DISTRIBUTED DATABASES ON LOCAL MULTIACCESS COMPUTER SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Yao-Nan Lien

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 1986

To my mother,

my brothers,

Shu-Man, and

Clinton

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Professor Benjamin W.-S. Wah for his guidance, advice, and support during the research for this thesis. Without his inspiration, encouragement and friendship, this research could not be finished. I also wish to thank my committee members Professor Frederic J. Mowle, Professor Rangasami L. Kashyap and Professor Bharat Bhargava for their suggestions and comments. A special thanks goes to Professor Y. W. Ma of the Pennsylvania University for her precious comments. I also want to thank Mrs. Wah, the colleagues in our research group, and my friends who helped me to have a pleasant time during my study and research in Purdue University.

A special thanks goes to the Department of Computer and Information Science, The Ohio State University, for kindly providing me an excellent research environment since January 1986.

This thesis is fortunate to receive consistent support from the National Science Foundation under grants ECS80-16580 and ECS81-059681, and the CID-MAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milieron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Lien, Yao-Nan. Ph.D., Purdue University. August 1986. Distributed Databases On Local Multiaccess Computer Systems. Major Professor: Benjamin W.-S. Wah.

Concurrency control, distribution design, and query processing are key issues in the design of distributed databases. In this research, we have studied these issues with respect to a relational database on a local computer system connected by a multiaccess broadcast bus. A broadcast bus not only allows information to be distributed efficiently but also provides a synchronization tool for the concurrency control. Consequently, it simplifies the distributed database design. Some NP-hard file placement problems are found to be solvable in polynomial time when updates are broadcast. A transaction model that integrates the concurrency control and query processing is proposed. In concurrency control, a protocol based on locking is used since much less communication overhead is needed for lock management. A dynamic strategy is used in query processing, as less data are transferred when compared to a static strategy. The status information needed in dynamic query processing can be conveniently obtained by broadcasting. Lastly, a new query processing strategy is proposed for the distributed database connected by the fast communication networks. The problem is formulated as a generalized traveling salesman problem, and is solved by various solution algorithms. The performance of the system is evaluated by simulations on a SUN workstation. The results shows the design is feasible and efficient.

CHAPTER I

INTRODUCTION

With digital computer technologies, *Database Management Systems* (DBMS's) provide better service for maintaining and accessing information over conventional information processing technologies. Expensive computer systems and the lack of computer networking technology in 1970's had limited the DBMS to be centralized in a computer complex, consisting of a few large computers, referred to as *centralized DBMS's* (CDBMS's). With the advance of computer networking technologies, a *Distributed Database Management System* (DDBMS) allows a single database to be efficiently and cost effectively spread over a computer network that consists of a number of geographically dispersed computers interconnected by a communication network. In the last decade, DDBMS's have been studied by a number of researchers [APE83] [BHA82a] [BHA83] [BER81a] [BER81b] [BER81c] [CER83] [CHI84] [CHU82] [DOW82] [HEV79a] [PAP79] [RAM83] [SAC84] [WAH85] [WON77] [HEV85] [YU84a] [YU84b]. There are some pioneering systems developed: SDD-1 [ROT80], System R^* [LIN84], Distributed INGRES [STO77], and LOCUS [POP83]. As the need of office automation and the capability of smaller computers grow rapidly, a DDBMS is now considered as a good candidate offering better information processing service for an environment in which users of a single database are spread within a limited distance area, referred to as *local area*. University environment is a typical example of such environments. DDBMS's in local area

environment are getting more and more attentions [CHA83] [HEV85] [KER82] [NGU81a] [SAC84] [WAH85]. In this research, we aim at studying the DDBMS's in local area environments and providing a feasible configuration accomplished with adequate control functions.

1.1 Distributed Database vs. Centralized Database

The major advantages of DDBMS's over CDBMS's are their superior achievements in reliability, data availability, throughput, and flexibility. Under some circumstances, a DDBMS may also provide faster response time and lower cost. On the other hand, the control functions in DDBMS's are much more complicated. There is a good discussion in Sykes's paper [SYK80].

(1) **Reliability.** Reliability is the probability of a system keeping functioning. In a distributed environment, failures on some components should not disable the entire system. Users' requests at the failed sites can be transferred to other sites; failures on a communication link can be handled by routing the traffic to alternate links. Although high reliability can also be achieved in centralized systems, the cost is much higher in general.

(2) **Availability.** Availability of data is the probability of at least one copy of each piece of data being available in the system. In addition to the reliability, data availability is also a desirable feature of DBMS's. By replicating the data at different sites in DDBMS's, the chance that all access pathes to a particular piece of data are broken is greatly reduced. The availabilities of other resources such as processing power may also be enhanced by the resources sharing capability of distributed systems.

(3) **Throughput**. Throughput is usually measured by the number of service requests (called *transactions*) that a system can process within a time unit. In a DDBMS, a transaction may be carried out by multiple processing units at different sites, or by the processing unit at any site with idle processing capacity. In other words, DDBMS's could have better load balancing, and hence larger throughput.

(4) **Flexibility/Extensibility**. The flexibility of upgrading system capacity is desirable whenever the future demand on the capacity is unpredictable at the design stage or is changing rapidly. Without such flexibility, a system may be over-specified at the design time to avoid the potential mismatch between the capacity and the future demand. Also, even if the future demand is predictable, the utilization of a system designed to match the future demand is very low before the demand matches the capacity. Those drawbacks can be avoided by allowing the system to be dynamically upgraded in response to changing demand. In a well-designed distributed system, adding new components or replacing old components have only little impact to the entire system. Consequently, high extensibility of DDBMS's is relatively easy to achieve, since the dependency among all components is usually much lower than that in CDBMS's.

(5) **Local Autonomy**. Each local site can have more design freedom to meet specific requirements of its local environment. Usually, the requirements of different user groups (eg., engineering, accounting, marketing, ... etc) in an organization may be very diverse and, even worse, may sometimes conflict with each other. In a DDBMS, each individual need can be easily satisfied

since each site can have its own autonomy. In contrast, it's very difficult for a CDBMS to satisfy all different needs. Even when tradeoffs among these needs are made, none of the users might be completely satisfied with the result.

(6) **Response time**. Most of the transactions in a well-designed DDBMS with good resource allocation strategies may have a great chance to be processed locally without extensive network activities. The average response time can be very short in this situation. However, certain types of transactions in DDBMS's may involve many complicated control functions making the response time relatively high comparing to CDBMS's.

(7) **Cost**. Accessing a centralized database from a remote site may involve expensive communications. A well-designed DDBMS may localize most of database accessing and hence, save expensive communication cost. However, under some circumstances, the overhead of the complicate control functions may be too high to be compensated by the benefit obtained from the localization of remote accesses.

1.2 Motivations of DDBMS's in Local Area Environment

In addition to the advantages described in Section 1.1, there are two driving forces that make the DDBMS a good candidate to provide good information processing service in a local area environment with physically dispersed users. The diameter of a local area environment is usually within a few kilometers. A hospital, a building with branch offices, or a manufacturing factory are some of the examples.

The first driving force comes from the need to improve office productivity due to the rapid growth of information volume and personnel cost. *Office Automation*, which tries to computerize most of the office functions, has been considered as the most promising approach to achieve this goal. *CAD/CAM/CAE*[*], as another example, have attracted a lot of attention for its potential to enhance engineering and manufacturing productivities. In these environments, information storage, maintenance, and retrieval are considered important services to support higher productivity.

The other driving force comes from the advance of technologies. Fast growing microelectronics and local area networking technologies in recent years have made local area networks, which connect smaller computers with reasonable processing power, inexpensive enough such that almost all companies can afford to install one.

The cost of fabricating a given logic element (processor, memory, peripheral controller) has been declining by a factor of 10 every three years over the last decade. Another way of looking at technological advances is to compare the size of memory on a single chip. Starting from 4 Kbits in 1973, it increased to 256 Kbits in 1979, and 1 Mbits DRAM today. The present density of logic on a chip is quite high such that any complex logic element can be built with a few chips at a cost lower than $1000. A workstation with 1-Mbyte main memory, 1-MIPS (million instruction per second) processing power, and 1-Mbps networking capability is available in less than $10,000. Current prices may fall in the future because of even larger production volumes and the rapidly developing trend to standardize chip interface logic. Further, smaller systems are much easier to

[*] CAD/CAM/CAE : computer aided design, computer aided manufacturing, and computer aided engineering.

operate and maintain than larger systems. This is even more important to small enterprises, since the shortage of skillful system engineers of large systems has made the central computing centers difficult to set up, even if they are affordable.

Recently, local area networks (LAN) have been extensively developed and used. These networks make use of such technologies as twisted pair, coaxial cable, radio transmission medium, as well as more sophisticated technologies like fiber optics. LAN can be installed in a building, on a campus, in a factory, etc. Many experimental and commercial LAN's have been in use for years, and both their feasibility and cost-effectiveness have been widely demonstrated.

Although CDBMS's may outplay DDBMS's on the efficiency in local area environments, a DDBMS is still considered as a better alternative for all its advantages other than efficiency and implementation complexity. In fact, the implementation complexity is not an important factor from the users' point of view as long as a good user interface is provided.

1.3 Design Objectives and Design Issues of DDBMS's

The objectives of DDBMS's are different from system to system. Most commonly addressed objectives are data integrity, cost effectiveness, good user interface, robustness, and simplicity. In fact, these objectives are generally applicable to most DBMS's.

1.3.1 Design Objectives

(1) **Data integrity**. In a DBMS capable of processing more than one transaction at a time, transactions may interfere with each other such that either unexpected results are produced or the database itself is converted into incorrect states. Therefore, maintaining data integrity in processing transactions is one of the most important objectives of any DBMS.

(2) **Cost effectiveness**. A DBMS should maximize its performance within a cost constraint, or in another way, meet the performance requirements with the least cost. The performance requirements are also different from system to system. These may range from minimizing response time or cost, to maximizing throughput or reliability, or a combination of more than one parameter.

(3) **Good user interface**. Without a good user interface, a high performance system may be of little use or even useless to the users. In addition to good query languages as needed in CDBMS's, distribution transparency is also needed in DDBMS's [CER84]. Users do not have to know the location of any piece of data; the update to multiple copies of any piece of data should be automatically taken care of by the system itself. Further, a DDBMS may consist of some different local databases. It would be very difficult for users to access the database through different local systems a global data model is not available. Therefore, a global conceptual data model is desirable.

(4) **Robustness**. This is the capability of a system to operate correctly in the presence of any failure or unexpected transactions. For example, communication-link failures may partition a system into isolated parts. A robust system should be able to continue the services with possibly degraded performance even if the hardware fails. Also, it should be able to

recover the system back to a correct state, once the failed components are operational. It is generally true that the more complex a system is, the higher the probability that failures may occur. Therefore, the system designers should anticipate the failures and provide some ways to recover the system from failures automatically. In a worse case, a DBMS may operate with undetected errors if the failures do not stop the system. For example, a noise on the data path along which an update request is transmitted may change the database incorrectly if the system does not have error-detection and error-correction capabilities. As another example, two or more transactions may deadlock to each other. The system may eventually stop if it does not have any deadlock-prevention or deadlock-detection mechanisms. Robustness is particularly important in a DDBMS since its control functions are generally much more complicated than those in a CDBMS.

(5) **Simplicity**. For a complicated system such as a DDBMS consisting of many components, it is difficult to prove its correctness, to make it robust, to document it, to maintain it, and to update it. Hence, simplicity is also an important objective in designing a DDBMS. However, very often, this can only be done at the cost of efficiency and performance. Compromises should be made in this case.

1.3.2 Design Issues

Although there is almost no difference between CDBMS's and DDBMS's on their objectives, there are some problems in DDBMS's that cannot be handled well by techniques developed for CDBMS due to the difference between their operating environments. The followings are some key design issues in DDBMS's.

(1) **Architecture**. The design of a DDBMS is a complex and difficult task requiring careful consideration in all design aspects. A systematic architectural design can help designers solve the problem easily. In the simplest view, a DDBMS consists a set of computer systems interconnected by a communication network as shown in Figure 1.1. Each local system is equipped with a local database. According to the homogeneity of local systems, three types of integration can be used [PEE78]. In the first approach, each local DBMS 'knows about' the existence of others. All the components work together cooperatively like a single integrated DBMS system. This approach is possible only if the entire DDBMS is designed and built from the ground up. If this is not the case, existing DBMS's should be used. One of the two forms of 'federated' models should be used. Integrators and translators must be used to integrate different DBMS's into an integrated DDBMS system and to translate different database models into a unified conceptual model. The logical model of this approach is shown in Figure 1.2. More detailed discussions can be found in Peeble's paper [PEE78].

(2) **Computer network**. This is one of the essential parts of the architecture design, providing the service for information exchange and resource

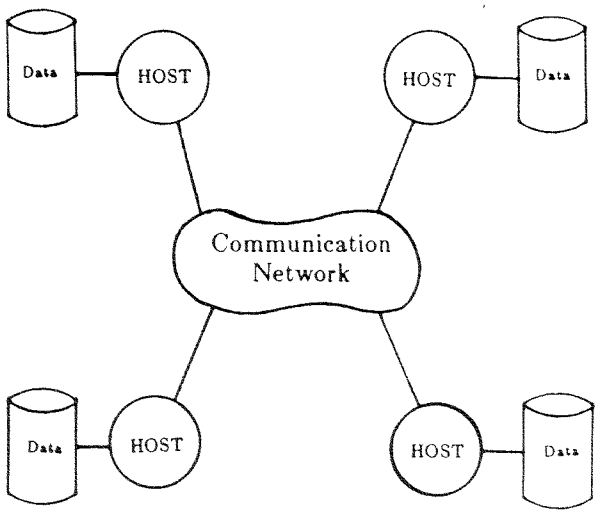


Figure 1.1 A simple model of a DDBMS.

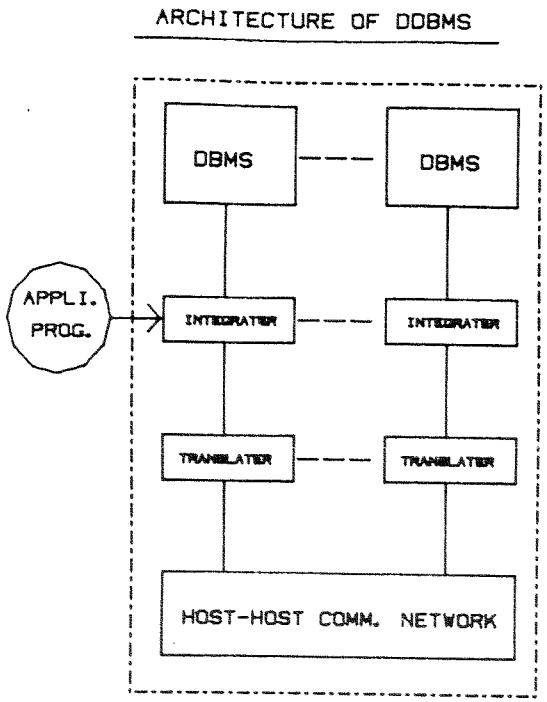


Figure 1.2 Logical Architecture of A Distributed Database System.

sharing. Basically, a network is a collection of two or more processing elements that offer various services and capabilities. A network consists of a certain type of transmission medium connecting all processing elements in a certain topology, the control mechanisms controlling the transmissions over the medium, some type of network interfaces, and a set of protocols controlling the transmission of information among the basic hardware elements. The combination of different choices on these components may result in a large number of networks differing in characteristics such as propagation delay time, communication capacity, addressing scheme, transmission cost, and effective network diameter. A good DDBMS needs a carefully designed computer network since most of the problems in DDBMS's cannot be solved well without considering the characteristics of the underlying networks.

(3) **Data Distribution**. The design of data distribution considers the allocation of data such that data can be accessed and maintained efficiently. In a DDBMS, replicated copies of a piece of data are allowed to be stored at different sites such that some expensive remote accesses can be localized. On the other hand, the overhead of maintaining the integrity of replicated data, the directory information, and the need for extra storage inhibits data from being over replicated. A careful allocation for the replicated data to balance the benefit and overhead is needed to maximize the cost effectiveness [CHU69] [DOW82].

(4) **Concurrency Control**. Concurrency Control maintains data integrity in a multiuser DBMS. It is complicated by some factors in DDBMS's as

follows. Multiple copies of a piece of data being stored, accessed, and updated at different sites at the same time; communication delays prohibit instantaneous distribution of status information such that it is difficult to maintain all sites in an identical state at all times; and the unexpected failures of some components may cause the database in different sites to be modified in different ways[BER81b].

(5) **Query Processing**. The operations performed on the data in a database is called a *query*. In a DDBMS, a query may originate from any site in the system. When processing a query, either the required files are assembled at a single site and the query is processed there, or the query and the intermediate results are sent and processed sequentially through the sites containing the files. A combination of the two strategies is also possible. The design of effective strategies of processing queries to maximize the system efficiency is known as the DQP, or *distributed query processing problem* [APE83] [YU84b].

Some other problems such as failure recovery, security/privacy, and logic database design are also relatively important.

All these issues are interrelated together as shown in Figure 1.3. For example, the files are partitioned and placed according to the characteristics of the network and the query; the design of the query processing strategy, the concurrency control algorithm, and the communication network depend on the placements and partitioning of the files. The issues shown in Figure 1.3 are also interrelated to other important problems, such as failure recovery, logical database design, and directory management. It is very difficult to solve these

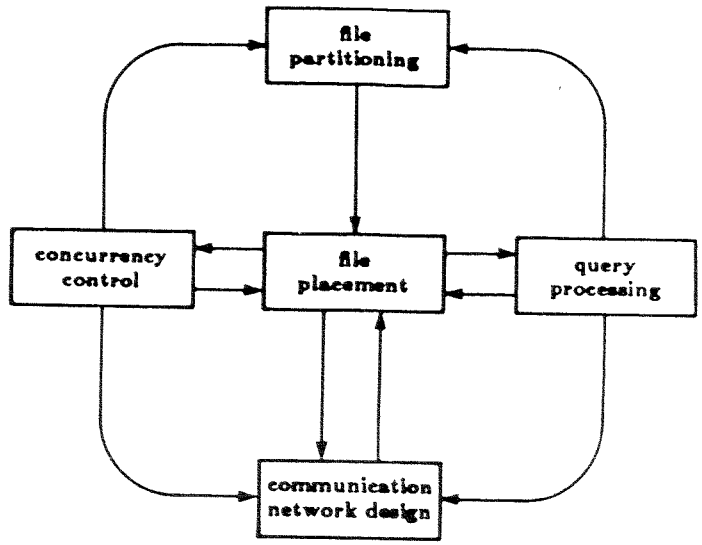


Figure 1.3 The relationships of design issues in distributed databases.

problems as a whole, so the designer usually decomposes them into independent problems and studies each based on simplified assumptions of others.

One potential problem of the above approach is that oversimplification may lead to undesirable results. An example is shown in the use of a network model that is independent of the hardware characteristic. A simple network model is often used because, besides simplifying the design of DDBMS strategies, it allows the software to be transportable, and permits many applications of different requirements to share the same network. A protocol hierarchy is used, which allows the DDBMS to interact with the lower levels through system calls. As a result, the status information needed in the control and the interchange of data among sites must be formulated into messages that are recognized by the lower levels of the hierarchy. This mismatch between the characteristic of the physical network and the requirements of the DDBMS results in inefficiency and increases the complexity of control strategies. To improve the performance, the capabilities of the network must be taken into account in the design. Based on this consideration, we have developed the objectives and the methodology of this research in Section 1.5.

1.4 DDBMS's in Local Area Networks

Since most of the problems of DDBMS's are so strongly dependent on the supporting network, it would be a good methodology to design the supporting network first and then solve the other problems based on the characteristics of the network. As a matter of fact, local-area-network (LAN) technology is getting matured and standardized * [IEEE83]. There is no need to design a special LAN

* In 1980, the IEEE set up a Technical Committee, known as the 802 Committee, whose purpose is to establish standards for local area networks.

to support the DDBMS on local area environment if any existing LAN can support the DDBMS well. Therefore, the best methodology is to choose an existing LAN whose characteristic is close to what a DDBMS may need and to design the DDBMS based on the chosen LAN.

1.4.1 Local Area Networks

A *local network* is a communications network that provides for the interconnection of a variety of data-communication devices within a small area. A *local area network* (LAN), a special case of local networks, is a general purpose local network supporting minis, mainframes, terminals, and other peripherals [STA84]. The data rate can range from 0.1 to 20 Mbps, the distance from 0.1 to 50 kilometer, and error rate from 10^{-8} to 10^{-11}. The principal technological alternatives that determine the nature of a LAN are the *transmission medium* and the *topology*. There is a good survey in Stalling's paper [STA84].

(1) **Transmission Medium**. Table 1.1 lists the types and the capacities of the transmission media that can be used in a LAN. The most commonly used media are twisted pair, coaxial cable, and optical fiber.

   (a) **Twisted pair wiring**. Twisted pair wiring is the simplest one with the lowest capacity and cost. It is the most cost effective choice for single-building, low-traffic requirements. Only baseband transmission, which uses digital signaling, is suitable for twisted-pair wiring. One weakness of twisted-pair wiring is in its susceptibility to interference and noise.

   (b) **Coaxial Cable.** High-performance requirements can be met by coaxial cables, which provide higher throughput, can support a large number of devices, and can span greater distances than twisted pair.

Table 1.1 Characteristic of local area networks.

Transmission medium	Twisted pair, coaxial cable, optical fiber
Topology	Bus, tree, ring, star
Transmission speed	0.1-20 Mbps
Maximum distance	~ 25 kilometers
Number of devices supported	10's-1000's
Attachment cost	500-5000

Both baseband and broadband transmission methods can be employed on a coaxial cable. In general, baseband transmission is simpler but less capable than broadband transmission. Baseband systems are typically from 1 to 10 Mbps and are generally limited to a single building. Broadband systems can support multiple data paths, each with 1 to 10 Mbps capacity. Existing CATV cabling technologies can be used in broadband LAN.

(c) **Optical Fiber.** Optical fiber has greatest capacity and is a promising candidate for future LAN's. It also has the following advantages over both twisted-pair wiring and coaxial cables: light weight, smaller diameter, low noise susceptibility, and no emissions. However, it has been little used so far due to its high cost and technical limitations. It is only suitable in to point-to-point configurations. The problem of insertion-tap loss must be overcome to use it in multipoint configurations.

(2) **Topology.** Characteristics of LAN's are also highly dependent on the topology used. Star, ring, bus, and tree are the most commonly used topologies. They are shown in Figure 1.4.

(a) **Star.** In star topology, a central switching elements is used to connect all the nodes in the network. Only one node can use the network to transmit or broadcast at any time. Therefore, a connection request to the central switching node is needed to establish a connection from one station to other stations.

(b) **Ring.** The ring topology consists of a closed loop, with each node attached to a repeating element, called *repeater*. Data circulate around the ring in a series of point-to-point data links between repeaters.

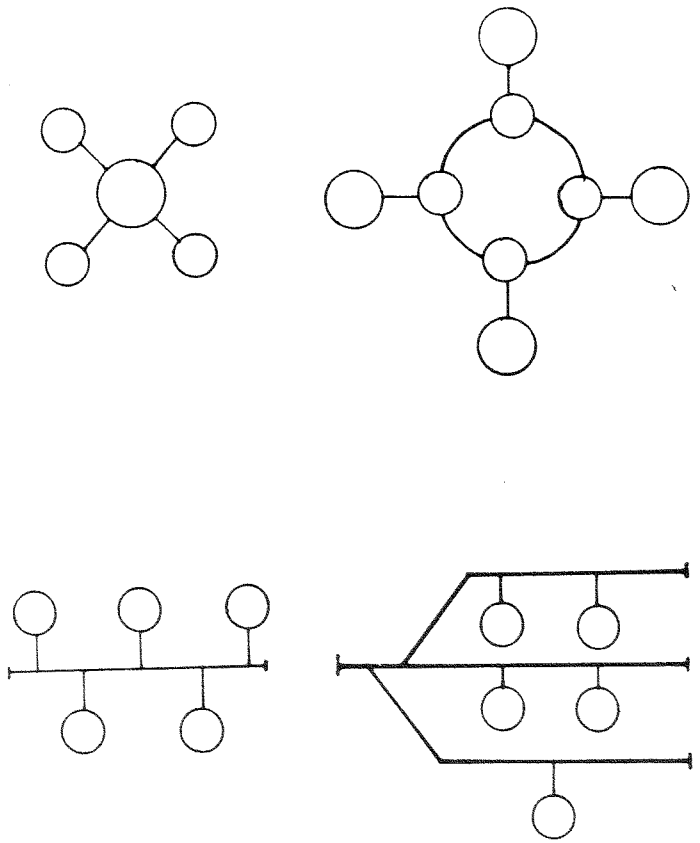


Figure 1.4 Topologies of local area networks (a) star, (b) ring (c) bus (4) tree.

Optical fiber is well suited in ring topologies such that its higher capacity can be utilized. Token rings, slotted rings, and insertion rings are the most commonly used access control mechanisms [STA85]. A station wishing to transmit data should get the access right first and then transmit the data through the ring in packets with source and destination addresses. As the packet reaches the destination, the destination node copies the data. There are some problems associated with a ring topology. The reliability of the ring is jeopardized by a single failure on cables or repeaters. These failures are difficult to isolate. Further, adding new stations into the ring will degrade the performance.

(c) **Bus/Tree.** Multipoint medium is used in bus or tree topology. The bus is a special case of the tree, in which there is only one trunk with no branches. Each station uses a passive interface to tap into the single bus. Since there is no active device on the bus, the reliability of bus LAN's is much higher than the ring topology LAN's. All stations share a common transmission medium, while only one station can use the medium to transmit at any time. A distributed-medium access protocol is needed to determine the station to transmit through the medium. Broadcast capability can be easily achieved since all stations 'listen' to the medium. The broadcast capability is very useful for DDBMS's. This will be discussed in the next section.

1.4.2 Local Multiaccess Bus Networks

Currently, the most popular bus LAN is the Ethernet [MET76] [SHO82]. Most of the low-cost, twisted-pair LAN's for microcomputers use a bus topology. The following are the key characteristics of local multiaccess bus LAN's.

(1) **Multipoint configuration.** More than one station are connected to the bus. Each transmitted message has the addresses of the source and destination stations. Bus access control protocols are needed.

(2) **Serial transmission.** Since all stations share a single data bus, only one station can transmit at any time. This could become the bottleneck of the system. The utilization of the bus is an important issue in system design.

(3) **Broadcast Capability.** All stations can listen to the bus simultaneously. The broadcast capability can be easily implemented. Many Ethernets support the broadcast capability [MET76] [SHO82]. Another useful potential capability is *multicasting*, in which messages may be targeted to more than one destination. This is also included in the Ethernet specification.

(4) **Baseband/Broadband**. Both baseband and broadband transmissions can be employed except in the twisted-pair wiring bus, in which only baseband transmissions are allowed. If broadband transmission is used, more than one channel can be available. Virtually, it can be modeled as a multi-bus LAN in some cases.

As other multipoint configurations, a multiaccess bus needs bus access control. The access control is the protocol to determine which station has the right to transmit data when more than one station wants to transmit. Either centralized or distributed control can be used. A centralized control may have

the reliability problem since the entire network will fail if the control station fails. Furthermore, a heavy traffic to the control station may be induced by the access requests generated by the slave stations The most commonly used distributed protocol is the *carrier sense multiple access with collision detection (CSMA/CD)* protocol. A station wishing to transmit should first sense the existence of the carrier signal on the bus to avoid destroying the current transmission. It can only transmit messages when the bus is idle (when no carrier signal presents on the bus). Otherwise, it should wait and keep sensing the bus until it is idle. A *collision* happens when more than one station transmit messages at the same time. Once a collision is detected, the transmitting stations should stop the transmission immediately and restart the procedure after a random period of time. Usually, a 'binary exponential backoff' algorithm is applied to determine the range of time within which the next attempt to retransmit will be taken.

1.4.3 End-to-end Transmission Speed of Multiaccess Bus Networks

The most important characteristic of a LAN in designing a DDBMS is the end-to-end communication speed, which is defined as the amount of information that can be transferred from the memory of one station in the original form to the memory of the other station in a ready-to-use form in a time unit. It is also called the *effective data rate*. We use these two terms interchangeably in this dissertation. It is much slower than the raw data rate on these networks because the overhead on the communication protocols, such as error detecting/correcting, medium-access control, flags, addressing, buffering, routing, packetization, flow control, sequencing, and synchronization, may reduce the effective date rate dramatically. For example, the typical end-to-end

transmission speed of a 10 Mbps Ethernet is only 1 Mbps. The TCP/IP protocols on a VAX-11/780 computer runing 4.2BSD UNIX with a Ethernet may have only 0.5-0.75 Mbps end-to-end transmission speed.

1.4.4 DDBMS's on Local Multiaccess Bus Networks (DDBLMN)

The broadcasting capability and multi-point configuration are the most important reasons that make the local multiaccess network a good choice for supporting a DDBMS in local area environments.

(a) The update of multiple copies of a single piece of data can be made by one broadcast. This not only saves operating cost but also simplify the design.

(b) The costs of remote accesses and updates are site independent. This can also simplifies the design.

(c) System status is almost completely available to all stations by monitoring the activities on the bus. The status information exchange is, therefore, extremely small.

(d) Every station gets the information on the bus almost instantaneously, since it is a multi-point configuration. For example, the minimum contention-slot time, which is at least twice the time to propagate a signal from one end of the bus to the other end, in Ethernet specification is 512 *bit − times** [SHO82]. The maximum propagation delay on the Ethernet, hence, will not excess 256 bit-times. Comparing this to the minimum packet size of 576 bits, the propagation delay will not cause any problem. If we assume that the database state cannot be changed within a packet time, it's reasonably

* A bit-time is defined as the time that the transmission system needs to transmit a single bit. For example, in a 1-Mbps system, 1 bit time is 1 μ second.

safe to assume that the status exchange is instantaneous.

(e) The multiaccess bus is a synchronization tool. When messages are not lost, all messages arrive at each site in the same order as they are sent. For example, if broadcast message m_1 is received at one site before a message m_2, m_1 is received at all sites before m_2. Therefore, the communication bus provides a synchronized communication environment [BAN79].

1.5 Relational Data Model

The *data model* of a database provides a logical view of the database to the users that can help users to access and manipulate the database without the knowledge of complicated physical organization of the database [COD70]. The two basic elements of a data model are:

(a) a mathematical notation for expressing data and relationships among data elements, and

(b) operations on the data that serve to express queries and other manipulations of the data elements[ULL82].

Among three different models (relational, hierarchical, and network) that have been extensively studied, the relational model is the most popular one due to its simplicity, symmetry, easy to use, and its strong theoretical foundation. For the same reason, this research is based on the relational data model.

The mathematical concept underlying the relational model is the set-theoretic relation, which is a subset of the Cartesian product of a list of domains. A *domain* is a set of values. The *Cartesian product* of domains $D_1, D_2, \ldots, D_k$, written as $D_1 \times D_2 \times \cdots \times D_k$, is the set of all k-tuples $(v_1, v_2, \ldots, v_k)$ such that v_1 is in D_1, v_2 is in D_2, and so on.

A *relation* is any subset of the Cartesian product of one or more domains. It is always assumed finite. The members of a relation are called *tuples*. Each relation that is a subset of $D_1 \times D_2 \times \cdots \times D_k$ is said to have *arity* or *degree* k. A relation can be viewed as a table, in which each row is a tuple. Columns are called *attributes*. The set of attribute names for a relation is called the *relation scheme*. If we name a relation R, and its relation scheme has attributes a, b, c, and d, then the relation scheme is written as R(a, b, c, d). The attribute a of relation R is denoted as R.a. The order of both the tuples and the attributes are unimportant. A couple of examples are shown in Figure 1.5.

An operation to a database can be decomposed into two parts, the target-data identification and the target-data retrieval or update. The target-data identification usually needs a sequence of basic operations to be applied on the database when users query the database using high level query languages. There is no unique way to process such queries. Also, the processing overhead for different processing sequences may be quite different. Hence, a major task of database design is to design a strategy that can produce efficient processing sequences for all possible queries.

There are five basic operations that can manipulate a relational database.

(1) **Selection**. Selection selects those tuples in a relation that satisfy a given condition, and is denoted as $\sigma_F(R)$, where F is the condition and R is the relation to be operated.

(2) **Projection**. Projection takes unique values in an attribute of a relation, and is denoted as $\pi_A(R)$, where A is the attribute to be projected and R is the relation to be operated.

(3) **Join**. When two relations share a common attribute, they can be joined. The result of a join operation on a common attribute of two relations is the

AJAX, INC.	NEW YORK
BROWN & SONS	LOS ANGELES
CARLTON ENTERPRISES	CHICAGO

(a)

COMPANY

COMPANY NAME	CITY

(b)

AJAX, INC.	WIDGET	4.00
AJAX, INC.	GISMO	4.50
BROWN & SONS	DOJIGGER	5.00
(NULL)	WHATCHET	3.50

PRODUCT

COMPANY NAME	PRODUCT NAME	COST

Figure 1.5 An example of relational database (a) relations (b) relational schemas.

Cartesian product[*] of the two relations, with the tuples which share the same value in the common attributes. Only one common attribute is kept in the resulting joined relation. This type of join operations is also called equi-join. There are other types of join operations [ULL82].

(4) **Union**. The union of relations R and S is the set of tuples that are in R or S or both. It is denoted as R ∪ S.

(5) **Set difference**. The difference of relations R and S, denoted as R − S, is the set of tuples that are in R but not in S.

The examples of these operations are shown in Figure 1.6. These operations can be concatenated together to form more complicated queries. There is no unique way to form a complicated query to get the final result. Furthermore, different basic-operator sets may have equivalent manipulating power. That is, the result obtained by applying a sequence of operations belonging to an operator set can also be obtained by applying operations belonging to another operator set. For example, the join operation can be replaced by the Cartesian product operation, since a join operation can be replaced by a combination of Cartesian product, selection, and projection. This is shown in Figure 1.7. Notice

(a) that the join operation is a binary operation with two relations as operands,

(b) that the join operation is symmetric,

(c) that the size of a joined relation would be much larger than the the sizes of each individual relations, and

[*] **Cartesian product.** Let R and S be relations of arity k_1 and k_2, respectively. Then R × S, is the set of $(k_1 + k_2)$-tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S.

A	B	C
a	b	c
d	a	f
c	b	d

(a) Relation R

A	D	E
c	g	a
c	b	f
d	a	f

(b) Relation S

A	B	C
a	b	c
c	b	d

(c) $\sigma_{B=b}(R)$

B
a
b

(d) $\pi_B(R)$

A	B	C	D	E
d	a	f	a	f
c	b	d	g	a
c	b	d	b	f

(e) R join S

a	b	c
d	a	f
c	b	d
c	g	a
c	b	f

(f) R union S

a	b	c
c	b	d

(g) R-S

Figure 1.6 Examples of operations in relational databases.

R

A	B
1	b
2	b
3	c

R Join S

S

B	C
a	4
b	5
b	6

1. $R \times S$

A	B	B	C
1	b	a	4
1	b	b	5
1	b	b	6
2	b	a	4
2	b	b	5
2	b	b	6
3	c	a	4
3	c	b	5
3	c	b	6

2. Selection (R.B=S.B)

A	B	B	C
1	b	b	5
1	b	b	6
2	b	b	5
2	b	b	6

3. Projection A, B, C

A	B	C
1	b	5
1	b	6
2	b	5
2	b	6

Figure 1.7 An example of join operation.

(d) that there is a selection embedded in a join operation.

The most commonly used operations are selections, projections, and joins. Selection and projection are unary operations and generate relations with smaller size. A lot of overhead may be involved in a join operation since it usually expands the size of the joined relation. Further, in a DDBMS, to perform a join operation on two relations stored in two different locations may involve a lot of network activities. Therefore, to process join operations efficiently is the most critical task in the design of either a CDBMS or a DDBMS.

Most of the existing query languages are based on three primitive query languages: relational algebra, tuple relational calculus, and domain relational calculus [ULL82]. For simplicity, we consider the join queries only in this thesis, which are in the form of :

GET R.a WHERE (R.a = S.a) AND (R.b = T.b)

The result of the query consists attribute a only, which is projected from the result of joining relations R, S, and T together.

1.6 Objectives of This Research

This dissertation intends

(1) to study the key problems of distributed databases in local multiaccess networks (DDBLMN) including distribution design, query processing, and concurrency control, as well as

(2) to propose strategies for solving these problems in an integrated fashion, such that a DDBLMN supporting efficient execution of user transactions can be easily implemented.

We have mentioned that solving all the DDBMS problems together in an integrated fashion is very difficult. Further, there is no need to design the communication system with the DDBMS together. Therefore the local multiaccess network, which has nice characteristics that can simplify the design, is chosen as the supporting network. Then, the DDBMS is designed based on these characteristics.

To further simplify the problem, the distribution design problem is solved separately although it is related to the other two problems. Distribution design is done before a DDBMS is operational while query processing and concurrency control problems is to be solved dynamically. As long as the system characteristics do not change rapidly, this approach can lead to stable and efficient design.

The relationship between query processing and concurrency control problems is so strong that it is difficult to combine the solutions of solving each problem individually.

1.7 Overviews of Current Researches

Most current researches of DDBMS's in local area environments have attempted to solve one or more individual problems. Actually, most of them consider the query processing problem without considering the concurrency control problem.

Hevner, Wu, and Yao [HEV85]

This research investigates distributed query processing on *address ring* and *broadcast* networks. Only communications cost is concerned.

An address ring is really a token ring [STA84], which uses a token to control the ring-access right. A station wishing to transmit a message to one or more other stations should get the token first. The addressees copy the messages down to the stations by matching their addresses to the addresses associated with the massages. The average transmission cost for a message is taken as the average time for the sending site to gain control of the token ring and a term proportional to the data volume and the distance to the addressees.

A broadcast network under consideration can have either a bus or ring topology with various protocols. The average communication cost of sending a message on the network is the average time for the sending site to gain control of the network and a term proportional to the data volume.

