objects in the system. The time based block analysis is done to take care of all the aspects. Therefore, a block for the triggers is implemented in a way to control the overall functions in the dynamically changing environment. Fig.



T-A-n : TERMINAL ADDRESS-n
 F-n : FREQUENCY OF REFERENCE-n
 T S-n : TIME STAMP-n
 FN-n : FIELD NAME-n
 L-n : LOCK NAME-n

F-P : FRAGMENT POINTER
 T-P : TERMINAL POINTER

TRIGGER BASED OBJECT TO CATALOG PROTECTION THROUGH EXTENDED STRUCTURES

FIG. 9.5

9.5, shows the maintenance of the catalog, log and the backlogs being protected and the trigger actions are activated at the various points.

(4) Form Triggers

Form triggers are used to construct a formatted form as per the requirement of the user. These formats are made standard initially and later activated with the present requirements. These form triggers are good for preparing reports and providing the database in the form, to fit them in various environments.

(5) Application Based Triggers

These kinds of triggers can be prepared for any general application, like, checking who are the users on the global system using a particular application environment in a time domain. Therefore, preparing the statistical information for the utility of certain applications at the various geographical terminal nodes on the system. This can be successful in collecting the data which can be analyzed later for certain person domain, application domain, and the resource domain etc. Which becomes applicable to find fruitful results in some environment.

9.5 SYSTEM ADMINISTRATOR AND USER LEVEL PROTECTION

System administrator fixes the different categories of the users in the different user domains, and the different kinds of computer resources in the resource domain. The users can be categorized in terms of the privileges, required to be assigned

for the different computer resources, to be provided to them. The computer resources can be categorized as hardware resources and the software resources. The all hardware resources cannot be allowed to be used by all categories of the users. Similarly, the all software resources cannot be allowed to all. Therefore, the two domains are assigned certain privilege levels. Further, the different users authorized to work on computers for using the DDBMS environment are identified with their identity codes. These identity codes are grouped to form a user domain. Each group must have the same privilege level so, it is called one domain. Therefore, the management of the users and the computer resources with time is performed by the system administrator. The system administrator also maintains the complete privacy and security locks among the different resources and the different users on the global network.

The users also have rights to allow their own resources to the other objects in the system. Like, providing the access rights on the objects owned by the user to the other out side environment. To transfer the privileges, for accessing a required object following are the instructions used in GSQL:

```
export privileges privilege-name to memory  
memory-name-list of object object-name;  
export privileges privilege-name to datafiles  
fragment-name-list of object object-name;  
export privileges privilege-name to devices  
device-name-list of object object-name;  
export privileges privilege-name to procedures  
procedure-name-list of object object-name;  
export privileges privilege-name to functions  
function-name-list of object object-name;
```

The above first instruction exports the privilege specified to the specified memory variables, memory-name-list, which belong to the object having the name specified as object-name. The second instruction above exports the assigned privileges to the fragment names specified as fragment-name-list of the specified object with name as object-name. Similarly, third, fourth and fifth of the above instructions assign the privileges to device-name-list, procedure-name-list and function-name-list respectively of the corresponding objects names specified.

The changes made with the above instructions in the primitive objects are recorded in the object data structures. These changes are maintained with the corresponding version numbers recorded in the backlogs. The backlogs maintain the complete history of the transaction made throughout the life of the applications. Any changes made in the system are recorded and in case, if a previous version is required, then the help of backlogs is taken. With most of the cases where transactions are not committed. The backlogs help in the recovery of the database records to achieve complete rollback. These versions are maintained with the timestamps. For the changes made in the object structures, the past object is maintained with the last time-stamp. Therefore, the all kinds of data can follow the right time based pattern.

As the privileges are changed for the different objects, the similar changes can also be made in the class structures, thereby changing the all objects derived from the changed class structures if changes occur in the classes. Following are the instructions which perform the changes with the class primitive objects as defined:

```
export privileges privilege-name to memory
memory-name-list of structure class-name;
export privileges privilege-name to datafiles
fragment-name-list of structure class-name;
export privileges privilege-name to devices
device-name-list of structure class-name;
export privileges privilege-name to procedures
procedure-name-list structure class-name;
export privileges privilege-name to functions
function-name-list of structure class-name;
```

The above instructions work exactly alike as in case of objects except instead of the object name, structure (class) names are mentioned.

The instructions, which display the different primitives of the primitive objects present in the objects and the display instructions also display the complete primitive object names in the form of assigned names to the primitive objects in classes are discussed as following:

```
display privilege of object object-name in structure
class-name;
display privilege of structure class-name in structure
class-name;
display privilege of object object-name in object
object-name;
display privilege of structure class-name in object
object-name;
display privilege of memory mem-var in object
object-name;
```

display object object-name structure;
display structure class-name;
display memory of object object-name;
display memory of structure class-name;
display procedures of object object-name;
display procedures of structure class-name;
display functions of object object-name;
display functions of structure class-name;
display devices of object object-name;
display devices of structure class-name;

The following are the instructions, which remove the objects and the classes:

delete objects object-name-list;
delete objects object-name-list with key key-name;
delete structures class-name-list;
delete structures class-name-list with key key-name;

The above first instruction removes the all objects which are present in the name list, object-name-list. The second instruction removes the all classes which are present in the class-name-list. The all objects removed from the list are also removed from the object identifier table. In this table, where ever these objects are found, they are all removed. The remaining structure left becomes the latest object structure. The similar is the case with the classes. All the classes deleted are removed from the class identifier table, and the inherited classes are also removed from the hierarchy of the objects. The remaining structure of the objects after removal of all the classes is the latest object structure. The objects and classes created with locks are only removed, if the right key of the lock is

specified. If no key or the wrong key is specified, it returns a message that the deletion is unsuccessful.

9.6 MESSAGE PROTECTION

All kind of communications among the various objects are done in the form of messages. The messages are sent from the source object to the destination object through the object hierarchy. The messages are communicated and accepted at the destination object, only if the proper privileges and the keys of the locks are specified. The communication is provided with the standard communication protocol and with the standard message formats adopted in GURU with the heterogeneous global communication network. The messages are treated in the same way, even if they come from the same terminal node or from the other terminal nodes placed geographically apart. Quite often, the messages are communicated to more than one terminal node at the same time. Therefore, the message packet maintains the number of addresses of the terminal nodes and the DBMS codes to which the message is being routed. After receiving the message by one terminal node on the network, the message is further communicated to the other terminal nodes, till all the terminal nodes receive the message packet. Each message packet consists of a packet number and the overall size of the packet being circulated on the network. In case if any of the message packets reach out of sequence then the sequence is automatically maintained after tallying the packet numbers. If a packet is not received by any terminal node, then the message is sent to the source terminal node to resend the message packet of the specified number, which was not successfully received. Later, the packet is resent. This communication takes some time. The privileges can be modified, other locks can be changed, objects and classes can be removed, primitive objects

can be read or written or executed etc. All such operations are performed through the message passing scheme implemented for the objects.

Certain procedures are not available at some terminal node. But these are available at some remote terminal nodes. Then the remote procedure calls (RPC) are issued. These procedures are executed at the different terminal nodes with the specified arguments from the source node through the message packet. The results are received at the local terminal node as if every thing is carried out by the local terminal node. This transparency is also made available with the help of proper privilege levels, being maintained throughout the network.

9.6.1 Ciphering

This is the technique used to encrypt the long data with some small codes. The key, on which the changes are made is protected. Therefore, the confidential data cannot be known outside. All the object and the class tables are encrypted. Therefore, in the faulty conditions like, some deadlock etc., the information from the system cannot be leaked. The names written in the tables are also encrypted. This process, makes the system secure to some extent from the outside threat to the data privacy and security, during the deadlock conditions.

9.6.2 Port Arbitration

This is the technique used to protect the message object from the outside network threat to the privacy and security of the data. The message enters to the computer terminal and passes through the input port. The usual procedure is that

the results are also send back from the same port or some other fixed port. But, in this procedure any one can temper the message by accessing the port. Therefore, a method suggested is changing the ports with some random number function for both the operations, for inputting and outputting the data. The pure random function is generated as $R(a)$. Where, a is the address of the input port. Therefore, the output port address would be a pure random port having the address, $R(a)$ among the available ports on the terminal node.

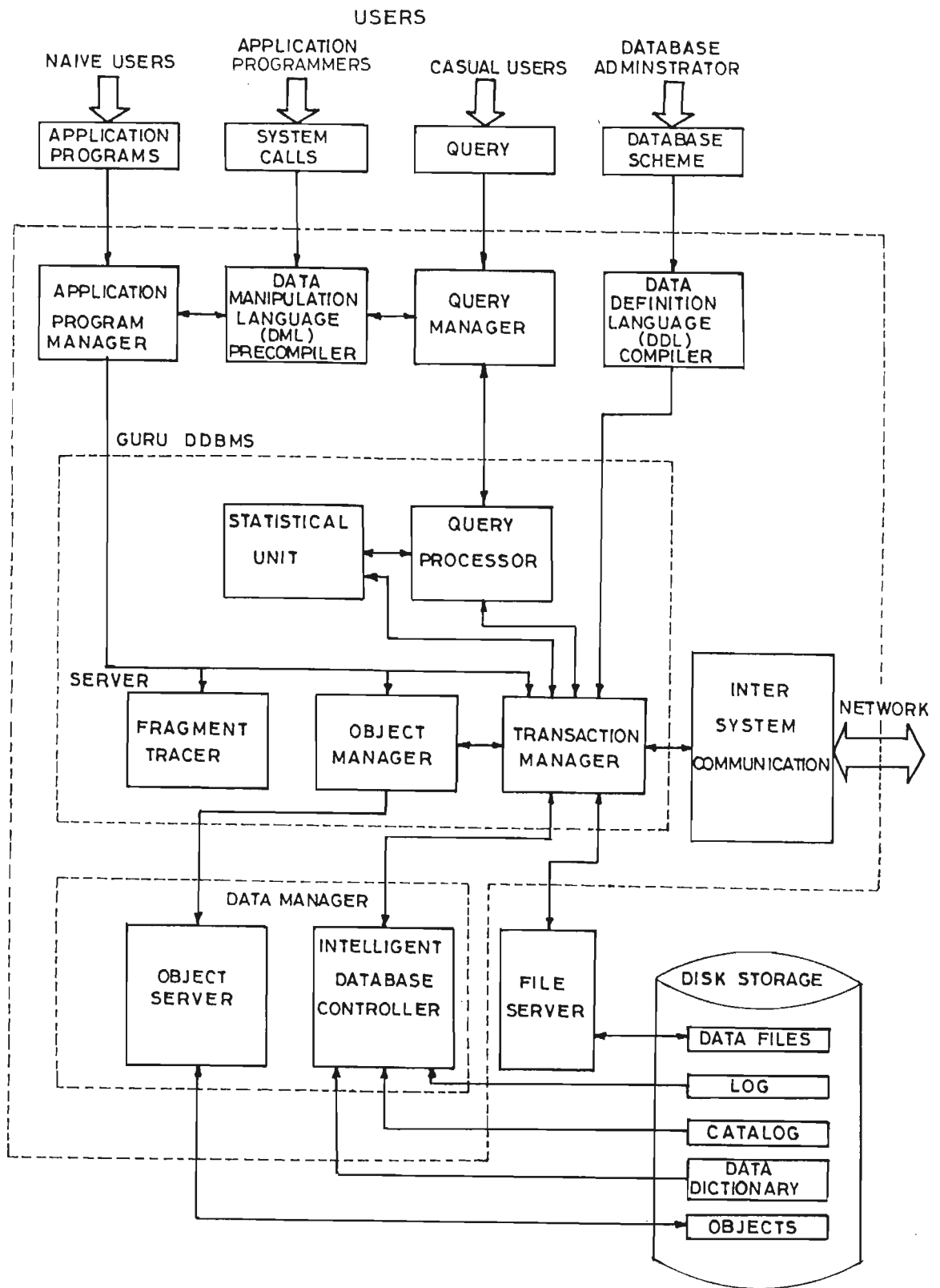
9.7 PROBLEM HANDLING

Here, we consider a problem of changing environment of the various privilege levels in a global problem domain. To change the privileges in the dynamic environment, the system triggers are implemented, which will handle the problem changes in a real problem, based on analysis. This analysis is carried out for a particular application domain adjustment. Like, certain privilege dependency in mutual domain is studied. Later it fixes up certain expression based conditions. These conditions are evaluated on the different object environment, which changes dynamically. Those conditions are picked up by the trigger procedures set for the particular purpose. The object environment is derived by the trigger procedures by invoking those procedures from the appropriate level in the object. These procedures are activated to set the different privileges and the locks, if required at some places by the objects.

9.8 CONCLUSION

The proposed methods are very effective in protecting the dynamically varying object environment using privileged

control. The integrity and consistency of the primitive objects can be maintained in the dynamically changing object environment. The object extension arrangement provides the wide range of triggers operated by the user and the system level operations in the heterogeneous environment. Therefore, a varying privilege environment is successfully maintained with the help of individual triggers. The different, changing views of the object can be maintained. The backlogs help in providing the previous versions of the objects. The backlogs are also protected with the object protection schemes. The discussed approaches can be used with other object-oriented heterogeneous DDBMSs.