Static query optimization algorithms, in which the processing strategy is determined before a query is processed and cannot be changed during processing, are proposed. For some restricted cases, these strategies may be capable of getting optimum solutions. These strategies can be extended into heuristic algorithms to find strategies for general queries.

Sacco [SAC84]

Sacco investigates the query processing problem on broadcast networks. Three heuristic algorithms are proposed. These algorithms can be implemented either statically or dynamically. However, no adequate approach for implementing dynamic query processing is shown.

LAMBDA Systems [CHA83] [CHA84a] [CHA85]

LAMBDA, a database system on a local multiaccess/broadcast network, is an experimental system designed at the AT&T Bell Laboratories, Murray Hill, for studying DDBMS issues in a local area network. It provides (1) high data availability through replicated data, (2) automatic recovery from site failures, and (3) uninterrupted transaction processing during site recovery. The design is simplified by assuming a reliable broadcast network. Algorithms for transaction commitment, concurrency control, and crash recovery are designed in LAMBDA.

Gouda and Dayal [GOU81]

Gouda and Dayal investigates the query processing problem in a local area network with one or more parallel broadcast busses. The database is assumed without data redundancy. This research pointed out that the query processing problem in this environment is $NP - hard$* and heuristic algorithms are needed to find good solutions to the query processing problem.

MICROBE Systems [NGU81a] [NGU81b]

MICROBE is a DDBMS on a network of LSI-11 micro-processors connected by a local broadcast network implemented at the Laboratoire IMAG in the University of Grenoble. A mix of static and dynamic query processing strategies is used in MICROBE. The primary optimization objective is to minimize local processing and communication costs.

* A problem is called NP-hard if there is no known algorithm that can solve the problem in a time polynomially proportional to the problem size.

LU [LU85]

Lu proposes a *load-balanced query processing* (LBQP) approach to solve the query processing problem using load balancing in a local DDBMS. The database is assumed to be fully replicated. A dynamic query processing strategy, which is capable of migrating a process to other sites during the query processing, is used in this research to enhance system load balancing and, consequently, reduces the response time.

UNITY Systems [KER82]

The UNITY system, developed at the AT&T Bell Laboratories, consists of a minicomputer and a set of microcomputers connected by a star network. Kershberg, Ting, and Yao presented several query optimization techniques that are specially designed for star networks.

Ceri, Paolini, Pelagatti and Schreiber [CER82]

Ceri et. al. were implementing a relational DDBMS on a star local area network in Italy. The major concern of this system is on the file partitioning problem (see Section 2.1.1). Conventional majority two-phase locking concurrency control protocol (see Chapter IV) is used for concurrency control.

DDBLMN

Similar to LAMBDA system, DDBLMN in this research is assumed supported by a reliable local multiaccess/broadcast network. However, file allocation problem, distributed query processing problem, and concurrency

control problem are solved together in DDBLMN. While all existing systems only focus on one particular problem. The performance of such systems may be quite different from the expectation when they are really implemented in a working environment. Further, the design of DDBLMN takes the advantages of supporting network such that simple but efficient control strategies can be developed to enhance the system performance. The overview of working environment of DDBLMN is shown in the Section 1.10.

1.8 Contributions of This Thesis

The contributions of this thesis are as follows.

1. *A new transaction processing model that is particularly suitable for DDBMS on local multiaccess networks.*

   The information exchange in general networks is inefficient such that a transaction is generally processed at the site it was entered. Either the needed information is gathered from other sites, or the entire transaction is migrated to related sites to be processed. These processing models do not take advantage of the distributed processing system, i.e., the parallelism is low. The proposed model allows all related sites to work together to carry out a transaction and to maintain a high degree of parallelism.

2. *Solution of the simple file allocation problem in local multiaccess networks in polynomial time.*

   In general networks, SFAP is an NP-hard problem. In local multiaccess networks, we have solved in polynomial time the standard SFAP and the SFAP with availability constraints. Although the problem is NP-hard when there is an average-delay constraint, it is isomorphic to the knapsack problem, which has good approximate solutions.

3. *Efficient solution of the general file allocation problem in local multiaccess networks.*

   The FAP with storage constraints in local multiaccess networks is NP-hard. An efficient algorithm combining a branch-and-bound search with the knapsack problem has been developed.

4. *Integration of distributed query processing and concurrency control.*

   Currently, distributed query processing and concurrence control are studied independently in general networks. The combined results may not be good for an integrated system. An integrated model that combines query processing optimization with concurrency control has been developed in this thesis.

5. *Generalization of the distributed extremum identification algorithm on local multiaccess networks to identify the sub-query to be processed.*

   Based on Wah and Juang's distributed extremum identification algorithm [WAH85b], which can identify the site with the extremum value efficiently, a general algorithm has been developed to synchronize transaction processings.

6. Realization of dynamic query processing on local multiaccess networks.

   The major obstacle to realize dynamic query processing, which performs better than static query processing, lies in the difficulty to collect global system status in real time. We have proposed an efficient dynamic query processing algorithm based on the extremum identification algorithm.

7. *Performance improvement by redundant materialization and non-profitable data identification.*

Redundant data are used in query processing to help reduce the size of intermediate results. The non-profitable data identification is included in the query processing algorithm to eliminate useless transmissions. These two methods work together to improve the performance of query processing.

8. *Transformation of the query processing problem in future computer networks into the generalized traveling salesman problem.*

   Current solutions for the query processing problem are not adequate for DDBMS in future computer networks, which are characterized by high-speed data links. We have modeled the problem as a generalized traveling salesman problem. An efficient algorithm that utilizes the existing solutions of the standard traveling salesman problem has been developed and analyzed.

9. *Development of a framework of intelligent query processing system.*

   Knowledge based on the history of the database operation can be used to generate better query processing strategies. This provides a good direction to improve the query processing strategies.

10. *Development of an efficient concurrency control protocol in local multiaccess networks.*

    Current concurrency control protocols are designed for general networks, which do not support efficient information exchange among local processing sites. Information exchanges in the local multiaccess networks are extremely efficient, and an efficient and highly concurrent protocol for DDBLMN has been developed.

1.9 Significance of This Research

The current trend in multi-user local-computer-system development tends to be the network of small computers. A number of interesting projects are in progress today on designing such systems. Among all applications in a local computer system, database management is definitely very important. As a consequence, the design of a robust, easy-to-use and cost-effective DDBMS in a local area environment is becoming an important issue. Although DDBMS on *wide area networks* have been studied extensively, their results cannot be applied directly for the following reasons. First, the characteristics of local area networks are quite different from global networks. Second, the users' behavior and users' requirements are different, e.g. the mobility of users in such an environment may be much higher. Lastly, the combination of solutions derived from individual problems in global networks may not perform well in an integrated local system. In short, a DDBMS study in an integrated fashion is needed for the local area environment.

Although the research in this thesis does not solve all the problems of DDBMS, we have studied three of the most important problems, namely, file allocation problem, distributed query processing problem, and concurrency control problem, in an integrated fashion. The feasibility and the effectiveness of the design are demonstrated by simulation studies. Based on this research, a prototype can be implemented easily. This research also provides a framework for DDBMS design in a local environment.

1.10 System Model

The DDBLMN to be designed is based on the environment described in this section.

1.10.1 System Overview

From the users' point of view, the system is nothing but an integrated relational database consisting of relations as shown in Figure 1.8(a). Physically, it's a set of local database systems interconnected by a multiaccess/broadcast communication bus as shown in Figure 1.8(b).

1.10.2 Logical Database

It is assumed that a very large relational database with a global conceptual schema is available to the users. Virtually, users can access the database from different locations through the same conceptual schema. Therefore, we assume that the probability of a user accessing the database from any location is non-zero and is not uniformly distributed. There are some locations that a user may access more frequently than other sites. These two assumptions are essential to the distributed database. With a strong locality of access, there is no need to design an integrated database. On the other hand, a random access pattern is also unrealistic. Each relation is stored as a file and is the unit to be locked, i.e. a relation is either available or not available to a user, without partial availability. The statistics about the frequencies of access and update is only available on a relation basis. The data directory is fully distributed over all sites. The directory is maintained in the same way as a relation. Using fully replicated directories simplifies the system design dramatically. The additional storage is not significant since the directory is much smaller than the database itself.

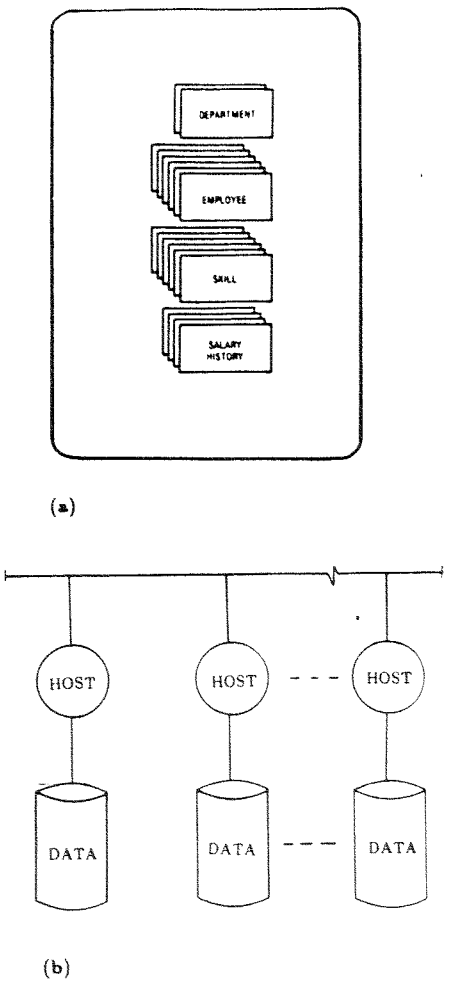


Figure 1.8 DDBLMN (a) from user's point of view (b) in the physical model.

Unless the database is reorganized very frequently, the update frequency to the directory is much less than the retrieval frequency. Moreover, the updating cost is independent of the number of copies in a network with broadcast capability. Therefore, the additional overhead of directory maintenance is also insignificant.

1.10.3 Transactions

The users activities to the system are called *transactions*. Each transaction is atomic, which should be either completely finished or completely aborted leaving the system unchanged. All transactions are independent of each other. We further assume that each transaction can not be aborted by the user. Therefore, a transaction is always terminated successfully in a reliable system unless the concurrency control protocol aborts the transaction. A transaction may consist of a set of independent queries. Each query consists of two parts: the specification of the target-data and the operations on the target-data. The specification is in the form of consecutive equi-joins. An example is shown in the following.

GET	R.a	WHERE (R.a = S.a) AND (R.b = T.b)
UPDATE	S.a	WHERE (S.a = T.a) AND (R.b = S.b)
GET	T.a	WHERE (T.a = S.a)

Notice that the values to be updated are not given here since it is not important.

1.10.4 System Architecture

As shown in Figure 1.8.(a), the DDBLMN consists of a set of local systems connected by a local multiaccess bus with the broadcast capability. A local database system is accomplished by each local system. The transmission system is assumed reliable, which simplifies the system design [CHA84a] [CHA84b]. The

transmission system provides guaranteed delivery of messages. A distributed multiaccess protocol, such as the CSMA/CD protocol, is assumed to control the bus accesses. In general, each station has equal priority to access the bus. However, the performance can be enhenced if priority messages are allowed. The end-to-end communication speed of the bus is about 1 Mbps (million bits per second). The number of local systems is around twenty to several hundred, each one with about 1 MIPS (million instructions per second) processing capability. The transmission cost, which can be either real cost or the elapsed time depending on the system objectives, is proportional to the volume of transmitted data. Under these assumptions, the processing capability per station is much larger than the communication bandwidth per station. Consequently, the local processing cost can be ignored.

1.10.5 Architecture of Local Systems

DDBLMN is not designed for any system with a specific local architecture. However, a local system architecture described here serves as an example. The logical structure of the local system at each site is shown in Figure 1.9. Each system consists of five different modules: Transaction Monitor, Transaction Handlers, File-Server subsystem, Concurrency-Control subsystem, and Network-Interface subsystem.

(1) **Transaction Monitor** (TM): This module coordinates all local and remote transactions by

   (a) initiating/terminating Transaction Handlers that handle each individual transaction, and

   (b) providing interface functions among all other modules.

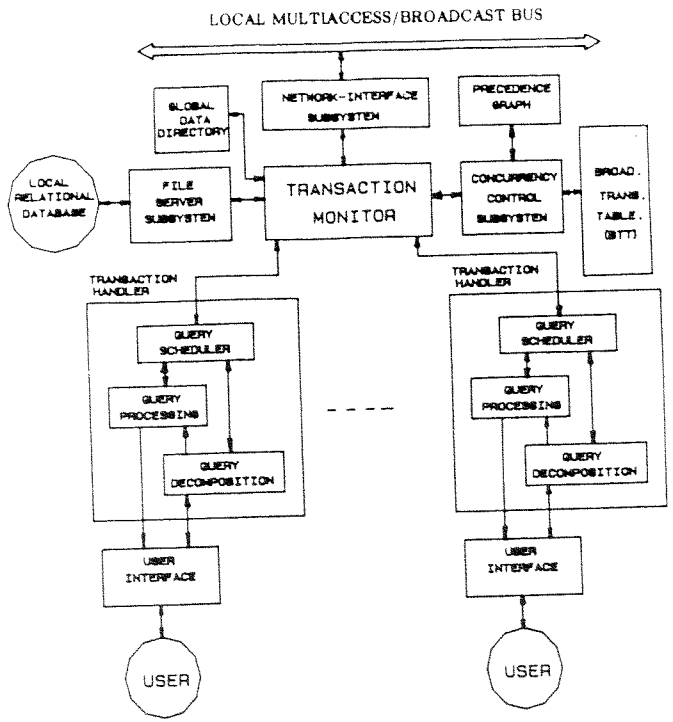


Figure 1.9 Logical architecture of local sites.

It also maintains a global directory and provides access transparency to users.

(2) **Transaction Handler** (TH): This module is created by the Transaction Monitor for handling each individual transaction. The TH can be initiated either locally or remotely. All the transaction processing functions, such as query decomposition, user interface, query processing, and query scheduling for a transaction, are performed in the TH designated for the transaction. It is eliminated when this transaction terminated.

(3) **File-Server Subsystem**: This module, which serves as a local DBMS, is responsible for accessing and updating the local database. A TH makes relation access/update requests through the TM to the file-server subsystem.

(4) **Concurrency-Control Subsystem**: This module monitors the actions of all local and remote transactions on the network through the Network-Interface Subsystem. All necessary status information is also maintained here. All the local data access/update requests should be checked by this module to maintain data integrity.

(5) **Network-Interface Subsystem**: This module, which performs all the interface functions to the communication network, provides guaranteed end-to-end message delivery. The network interface functions include addressing, network access control, flow control, error control, message receiving, format conversion, protocol control, sequencing, synchronization, fragmentation, and reassembling.

The detailed relationship between TM and the TH's is discussed in next section.

1.10.6 Transaction Processing

Users of a DDBMS interact with the database by executing transactions. A transaction, which consists of a sequence of operations (queries) on one or more database objects, transforms a current consistent database state into a new consistent state. Query processing and concurrency control are two key tasks in transaction processing. In this section, we discuss how a transaction is processed.

1.10.6.1 Transactions and Processes

A process is a program initiated by the system for handling a particular task. A process is called *active* if it is currently executed by the system. In a concurrent environment, more than one process can exist in the system at the same time although they may not be all active. One processor can only execute one process at any time. If the number of processors in a system is less than the number of processes to be executed, then either a time sharing or queuing system is needed for processor sharing. The concept of processes helps to simplify the design of a complicated system. Virtually, a system can initiate a separate process to handle a particular request. A process, hence, needs only to handle a single job. Instead of considering all the job requests together in a single complicated program, system design is decomposed into two parts: the design of each individual process which handles a given job, and the management of processes. The design process is therefore greatly simplified. In DDBMS's, a transaction may involve more than one local system and hence system design is much more difficult. It is better to handle the transactions using the concept of processes.

There are three different approaches to handle transactions in a DDBMS. In Bernstein's model [BER81b], as shown in Figure 1.10, a Transaction Manager (TM) and a Data Manager (DM) reside in each site. A TM handles all the transactions entering the site and a DM handles all the queries issued by either the local or remote TM's. Obviously, the design of TM's and DM's in this approach is not easy.

In another approach, a process is created for each data access/update request entering the site either locally or remotely [BIR82] [STU80]. In this approach, creating and initializing a process to handle each remote request require too much overhead. Further, the sequence of requests within a transaction must somehow be tied together and kept in order. With respect to data access authorization, each request must be individually authenticated, which is an expensive operation.

In the third approach, used in R^* [LIN84], a process is created for a transaction and retained for subsequent use for the duration of the transaction session. There is only one setup cost for each transaction. More importantly, keeping track of the sequence of requests within a transaction is no longer a complicated job. The transactions in this research are processed in a way similar to System R^*'s approach.

1.10.6.2 Transactions and Processes in DDBLMN

Refer to the system model shown in Figure 1.9, a transaction entering the system through a local site (called the *home site* of the transaction) is processed by a TH initiated by the TM in the home site. TH may then ask the TM's in remote sites to initiate TH's to process the transaction together. All these TH's

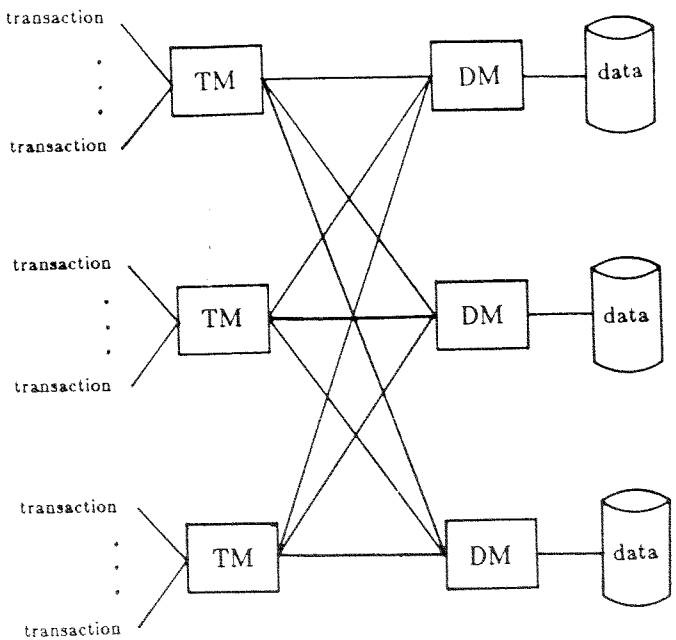


Figure 1.10 Bernstein's transaction processing model.

belong to the same transaction and are retained until the transaction is terminated. In this way, a transaction is carried out by a set of TH's cooperatively as shown in Figure 1.11.

In a DDBMS, a transaction cannot be processed independent of other transactions. In addition to the TH's, other processes are needed to coordinate the activities among the transactions. The Concurrency-Control subsystem in our model is the process in each site to handle concurrency control. Each data access or update request generated in a site is sent to the Concurrency-Control subsystem in the site, which will in turn check the consistency between the request and the current database state. If there is no consistency problem, the request will be forwarded to the local File-Server subsystem, which is another process handling the local database, and is then processed there.

1.10.6.3 Transaction Processing Strategies in DDBLMN

A transaction consists of a sequence of independent queries. After entering the system, a transaction is first processed by the home site in a phase called the *preprocessing phase*. All necessary initializations are performed in this phase. The transaction is then broadcast to other sites for further processing. Lastly, each individual query is processed in sequence by the system. The processing sequence of a transaction with three queries is shown in Figure 1.12. The preprocessing, initial-broadcast, and query processing phase will be discussed in Chapter III.

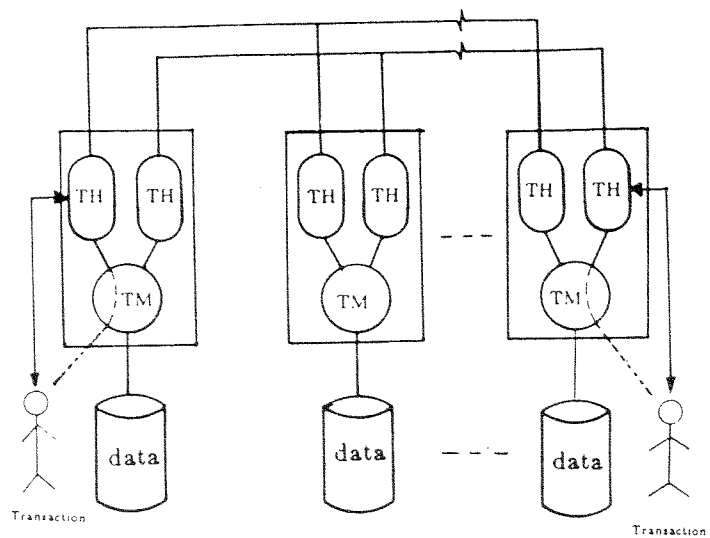
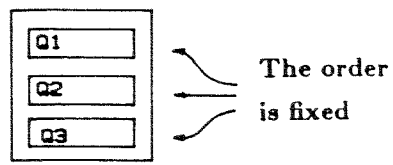


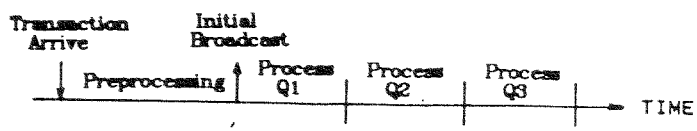
Figure 1.11 Example of two transactions being processed by the system.

A Transaction

```
GET      R.a  WHERE  (R.a = S.a) AND (R.b = T.b)
UPDATE   S.a  WHERE  (S.a = T.a) AND (R.b = S.b)
GET      T.a  WHERE  (T.a = S.a)
```



(a)



(b)

Figure 1.12 The processing sequence of a transaction with three queries Q_1, Q_2, Q_3 (a) the transaction; (b) the processing sequence.

1.11 Thesis Organization

In Chapter II, the distribution design problems on DDBLMN are analyzed. A detailed cost model of file allocation problem, which is a special case of the general distribution design problem, is developed. Different cases of the problem are analyzed.. Both solutions for some solvable problems and the proof for the other NP-hard problems are presented. Efficient approximation solutions for NP-hard problems are also developed.

Distributed query processing is analyzed in Chapter III, while concurrency control problem is studied in Chapter IV. A processing strategy consisting of five different phases are proposed, and various strategies for each phase are developed. A locking based concurrency control protocol for local multiaccess networks is also presented.

Under certain conditions, the query processing problem can be solved as a generalized traveling salesman problem. In Chapter V, an extensive analysis of the generalized traveling-salesman problem is presented.

In Chapter VI, simulation results to demonstrate the feasibility of the proposed DDBMS system are shown. Finally in Chapter VII, the summary of the research is given and the directions for future research are suggested.

CHAPTER II

FILE ALLOCATION

2.1 Introduction

The distribution design problem entails

(1) the fragmentation of the database and

(2) the allocation of these fragments with possibly redundant copies.

The objective of data distribution is to maximize the efficiency or performance of database operations subject to the system constraints. The performance can be measured by various parameters such as the overall operation cost, response time, and throughput. The constraints can be the average response time, availability, degree of redundancy, etc. The operation cost may consist of the communication cost, storage cost, and processing cost. Although the problem depends on the transaction processing strategies, solving the two problems together is very complex. In our design methodology, the distribution-design problem is solved independently from the transaction-processing problem in the database design phase.

2.1.1 Fragmentation

Fragmentation partitions a relation into several fragments, each of which has some special properties. There are two reasons that make fragmentation necessary.

(1) A relation may be too large to be stored in a single place. A large relation causes not only the storage problem but also the processing problem. For example, the buffer space in a system is limited such that there is no way to bring entire relation into the buffer to process. A considerable overhead may be induced to solve this problem.

(2) Most of the data retrievals from a certain class of users may be targeted to a small portion of the database most of the time. That is, there may exist a strong locality within a relation. A lot of the processing costs can be saved if the relations can be further partitioned such that a certain class of users may need to access a certain fragment of each relation. This may reduce processing overhead, as only the needed fragments are searched. Fragmentation can be viewed as a 'pre-selection,' which can reduce the overhead of future query operations.

As an example, a relation storing the personal information of all students in a university contains 40,000 tuples. To simplify the user interface, this information is organized as a single relation. However, most of the queries initiated from a department will only involve the portion containing information about students in the department. It is natural to decompose the relation such that each fragment only contains the information about students in a given department. Not only the processing cost is saved, but the communication cost is also saved if each fragment can be stored in the local computer of each department. An example illustrating fragmentation is shown in Figure 2.1.

There are two conditions that a fragmentation should satisfy.

(1) **Completeness.** All data of the original relation should be mapped into fragments; i.e., a data item in the original relation must exist in at least one fragment.

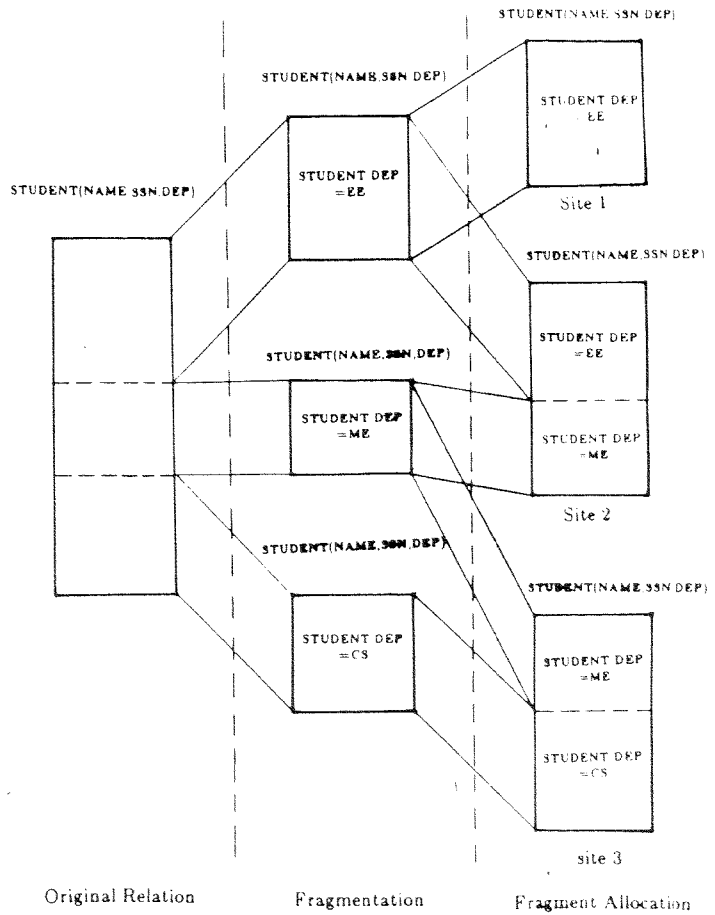


Figure 2.1 Example to illustrate the fragmentation problem

(2) **Recoverability**. It should be possible to reconstruct the original relation from the fragments.

There are three basic types of fragmentations: horizontal, vertical, and mixed.

(1) A **horizontal fragmentation** partitions the relations by tuples, in which all tuples satisfying a given condition are grouped together into a fragment. This can be done by the *selection* operation. The original relation can be recovered by the *union* operations.

(2) A **vertical fragmentation** partitions the relations by attributes, in which a number of attributes are grouped together into fragments. Vertical fragmentation can be done by the *projection* operations. The original relation can be recovered by the *join* operations.

(3) A **mixed fragmentation** is the combination of horizontal and vertical fragmentations. A horizontal fragmentation may be followed by a vertical fragmentation or vice versa.

The fragmentation problem in DDBMS has been studied extensively by Ceri, Navathe, and Wiederhold [CER84]. We will not consider the fragmentation problem in this research. It is assumed that fragmentation is done by existing methodologies and is separated from other design problems. Consequently, it is reasonable to assume that each data fragment is a relation by itself.

2.1.2 Fragment Allocation and File Allocation Problem

The next problem after fragmentation is to allocate and place these fragments (or relations) in the system. The allocation should be based on the way that the database is queried by the users. It is impossible for a system to collect all the access pattern down to the tuple level. We have assumed that this information is only available on the relation level. Therefore, the fragment allocation can be solved as a *file allocation problem* (FAP). The FAP entails the distribution of files (or relations) on a DDBMS such that the objectives of the system are satisfied and that the overall system is operating at its most efficient level. If the constraints, requirements, query patterns, or even the database state are changing dynamically, the files must be allowed to migrate. The *file migration problem* (FMP) entails the study of the migration sequence of files to adapt to changes in the DDBMS [WAH84a]. Solutions developed for a general network can also be applied to a local multi access network. In this research, we will only consider the FAP as more efficient solutions can be obtained.

The FAP was originally investigated by Chu [CHU69] who studied it with respect to multiple files on a multi-processor system. He modeled the problem as an integer programming problem in which the objective is to minimize the overall operating costs under the constraints of response time and storage capacity. Subsequently, a lot of work has been done in this area. There are good comparative surveys in Dowdy's [DOW82] and Wah's [WAH84a] paper.

To allocate a copy of a file to a site, it has the following effects on the overall operating cost and constraints.

(1) The communication cost of remote accesses initiated from this site is saved. It may also provide lower access cost for other nearby sites.

(2) The storage cost, which is assumed to be proportional to the data volume, in this site is increased.

(3) The communication cost to update this file may be increased.

(4) The storage space in this location is reduced.

(5) The average response time may be reduced since some remote accesses are localized.

(6) The availability of this file is increased.

In general, the decision to allocate a specific file to a specific location cannot be made independent of the allocation of other files. The problem is formulated into an integer programming problem with the overall operating cost as the objective to be minimized. The problem is NP-hard on general networks. However, the problem is simplified on local multiaccess networks and is polynomially solvable in some special cases. There are two reasons why the problem is simplified.

(1) The remote access/update costs on the network is site independent.

(2) The update cost for a file is independent of the number of copies allocated if more than one copy is allocated.

In this dissertation, the different cases of the *simple file allocation problem* (SFAP), which is a special case of FAP, and the general file allocation problem with storage constraints are studied and solved. The following notation are defined.

N - number of sites in the system;

F - set of files in the database, $|F| = m$;

I^f - index set of sites with a copy of file f;

λ_j^f - query load originating at site j for file f per unit time;

ϕ_j^f - update load originating at site j for file f per unit time;

d - cost per unit size of query from any site to any other site;

d' - cost per unit size of update from any site to all other sites;

σ_k^f - storage cost per unit time of file f at site k;

L^f - length of file f;

S_k - storage capacity at site k;

$$Y_k^f = \begin{cases} 1 & \text{if file } f \text{ exists at site } k \\ 0 & \text{otherwise} \end{cases}$$

2.2 Simple File Allocation

The SFAP considers the allocation of a single file, in which the effects of queries, updates, and data storage are represented as costs [RAM83]. It has been shown to be NP-hard [ESW74] [GAR79], which means that the computation time for all known optimal algorithms for this class of problems increases at least exponentially with the problem size. Numerous exhaustive optimal algorithms and suboptimal heuristics have been studied [WAH84a] [DOW82].

In this section, a solution to the SFAP for local multiaccess networks is proposed. Since only one broadcast is required to either lock, unlock, or update a file, the problem is found to be polynomially solvable with respect to the number of sites.

In the following cost function for a given placement I, the index f is dropped because the problem is defined with respect to a single file.

$$C(I) = \sum_{j=1}^{N} \lambda_j d(1 - Y_j) + \sum_{j=1}^{N} \phi_j d' \left(1 - Y_j \prod_{\substack{k=1 \\ k \neq j}}^{N} (1 - Y_k) \right) + \sum_{k=1}^{N} \sigma_k Y_k \quad (2.1)$$

The first and last terms on the RHS of the above equation represent the query and storage costs, respectively. The second term accounts for the update costs.

When multiple copies of a file exist in the system, each unit of update incurs a constant cost of d'. However, when only one copy of the file exists, updates originating from the site with the copy do not incur any cost because they do not have to be broadcast.

This problem can be formulated as an integer program with Eq. (2.1) as the cost function and constraints that at least one copy exists and Y_k is either 0 or 1. Three different cases are considered in this section: the conventional SFAP, SFAP with availability constraints, and SFAP with average delay-time constraints.

2.2.1 Conventional SFAP

The solution to this problem can be simplified by considering the single-copy and the multi-copy cases separately.

For the single-copy case, the cost function becomes

$$C(I) = \sum_{j=1}^{N} [(\lambda_j d + \phi_j d')(1 - Y_j) + \sigma_j Y_j] \qquad (2.2)$$

$$= \sum_{j=1}^{N} (\lambda_j d + \phi_j d') + \sum_{j=1}^{N} (\sigma_j - \lambda_j d - \phi_j d') Y_j$$

In addition to the fixed cost in Eq. (2.2), the cost of not placing the file at site j is $\lambda_j d + \phi_j d'$, while the cost of placing the file there is σ_j. Thus the file should be placed at site j if the cost difference $(\sigma_j - \lambda_j d - \phi_j d')$ on this site is the minimum. The optimal allocation can, thus, be found in O(N) time.

For the multi-copy case, at least two copies must be allocated. The cost function is

$$C(I) = \sum_{j=1}^{N} [\lambda_j d(1 - Y_j) + \phi_j d'] + \sum_{k=1}^{N} \sigma_k Y_k \qquad (2.3)$$

$$= \sum_{j=1}^{N} (\lambda_j d + \phi_j d') + \sum_{j=1}^{N} (\sigma_j - \lambda_j d) Y_j .$$

The first term in the above equation is constant. The optimal allocation can be found by allocating a copy at site j if $(\sigma_j - \lambda_j d)$ is negative. This will decrease the overall cost, C(I). In case that none or one of the cost differences is negative, two copies with the minimum cost differences are selected. The optimal allocation can hence be found in O(N) time.

The global optimum is obtained by comparing the costs under the single-copy and multi-copy cases. Therefore, the SFAP is optimally solvable in polynomial time. As an example, consider a system with four nodes, query rates λ_i = [6, 7, 4, 5], update rates ϕ_i = [3, 6, 2, 4], and storage costs σ_i = [3, 5, 2, 5]. The per unit query and update costs, d and d', are assumed to be unity. In the single-copy case, $[\sigma_i - \lambda_i d - \phi_i d'] = [-6, -8, -4, -4]$. The single copy should be allocated at site 2 with cost 29. In the multi-copy case, $[\sigma_i - \lambda_i d] = [-3, -2, -2, 0]$. Copies should be allocated to sites 1, 2, and 3 with a total cost 30. Comparing the single-copy and multi-copy cases, a single copy should be placed at site 2.

2.2.2 SFAP With Availability Constraints

The SFAP can also be solved efficiently when an availability constraint is included. As mentioned in Chapter I, the availability is defined as the probability that at least one copy of a file is available. Since the network is reliable and the failure rates of sites are assumed to be identical, the availability constraint can be expressed into a minimum number of copies of the file. The locations for these copies can be determined as above.

There is no difference between the the SFAP with an availability constraint and the conventional SFAP if the minimum number of copies of the file is one. If the number of copies should be more than two, then only the multi-copy case needs to be considered.

A copy of the file is allocated at each site with negative $(\sigma_j - \lambda_j d)$. Additional copies are allocated such that the availability constraint is met and the cost increase is minimum. The complexity of this algorithm is also O(N).

2.2.3 SFAP With Average-delay Constraints

The SFAP becomes NP-hard when an average-delay constraint is imposed. Given that the average delay of messages on the multiaccess bus is a constant, c, the average delay for all queries and updates must satisfy:

$$\sum_{j=1}^{N} \left[\lambda_j c(1 - Y_j) + \phi_j c\right] \le D' \quad or \quad \sum_{j=1}^{N} \lambda_j c(1 - Y_j) \le D \tag{2.4}$$

In Eq. (2.4), updates are assumed to be always broadcast. The SFAP with the above average delay constraint can also be separated into the single-copy and multi-copy cases. The single-copy case is solvable in O(N) time. For the multi-copy case, the cost formula in Eq. (2.3) can be rewritten as

$$C(I) = \sum_{j=1}^{N} (\phi_j d' + \sigma_j) + \sum_{j=1}^{N} (\lambda_j d - \sigma_j)(1 - Y_j).$$

Since the first term on the RHS is constant, the optimization problem for the multi-copy case can be written as

$$maximize \quad C(I) = \sum_{j=1}^{N} (\sigma_j - \lambda_j d)(1 - Y_j). \tag{2.5}$$

$$subject\ to \sum_{j=1}^{N} \lambda_j (1 - Y_j) \le \frac{D}{c} \quad and$$

$$1 - Y_j = 0 \text{ or } 1$$

This problem is reducible from the 0-1 knapsack *problem** with N objects by the variable transformation $Z_j = 1 - Y_j$, where the profit and weight of the j^{th} object are $(\sigma_j - \lambda_j d)$ and λ_j, respectively. However, the profits may be negative for some of the objects, and they are excluded from consideration in the optimization. The problem can be solved by dynamic programming algorithms or fully polynomial-time approximation schemes.

2.3 General FAP With Storage-Capacity Constraints

The general FAP considers the allocation of multiple files under such design requirements as delay, storage capacity, parallelism, and availability. As a simplification, each query is assumed to access a single file, and all file accesses are assumed to be independent. Only the storage capacities of sites are considered in the following formulation.

The problem with storage-capacity constraints can be formulated as:

$$minimize \quad C(I^1, \ldots, I^{|F|}) = \sum_{f,j} \lambda_j^f d(1 - Y_j^f) +$$

$$\sum_{f,j} \phi_j^f d' \left(1 - Y_j^f \prod_{k \ne j} (1 - Y_k^f)\right) + \sum_{f,j} \sigma_j^f Y_j^f \tag{2.6}$$

subject to:

* Given a finite set U of m objects and a knapsack of size B, there are a size $s(u) \in Z^+$ and a profit $p(u) \in Z^+$ for each $u \in U$, where Z^+ is the set of positive real numbers. The knapsack problem searches for a subset $U' \subseteq U$ that maximizes $\sum_{u \in U'} p(u)$ such that $\sum_{u \in U'} s(u) \le B$.