GURU DISTRIBUTED DATABASE MANAGEMENT SYSTEM
FIG. 10-1

CONCLUSION AND FUTURE SCOPE OF WORK

10.1 CONCLUSION

The object-oriented heterogeneous distributed database management system (OOHDDBMS) has been designed and implemented successfully. Its overall *system block diagram* has been illustrated in Fig. 10.1. The salient features of GURU DDBMS are:

- * An efficient query automation technique has been proposed.

The intelligent design of indexes and their loading into memory with database fragments has been provided. The right selection of indexes on different orders of the attributes decides the overall efficiency. Therefore, the pattern selection using the knowledgebase with the relative indexes and foreign keys reduces the overall query response time. A wide range of instructions have been implemented to support query automation. The new intelligent approach for indexes is better than the present OOHDDMSs (For example, Gemstone and Encore etc.).

- * A concept, intelligent schema has been introduced and implemented.

The intelligent schema carries the complete information about the various kinds of changes in the system with the complete past history. And it provides the directives to the server for splitting query appropriately into different intraqueries, based on the fragment allocation status. A concept of virtual

tasks, virtual programmers and the virtual terminal nodes is introduced in the system.

- * An efficient algorithm for heterogeneous object-oriented integrated schema has been implemented.

A procedure is implemented which integrates a complete set of database fragments into a cohesive structure. Which works as the best available structure for the intraqueries or interqueries, to process the horizontal, vertical and mixed kind of basic fragments. The replicated and overlapped fragments are also processed. The horizontal and mixed fragments use multiple keys for their allocation. Schema integration in heterogeneous environment is not better in any existing system (Orion, Iris, Vbase etc.) than the suggested method.

- * Some complexity measures are successfully employed for dynamically changing environment.

The different procedures are implemented to tackle complexity of a heterogeneous system. A procedure for efficient allocation of heterogeneous fragments is illustrated. An approach is suggested to implement intensional and extensional notions. Procedures are explained to resolve the complexity in the system using trigger and extended object-structure methods. The different kinds of triggers which include the base, arbitrary and programmable methods are implemented in the system. GURU provides better trigger methods over Vbase, Orion etc.

- * Message-task scheduling is implemented through active object procedures.

An approach is suggested to implement the message as a task. The variable size message packets can be handled with the changing object structures, where objects act as active objects and execute the task, as the distributed shared memory process. The dynamically changing structures are trigger controlled. The task scheduling with synchronously and asynchronously exited methods for concurrency control are implemented. GURU provides message-task concept which does not exist in Vbase, O₂, Orion, Gemstone, Encore and Iris.

* Dynamically changing environment with extended object shell has been implemented.

Objects change their structures in the dynamically changing environments by changing the properties and behavior. A method is explained which changes the object structures without any limits. The method uses triggers for the purpose. The different versions of the objects are maintained in the system. The metaclass support is provided in the system. The approach is totally new and better than MORE, which does not support unlimited object structure modifications.

* A scheme is implemented to provide optimal privacy and security in a large application domain.

The complexity of access procedures and the degree of prevention from an unauthorized process with a limited privilege level is suggested. So that, with dynamically changing access procedure privileges, the proper privacy and security to the primitive and complex objects is provided. The different schemes are suggested to protect the objects from unauthorized access like by providing the random allocation to ports for the differ-

ent threads, the data ciphering, encrypting the object names, encrypting the process addresses, the primitive and complex object locking, and the lock exchanging. The existing objects-oriented databases do not provide better privacy and security of objects than the suggested approach.

GSQL supports a very large instruction set having most of the features of modern compilers. It provides a feature of building overlays automatically; makes available the large memory to place the object structures; a version support is provided with objects; makes a big room for placing the object and class tables. The objects are using the most recent structures to support the environment for modern CAD/CAM and CAI applications. A wide range of programming capabilities are provided with GURU. A virtual programmer concept enhances the features in the system. The dynamically created knowledgebase and A.I. techniques have made the enormous power to deal with heterogeneous data. Since, GURU is written in C language, it is portable on any computer system running a C compiler. The system has a wide range of privacy and security locks which add to its qualifications for extremely confidential data handling. GSQL uses mostly close to the semantic constructs and provide a vast range of *clause switching* therefore, it is better userfriendly. A variety of applications can be programmed in GSQL because, it carries a wide range of instruction support. The enhanced features in object-oriented programming have made the system extremely useful for the engineering design applications.

The view design (heterogeneous schema design) attracts the commercial organizations to choose their systems in the distributed environment. The powerful library functions have provided a facility to test the different environment support to

provide foolproof functioning in the adverse conditions with a large number of integrated environments.

The block structuring, deadlock detection and prevention have made the system working in the adverse conditions on the network. The system competes better with a few existing prototype models, such as GEMSTONE, ORION, MORE and O₂ DDBMSs.

10.2 FUTURE SCOPE OF WORK

In this area, further research can be done to develop some approaches, to find the ideal solution, for the queries not having the entities and attributes at all in the system. This can be done to find out a way, so that the system administrator finds the most likely required data on the system, which can tune the system. Some new ways can be devised to optimize query fragmentation for the adverse network conditions.

Efforts can be made in optimizing the size of the fragments with different network conditions. Various models can be studied and implemented to show the best performance of the database manipulation considering the local autonomy into account. The new ways can be devised to study behavioral aspects of object-oriented heterogeneous schema and the linked application domains.

Problems can further be extended for optimizing size of the fragments with different network conditions and resolving the complex sets, which require heterogeneous integration to their schemas. Various models can be studied and implemented to show the best performance of the database manipulation considering the local autonomy into account.

Possibilities can be explored to suggest some useful methods for resolving the complexities of complex sets in the integrated heterogeneous environment to achieve site autonomy, and efficiency. Various models can be studied and implemented to show the best performance of the database manipulation considering the local site autonomy into account.

Schemes can be suggested for providing the fault tolerance with the system using threads and trigger procedures.

New techniques can be developed to maintain the efficient memory allocation policies for object mapping. The triggers can be used with other tools to check the system performance in the heterogeneous environment.

A system can be realized in changing environment for different objects with a large problem domain. Further, performance of such system can be tested with different kinds of problem domains. The performance with the different kind of privacy and security locks in dynamically changing environment on different network conditions can be tested with the different test sets and the areas can be devised for further improvement for GURU OOHDBMS.

APPENDIX - A

A sample problem using GSQL is given bellow:

A shopkeeper has two sections (purchase and sales sections) connected through a network where he purchases some motor parts and sales them after taking 7 percent profit. The sales section has to get a list of parts from the purchase section. The following GSQL instructions perform this task.

```
procedure main
define structure purchase
datafiles p_f
fields p_n, p_n, p_p
devices printer;

define structure sales
datafiles s_p
fields p_n, p_na, p_s
procedure sales_pro
devices printer1;

create object purc_p with structures purchase;
create object sale_p with structures sales;
do sales_pro of sales_p
?"Final task"

procedure sales_pro of sales_p
select alias pp
use pf index pi
select alias sp
use s_p index si
```

```
set relation to p_n into pp
do while .not. eof()
p_s=0.7*pp->p_p
? p_n, p_na, p_s, pp->p_p
skip
enddo
?"Finished job"
```

APPENDIX - B

The following data structures are used for GSQL instructions:

!

<21>

&&

<09>

*

<09>

?

0

<36>

FUNCTION <57>

PICTURE <39>

STYLE <40>

AT <01>

??

0

<36>

FUNCTION <57>

PICTURE <39>

STYLE <40>

AT <01>

???

0

<36>

%

<41> TO <41> DOUBLE

<41> TO <41> PANEL
<41> TO <41> <42> COLOR <07>
<41> TO <41>
%
<41> CLEAR TO <41>
%
<41> FILL TO <41>
COLOR <07>
@
<41>
SAY <36>
FUNCTION <57> PICTURE <39>
PICTURE <57>
GET <16> FUNCTION <57> PICTURE <39> RANGE <05>
GET <16>
VALID <03> ERROR <00>
VALID <03>
WHEN <03>
DEFAULT <01>
MESSAGE <00>
OPEN WINDOW <06>
WINDOW <06>
COLOR <03>
ACCEPT
<14>
TO <06>
ACTIVATE
MENU <28>
PAD <29>
ACTIVATE
POPUP <28>
ACTIVATE

SCREEN
ACTIVATE
WINDOW ALL
WINDOW <27>
ALTER *
TABLE <04>
ADD <48>
APPEND
0
BLANK
APPEND
FROM ?
FOR <03>
DELIMITED WITH BLANK
DELIMITED WITH <00>
TYPE SDF
TYPE DIF
TYPE SYLK
TYPE WKS
SDF
DIF
SYLK
WKS
APPEND
FROM <04>
FOR <03>
DELIMITED WITH BLANK
DELIMITED WITH <00>
TYPE SDF
TYPE DIF
TYPE SYLK
TYPE WKS

SDF
DIF
SYLK
WKS
APPEND
FROM ARRAY <33>
FOR <03>
APPEND
MEMO <34> FROM <04>
OVERWRITE
ASSIST
0
AVERAGE
<02>
WHILE <03>
FOR <03>
TO ARRAY <33>
TO <13>
<13>
BEGIN
TRANSACTION <49>
BROWSE
0
FIELDS <52>
LOCK <01>
FREEZE <16>
WIDTH <01>
WINDOW <27>
NOINIT
NOFOLLOW
NOMENU
NOAPPEND

NOEDIT
NODELETE
NOCLEAR
COMPRESS
FORMAT
CALCULATE
<02>
FOR <03>
WHILE <03>
TO <13>
TO <33>
<13>
CALL
<17>
WITH <00>
WITH <06>
CANCEL
0
CASE
<03>
CHANGE
<02>
WHILE <03>
FOR <03>
FIELDS <16>
NOINIT
NOFOLLOW
NOMENU
NOAPPEND
NOEDIT
NODELETE
NOCLEAR

CLEAR
0
ALL
FIELDS
GETS
MEMORY
TYPEAHEAD
MENUS
POPUPS
WINDOWS
CLOSE
ALL
ALTERNATE
DATABASES
FORMAT
INDEX
PROCEDURE
CLOSE *
<37>
COMPILE
<04>
RUNTIME
CONTINUE
0
CONVERT
0
TO <01>
TO <06>
COPY
FILE <04> TO <04>
COPY
INDEXES <13>

TO <04>
COPY
MEMO <34> TO <04>
ADDITIVE
COPY
STRUCTURE TO <04>
FIELDS <16>
COPY
STRUCTURE EXTENDED TO <04>
COPY
TAG <06>
OF <04>
TO <04>
COPY
TO ARRAY <33>
<02>
FOR <03>
WHILE <03>
FIELDS <16>
COPY
TO <04>
FIELDS <16>
WHILE <03>
FOR <03>
DELIMITED WITH BLANK
DELIMITED WITH <00>
TYPE SDF
TYPE DIF
TYPE SYLK
TYPE WKS
SDF
DIF

SYLK
WKS
COUNT
<02>
WHILE <03>
FOR <03>
TO <06>
CREATE
<04>
CREATE
<04> FROM <04>
CREATE
APPLICATION <04>
CREATE
LABEL <04>
CREATE
QUERY <04>
CREATE
REPORT <04>
CREATE
SCREEN <04>
CREATE
VIEW <04>
CREATE
DATABASE <04>
CREATE *
UNIQUE
INDEX <04> ON <04> <24>
CREATE *
SYNONYM <34> FOR <04>
CREATE *
TABLE <04> <48>

CREATE *
VIEW <04>
<16>
AS SELECT
WITH CHECK OPTION
CREATE
VIEW <04> FROM ENVIRONMENT
DBCHECK
<04>
DBDEFINE
<04>
DEACTIVATE
MENU
DEACTIVATE
POPUP
DEACTIVATE
WINDOW ALL
WINDOW <27>
DEBUG
<04>
WITH <13>
DECLARE
<50>
DECLARE *
CURSOR <37>
FOR UPDATE OF <16>
ORDER BY <24>
FOR SELECT
DEFINE
BAR <01> OF <30> PROMPT <14>
MESSAGE <00>
SKIP FOR <03>

SKIP
DEFINE
BOX FROM <01> TO <01> HEIGHT <01>
AT LINE <13> SINGLE
AT LINE <13> DOUBLE
AT LINE <13> <10>
AT LINE <13>
DEFINE
MENU <28>
MESSAGE <00>
DEFINE
PAD <29> OF <28> PROMPT <00>
AT <41>
MESSAGE <00>
DEFINE
POPUP <30> FROM <41> TO <41>
MESSAGE <00>
PROMPT FIELD <16>
PROMPT FILES LIKE <22>
PROMPT FILES
PROMPT STRUCTURE
DEFINE
WINDOW <27> FROM <41> TO <41>
DOUBLE
PANEL
NONE
<25>
COLOR <07>
DEFINE
OBJECT <43>
INCLUDE OBJECTS <47>
DATAFILES <04>

FIELDS <16>
MEMORY <13>
PROCEDURES <45>
FUNCTIONS <46>
PRIVILEGE <44>
DELETE
<02>
WHILE <03>
FOR <03>
DELETE *
FROM <20>
WHERE <03>
DELETE *
FROM <04> WHERE CURRENT OF <37>
DELETE
TAG
<06> OF <04>
DIR
0
<22>
DIRECTORY
0
ON <08>
<08>
LIKE <22>
<22>
DISPLAY
FILES
LIKE <22>
<22>
TO PRINTER
TO FILE <04>

DISPLAY
HISTORY
LAST <01>
TO PRINTER
TO FILE <04>
DISPLAY
MEMORY
TO PRINTER
TO FILE <04>
DISPLAY
STATUS
TO PRINTER
TO FILE <04>
DISPLAY
STRUCTURE
IN <20>
TO PRINTER
TO FILE <04>
DISPLAY
USERS
DISPLAY
<02>
WHILE <03>
FOR <03>
TO PRINTER
TO FILE <04>
OFF
FIELDS <16>
<16>
DO
CASE
DO

```
WHILE <03>
DO
<23>
WITH <13>
DROP *
DATABASE <04>
DROP *
INDEX <06>
DROP *
SYNONYM <06>
DROP *
TABLE <04>
DROP *
VIEW <04>
EDIT
<02>
WHILE <03>
FOR <03>
FIELDS <16>
NOINIT
NOFOLLOW
NOMENU
NOAPPEND
NOEDIT
NODELETE
NOCLEAR
EJECT
0
PAGE
ELSE
0
END
```

TRANSACTION

ENDCASE

0

ENDDO

0

ENDIF

0

ENDPRINTJOB

0

ENDSCAN

0

ENDTEXT

0

ERASE

?