(a) $Y_j^f = 0$ or 1, $\quad j = 1, \ldots, N, \ f \in F$

(each site contains at most one copy of each file);

(b) $\sum_{j=1}^{N} Y_j^f \geq 1, \quad f \in F$

(at least one copy of each file exists in the system);

(c) $\sum_{f \in F} L^f Y_j^f \leq S_j, \quad j = 1, \ldots, N$

(the storage capacity at each site is not exceeded).

The above problem is non-linear, but can be linearized easily by using different index variables [GEO72]. This problem is NP-hard as shown in the following theorem.

Theorem 2.1: The optimization problem defined by Eq. (2.6) is NP-hard.

Proof: We show that the 0-1 knapsack problem reduces to this problem in polynomial time. Given an instance of the 0-1 knapsack problem, an instance of the problem in Eq. (2.6) can be formed with the following parameters: $N = 2$; $F = U$; $L^f = s(f)$ for $f \in F$; $S_1 = \sum_{f \in F} L^f$; $S_2 = B$; $\sigma_1^f = 0$ for $f \in F$; λ_1^f = a large constant for $f \in F$; ϕ_1^f, λ_2^f, and σ_2^f are chosen such that $\lambda_2^f d - \phi_1^f d' - \sigma_2^f = p(f)$ for $f \in F$. Basically, the first site has large enough capacity to hold a copy of all files in F. The query and storage costs are chosen such that a copy of all files are allocated at site 1. Therefore, the problem becomes the packing of the second site, which is the standard knapsack problem. □

2.3.1 Optimal Solutions of FAP With Storage Constraints

Although the problem in Eq. (2.6) resembles the packing of $m*n$ copies into m knapsacks, there is a subtle difference here. The profit of allocating the first copy of file f to site j is $(\lambda_j^f d + \phi_j^f d' - \sigma_j^f)$, while the profit of allocating the second copy of file f to site k, $k \neq j$, is $(\lambda_k^f d - \phi_j^f d' - \sigma_k^f)$. The extra term $\phi_j^f d'$ in the profit of the second copy offsets the additional profit incurred when the first copy is allocated. The profit of allocating other copies of file f to site q, $q \neq j$ and $q \neq k$, is $(\lambda_q^f d - \sigma_q^f)$. Note that these profits may be negative. Due to the above and since one or more copies must exist in the system, the problem is decomposable into multiple independent knapsack problems only if at least two copies have been allocated in the system. The overall profit can then be maximized by independently optimizing the profit of each site with the remaining capacity. An optimal algorithm, therefore, consists of enumerating the allocations of the first two copies of each file, and solving N knapsack problems for each combination. The structure of the state-space tree is as follows.

(1) The root is in level 0.

(2) In the first m levels, the first copy of all files are allocated. Level f, $1 \leq f \leq m$, represents the allocation of file f. Each file can be allocated to one of the N sites. Thus the degree of the tree in the first m levels is N. If a file cannot fit in a site, the subtree representing this allocation is considered infeasible and terminated.

(3) The allocation of the second copy of each file is carried out in the $(m+1)$'st to the $2m$'th levels of the tree. Since the second copy must not be allocated to the site containing the first copy and may not have to be allocated in the DDBMS, the degree of branching in each level is N. If a copy cannot fit in a

site, then the subtree representing this allocation is considered infeasible and terminated. Further, if an allocation of the second copy at site k results in negative value of $(\lambda_k^f d - \sigma_k^f)$, then the second copy should not be allocated at site k, and the corresponding subtree is also terminated.

(4) For all active nodes left in the state-space tree in level $2m+1$, the allocation of the remaining copies is solved as N single-knapsack problems. For a particular site, the files to be considered for packing into the remaining capacity are those that have not been allocated at this site and those with the second copy allocated in the system. The algorithm for solving the conventional 0-1 knapsack problem has to be modified to take into account the negative profits.

Bounding criteria similar to those used in branch-and-bound algorithms can be developed here [GEO72]. In particular, an upper bound on the profit can be computed for each node in the state-space tree by a linear program or a greedy algorithm without the integrality constraints. The maximum profit of the currently available feasible solutions is kept in the incumbent. If the upper bound of a node is smaller than the incumbent, then the subtree originating from this node is pruned because it cannot possibly lead to an optimal solution. An example illustrating the allocation of two files in a system with two sites using a depth-first search is shown in Figure 2.2. In computing the upper bound for a node in the branch-and-bound tree, each site is treated as an independent knapsack. Given that i, i=0, 1, 2, copies of a file to be included in the knapsack have been allocated in ascendant levels of this node, the profit of the file to be used is that of the $(i+1)$'th copy. A greedy algorithm computes the upper bound by assigning the files to each knapsack in descending ratios of profit to length until the knapsack is full. To keep the knapsack full, the last file may be

Cost of query, update, and storage				
	file 1		file 2	
	site 1	site 2	site 1	site 2
$\lambda_j^f d$	8	9	6	7
$\phi_j^f d'$	1	1	1	1
σ_k^f	4	6	4	6

Storage limit of sites	
site 1	site 2
5	6

Length of files	
file 1	file 2
3	2

Profits of first and second copies				
	file 1		file 2	
	site 1	site 2	site 1	site 2
First copy	5	4	3	2
Second copy	3	2	1	0

Allocations:

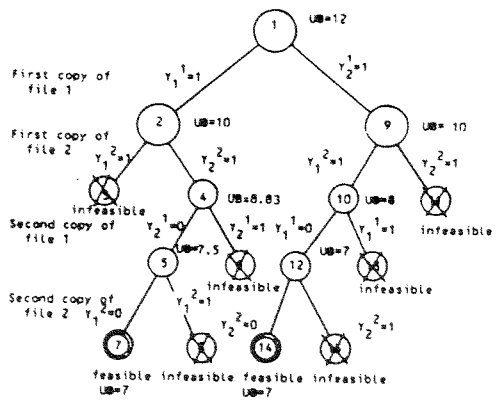


Figure 2.2 Example to illustrate the general file allocation problem

partially allocated. The worst-case number of knapsack problems to be solved in this approach is N^{2m}, which is much better than an exhaustive enumeration.

2.3.2 Heuristic Solutions of FAP With Storage Constraints

When the problem size is large, an optimal solution may not be possible to get in a reasonable amount of time. More efficient approximate solutions are needed.

The algorithm in Section 2.3.1 can be easily modified to get good approximate solutions. The knapsack problems in the algorithm can be solved heuristically, instead of optimally. Although the search method in Section 2.3.1 requires a total of N^{2m} knapsack problems to be solved, it is still a useful algorithm because the FAP can be solved in design time instead of in real time.

2.4 Summary

In summary, FAP on local multiaccess networks is simplified by the broadcast capability and the property that the communication costs are site independent. The solution algorithms for the SFAP are efficient and can be evaluated in either polynomial or pseudo-polynomial time. These algorithms can be applied in real time when the access rates change. However, optimal algorithms for the general FAP are complex and should be used in the design stage only. The algorithm can be easily modified into a less complex heuristic algorithm for large problems.

CHAPTER III

DISTRIBUTED QUERY PROCESSING

As mentioned in Section 1.10, a transaction is processed in a strategy that consists of preprocessing phase, initial broadcast, and query processing phase. The preprocessing phase and initial broadcast are discussed in Section 3.1. The query processing phase is discussed in the rests of this chapter.

3.1 Preprocessing Phase and Initial Broadcast

Preprocessing Phase

The operations which need to be performed in this phase are :

(1) **Administration**. These include authorization, accounting, etc.

(2) **Query decomposition and transformation**. Users of a database usually query the database using a high level 'user friendly' query language. Direct execution of queries presented in the original form is usually inefficient. A decomposition and a transformation of the given query may result in more efficient execution. For example, in target-data identification, a sequence of basic operations are applied to the database to identify the needed data. The size of the intermediate results may be different in different processing sequences. It is generally more efficient to apply unary operations, such as selections and projections, as early as possible.

(3) **Related-site identification and materialization.** *Related sites* of a transaction are the sites participating in the transaction processing. Materialization identifies the specific copies of relations needed by a transaction. A relation in a site is *materialized* if it is identified in the materialization. The related sites of a transaction in DDBLMN are the sites where at least one materialized relation is stored.

Initial Broadcast

After preprocessing, the transformed transaction with the necessary information is broadcast to all related sites. This is called *initial broadcast*. Each related site, then, initiates a TH to participate in the transaction processing.

Since the network is using a contention based bus access protocol, it is remotely possible that a transaction can never perform its initial broadcast. Further, a transaction is not known to other related sites until its initial broadcast has been carried out. Hence if the network can support priority messages, initial broadcasts should be given higher priorities for the transaction.

3.2 Introduction to Distributed Query Processing Problem

Minimizing processing cost is the most important goal of query processing. A large amount of communication traffic may be involved in distributed query processing (DQP). The communication cost is usually proportional to the volume of transmitted data in most DDBMS's, including DDBLMN. Hence the objective of query processing in these systems is to reduce the total data volume to be transmitted.

The main difficulty of query processing is the target data identification, which involves the identification of the set of physical fragments (relations) referenced by the query (*materialization*), the selection of the order of execution of operations, and the selection of the method for executing each operation.

The *raw-target* in the execution of a target-data identification is defined as the smallest portion of the database that contains the target-data. The procedure of target-data identification is like applying a sequence of 'cuts' on the 'raw-target' to get the target-data. Initially, the raw-target is the entire database. After applying an operation, the portion that is known not to contain the target-data is cut off. After a sequence of cuts, the target-data is finally identified. This procedure in DDBMS is known as *reduction* and is shown in Figure 3.1. One difference between the procedure in the example and the real DQP is that the size of a raw-target may be increased after a join operation. Obviously, the query processing cost is a function of the total volume of raw-targets in all the intermediate steps. The selection of the order of reductions becomes the major task of DQP.

3.2.1 Redundant Materialization

In most of the previous studies, the materialization is assumed solved independently and is non-redundant, in which one copy of each relation is identified in processing the query. A possible alternative is the redundant materialization, in which more than one copy of each relation may be identified. The redundant materialization has the following advantages over the non-redundant materialization.

(1) Non-redundant materialization is a special case of redundant materialization. For each schedule obtained from a system using non-

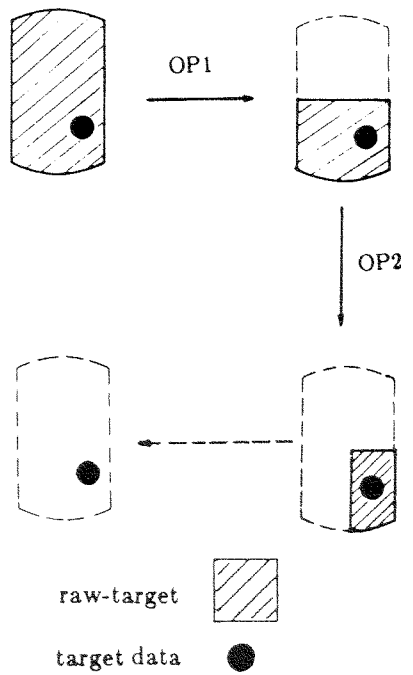


Figure 3.1 An Example to illustrate the procedure of target data identification.

redundant materialization, a schedule with less or equal cost can always be found from a system using redundant materialization.

(2) Redundant materialization is more flexible. In a DDBMS, a local database may have to answer queries initiated from more than one transaction at the same time. A transaction has to wait if the local database is busy. With redundant materialization, the availability of data is higher.

(3) Redundant materialization may have less query processing cost in a heuristic query processing. Whether a piece of the raw-target contains the target-data or not usually depends on the relationship among different components of the raw-target. For example, in the following query

GET R.a WHERE (R.a = S.a) AND (R.a = T.a),

the target-data is a portion of R.a and the initial raw-target consists of three components R, S, and T. They can be easily reduced into R.a, S.a, and T.a in the beginning. To perform further reduction, the relationship between R.a and S.a, and the relationship between R.a and T.a should be known. If relations R, S, T are stored at different sites, then there is no way to know the relationship among them to perform any reduction in each local site. Without an exchange of information among sites, the smallest raw-target which can be identified is the set containing R.a, S.a, and T.a. If some of them are stored at the same site, say R and S, then the raw-target can be reduced to a set containing T.a and the portion of R.a whose values are also found in S.a. From the above example we can see that the more relations a site has, the more relationships among target-data components for data reduction it can have. Using redundant materialization in target-data identification, the materialized relations in each site are generally more than that of using non-redundant materialization, and consequently, more

relationships among the raw-target components are available for the raw-target reduction. Hence, the volume of final raw-target in a site can be smaller. For example, in the same query as above, the volume of raw-target in a site storing R, S, and T should be the smallest (even the target-data can be identified in this case) among all sites.

Although the redundant materialization may have smaller communication cost, more local processing cost may be involved. A system emphasizing on local processing cost may not be adequate for redundant materialization.

3.2.2 Processing of Joins

Among three most frequently used operations (selection, projection, and join), the join operation is the only one that is a binary operation and that may expand the volume of raw-targets. The efficient processing of join operations becomes the most important job of target-data identification in both CDBMS and DDBMS.

3.2.2.1 Multi-site Joins and Semi-joins

Processing a join operation is more complicated in DDBMS since the two operands of a join operation may reside in two different sites. One approach to process such a join operation is to send one relation in one site to the other site and execute the join operation in the second site. The cost of this approach is a function of the size of entire relation sent. A better approach proposed by Bernstein and Chiu [BER81a] [BER81c] is *semi-join*. They observed that when two distributed relations are joined, one of the operand relations can be reduced in size by deleting those tuples that do not appear in the final result. This may be achieved by sending values of the joining attribute of the second relation to

the first one and performing a join there. This reduction effect is called a *join restriction* [YAO79]. The join is completed by sending the reduced form of the first relation to the second one. Referring to the raw-target model, a semi-join is essentially cutting the raw-target at one side first, then sending the remaining raw-target to the other side. As an example, relations $R(a,b)$ and $S(b,c)$ in Figure 3.2(a) are stored at sites A and B, respectively, and are to be joined on the joining attribute, b. In the first approach, as shown in Figure 3.2(b), relation S is sent to site A and is joined with relation R there. The communication cost is a function of the size of relation S. The semi-join approach is shown in 3.2(c), in which the unique values of attribute R.b is first projected and sent to site B, then relation S is reduced by eliminating those tuples whose values on attribute b are not in attribute R.b. Finally, the remaining tuples of relation S are sent to site A and are joined to R there. The communication cost is a function of R.b and a portion of S.

When multiple relations are to be joined, the most promising approach currently is to first execute a sequence of semi-joins among relations to reduce the size of relations, then the remaining fragments are transferred to a single site at which the complete join is performed. Since the major overhead is due to semi-joins, the minimization of this overhead is the objective in most DQP studies. A comprehensive approach of query processing that incorporates semi-joins has been developed by Apers, Hevner, and Yao [HEV79a] [APE83].

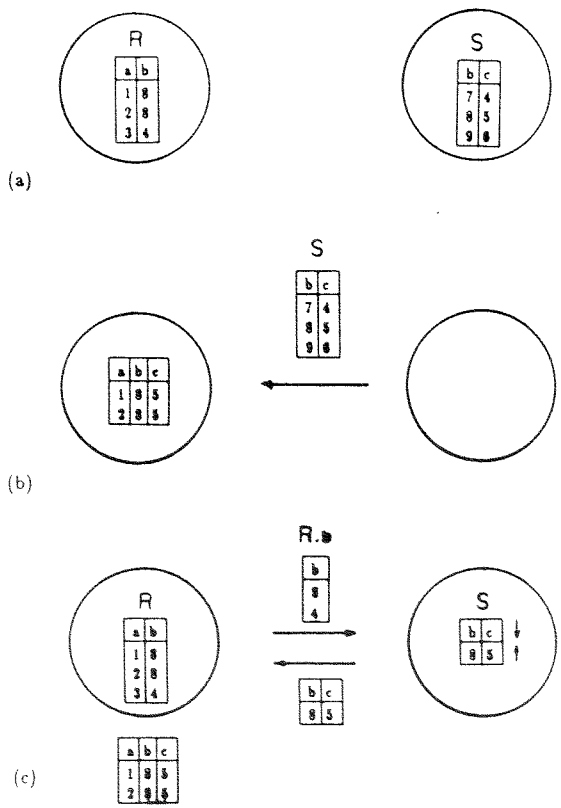


Figure 3.2 An example to illustrate the procedure of join and semi-join (a) relation R and S; (b) sending relation S to join with R; (c) semi-join.

3.2.2.2 Size Estimation and Selectivity

It's obvious that the performance of a DQP algorithm highly depends on the estimation of the sizes of raw-targets. The choice of a reasonable estimation algorithm is, therefore, extremely important.

Suppose that relations $\mathbf{R}_1$ and $\mathbf{R}_2$ are single-attribute relations and the values of the common attribute 'a' are uniformly and independently distributed on the relations. The the size of $\mathbf{R}_2$ after joined with R_1.a has to be estimated. By letting $p_{i.a}$ be the probability that a value appears in R_i.a, i = 1, 2, then $p_{i.a}$ is called the *selectivity* of $\mathbf{R}_i$ on attribute a. Since the expected total number of distinct values existing in attribute a of both relations is $|a|\times p_{1.a}\times p_{2.a}$, where $|a|$ is the cardinality of the domain a. The size of the reduced $\mathbf{R}_2$ can be estimated to be $|a|\times p_{1.a}\times p_{2.a}\times w.a$ or $|\mathbf{R}_2|\times p_{1.a}\times w.a$, where $|\mathbf{R}_2|$ is the cardinality of $\mathbf{R}_2$, and w.a is the width of attribute a. Essentially, the size of $\mathbf{R}_2$ is reduced by a factor of $p_{1.a}$.

In case of joining multiple-attribute relations, not only the size of the joined attribute and the size of the joined relation are reduced, but the size of other attributes are also reduced. This is called an *indirect semi-join*. In the above example, suppose that $\mathbf{R}_2$ is a relation with two attributes, a and b. After semi-joining $\mathbf{R}_1$.a with $\mathbf{R}_2$, the cardinality of $\mathbf{R}_2$ can be estimated as $|\mathbf{R}_2|\times p_{1.a}$. The cardinality of R_2.b was estimated by Bernstein and Chiu [BER81a], which can be demonstrated in the following *ball-color* problem. In this problem, there are n balls with m different colors. It is necessary to find the expected number of colors if t balls are randomly selected from the n balls. The correspondences between the ball-color problem and semi-joins are as follows. The value n corresponds to the number of tuples of $\mathbf{R}_2$ before the semi-join, m is the number in distinct values of $\mathbf{R}_1$ projected on attribute b before the semi-join, and t

corresponds to the number of tuples of R_1 after the semijoin. The expected number of colors of the t selected balls is

$$m \times \left[1 - \prod_{i=1}^{t} \left(\frac{n((m-1)/m) - i + 1}{n - i + 1} \right) \right].$$

The computation of this formula is expensive. An approximation is needed and was given by Bernstein and Chiu. This solution is the same as the solution in Yao's paper [YAO77] for estimating the hit ratio of the block-access problem.

3.2.2.3 Static Semi-join Scheduling

Yao, Hevner, and Apers [APE83] [HEV79a] proposed a *static semi-join scheduling* strategy to solve the DQP. In which, the processing costs of different semi-join schedules are estimated first based on their selectivities; the schedule with the least estimated cost is selected; and the query is executed according to the selected schedule. The problem is NP-hard even for simple queries with one common joining attribute [HEV79b]. The main advantage of this approach is that the optimal schedule of a certain type of queries can be precompiled and reused later. There is no need to produce a schedule for each query processed in real time. On the other hand, there are some disadvantages associated with the static semi-join scheduling.

(1) An accurate estimation is hard to get. The computation of selectivities does not consider the dependency among attributes. In fact, attributes in a real database are usually dependent, although the effects of semi-joins on dependent attributes are difficult to estimate. Further, the assumption on uniform distribution may not be realistic. Lastly, the indirect semi-join effect complicates the problem. In most current research on static semi-join scheduling, the indirect semi-join effect is simply not taken into account.

(2) The errors in estimation usually propagate and accumulate after a sequence of semi-joins. This is especially true when the model of estimation is simple.

(3) A static schedule is inflexible . Once a semi-join schedule is determined, the schedule is generally not allowed to change during the execution. The entire transaction is blocked if the needed relation in any stage is not available. The probability of being blocked is increased as the number of required relations increases or the system load increases.

(4) The computational overhead of static strategies is high. Although heuristic algorithms are generally used to determine semi-join schedules, the computational time is still very high, especially when more accurate solutions are needed. This overhead can only be reduced by using pre-compiled semi-join schedules in a relatively stable system.

3.2.2.4 Dynamic Semi-join Scheduling

An alternative is to use *dynamic semi-join scheduling* [YU83] [SAC84] [WAH84b], in which each individual step in the schedule is determined immediately before the execution of that step based on the the current state of the database. Such a dynamic scheduling algorithm is a greedy heuristic algorithm. Some of the problems associated with static scheduling do not exist in dynamic scheduling.

(1) There is no accumulated estimation error. In each step, at most one estimation is made in dynamic scheduling.

(2) Dynamic scheduling is more flexible, and the probability of being blocked is lower. Whenever the required relation is not available during the execution, the schedule can be changed. Another way is to only schedule semi-joins for available relations.

The success of dynamic semi-join scheduling relies on the selection of the joining attribute. In greedy heuristic, a function called *heuristic function* is designed to measure the possibility of leading to the optimal solution for each candidate. Therefore, the key problems are the design of the heuristic function and the collection of database statistics. Designing a good heuristic function is as complicated as designing a good static scheduling algorithm. There is no easy way to collect statistics on general networks. Status exchange on a distributed system not only involves a lot of communication overhead, but also slows down the query processing. Further, real-time statistics exchange is almost impossible in general networks because the system statistics may be already outdated when it is collected. Therefore, a necessary condition for a DDBMS to employ a dynamic semi-join scheduling algorithm is that an efficient system statistics collection mechanism must be provided. This difficulty has prohibited the use of dynamic schedules in general networks. Fortunately, the broadcast capability of local multiaccess/broadcast bus allows an efficient distribution of status information. Hence, dynamic semi-join scheduling is employed in DDBLMN.

3.3 Previous Work

Most of the previous studies on distributed query processing were based on relational databases in a non-broadcast system. A materialization is assumed for each query, the order of processing is optimized, and the operations are clustered into local actions. Further, the transmission cost per unit data is

assumed to be constant for any two points in the network, and the cost of local processing is negligible. Since the DQP is NP-hard, heuristic algorithms were generally used [WON77]. The semi-join approach was first incorporated into DQP strategies in SDD-1 system by Bernstein and Chiu and was shown to be more efficient than full joins [BER81a] [BER81c]. Yao et al. improved this approach with a static semi-join scheduling strategy. Considerable amount of research have been carried out on finding good static semi-join scheduling strategies.

Further, studies have been made to improve semi-joins. Yu et al. reduced the transmission cost by incorporating complement transmissions into semi-joins [YU82]. The *complement* of an attribute is defined as the set of values absent from a given attribute. In semi-join operations, the complement of an attribute is sent if its cardinality is smaller than that of the original attribute. Nonetheless, there is no efficient algorithm to determine the optimal semi-join sequence, even for simple queries. Data compression techniques have also been applied in semi-joins. For instance, a bit vector indicating the absence or presence of a value can be sent instead of the attribute if the size of the bit-vector is smaller than the size of the attribute [GOU81]. Suboptimal techniques for semi-joins with smaller optimization overhead have also been studied [KRI84]. Chen and Li employed a graphical approach to identify and eliminate redundant transmissions in a semi-join schedule [CHE84]. Query processing on a star network was studied in [KER82]. Static query processing strategies for local area broadcast networks and address ring were developed in Hevner, Wu, and Yao's paper [HEV85]. A good survey was given by Yu and Chang [YU84b].

Chu and Hurley have developed a unified approach that considered both the local processing and transmission costs [CHU82]. A query-tree model for

selecting the transmission sequence and the sites for executing a set of subqueries was proposed.

In short, previous studies on DQP assumed that attributes were independent, ignored the effect of semi-joins on non-joining attributes, and used a non-redundant materialization as well as a static strategy.

3.4 Query Processing Strategy in DDBLMN

In this section, we describe a solution of the DQP with possibly redundant materialization and a dynamic strategy. An attribute is first selected for broadcasting. Semi-joins are performed and the statistics of the resulting database are collected at all sites. Based upon these statistics, the sites cooperatively select the next attribute to broadcast. The process is repeated. This strategy has been proposed before [YU83] [SAC84], but was hampered by the high cost of collecting statistics after each semi-join. The use of a broadcast network allows the collection of statistics at a reasonable cost. The materialization can be either non-redundant or redundant. The system designer should make the decision based on the ratio of the communication cost and the actual local processing cost if it is important. Since non-redundant materialization is a special case of redundant materialization, only redundant materialization is discussed here. We assume that an independent materialization algorithm is provided if the non-redundant materialization is used.

3.4.1 Five Phases in Query Processing

A query processing strategy consisting of five phases is executed at each related site: Concurrency-Control (CC), Local-Processing (LP), Global Semi-join (GSJ), Relations-Transmission (RT), and Post-Processing (PP) phases.

3.4.1.1 Concurrency-Control (CC) Phase

Our proposed concurrency control algorithm utilizes locks. Hence, the relations used in a query must first be locked before any processing is carried out. Locking can be achieved in a distributed fashion by utilizing the transaction information broadcast in the Initial Broadcast Phase. If all relations are to be locked initially before the transaction is processed, this phase is completed at each related site when all needed local relations are not locked by other transactions. However, if the precedence relationship is resolved whenever conflicts are detected, as proposed in the concurrency control algorithm in Chapter IV, then this phase has to be repeated before each semi-join broadcast.

3.4.1.2 Local-Processing (LP) Phase

The LP phase may be started immediately at a related site as soon as the CC Phase is completed. Selections, projections, and local joins are performed on all materialized relations, which are all the relations referenced to by the query, in this phase. After local processing, the relations needed by the query in a related site are joined together into a single relation called the *site-relation*. An attribute in a site-relation is called a *site-attribute*. For clarity, the original relations and attributes are referred to as the *original-relations* and *original-attributes*. The site-relations and site-attributes are used in the discussion of the query processing strategy. The use of the original-attributes will be discussed after the various phases are presented.

Depending on the lock requests received from the bus, the results of the LP phase for a transaction may have to be retracted if a conflicting lock request is received before the first semi-join broadcast for this transaction is sent. This problem can be avoided by broadcasting a message to inform all sites that the LP Phase for a transaction has started, so other sites will not send conflicting lock requests. The synchronization of the LP Phase will be discussed in the next section. The LP Phase for all related sites should be completed before the next phase can begin. This requires the identification of the slowest site and will be discussed in the next section.

3.4.1.3 Global Semi-join (GSJ) Phase

In this phase, semi-joins on site-attributes are carried out. An initial site-attribute is broadcast, and site-relations at other related sites are semi-joined with the broadcast site-attribute. The statistics of the site-attributes are then collected at each related site. Based on measures to be discussed in Section 3.4.4, the site-attribute with the minimum *heuristic value*, called the *minimum attribute*, is selected as the next site-attribute to be broadcast. An algorithm to identify the minimum site-attribute is discussed in the next section. A semi-join may be blocked because one or more relations containing the attribute to be semi-joined are locked by other transactions. To avoid the delay of waiting for these relations to be unlocked, the processing order of semi-joins may be rearranged, so attributes that are not locked can be processed first. The process of selecting and broadcasting minimum attributes is repeated until all site-attributes are broadcast. Two techniques, complement broadcast and non-profitable attribute identification, can be used to reduce the communication

cost and are built into the scheduling algorithm.

Complement Broadcast

Semi-joins can be improved by the use of complement attributes [YU82]. The *initial complement* denotes the complement of a site-attribute before the GSJ Phase. As global semi-joins are carried out, the size of a site-attribute will decrease, while the size of its complement will increase. The size of the complement of an updated site-attribute will always be larger than that of its initial complement. Figure 3.3 shows the relationships among the complement, the initial complement, and the site-attribute itself. When the cardinality of the updated site-attribute is smaller than that of its initial complement, a *normal broadcast* that broadcasts the updated site-attribute should be used; otherwise, a *complement broadcast* that broadcasts the initial complement should be used. A semi-join can be carried out with either the site-attribute or its complement.

Non-Profitable Attribute Identification

Since the materialization is redundant, the information contained in one site-attribute may be a subset of the information contained in another site-attribute at a different site. For example, if $\mathbf{R}_1$ is stored in sites 1 and 2, then the information of the site-attributes at sites 1 and 2 will overlap to each other. More than one copy of $\mathbf{R}_1$ may be joined together. The result of joining redundant relations is equivalent to that of joining single copy relations. Because the result of equi-joining a relation to itself is equivalent to the relation itself, and join operations are commutable. However, redundant information had better not to be broadcast in GSJ phase. The following definitions are defined for the convenience of identifying the non-profitable attributes in GSJ phase.

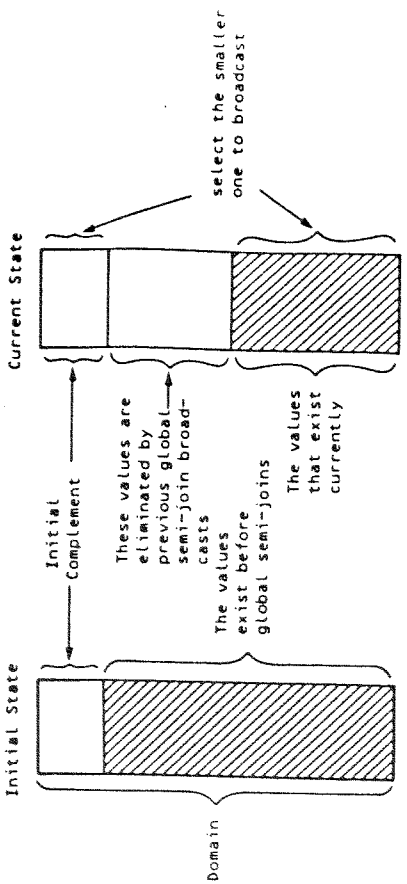


Figure 3.3 The set of values of an attribute, its complement, and the initial complement.

The *component set* of site-attribute j in site-relation $\mathbf{R}_x$ is defined as the set of original-relations in site x that contain site-attribute j. For example, let $\mathbf{R}_1$ with attributes a and b (R_1.a, R_1.b) and $\mathbf{R}_2$ with attributes b and c (R_2.b, R_2.c) be stored in site x. After the LP Phase, a site-relation ($\mathbf{R}_x$) with site-attributes R_x.a, R_x.b, and R_x.c is created by joining $\mathbf{R}_1$ and $\mathbf{R}_2$. The component sets of R_x.a, R_x.b, and R_x.c are $\{\mathbf{R}_1\}$, $\{\mathbf{R}_1, \mathbf{R}_2\}$ and $\{\mathbf{R}_2\}$ respectively.

The *broadcast component set of domain j* is defined as the union of the component sets of site-attributes in domain j that have been broadcast. In GSJ phase, a site-attribute j does not have to be broadcast if its component set is a subset of the broadcast component set of domain j, because the information contained in this site-attribute is already known to all sites. After a site-attribute has been broadcast, the corresponding broadcast component set is updated at all sites. For example, suppose $\mathbf{R}_1$ is stored in sites x and y, $\mathbf{R}_2$ is stored in site y, and $\mathbf{R}_3$ is stored in site z. Assuming that these relations have one common site-attribute j, then site-attribute R_x.j, with component set $\{\mathbf{R}_1\}$, is constructed at x; site-attribute R_y.j, with component set $\{\mathbf{R}_1, \mathbf{R}_2\}$, is constructed at y; and site-attribute R_z.j, with component set $\{\mathbf{R}_3\}$, is constructed at z. If R_y.j is broadcast first in the GSJ Phase, then the broadcast component set of domain j is updated from empty to $\{\mathbf{R}_1, \mathbf{R}_2\}$. Since the component set of R_x.j is one of its subset, R_x.j does not have any reduction effect on R_z.j and does not have to be broadcast.

3.4.1.4 Relations-Transmission (RT) Phase

In this phase, the resulting fragments of the site-relations are broadcast sequentially to the *post processing sites*, where the complete join will be performed. The problem here is the determination of the order of broadcast. Since semi-joins may not execute a join completely [BER81a], an appropriate broadcast order may result in some reduction effects (join restrictions) on the unsent fragments. However, these effects are insignificant, especially when the number of semi-joins in the GSJ Phase is large. Therefore, the site-relations may be broadcast in an arbitrary order. Of course, site-relations at the post processing sites do not have to be broadcast.

3.4.1.5 Post-Processing (PP) Phase

In this phase, the full join and subsequent operations are executed, and the results are sent to the site from which the query originates. The process of determining the post processing sites is different for retrievals and updates. For a retrieval, the set of post processing sites must be selected such that the total overheads of transmission in the RT Phase and transmission of results from the post processing sites to the originating site is minimum. Since the set of post processing sites must be determined before the RT Phase begins, and the difference in overheads of processing at different post processing sites is usually small, the originating site can be chosen as the post processing site arbitraryly. For updates, the results of the post processing sites must also be sent to all sites containing relations to be updated. By an argument similar to that of retrievals, the originating site is chosen as the post processing site. At the beginning of the PP Phase, the originating site will decide to broadcast either the update set or the relations that haven't been broadcast. In the latter case, the update set can

be constructed at any site. Of course, consistency checks must be performed before the corresponding relations are updated. The query is completed after this phase.

An example of query processing in DDBLMN is illustrated in Figure 3.4, 3.5, and 3.6. The query is a relational query with the following predicate,

$$(R_1.a = R_3.a) \text{ AND } (R_1.b = R_2.b = R_4.b) \text{ AND } (R_2.c = R_3.c = R_4.c)$$

The distribution of the original-relations and the resulting site-relations are shown in Figure 3.4. The various phases of query processing are shown in Figure 3.5.

3.4.2 Improvements to The Five-Phase Query Processing

Because of the expansion effect of joins, the size of a site-relation at a site may be much larger than the total size of the original-relations there. To reduce the cost of constructing site-relations, site-relations are not really constructed in LP phase. Instead, original-relations are used. Site-attributes are extracted from the original-attribute to broadcast only when they are needed. As a result, minor changes have to be made in several phases. In the LP Phase, only local semi-joins, instead of full joins, are performed in each related site among the original-attributes. Since only equi-joins are considered, the set of distinct values in a site-attribute at a related site is the same as that in any original-attribute of the same domain there. Therefore, broadcasting a site-attribute $R_x.j$ in the GSJ Phase is equivalent to broadcasting any original-attribute of domain j in site x. After an attribute is broadcast, all original-relations are semi-joined with this attribute. In the RT Phase, fragments of the original-relations, instead of the site-relations, are broadcast. A unique copy of every relation must be selected to broadcast. Since the directory is redundant and the

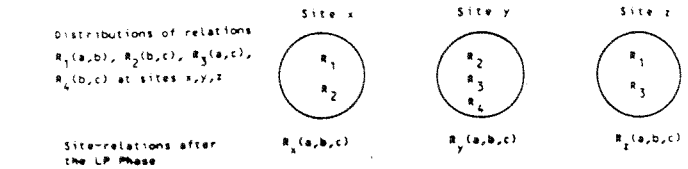


Figure 3.4 Distribution of relations and the result of the LP Phase.

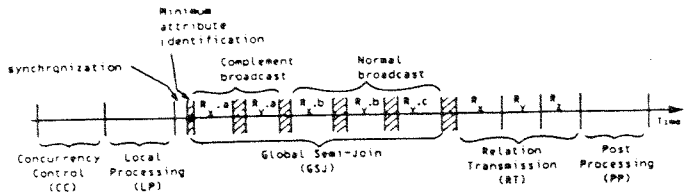


Figure 3.5 An example of the schedule generated by the basic query processing strategy.

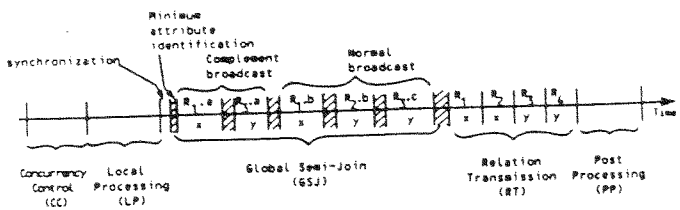


Figure 3.6 An example of the schedule generated by the improved query processing strategy.

size difference among original-relations tends to be small after the GSJ Phase, a tie-breaking rule using the site number can be applied to identify the unique copy to broadcast. An example of the broadcast sequence using the original-attributes for the schedule in Figure 3.5 is illustrated in Figure 3.6.

3.4.3 Protocol for Extremum Identification

In the proposed query processing strategy, it is necessary to synchronize all related sites at the end of each phase and to identify the minimum attribute in each step of GSJ phase. These two problems are equivalent to finding the site with the extremum value and can be solved by the protocol discussed below. Although the proposed protocol does not require additional hardware support, extremum-identification protocols with special hardware interface can also be used [WAH83] [JUA84] [WAH85b]. Without loss of generality, only the problem of identifying the site with the maximum value is studied here. The algorithm is shown in Figure 3.7. It is a distributed algorithm, executed at each site, that determines whether the local parameter, y, is the maximum. Each site contends for the bus and broadcasts its local value. It also listens to the bus for values broadcast by other sites. If the value received is better than y, then it drops out of contention. This process continues until all sites have either broadcast or h .ve been eliminated. The last broadcast value is then the maximum.