ERASE

<04>

EXIT

0

EXPORT

TO <04>

<02>

FOR <03>

WHILE <03>

TYPE PFS

TYPE DBASE

TYPE FW2

TYPE RPD

PFS

DBASE

FW2

RPD

FIELD <16>
FETCH *
<37> INTO <13>
FIND
<36>
FUNCTION
<35>
GO
RECORD <36> IN <20>
<36> IN <20>
TOP IN <20>
TOP
BOTTOM IN <20>
BOTTOM
<36>
GOTO
RECORD <36> IN <20>
<36> IN <20>
TOP IN <20>
TOP
BOTTOM IN <20>
BOTTOM
<36>
GRANT *
ALL PRIVILEGES
ALL
<44>
ON TABLE <12>
ON <12>
TO PUBLIC
<53>
WITH GRANT OPTION

HELP
<05>
IF
<03>
IMPORT
FROM <04>
TYPE PFS
TYPE DBASE
TYPE FW2
TYPE RPD
TYPE WK1
PFS
DBASE
FW2
RPD
WK1
INCLUDE
<12>
INDEX
ON <16> TO <04>
ON <16> TAG <34> OF <04>
UNIQUE
DESCENDING
INPUT
<14>
TO <06>
INSERT
BLANK
BEFORE
INSERT *
INTO <04>
<16>

VALUES <05>
INSERT *
INTO <04>
<16>
SELECT
JOIN
WITH <20> TO <04> FOR <03>
FIELDS <16>
LABEL
FORM ?
LABEL
FORM <04>
<02>
WHILE <03>
FOR <03>
SAMPLE
TO PRINTER
TO FILE <04>
LIST
HISTORY
LAST <01>
TO PRINTER
TO FILE <04>
LIST
MEMORY
TO PRINTER
TO FILE <04>
LIST
STATUS
TO PRINTER
TO FILE <04>
LIST

STRUCTURE
TO PRINTER
TO FILE <04>
LIST
USERS
TO PRINTER
TO FILE <04>
LIST
0
<02>
WHILE <03>
FOR <03>
TO PRINTER
TO FILE <04>
OFF
FIELDS <16>
<16>
LOAD *
DATA FROM <04>
DELIMITED WITH BLANK
DELIMITED WITH <00>
TYPE SDF
TYPE DIF
TYPE SYLK
TYPE WKS
TYPE DBASEII
TYPE FW2
TYPE RPD
TYPE WK1
SDF
DIF
SYLK

WKS
DBASEII
FW2
RPD
WK1
INTO <04>
LOAD
<04>
LOCATE
<02>
WHILE <03>
FOR <03>
LOGOUT
0
LOOP
0
MODIFY
APPLICATION ?
APPLICATION <04>
MODIFY
COMMAND <04>
WINDOW <27>
MODIFY
FILE <04>
WINDOW <27>
MODIFY
STRUCTURE <04>
MODIFY
LABEL ?
LABEL <04>
MODIFY
QUERY ?

QUERY <04>
MODIFY
REPORT ?
REPORT <04>
MODIFY
SCREEN ?
SCREEN <04>
MODIFY
VIEW <04>
MOVE
WINDOW <06> TO <41>
NOTE
<09>
ON
ERROR <21>
ON
ESCAPE <21>
ON
KEY <21>
KEY LABEL <00> <21>
ON
PAD <29> OF <28>
ACTIVE POPUP <30>
ON
PAGE
AT LINE <01> <21>
ON
READERROR <21>
ON
SELECTION PAD <29> OF <28> <21>
ON
SELECTION POPUP ALL <21>

SELECTION POPUP <30> <21>
OPEN *
<06>
OTHERWISE
0
PACK
0
PARAMETERS
<13>
PLAY
MACRO <06>
PRINTJOB
0
PRIVATE
<13>
PRIVATE
ALL
PRIVATE
ALL LIKE <22>
ALL EXCEPT <22>
PROCEDURE
<23>
PROTECT
0
PUBLIC
ARRAY <51>
PUBLIC
<13>
QUIT
0
READ
0

SAVE
RECALL
<02>
WHILE <03>
FOR <03>
REINDEX
0
ALL
RELEASE
MODULE <17>
RELEASE
MENUS <28>
RELEASE
POPUPS <30>
RELEASE
WINDOWS <27>
RELEASE
ALL LIKE <22>
ALL EXCEPT <22>
ALL
RELEASE
<13>
RENAME
<04> TO <04>
REPLACE
FIELD <34> WITH <36>
<02>
WHILE <03>
FOR <03>
REPORT
FORM ?
REPORT

FROM <04>
<02>
WHILE <03>
FOR <03>
PLAIN
HEADING <00>
NOEJECT
TO PRINTER
TO FILE <04>
SUMMARY
RESET
IN <20>
RESTORE
FROM <04>
ADDITIVE
RESTORE
MACROS FROM <04>
RESTORE
WINDOW ALL FROM <04>
RESTORE
WINDOW <27> FROM <04>
RESUME
0
RETRY
0
RETURN
<36>
RETURN
0
TO MASTER
TO <23>
REVOKE

ON TABLE <12>
ON <12>
&
FROM PUBLIC
FROM <53>
&
ALL PRIVILEGES
ALL <44>
ALL
ROLLBACK
0
<04>
ROLLBACK *
WORK
RUN
<21>
RUNSTATS *
<04>
SAVE
TO <04>
TO ALL LIKE <22>
TO ALL EXCEPT <22>
SAVE
MACRO TO <04>
SAVE
WINDOW ALL TO <04>
WINDOW <13> TO <04>
SCAN
0
<02>
WHILE <03>
FOR <03>

SEEK
<36>
SELECT *
ALL <16>
DISTINCT <16>
<16>
FROM <12>
WHERE <03>
CONNECTED BY <03> START WITH <03>
CONNECTED BY <03>
GROUP BY <16> HAVING <03>
UNION SELECT
INTERSECT SELECT
MINUS SELECT
ORDER BY <24>
FOR UPDATE OF <16> NOWAIT
FOR UPDATE OF <16>
SAVE TO TEMP <04> <16>
KEEP
SELECT
ALIAS <20>
SET
0
SET
ALTERNATE ON
ALTERNATE OFF
SET
ALTERNATE TO <04>
ADDITIVE
SET
AUTOSAVE ON
AUTOSAVE OFF

SET
BELL ON
BELL OFF
SET
BELL TO <13>
SET
BLOCKSIZE TO <06>
SET
BORDER TO SINGLE
BORDER TO DOUBLE
BORDER TO PANEL
BORDER TO NONE
SET
BORDER TO <25>
SET
CARRY ON
CARRY OFF
SET
CARRY TO <16>
ADDITIVE
SET
CATALOG ON
CATALOG OFF
SET
CATALOG ?
CATALOG TO <04>
SET
CENTURY ON
CENTURY OFF
SET
CLOCK ON
CLOCK OFF

CLOCK TO <13>
SET
COLOR ON
COLOR OFF
COLOR TO <07>
COLOR OF <54> TO <07>
SET
CONFIRM ON
CONFIRM OFF
SET
CONSOLE ON
CONSOLE OFF
SET
CURRENCY LEFT
CURRENCY RIGHT
CURRENCY TO <00>
SET
DATE TO AMERICAN
DATE AMERICAN
DATE TO ANSI
DATE ANSI
DATE TO BRITISH
DATE BRITISH
DATE TO FRENCH
DATE FRENCH
DATE TO GERMAN
DATE GERMAN
DATE TO INDIAN
DATE INDIAN
DATE TO JAPAN
DATE JAPAN
SET

DEBUG ON
DEBUG OFF
SET
DECIMALS TO <01>
SET
DEFAULT TO <08>
SET
DELETED ON
DELETED OFF
SET
DELIMITERS ON
DELIMITERS OFF
DELIMITERS TO <00>
DELIMITERS TO DEFAULT
SET
DESIGN ON
DESIGN OFF
SET
DEVELOPMENT ON
DEVELOPMENT OFF
SET
DEVICE TO PRINTER
DEVICE TO SCREEN
DEVICE TO FILE <04>
SET
DISPLAY TO MONO
DISPLAY TO COLOR
DISPLAY TO EGA25
DISPLAY TO EGA43
DISPLAY TO MONO43
SET
DOHISTORY ON

DOHISTORY OFF
SET
ECHO ON
ECHO OFF
SET
ENCRYPTION ON
ENCRYPTION OFF
SET
ESCAPE ON
ESCAPE OFF
SET
EXACT ON
EXACT OFF
SET
EXCLUSIVE ON
EXCLUSIVE OFF
SET
FIELDS ON
FIELDS OFF
FIELDS TO ALL LIKE <22>
FIELDS TO ALL EXCEPT <22>
FIELDS TO ALL
FIELDS TO <16>
FIELDS TO
SET
FILTER TO FILE ?
FILTER TO FILE <04>
FILTER TO <03>
SET
FIXED ON
FIXED OFF
SET

FORMAT TO ?
FORMAT TO <04>
SET
FULLPATH ON
FULLPATH OFF
SET
FUNCTION <57> TO <00>
SET
HEADING ON
HEADING OFF
SET
HELP ON
HELP OFF
SET
HISTORY ON
HISTORY OFF
HISTORY TO <01>
SET
HOURS TO 12
HOURS TO 24
SET
INDEX TO ?
INDEX TO <12> ORDER TAG <04> OF <04>
INDEX TO <12> ORDER <04> OF <04>
INDEX TO <12> ORDER <04>
INDEX TO <12>
INDEX TO
SET
INSTRUCT ON
INSTRUCT OFF
SET
INTENSITY ON

INTENSITY OFF
SET
LOCK ON
LOCK OFF
SET
MARGIN TO <01>
SET
MARK TO <00>
SET
MEMOWIDTH TO <01>
SET
MENU ON
MENU OFF
SET
MESSAGE TO <00>
SET
NEAR ON
NEAR OFF
SET
ODOMETER TO <01>
SET
ORDER TO TAG <06> OF <04>
ORDER TO <01>
ORDER TO <06>
ORDER TO <04>
SET
PATH TO <18>
SET
PAUSE ON
PAUSE OFF
SET
POINT TO <00>

SET
PRECISION TO <01>
SET
PRINTER ON
PRINTER OFF
PRINTER TO FILE <04>
PRINTER TO <08>
SET
PROCEDURE TO <04>
SET
REFRESH TO <01>
SET
RELATION TO <16> INTO <20>
RELATION TO <56> INTO <20>
RELATION TO <01> INTO <20>
RELATION TO
SET
REPROCESS TO <01>
SET
SAFETY ON
SAFETY OFF
SET
SCOREBOARD ON
SCOREBOARD OFF
SET
SEPARATOR TO <00>
SET
SKIP TO <26>
SET
SPACE ON
SPACE OFF
SET

SQL ON
SQL OFF
SET
STATUS ON
STATUS OFF
SET
STEP ON
STEP OFF
SET
TALK ON
TALK OFF
SET
TITLE ON
TITLE OFF
SET
TRAP ON
TRAP OFF
SET
TYPEAHEAD TO <01>
SET
UNIQUE ON
UNIQUE OFF
SET
VIEW TO ?
VIEW TO <04>
SET
WINDOW OF MEMO TO <06>
SHOW *
DATABASE
SHOW
MENU <28> PAD <29>
MENU <28>

SHOW
POPUP <30>
SKIP
0
<01> IN <31>
<01>
SORT
ON <24>
<02>
WHILE <03>
FOR <03>
TO <04>
START
DATABASE <32>
STOP
DATABASE <32>
STORE
<05> TO <13>
SUM
TO <13>
TO <33>
<02>
WHILE <03>
FOR <03>
<05>
SUSPEND
0
TEXT
0
TOTAL
ON <16> TO <04>
<02>

WHILE <03>
FOR <03>
FIELDS <34>
TYPE
<04> TO PRINTER <01>
<04> TO PRINTER
<04> TO FILE <04> <01>
<04> TO FILE <04>
<04> <01>
<04>
UNLOAD
DATA TO <04>
&
FROM TABLE <04>
DELIMITED WITH BLANK
DELIMITED WITH <00>
TYPE SDF
TYPE DIF
TYPE SYLK
TYPE WKS
TYPE DBASE
TYPE FW2
TYPE RPD
TYPE WK1
SDF
DIF
SYLK
WKS
DBASE
FW2
RPD
WK1

UNLOCK
0
ALL
IN <31>
UPDATE
ON <16> FROM <20> REPLACE
&
FIELD <16> WITH <05>
RANDOM
UPDATE *
<04> SET <55>
WHERE <03>
UPDATE *
<04> SET <55>
WHERE CURRENT OF <37>
USE
0
USE
? IN <20>
<04> IN <20>
?
<04>
INDEX <12>
ORDER TAG <38> OF <04>
ORDER <38> OF <04>
ALIAS <21>
EXCLUSIVE
NOUPDATE
AGAIN
WAIT
0
<14>

TO <06>

ZAP

0

APPENDIX - C

The following next reserve words are used for GSQL:

&

0

12

24

<00>

<01>

<02>

<03>

<04>

<05>

<06>

<07>

<08>

<09>

<10>

<11>

<12>

<13>

<14>

<15>

<16>

<17>

<18>

<19>

<20>

<21>

<22>
<23>
<24>
<25>
<26>
<27>
<28>
<29>
<30>
<31>
<32>
<33>
<34>
<35>
<36>
<37>
<38>
<39>
<40>
<41>
<42>
<43>
<44>
<45>
<46>
<47>
<48>
<49>
<50>
<51>
<52>
<53>

<54>

<55>

<56>

<57>

<58>

<59>

<60>

<61>

<62>

<63>

<64>

<65>

<66>

<67>

<68>

<69>

<70>

?

@

ACTIVE

ADD

ADDITIVE

AGAIN

ALIAS

ALL

ALTERNATE

AMERICAN

ANSI

APPLICATION

ARRAY

AS

AT

AUTOSAVE
BAR
BEFORE
BELL
BLANK
BLOCKSIZE
BORDER
BOTTOM
BOX
BRITISH
BY
CARRY
CASE
CATALOG
CENTURY
CHECK
CLEAR
CLOCK
COLOR
COMMAND
COMPRESS
CONFIRM
CONNECTED
CONSOLE
CURRENCY
CURRENT
CURSOR
DATA
DATABASE
DATABASES
DATAFILE
DATAFILES

DATE
DBASE
DEACTIVATE
DEBUG
DECIMAL
DECIMALS
DEFAULT
DELETED
DELIMITED
DELIMITERS
DESCENDING
DESIGN
DEVELOPMENT
DEVICE
DIF
DISPLAY
DISTINCT
DOHISTORY
DOUBLE
ECHO
EGA25
EGA43
ELSE
ENCRYPTION
ENDCASE
ENDDO
ENDIF
ENDTEXT
ENVIRONMENT
ERROR
ESCAPE
EXACT

EXCEPT
EXCLUSIVE
EXIT
EXTENDED
FIELD
FIELDS
FILE
FILES
FILL
FILTER
FIXED
FOR
FORM
FORMAT
FREEZE
FRENCH
FROM
FULLPATH
FUNCTION
FUNCTIONS
FW2
GERMAN
GET
GETS
GRANT
GROUP
HAVING
HEADING
HEIGHT
HELP
HISTORY
HOURS

IN
INCLUDE
INDEX
INDEXES
INDIAN
INSTRUCT
INTENSITY
INTERSECT
INTO
JAPAN
KEEP
KEY
LABEL
LAST
LEFT
LIKE
LINE
LOCK
LOOP
MACRO
MACROS
MARGIN
MARK
MASTER
MEMO
MEMORY
MEMOWIDTH
MENU
MENUS
MESSAGE
MINUS
MODULE

MONO
MONO43
NEAR
NOAPPEND
NOCLEAR
NODELETE
NOEDIT
NOEJECT
NOFOLLOW
NOINIT
NOMENU
NONE
NOUPDATE
NOWAIT
OBJECT
OBJECTS
ODOMETER
OF
OFF
ON
OPEN
OPTION
ORDER
OTHERWISE
OVERWRITE
PAD
PAGE
PANEL
PATH
PAUSE
PFS
PICTURE

PLAIN
POINT
POP
POPUP
POPUPS
PRECISION
PRINT
PRINTER
PRIVILEGE
PRIVILEGES
PROCEDURE
PROCEDURES
PROMPT
PUBLIC
QUERY
RANDOM
RANGE
READERROR
RECORD
RECORDS
REFRESH
RELATION
REPLACE
REPORT
REPROCESS
RIGHT
RPD
RUNTIME
SAFETY
SAMPLE
SAVE
SAY

SCOREBOARD
SCREEN
SDF
SELECT
SELECTION
SEPARATOR
SET
SINGLE
SKIP
SPACE
SQL
START
STATUS
STEP
STRUCTURE
STYLE
SUMMARY
SYLK
SYNONYM
TABLE
TAG
TALK
TEMP
TITLE
TO
TOP
TRANSACTION
TRAP
TYPE
TYPEAHEAD
UNION
UNIQUE

UPDATE
USERS
VALID
VALUES
VIEW
WHEN
WHERE
WHILE
WIDTH
WINDOW
WINDOWS
WITH
WK1
WKS
WORK

APPENDIX - D

The following are the GSQL *first* reserve words:

!
&&
*
?
??
???
@
ACCEPT
ACTIVATE
ALTER
APPEND
ASSIST
AVERAGE
BEGIN
BROWSE
CALCULATE
CALL
CANCEL
CASE
CHANGE
CLEAR
CLOSE
COMPILE
CONTINUE
CONVERT
COPY
COUNT
CREATE

DBCHECK
DBDEFINE
DEACTIVATE
DEBUG
DECLARE
DEFINE
DELETE
DIR
DIRECTORY
DISPLAY
DO
DROP
EDIT
EJECT
ELSE
END
ENDCASE
ENDDO
ENDIF
ENDPRINTJOB
ENDSCAN
ENDTEXT
ERASE
EXIT
EXPORT
FETCH
FIND
FUNCTION
GO
GOTO
GRANT
HELP

IF
IMPORT
INCLUDE
INDEX
INPUT
INSERT
JOIN
LABEL
LIST
LOAD
LOCATE
LOGOUT
LOOP
MODIFY
MOVE
NOTE
ON
OPEN
OTHERWISE
PACK
PARAMETERS
PLAY
PRINTJOB
PRIVATE
PROCEDURE
PROTECT
PUBLIC
QUIT
READ
RECALL
REINDEX
RELEASE

RENAME
REPLACE
REPORT
RESET
RESTORE
RESUME
RETRY
RETURN
REVOKE
ROLLBACK
RUN
RUNSTATS
SAVE
SCAN
SEEK
SELECT
SET
SHOW
SKIP
SORT
START
STOP
STORE
SUM
SUSPEND
TEXT
TOTAL
TYPE
UNLOAD
UNLOCK
UPDATE
USE

WAIT

ZAP

APPENDIX - E

The following GURU system programs are loading an index, preparing relations to be recorded with schema, global searching with multiple indexes, and searching with an efficient search method on the loaded indexes.

```

/*****
index_f_load(x,st1,fin,cac)
char x[];
int st1,fin;
unsigned char cac;
{
  unsigned char fce,seq;
  unsigned int i,i1,j,k,l,m,st,tif,rec_max,rec_size,max_len;
  if((tif=open_i_files(x,st1,fin,cac,&fce,&max_len))==0) return(0);
  search_f_s_point_node(&(*curr_p)->fsp,curr_fsp,AFAC);
  (*curr_fsp)->index_stat=1;
  mal=(unsigned long) (2L*(*curr_fsp)->rec_max);
  if((mal!=0L)&&((r_n_p=(unsigned int far *) farmalloc(mal))==NULL)
  mess_no_mem(31);
  mal=(unsigned long) (*curr_fsp)->rec_max;
  if((mal!=0L)&&((r_n_s=(unsigned int far *) farmalloc(mal))==NULL)
  mess_no_mem(32);
  mal=(unsigned long) ((*curr_fsp)->rec_max*max_len);
  if((mal!=0L)&&((mem=(char far *) farmalloc(mal))==NULL)) mess_no_
  index_nos_read(r_n_p,&rec_max,&rec_size,FACI[fce][0]);
  *r_n_s=0;
  *(r_n_s+1)=(unsigned int) (*curr_fsp)->rec_max-1;
  *(r_n_s+2)=0xFFFF;
  st=3;

```



```

j=0;
for(i=0; i < tif; ++i)
{
    l=st;
index_names_read(mem, &rec_max, &rec_size, FACI[fce][i], &seq);
if(i==0)
(*curr_fsp)->index_stat=seq+1;
    for(m=0; *(r_n_s+m) != 0xFFFF; m+=2)
    {
        if(i!=0)
        {
if(seq==0)
            {
                for(il=*(r_n_s+m); il < *(r_n_s+m+1); ++il)
                    for(j=i1+1; j < (*(r_n_s+m+1)+1); ++j)
                        if(strncmp(mem+(rec_size*(*(r_n_p+i1))), \
mem+(rec_size*(*(r_n_p+j))), \
rec_size) > 0)
                            swap(r_n_p+i1, r_n_p+j);
            }
        else
        {
                for(il=*(r_n_s+m); il < *(r_n_s+m+1); ++il)
                    for(j=i1+1; j < (*(r_n_s+m+1)+1); ++j)
                        if(strncmp(mem+(rec_size*(*(r_n_p+i1))), \
mem+(rec_size*(*(r_n_p+j))), \
rec_size) < 0)
                            swap(r_n_p+i1, r_n_p+j);
            }
        }
    }
if(tif!=1)
{

```

```

for(i1=*(r_n_s+m); i1 < (*(r_n_s+m+1)); ++i1)
{
    k=i1;
    while((i1 < (*(r_n_s+m+1)))&&\
        (same_str(mem+(rec_size*(r_n_p+i1)),\
            mem+(rec_size*(r_n_p+i1+1))),\
            rec_size,0)) ++i1;
    if(k!=i1)
    {
        *(r_n_s+1)=k;
        *(r_n_s+1+1)=i1;
        l+=2;
    }
}
}
}
*(r_n_s+1)=0xFFFF;
copy_blk(r_n_s,st,l);
st=l-st+1;
}
if(r_n_s!=NULL)
farfree(r_n_s);
r_n_s=NULL;
if(mem!=NULL)
farfree(mem);
mem=NULL;
(*curr_fsp)->index_arr= r_n_p;
IFL=0;
IND_STAT=1;
return(1);
}
/*****/

```

```

relation_index_build(a,f_n,fl)
char a[],f_n[],fl;
{
unsigned int i,j,k,l,m,n;
unsigned char ac,temp_ac;
int n1,x,y;
long jj,ii;
char sv[512];
char **dv;
unsigned long ad,rm1,rm2;
unsigned int *r_n_pl,*r_n_ri;
char far *mem1;
r_n_pl=r_n_ri=NULL;
mem1=NULL;
dv=(char far **) (unsigned long) &ad;
*dv=NULL;
REL_FLAG=1;
if(exist_fn_alias(f_n,"\0",&k)==0)
{
printf("File %s is not loaded in memory\n",f_n);
REL_FLAG=0;
return(0);
}
ac=FAC_C[k];
if(search_f_s_point_node(&(*curr_p)->fsp,curr_fsp,AFAC)==0)
{
printf("File not loaded in area %d\n",AFAC);
REL_FLAG=0;
return(0);
}
IFL=0;
rm1=(*curr_fsp)->rec_max;

```

```

strcpy(sv, a);
expression_proc(dv, sv);
if(fl==0)
l=strlen(*dv);
else
l=5;
mal=(unsigned long) l*(*curr_fsp)->rec_max;
if((mal!=0L)&&(mem=(char far *) farmalloc(mal))==NULL)
{
mess_no_mem(34);
REL_FLAG=0;
return(0);
}
if((*curr_fsp)->index_stat!=0)
{
r_n_p=(*curr_fsp)->index_arr;
x=(*curr_fsp)->index_stat;
}
else
{
mal=(unsigned long) (2L*(*curr_fsp)->rec_max);
if((mal!=0L)&&(r_n_p=(unsigned int far *) farmalloc(mal))==0)
{
mess_no_mem(35);
REL_FLAG=0;
return(0);
}
for(i=0; i < (*curr_fsp)->rec_max; ++i) *(r_n_p+i)=i;
x=1;
}
for(i=0; i < (*curr_fsp)->rec_max; ++i)
{