This process is complicated by the non-deterministic time to generate the local parameter at each site. After a site has broadcast its local parameter, it sets a timer to allow sites with larger parameters to broadcast their parameters and sites that have not finished evaluating their parameters to broadcast an 'unfinished' message. If a timeout occurs before such a message is received, it will presume that the current maximum is the real maximum. Since the elapsed

```
process maximum (x, y);

{ /* This process executed at site x with value y identifies whether y is the maximum of all sites.
     current_max represents the broadcast maximum.
     next_x is the site that has been selected to broadcast next */

current_max := - x;
compute next_x;   /* estimate the site that will broadcast next */
fork (generate_y);   /* create process to generate y */

while (true) do {
     wait (event);
     case (event) of {

       (y is generated): {
          if (x = next_x) then   /* broadcast y only if this site is next_x */
             signal network layer to contend for bus and to broadcast y;
          case (y) of {
            (y > current_max): current_max := y;
            (y ≤ current_max): return(failure) }

       (receive new_current_max): {
          current_max := max(new_current_max, current_max);
          next_x := x;   /* any site could broadcast next */
          case (y) of {
            (y is not available): {set timer; signal to contend for bus to broadcast 'unfinished'};
            (y > current_max): {set timer; signal to contend for bus to broadcast y};
            (y ≤ current_max): {
               if (trying to broadcast y) then withdraw this broadcast;
               return(failure) } } };

       (receive 'unfinished'): {
          unset timer;   /* there are unfinished sites */
          next_x := site sending the 'unfinished' message;
          if (trying to broadcast) then withdraw this broadcast };

       (get bus): {
          if (y is available) then {broadcast(y); set timer}
          else {broadcast('unfinished'); unset timer};

       (timeout): {
          if (trying to broadcast) then {
               /* timeout occur, all unfinished sites stop any further broadcast */
               withdraw this broadcast;
               return(failure)}
          else return(success) }
       }
     }
}
```

Figure 3.7 Protocol to identify the site with the maximum.

time before receiving such a message depends on the network load, the delivery of these messages should be given higher priority, especially for 'unfinished' messages. To prevent the network from flooded by the unfinished messages, all sites that are still evaluating their parameters will refrain from sending further 'unfinished' messages once the first 'unfinished' message is received. The sequence of events is illustrated in Figure 3.8. The shaded interval indicates a contention period during which sites with larger parameters or 'unfinished' messages contend for the bus. The contention period ends when an 'unfinished' message is broadcast.

The performance of the proposed protocol can be enhanced by making a good initial estimate of the site containing the maximum. In the optimal case, only one broadcast and a timeout period are necessary. If the distribution of the parameters is known, then the maximum can be identified in a constant number of messages (on the average) independent of the number of sites [WAH83] [JUA84] [WAH85b]. On the other hand, if the initial estimate is random, the average number of messages is $O(\log_2 n)$, where n is the total number of sites.

The above extremum-identification procedure can be applied to identify the completion of the LP Phase. Parameter y in Figure 3.7 represents the local completion time, and the site with the longest completion time is sought. The time that each site spends between the Initial Broadcast and the end of the LP Phase is determined by the length of the CC and LP Phases. Since the time to resolve conflicts in each site is known globally (all remote actions are broadcast), and the processing times of selections, projections, and local semi-joins are proportional to the cardinality of the relations, a good estimate of the slowest site can be made by all sites. All other sites must wait for the estimated slowest site to broadcast its completion time before determining the next 'unfinished'

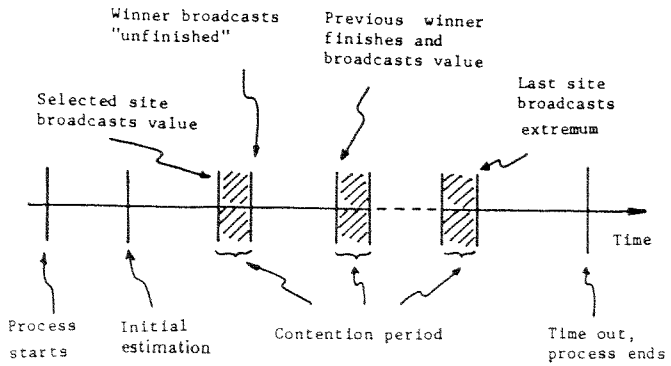


Figure 3.8 The sequence of events in identifying the maximum.

site. To simplify the problem, the time of broadcast can be considered as the completion time of the slowest site, and only a 'finished' message to indicate completion has to be broadcast. After the estimated slowest site has broadcast its 'finished' message, the remaining unfinished sites contend for the bus to elect another site to broadcast the next 'finished' message. If all sites have finished, the last site waits for a timeout period before asserting that the LP Phase has completed.

The procedure in Figure 3.8 can also be applied directly to identify the completion time of each semi-join and the minimum attribute in the GSJ Phase. After one elected site has broadcast its heuristic value, finished sites with larger heuristic values are dropped out of contention, and unfinished sites contend to elect the next site to broadcast. If all sites have finished, sites with smaller heuristic values contend to elect the site with the minimum heuristic value. Since the estimation of the site containing the minimum attribute could be computationally expensive, it may be better to randomly elect it. Another improvement is to allow each site to estimate its heuristic value before it is available, and to drop out of contention if the estimated value is larger than the broadcast value. The next GSJ phase do not have to wait this site. In this way, the slower sites with large heuristic values will not slow down the GSJ phases.

3.4.4 Computing the Heuristic Value

The heuristic function, which estimates the potential of leading to the optimal solution in the GSJ Phase, is another important factor to the system performance. The system performance is affected by the following factors.

(1) The size of an attribute is important because it affects the broadcast cost directly. Only the *effective size*, which is the product of the domain width

and the larger of the current cardinality of the attribute and the cardinality of its initial complement, should be used.

(2) Attributes with larger size-reduction effects on other attributes should be broadcast first because they affect the broadcast cost in successive semi-joins.

(3) The number of unbroadcast attributes on a commor domain is another factor of consideration. When this number is large, the broadcast of any attribute in this domain will have a greater size-reduction effect on other relations.

It is very difficult to have a heuristic function that satisfies all the these rules. Depending on the application, the designer has to combine these parameters with appropriate weights to get a good heuristic function.

As defined in section 3.2.2.2, the selectivity of an attribute is usually used to estimate its size-reduction effect, which is defined as the ratio of the current cardinality of the attribute to the initial cardinality of the domain containing the attribute. When an attribute is semi-joined with another attribute, its new size is computed as the product of its current size and the selectivity of the other attribute. This estimation method is not suitable in a broadcast network because the selectivity of a broadcast attribute may be used more than once in computing the size of other attributes. For example, suppose attributes $R_x.j$, $R_y.j$, $R_z.j$, with sizes $s_{x,j}$, $s_{y,j}$, $s_{z,j}$, and selectivities $p_{x,j}$, $p_{y,j}$, $p_{z,j}$, are the attributes in domain j that are broadcast in sequence. After $R_x.j$ is broadcast, the sizes of $R_y.j$ and $R_z.j$ are changed to $s_{y,j}\times p_{x,j}$ and $s_{z,j}\times p_{x,j}$, and their selectivities are changed to $p_{y,j}\times p_{x,j}$ and $p_{z,j}\times p_{x,j}$. When $R_y.j$ is broadcast next, the size of $R_z.j$ is estimated as $s_{z,j}\times p_{x,j}\times p_{x,j}\times p_{y,j}$, in which $p_{x,j}$ is used twice. This contradicts to the fact that any attributed can only be reduced by another attribute at most once.

To overcome the above problem, the *relative selectivity* is used to measure the size-reduction effect of broadcast attributes. The *relative selectivity* of an attribute in domain j is the ratio of its current cardinality to the cardinality of *current-domain j*, where current-domain j is domain j with the reduction effects taken into account. When an attribute in domain j with relative selectivity p is broadcast, the cardinalities of all attributes in domain j, as well as the cardinality of current-domain j, are reduced by a factor p. As a result, the relative selectivities of all attributes in domain j will not be changed, and the duplicate reduction effects using selectivities will not occur. In the last example, suppose $p'_{x,j}$, $p'_{y,j}$, and $p'_{z,j}$ represent the relative selectivities. Initially, $p'_{x,j}=p_{x,j}$, $p'_{y,j}=p_{y,j}$, and $p'_{z,j}=p_{z,j}$. After $R_x.j$ is broadcast, the sizes of $R_y.j$ and $R_z.j$ are changed to $s_{y,j}\times p'_{x,j}$ and $s_{z,j}\times p'_{x,j}$, but their relative selectivities remain unchanged. Hence, when $R_y.j$ is broadcast, the reduction effects on $R_z.j$ is estimated to be $p'_{x,j}\times p'_{y,j}$.

Four different heuristic functions have been evaluated:

(1) effective size of an attribute;

(2) effective-size × selectivity (selectivity in the conventional sense);

(3) relative selectivity; and

(4) effective-size × relative-selectivity.

These heuristic functions were evaluated by simulations on randomly generated relations. The number of relations and the size of each are limited by the time it needs to find the optimal semi-join schedule in exhaustive enumeration. On the VAX 11/780 computer, it took about three minutes CPU time to simulate a case of three relations with two common attributes each, and 24 hours of CPU time

to simulate a case of three relations with three common attributes each. It was assumed that one copy of each relation existed in the database, and that each existed at a distinct site. The widths of the three attributes were six, eight, and ten respectively. The width of tuples in each relation, including attributes not in the joining domains, was thirty. The domain of each joining attribute was assumed to be integers between one and ten. The distinct values in an attribute were generated by first selecting a threshold, and then including a domain value in the attribute if a random number generated exceeded the threshold. After the attribute values were generated, a cross-product was formed for each relation to obtain the set of possible tuples. Tuples in this set were randomly selected for the relation by a process similar to that of selecting the attribute values. A number of database instances were obtained by using different thresholds.

The simulation results are shown in Table 3.1. The ratios of the total communication costs of various heuristic schedules to that of the optimal schedule are compared. These results demonstrate that dynamic query processing, using relative selectivity as the heuristic function, is very effective. The merits of the heuristic functions must be evaluated again when the relations are generated differently.

3.5 Intelligent Query Scheduling

To take the advantage of available history knowledge in a relatively stable database system, an expert-system approach is proposed here. An expert system is included into the DDBMS to help the transaction handler to semi-join schedules. The interactions between the semijoin scheduler and the *Scheduling Expert System* may be in three different modes: **real-time, offline,** and **hybrid.**

Table 3.1 Relative performance of different heuristic schedules as compared to the optimal schedule.

(a) Average behavior of 64 cases for three relations with two common attributes each.

	Effective-Size	Effective-Size · selectivity	Relative Selectivity	Effective-Size · Relative-Selectivity
Average	1.912	2.037	1.138	2.037
Stand. Dev.	0.338	0.388	0.080	0.388
Maximum	2.537	2.537	1.200	2.537
Minimum	1.099	1.099	1.022	1.099

(b) Average behavior of 17 cases for three relations with three common attributes each.

	Effective-Size	Effective-Size · selectivity	Relative Selectivity	Effective-Size · Relative-Selectivity
Average	1.742	1.070	1.002	1.058
Stand. dev.	0.459	0.215	0.004	0.206
Maximum	2.200	1.878	1.017	1.878
Minimum	1.000	1.000	1.000	1.000

In the *real-time mode*, the expert system provides the necessary knowledge to the scheduler in the scheduling. Both static and dynamic scheduling are capable of operating in this mode. In the *offline mode*, the expert system provides the scheduler with precompiled schedules stored in its Knowledge Base. *Query-reusable systems* that store precompiled schedules and apply them later fall into this category. Only static scheduling can operate in this mode. Lastly, in the *hybrid mode*, the expert system provides precompiled schedules as well as the necessary expertise to the scheduler. Only dynamic scheduling is suitable in this mode. The expert-system approach allows better scheduling algorithms to be employed that are often too complicated to use in real time. Moreover, the expert system can learn and adjust its scheduling strategies according to changes in the database.

The logical structure of the proposed approach is illustrated in Figure 3.9. The semijoin scheduler in a Transaction Handler is the part of the code that generates semijoin sequences. When processing a query, the semijoin scheduler consults the Scheduling Expert System to get ready-to-run schedules or scheduling expertise. The Scheduling Expert System has the following modules.

(1) *Pattern Recognizer*: This module classifies the given queries coupled with the database states into system recognizable classes. Without this module, the input domain of the Scheduling Expert System is too large to manage.

(2) *Expert Scheduler*: This module is the inference module that serves the Semijoin Scheduler and the Learning Module by returning schedules in its Knowledge Base. In the simplest form, this module can be a knowledge retrieval system.

(3) *Knowledge Base*: This module stores all the knowledge of the system. The knowledge can be inference rules, expertise of semijoin scheduling,

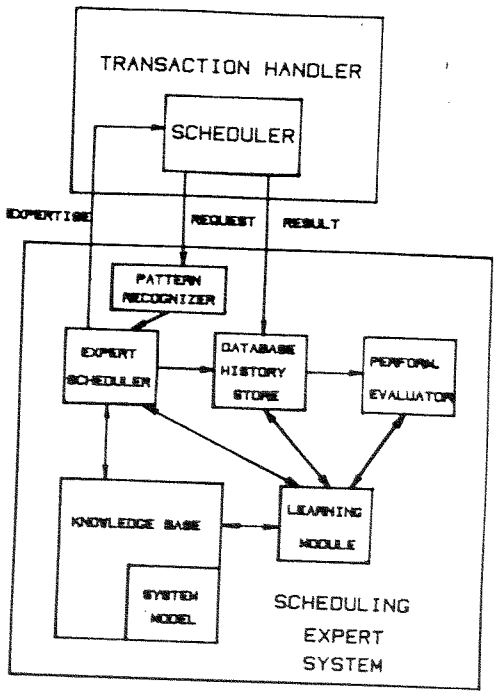


Figure 3.9 Intelligent semijoin scheduling model.

weighting functions, or even the database history. A system model can also be maintained in the Knowledge Base.

(4) *Database History Store*: The most recent database history including queries with schedules, results, and database states is stored here. This raw knowledge is analyzed by the Learning Module periodically.

(5) *Performance Evaluator*: When requested by the Learning Module, this module evaluates the performance of the schedules. It is not practical to get an absolute measure on how good a schedule is because the system cannot exhaust different schedules for one query on the same database state. The evaluation may have to be statistical.

(6) *Learning Module*: This module analyzes the database history to extract the scheduling expertise. It can be as simple as a copy mechanism that merely copies the database history into its Knowledge Base, or it can extract the useful knowledge from the history and compiles them into statistics in the Knowledge Base. It is the heart of the Scheduling Expert System for the following reasons. First, no substantial experience has been collected and built into the database at design time, hence new experience must be learned. Second, the scheduling problem is an NP-hard problem, and no simple rule can guarantee its optimality. More experience would probably help in producing better schedules. Lastly, the performance of the scheduling strategies is application-dependent as well as time-dependent. Thus learning capability can help to adapt to changes in the system.

3.6 Intelligent Scheduling in DDBLMN

The current major difficulty of implementing such an intelligent scheduling system in a DDBMS system is that we do not have enough real-time experience with DQP. We also do not have access to a real DDBMS. Currently, only pure dynamic scheduling is adopted in DDBLMN. This may suffer from high local processing overheads due to possible redundant semijoins in a system with large disk overheads. Further, the communication overheads in synchronization and in identifying the site with the minimum attribute may hamper the effectiveness of dynamic scheduling. To reduce these overheads, we will adopt more intelligent scheduling in DDBLMN in the future.

Three alternatives in implementing intelligent scheduling will be considered. In the first case, dynamic scheduling is used with additional knowledge provided by the Scheduling Expert System to estimate the site with the minimum attribute in semijoins and the slowest site in synchronization. The number of network messages is expected to be reduced, while the local processing overhead will not be reduced significantly. The second alternative, *adaptive hybrid scheduling*, combines static and dynamic scheduling. The scheduler will schedule semijoin broadcasts according to precompiled schedules. When a scheduled broadcast cannot proceed because one or more related relations are locked by other transactions and are inaccessible, the scheduler either chooses the next candidate for broadcast or switches to dynamic scheduling. In this way, both communication and local-processing overheads are expected to be reduced. The cost of implementing the Scheduling Expert System in the above cases would be approximately the same. In the third alternative, scheduling expertise extracted from the database history is used by the scheduler. This is most likely to exist in the form of weighting functions.

The scheduler produces a schedule for each query with the help of the Scheduling Expert System and schedules the semijoin broadcasts as in adaptive hybrid scheduling. The biggest challenge in this approach is the design of the Learning Module.

3.7 Summary

In summary, the most important problem in distributed query processing is to find a strategy to identify the target-data with the minimum processing cost in the processing of multi-relation joins. For a system in which the communication cost dominates the overall processing cost, the most promising approach is to execute a sequence of semi-join to reduce the raw-target before a complete join is executed. A five phase query processing strategy with redundant materialization and dynamic semi-join scheduling is proposed in this chapter. Coupled with concurrency control and redundant materialization, dynamic query processing strategies provide better flexibility than static strategies. Further, the propagation of errors in static semi-join scheduling is avoided. An $O(\log_2 N)$ algorithm based on the characteristic of local multiaccess/broadcast networks is developed to provide an efficient method of collecting statistics in the dynamic strategies.

Finally, an intelligent query processing model is proposed for future research to improve the efficiency of query processing.

CHAPTER IV

CONCURRENCY CONTROL

In this chapter, a concurrency control protocol for local multiaccess/broadcast networks based on the conflict graph and Thomas' [THO79] algorithm is presented. Using the special characteristics of the network, we modify existing protocols of wide-area networks, such that the proposed protocol can suit to this local system. The broadcast capability and short propagation delay time of the network simplify the distribution of complete transaction status to every site. Hence, the protocol behaves similarly as a single site concurrency control protocol.

4.1 Introduction

In Bernstein and Goodman's paper [BER81b], concurrency control is defined as "the activity of coordinating concurrent accesses to a database in a multiuser environment." Concurrency control permits users to share a database while preserving the illusion that each user is executing alone in a dedicated system. The consistency of a DDBMS is endangered by three factors that have been stated in Chapter I.

(1) Multiple copies of a piece of data may be stored, accessed, and updated by different users at different sites at the same time.

(2) Communication delays usually prohibit instantaneous distribution of system status information.

(3) The information exchange may be impossible when some components fail.

The following are two typical anomalies that may be resulted in an ill-controlled electronic funds transfer system.

(1) **Lost Updates**. Suppose two customers try to deposit money into the same account simultaneously through automatic teller machines (ATM). These two transactions may interfere with each other. The two ATM's handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances back into the database. One deposit is overwritten by the other and is lost.

(2) **Inconsistent Retrievals.** Suppose a company has a saving account and a checking account in the system, and each initially contains balances of $1000. A transaction TR_1 is to transfer $500 from the saving account to the checking account, and another transaction TR_2 is to query the total balance of both accounts. In the absence of concurrency control, TR_2 may read the balance of the saving account before TR_1 is processed ($1000) and read the balance of the checking account after TR_1 is processed ($1500). The total balances that TR_2 gets is $2500, rather than $2000.

The objectives of concurrency control are

(1) correctness,

(2) maximizing the system throughput,

(3) minimizing the overhead, and

(4) simplicity.

Objective (1) is essential in a concurrency control protocol, while a compromise among objectives (2), (3), and (4) may be needed. Simplicity is important in practice since a complicated protocol is hard to prove to be correct, hard to implement, and less robust.

4.1.1 Consistency and Serializability

The correctness of the concurrency control algorithm depends on the requirements of the database. When the processing order of a set of transactions is order independent, i.e. any order is acceptable, the general condition to be enforced is *serializability* [PAP79]. Let E denote an execution of transactions $TR_1, TR_2, \ldots, TR_n$. E is a *serial execution* if no transaction is executed concurrently in E; that is, each transaction is executed to completion before the next one begins. An execution is *serializable* if it is computational equivalent to a serial execution, that is, if it produces the same output and has the same effect on the database as some serial execution. The concurrency control of a DDBMS should ensure that all executions are serializable.

The *read (or write) set* of a transaction is the set of data items that are read (or written) by the transaction. A *conflict* occurs between two transactions if the write set of one transaction intersects with the read set (r-w conflict) or write set (w-w conflict) of the other. A set of transactions can be executed in parallel and are serializable if the execution order of conflicting transactions is carefully arranged.

4.1.2 Throughput

The system throughput is defined as the average number of transactions a system can process in a time unit. In a real DDBMS, the definition is a little vague since the lengths of transactions and the requested data entities are variant. In some cases, the '*degree of concurrency*,' which is the number of transactions that can be processed at the same time, can also be used to measure the performance of the concurrency control protocols. We use 'throughput' and 'degree of concurrency' interchangeably in this chapter. More

precise definitions can be found in Papadimitriou's papers [PAP79] [PAP82].

Given a set of transactions, we can always find a serializable execution with the maximum throughput. However, it's not practical to achieve in a real DDBMS for the following reasons.

(1) The arrival of transactions and the behavior of transactions are difficult or even impossible to predict.

(2) Even if the transactions are predictable, finding an optimal solution is computational expensive. To determine whether an execution is serializable or not has been proved to be NP-complete [PAP79].

(3) An optimal schedule may result in longer delays for some transactions. A real DDBMS may wish transactions to be serviced in a *first-come-first-serve* order.

As a consequence, the most commonly used strategy in concurrency control protocols is a combination of '*first-come-first-serve*' and '*trial-and-error.*' When a transaction enters the system, it is immediately processed until it is terminated or is stopped for some reasons. In the latter case, the stopped transaction either waits until the cause is resolved or restarts.

4.2 Previous Work

Many concurrency control algorithms have been proposed before. They can be classified into three categories: locking, timestamp ordering, and optimistic. Bernstein and Goodman provided a good survey on locking and timestamp ordering based protocols [BER81b]. In a *locking protocol*, data items are locked before they are accessed. Locks must be issued in such a way that the transactions are serializable, and if deadlock occurs, recovery must be possible. Locks are released after the access is completed. The most popular locking

protocol is the *two-phase protocol* in which unlocks for a transaction are issued after all locks have been made. The execution is divided into two phases. During the first phase, called the *growing phase*, the transaction can only request locks, and during the second phase, called the *shrinking phase*, the transaction can only release locks.

In a *timestamp-ordering protocol*, a transaction is assigned a unique timestamp equal to the value of the local clock when it is initiated. In addition, each data item is also timestamped with its most recent read and write. Before a transaction is processed, its timestamp is compared with the timestamps of the accessed data. A transaction is rejected if a conflict is detected. When a transaction is rejected, it should be restarted with a new timestamp, and all intermediate results have to be discarded. An important timestamp ordering protocol is Thomas's *majority consensus timestamp ordering protocol* [THO79].

In an *optimistic protocol* [KUN81] [BHA82b], each transaction works on a private copy of the database and no control is imposed on the execution of transactions. The consistency is checked on the termination of a transaction. The transaction is terminated if it has operated on a consistent state; otherwise, the transaction is restarted.

4.2.1 Comparisons Among the Three Types Of Protocols

In both timestamp-ordering and optimistic protocols, there is no overhead on lock management. A very high degree of concurrency can be achieved if no conflict happens. Obviously, the success of both protocols is highly dependent on the probability of conflicts among transactions, and the throughput tends to decrease as the system load increases. The major problem of using these two types of protocols on a DDBMS is the penalty of restarts, as the query processing

cost is high is restarting a transaction. In addition to the restart overhead, the additional overhead is needed in both protocols for the authorization or the validation of updates.

The expensive restarts can be avoided in lock based protocols. However, they incur the following overheads when the network does not have the broadcast capability.

(1) *Messages for requesting/releasing locks.* For each data access, three sets of messages are needed: a set of messages for requesting locks on all accessed data, a set of acknowledgements in granting the locks, and a set of messages for releasing the locks. Additional time is also spent in waiting for locks.

(2) *Extra overhead for deadlock prevention, detection, and recovery.* Since deadlocks may occur in lock based protocols, additional overheads are incurred on deadlock detection and recovery. The degree of concurrency may be degraded when deadlock prevention is enforced.

(3) *Extra waiting time for overlocking.* A data item may be locked unnecessarily to minimize the possibility of deadlocks and to enforce serializability. Overlocking exists in protocols that request locks initially and in two-phase protocols.

In summary, information exchange is necessary in all three types of protocols. In a lock based protocol, it is done at the operation (read, write) level to ensure serializability. In timestamp ordering protocols, it is done at the update level, while in optimistic protocols, it is done at the transaction level. When the number of writes is much less than the number of reads, the latter two types of protocols are more efficient.

4.3 Concurrency Control in DDBLMN

Based on the characteristic of local multiaccess/broadcast networks, a lock based protocol is proposed in this section for its high efficiency and throughput.

Recall the characteristics of local multiaccess/broadcast networks stated in Section 1.4.4.

(1) The system status is almost completely available to all stations by monitoring the activities on the bus.

(2) Every station gets the information on the bus almost instantaneously, since it is a multi-point configuration.

(3) All messages arrive at each site in the same order as they are sent, when messages are not lost.

Due to properties (1) and (2), status exchange on bus networks is extremely efficient and fast. It is a natural to use lock based protocols on these networks. Due to the property (3), the transactions are not distinguished from global transactions. Hence, every transaction refers to the same database state.

4.3.1 General Description

It is assumed that the transaction processing model discussed in Section 1.8 is used, that a relation is the basic unit of a data item, and that an access may be either a read or a write but not both.

Broadcast Transaction Table and Precedence Graph

To record the global database state, each site maintains a *Broadcast Transaction Table (BTT)* that records the status of all transactions and the status of relations (locked or unlocked). All Broadcast Transaction Tables are always identical. Based on the information stored in BTT, a *Precedence Graph* (PG) recording the precedence relationship among transactions is also maintained in each site. A transaction is registered into the BTT and PG since its initial broadcast and is retained until its termination. The precedence relationship between two transactions can be added due to a conflict, deleted due to a termination, but cannot be reversed in any case. The serializability of transactions is ensured by this property.

Interaction between transactions and Concurrency-Control subsystem

When a transaction wants to place a lock on a relation, it makes the lock request to the Concurrency-Control subsystem in the local site. The Concurrency-Control subsystem checks the local BTT and PR to determine the serializability condition. The transaction has to wait until serializability is satisfied. With a common bus and assuming that locks received are processed in real time before a lock request is broadcast, only lock requests that will not be rejected by other sites are broadcast. Once a lock is broadcast, the lock is automatically granted to the transaction and is recorded in the the BTT and PG in each site. The newly granted lock will not conflict with other locks that had been granted since the consistency had been checked by the local Concurrency-Control subsystem. If this causes any conflict to the lock requests under processing, then all those conflicting requests must be rejected. Since at most one broadcast can be made at any time on the bus, it is impossible to grant two

conflicting locks simultaneously. In this sense, the common bus acts as a gateway for concurrency control as in a centralized database.

Information exchange

Essentially, information exchange is not needed for concurrency control in DDBLMN. The read set and write set of a transaction are known to all other transactions through the initial broadcast. Moreover, lock and unlock messages can be embedded into the corresponding network activities. Recall the procedure of query processing in Chapter III, the lock and unlock messages can be embedded in each semi-join broadcast and relation broadcast. Although there is no need to explicitly specify the granted locks or released locks in these broadcasts (they can be figured out from the information in BTT), we recommend that the system designer piggybacks this information on the broadcasts to avoid significant overhead to trace the transactions.

The efficiency of information exchange not only saves the communication overhead, but also saves the time waiting for locks and the time of overlocking. The time of overlocking is saved because a lock is released as soon as it is not used by the transaction.

Given a processing order for a set of transactions, the *latest locking time* of a lock on a relation by a transaction is the time that the transaction begins to use (read or write) it. The *earliest unlocking time* is the time that the transaction no longer needs it. The minimum locking time is the difference between these two time instances. Within this time period, a conflict accessing to the relation will introduce inconsistency into the database. Since the information exchange on the network is almost instantaneous, the minimum locking time can be achieved if each transaction issues the lock request at the

right time and responds to lock releasing quickly. This will be discussed later in this section. It must be pointed out that minimum locking does not imply the maximum degree of concurrency which can only be achieved when all the permutations on the transactions are compared.

Precedence order

The precedence order of transactions is governed by the times that a conflict is found. The algorithm detects access conflicts between two transactions from their read and write sets when they are broadcast; however, the order of processing is not imposed until the conflicting access of one of these transactions is made. After the order of processing is defined between TR_1 and TR_2 (say TR_1 precedes TR_2), then all conflicting accesses made by TR_1 must precede accesses of TR_2 to enforce serializability.

This protocol is deadlock free because preceding transactions will never wait for locks from succeeding transactions.

4.3.2 The Locking/Unlocking Time

The correctness and the performance of the protocol are dependent on the time to lock and unlock relations. These are discussed in this section. Depending on the query processing strategy discussed in Chapter III, the time instances of locking and unlocking for read and write operations are different.

Locking time for read-locks

The time instances for a transaction to read-lock a relation is the time that the relation is first read by the transaction. Consistency checks are performed before the relation is locked. For relations that are semi-joined in the LP Phase, they should be locked before the LP Phase begins. For other relations that belong to the read set of the transaction but are not used in the LP Phase, they should be locked once one of their attributes is either broadcast or semi-joined with a broadcast attribute.

Unlocking time for read-locks

Depending on the storage capacity, there are three different time instances to unlock read-locks. In the first case, when the storage capacity is large enough and the access to a relation is read only, a copy of the relation (with only the attributes to be accessed) is made in the working storage when it is read-locked, and the relation is unlocked immediately. An update on the relation by the succeeding transactions will not affect the values obtained by the current transaction.

In the second case, when duplicate copies cannot be made due to storage limitation, then the relation must be read-locked until it is no longer read by the transaction. The locking time in this case may be too long due to the long query processing delay. Thus, the overall system performance may be degraded drastically.

For a system with some extra storage capacity, the following approach can provide much better performance at the cost of some extra storage. When duplicated copies cannot be made, a projection on the attributes that will be used in the global semi-join phase is stored in the working storage. Thus, these

read-only relations can be unlocked at the same time as that in the first case. These projected attributes provide less reductions in the semi-join operations than those provided by the entire relations in the first approach, as indirect semi-joins cannot be used. As a consequence, the query processing overhead may be slightly increased; however the reduction effects of indirect semi-joins are generally small.

Locking and unlocking times of write-locks

A relation that is updated must be write-locked at the beginning of the PP Phase by the first query in the transaction that updates the relation, and released at the end of the PP Phase by the last query in the transaction that updates the relation. For relations that are read and written in the same transaction, the read and write locks must be released together.

4.3.3 Minimum Locking Concurrency Control Protocol

The detailed description of the protocol is given in this section. The distributed concurrency control algorithm is shown in Figure 4.1, 4.2, and 4.3. The precedence relationships among transactions are stored in a precedence graph and are maintained by procedure *consistency* (Figure 4.3). Procedure *precedencecheck* (Figure 4.2) checks whether two transactions conflict with each other.

Process *cc_minimumlock* (Figure 4.1) is triggered by the arrivals of lock requests, the successful contention in getting the bus, the termination of transactions, and the release of locks. Transactions originating from a site communicate with the local process *cc_minimumlock*. Each transaction is a process, which can be in one of the following states: THINKING, SUSPENDED,


```
process cc_minimumlock (BTT, PG);
/* This process serves the asynchronous arrivals of lock requests, unlock requests, and termination of
   transactions. The relations are locked for the minimum amount of time with respect to precedent
   transactions. It is assumed that a transaction can lock one relation in each request, and that only
   one request is processed at a time (due to the single bus). A transaction that has been initiated
   can be in one of the following states: PENDING: all lock requests received must be checked first
   to determine if they conflict with the current request; ACTIVE: the transaction is waiting for the
   availability of the bus; SUSPENDED: the transaction is not processed because there is another
   transaction that is ACTIVE or PENDING, or it conflicts with another transaction that has been
   broadcast from another site; and THINKING: transaction is not making any lock request.
   Information about the active transactions and locked relations are available in BTT, the Broadcast
   Transaction Table. Information about the precedence relations of transactions are stored in PG,
   the Precedence Graph. */
{  while (true) do {
       wait (event);

       case (event) of {
          (arrival of lock request from TR_i): {
             if ((there is a PENDING or ACTIVE request) then
                set state of TR_i to SUSPENDED and suspend TR_i
             else {
                PG' = PG;  /* make temporary copy of PG */
                if (consistency(TR_i, PG') = success) then
                   if (new lock requests have been received from other sites) then
                      set state of TR_i to PENDING
                   else {
                      set state of TR_i to ACTIVE;
                      signal network layer to contend for bus }
                else {
                   if (consistency(TR_i, PG') = waive) then { update BTT; return (waive) }
                   else set state of TR_i to SUSPENDED and suspend TR_i;
                   if (there are SUSPENDED requests) then wake up one SUSPENDED request }}}
          (receive lock request from TR_i from bus): { /* new lock is granted at another site */
             update BTT and PG;
             if (there is a PENDING or ACTIVE request TR_j) then {
                update PG';
                if (precedencecheck(TR_i, TR_j, PG') = reject or waive) then {
                   if (TR_j is ACTIVE) then
                      withdraw bus-contention request;
                   if (TR_j is to be rejected) then
                      set state of TR_j to SUSPENDED and suspend TR_j
                   else return (waive);
                   if (there are SUSPENDED requests) then wake up one SUSPENDED request } }
                else if (TR_j is PENDING) then{
                      set state of TR_j to ACTIVE;
                      signal network layer to contend for bus }}
          (a transaction is terminated at another site): update BTT, PG, and PG' (if necessary);
          (unlock is received from bus): update BTT;
          (bus is obtained): {
             broadcast lock request for ACTIVE transaction;
             PG = PG';
             update BTT for locks obtained by the broadcast transaction;
             set state of ACTIVE transaction to THINKING;
             if (there are SUSPENDED requests) then wake up one SUSPENDED request } }
   }
}
```

Figure 4.1 Process for concurrency control with minimum locking time.

```
procedure precedenceCheck (TR_global, TR_local, PG);
 * This procedure checks the precedence graph PG to determine whether the lock request by TR_local
   conflicts with TR_global.  TR_global is the transaction that generates the new lock; TR_local is the local
   transaction that is in the PENDING or ACTIVE state. */
{  /* check the consistency for r-w conflicts */
   if (edge exists from N_global to N_local in PG and there is r-w or w-w conflict) then
      return (reject);
   else if (edge exists from N_local to N_global and there is a w-w conflict) then
      return (waive);
   else return (success)
}
```

Figure 4.2 Procedure for checking precedence relation between two transactions.

```
procedure consistency (TR_i, PG);
/* TR_i is the requesting transaction.  Transaction TR_i is requesting to read or write Relation R and
   is represented as a node N_i in PG.  This procedure checks (and updates if TR_i is a new
   transaction) PG for conflicts.  An edge will only be added only when one of the conflicting
   relations has been accessed.  All transitive arcs exist in PG since the precedence relation is
   transitive */

{  if (N_i does not exist in PG) then add a node N_i to PG;

   /* check TR_i for r-w conflicts */
   for (all nodes N_j in PG ≠ N_i) do {
      if (there is no edge between N_i and N_j) then
         if (TR_i has r-w conflict with TR_j) then
            if (R has not been accessed by TR_j) then {
               add an edge from N_i to N_j; add all necessary transitive arcs }
            else if (R is locked by TR_j) then {
               add an edge from N_j to N_i; add all necessary transitive arcs; return (reject) }
            else { /* R is unlocked by TR_j */
               add an edge from N_j to N_i; add all necessary transitive arcs };
      else if (there is an edge from N_j to N_i)
      and (r-w conflict is on R that has not been released by TR_j) then
            return (reject) }
   if (TR_i is a read request on R) then return (success);   /* all r-w conflicts have been checked */

   /* check TR_i for w-w conflicts if TR_i is making a write request on R */
   for (all nodes N_j in PG ≠ N_i) do {
      if (there is no edge between N_i and N_j) then
         if (TR_i has w-w conflict on R with TR_j) then {
            /* several alternatives exist here, the strategy used minimizes the number of updates */
            add an edge from N_i to N_j; add all necessary transitive arcs; return (waive) }
      else if (edge exists from TR_j to TR_i) and (R is write-locked by TR_j) then
         return (reject)   /* write on R for TR_i will be waived later if write has not started */
      else if (edge exists from TR_i to TR_j) and (TR_i has w-w conflict with TR_j on R)
      and (R is locked or has been unlocked by TR_j) then
         return (waive) };   /* R has been modified by another transaction of lower precedence */
   return (success)   /* all w-w conflicts have been checked */
}
```

Figure 4.3 Procedure for maintaining the precedence graph.

PENDING, or ACTIVE. When a transaction is in the THINKING state, it does not generate any request to the database. After a transaction, say TR_i, makes a request, consistency checks are made on a temporary copy of the precedence graph to determine if any conflict exists between TR_i and other transactions in the precedence graph. If a conflict is found, then TR_i is SUSPENDED and processed later. When no conflict exists, TR_i is put into the PENDING state, so lock messages received from other sites can be processed. TR_i will be SUSPENDED if a conflict is found in this state; otherwise, it will signal the network layer to contend for the bus and will be put into the ACTIVE state. The ACTIVE state is terminated when the bus is granted to TR_i and the request is serviced. At this time, the original precedence graph is updated. If a lock request received conflicts with TR_i when it is ACTIVE, then TR_i is SUSPENDED and the request to contend for the bus is withdrawn.

Procedure *consistency* (Figure 4.3) maintains the precedence graph. It defines the processing order of two transactions if they have (r-w or w-w) conflict on accessing a common relation. Once this order is defined, all accesses with a r-w conflict on a common relation by a transaction of lower precedence must be initiated after the conflicting relation has been unlocked by transactions of higher precedence. For transactions with a w-w conflict, the transaction with a higher precedence can be waived because the effect on the database is equivalent to the update by the other transaction alone. To allow transactions to be processed efficiently, the precedence order should be defined as late as possible because the transactions are processed asynchronously, and the transaction which first makes the conflicting access is unknown at the Initial Broadcast Time. If a conflict is detected when a transaction, say TR_i, is requesting an access, and if an edge does not exist between the two nodes representing TR_i

and the conflicting transaction, say TR_j, in the precedence graph, then an edge is added between the nodes. Transitive edges are also added to the precedence graph since the precedence relationship is transitive. The rules for determining whether a transition is accepted, rejected, or waived are specified in procedure *consistency*.