```

```

    IFL=i;
    strcpy(sv, a);
    expression_proc(dv, sv);
    if(fl==0)
        strncpy(mem+l*i, *dv, l);
    else
        sprintf(*dv, "%05d", i);
}
temp_ac=AFAC;
AFAC=ac;
if(search_f_s_point_node(&(*curr_p)->fsp, curr_fsp, ac)==0)
{
    printf("File not loaded in area %d\n", ac);
    REL_FLAG=0;
    return(0);
}
if((*curr_fsp)->relation_stat==1)
{
    if((*curr_fsp)->relation_arr!=NULL)
        farfree((*curr_fsp)->relation_arr);
    (*curr_fsp)->relation_arr=NULL;
    (*curr_fsp)->relation_stat=0;
    farfree((*curr_fsp)->relation_val);
    if((*curr_fsp)->relation_val!=NULL)
        (*curr_fsp)->relation_val=NULL;
    (*curr_fsp)->relation_bac=0;
}
IFL=0;
rm2=(*curr_fsp)->rec_max;
strcpy(sv, a);
expression_proc(dv, sv);
m=strlen(*dv);

```

```

mal=(unsigned long) (m*(*curr_fsp)->rec_max);
if((mal!=0L)&&((mem1=(char far *) farmalloc(mal))==NULL))
{
    mess_no_mem(36);
    REL_FLAG=0;
    return(0);
}
if((*curr_fsp)->index_stat!=0)
{
    r_n_pl=(*curr_fsp)->index_arr;
    y=(*curr_fsp)->index_stat;
}
else
{
    mal=(unsigned long) (2L*(*curr_fsp)->rec_max);
    if((mal!=0L)&&((r_n_pl=(unsigned int far *) farmalloc(mal))==0))
    {
        mess_no_mem(37);
        REL_FLAG=0;
        return(0);
    }
    for(i=0; i < (*curr_fsp)->rec_max; ++i) *(r_n_pl+i)=i;
    y=1;
}
for(i=0; i < (*curr_fsp)->rec_max; ++i)
{
    IFL=i;
    strcpy(sv,a);
    expression_proc(dv,sv);
    strncpy(mem1+m*i,*dv,m);
}
if((*curr_fsp)->relation_stat==1)

```

```

r_n_ri=(*curr_fsp)->relation_arr;
else
{
mal=(unsigned long) (2L*rm1);
if((mal!=0L)&&((r_n_ri=(unsigned int far *) farmalloc(mal))==0))
{
mess_no_mem(38);
REL_FLAG=0;
return(0);
}
}
for(i=0; i < (unsigned int) rm1; ++i)
*(r_n_ri+i)=0xFFFF;
switch(x)
{
case 1:
if(y==1)
{
for(i=0,j=0; i < (unsigned int) rm1; ++i)
for(; j < (unsigned int) rm2 ; ++j)
{
if((n1=strncmp(mem+(l*i),mem1+(m*j),l))<=0)
{
if(n1==0)
{
*(r_n_ri+i)= *(r_n_p1+j);
++j;
}
break;
}
}
}
}
}
}

```

```

else
{
  for(i=0,jj=(long) (rm2-1); i < (unsigned int) rm1; ++i)
  for(; jj >= 0 ; --jj)
  {
    if((n1=strncmp(mem+(l*i),mem1+(m*jj),l))<=0)
    {
      if(n1==0)
      {
        *(r_n_ri+i) = *(r_n_p1+jj);
        --jj;
      }
      break;
    }
  }
  break;
}

case 2:
  if(y==1)
  {
    for(ii=(long) rm1-1,j=0; ii >=0; --ii)
    for(; j < (unsigned int) rm2 ; ++j)
    {
      if((n1=strncmp(mem+(l*ii),mem1+(m*j),l))<=0)
      {
        if(n1==0)
        {
          *(r_n_ri+ii) = *(r_n_p1+j);
          ++j;
        }
        break;
      }
    }
  }
}

```



```

    }
}
else
{
    for(ii=(long) (rm1-1),jj=(long) (rm2-1); ii >=0; --ii)
    for(; jj >= 0 ; --jj)
    {
        if((n1=strncmp(mem+(l*ii),mem1+(m*jj),l))<=0)
        {
            if(n1==0)
            {
                *(r_n_ri+ii)= *(r_n_pl+jj);
                --jj;
            }
            break;
        }
    }
}
break;
}

AFAC=temp_ac;
(*curr_fsp)->relation_stat=1;
(*curr_fsp)->relation_arr=r_n_ri;
(*curr_fsp)->relation_bac=AFAC;
if((*curr_fsp)->relation_val!=NULL)
farfree((*curr_fsp)->relation_val);
i=strlen(a);
if(((curr_fsp)->relation_val=(char *) farmalloc((unsigned long) (i+1)
mess_no_mem(50);
sprintf((*curr_fsp)->relation_val,"%s",a);
    if(mem!=NULL)
        farfree(mem);

```

```

    mem=NULL;
    if (mem1!=NULL)
        farfree(mem1);
    mem1=NULL;
    if ((*curr_fsp)->index_stat==0)
        if (r_n_p1!=NULL)
            farfree(r_n_p1);
        r_n_p1=NULL;
        search_f_s_point_node(&(*curr_p)->fsp,curr_fsp,AFAC);
        if ((*curr_fsp)->index_stat==0)
            if (r_n_p!=NULL)
                farfree(r_n_p);
            r_n_p=NULL;
            REL_FLAG=0;
            return(1);
}
/*****
find_index_value(x)
char x[];
{
    char a[3];
    unsigned char fce,tfn,seq;
    unsigned int i,m,n,rec_max,rec_size,max_len,p;
        int k,j;
        p=strlen(x);
        for(i=0; i < 26; ++i) if(FAC_C[i]==AFAC) break;
        fce=(unsigned char) i;
        if(search_f_s_point_node(&(*curr_p)->fsp,curr_fsp,AFAC)==0)
            return(0);
        if ((*curr_fsp)->index_stat==0)
            {
                printf("No index file(s) loaded with the current area\n");

```

```

return(0);
}
max_len=0;
for(i=0; (i < 8)&&(FACI[fce][i]!=0); ++i)
{
lseek((int) FACI[fce][0],6L,0);
mal=0L;
file_rd((int) FACI[fce][0],&mal,&k,2);
lseek((int) FACI[fce][0],(unsigned long) (8+k),0);
file_rd((int) FACI[fce][0],&mal,&j,2);
lseek((int) FACI[fce][0],(unsigned long) (12+j+k),0);
_read((int) FACI[fce][i],a,2);
strtoin(&n,a);
if(n > max_len) max_len=n;
}
tfn=i;
r_n_p=(*curr_fsp)->index_arr;
mal=(unsigned long) ((*curr_fsp)->rec_max*max_len);
if((mal!=0L)&&((mem=(char far *) farmalloc(mal))==0))
mess_no_mem(39);
for(i=0; (i < 8)&&(FACI[fce][i]!=0); ++i)
{
index_names_read(mem,&rec_max,&rec_size,FACI[fce][i],&seq);
if(p > rec_size) p=rec_size;
for(m=IFL; m < rec_max; ++m)
{
for(n=0; n < p; ++n)
printf("%c",*(mem+n+(rec_size*(*(r_n_p+m)))));
printf("\n");
getch();
if(((tfn==1)&&((k=strncmp(mem+(rec_size*(*(r_n_p+m)))\
,x,p)) >= 0))\

```

```

|| ((k=strncmp(mem+(rec_size*(*(r_n_p+m))),x,p))==0))
{
    if(k==0)
    {
        if(mem!=NULL)
            farfree(mem);
        mem=NULL;
        IFL=m+1;
        (*curr_fsp)->rec_no=*(r_n_p+m);
        return(1);
    }
    else
        if(tfn==1) break;
}
}
IFL=0;
}
if(mem!=NULL)
    farfree(mem);
mem=NULL;
return(0);
}
/*****
find_index_pvalue(x)
char x[];
{
    char a[3];
    unsigned char fce,flag,seq;
    unsigned int i,m,n,rec_max,rec_size,p,k1,k2,fv,iter;
        int k,j;
        n=IFL;
        p=strlen(x);

```

```

for(i=0; i < 26; ++i) if(FAC_C[i]==AFAC) break;
fce=(unsigned char) i;
if(search_f_s_point_node(&(*curr_p)->fsp,curr_fsp,AFAC)==0)
return(0);
if((*curr_fsp)->index_stat==0)
{
printf("No index file(s) loaded with with the current area\n");
return(0);
}
fv=(unsigned int) (log((*curr_fsp)->rec_max-IFL)+2);
iter=0;
lseek((int) FACI[fce][0],6L,0);
mal=0L;
file_rd((int) FACI[fce][0],&mal,&k,2);
lseek((int) FACI[fce][0],(unsigned long) (8+k),0);
file_rd((int) FACI[fce][0],&mal,&j,2);
lseek((int) FACI[fce][0],(unsigned long) (12+j+k),0);
_read((int) FACI[fce][0],a,2);
strtoin(&rec_size,a);
r_n_p=(*curr_fsp)->index_arr;
mal=(unsigned long) ((*curr_fsp)->rec_max*((unsigned long) rec_si
if((mal!=0L)&&((mem=(char far *) farmalloc(mal))==0))
mess_no_mem(40);
index_names_read(mem,&rec_max,&rec_size,FACI[fce][0],&seq);
if(p > rec_size) p=rec_size;
flag=0;
k1=rec_max-1;
if(seq==0)
{
k1=IFL;
j=k2=rec_max-1;
if((strncmp(mem+(rec_size*(*(r_n_p))),x,p)<=0)&&

```

```

(strncmp(mem+(rec_size*(*(r_n_p+(rec_max-1)))) ,x,p)>=0)
{
while((k=strcmp(mem+(rec_size*(*(r_n_p+j))),x,p))!=0)
{
if(k > 0) k2=j;
else
if(k < 0) k1=j;
if(j==(k1+k2)/2)
++j;
else
j=(k1+k2)/2;
++iter;
if((k2==k1)|| (iter>fv)) { flag=1; break;}
}
}
}
else
{
j=k1=rec_max-1;
k2=IFL;
if((strcmp(mem+(rec_size*(*(r_n_p))),x,p)>=0)&&\
(strcmp(mem+(rec_size*(*(r_n_p+(rec_max-1)))) ,x,p)<=0))
{
while((k=strcmp(mem+(rec_size*(*(r_n_p+j))),x,p))!=0)
{
if(k < 0) k1=j;
else
k2=j;
++iter;
if(j==(k1+k2)/2)
++j;
else

```

```

    j=(k1+k2)/2;
    if((k2==k1)|| (iter>fv)) { flag=1; break;}
}
}
}
if((flag==1)|| (iter==0))
{
if(mem!=NULL)
farfree(mem);
mem=NULL;
return(0);
}
if(seq==0)
{
for(i=j-1; i >= n; --i)
if(strncmp(mem+(rec_size*(*(r_n_p+i))),\
mem+(rec_size*(*(r_n_p+j))),p)!=0) break;
k1=i+1;
IFL=k1;
}
else
{
for(i=j+1; i <= n; --i)
if(strncmp(mem+(rec_size*(*(r_n_p+i))),\
mem+(rec_size*(*(r_n_p+j))),p)!=0) break;
k1=i-1;
IFL=k1;
}
if(mem!=NULL)
farfree(mem);
mem=NULL;
(*curr_fsp)->rec_no=k1+1;

```

```
return(1);
```

```
}
```


APPENDIX - F

Following are some library functions working with GURU:

1	&
2	ABS ()
3	ACCESS ()
4	ACOS ()
5	ALIAS ()
6	ASC ()
7	ASIN ()
8	AT ()
9	ATAN ()
10	ATN2 ()
11	BAR ()
12	BOF ()
13	CALL ()
14	CDOW ()
15	CEILING ()
16	CHANGE ()
17	CHR ()
18	CMONTH ()
19	COL ()
20	COMPLETED ()
21	COS ()
22	CTOD ()
23	DATE ()
24	DAY ()
25	DBF ()
26	DELETED ()
27	DIFFERENCE ()

28 DISKSPACE()
29 DMY()
30 DOW()
31 DTOC()
32 DTOR()
33 DTOS()
34 EOF()
35 ERROR()
36 EXP()
37 FIELD()
38 FILE()
39 FIXED()
40 FKLABEL()
41 FKMAX()
42 FLOAT()
43 FLOCK()
44 FLOOR()
45 FOUND()
46 FV()
47 GETENV()
48 IIF()
49 INKEY()
50 INT()
51 ISALPHA()
52 ISCOLOR()
53 ISLOWER()
54 ISMARKED()
55 ISUPPER()
56 KEY()
57 LASTKEY()
58 LEFT()
59 LEN()

60 LIKE()
61 LINENO()
62 LKSYS()
63 LOCK()
64 LOG()
65 LOG10()
66 LOOKUP()
67 LOWER()
68 LTRIM()
69 LUPDATE()
70 MAX()
71 MDX()
72 MDY()
73 MEMLINES()
74 MEMORY()
75 MENU()
76 MESSAGE()
77 MIN()
78 MLINE()
79 MOD()
80 MONTH()
81 NDX()
82 NETWORK()
83 NPV()
84 ORDER()
85 OS()
86 PAD()
87 PAYMENT()
88 PCOL()
89 PI()
90 POPUP()
91 PRINTSTATUS()

92 PROMPT()
93 PROW()
94 PV()
95 RAND()
96 READKEY()
97 RECCOUNT()
98 RECNO()
99 RECSIZE()
100 REPLICATE()
101 RIGHT()
102 RLOCK()
103 ROLLBACK()
104 ROUND()
105 ROW()
106 RTOD()
107 RTRIM()
108 SEEK()
109 SELECT()
110 SET()
111 SIGN()
112 SIN()
113 SOUNDEX()
114 SPACE()
115 SQRT()
116 STR()
117 STUFF()
118 SUBSTR()
119 TAG()
120 TIME()
121 TRANSFORM()
122 TRIM()
123 TYPE()

124 UPPER()
125 USER()
126 VAL()
127 VARREAD()
128 VERSION
129 YEAR()

BIBLIOGRAPHY

1. M. Abdelguerfi, and Arun K. Sood, "Computational Complexity of Sorting and Join Relations with Duplicates," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 496-503, Dec. 1991.
2. S. Abiteboul and N. Bidoit, "An Algebra for Non-normalized Relations," in Proc. 3rd ACM Int. Symp. Principles Database Syst., 1984.
3. M. E. Adiba and B. G. Lindsay, "Database Snapshots," Proc. Sixth Int. Conf. Very Large Databases, pp. 86-91, 1980.
4. G. Agha and C. Hewitt, "Actors: A conceptual foundation for concurrent object-oriented programming," in Research Directions in Object-Oriented Programming, B. Shriver and P. Wegner, Eds. Cambridge MA: The MIT Press Series in Computer Systems, pp. 49-74, 1987.
5. M. Agosti, "Database design: A classified and annotated bibliography," British Computer Soc. Monographs in Informatics., Cambridge University Press, 1986.
6. Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammed A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming-Chien Shan, "The Pegasus Heterogeneous Multidatabase System," IEEE Computer, pp. 19-27, Dec. 1991.
7. M. Ahamad, P. Dasgupta, and R. J. LeBlanc, "Fault-tolerant automatic computations in object-based distributed system," Distributed Computing, vol. 4, no. 2, pp. 69-80, May 1990.
8. M. Ahamad and L. Lin, "Using check-points to localize the effects of faults in distributed systems," in Proc. 8th Symp. Reliable Distributed Syst., Seattlem, WA, pp. 2-11, Oct. 1989.
9. A. Ahmed and A. Rafii, "Relational Schema and Mapping and Query Translation in Pagasus," Proc. Workshop in Multidatabases and Semantic Interoperability, pp. 22-25, 1990.
10. G. T. Almes et al., "The Eden system: A technical review,"

IEEE Trans. Software Eng., Piscataway, N. J., vol. SE-11, no. 1, pp. 43-58, Jan. 1985.

11. G. T. Almes, C. L. Holman, "Edmas: An Object-Oriented, Locally Distributed Mail System," IEEE Trans. on Soft. Eng., vol. SE-13, no. 9, pp. 1001-1009, Sep. 1987.

12. T. Andrews and C. Harris, "Combining Language and Database Advances in Object-Oriented Development Environment," Proc. Object-Oriented Programming Systems, Languages and Applications, Addison-Wesley, Reading Mass., pp. 430-440, 1987; and also SIGPlan Notices, special issue, ACM. vol. 22, no. 12, Dec. 1987.

13. M. Atkinson et al., "The Object-Oriented System Manifesto," Proc. First International Conf. Deductive and Object-Oriented Databases, Elsevier Science Publishers B. V., Amsterdam, pp. 40-57, 1989.

14. M. Atkinson and P. Buneman, "Types and persistence in database programming languages," ACM Comput. Surveys, vol. 19, no. 2, June 1987.

15. T. M. Atwood, "An object-oriented DBMS for design support applications," in Proc. IEEE Compint, pp. 299-307, 1985.

16. F. Bancilhon et al., "The design and implementation of O₂, An object-oriented database system," in Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science 334, K. R. Dittrich, Ed. Berlin Germany: Springer, pp. 1-22, 1988.

17. J. Banerjee et al., "Data model issues for object-oriented applications," ACM Trans. Office Infom. Syst., vol. 5, pp. 3-26, 1987.

18. C. Batini and M. Lenzerini, "A methodology for data schema integration in the entity relationship model," IEEE Trans. Software Eng., vol SE-10, pp. 650-664, Nov. 1984.

19. C. Batini, M. Lenzerini, and S. B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Surveys, vol. 18, no. 4, pp. 323-364, Dec. 1986.

20. D. S. Batory and A. P. Buchmann, "Molecular objects, abstract data types, and data models: A framework," in Proc. 10th Int. VLDB Conf., pp. 172-184, 1984.
21. R. Bayer, H. Heller, and A. Reiser, "Parallelism and recovery in database systems," ACM Trans., Database Syst., vol. 5, no. 2, pp. 139-156, June 1980.
22. P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, vol. 13, no. 2, pp. 185-221, June 1981.
23. P. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems. Reading, MA: Addison-Wesley, 1987.
24. E. Bertino et al., "Integration of Heterogeneous Applications Through an Object-Oriented Interface," Information Systems, Pergamon Press, vol. 14, no. 5, pp. 407-420, 1989.
25. E. Bertino, M. Negri, G. Pelagatti and L. Sbatella, "Object-Oriented Query Languages: The Notion and the Issues," IEEE Trans. on Know. and Data Eng., vol. 4, no. 3, pp. 223-237, June 1992.
26. Elisa Bertino and Lorenzo Martino, "Object-Oriented Database Management System: Concepts and Issues," IEEE Computer, pp. 33-47, Apr. 1991.
27. A. Bjornerstedt and C. Hulten, "Version Control in an Object-Oriented Architecture," in Object-Oriented Concepts. Databases, and Applications, W. Kim and F. Lochovsky, eds., Addison-Wesley, Reading, Mass., pp. 451-485, 1989.
28. Black et al., "Distribution and Abstraction Types in Emerald," IEEE Software Eng., vol. SE-13, no. 1, pp. 65-76, 1987.
29. J. A. Blakeley, N. Coburn, and P. A. Larson, "Updating derived relations: Detecting irrelevant and autonomous computable updates," ACM Trans. Database Syst., vol. 14, no. 3, pp. 369-400, Sept. 1989.

30. H. Boral, "Parallelism in Bubba," in Proc. Int. Symp., Databases in Parallel and Distributed System, Austin in Dec. 1988.
31. R. Brachman and J. G. Schmolze, "An Overview of KI- One Knowledge Representation System," Cognitive Science, vol. 9, no. 2, pp. 171-216, Apr.-June, 1985.
32. D. Briatico, A. Ciuffolett, and L. Simoncini, "A distributed domino effect free recovery algorithm," in Proc. IEEE Symp. Reliability in Distributed Software and Database Syst., Silver Spring, MD, pp. 207-215, Oct. 1984.
33. Y. Brietbart, P. L. Olson and G. L. Thompson, "Database Integration in a Heterogeneous Distributed Database System," Proc. Data Eng. Conf. IEEE CS Press, Los Alamitos, Calif. pp. 310-310, 1986.
34. Y. Breitbart, ed., Proc. Workshop Multidatabases and Semantic Interperability, National Science Foundation with the cooperation of University of Kentucky and Amoco, Nov. 1990.
35. Y. Breitbart, A. Silberschatz, and G. Thompson, "Reliable Transaction Management in Multidatabase Systems," Proc. SIGMOD Int'l Conf. Management of Data, ACM New York, pp. 215-224, 1990.
36. R. Breitle et al., "The Gemstone Data Management System," in Object-Oriented Concepts, Databases, and Applications, K. Kim and F. Lochovsky, eds., Addison-Wesley, Reading Mass., pp. 283-308, 1989.
37. S. Cammarata, "Deferring updates in a relational database system," in Proc. Seventh Int. Conf. Very large databases, pp. 286-292, 1981.
38. M. J. Carey, D. J. Dewitt, and S. L. Vandenberg, "A data model and query language for EXODUS," in Proc. ACM SIGMOD Conf., pp. 413-423, 1988.
39. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analysis models for rollback and recovery strategies in database

- systems," IEEE Trans., Software Eng., vol. SE-1, no. 1, pp. 100-110, Mar. 1975.
40. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Trans. Comput. Syst., vol. 3, no. 1, pp. 63-75, Feb. 1985.
41. M. Chandy and J. Misra, Parallel Program Design: A Foundation Reading, MA: Addison-Wesley, 1988.
42. C. Chang, "Dataplex: An access to Heterogeneous Distributed Databases," ACM, vol. 33, no. 1, pp. 70-80, Jan. 1990.
43. T. H. Chang and A. Sciore, "A Universal Relation Data Model with Semantic Abstraction," IEEE Trans. on Know. and Data Eng., vol. 4, no. 1, pp. 23-33, Feb. 1992.
44. R. Chen and P. Dasgupta, "Linking consistency with object/thread semantics: An approach of robust computations," Proc. Ninth Int'l Conf. Distributed Computing Systems, IEEE CS Press, Los Alamitos, Calif., order no. 1953, pp 121-128, 1989.
45. D. R. Cheriton, "The V distributed system," Comm. ACM, vol. 31, no. 3, pp. 314-333, Mar. 1988.
46. D. R. Cheriton, "VMTP: A transport protocol for the next generation of communication systems," Proc. SIGcomm, pp. 406-415, 1986.
47. Pai-Cheng Chu, "Estimating Block Selectivities For Physical Database Design," IEEE Trans. on Know. and Data Eng., vol. 4, no. 1, pp. 89-98, Feb. 1992.
48. R. S. Chin and S. T. Chanson, "Distributed Object-Based Programming Systems," ACM Comp. Sur., vol. 23, no. 1, pp. 91-124, Mar. 1991.
49. E. F. Codd, "Relational Database: A Practical Foundation for Productivity," Communications of the ACM vol. 25, no. 2, pp. 109-117, Feb. 1982.
50. E. F. Codd, "Extending the Database Relational Model to

Capture More Meaning," ACM Transactions on Database Systems vol. 4, no. 4, pp. 397-434, Dec. 1979; (also ACM-SIGMOD Conference Proceedings, 1979 May).

51. E. F. Codd, "Normalized Database Structure: A Brief Tutorial," Proceedings of 1971 ACM-SIGFIDET Workshop "Data Description, Access, and Control," edited by E.F. Codd and A.L. Dean, New York: Association of Computing Machinery, pp. 1-17, 1971.

52. E. F. Codd, "A relational Model of Data for Large Shared Data Banks," Communications of the ACM vol. 13, no. 6, pp. 377-387, Jun. 1970.

53. D. E. Comer, and L. L. Peterson, "Understanding Naming in Distributed Systems," Distributed Computing, pp. 51-60, May 1989.

54. G. Copeland and D. Maier, "Making Smalltalk a database system," in Proc. ACM Trans. Database Syst., vol. 2., no. 3., Sept. 1977.

55. P. Dadam et al., "A DBMS prototype to support extendedNF-2-relations: An integrated view on flat tables and hierarchies," in Proc. ACM SIGMOD Conf., pp. 356-367, 1986.

56. P. Dadam and g. Schlageter, "Recovery in distributed databases based on nonsynchronized local checkpoints," in Information Processing. Amsterdam, The Netherlands: North-Holland, pp. 457-462, 1980.

57. Scott Danforth and Patrick Valduriez, "A FAD for Data Intensive Applications," IEEE Trans. on Knowledge and Data Engineering, Vol. 4. No. 1, pp. 34-51, Feb. 1992.

58. Scott Danforth and Chris Tomlinson, "Type Theories and Object-Oriented Programming," ACM Comp. Sur., vol. 20, no. 1, pp. 29-72, Mar. 1988.

59. P. Dasgupta et al., "The design and implementation of the clouds distributed operating system," Usenix Computing Systems J., vol. 3, no. 1, pp. 11-46, Winter 1990.

60. P. Dasgupta, R. J. LeBlanc, J. M. Ahamad, and U.

Ramachandran, "The Clouds Distributed Operating System," IEEE Computer, pp. 34-44, Nov. 1991.

61. S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Network," ACM Computing Surveys, vol. 17, no. 3, pp. 341-370, Sept. 1985.

62. U. Dayal and P. A. Bernstein, "On the updatability of relational views," in Proc. Fourth Int. Conf. Very large databases, pp. 368-377, 1978.

63. U. Dayal and J. M. Smith, "PROBE: A knowledge-oriented database management system," in On Knowledge Base Management Systems, M. L. Brodie and J. Mylopoulos, Eds. Berlin Germany: Springer, pp. 227-258, 19986.