An example of concurrent accesses of four transactions as controlled by the proposed algorithm is shown in Figure 4.4. Boxed requests are not executed because they are either rejected or waived. In Step 4, when TR_2 requests a write-lock on A, there is a r-w conflict between TR_1 and TR_2, and TR_2 is delayed because an edge exist in the precedence graph from N_1 to N_2. In Step 6, the write request for TR_2 can proceed because A has been unlocked. In Step 8, N_3 is added to the graph. Since B was accessed by TR_2, an arc is included from N_2 to N_3, and a transitive arc from N_1 to N_3 is also added. TR_3 is allowed to proceed because B has been released by TR_2. In Step 14, the request to update C by TR_2 is waived because the update on C by TR_3 has been carried out and would have over-written the update by TR_2 (an edge exists from N_2 to N_3).

The serializability and deadlock-free properties of the proposed algorithm are proved in the following theorems.

Theorem 4.1: The sequence of requests as controlled by procedure *cc_minimumlock* represents a serializable execution of the transactions.

Proof: Serializability for a read-only write-only model is satisfied when the requests are scheduled according to an acyclic precedence graph [PAP79] [ULL82]. To prove that the sequence of requests as scheduled by *cc_minimumlock* is serializable, it is necessary to show that the precedence graph as maintained by *consistency* is acyclic, and that *cc_minimumlock* follows this

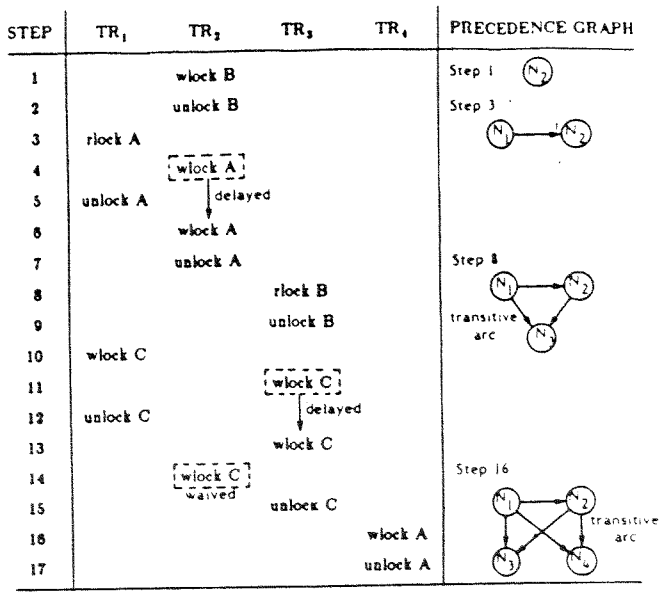


Figure 4.4 An example to illustrate the concurrent execution of four transactions.

precedence graph in scheduling the requests. The precedence graph as maintained by *consistency* is acyclic because transitive arcs are always added whenever a new arc is added to the graph, and no arc will be added between two nodes if an arc already exists between them. To show that the resulting precedence graph is followed by *cc_minimumlock* in scheduling all requests, it can be seen that the precedence relationship of currently active transactions as defined by the precedence graph is always followed whenever a lock is requested, that an arc is added to the precedence graph whenever a conflicting access is first discovered between two transactions, and that this arc is maintained until one of these transactions is terminated. Hence, the resulting precedence graph obtained at the end of a sequence of requests can be assumed to exist before the requests are executed, and a linear order of execution can be enforced for all transactions, which results in a serializable execution. □

Theorem 4.2: The proposed protocol *cc_minimumlock* is deadlock-free.

Proof: Given a set of transactions $TR_1, TR_2, ..., TR_n$, the wait-for relationship in an execution controlled by *cc_minimumlock* can be represented in the following wait-for graph. For each transaction TR_i, there is a node N_i in the wait-for graph. A directional arc connects N_i to N_j if TR_i waits for TR_j on a conflicting access. Since only transactions of lower precedence must wait for transactions of higher precedence, and the precedence graph is acyclic, the wait-for graph is also acyclic, and the algorithm is deadlock free. □

The proposed concurrency control algorithm does not impose the precedence relationship between two transitions until the first conflict between them is detected. This allows shorter transactions to proceed first without waiting for the longer ones. However, the maintenance of the precedence graph

at each site may pose a significant overhead. In this case, a predefined precedence relationship may be used when the transactions are initiated. One convenient way is to define a higher precedence for a transaction with a smaller Initial Broadcast Time. The serializability and deadlock-free properties will still be satisfied.

4.4 Local Queries

We have assumed that the status of all transactions is known to all sites in the proposed concurrency control algorithm. However, a local query that only accesses local relations may block other global transactions and this fact is unknown to other sites. To solve this problem, additional status messages can be broadcast during the processing of local queries. This overhead may be large because a database with a good data distribution would have many local queries. Another method is to allow the local queries to be processed when the accessed relations are not locked. When a global transaction TR_1 wishes to access a relation that is locked by a local query TR_2, it will be blocked in the LP Phase until the relation is unlocked. The site at which TR_2 originates will broadcast an 'unfinished' message when other sites involved in TR_1 are trying to identify the completion of the LP Phase. When TR_2 is finished, this site will broadcast a 'finished' message and start the GSJ Phase for TR_1.

4.5 Summary

In this chapter, the problem of concurrency control in DDBMS's is examined. Based on the property that information exchange on local multiaccess/broadcast networks is extremely efficient and fast, a lock based protocol using complete system information is developed. The complete system

information is available to each site without additional overhead by monitoring the activities on the networks. A lock based protocol on such networks avoids the large overhead and restarts of a similar protocol in general networks. Although the maximum degree of concurrency is difficult to achieve, the protocol places locks on relations with the minimum locking time. The execution of transactions as controlled by the protocol is proved to be serializable and deadlock free. Through a careful design of locking and unlocking times, the concurrency control protocol is integrated into the distributed query processing strategies.

CHAPTER V

GENERALIZED QUERY PROCESSING

5.1 DQP in Future Communication Networks

Currently, all DQP strategies are developed based on the assumption that the communication cost is in the form of $c_0 + c_1 \cdot x$ where c_0, c_1 are constants and x is the volume of data to be transmitted. Further, the fixed cost c_0 is a cost representing the overhead in setting up the communication channel, and is usually ignored. The objective of DQP becomes the minimization of the volume of transmitted data. This assumption may not be true or even close to the characteristic of future computer networks. In future local networks, data are likely to be transmitted in a store-and-forward fashion rather than through a shared bus. The transmission cost of each "hop" can be modeled as $c_0^{i,j} + c_1^{i,j} \cdot x$, where i and j are the end nodes of the hop, $c_0^{i,j}$ is the fixed cost for a transmission from node i to node j, and $c_1^{i,j}$ is the transmission cost per unit of data from node i to node j. The total communication cost for a transmission along the path $Node_1$, $Node_2$, ..., $Node_n$ can be modeled as

$$\sum_{i=1}^{n-1} \left(c_0^{i,i+1} + c_1^{i,i+1} \cdot x \right). \tag{5.1}$$

With high-speed communication technologies, such as optical fiber communication technology, the per unit cost of transmission between each pair

of nodes may be very low as compared to the cost of setting up a communication link, which involves circuit establishment, buffer reservation, queuing, etc. The fixed-link initiation cost is then dominating the communication cost. In this case, Equation 5.1 can be simplified as :

$$\sum_{i=1}^{n-1} \left(c_0^{i,i+1} \right). \tag{5.2}$$

Eq. 5.2 shows that the communication cost between any pair of nodes is only a fixed cost for each transmission session. All the DQP strategies being developed for solving multi-site joins, including semi-joins, are no longer adequate. New processing strategies are needed in this case. In this research, we propose a promising strategy as follows.

Algorithm DQPGTSP

(1) Set current site to the home site.

(2) Join all queried relations in the current site.

(3) Send the result to a site storing at least one queried relation that has not been joined. Set this site to be the next current site.

(4) Repeat steps 2, 3, and 4 until all queried relations are joined together.

(5) Send the result back to the home site.

An example is shown in Figure 5.1. A query entering site X wants to join $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$, and $\mathbf{R}_4$ together. $\mathbf{R}_1$ in site X is first sent to site Y to join with $\mathbf{R}_2$ and $\mathbf{R}_3$. The result is then sent to site Z to join with $\mathbf{R}_4$. Finally, the result is sent back to site X, and the execution terminates. The total communication cost is the summation of the cost incurred in all communication links used in algorithm DQPGTSP. We observe

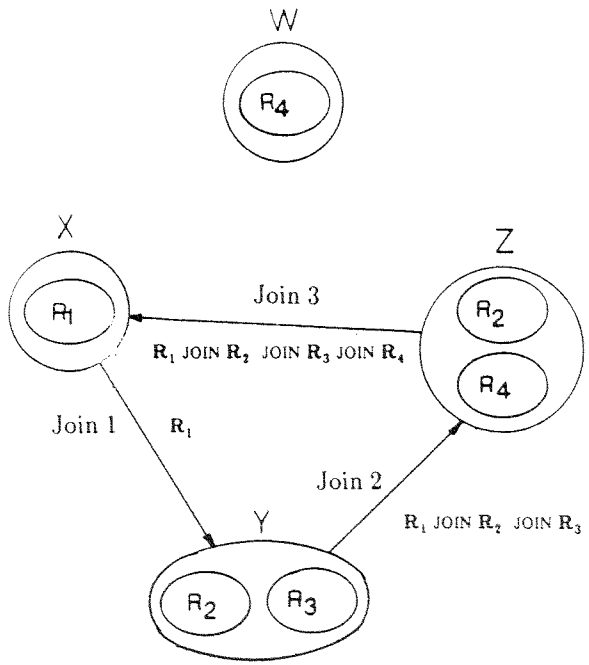


Figure 5.1 An example of Algorithm DQPGTSP.

(1) that each relation is accessed at least once;

(2) that each site is actived at most once except the home site; and

(3) that all the links used by the algorithm connect together as a closed loop.

To minimize the communication cost of Algorithm DQPGTSP, a sequence of connected links that access each relation at least once with the minimum cost should be found. If we cluster all sites storing a given file together, the problem can be viewed as finding a closed path connecting all clusters with the minimum cost. The example in Figure 5.2 shows the result of the clustering on the example in Figure 5.1. Cluster $\mathbf{R}_1$ consists of sites X and Y; Cluster $\mathbf{R}_2$ consists of sites Y and Z; Cluster $\mathbf{R}_3$ consists of site Y only; and Cluster $\mathbf{R}_4$ consists of sites W and Z. The problem can be modeled as the *generalized traveling salesman problems* (GTSP), which is described and solved in the rest of this chapter.

5.2 Introduction to GTSP

The *standard traveling-salesman problem* (TSP) can be stated as follows: given a set of cities and the distances (costs) between them, find a tour (a closed path passing through each city exactly once) with the minimum cost [DAN54]. In the *generalized traveling-salesman problem* (GTSP), the cities are partitioned into possibly intersecting clusters, and the objective is to find a tour that visits at least one node in each cluster with the minimum cost [SRI70] [KUM70].

Being applicable to solve many real world problems, the TSP have been studied for many years and many efficient heuristic algorithms have been developed. However, the structures of large real world problems tend to be hierarchical. In this case, GTSP will be a better model to represent them.

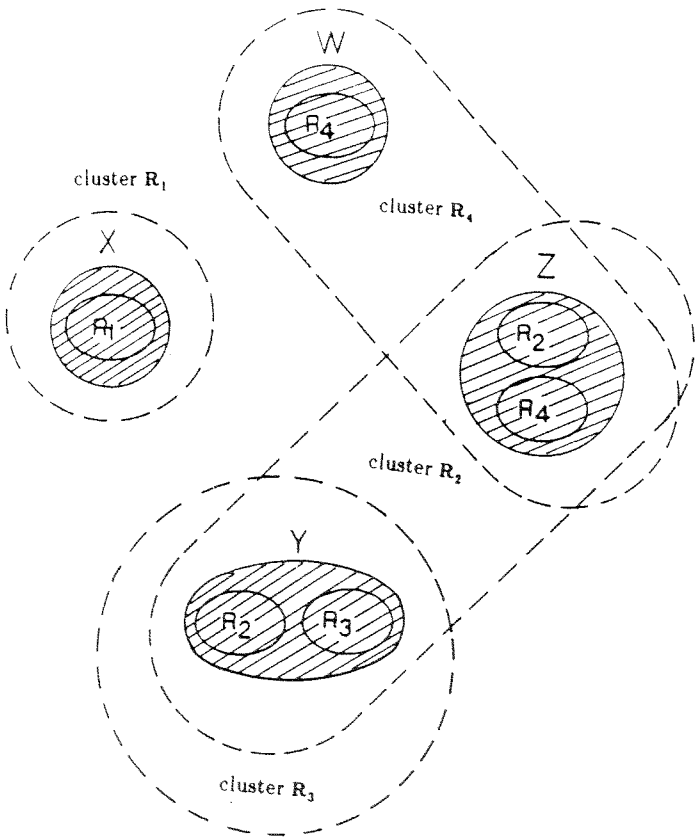


Figure 5.2 The clustering presentation of Figure 5.1.

For example, a traveling salesman wants to visit all dealers in each state of the country. Only one out of a number of possible cities in each state is chosen as the place to meet all local dealers in the state. To minimize the travel expenses, a route visiting a city of every state is to be chosen. This problem can be represented by a GTSP. As a second example, the post office wants to choose the locations of mailboxs for all residential areas (one mailbox for each residential area) as well as a route to collect mail from these mailboxes. A finite number of possible locations in each residential area are under consideration. This problem can also be represented as a GTSP. The third example is the construction of a ring type optical fiber communication network. One of several possible sites is to be selected as the concentrators for each service region. The network is to connect all the concentrators together as a ring. The cost of the construction is proportional to the total length of the network and is to be minimized. The forth example is a job sequencing problem in which some subtasks are to be cascaded in various ways to carry out a given task with the minimum cost. Each subtask can be carried out by various processes, and the cost of setting up a process depends on the preceding process. By viewing a subtask as a cluster, the process of carrying out a subtask as a city, and the cost of changing from one process to another as the distance between the two cities, the problem can be represented as a GTSP problem. The last example is a distributed query processing problem solved by the algorithm DQPGTSP in Section 5.1. As shown in Figure 5.2, each site storing a copy of queried file is a city; all sites storing a copy of a particular file are clustered into a cluster corresponding to that file; and the traveling cost between a pair of cities is the communication cost of the link connecting the pair of corresponding sites. The algorithm DQPGTSP becomes finding a tour that visits each cluster at least once with the minimum cost.

The problem instances of both GTSP and TSP can be represented as graphs. For any city in a problem instance, there is a node in the corresponding graph representation. The cost of traveling from one city to another is represented by a weighted arc connecting the nodes that represent the cities. Various approaches solving the GTSP are discussed in Section 5.3. Formal graph definition of the GTSP and some properties regarding its optimality are discussed in Section 5.4. A set of transformations solving the GTSP by the TSP-transformation approach are presented in Section 5.5. In Section 5.6, the properties of the transformed graph and the application of existing TSP algorithms to solve the GTSP are discussed. A set of GTSP instances are tested using the proposed algorithm. Finally, a summary is given in Section 5.7.

5.3 Solution Methods of GTSP

The TSP is a well known NP hard problem. It is easy to prove that the GTSP is also NP hard since the TSP is a special case of the GTSP. Three different possible approaches to solve the GTSP are discussed here. In the first approach, the *direct-solution* approach, solution algorithms are directly applied on the given problem instances. There are some standard techniques applicable for solving the problem optimally, such as dynamic programming [SRI70] [KUM70], and branch-and-bound algorithm [LIT63] [GUP78]. Heuristic algorithms that give suboptimal solutions are needed for large problem instances. However, the development of efficient and provable heuristics may involve considerable research efforts, since the problem is much more complex than the TSP, which has drawn extensive research for the past 30 years. More extensive research is needed before satisfactory solution algorithms can be

found. The following two approaches utilize existing results in the TSP to solve the GTSP. They may not be better than the direct-solution approach, but a number of available solution algorithms for the TSP can be immediately applied.

In the second approach, the *cartesian-decomposition* approach, a TSP solution is found for each member of the Cartesian product of the clusters. Each TSP solution is a feasible solution of the GTSP. Further, if the graph satisfies triangle inequality, then the TSP solution with the minimum cost provides an optimal solution to the GTSP. Assuming that there are m clusters with k_i nodes in cluster i, the total number of TSP's to be solved is $\prod_{i=1}^{m} k_i$. In the third approach, the *TSP-transformation* approach, the original problem is transformed into another problem such that the TSP solution to the transformed problem provides a solution to the GTSP of the original problem. In this chapter, we show a transformation procedure in which every node in a cluster of the original graph is expanded into three nodes. A single TSP with $3 \cdot \sum_{i=1}^{m} k_i + m$ nodes has to be solved.

In general, the Cartesian-decomposition approach has less complexity than the TSP-transformation approach when exponential-time algorithms are used to solve the standard TSP. On the other hand, the TSP-transformation approach is more efficient than the Cartesian-decomposition approach if polynomial-time approximation algorithms are used.

5.4 Graph Representation and Some Optimality Properties

Let $G = (N, A)$ be a directed graph. The set of nodes N are indexed by 0, 1, 2, ..., n. For each arc $(i, j) \in A$, there is a non-negative integer d(i, j) representing a cost measure of the arc. The costs satisfy the *triangle inequality;* that is, d(i, j) + d(j, k) $\geq$ d(i, k) for all i, j, k $\in$ {0, ..., n}. Nodes 1, 2, ..., n are partitioned into m possibly intersecting subsets (*clusters*) $C_1, C_2, \ldots, C_m$, with $k_i = |C_i| > 0$ nodes in subset C_i, and $k_1 + k_2 + \cdots + k_m \geq n$. For convenience, denote {0} as cluster C_0, $C = \{1,2,...,m\}$, and $C^* = \{0, 1,..,m\}$. An arc (i, j) is an *intercluster arc* if i and j are not in the same cluster; otherwise, it is an *intracluster arc*. A sequence of nodes $i_0, i_1, ..., i_{\ell'}, i_{\ell'+1}$ is called a *generalized tour (g-tour)* on G, when

(i) $i_0 = i_{\ell'+1} = 0$,

(ii) $(i_k, i_{k+1}) \in A$, for $k = 0, 1, ..., \ell'$, and

(iii) $\{i_1, i_2, ..., i_{\ell'}\} \cap C_k \neq \emptyset$, $k = 1, 2, ..., m$.

The cost Z of a tour is defined as

$$Z = \sum_{h=0}^{\ell'} d(i_h, i_{h+1})$$

The GTSP is to find a minimum-cost g-tour (optimal g-tour) on the graph G over all such g-tours on G. If $k_i = 1$, for $i = 1, ..., n$, and $C_i \cap C_j = \emptyset$, for $i, j \in \{1,...,n\}$, $i \neq j$, then the GTSP degenerates into a TSP.

Call a cluster that is (not) intersected with any other cluster in a graph as a *intersecting (non - intersecting) cluster*, and define the *covering cluster set* of a node as the set of clusters containing the node. We denote the covering cluster set of v_x as $S_x = \{\ell : v_x \in C_\ell\}$. Two nodes are *mutually exclusive* to each other (referring to as *mutually exclusive nodes*) if the covering cluster set of one node is

contained in that of the other. A g-tour *dominates* another g-tour if it visits fewer nodes with a cost no more than that of the other. A g-tour is *promising* if it is not dominated by any other g-tour on the graph, and is *non-promising* otherwise. By first excluding as many non-promising g-tours as possible out of the search space in a solution algorithm, the complexity will be greatly reduced. It is not practical trying to identify all such g-tours, because the problem is equivalent to that of finding all optimal g-tours connecting the minimum number of cities. From the definition of promising g-tours, we know there exists at least one promising optimal g-tour in each graph. Therefore, excluding all non-promising g-tours from the solution space will not eliminate all optimal g-tours, although an optimal g-tour may be non-promising. The following definitions and Theorem 5.1 are helpful in identifying some non-promising g-tours in terms of topological properties. A *canonic g-tour* is a g-tour visiting no mutually exclusive node. For example in Figure 5.3, C_1 is a non-intersecting cluster, while C_2 and C_3 intersect to each other. The covering cluster sets of v_1, v_2, and v_3 are {1}, {2}, and {3}, respectively, while the covering cluster set of v_4 is {2, 3}. The node v_4 is exclusive to both v_2 and v_3. G-tour $(v_1, v_2, v_4, v_3, v_1)$ is dominated by g-tour (v_1, v_2, v_3, v_1), and hence is non-promising. Further, it is not a canonic g-tour, because it visits mutually exclusive nodes, $\{v_2, v_4\}$ and $\{v_3, v_4\}$. The g-tour (v_1, v_2, v_3, v_1) is a canonic g-tour.

Since all nodes in a non-intersecting cluster are mutually exclusive to each other, a canonic g-tour visits such a cluster only once. A canonic g-tour on a graph without intersecting cluster is always promising, but it is not true if the graph contains intersecting clusters. Theorem 5.1 shows that a non-canonic g-tour is non-promising. For this reason, we are studying canonic g-tours only in this chapter.

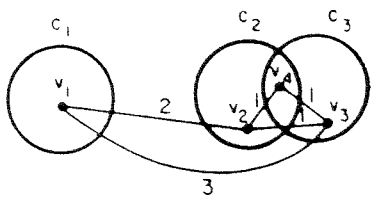


Figure 5.3 Some g-tours on a graph.

Theorem 5.1. For a graph satisfying the triangle inequality, any g-tour that visits at least two mutually exclusive nodes is dominated by another g-tour on the graph.

Proof: Let T be a g-tour that visits two mutually exclusive nodes v_x and v_y, with $S_y \subseteq S_x$. We can construct another g-tour T' with no additional cost which visits the same set of nodes visited by T except v_y. With no loss of generality, let $T=(t_1, v_1, v_x, v_2, t_2, v_3, v_y, v_4)$, as shown in Figure 5.4, where v_1 and v_2 (resp. v_3 and v_4) are the nodes connected to node v_x (resp. v_y), and t_1 (resp. t_2) is the path connecting v_1 and v_4 (resp. v_2 and v_3). A closed path T' $(t_1, v_1, v_x, v_2, t_2, v_3, v_4)$ as shown in Figure 5.5 can be constructed. T' is a g-tour since all clusters visited by T are visited by T'. The cost of T' is no more than that of T, since $d(v_3, v_4) \leq d(v_3, v_y) + d(v_y, v_4)$. Therefore, T is dominated by T'. □

The following corollary states that a g-tour visiting more than one node of a non-intersecting cluster is non-promising.

Corollary 5.1. For a graph satisfying the triangle inequality, any g-tour that visits more than one node of a non-intersecting cluster is dominated by another g-tour that visits only one node of the cluster.

Proof: Since the covering cluster sets of all nodes in a non-intersecting cluster are all the same, by Theorem 5.1, such a g-tour is non-promising. □

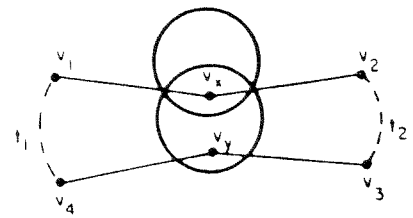


Figure 5.4 A g-tour T containing mutually exclusive nodes (v_x and v_y).

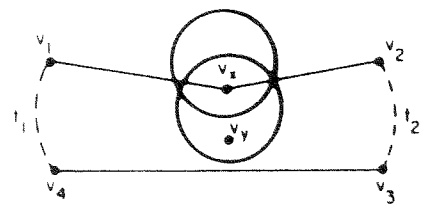


Figure 5.5 A g-tour T' that dominates g-tour T in Fig. 5.4.

5.5 TSP-Transformation Approach

In this section, the GTSP is solved by the TSP-transformation approach. A transformation for graphs with non-intersecting clusters is presented in Section 5.5.1. Another set of transformations are developed in Section 5.5.2 to solve the GTSP on graphs with intersecting clusters. In Section 5.6, heuristic algorithms for TSP, which are applicable to the transformed problem, are discussed and are evaluated on some test cases.

5.5.1 G-tours on Graphs with Non-intersecting Clusters

We develop a *G-S transformation* that transforms a graph G with non-intersecting clusters into another graph $G' = (N', A')$, such that any tour on G' is a canonic g-tour on G with the same cost and vice-versa. Denote the set of nodes in C_i by $v_{i,0}, v_{i,1}, \ldots, v_{i,k_i-1}$, for $i = 1, \ldots, m$. The set of nodes N' in G' is obtained as follows.

(i) For each node $v_{i,j}$ in G, create three nodes $i_{i,j}$ (i-node), $b_{i,j}$ (b-node) and $o_{i,j}$ (o-node) in G'.

(ii) For node 0 in G, create two nodes $i_{0,0}$ and $o_{0,0}$ in G'.

(iii) For each cluster C_i in G, create a node g_i (g-node) in G'.

Let C_i' denote the set of nodes $\{i_{i,0}, b_{i,0}, o_{i,0}, \ldots, i_{i,k_i-1}, b_{i,k_i-1}, o_{i,k_i-1}, g_i\}$ in G' created for the set of nodes in C_i. $C' = \{C_i', i \in C\}$ are the clusters in G'. We also define the set $\{i_{0,0}, o_{0,0}\}$ as C_0', and $C'' = C_0' \cup C'$. The set A' contains the following arcs:

(iv) For the i-nodes, b-nodes and o-nodes in cluster C_i', they are connected into a cycle with arcs: $\{(o_{i,j}, b_{i,j}), (b_{i,j}, i_{i,j}), (i_{i,j}, o_{i,(j+1) \bmod k_i}) : j = 0, 1, \ldots, k_i - 1\}$. These arcs have zero cost.

(v) The g-node in cluster C_i' is connected by arcs: $\{(i_{i,j}, g_i), (g_i, b_{i,j}) : j = 0, 1, ..., k_i - 1\}$. Additional arcs are created between the b-nodes and o-nodes: $\{(b_{i,j}, o_{i,j}) : j = 0, 1, ..., k_i - 1\}$. These arcs have zero cost.

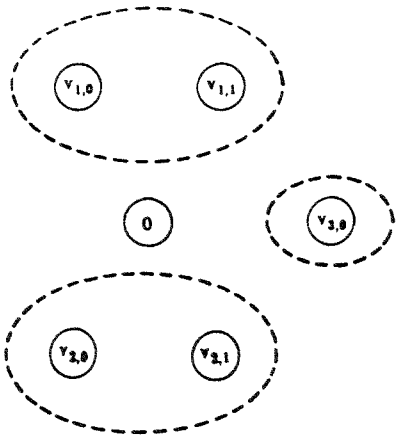
(vi) An arc $(i_{0,0}, o_{0,0})$ with zero cost is created for cluster 0.

(vii) For an intercluster arc $(v_{i_1,j_1}, v_{i_2,j_2}) \in A$, $0 \le i_1, i_2 \le m$, $i_1 \ne i_2$, $0 \le j_1 \le k_{i_1} - 1$, $0 \le j_2 \le k_{i_2} - 1$, there is a corresponding intercluster arc $(o_{i_1,j_1}, i_{i_2,j_2})$ in A' with the same cost. That is, a cluster can only be entered through i-nodes and departed through o-nodes.

An example graph G is shown in Figure 5.6. There are 24 *g-tours*. The optimal *g-tour* is $(0, v_{1,1}, v_{2,0}, v_{3,0}, 0)$ with cost 12. The arcs added to G' in Steps (iv), (v) and (vi) are shown in Figure 5.7. Actually, *single-node clusters*, that contain only one node, need not to be expanded.

The costs of intercluster arcs created by step (vii) are the same as the corresponding arcs in G, while the other arcs are intracluster arcs with zero costs. Therefore, the cost of tours on G' consists of the costs of intercluster arcs only. Any tour on G' can only visit clusters in the following restricted way: when visiting a cluster, a tour first visits an arbitrary i-node $i_{i,j}$, then visits all other nodes in the cluster, and leaves the cluster through the o-node $o_{i,j}$. If intracluster arcs are taken away from a tour, then the remaining arcs represent a canonic g-tour on G. This is shown in the following theorems.

Theorem 5.2: Every tour on G' visits the nodes in C_i', $i \in \{1, 2, ..., m\}$, using a subtour of the form: $(i_{i,j}, o_{i,(j+1) \bmod k_i}, b_{i,(j+1) \bmod k_i}$,



cost matrix

	0	$v_{1,0}$	$v_{1,1}$	$v_{2,0}$	$v_{2,1}$	$v_{3,0}$
0		3	2	4	6	5
$v_{1,0}$	2			6	5	6
$v_{1,1}$	2			2	7	4
$v_{2,0}$	6	7	4			3
$v_{2,1}$	6	4	6			7
$v_{3,0}$	5	8	6	8	4	

Figure 5.6 Clustered graph G with non-intersecting clusters.

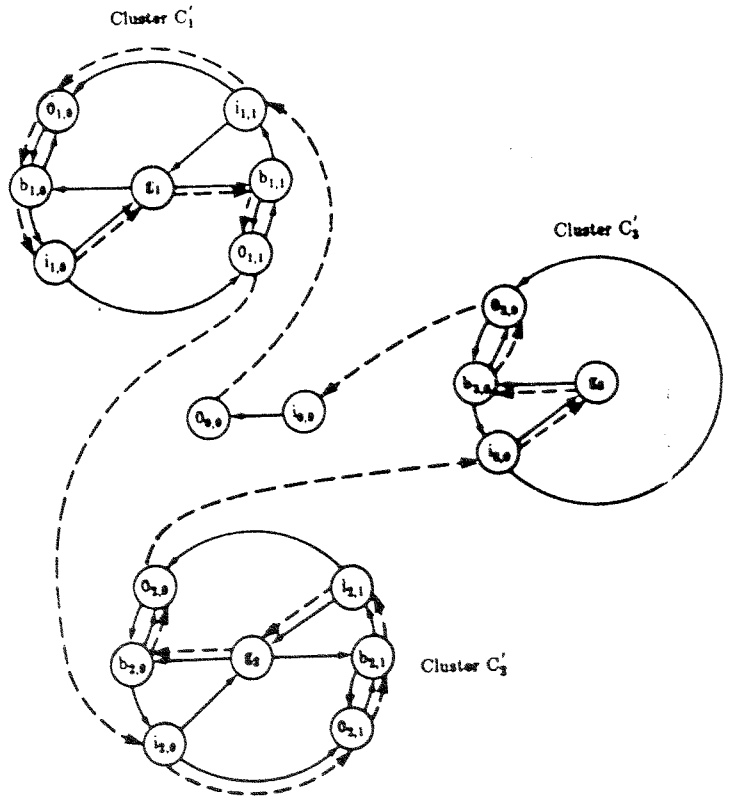


Figure 5.7 Arcs added to G' in Steps (iv), (v) and (vi) of the construction process.

$i_{t,(j+1) \bmod k_t}$, ..., $o_{t,(j+k_t-1) \bmod k_t}$, $b_{t,(j+k_t-1) \bmod k_t}$, g_t, $b_{t,j}$, $o_{t,j}$) for $j \in \{0, ..., k_t - 1\}$.

Proof: If i_{t,j_1} is the first node in cluster C_t' that is visited in the tour, then the theorem asserts that all the other nodes in cluster C_t' would be visited before the tour exits through o_{t,j_1}. The existence of such a subtour is obvious from the construction of G'. To show that the subtour is unique, suppose the theorem is false, and the tour exits through o_{t,j_2}, $j_2 \neq j_1$, then a contradiction can be derived. Referring to Figure 5.8 , this implies that b_{t,j_2} cannot be entered through o_{t,j_2} and must be entered through g_t. After traversing node o_{t,j_2}, the tour can either exit the cluster or visit node b_{t,j_1}. In the first case, b_{t,j_1} cannot be visited from o_{t,j_1}, and must be visited through g_t. Node g_t is, thus, visited twice. In the second case, the tour cannot exit without visiting nodes i_{t,j_1} or o_{t,j_1} again. These imply that the tour must exit through o_{t,j_1}, and the theorem is proved. □

Let $\sum_m$ denotes the set of all the permutations of $\{1, 2, ..., m\}$. The following three theorems prove the equivalence between the tours on G' and the canonic g-tours on G.

Theorem 5.3: For every canonic g-tour on $G = (N, A)$, there is a tour on $G' = (N', A')$ with $Z' = Z$.

Proof: Let $T = (0, v_{i_1^*,j_1^*}, v_{i_2^*,j_2^*}, ..., v_{i_m^*,j_m^*}, 0)$ be any given canonic g-tour on G where $(i_1^*, i_2^*, ..., i_m^*) \in \sum_m$ and $j_l^* \in \{0, 1, ..., k_{i_l^*} - 1\}$ for $l = 1, ..., m$. A

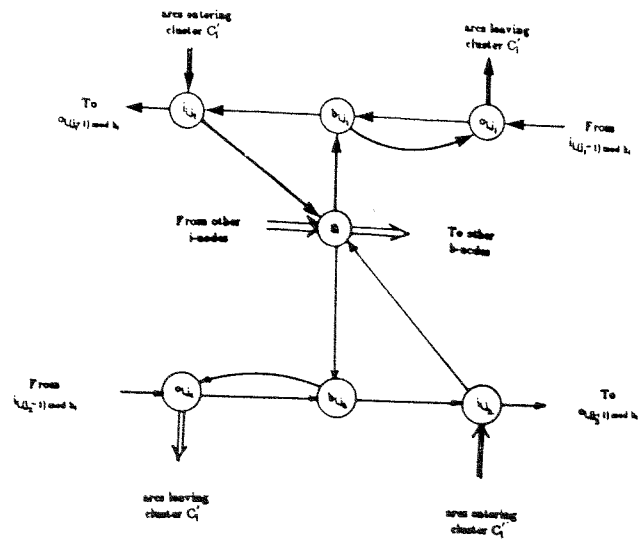


Figure 5.8 Proof of Theorem 2.

tour $T' = (i_{0,0}, o_{0,0}, t_{i_1}^*, t_{i_2}^*, \ldots, t_{i_m}^*, i_{0,0})$ on G' can be constructed from T by having $t_{i_p}^* = (i_{i_p,j_p}^*, o_{i_p,(j_p^*+1) \bmod k_{i_p}}^*, b_{i_p,(j_p^*+1) \bmod k_{i_p}}^*, i_{i_p,(j_p^*+1) \bmod k_{i_p}}^*, \ldots, o_{i_p,(j_p^*+k_{i_p}-1) \bmod k_{i_p}}^*, b_{i_p,(j_p^*+k_{i_p}-1) \bmod k_{i_p}}^*, i_{i_p,(j_p^*+k_{i_p}-1) \bmod k_{i_p}}^*, g_{i_p}^*, b_{i_p,j_p}^*, o_{i_p,j_p}^*)$, for $p = 1, 2 \ldots, m$. It is easily seen that all the arcs required to construct the tour T' in the above sequence are present in G', and the cost Z' of T' is the same as the cost Z of T. □

Theorem 5.4: For every tour on G', there is a canonic g-tour on G with $Z = Z'$.

Proof: Let $T' = (i_{0,0}, o_{0,0}, t_{i_1}^*, t_{i_2}^*, \ldots, t_{i_m}^*, i_{0,0})$ be any given tour on G', where $(i_1^*, i_2^*, \ldots, i_m^*) \in \sum_m$ and t_p^* is defined as in the proof of Theorem 5.3. A canonic g-tour T can be constructed on G by visiting node $v_{i_p^*, j_p^*}$ of C_i for $p = 1, 2, \ldots, m$; that is, $T = (0, v_{i_1^*,j_1^*}, v_{i_2^*,j_2^*}, \ldots, v_{i_m^*,j_m^*}, 0)$. It is easily seen that such a tour can be constructed and $Z = Z'$. □

Theorem 5.5: If a tour T' is an optimal tour on G', then the corresponding canonic g-tour T obtained in Theorem 5.4 is an optimal g-tour on G.

Proof: Assume that T is not an optimal g-tour, then there is a canonic g-tour τ with smaller cost on G. By Theorem 5.3, we can construct a tour τ' on G' with the same cost as τ, which contradicts the assumption that T' is the tour with the minimum cost on G'. □

As an example, the corresponding optimal tour of G' for the graph in Figure 5.6 is shown as dashed lines in Figure 5.7.

5.5.2 G-tours on Graphs with Intersecting Clusters

For graphs with intersecting clusters, it is not easy to obtain a transformation which provides a one-to-one correspondence between canonic g-tours on a given graph and the tours on the transformed graph. The transformation developed in this section can only guarantee that the optimal g-tours on a graph can be obtained from the optimal tours on its transformed graph and vice versa. A tour obtained from the transformed graph may represent a non-canonic or non-feasible g-tour on G. However, another *canonical transformation* can transform such a g-tour into a canonic g-tour. The transformation for the GTSP with intersecting clusters consists of two stages. First, an *I-N transformation* transforms the original graph into an intermediate graph (*I-N graph*) with non-intersecting clusters. Next, the I-N graph is transformed into a third graph (*G-S graph*) by the G-S transformation developed in Section 5.5. The G-S graph is then solved as a TSP. The I-N transformation that transforms the original graph $G = (N, A)$ into the I-N graph $G'' = (N'', A'')$ with non-intersecting clusters $C_0'', C_1'', \ldots, C_m''$ is described as follows.

In a g-tour on G, suppose the tour passes through a node v_x whose covering cluster set is $S_x = \{i : v_x \in C_i\}$, i.e., $v_x \in \bigcap_{i \in S_x} C_i$. In G'', v_x is expanded into $|S_x|$ nodes, referred to as the *expanded nodes* of v_x, one for each cluster in the covering cluster set. A g-tour that visits v_x in G implies that the corresponding g-tour in G'' would visit all the nodes expanded from v_x.

With $N'' = \emptyset$ and $A'' = \emptyset$ initially, the I-N graph $G'' = (N'', A'')$ is constructed as follows:

(i) For every node v_x in G, it is expanded into $|S_x|$ nodes in G''. $N'' = N'' \cup \{v_{i,x} : i \in S_x\}$. Notice that $|S_x| \geq 1$. Denote the expanded nodes of v_x as E_x, i.e., $E_x = \{v_{i,x} : v_x \in C_i\}$.