64. U. Dayal and H. Y. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," IEEE Trans. Software Eng., vol. SE-10, no. 6, pp. 628-645, Nov. 1984.

65. L. DeMichiel, "Performing Operations Over Mismatched Domains," Proc. Fifth IEEE Data Eng. Conf. CS Press, Los Alamitos, Calif., order no. 1915, pp. 36-45, 1989.

66. F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," IEEE Trans. Software Eng., vol. SE-2, pp. 80-86, 1976.

67. Deux et al., "The Story of O₂," IEEE Trans. Knowledge and Data Eng., vol. 2, no. 1, pp. 91-108, 1990.

68. E. W. Dijkstra, "The distributed snapshot of K. M. Chandy and L. Lamport," in Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series F: Computer System Sciences, vol. 14, M. Broy, Ed. Berlin, Germany: Springer-Verlag, pp. 513-518, 1985.

69. Klaus R. Dittrich and Raymond A. Lorie, "Version Support for Engineering Database Systems," IEEE Trans. on Soft. Eng., vol. 14, no. 4, pp. 429-437, Apr. 1988.

70. Vishweshwar V. Dixit and Don I. Moldovon, "Minimal State Space Search in Parallel Production Systems," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 435-443, Dec. 1991.
71. Gordon. C. Everest, "Basic Data Structure Models Explained with a Common Example," Proceedings fifth Texas Conference on Computing Systems, Ustin, pp. 39-46, Oct. 1976.
72. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in database systems," Comm. ACM, vol. 19, no. 11, pp. 624-633, Nov. 1976.
73. M. J. Fischer, N. D. Griffeth, and N. A. Lynch, "Global states of a distributed system," IEEE Trans. Software Eng., vol. SE-8, no. 3, pp. 198-202, 1982.
74. D. H. Fishman et al., "Iris: an Object-Oriented Database Management System," ACM Trans. Office Information Systems, vol. 5, no. 1, pp. 48-69, 1987.
75. A. Furtado and L. Kerschberg, "An algebra of quotient relations," in Proc. ACM Int. SIGMOD Conf. 1977.
76. K. S. Gadia, "A homogeneous relational model and query languages for temporal databases," ACM Trans. Database Systems vol. 13, no. 4, pp. 418-448, Dec. 1988.
77. H. Garcia-Molina and B. Kogan, "Node Autonomy in Distributed Systems," Proc. Int'l Symp. Databases in Parallel and Distributed Systems, CS Press, Los Alamitos, Calif., Order no. 893, pp. 158-166, 1988.
78. N. Gehani, "Databases and unit of measure," IEEE Trans. Software Engg., vol. SE-8, no. 6, pp. 605-611, Nov. 1982.
79. A. Goldberg and D. Robson, Smalltalk-80, The language and its implementation. Reading, MA: Addison-Wesley, 1983.
80. W. Gotthard, P. C. Lockemann, and A. Neufeld, "System-guided view integration for object-oriented databases," IEEE Trans. on Know. and Data Eng., vol. 4, no. 1, pp. 68-82, Feb. 1992.

81. J. N. Gray, "Notes on database operating system," in *Operating Systems: An Advance Course*. Berlin Germany: pp. 393-481, Springer-Verlag, 1979.
82. Nabil I. Hachem and Bruce Berra, "New Order Preserving Access Methods for Very Large Files Derived from Linear Hashing," *IEEE Trans. on Know. and Data Eng.*, vol. 4, no. 1, pp. 68-82, Feb. 1992.
83. B. Halipern and V. Nguyen, "A model for object-based inheritance," in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. Cambridge MA: The MIT Press Series in Computer Systems, pp. 147-164, 1987.
84. M. Hammer and D. Macleod, "Database description with SDM: A semantic database model," *ACM Trans. Database Systems*, vol. 6, no. 3, Mar. 1981.
85. S. Hayne and S. Ram, "Multi-User View Integration System (MUVIS): An Expert System for View Integration" *Proc. Sixth IEEE Data Eng. Conf.*, CS Press, Los Alamitos, Calif., order no. 2025, pp. 402-409, Feb. 1990.
86. S. Heiler and S. Zdonik, "Views data abstractions, and inheritance in the FUGUE data model," in *Advance in Object-Oriented Database Systems*, K. R. Dittrich, Ed., *Lecture Notes in Computer Science* 334, pp. 225-241, Springer 1987.
87. R. Hull and C. Yap, "The format model: A theory of database organization," *J. ACM*, vol. 31, no. 3, July 1984.
88. R. Hull and R. King, "Semantic Database Modeling: Survey Applications and Research Issues," *ACM Computing Surveys*, vol. 19, no. 3, pp. 201-260, 1987.
89. T. Imielinski and W. Lipski, "Incomplete Information in Relational Databases," *J. ACM*, vol. 31, no. 4, pp. 761-791, Oct. 1984.
90. Christian S. Jensen, Leo Mark, and Nick Rousspoulos, "Incremental Implementation Model for Relational Databases with

Transaction Time," IEEE Trans. on Knowledge and Data Engineering, Vol. 3. No. 4. Dec. 1991.

91. S. Karl and P. C. Lockemann "Design of engineering databases: A case for more varied semantic modeling concepts," Inform. Syst. vol. 13, pp. 335-357, 1988.

92. A. Kemper, P. C. Lockemann, and M. Wallrath, "An object-oriented database system for engineering applications," in Proc. ACM SIGMOD Conf., pp. 299-311, 1987.

93. W. Kent, "Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language," Proc. VLDB Conf., Morgan Kaufmann, San Mateo, Calif., pp. 147-160, 1991.

94. S. Khoshafian and G. Copeland, "Object identity," in Proc. 1st Int. Workshop Object-Oriented programming language Syst., Languages and Appl., Portland, 1986.

95. W. Kim et al., "A Transaction Mechanism for Engineering Design Databases," Proc. International Conference, Very Large Databases, Morgan Kauffmann, Los Altos, Calif., pp. 255-362, 1984.

96. W. Kim et al., "Architecture of Orion Next-Generation Database System," IEEE Trans. Knowledge and Data Eng., vol. 2, no. 1, pp. 109-124, 1990.

97. W. Kim, H. T. Chou, and J. Banerjee, "Operations and implementation of complex objects," in Proc. 3rd Int. Conf. Data Eng., pp. 626-633, 1987.

98. Won Kim and Jungyun Seo, "Classifying Schematic and Data Heterogeneity in Multidatabase Systems," IEEE Computer, pp. 12-18, Dec. 1991.

99. Ramesh Krishnamurty, "An Approximation Algorithm for Scheduling Tasks on Varying Partition Sizes in Partitionable Multiprocessor Systems," IEEE Trans. on Know. and Data Eng., vol. 41, no. 12, Dec. 1992.

100. C. Koelbel, and P. Mehrotra, "Compiling Global Name-Space Parallel Loops for Distributed Execution," IEEE Trans. on Para. and Dist. Sys., vol. 2, no. 4, pp. 440-451, Oct. 1991.
101. R. Koo and S. Toueg, "Checkpoints and rollback-recovery for distributed systems," IEEE Trans. Software Eng., vol. SE-13, no. 1, pp. 23-31, Jan. 1987.
102. H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," ACM Trans. Database Syst., vol. 6, no. 2, pp. 213-226, June 1981.
103. G. Kuper and M. Vardi, "On the expressive power of the logic data model," in Proc. ACM Int. SIGMOD Conf. 1985.
104. H. Kuss, "On totally ordering checkpoints in distributed databases," in Proc. ACM-SIGMOD Int. Conf. Management Data, Orlando, FL, pp. 293-302, 1982.
105. L. Lamport, "Time, clocks and ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558-564, 1978.
106. T. A. Landers and R. L. Rosenberg, "An Overview of Multibase - Heterogeneous Database System," Distributed Databases, H. J. Schneider, ed., North Holland, Amsterdam, pp. 153-184, 1982.
107. C. E. Landwehr, "Formal Models for Computer Security," ACM Comp. Sur., vol. 11, no. 1, pp. 247-278, Sept. 1981.
108. C. Lecluse, P. Richard, and F. Velez, "O2, An object-oriented data model," Proc. ACM Int. SIGMOD Conf., 1988.
109. A. Leff and C. Pu, "A Classification of Transaction Processing System," Computer, vol. 24, no. 6, pp. 63-76, June 1991.
110. G. Lelann, "Distributed systems - Towards a formal approach," in Proc. IFIP Congress, pp. 155-160, Aug. 1977.
111. P. J. Leu and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," in Proc. ACM-

SIGMOD Int. Conf. Management Data, pp. 154-163, 1988.

112. K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," ACM Trans. Computer Systems, vol. 7, no. 4, pp. 321-359, Nov. 1989.

113. B. Lindsay et al., "Computation and Communication in R*: A Distributed Database Manager," ACM Trans. Computer Systems, vol. 2, no. 1, pp. 24-38, Feb. 1984.

114. B. Liskov, "Distributed programming in Argus," Comm. ACM, vol. 31, no. 3, pp. 300-313, Mar. 1988.

115. W. Litwin et al., "MSQL: A Multidatabase Language," Information Sciences, vol. 49, June 1987.

116. W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," ACM Computing Surveys, vol. 22, no. 3, pp. 267-293, Sept. 1990.

117. W. Litwin, and T. Risch, "Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates," IEEE Trans. on Know. and Data Eng., vol. 4, no.6, pp. 517-528, Dec. 1992.

118. P. C. Lockemann and H. C. Mayr, "Information system design: Techniques and software support," Infom. Processing 86, North-Holland, pp. 617-634, 1986.

119. R. Lorie et al, "Supporting complex objects in a relational system for engineering databases," in Query Processing in Database Systems, W. Kim, D. Reiner, and D. Batory, Eds Berlin Germany: Springer, 1985.

120. J. McDermid, "Checkpointing and error recovery in distributed systems," in Proc. 2nd Int. Conf. Distributed Comput. Syst., pp. 271-282, 1981.

121. N. H. Madhavji, "Fragtypes: A Basis for Programming Environments," IEEE Trans. on Soft. Eng., vol. 14, no. 1, Jan. 1988.

122. E. Mafla, B. Bhargava, "Communication Facilities for

Distributed Transaction-Processing Systems," IEEE Computer, pp. 61-66, Aug. 1991.

123. Roberto Maiocchi and Barbara Pernici, "Temporal Data Management Systems: A Comparative View," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 504-524, Dec. 1991.

124. D. Maier and J. Stein, "Development and implementation of object-oriented DBMS," in Research Directions of Object-Oriented Programming, B. Shriver and P. Wegner, Eds. Cambridge, MA: The MIT Press Series of Computer Systems, pp. 355-392, 1987.

125. Roberto Maiocchi and Barbara Pernici, "Temporal Data Management Systems: A Comparative View," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 504-524, Dec. 1991.

126. E. B. Martin, C. H. Pedersen and J. B. Roberts, "An Object-Based Taxonomy for Distributed Computing Systems," IEEE Computer, pp. 17-27, Aug. 1991.

127. K. Marzullo, R. Cooper, M. D. Wood and K. P. Birman, "Tools for Distributed Application Management," IEEE Computer, pp. 42-51, Aug. 1991.

128. D. A. Moon, "The Common Lisp Object-Oriented Programming Standard," in Object-Oriented Concepts, Databases and Applications, W. Kim and F. Lochovsky, eds., Addison-Wesley, Reading, Mass, pp. 49-78, 1989.

129. S. J. Mullender et al., "Amoeba: A distributed operating system for 1990s," IEEE Computer, vol. 23, no. 5, pp. 44-53, May 1990.

130. Mullender and A. S. Tanenbaum, "Protection and Resource Control in Distributed Operating Systems," Computer Networks, pp. 421-432, Nov. 1984.

131. S. J. Mullender, G. V. Rossum, A. S. Tanenbaum, R. V. Renesse, and H. V. Staveren, "Amoeba, A Distributed Operating System for the 1990s," IEEE Computer, pp. 44-53, May 1990.

132. S. J. Mullender and A. S. Tanenbaum, "A distributed file

server based on optimistic concurrency control," ACM Oper. Syst. Rev., vol. 19, no. 5, pp. 51-62, 1985.

133. K. Narayanaswamy and W. Scacchi, "maintaing Configurations of evolving software systems," IEEE Trans. Software Eng., vol. SE-13, pp. 324-334, 1987.

134. S. Navathe, T. Sashidhar, and R. Elmasri, "Relationship merging in schema integration," Proc. 10th Int. VLDB Conf., pp. 78-90, Singapore, Aug. 1984.

135. S. Navathe, R. Elmasri, and J. Larson, " Integrating user views in database design," IEEE Computer Mag., pp. 50-62, Jan. 1986.

136. W. Nejdl, S. Ceri and G. Wiederhold, "Evaluating Recursive Queries in Distributed Databases," IEEE Trans. on Know. and Data Eng., vol. 5, no. 1, pp. 104-121, Feb. 1993.