(ii) For any two nodes $v_{i,x}, v_{j,x}$, $i, j \in S_x$ in G'', create two arcs $A'' = A'' \cup \{(v_{i,x}, v_{j,x}), (v_{j,x}, v_{i,x})\}$ with $d(v_{i,x}, v_{j,x}) = d(v_{j,x}, v_{i,x}) = 0$.

(iii) For every arc $(v_x, v_y) \in A$, let $A'' = A'' \cup \{(v_{i,x}, v_{j,y}) : i \in S_x, j \in S_y\}$ and $d(v_{i,x}, v_{j,y}) = d(v_x, v_y)$.

An example is shown in Figure 5.9 . Figure 5.9(a) is part of G. Its corresponding part on G'' is shown in Figure 5.9(b).

Essentially, every node in the intersection of multiple clusters is expanded such that there is one expanded node for each cluster containing the node. The cost on the arcs among the expanded nodes is zero. The arcs directed to and emanating from the original node are duplicated for all the expanded nodes.

Define the *internal tour* corresponding to a node v_x to be the path connecting consecutively all the expanded nodes of v_x in G''. The cost of an internal tour is zero since the cost of traversing the expanded nodes is zero. The follows theorems show the equivalence between the optimal g-tours on G and G''.

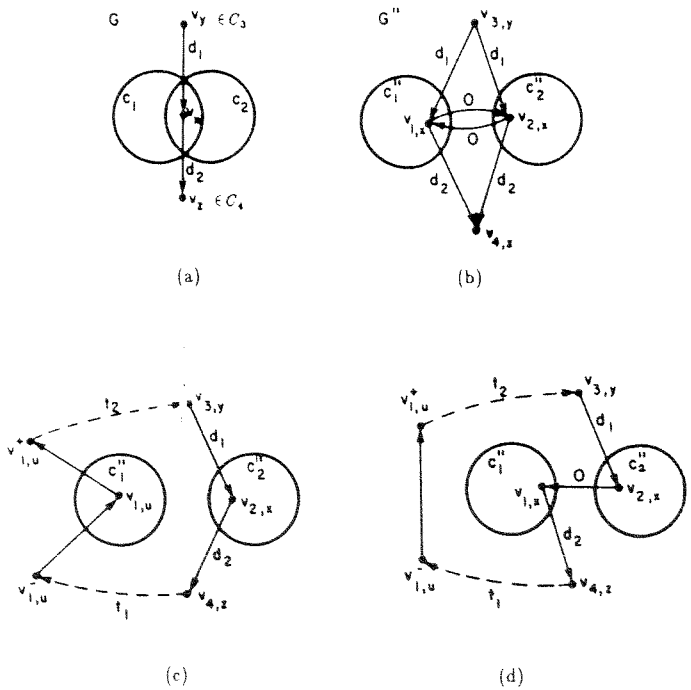


Figure 5.9 An example of I-N transformation;
(a) a part of graph G; (b) the I-N graph G'' of G;
(c) a g-tour T'' on G''; (d) a better g-tour T'' on G''.

Theorem 5.6. For every g-tour on G, there is a g-tour on the I-N graph G'' with $Z'' = Z$.

Proof: Given a g-tour T on G, we can obtain an equivalent g-tour T'' on G'' as following: For every node v_z visited on G, an internal tour can be created on G'' that visits all nodes $v_{i,z}$ in E_z. For an arc (v_z, v_y) in T, the arc $(v_{i,z}, v_{j,y})$ is contained in T'', where $v_{i,z}$ is the last node of the internal tour connecting the nodes in E_z, and $v_{j,y}$ is the first node of the internal tour connecting the nodes in Ey. It is obvious that T'' is a g-tour of G'' with cost $Z'' = Z$. □

Though every g-tour on G has an equivalent g-tour on G'', there is no such correspondence from G'' to G except for the optimal g-tours. This is shown in the next theorem. Denote the preceding (succeeding) node of a node $v_{i,z}$ in a g-tour T'' as $v^-_{i,z}$ ($v^+_{i,z}$).

Theorem 5.7. For every optimal g-tour on the I-N graph G'', there is an optimal g-tour on the original graph G with $Z = Z''$.

Proof: Let T'' be any promising optimal g-tour in G''. By Corollary 5.1 in Section 5.4, T'' visits each cluster exactly once. Next, we show that from T'', we can always construct another optimal g-tour $\bar{T}''$, such that for any E_z in G'', if T'' visits any node in E_z, then $\bar{T}''$ will visit all nodes in E_z consecutively. Without loss of generality, let $T'' = (v^-_{2,z}, v_{2,z}, v^+_{2,z}, t_1, v^-_{1,u}, v_{1,u}, v^+_{1,u}, t_2)$, where t_1 (resp. t_2) is a path connecting $v^+_{2,z}$ and $v^-_{1,u}$ (resp. $v^+_{1,u}$ and $v^-_{2,u}$). T'' visits $v_{2,z}$ in cluster C_2'', but

visits $v_{1,u}$ instead of $v_{1,z}$ in cluster C_1'', where $C_1'', C_2'' \in S_z$. The node $v_{1,u}$ may or may not be $v_{1,z}$. The g-tour T'' is depicted in Figure 5.9(c). The new g-tour $\bar{T}''$ can be constructed by visiting $v_{1,z}$ immediately after visiting $v_{2,z}$. The arc $(v_{2,z}, v_{2,z}^{\tau})$ is replaced by the arc $(v_{1,z}, v_{2,z}^{+})$, and the arcs $(v_{1,u}^{-}, v_{1,u})$ and $(v_{1,u}, v_{1,u}^{+})$ are replaced by the arc $(v_{1,u}^{-}, v_{1,u}^{\tau})$. Since $d(v_{1,u}^{-}, v_{1,u}^{+}) \leq d(v_{1,u}^{-}, v_{1,u}) + d(v_{1,u}, v_{1,u}^{+})$, $d(v_{2,z}, v_{1,z}) = 0$, and $d(v_{1,z}, v_{2,z}^{+}) = d(v_{2,z}, v_{2,z}^{+})$, the cost of $\bar{T}''$ is no more than that of T''. Further, since T'' is optimal, the cost of T'' and $\bar{T}''$ have to be equal. Therefore, every optimal g-tour T'' on G'' can be reconstructed to another optimal g-tour $\bar{T}''$ in G'' which visit all nodes expanded from the same node in G consecutively. From $\bar{T}''$, it can easily be seen that a corresponding g-tour T on G can be constructed (by grouping all the nodes in an internal tour in $\bar{T}''$) with the cost $Z = Z''$.

Next, we prove that this g-tour must be optimal on G. Suppose that it were not optimal, then this implies that there is another g-tour on G with cost less than Z. By Theorem 5.6, there exits a g-tour on G'' with cost less than Z. This contradict the assumption that the g-tour T'' on G'' is an optimal g-tour. □

From Theorem 5.6 and 5.7, the optimal g-tours on a graph with intersecting clusters can be obtained from the optimal g-tours on its I-N graph. The optimal g-tours on the I-N graph, which contains non-intersecting clusters only, can be obtained from the solution methods developed in Section 5.5. However, the optimal TSP tours may not be easy to obtain if the problem size is large. We should be able to transform any TSP tour on the G-S graph into a feasible g-tour or a canonic g-tour on the original graph, such that any heuristic TSP solution to the G-S graph provides a heuristic GTSP solution to the original graph. From the theorems in Section 5.5.1, we know every TSP tour

on the G-S graph represents a g-tour on the I-N graph. So we only need to transform each g-tour on the I-N graph into a canonic g-tour on the original graph.

Given a graph G and its I-N graph G'', an arbitrary g-tour T'' on G'' may visit the expanded nodes of a node v_z of G in the following ways.

(1) None of the expanded nodes is visited by T''.

(2) All expanded nodes are visited by T'' consecutively. That is, they are connected as an internal tour.

(3) The expanded nodes are visited by T'' in more than one non-consecutive sub-tours. In this case, T'' represents a g-tour T in G visiting v_z more than once.

(4) Some but not all of the expanded nodes of v_z are not visited by T''.

(5) A combination of cases (3) and (4).

Some examples are shown in Figure 5.10 to Figure 5.13. A node v_z in a graph G and its expanded nodes in the I-N graph G'' are shown in Figure 5.10. An example of T'' in G, which represents a g-tour visiting v_z more than once (case (3)), is shown in Figure 5.11. Another example of T'', which visits only part of the expanded nodes of v_z (case (4)), is shown in Figure 5.12. While T'' in Figure 5.13 is an example of case (4), sub-tours t_1, t_2, and t_3 in each figure are subtours. The equivalent sub-tours on G and G'' carry the same symbols.

T'' should be reconstructed except that all expanded nodes are visited by either case (1) or case (2). A *canonic transformation* algorithm *CAN_TRANS* to be described first transforms a g-tour T'' on G'' into another g-tour, which represens a feasible g-tour in G, then transforms any g-tour in G into a canonic g-tour. Given a g-tour T'' in the I-N graph G'', which is transformed from an

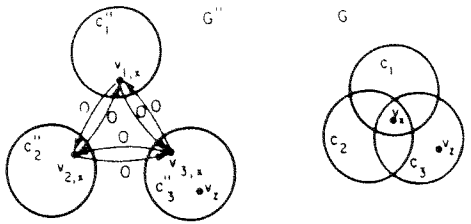


Figure 5.10 A graph G and its I-N graph G''.

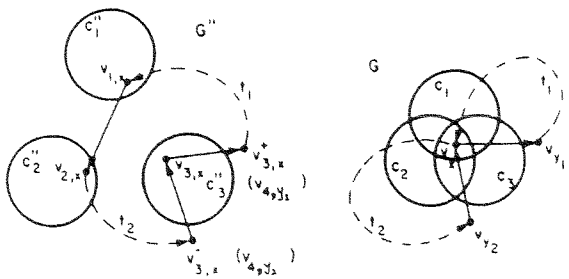


Figure 5.11 A tour on G'' that visit v_z on G twice.

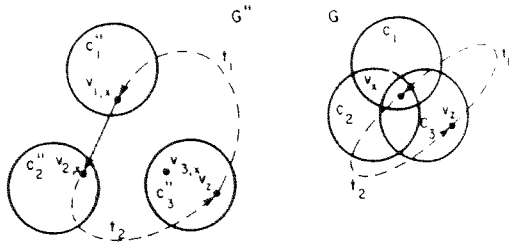


Figure 5.12 A tour on G'' that does not visit all expanded nodes of v_z.

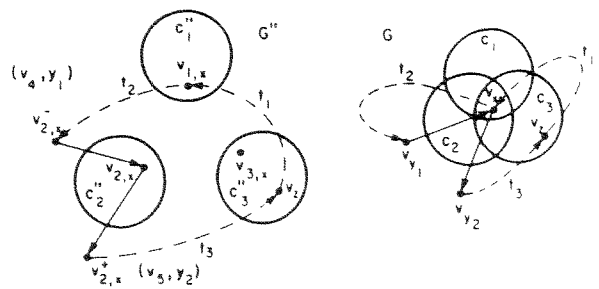


Figure 5.13 A tour on G'' that is a combination of Case (3) and (4).

original graph G, another g-tour in G'' representing a feasible g-tour in G is first reconstructed by applying steps (1) on T''. Next, the new g-tour is transformed into a canonic g-tour by applying step (2) on every pair of mutual exclusive nodes. The algorithm is shown as follows.

Algorithm CAN-TRANS

(1) For each node v_x with expanded nodes E_x in G'',

(1.a) If its expanded nodes in G'' are visited by T'' in either case (1) or case (2), then do nothing;
otherwise, execute steps (1.b) and (1.c).

(1.b) Assuming that the expanded nodes that are contained in T'' forms several pieces of possibly zero length (a single node) disconnected paths, called *internal paths*,
$(v_{i_{1},x}, \ldots, v_{i_{2},x})$, $(v_{i_{3},x}, \ldots, v_{i_{4},x})$, $\ldots$, $(v_{i_{q},x}, \ldots, v_{i_{q+1},x})$, do following.

Construct the internal tour T_x that connects all nodes in E_x in an arbitrary order except the first node and the last node. There are three different cases in the selection of the first node and the last node of T_x:

Case (a) : If at least one internal path has non-zero length, then select the first node (resp. last node) of such an internal path to be the first node (resp. last node) of T_x. If more than one alternative exists, then select the pair of nodes, say $v_{i_{1},x}$ and $v_{i_{2},x}$, such that the value of $d(v^{-}_{i_{1},x}, v_{i_{1},x}) + d(v_{i_{1},x}, v^{+}_{i_{1},x})$ is the minimum among all alternatives.

Case (b) : If the lengths of all internal paths are zero, i.e., they are $(v_{i_{1},x}), (v_{i_{2},x}), \ldots, (v_{i_{q},x})$, then the selection is the same as Case (a) but the first node and the last node should belong to two different internal paths. Assuming that the first node is $v_{i_{1},x}$, and the last node is one of $\{v_{i_{3},x}, \ldots, v_{i_{q},x}\}$, say $v_{i_{3},x}$, take the arc $(v_{i_{3},x}, v^{+}_{i_{3},x})$ to emanate from the internal path instead of arc $(v_{i_{1},x}, v^{+}_{i_{1},x})$.

Case (c) : If only one expanded node in E_x, say $v_{i_{1},x}$, is visited by T'', then select $v_{i_{1},x}$ as the first node and another arbitrary expanded node of v_x as the last node, say $v_{i_{2},x}$. Take the arc $(v_{i_{2},x}, v^{+}_{i_{2},x})$ to emanate from the internal path, and step (1.c) is skipped.

(1.c) Replace the arcs directed to and emanated from the paths $(v_{i_{3},x}, \ldots, v_{i_{4},x}), \ldots, (v_{i_{q},x}, \ldots, v_{i_{q+1},x})$ with the arcs $(v^{-}_{i_{3},x}, v^{+}_{i_{3},x}), \ldots, (v^{-}_{i_{q},x}, v^{+}_{i_{q+1},x})$, i.e., connect the first node and the last node of each internal path together.

(2) For each pair or mutual exclusive nodes v_x and v_y do followings.
If the covering cluster set of v_y is contained in the covering set of v_x, then bypass node v_y by connecting the preceding node to the succeeding node of v_y.

For example, the new g-tours of the g-tours in Figure 5.11 and Figure 5.13 are shown in Figure 5.14 and Figure 5.15 respectively. The cost of the resulting g-tour depends on the strategies in interrelated steps of algorithm CAN-TRANS. The best solution can only be made by comparing all possible alternatives. Ad hoc strategies are preferable to an optimal one since this procedure itself is a part of a heuristic algorithm.

Although the arcs of I-N graph does not satisfy the triangle inequality (which is discussed in next section), the canonical transformation will guarantee that the cost of the resulting g-tour will not be more than that of the original g-

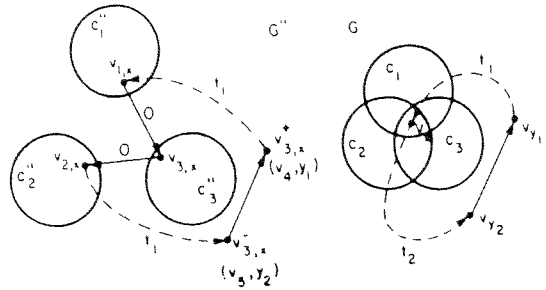


Figure 5.14 The new g-tour of T'' in Fig. 5.11 after canonical transformation.

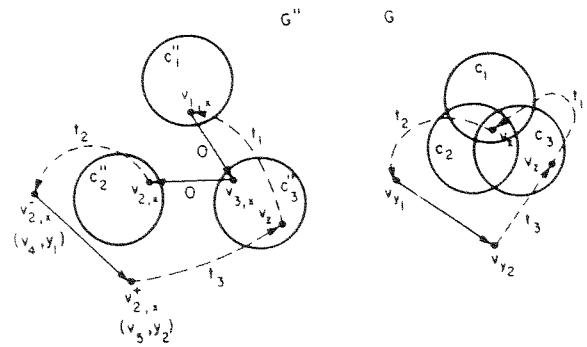


Figure 5.15 The new g-tour of T'' in Fig. 5.13 after canonical transformation.

tour as long as the arcs of the original graph satisfies the triangle inequality. This is proved in the next theorem.

Theorem 5.8. If the arcs of a graph G satisfy the triangle inequality, then the canonical transformation transforms any g-tour T'' on the I-N graph G'' into another g-tour T in G with equal or less cost.

Proof: Since arcs in G satisfy the triangle inequality, step (2) of Algorithm CAN-TRANS will transform a g-tour into another g-tour with no more cost. We only need to show that steps (1.b) and (1.c) transform a g-tour on G'' into another with no more cost (step (1.a) will not change T''). Suppose v_z is the node under consideration, and the expanded nodes of v_z that are contained in T'' are connected as the following internal paths, $(v_{i_1,z}, \ldots, v_{i_2,z})$, $(v_{i_3,z}, \ldots, v_{i_4,z})$,..., $(v_{i_\varrho,z}, \ldots, v_{i_\varrho+1,z})$.

Case (a), (b), and (c) in the following refer to Case (a), (b), and (c) respectively in step (1.b) of Algorithm CAN-TRANS. First, if they are in Case (a), then the arcs $(v^-_{i_1,z}, v_{i_1,z})$ and $(v_{i_2,z}, v^+_{i_2,z})$ are kept unchanged in the new T''. However, the arcs $(v^-_{i_3,z}, v_{i_3,z})$ and $(v_{i_4,z}, v^+_{i_3,z})$ are replaced by $(v^-_{i_3,z}, v^+_{i_4,z})$, the arcs $(v^-_{i_\varrho,z}, v_{i_\varrho,z})$ and $(v_{i_\varrho+1,z}, v^+_{i_\varrho+1,z})$ are replaced by $(v^-_{i_\varrho,z}, v^+_{i_\varrho+1,z})$, and so on. The replaced arcs are all existing arcs in G, so they should satisfy the triangle inequality. Each replaced arc is either an existing arc in G or a new one with zero cost. The replacement will not increase the cost of T'' in both cases. Further, the cost of any internal tour is zero. Therefore, the cost of the new g-tour is not increased. Second, if they are in Case (b), then the arc $(v^-_{i_1,z}, v_{i_1,z})$ is kept unchanged in the new T'' and the cost of $(v_{i_1,z}, v^+_{i_1,z})$ is

the same as the cost of $(v_{i_{1,2}}, v^{-}_{i_{1,2}})$. The replacement of other arcs will not increase the cost for the same argument as in Case (a). So, the cost of the new g-tour is not increased. Third, if they are in Case (c), then the arc $(v^{-}_{i_{1,2}}, v_{i_{1,2}})$ is kept unchanged in T'', and the arc $(v_{i_{1,2}}, v^{+}_{i_{1,2}})$ is replaced by the arc $(v_{i_{2,1}}, v^{+}_{i_{1,2}})$, which has the same cost as that of $(v_{i_{1,2}}, v^{+}_{i_{1,2}})$. Therefore, the cost of the new g-tour is the same as that of the old one. □

In summary, a transformation consisting of two transformational phases transforms the GTSP on a graph with intersecting clusters into the TSP of another graph. The optimal GTSP solution to the original graph can be obtained from the optimal TSP solution to the transformed graph. The canonic transformation transforms any TSP tour on the transformed graph back to a canonic g-tour on the original graph, such that the heuristic TSP algorithms solves the GTSP.

5.6 Solutions For The Transformed Graph

In this section, we examine the properties of the transformed graphs (G-S graph) and suitable heuristic algorithms for solving TSP on these graphs. The properties of a G-S graph are as follows.

(1) The graph is asymmetric.

(2) The arcs do not satisfy the triangle inequality.

(3) The graph is not a complete graph.

The arcs do not satisfy the triangle inequality since there exists arcs with zero cost. The graph is not complete since certain arcs are omitted to force the TSP tour to be a feasible and canonical g-tour on the original graph. However, the graph can be transformed into a complete graph by adding dummy arcs with a very large cost. Heuristic TSP algorithms applicable only to complete graphs can then be applied to the resulting graphs.

Different approaches to solve the TSP optimally have been studied extensively in the literature [BEL68] [BUR79] [GUP68]. These include (a) integer programming [DAN54] [MIL76] [MIL78], (b) dynamic programming [BEL62] [HEL62] [GON62], and (c) branch-and-bound [LIT63]. However, since the TSP is NP-hard, it is difficult to apply these algorithms to solve the GTSP optimally. For this reason, it is more realistic to apply polynomial-time heuristics for the TSP to solve the GTSP.

Over the past fifteen years, a large number of polynomial-time heuristics have been proposed to solve the TSP. These heuristics do not guarantee an optimal solution but seem likely to lead to "good solutions." These heuristics can be classified into the following categories.

(1) Tour building : Starting from any node, a tour is built by successively including other nodes into the tour. *Nearest neighbor* [BEL68], and *insertion* (arbitrary insertion, nearest insertion, cheapest insertion, farthest insertion) [KAR64] are two examples.

(2) Tour-to-tour improvement : Starting from a tour, better tours are obtained by changing the order of some nodes iteratively. An example is the *K-exchange* [LIN65] [CHR72].

(3) Partitioning and decomposition: A large problem is decomposed into a series of subproblems first, then each subproblem is solved independently. *Strip* is such an example [BEA59] [BEN83].

(4) Relaxation: The TSP is relaxed into another problem, such as the minimum spanning tree [HEL70] [HEL71] [HON76], optimal assignment problem [KAR77] [KAR79] [GIL64], n-path [HOU80], and matching problem [BEL71], then the solution is transformed into a TSP tour.

(5) Others: This category includes the longest-path-problem transformation [HAR62] [PAN64], analog computer approach [CLA78], etc.

Given a heuristic H, let $R_n(H)$ be the ratio of the obtained tour length to the minimal tour length. For the TSP unconstrained by the triangle inequality, for any $k > 1$, the problem of finding a tour with R_n of length bounded by k is NP-complete [GAR79]. As a result, it is unlikely that polynomial-time heuristic with constant upper bound on $R_n(H)$ exists. However, for the asymmetric TSP unconstrained by the triangle inequality, Karp [KAR79] gave a polynomial-time heuristics with complexity $O(n^3)$. If the distances of the graph are drawn independently from a uniform distribution, with probability tending to 1, this heuristic has $R_n < 1 + \epsilon(n)$, where $\epsilon(n)$ goes to zero as $n \to \infty$.

Although the distance of the transformed graph of the GTSP do not satisfy the above assumption, the algorithm still work well for the transformed graph. We have evaluated this algorithm on a set of test problem instances generated by randomly clustering cities in a map of the Unite States, which is the same as the 33-city problem in Karg's paper [KAR64]. In Karp's algorithm [KAR80], the problem is first solved as an assignment problem. The resulting assignment may contain some cycles. A patching algorithm is then applied on these cycles to produce a feasible TSP tour. The patching algorithm iteratively selects two cycles and replaces an arc from each cycle with a pair of arcs that connect the two cycles together. However, since the transformed graph is not complete, the patching may not always be possible. Dummy arcs with a very large cost may be

included into the patched cycles. These arcs are likely to be replaced by other arcs in later iterations. However, if the result of patching still contains such dummy arcs, then the resulting g-tour is permuted by an exhaustive search algorithm to eliminate the dummy arcs. Figure 5.16, 5.17, 5.18, 5.19 and 5.20 are the results on some test graphs whose cities are clustered in a more or less geographically related way. Table 5.1 shows the results of a set of test graphs whose cities are clustered randomly. The number of clusters varies from 3 to 8. The average values shown in the table are the average on the ratios of the heuristic solution to the optimal solution of all instances. The optimal solution of a problem instance is obtained by a branch-and-bound algorithm. Due to the lack of a good bounding estimate for graphs with intersecting clusters, the clusters in all testing graph are non-intersecting. On the VAX 11/780 computer, in most cases, it took about nine hours CPU time on the average to get an optimal solution for a graph with eight non-intersecting clusters. The long execution time is due to the inability of purging large number of solutions in the branch-and-bound algorithm.

The following example is to show how the query processing problem in Figure 5.1 and 5.2 is solved. The graph representation of the problem is shown in Figure 5.21. In the problem, a query initiated form site X is to join $\mathbf{R}_1,\mathbf{R}_2,\mathbf{R}_3,\mathbf{R}_4$ together. These files are allocated at sites W,X,Y,Z. In the graph, each site is a node; the cost of arcs are the communication cost between two sites; and the sites storing the same file are grouped into the same cluster. For simplicity, the graph is assumed symmetric and arcs are bidirectional with the same communication cost in both directions. As discussed in Section 5.1, the query processing problem can be solve as a GTSP problem if the communication

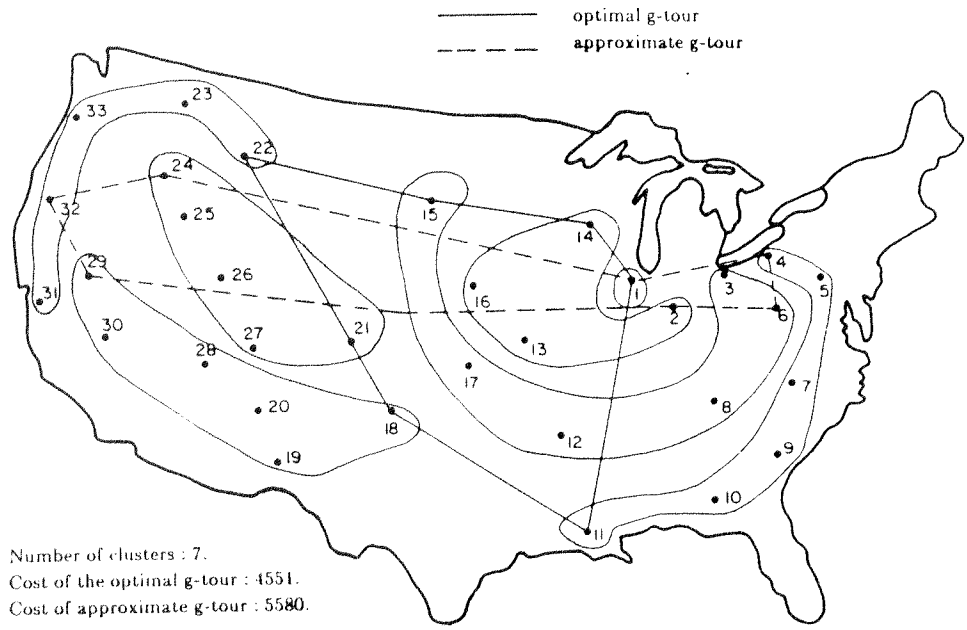


Figure 5.16 Example graph 1: optimal and approximate GTSP solutions.

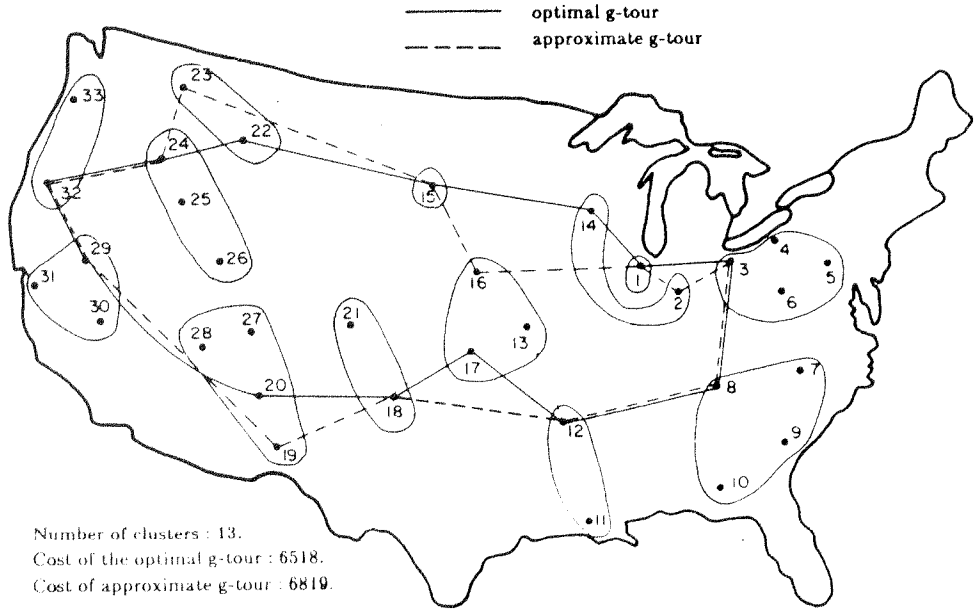


Figure 5.17 Example graph 2: optimal and approximate GTSP solutions.

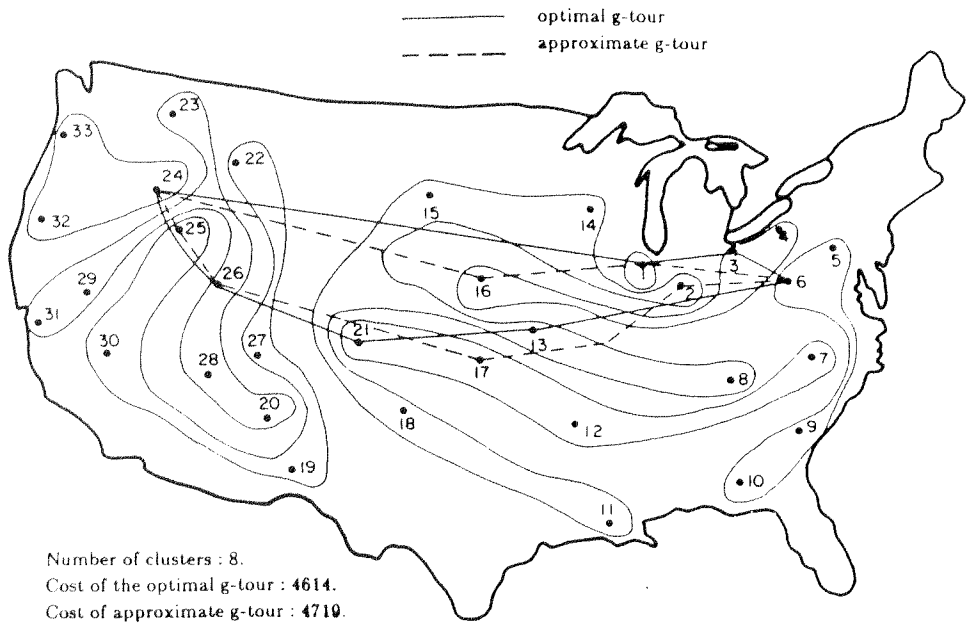


Figure 5.18 Example graph 3: optimal and approximate GTSP solutions.

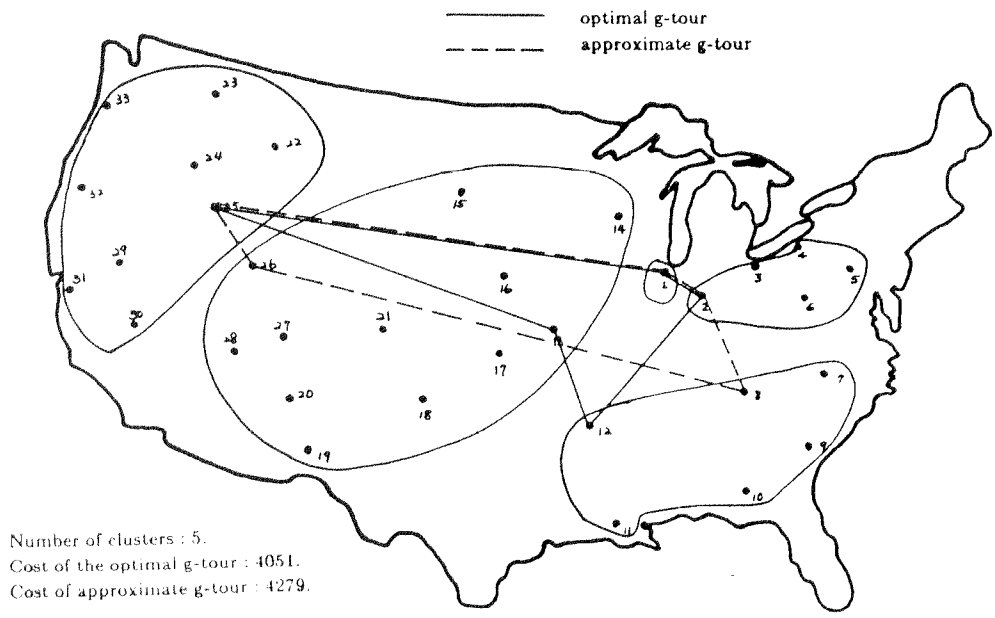
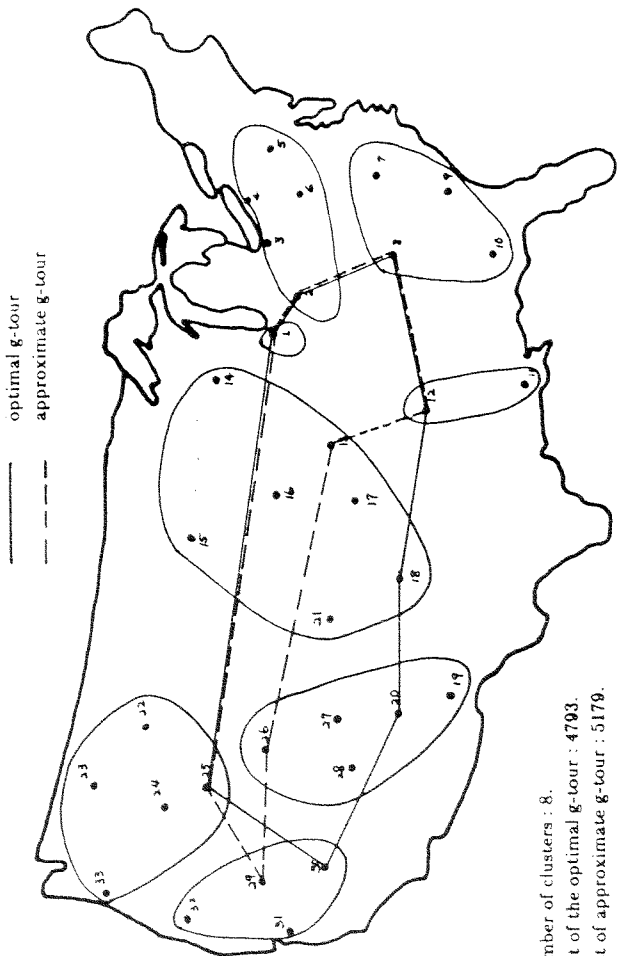


Figure 5.19 Example graph 4: optimal and approximate GTSP solutions.



Number of clusters : 8.
Cost of the optimal g-tour : 4793.
Cost of approximate g-tour : 5179.

Figure 5.20 Example graph 5: optimal and approximate GTSP solutions.

Table 5.1 Performance of the TSP-transformation approach as solved by Karp's TSP algorithm.

Cost of G-tours				
Number	Average	Standard Dev.	Maximum	Minimum
3	1.00163	0.001551	1.01635	1
4	2.35578	0.331121	3.54366	1
5	1.58284	0.153909	2.37311	1
6	1.95684	0.133881	2.62162	1.1785
7	1.52378	0.117846	2.16867	1.03278
8	1.40966	0.109277	2.00763	1.01676

- Optimal cost is 1
- 10 cases are executed for each type of gaphs with a certain number of clusters

Execution Time								
Number of clusters	Average Karp	Average Opt.	Stand Dev. Karp	Stand Dev. Opt.	Maximum Karp	Maximum Opt.	Minimum Karp	Minimum Opt.
3	16907	7	842	0.7	23778	12	14528	4.5
4	16310	21	319	2.2	18985	36	15383	13.7
5	16102	83	243	18.7	17121	208	14907	15.1
6	16328	306	140	55	17368	674	15892	112
7	25343	4688	1605	2083	33147	18392	17594	280
8	24624	33218	2795	15980	42595	159997	18403	849

- CPU time unit : seconds on a VAX-11/780

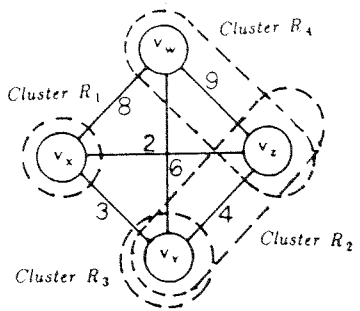


Figure 5.21 Graph representation of Figure 5.2.

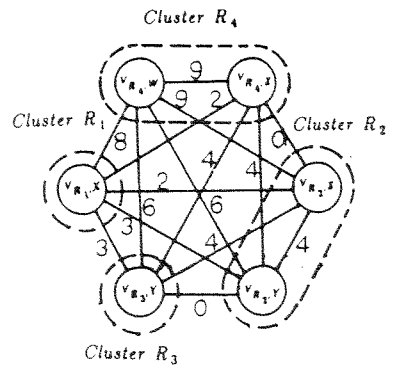


Figure 5.22 I-N graph of the graph in Figure 5.21.

cost between each pair of nodes is independent of the volume of transferred data. The corresponding I-N graph and G-S graph are shown in Figure 5.22 and 5.23 respectively. For simplicity, single-node-clusters, clusters R_1 and R_3, are not expanded in Figure 5.23, and the arcs connecting expanded nodes are bidirectional. One of the TSP-tour in the G-S graph is shown in Figure 5.24. The corresponding tour and g-tour in the I-N graph and the original graph are shown in Figure 5.25 and 5.26 respectively. The joining sequence shown in Figure 5.1 can then be obtained from Figure 5.26.