137. G. M. Nijsen, "Current Issues in Conceptual Schema," In Architecture and Models in Database Management Systems, Edited by G. M. Nijsen, Amsterdam: North Holland Publishing, pp. 31-65, 1977.

138. B. Nitzberg, and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," IEEE Computer, pp. 52-60, Aug. 1991.

139. H. L. Ossher, "A mechanism for specifying the structure of large, layered systems," In Research Directions in Object-Oriented Programming, B. Shriver and P. Wegner, Eds. Cambridge MA: The MIT Press Series in Computer Systems, pp. 219-252, 1987.

140. M. Ozsoyoglu and L. Yan, "A Normal Form for Nested Relations," In Proc. 4th ACM Int. Symp. Principles Database Syst., 1985.

141. M. T. Ozsü and Patrick Valduriez, Principles of Distributed Database Systems, 1st ed., Addison-Wesley, Reading, Mass. 1990.

142. M. T. Ozsü and Patrick Valduriez, "Distributed Database

- Systems: Where Are We Now?," IEEE Computer, pp. 68-78, Aug. 1991.
143. Sudha Ram, "Heterogeneous Distributed Database Systems," IEEE Computer, pp. 7-10, Dec. 1991.
144. C. H. Papadimitrion, The Theory of Concurrency Control. Rockville, MD: Computer Science Press, 1986.
145. J. Peckham and F. Maryanski, "Semantic Data Models," ACM Comp. Sur., vol. 20, no. 3, pp. 153-189, Sep. 1988.
146. S. Pilarski and T. Kameda, "A novel checkpoint scheme for distributed database systems," in Proc. 9th ACM Symp. Principles Database Syst., Nashvillem, TN, pp. 368-378, Apr. 1990.
147. S. Pilarski, and T. Kameda, "Checkpointing of Distributed Databases: Starting from the Basics," IEEE Trans. on Para. and Dist. Sys., vol. 3, no. 5, Sep. 1992.
148. C. Pu, "On-the-fly, incremental; consistent reading of entire databases," Algorithmica, vol. 1, no. 3, pp. 271-287, Oct. 1986.
149. C. Pu, "Superdatabases for Composition of Heterogeneous Databases," in Integration of Information Systems: Brdging Heterogeneous Databases, A. Gupta ed., IEEE Press, Piscataway, N. J., pp. 150-157, 1989; also in Proc. of Fourth Internation Conference Data Eng., CS Press, Los Alamitos, Calif., Order No. 827, pp. 548-555.
150. C. Pu and A. Leff, "Replica Control in Distributed Systems: An Asynchronous Approach," Proc. 1991, ACM SIGMOD Intern. Conf. Management of Data, ACM, New York, pp. 377-386, May 1991.
151. C. Pu, C. H. Hong, and J. M. Wha, "Performance evaluation of global reading of entire databases," in Proc. Int. Symp. Databases in Parallel and Distributed Syst., Austin, pp. 167-176, Dec. 1988.
152. J. Ramanujam, P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," IEEE Trans. on

Para. and Dist. Sys., vol. 2, no. 4, pp. 472-482, Oct. 1991.

153. D. P. Read, "Implementing atomic actions on decentralized data," in Proc. 7th ACM Symp. Oper. Systems Principles, pp. 66-74, Dec. 1979.

154. P. Reisner, "Human Factor Studies of Database Query Languages: A Survey and Assessment," ACM Comp. Sur., vol. 13, no. 1, pp. 13-31, Mar. 1981.

155. N. Roussopoulos, "The logical access path schema of a database," IEEE Trans. Software Eng., vol. 8, no. 6, pp. 563-573, Nov. 1982.

156. N. Roussopoulos and H. Kang, "A Pipeline N-way Join Algorithm Based on 2-way Semijoin Program," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 486-495, Dec. 1991.

157. E. A. Rundensteiner and L. Bic, "Set Operations in Object-Based Data Models," IEEE Trans. on Know. and Data Eng., vo. 4, no. 3, pp. 382-399, June 1992.

158. M. Rusinkiewicz, A. Sheth, and G. Karabatis, "Specifying Interdatabase Dependencies in Multidatabase Environment," IEEE Computer, pp. 46-53, Dec. 1991.

159. K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," Tech. Rep. CS-TR-126-87, Dep. Compt. Sci., Princeton Univ., Dec. 1987.

160. R. E. Schantz, R. M. Thomas, and G. Bono, "The architecture of the Cronus distributed operating system," Proc. Sixth Int'l, Conf. on Distributed Computing Systems, CS Press, Los Alamitos, Calif., Order no. 697, pp. 250-259, 1986.

161. H. Schek and M. Scholl, "The relational model with relational valued attributes," Inform. Syst, vol. 11, no. 2, pp. 137-147, 1986.

162. G. Schlageter and P. Dadam, "Reconstruction of consistent global states in distributed databases," in Proc. Int. Symp. Distributed Databases, pp. 191-200, 1980.

163. A. Sheth and J. Larson, "Federated Database Systems and Managing Distributed, Heterogeneous, and Autonomous Databases," ACM Computing Surveys, Special Issue on Heterogeneous Databases, vol. 22, no. 3, pp. 183-236, Sept. 1990.
164. M. Shapiro et al., "SOS: An Object-Oriented Operating System - Assessment and Perspectives," Computing Systems, vol. 2, no. 4, pp. 287-337, Dec. 1989.
165. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna Distributed Operating System," IEEE Software vol. 8, no. 1, pp. 66-73, Jan. 1991.
166. B. Shriver and P. Wegner, Eds., Research Directions in Object-Oriented Programming. Cambridge, MA: The MIT Press Series in Computer Systems, 1987.
167. T. W. Sidle, "Weakness of commercial database management systems for engineering applications," in Proc. 17th Design Automat. Conf., pp. 57-61, 1980.
168. P. K. Sinha, M. Maekawa, K. Shimizu, X. Jia, H. Ashihara, N. Utsunomiya, K. S. Park, and H. Nakano, "The Galaxy Distributed Operating System," IEEE Computer, pp. 34-41, Aug. 1991.
169. R. Snodgrass, "The Temporal query language TQuel," ACM Trans. Database Syst., vol. 12, no. 2, pp. 247-298, June 1987.
170. A. Z. Spector, R. F. Pausch, and G. Bruell, "Camelot: A Flexible Distributed Transaction Processing System," Proc. Comcon Spring 88, CS Press, Los Almitos, Calif., Order no. 828, pp. 432-439, Mar, 1988.
171. D. Stemple, A Socorro, and T. Sheared, "Formalizing objects for databases using ADABTPL," in Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science 334, K. R. Dittrich, Ed. Berlin, Germany: pp. 110-128, Springer 1988.
172. M. Stonebreaker, "The case for shared-nothing," Database Eng., vol. 9, no. 6, June 1986.
173. M. Stonebreaker and L. Rowse, "The design of POSTGRES,"

in Proc. ACM SIGMOD Conf., pp. 340-355, 1986.

174. Sudha Ram, "Heterogeneous Distributed Database Systems," IEEE Computer, pp. 7-10, Dec. 1991.

175. Tzong-An Su, and Gultekin Ozsoyoglu, "Controlling FD and MVD Interfaces in Multilevel Relational Database Systems," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 474-485, Dec. 1991.

176. Y. Takahashi, "Fuzzy Database Query Languages and Their Relation Completeness Theorem," IEEE Trans. on Know. and Data Eng., vol.5, no. 1, pp. 122-125, Feb. 1993.

177. A. S. Tanenbaum et al., "Experiences with the Amoeba Distributed Operating System," Comm. ACM, vol. 33, no. 12, pp. 46-63, 1990.

178. Buuba Team, "Prototyping Bubba, A highly parallel database system," IEEE Trans. Know. Data Eng., vol. 2, Mar. 1990.

179. G. Thomas et al., "heterogeneous Distributed Database Systems for Production Use," ACM Computing Surveys, vol. 22, no. 3, pp. 237-266, Sept. 1990.

180. K. Tsuda, K. Yamamoto, M. Hirakawa, M. Tanaka and T. Ichikawa, "MORE: An Object-Oriented Data Model with a Facility of Changing Object Structures," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 444-460, Dec. 1991.

181. M. Tsur and C. Zaniolo, "An implementation of GEM - Supporting a semantic model on a relational backend," in Proc. ACM Int. SIGMOD Conf., 1984.

182. G. S. D. Varma, R. C. Joshi and K. Singh, "Intelligent Environment Design for the Integration of Schemas in Object-Oriented Heterogeneous GURU Distributed Database Management System," Accepted at Sixth ISCA* International Conference on Parallel and Distributed Computing Systems, October 14-16, 1993 * Louisville, Kentucky, USA.

183. G. S. D. Varma, R. C. Joshi and K. Singh, "Complexity

Measures in GURU Heterogeneous Object-Oriented Distributed Database Management System," Accepted at the 36th Midwest Symposium on Circuits and Systems, August 16-18, 1993, Co-sponsored by Wayne State University, Detroit, MI, USA and IEEE Circuits and Systems Society * Detroit, Michigan, USA.

184. G. S. D. Varma, R. C. Joshi and K. Singh, "Intelligent Object-Oriented Heterogeneous Local Query Automation in GURU Distributed Database Management System," Communicated to Special Issue of the Journal of Parallel and Distributed Computing on Scalability and Parallel Algorithms and Architectures, USA.

185. G. S. D. Varma, R. C. Joshi and K. Singh, "Intelligent Object-Oriented Heterogeneous Autonomous Approach to Design Schema in GURU Distributed Database Management System," To be Communicated.

186. R. C. Joshi, G. S. D. Varma and K. Singh, "Message-Task Scheduling in Object-Oriented Heterogeneous GURU DDBMS," Accepted at 23rd Annual Conference, 1994, International Conference on Parallel Processing, August 15-19, 1994, Pennsylvania State University, USA.

187. G. S. D. Varma, R. C. Joshi and K. Singh, "Dynamically Changing Object Structures in Object-Oriented Heterogeneous GURU DDBMS," Communicated to 23rd Annual Conference, 1994, International Conference on Parallel Processing, August 15-19, 1994, Pennsylvania State University, USA.

188. G. S. D. Varma, R. C. Joshi and K. Singh, "Object View Protection in Object-Oriented Heterogeneous GURU Distributed Database Management System," Communicated to 23rd Annual Conference, 1994, International Conference on Parallel Processing, August 15-19, 1994, Pennsylvania State University, USA.

189. C. Wang, A. L. P. Chen, Shio-Chen Shyu, "A Parallel, Execution Method for Minimizing Distributed Query Response Time,"

IEEE Trans. on Para. and Dist. Sys., vol. 3, no. 3, pp. 325-333, May 1992.

190. Y. R. Wang and S. E. Madnick, "A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective," Proc. 16th VLDB Conf., Morgan Kaufman, Palo Alto, Calif., pp. 519-538, 1990.

191. W. Wegner, "The Object-Oriented Classification Paradigm," in Research and Directions in Object-Oriented Programming, B. Shriver and P. Wegner, eds., MIT Press, Cambridge, Mass., pp 477-560, 1987.

192. K. Wilkinson, P. Lyngback, and W. Hasan, "The Iris Architecture and Implementation," IEEE Trans. Knowledge and Data Eng., vol. 2, no. 1, pp. 63-75, 1990.

193. T. Y. C. Woo, and S. S. Lam, "Authentication for Distributed Systems," IEEE Computer, pp. 39-52, Jan. 1992.

194. D. Woelk and W. Kim, "Multimedia information management in an object-oriented database system," in Proc. 13th Int. VLDB Conf., pp. 319-329, 1987.

195. T. Y. C. Woo, and S. S. Lam, "Authentication for Distributed Systems," IEEE Computer, pp. 39-52, Jan. 1992.

196. Seung-Min Yang, K. H. (Kane) Kim, "Implementation of Conversation Scheme in Message-Based Distributed Computer Systems," IEEE Trans. on Para. and Dist. Sys., vol. 3, no. 5, pp. 555-572, Sep. 1992.

197. Cheong Youn, Hyung-Joo Kim, Lawrence J. Henschen and Jiawei Han, "Classification and Compilation of Linear Recursive Queries in Deductive Databases," IEEE Trans. on Know. and Data Eng., vol. 4, no. 1, pp. 52-67, Feb. 1992.

198. P. S. Yu, H. U. Heiss and D. M. Dias, "Modeling and Analysis of a Time-Stamp History Based Certification Protocol for Concurrency Control," IEEE Trans. on Know. and Data Eng., vol. 3, no. 4, pp. 525-537, Dec. 1991.

199. C. Zaniolo, "The representation and deductive retrieval of Complex Objects," in Proc. 11th Int. Conf. Very Large Databases, 1985.

200. S. Zdonik, "Object-Oriented Type Evolution," in Advances in Database Programming Languages, F. Bancilhon and P. Buneman, eds., Addison-Wesley, Reading, Mass., pp. 277-288, 1990.

201. S. Zhou, M. Stumm, K. Li, and D. Wortman, "Heterogeneous Distributed Shared Memory," IEEE Trans. on Para. and Dist. Sys., vol. 3, no. 5, Sep. 1992.