5.7 Summary

In this chapter, we have examined the characteristic of communication networks in the future and have proposed an algorithm to solve the distributed query processing problem on such networks. The problem is then formulated as a generalized traveling salesman problem. The properties of the generalized traveling salesman problem and various approaches to solve it are studied. A set of transformations have been proposed to solve GTSP by applying the solution algorithms of TSP on the transformed graph. A graph with intersecting clusters is first transformed into another graph with non-intersecting clusters. The only requirement for this transformation to be valid is that the triangle inequality must be satisfied in the original graph. The graph with non-intersecting clusters is then transformed into a third graph such that the solution of the standard traveling-salesman problem on the third graph provides a solution to the original problem. The transformation is done by expanding each node in the original graph into three nodes and adding m nodes to control the routing. For

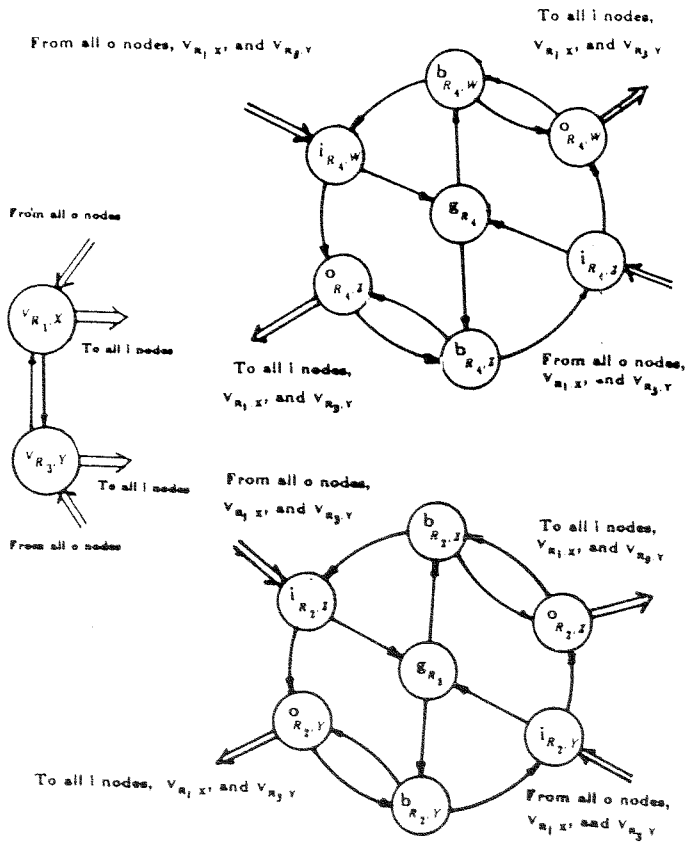


Figure 5.23 G-S graph of the graph in Figure 5.22.

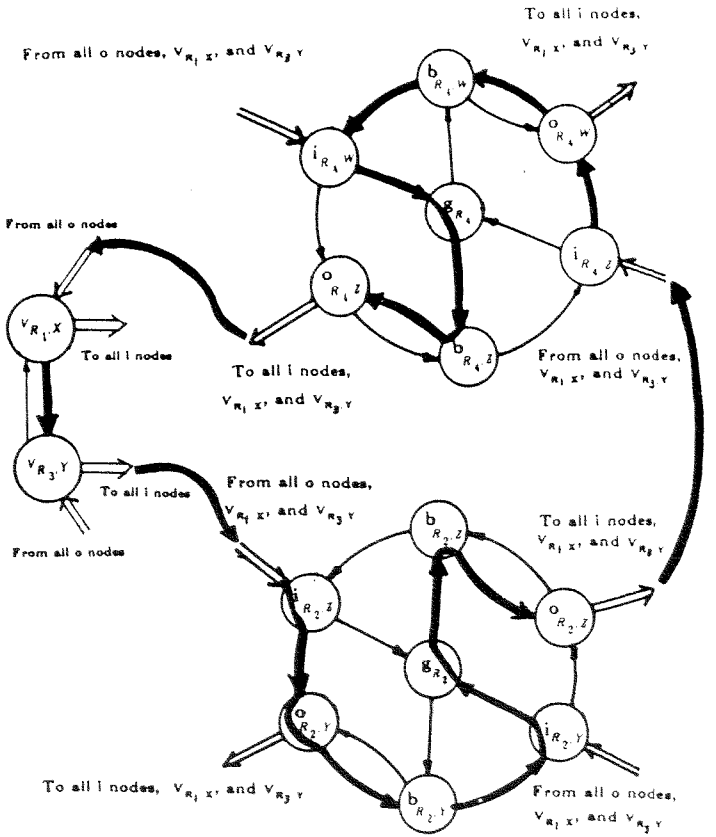


Figure 5.24 A TSP tour found from the G-S graph.

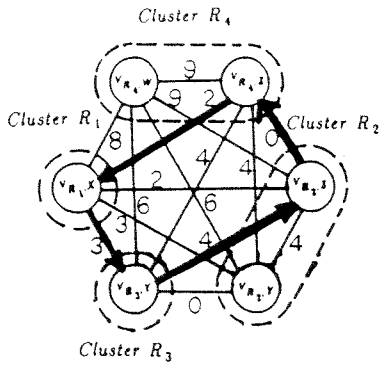


Figure 5.25 The TSP tour on the I-N graph.

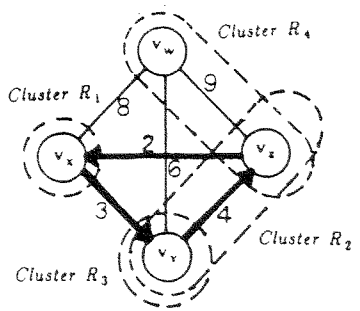


Figure 5.26 Final G-tour on the original graph.

graphs with intersecting clusters, another transformation converts the possibly infeasible solution obtained by solving TSP on the transformed graph into a feasible and canonic solution of the original graph.

CHAPTER VI

PERFORMANCE EVALUATION

One of the best ways to evaluate the DDBLMN is to implement a prototype system and then benchmark its performance. However, such a study would need extensive time and resources, which is far beyond the limit of this thesis. Further, it's inefficient to adjust the design by building new prototypes. Instead, a simulation study is conducted to demonstrate the feasibility, correctness, and effectiveness of the DDBLMN. The performance of different processing strategies are compared in the simulations.

6.1 Objectives of the Simulation

The objectives of the simulations are presented in this section.

1. **Showing the Correctness.** The design of a complicated integrated system such as DDBLMN may not be bug-free even if each individual algorithm is proved correct. Further, the correctness of some algorithms in this thesis, such as cc_minimumlock, is hard to prove formally. Hence, simulations are necessary to demonstrate their correctness.

2. **Discovering the implementation difficulties.** Some assumptions made in this research may not be easy to implement in a particular system. For example, no restriction is made on the number of Transaction Handlers that a system can initiate concurrently. This, in turn, makes an implicit assumption that each

local system can initiate as many TH's as possible. However, this is very difficult to implement in a small computer, since the number of concurrent processes that a system can handle is usually limited. As a consequence, a transaction may be blocked after the initial broadcast in the home site due to the lack of available processes in other sites. Such problems can be easily discovered through simulations and then solved before the system is actually implemented.

3. **Comparing different strategies.** When selecting a control strategy among possible alternatives in implementing the DDBLMN, the system designer should be aware of the differences among various strategies on the potential of achieving system goals under different conditions. Henceforth, a comparative study is one of the objectives in this simulation study. We have compared between

(a) redundant and non-redundant materialization, and

(b) releasing locks after the LP phase and after the RT phase.

These strategies are compared under the following conditions:

(a) the ratio of the number of update transactions to the total number of transactions,

(b) the number of transactions a site can handle,

(c) the ratio of packet size in the communication system to the block size in disks, and

(d) the transaction arrival rate.

6.2 Queuing Network Model

Figure 6.1 depicts the *event-driven* queuing model of DDBLMN. The physical resources are modeled as servers, which include DISK's, CPU's and the NETWORK. The logical resources (relations) are not shown in Figure 6.1, which are controlled by the concurrency control protocols and cannot be modeled as a simple server. However, a pseudo server "THINK" is used to model the lumped waiting time in the transaction processing. A request for a service, the start of a service, or the termination of a service is an event. The routing of tokens is dependent on the state of the system. An active TH in a site is represented as a token, which circulates around the servers in a site and the server NETWORK. A token in a site is generated for an incoming transaction, which in turn generates another set of tokens in other sites after the initial broadcast. Each such token represents a TH in a related site. A token is removed either after the transaction terminates or after the site where it is located is no longer involved in the processing. A token is *busy* if it is being served by a physical server (a physical server represents a physical resource); *waiting* if it is waiting for service from a physical server; and *sleeping* otherwise. A token is also called *active* if it is not sleeping.

All processes in one site share the single CPU in a 'process sharing' mode. The service time of a CPU server is proportional to the complexity of the job and the volume of the data to be processed in the CPU. Since the processing time of the local database in each site is dominated by the disk accessing time, it is modeled as a DISK server working in FCFS (first come first serve) mode. Only one token can be serviced at a time, and other tokens requesting a DISK service should wait in the DISK queue. The service time of a DISK server is a function of the disk seek and data transfer times. The disk seek time is a

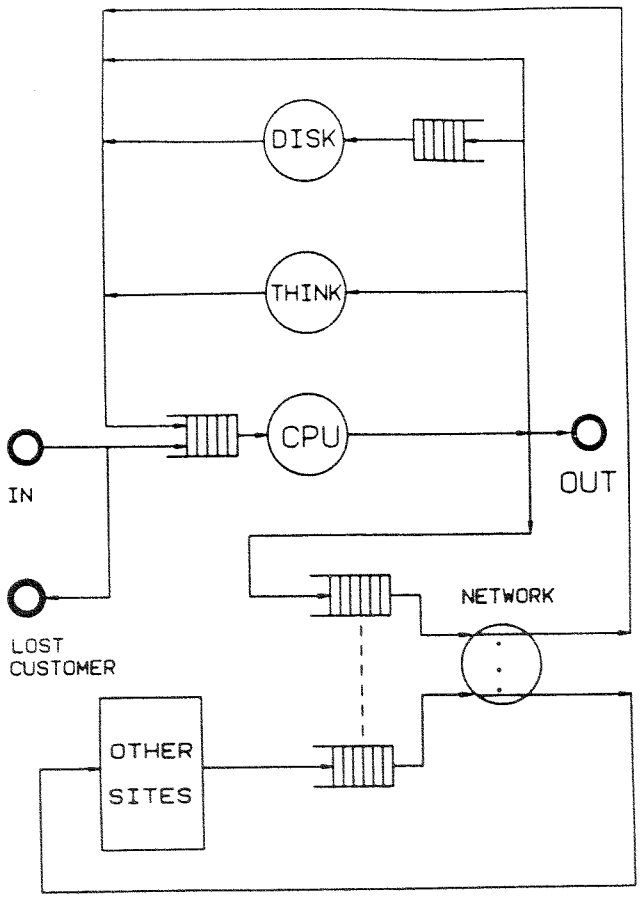


Figure 6.1 Queuing model of simulator DDBLMN-SIM.

uniformly distributed random variable. The data transfer time is proportional to the volume of data to be read from or written to the disk. All sleeping tokens in a site are put in the local server THINK until wakened up by other tokens. There is only one NETWORK server in the model, which is operated in a "random selection" mode to model the behavior of the CSMA/CD protocol. In each site, a first-in-first-out (FIFO) queue is used to store all local tokens (tokens generated from this site) that request NETWORK services. The NETWORK server randomly picks one of the non-empty queues and serves the first-arrived token in this queue. This type of servers is called a RANDOM server in this thesis. The service time is also proportional to the volume of data to be transferred.

6.3 Process Flow of DDBLMN

The process flow of a transaction in the system is shown in Figure 6.2. A transaction is initialized in the home site first. After the initial broadcast, all sub-queries are processed by a set of sites cooperatively. In each sub-query processing, a TH requests read-locks from the local Concurrency-Control subsystem for those relations stored locally and used in the LP phase. The Concurrency-Control subsystem grants the locks to the TH temporarily if the request passes the consistency checking. At the end of LP processing, a synchronization phase is proceeded to synchronize all sites. The synchronization process also serves as lock confirmation. After the synchronization process starts successfully, the temporarily granted locks are permanently granted to the transaction. Before the locks are confirmed, any conflicting lock broadcasts on the network will invalidate these temporary locks. The LP phase should be restarted in this case.

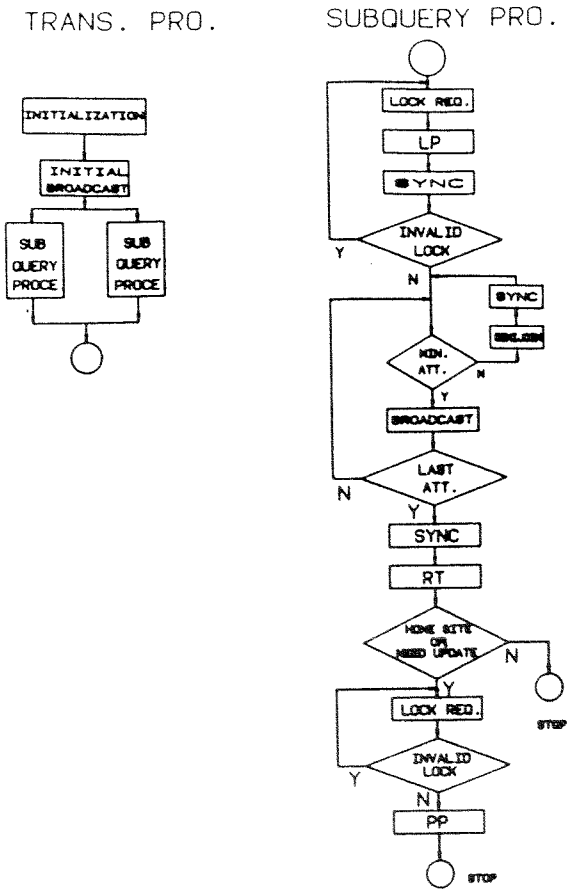


Figure 6.2 Process-flow diagram of DDBLMN-SIM.

Following the successful synchronization, the GSJ phase starts immediately. During this phase, a sequence of global semi-joins are processed iteratively. In starting an iteration, a TH requests the read-locks for those relations not locked in the LP phase. The minimum attribute is then identified and broadcast to other sites which execute the semi-joins with the minimum attribute. The iteration then terminates, and the next iteration begins. The RT phase is started after all attributes are broadcast. All related sites broadcast in turn their relations that have not been broadcast by other sites to the home site. A TH not in the home site terminates at the end of the RT phase if there is no relation in this site to be updated. All sites with relations to be updated continue their processing in the PP phase. The write locks are first obtained in this phase. After the write-locks are temporarily granted by the Concurrency-Control subsystem, the home site broadcasts the identity of the target data with new values to all sites. The write-locks are confirmed by the broadcast. All sites will update their database and terminate the transaction. The write-locks may be invalidated before the broadcast if there is a conflicting lock broadcast during this phase. The PP phase should be restarted in this case. All relations to be updated are still read-locked in this phase, therefore, there is no need to restart from the beginning.

6.4 Assumptions in The Simulation

To simplify the simulation so it can be conducted within a reasonable amount of time, the following assumptions are made.

(1) There is only one query in each transaction.

(2) There is at most one attribute to be updated.

(3) The attribute to be updated is also a queried attribute. Hence, a write is always preceded by a read.

(4) If non-redundant materialization is used, a copy of each queried relation is randomly selected. (In the simulations, the non-redundant materialization is really implemented by a non-replicated file allocation.)

(5) After semi-joined of joining attribute of selectivity p, a relation (say, R_i) is reduced from the original size of s_i to a size distributed within 30% deviation of $s_i \times p$. The estimated size of a relation after a semi-join is $s_i \times p$. The indirect semi-join effect is ignored.

(6) The heuristic function used in the GSJ phase is assumed to be the product of the effective size and the relative selectivity of the candidate minimum attributes.

(7) The join restriction effect in the RT phase is ignored since it is insignificant and is expensive to calculate in the simulations.

(8) The transmission order in the RT phase is governed by the site numbers.

(9) Since only one message can be broadcast from any site at any time, we assume that the Network-Interface subsystem in each site can only process one network access request at any time.

(10) The service time of a CPU server is independent of the number of processes sharing the CPU. We make this assumption for two reasons. First, the CPU time is about one to two orders of magnitude less than the disks and network service times (the minimum overhead of the disk/network is a block/packet). Second, inexpensive multiprocessors are going to be available soon such that each TH may be served by a dedicated processor in such systems. (Intel has already produced a hypercube computer system consisting of 32 80286 microprocessors with a cost less than $100,000.) Finally, switching the CPU to service another token during the simulations is very computationally expensive.

6.5 Simulator DDBLMN-SIM

The simulation is implemented by a simulator DDBLMN-SIM, which is written in the C programming language and developed on a *UNIX** systems runing *BSD* 4.2* at the Department of Computer and Information Science, the Ohio State University. The simulator is executed on a SUN workstation runing the same operating system.

A set of primitive subroutines in a simulation management package DDBLMN-SMPL, which is a modification of SMPL [MAC80] is used to control the simulations. These subroutines include initialization operations, event scheduling, facility reservations, random variable generation, data collection, and reporting routines. A simulation model using DDBLMN-SMPL subroutines is a static network of FCFS and RANDOM facilities. Each of these facilities is associated with a queue in which incoming tokens wait for service. When the facility becomes available, a token is taken off the queue and serviced. The service involves reserving the facility for a given time period and, when that time has expired, passing the token to either another facility or the THINK state to sleep. After initialization of facilities, the simulator performs four basic operations iteratively.

(1) Reserves a facility for a token to disallow any other users from using it. If the facility is already in use, the requesting token is placed in a queue waiting for the facility.

* UNIX is the trademark of AT&T. BSD 4.2 is Berkeley version 4.2.

(2) Releases a facility, and allows a token (either the first one or randomly selected) in its queue to access to it if the queue is not empty.

(3) Schedules a new event to happen at some future time. This action places the new event in a time-ordered queue of future actions. This step is referred to *new-event-scheduling*.

(4) Causes the next event to occur. This action removes an event from the time-ordered queue.

Figure 6.3 shows the fundamental flow of events used in the simulations. After the queuing network is initiated, events are repeatedly generated until the following termination conditions are satisfied:

(1) the system response time is stablized, and

(2) at least a certain number of transactions are executed.

Condition (2) is needed to increase the confidence on the property of deadlock free.

Each event asks the simulator to either reserve or release a facility. If the facility is successfully reserved, the simulator schedules a release event to occur at the end of the service time. If the event was to release a facility, then the program releases it, routes the token to the next facility according to the state of the system, and schedules a new event to reserve the new facility immediately.

A token in any queue can be removed and rerouted to the local THINK server. This is useful for modeling the behavior of a Transaction Handler when the temporary locks it holds are invalidated by another transaction.

It is not very difficult to implement the queuing network model in Figure 6.1 by DDBLMN-SMPL. For DISK servers, the implementation is

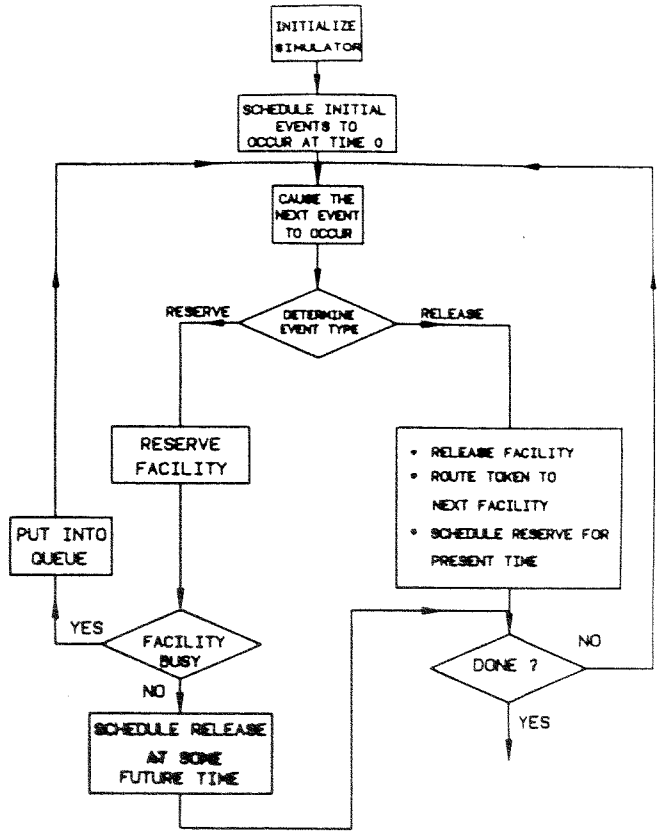


Figure 6.3 Event flow of DDBLMN-SMPL.

straightforward since they are FCFS servers. However, modifications to DDBLMN-SMPL are needed to model resources operating in other types of service and queuing disciplines. For the servers that can service all incoming token parallelly, such as THINK and CPU, there is no associated waiting queue. The finish time can be immediately calculated. Thus, next event can be immediately scheduled when a service is requested. When a token needs a service from the CPU at time t_1, the service time t_2 is first computed and added to t_1. The next event after the CPU service is scheduled at time $t_1 + t_2$. When a token is routed to a THINK server (either blocked or halted) in new-event-scheduling, it stays there until wakened up by another token. There is no new event to be scheduled in new-event-scheduling in this case. For the NETWORK server, which serves in a "random selection" discipline with multiple FIFO queues, we add a logical server in front of each queue to model the Network-Interface subsystem in each site. This is due to Assumption (9) in Section 6.4. Another queue is attached to the NETWORK sever itself. A token leaving the Network-Interface server is routed to the NETWORK queue. Also, at most one token from each site can be routed to the NETWORK queue. In the mean time, Network-Interface is not released to service another token unless the token in the NETWORK queue is serviced. When the NETWORK server is free, a token in the NETWORK queue is randomly picked to be served and the Network-Interface subsystem from which the token came from is free to service another token in its queue. The tokens in the Network-Interface servers of other sites should keep waiting for a NETWORK service. The DDBLMN-SMPL implementation of the queuing model shown in Figure 6.1 is detailed in Figure 6.4. As we can see, there is no FIFO queue associated with servers THINK and CPU. The Network-Interface subsystem in each site is modeled as a FCFS

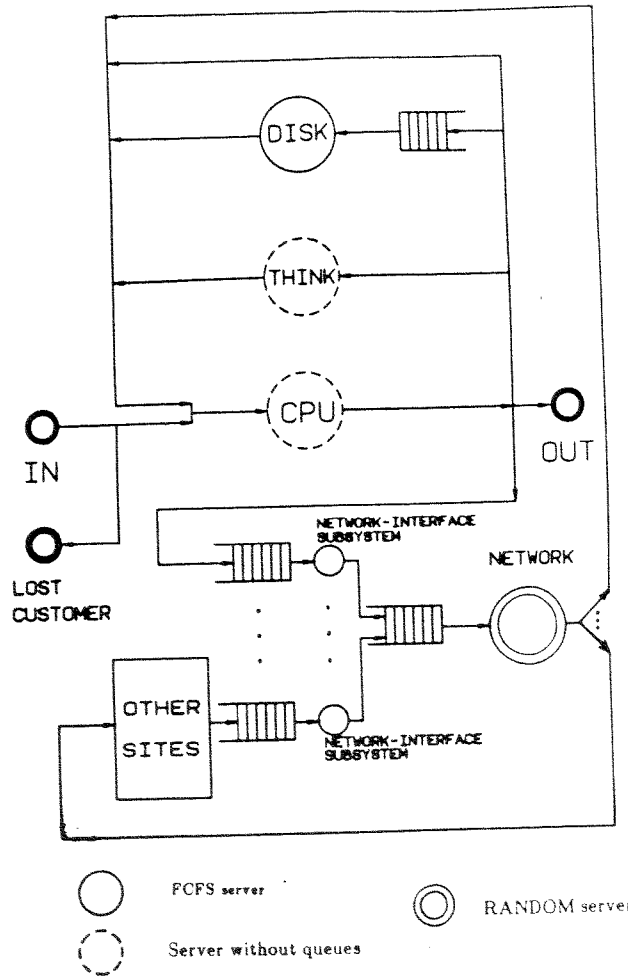


Figure 6.4 DDBLMN-SMPL implementation of DDBLMN-SIM.

facility in DDBLMN-SIM. The server NETWORK is modeled as a RANDOM facility.

In addition to the capabilities of supporting non-FCFS facilities and removing tokens from any queue, DDBLMN-SMPL differs from the original SMPL in that it is improved in DDBLMN-SIM. For example, a number of subroutines in SMPL are translated into macro definitions in DDBLMN-SMPL to reduce the overhead of subroutine calls in the operating system level.

The greatest difficulty in implementing the DDBLMN-SIM is token routings which is controlled by the complicated system state. The state transition diagram is shown in Figure 6.5. The detailed description of these states can be found in the program listed in the Appendix.

6.6 Simulation Parameters

The parameters used to drive DDBLMN-SIM are shown in Table 6.1 to Table 6.8. Table 6.1 shows the parameters for system configuration. Due to the CPU time constraints on the simulations, only four sites, four relations, and four joining domains are set up for the system. We assume that each site can initiate as many Transaction Handlers as possible. However, the maximum number of transactions entering a site is limited. As a result, the number of TH's that a site may generate is also limited. The cardinalities of the joining domains, the width of the joining domains, the width of the relations, and the initial cardinalities of relations are also shown in Table 6.1. The configurations of relations, which tell whether a relation has a particular joining domain, is shown in Table 6.2. The initial selectivity of each joining attribute, which is the ratio of the initial cardinality of the attribute to the cardinality of the domain, is listed in Table 6.3. The allocation of relations is shown in Table 6.4.

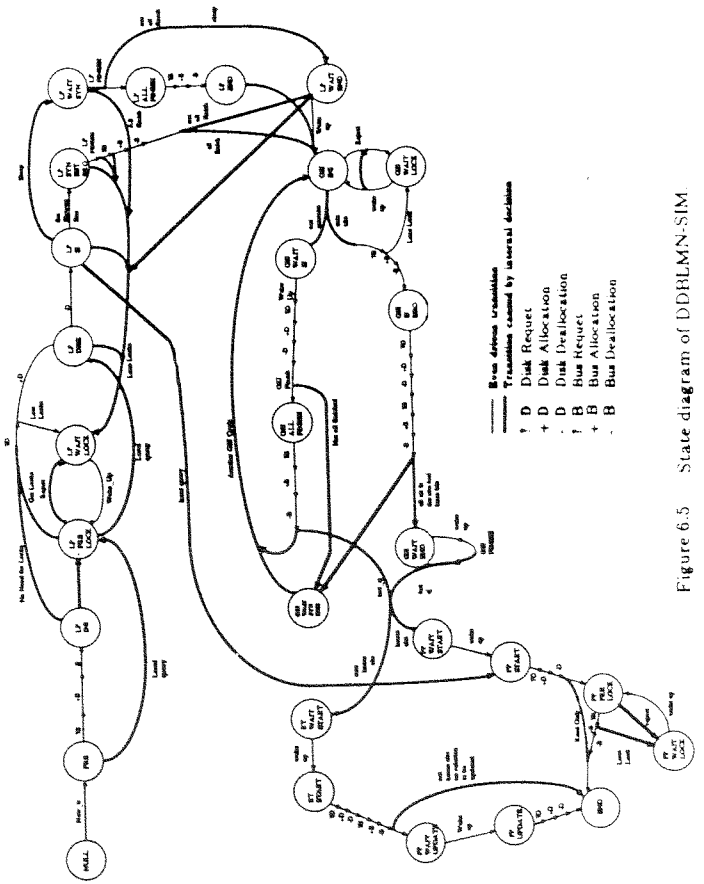


Figure 6.5 State diagram of DDBLMN-SIM

Table 6.1 Parameters of DDBLMN-SIM system configuration.

Number of sites	4
MAX_TR_OF_SITE	5,8,10
Number of relations	4
Number of joining domains	4
Cardinalities of joining domains	80, 80, 160, 200
Width of joining domain	16, 8, 24, 40
Width of relations	240, 376, 408, 368
Initial cardinalities of relations	38707, 40960, 193536, 30720

MAX_TR_OF_SITE: max. number of transactions that can enter a site at the same time.

Table 6.2 Configuration of relations in DDBLMN-SIM.

(0/1: existence/nonexistence of an attribute in a relation).

	domain 1	domain 2	domain 3	domain 4
Relation 1	0	0	1	1
Relation 2	0	1	1	0
Relation 3	1	1	1	0
Relation 4	1	0	0	1

Table 6.3 Initial selectivities of all joining attributes in DDBLMN-SIM.

	attribute 1	attribute 2	attribute 3	attribute 4
Relation 1	-	-	0.6	0.9
Relation 2	-	0.4	0.9	-
Relation 3	0.8	0.5	0.1	-
Relation 4	0.8	-	-	0.5

Table 6.4 Relation allocation in DDBLMN-SIM.

	Site 1	Site 2	Site 3	Site 4
Relation 1	1	0	1	0
Relation 2	0	1	1	0
Relation 3	1	0	0	1
Relation 4	0	1	0	1

Table 6.5 Service time of servers in DDBLMN-SIM.
(time unit : 10^{-6} second)

CPU	3.0 * number of bytes
DISK	uniform [0, 20000] + 1000 * number of blocks
BUS	1.0 * bits/packet * number of packets

Table 6.6 Other timing parameters in DDBLMN-SIM.
(time unit : 10^{-6} second)

Bus contention slot	50
Preprocessing time	1000

Table 6.7 Data format in DDBLMN-SIM.

Number of bits/byte	8
Disk block length	1000 bytes
Packet length (bytes)	100, 500, 1000, 2000

Table 6.8 Parameters used for query generation in DDBLMN-SIM.

QUERY_REL_HIT	0.8
TARGET_REL_HIT	0.4
QUERY_ATT_HIT	0.4
TARGET_ATT_HIT	0.4
UPDATE_QUERY_HIT	0, 0.4, 0.7, 1
UPDATE_ATT_HIT	0.4

QUERY_REL_HIT: probability that a relation is queried by a query.
TARGET_REL_HIT: probability that a relation is to be retrieved by a query.
QUERY_ATT_HIT: probability that an attribute is queried by a query.
TARGET_ATT_HIT: probability that an attribute is to be retrieved by a query.
UPDATE_QUERY_HIT: probability that a query is an update.
UPDATE_ATT_HIT: probability that an attribute in an update is to be updated.
UPDATE_QUERY_HIT is the probability that a query is an update.
UPDATE_ATT_HIT is the probability that an attribute in an update is to be updated

All timing parameters are shown in Table 6.5 and 6.6. At a speed of 1 MIPS, we assume that a CPU can process a byte (8 bits) in 3 instruction cycles and that the processing time is proportional to the number of bytes to be processed. The disk access time consists of two parts: the block seek time and the data transfer time. The length of each block is 1000 bytes. The block seek time is a uniformly distributed random variable between 0 and 20 miliseconds, which represents the time of moving the disk head and locating the sector. It is assumed that there is no seek time between adjacency blocks transferred in the same request. Once the first block is located, successive blocks can be transferred continuously with a speed of 1 milisecond per block. The bus communication overhead is represented by a constant and a variable proportional to the volume of transferred data. We assume that all necessary initialization work is done in the Network-Interface subsystem, and the contention overhead is accounted for separately. Consequently, the service time of the BUS server is only dependent on the number of transferred packets. The number of overhead bits is assumed to be 208, which is the same as the Ethernet specification. A contention slot in the CSMA/CD protocol is assumed to be 50 bit-times. The preprocessing phase lasts for 1 milisecond. The data format is listed in Table 6.7. The size of a disk block is fixed to be 1000 bytes and 8 bits per byte, while the packet length varies from 100, 500, 1000, to 2000 bytes.

There is a single queue of transaction arrivals with an exponentially distributed interarrival time. The mean interarrival time (MIT) varies from 0.1, 0.3, 0.5, to 0.7 seconds. The incoming transactions are randomly routed to a site. A transaction to a site is lost if the number of pending transactions at this site has reached the maximum, which is a parameter defined to the simulator. The query pattern of a transaction is also randomly generated. Table 6.8 shows the

parameters controlling the query pattern generation. We call a relation/attribute as a *hit relation/attribute* if it is involved in the target data identification in a query. QUERY_REL_HIT/QUERY_ATT_HIT are the probabilities of a particular relation/attribute hit by a query. A hit relation must contain at least one hit attribute. The relation containing at least one hit attribute should be a hit relation. We call a relation containing at least one target attribute as a target relation. TARGET_REL_HIT/TARGET_ATT_HIT are the probabilities of a particular relation/attribute to be a target relation/attribute in a query. UPDATE_QUERY_HIT is the probability that a query is an update query. A queried relation is randomly picked to be updated if the query is an update. UPDATE_ATT_HIT is the probability that a target attribute is to be updated in a relation to be updated. In an updated relation, at least one attribute is to be updated. All these probabilities are set to 0.4 except that QUERY_REL_HIT is 0.8 and UPDATE_QUERY_HIT is varying from 0.0, 0.4, 0.7, to 1.0. All queries are read-only when UPDATE_QUERY_HIT is set to 0.0, and are updates when it is set to 1.0. Those tunable variables are summerized in Table 6.9. Each combination is run with a rates of mean interarrival time of 0.1, 0.3, 0.5, and 0.7 seconds.

6.7 Simulation Results

Due to resource contentions in distributed systems the performance measure of transaction response times is an autocorrelated quantity [FIS78]. Hence, the initial conditions of the simulation model may produce transient measurements in the simulation outputs. We ran a series of test simulations to determine the number of response time observations to be truncated in the beginning of each simulation run. By observing the resulting graphed data, we

Table 6.9 Tunable variables in DDBLMN-SIM.

MAX_TR_OF_SITE	5,8,10
PACKET_SIZE (bytes)	100, 500, 1000, 2000
UPDATE_QUERY_HIT (or UQH)	0, 0.4, 0.7, 1
R_LOCK (or r_unlock_time)	0, 1
RED	0, 1

PACKET_SIZE: packet length in bytes

R_LOCK: the time that a read lock can be released
0: after RT phase
1: any time after use

RED: materialization strategy
0: non-redundant
1: redundant

chose the truncation point on the graph where it appeared that the mean response time measurement had achieved steady-state stability. For the data shown in Figure 6.6, the value 80 is found to be a good truncation point, which is the minimum number of transaction to be observed. A value 2% is a good allowed deviation, which is the difference between the simulated response time of current terminated transaction and the average simulated response time of all terminated transactions in the simulation run. Notice that there are some unstable cases, in which the systems are saturated, in Figure 6.6. The minimum number of transactions to be executed in a simulation is set to be 100 for the safety margin, and the allowed deviation is set to be 0.5%. We now present the simulation results of different combinations of parameters generated from Table 6.9. It must be emphasized that the numerical values found in this section are accurate only to the extent that the data values and assumptions used to design the simulator are representative of an actual system. The main concern of this simulation is to show the correctness of control algorithms, to discover the implementation difficulties, and to compare the relative performance of different control strategies as discussed at the beginning of this chapter.

6.7.1 Deadlock-free and Implementation Difficulty

Since it is impossible to prove the deadlock-free property of the system by simulation, demonstrating it in the simulation is complementary work to the proof in Chapter IV. During the testing stage of the simulator development, we have tried to push the simulator to the extreme cases to find any deadlock. The incoming rate of transactions is increased to an extent far beyond the capacity of the simulated system, the percentage of updating queries to the total number of queries is increased to 100%, and the simulator is run up to thousands of CPU

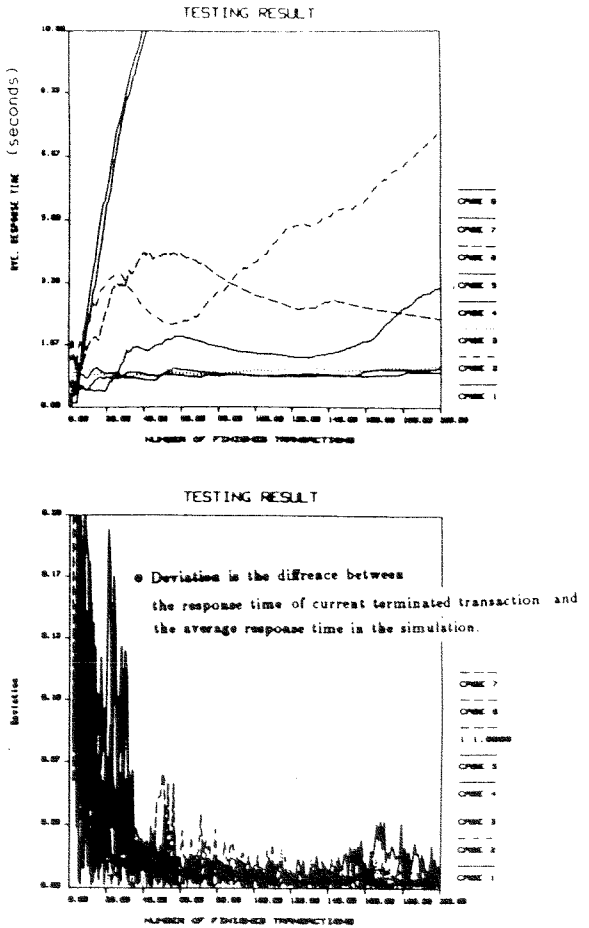


Figure 6.6 Testing results of DDBLMN-SIM
(a) mean response time; (b) deviation of response time.

minutes. There is no deadlock detected. We believe that these are sufficient to demonstrate the correctness of the concurrency control algorithms.

The implementation of DDBLMN is fairly complicated. We will not discuss the programming difficulty of the system since it is a common problem of any complicated working system. However, we would like to point out following problems.

(1) The simplicity of the concurrency control algorithms does contribute to the success of the simulator design. The major difficulties of the program development are on the state transition and the token routing, instead of on the concurrency control itself. Simplicity of the design is important in the implementation of a real system. We believe that DDBLMN would be a good choice to implement a DDBMS in local environment.

(2) The system requires each site to keep track of the complete state of the database. To save communication overhead, state transition is really calculated by each site independently based on the network messages received. The complexity of such a distributed operation is fairly high. This not only causes some processing overhead, but also makes the system development and maintenance difficult. This is especially true in a larger system. The system designer should implement this mechanism with care. The possibility of errors in this part is high and may also lead to possible deadlocks. Comparing this to other portions of the system, this should deserve more attention. For example, a mistake on the minimum attribute identification or on the cardinality estimation may only affect the performance, but a mistake in this part may cause errors to the transaction processing. We would recommend that piggybacking the necessary

information in the broadcast message would help to reduce the complexity of the design and processing overhead. This is particularly true when the communication overhead is independent to the volume of data transfer.

(3) The complexity of computing the transitive closure is $O(N^3)$, which is a substantial overhead when the system is large. The most commonly used algorithm to compute the transitive closure is a matrix operation, in which the precedence graph is represented as an adjacency matrix [AHO74]. The overhead of this algorithm may be wasteful when the probability of conflict among transactions is low. For example, in an ATM system, customers do not issue conflicting transactions frequently. In this situation, the adjacency matrix representing the precedence graph is most likely sparse. Furthermore, since the transitive closure is always kept in the precedence graph, the changes to the precedence graph would not be substantial whenever a new precedence relationship is imposed or a node is removed. There is no need to recompute the precedence order for the entire graph. The algorithm discussed in Chapter IV can be modified as follows.

Algorithm DDBLMN-TRANSITIVE

Consider the graph as a family of isolated acyclic graphs, with one and only one directed edge between any pair of nodes in the graph. Node N_i represents the transaction TR_i as stated in Chapter IV.

(a) Remove the node and all the edges connected to it when the transaction that this node represents terminates.

(b) Whenever a new precedence relationship is imposed on TR_i and TR_j (say, TR_i precedes TR_j),

(b1) connect N_i to N_j;

(b2) connect all preceding nodes of N_i to node N_j and all succeeding nodes of N_j.

We emphasis that this algorithm may not be better than the matrix approach when the probability of conflict is high and the adjacency matrix is not sparse.

6.7.2 Observations and Suggestions

The numerical simulation results are shown in Table 6.10. Each combination generated from Table 6.9 was simulated. The average, maximum, minimum, and standard deviation of all 192 combinations of different MIT on the following observations are listed.

(a) **Mean Response time** is the average response time of all transactions in a simulation.

(b) **Mean queue length of NETWORK** is

$$\left(\frac{\sum_{\text{all sites}} \left(\begin{array}{c} \text{mean queue length of} \\ \text{NETWORK-INTERFACE in the site} \end{array} \right)}{\text{number of sites}} \right) + \text{mean queue length of NETWORK}$$

in a simulation.

(c) **Mean queue length of DISK** is the average value of the mean DISK queue length in all sites, i.e.

$$\left(\frac{\sum_{\text{all sites}} \left(\begin{array}{c} \text{mean queue length of} \\ \text{DISK in the site} \end{array} \right)}{\text{number of sites}} \right).$$

(d) **Utilization of NETWORK** is the busy time of the NETWORK over the total time in a simulation.

Table 6.10 Simulation results of DDBLMN-SIM

Mean Response Time				
MIT	Ave	Max	Min	Stand. dev
0.1	7.053	27.900	0.190	0.5410
0.3	2.739	15.900	0.160	0.2995
0.5	0.833	5.190	0.170	0.0723
0.7	0.482	2.080	0.140	0.0258

Mean (NETWORK time / total trans. time)				
MIT	Ave	Max	Min	Stand. dev
0.1	0.073	0.388	0.018	0.0038
0.3	0.198	0.371	0.043	0.0071
0.5	0.284	0.442	0.143	0.0068
0.7	0.338	0.485	0.150	0.0078

Utilization of NETWORK				
MIT	Ave	Max	Min	Stand. dev
0.1	0.820	1.000	0.280	0.0200
0.3	0.585	0.999	0.105	0.0247
0.5	0.407	0.949	0.055	0.0209
0.7	0.291	0.791	0.040	0.0180

Mean (DISK time / total trans. time)				
MIT	Ave	Max	Min	Stand. dev
0.1	0.055	0.173	0.003	0.0043
0.3	0.104	0.209	0.007	0.0051
0.5	0.124	0.220	0.028	0.0043
0.7	0.131	0.220	0.040	0.0040

Mean Utilization of DISK				
MIT	Ave	Max	Min	Stand. dev
0.1	0.259	0.453	0.084	0.0082
0.3	0.128	0.185	0.084	0.0014
0.5	0.084	0.104	0.084	0.0007
0.7	0.057	0.080	0.045	0.0005

Mean (CPU time / total trans. time)				
MIT	Ave	Max	Min	Stand. dev
0.1	0.022	0.076	0.001	0.0018
0.3	0.042	0.084	0.002	0.0020
0.5	0.050	0.085	0.012	0.0016
0.7	0.054	0.088	0.016	0.0015

Lost Transactions (trans. / second)				
MIT	Ave	Max	Min	Stand. dev
0.1	0.221	2.081	0.000	0.0306
0.3	0.000	0.000	0.000	0.0000
0.5	0.000	0.000	0.000	0.0000
0.7	0.000	0.000	0.000	0.0000

Mean queue length of NETWORK				
MIT	Ave	Max	Min	Stand. dev
0.1	4.033	8.331	0.093	0.1851
0.3	1.740	6.669	0.012	0.1560
0.5	0.808	3.482	0.003	0.0685
0.7	0.197	1.598	0.001	0.0201

Total CPU Time (SUN)				
MIT	Ave	Max	Min	Stand. dev
0.1	936.593	1914.600	464.900	21.4398
0.3	732.768	1389.800	457.900	13.9473
0.5	663.789	944.400	453.000	7.6510
0.7	647.708	895.700	458.000	7.1943

Mean queue length of DISK				
MIT	Ave	Max	Min	Stand. dev
0.1	0.108	0.388	0.002	0.0035
0.3	0.019	0.067	0.002	0.0008
0.5	0.008	0.021	0.002	0.0003
0.7	0.002	0.006	0.001	0.0001

Time Unit : second

(e) **Mean utilization of DISK** is the average utilization of DISK's over all sites in a simulation.

(f) **Mean ratio of NETWORK service time to the total transaction time** is the average of

$$\left(\frac{\textit{total time serviced by the NETWORK}}{\textit{total transaction time}} \right)$$

of each individual transaction in a simulation.

(g) **Mean ratio of DISK service time to the total transaction time** is the average value of

$$\left(\frac{\textit{total time serviced by the DISK's}}{(\ \textit{total transaction time}\ ) \cdot (\ \textit{number of sites}\ )} \right)$$

of each individual transaction in a simulation.

(h) **Mean ratio of CPU service time to the total transaction time** is the average value of

$$\left(\frac{\textit{total time serviced by the CPU's}}{(\ \textit{total transaction time}\ ) \cdot (\ \textit{number of sites}\ )} \right)$$

of each individual transaction in a simulation.

(i) **Rate of lost transaction per second** is the ratio of total number of lost transactions to the total simulation time in a simulation.

(j) **Total CPU time** is the total CPU time used in a simulation on SUN workstations.

The following are observations and suggestions we draw from the simulation results.

(1) From Figure 6.7, 6.8, and 6.9, it is clear that the system can handle the load when the MIT is longer than 0.5 seconds, but will be saturated when

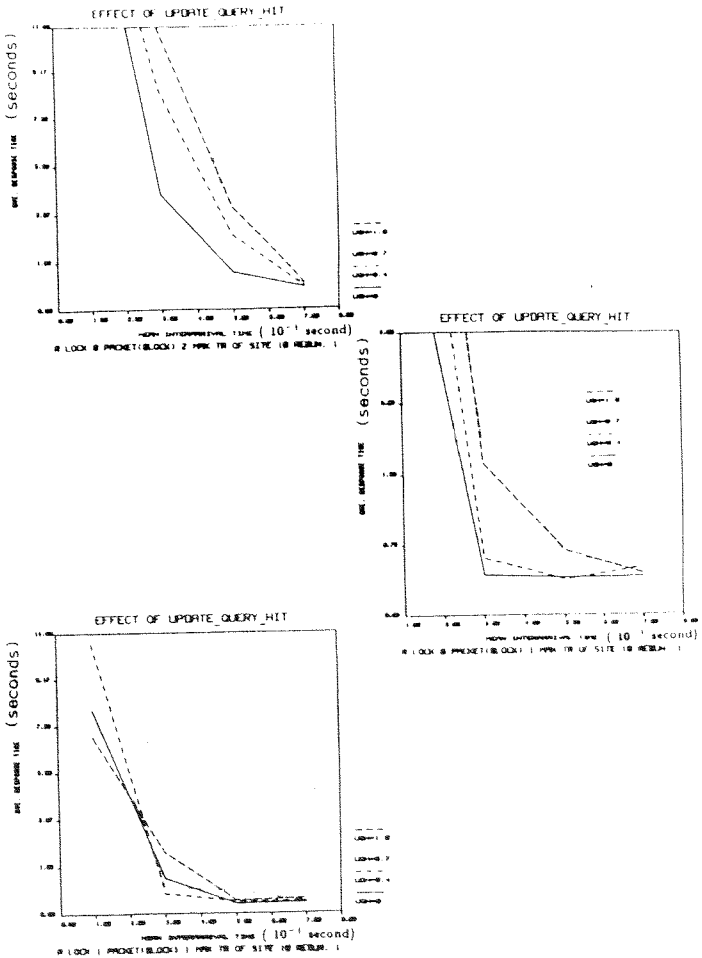


Figure 6.7 Mean response time.

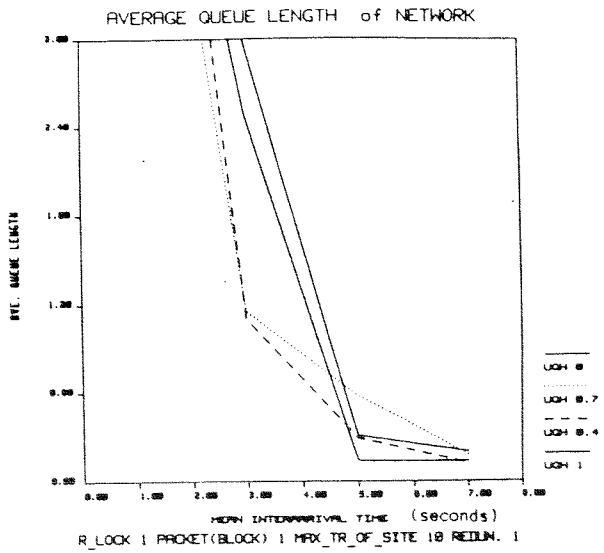


Figure 6.8 Mean NETWORK queue length.

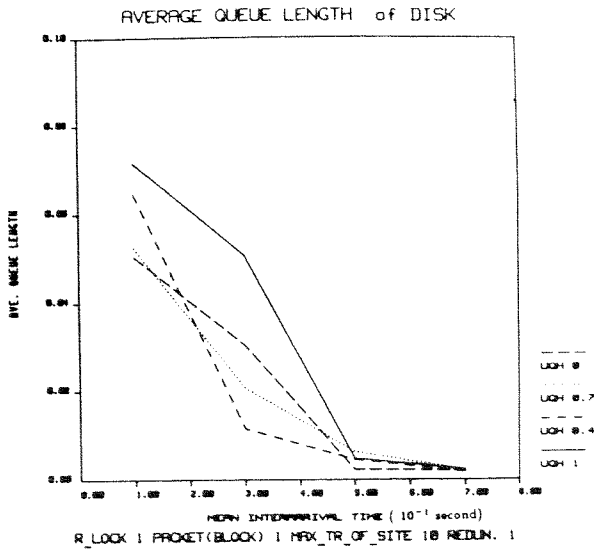


Figure 6.9 Mean DISK queue length.

the MIT reduced to 0.1 in many cases. When saturated, the queue of NETWORK (including NETWORK-INTERFACE) is full and a lot of incoming transactions are lost. This occurs even when all queries are not updates. When the MIT is longer than 0.3 seconds, the rate of lost transactions is almost invisible.

(2) Figure 6.7 shows that the percentage of updating queries does affect the response time. The more the updates are, the longer the response time.

(3) From Figures 6.8 and 6.9, we can also see that the UPDATE_QUERY_HIT has no significant effect on the length of either the DISK or NETWORK queues. That is true because we assume that at most one relation is updated and the volume of updated data is usually small. This fact also indicates that the communication overhead for lock management in this system is very low as we expected.

(4) Figure 6.10 shows the comparison of the utilization of DISK's and NETWORK. It is very clear that the NETWORK is heavily utilized, especially, when the system load is high. When the system is approaching saturation point, the utilization of the NETWORK is close to 100%. The DISK utilization is kept as low as 20% even at this point. This tells us that the NETWORK is the bottleneck of the system.

(5) Figure 6.11 shows the comparison of the average ratio of the simulated time serviced by CPU, DISK's and NETWORK to the total simulated transaction time. It is also very clear that a large portion of the transaction processing time is spent on the NETWORK. Only less than 10% of the time is spent on the DISK's and less than 5% on the CPU. Less than 50% of the time is waiting when MIT is kept longer than 0.3.

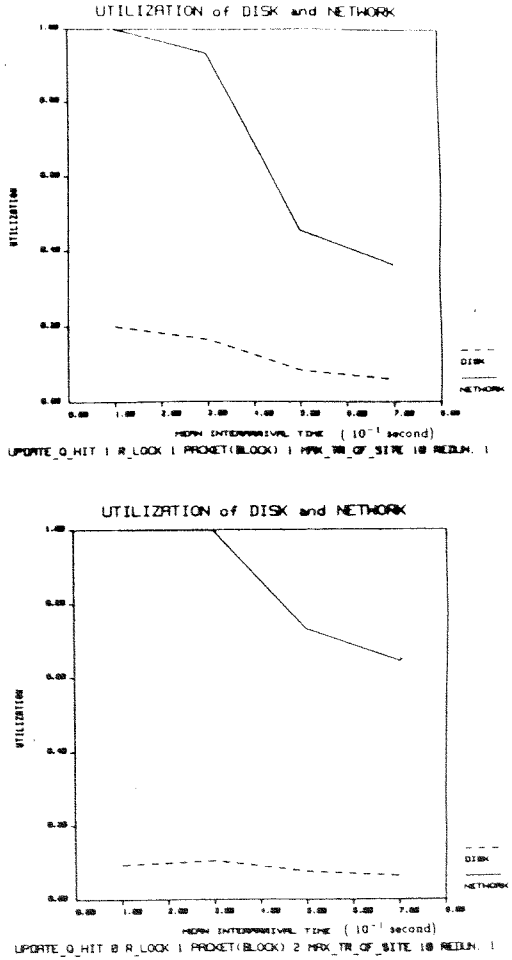


Figure 6.10 Comparison of the utilization of DISKs and NETWORK.

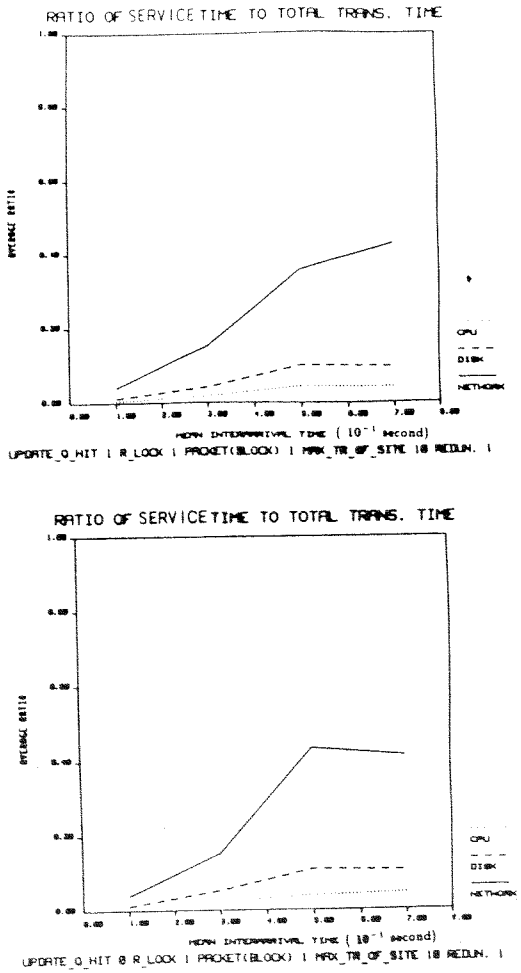


Figure 6.11 Comparison of the served time of CPU's, DISKs, and NETWORK.

(6) The size of packets in the NETWORK has significant effect on the response time as we can see from Figure 6.12. It seems unrealistic that the smaller the packets are, the better the system is. This is due to the fact that the fixed overhead in each packet is only 208 bits and the minimum packet length we simulated is 800 bits such that the fixed overhead does not affect the performance significantly. Really, there are many short broadcasts during the transaction processing. For fixed length packets, a large portion of their capacity is wasted.

There are two way to reduce this overhead: either using variable length packets or piggybacking the necessary information into the sparse packets. Due to the time constraints, we did not simulate either case in our simulations.

(7) Figure 6.13 shows the effect of read_lock releasing time. Two alternatives are simulated. When the UPDATE_QUERY_HIT is 0.0, in which all queries are read-only, all relations can be released right after use. Therefore, having a second copy in the working space has no advantage. Moreover, the induced overhead may even worsen the performance. As the UPDATE_QUERY_HIT is increased, having second copy does make the system performs better.

(8) The effect of materialization can be seen from Figure 6.14. Redundant materialization does have some improvement over non-redundant materialization. The difference is not great, this is because the scale of the simulation is too small such that at most two copies of a file are allocated. We believe that redundant materialization will be more attractive when the scale of the system is larger than DDBLMN-SIM. Although there are extra CPU and disk overhead, they are not as significant as we observed from this simulation.

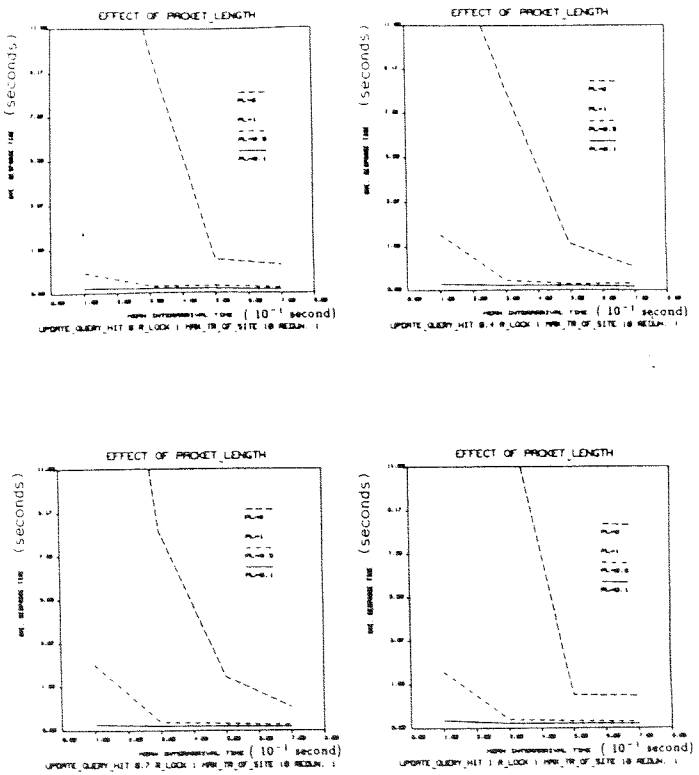


Figure 6.12 The effect of packet size to the mean response time.

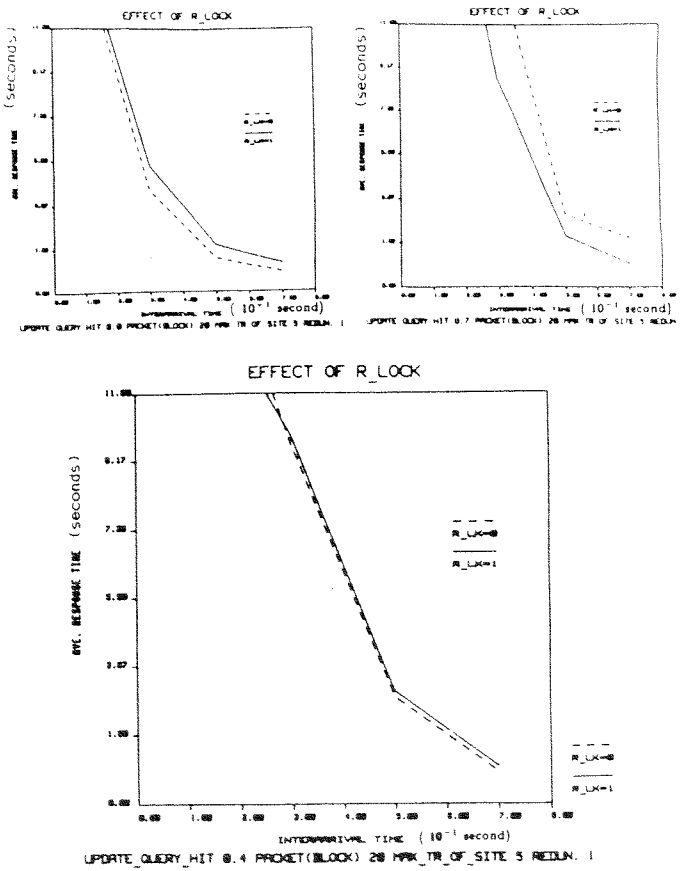


Figure 6.13 The effect of read_lock releasing time to the mean response time.

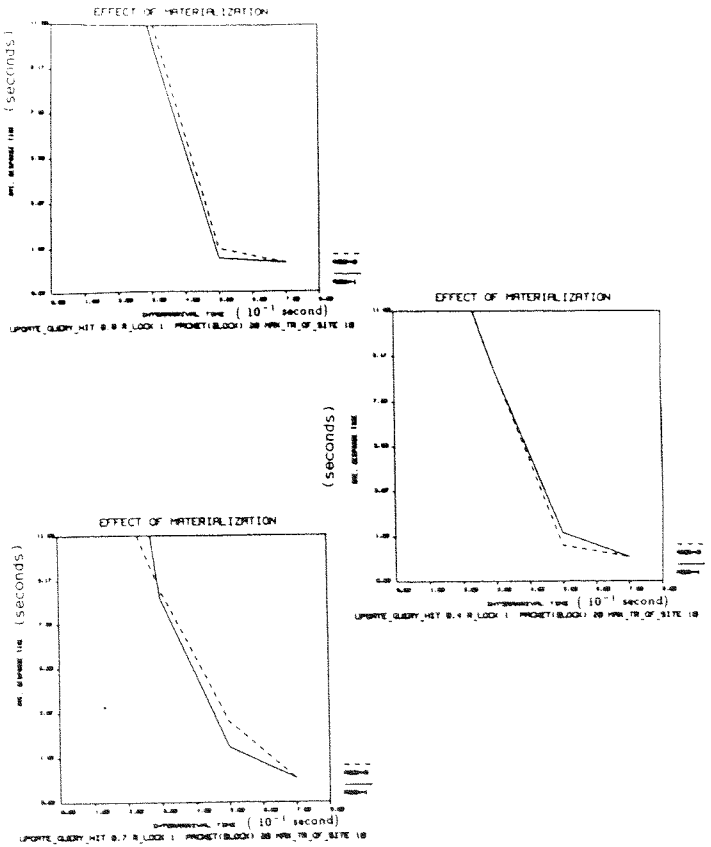


Figure 6.14 The effect of materialization to the mean response time.

6.8 Summary

In this chapter, we have developed a queuing model to simulate DDBLMN on a UNIX systems. The simulator DDBLMN-SIM implements the queuing model using DDBLMN-SMPL, which is a queue management package embedded in DDBLMN-SIM. DDBLMN was simulated under various conditions and control strategies. The statistical results show that DDBLMN is feasible and correct. The relative performance of various control strategies are compared in the simulations. The results are consistent with the expectation projected from the qualitative analysis.

The absolute values obtained from the simulation may not have significant meaning to a system designer who wants to adapt the simulation results into a real system. However, the relative performance among various control strategies would be a precious reference.

CHAPTER VII

CONCLUSIONS

7.1 Summary and Conclusions

In this research, we studied the design of a distributed database on a local computer system connected by a multiaccess network. Three key issues are addressed: data distribution, distributed query processing, and concurrency control. Based on the characteristics of local multiaccess networks, efficient strategies are proposed. The system is simulated in an event-driven queuing model to show the correctness, feasibility, and efficiency.

Due to the broadcast capability of multiaccess networks, each update can be processed at a cost independent of the allocation and the number of copies of files to be allocated. As a result, the file allocation problem, which is a subproblem of data distribution, is simplified from the problem of "where to locate copies of files?" on general networks to the problem of "should a copy of the file be located in a particular site?" Some special cases are reduced from NP-hard on general networks to polynomially solvable on local multiaccess networks. Although the general problem remains NP-hard, the complexity is reduced by decomposing the problem into N^{2m} knapsack problems. Since there exists fully polynomial approximation algorithms for the knapsack problem, efficient heuristic solutions can be easily developed based on this framework.

A transaction processing model that integrates query processing and concurrency control is proposed in the thesis. In identifying the target data in query processing, redundant materialization and semi-joins with complement transmissions are adopted. Redundant materialization helps reduce the size of intermediate results during the target data identification. On local multiaccess networks, a broadcast attribute can be semi-joined with attributes at all sites simultaneously, hence reducing the amount of data transfer as compared to that of general networks. The broadcast network allows efficient distribution of status information and permits queries to be scheduled dynamically based on the current status of the database. Although suboptimal, dynamic query processing eliminates the errors incurred in estimating status information in static algorithms, and has been found to perform better than static algorithms. An extremum-identification algorithm is proposed to aid the dissemination of status information. One of the potential direction for future research on query processing is intelligent query processing, in which knowledge is collected and extracted during the system operation and employed in later query processing. Due to the prematurity of the distributed databases, there is no sufficient knowledge available in this field to design a meaningful intelligent system. Only a model is proposed for intelligent query processing in this thesis, and detailed solutions are opened to future research.

This thesis also studies the query processing problem in future communication networks, in which the communication cost between any pair of nodes only depends on a fixed constant cost. The volume of transferred data is no loger a dominating factor to the total communication cost. Instead, the cost of communication channel usage dominates the overall cost. The DQP problem can be formulated into the generalized traveling salesman problem in

this case. A number of solution methods are considered. Among them, the most promising one that can be applied in the near future is to transform the problem into the standard traveling salesman problem and then solve it with heuristic algorithms. This strategy was evaluated. Results show that the algorithm is efficient with insignificant sacrifice to accuracy.

In contrast to conventional approach, our proposed concurrency control algorithm is embedded into dynamic query processing such that potential blocking due to unavailability of queried relations can be avoided. As a result of using multiaccess/broadcast networks, an efficient, serializable, and deadlock-free concurrency control algorithm is developed in the thesis. The broadcast capability allows locks to be known to all sites simultaneously, and hence eliminates the use of explicit lock messages. The serial transmission property of the network allows the database to be driven by network events. Therefore, the network itself serves as a synchronization mechanism.

Finally, the system is simulated based on a queuing network model. The correctness and the feasibility are demonstrated by the simulations. Furthermore, different strategies under various conditions are also compared by simulations.

In summary, the use of local multiaccess/broadcast networks not only simplifies the design of distributed databases, but also reduces the processing overhead. As a consequence, the distributed approach for a database in a local environment becomes promising as the technology of small computer systems and the local area networks advances.

7.2 Suggestions For Future Research

There remains problems to be studied in the future. These include the problem of recovery when the network or a site fails, the reliable broadcast of messages, the use of multiple broadcast busses or a bus with high bandwidth, the necessary modification on the operating system to support real-time processing of information received from the bus, the dynamic file allocation problem (file migration problem), the integration of file allocation with transaction processing, better solutions for GTSP that models future high speed communication networks, and the intelligent query processing. Some of them are discussed in the rest of this chapter.

7.2.1 High Bandwidth and Clustered Local Multiaccess Networks

The most serious drawback of a multiaccess/broadcast network with a single bus is that only serial transmission is allowed on the bus. Using current communication technologies, the bus may become the bottleneck of the system when network activities increase. One way to enhance the transmission capability is to use high speed transmission medium, such as optical fibers, with enhanced network interfaces. Another way is to use multiple bus networks. Both approaches retain the broadcast capability. Therefore, there may be only a little effect on our proposed concurrency control algorithms. For the query processing problem, the existing algorithms may have to be changed since the overhead of local processing is more significant. As discussed in Chapter IV, the semi-join approach may no longer be adequate in query processing. Totally new strategies may be needed. For the file allocation problem, the integer programming approach may remain adequate, but the solutions are much more complicated.

As the number of computer systems and other devices in the network grows, a single multiaccess network may not be sufficient to support the entire system. It is not unusual that the computing facilities owned by an organization are made by different vendors. The systems made by a a particular vendor are grouped as a cluster and interconnected by a distinct vender-supported network with its own communication protocols. These "local" networks are interconnected by one or more backbone networks as a logically integrated network. The Andrew system of the Carnegie-Mellon University is an example of such systems [MOR86]. Currently, Andrew consists of 17 Ethernets and 2 ProNet rings linked by fiber-optic cables. Over 600 computers are supported by the network currently, and over 5000 workstations are to be supported in the future. It is perceivable that this type of frameworks will become one of the most popular computing environments in the near future due to the success of the Andrew project.

Although Ethernet type networks seem to be the most attractive alternative for each individual network, the characteristics of the network as a whole may not be the same as that of a single Ethernet network. For example, the communication overhead across different networks is obviously higher than that within a local network. Different control strategies are required for a DDBMS on this type of environment. For example, the overhead of remote file access and update is no longer independent of the locations of files. Each local network may have independent communication such that communications can proceed in all local networks simultaneously. The network no longer serves as a synchronization mechanism for concurrency control. Furthermore, new techniques to collect statistics are needed to support dynamic query processing.

7.2.2 Integrated Design of FAP and DQP

In general, the file allocation and query processing problems are interdependent of each other. To process a query, the allocation of files should be known. Likewise, to allocate files, the query access pattern and query processing strategies should also be known. Currently, only the allocation of files is considered in query processing, while query processing strategies and query patterns are not considered in the file allocation. Also, each query is assumed to be accessing a single file. Solving query processing and file allocation problems together is extremely complex, especially when delay and availability constraints are also included. Apers proposed a solution for general networks [APE82]. The problem is solved as follows. First, the query processing order is optimized independently for each query or update using distinct copies of files. Distinct copies are located at virtual sites and the file allocation problem is the mapping of the virtual sites to physical sites such that the total communication costs are minimal. This framework may not be the best approach for a DDBLMN, which uses dynamic scheduling and redundant materialization. In allocating files for a system with dynamic scheduling, it is really difficult to predict the queries a system may take during query processing. As a result, the schedules that the file allocation relied upon may be quite different from the real schedule that the query scheduler generates.

One of the key factors to reduce the communication overhead in query processing is to store as many related relations as possible at a site, where the related relations are the relations to be joined. One of the better approaches we are considering is to impose such factors into the file allocation algorithm such that it tends to allocate queried relations by a transaction into as few sites as possible, in addition to allocating a file to sites that frequently access it. We

believe that this approach will be a good direction for future research to integrate FAP and DQP together.

7.2.3 File Migration problem

The problem of file migration arises from the dynamic nature of queries. Although there may be a locality of access for a file, there are occasionally very few activities inside the locality, and the file is accessed outside its locality. Moreover, the locality of access may be time-varying, as in global networks distributed over different time zones. It is efficient to allow the file to migrate to major access sites.

A typical migration method is to examine the allocation periodically and to reallocate a file if necessary. Over a period of t intervals, there are a minimum of 2^t possible alternatives. The problem of determining the best time to execute a migration with minimum overhead is NP-hard on general network. The problem complexity on DDBLMN may be reduced, however, efficient solutions are still opened to future research.

7.2.4 Intelligent Query Processing

As we mentioned in Section 7.1, the detailed design of intelligent query processing is still open. The problem includes the representation, acquisition and abstraction of knowledge, learning module design, expert scheduler design, knowledge base design, as well as the interactions among system components. The major difficulty is on the knowledge acquisition due to the lack of expertise in this field. Comparing to other intelligent systems, we do not have good knowledge on query processing as the internal database activities are transparent implicitly to the users. Thus, a powerful learning model is the key

design issue of such intelligent query processing systems.

7.2.5 Generalized Traveling Salesman Problem

The performance of the TSP-transformation approach for the GTSP problem is obviously bounded by the performance of the TSP problem. Furthermore, the problem size after transformation is expanded to N=3n+m, where n is the number of nodes and m is the number of clusters. For a large problem, even a complexity of $O(N^3)$ could be substantial. In this sense, a direct-solving solution without transformation may be a good candidate to provide better solutions. One of the potential approaches we are studying is the "decomposition" approach, in which the GTSP problem is decomposed into several sub-problems. An example algorithm in this approach is shown as follows.

Example GTSP Algorithm 1

(a) Select an arbitrary node from each cluster.

(b) Connect all selected nodes together as a TSP tour.

(c) Refine the solution.

There are many ways to select nodes from clusters in step (a). Obviously, the center node, which is the node closest to the geographic center of the cluster, is a good candidate. Another good alternative is the node closest to the geographic center of the graph.

Among various potential algorithms we will consider for the refinement process, the following *rubber-band* process is one of the promising approaches.

Rubber-Band Process

Assume the TSP path $P=(0, 1, \ldots, n)$ is produced in step (b) and node i is

the node selected from cluster r in step (a). Consecutive nodes on the TSP path is numbered consecutively, and node 0 is assumed to be the node closest to the geographic center.

For all k **in** $(0,1,\ldots,n)$ **do**

set $k_1 = (k+1) mod (n+1)$

set $k_2 = (k+2) mod (n+1)$

draw a straight line between nodes k and k_2

from cluster $k+1$, select node k_1'' which is closest to the line

replace node k_1 in P with k_1''

done

The solution can be further refined as many times as needed. Such a process is analogous to the following process:

"Consider each node as a pole, a rubber band is used to circle at least one pole from each cluster with the minimum length."

It is obvious that a necessary condition for an optimal solution is that the rubber band should touch the pole closest to the line drawn between the two poles selected from two neighboring clusters.

Since the algorithms are heuristic, substantial experimentation is required to study the the performance of different algorithms under different conditions. Currently, there is no way to draw any conclusion from the above discussion. However, this could be a good direction for future research.

LIST OF REFERENCES

LIST OF REFERENCES

[AHO74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analusis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974, pp. 199-200.

[APE82] P. M. G. Apers, "Centralized or Decentralized Data Allocation," *Distributed Data Sharing Systems*, R. P. Van de Riet and W. Litwin, eds., North-Holland, New York, 1982.

[APE83] P. M. G. Apers, A. R. Hevner, and S. B. Yao, "Optimization Algorithms for Distributed Queries," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 1, Jan. 1983, pp. 57-68.

[BAN79] J-S Banino, C. Kaiser, and H. Zimmermann, "Synchronization for Distributed Systems using a Single Broadcast Channel," *Proceedings of the 1st Int'l Conf. On Distributed Computing Systems*, Huntsville, Oct. 1979, pp. 330-338.

[BEA59] J. Bearwood, J. H. Halton and J. M. Hammersley, "The Shortest Route Through Many Points," *Proceedings of the Cambridge Philosophical Society 55*, 1959, pp. 299-327.

[BEL62] R. Bellman, "Dynamic Programming Treatment of the Traveling Salesman Problem," *JACM*, Vol. 9, No. 1, 1962, pp. 61-63.

[BEL68] M. Bellmore and G. L. Nemhauser, "The Traveling Salesman Problem: A Survey," *Operations Research*, Vol. 16, 1968, pp. 538-558.

[BEL71] M. Bellmore and J. C. Malone, "Pathlogy of Traveling Salesman Subtour-Elimination Algorithms," *Operations Research*, Vol. 19, 1971, pp. 278-307.

[BEN83] J. L. Bentley, "A Case Study in Applied Algorithm Design: The Traveling Salesman Problem," *Dept. of Comp. Sci. and Math., Carnegie-Mellon Univ.*, April 7, 1983.

[BER81a] P. A. Bernstein and D. M. Chiu, "Using Semi-Joins to Solve Relational Queries," *JACM*, Vol. 28, No. 1, Jan. 1981, pp. 25-40.

[BER81b] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[BER81c] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems*, Vol. 6, No. 4, Dec. 1981, pp. 602-625.

[BHA82a] B. Bhargava and C. T. Hua, "Classes of Serializable Histories and Synchronization Algorithms in Distributed Database Systems," *Proceedings of IEEE International Conference on Distributed Computing Systems*, Miami, Florida, Oct. 1982, pp. 438-446.

[BHA82b] B. Bhargava, "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comarison," *Proceedings of IEEE International Conference on Distributed Computing Systems*, Miami, Florida, Oct. 1982, pp. 508-517.

[BHA83] B. Bhargava and C. T. Hua, "A causal Model for Analyzing Distributed Concurrency Control Algorithms," *IEEE Trans. On Software Engineering*, Vol. SE-9, NO. 4, July 1983, pp 470-486.

[BIR82] A. Birrell, "Grapevine: An Exercise In Distributed Computing," *CACM*, Vol. 15, No. 4, April 1982, pp 260-273.

[BUR79] R. E. Burkard, "Travelling Salesman and Assignment Problems: A Survey," *Annals of Discrete Mathematics*, Vol. 4, pp. 193-215, North-Holland, 1979

[CER82] S. Ceri, G. Paolini, G. Pelagatti and F. A. Schreiber, " Distributed Database Research at the Politecnico Milano," *IEEE Data Engineering*, Vol. 5, No. 4, Dec. 1982, pp. 9-13.

[CER83] S. Ceri, S. Navathe, and G. Wiederhold, "Distribution Design of Logical Database Schemas," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 4, July 1983, pp. 487-504.

[CER84] Stefano Ceri and Giuseppe Pelagatti, *"Distributed Databases: Principles and Systems,"* McGraw-Hill, New York, 1984.

[CHA83] J. M. Chang, "LAMBDA: A Distributed Database System" Bell Telephone Lab. Tech. Report, March 1983.

[CHA84a] J. S. Chang, "Simplifying Distributed Database Systems Design by Using a Broadcast Network," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, June 1984, pp. 223-233.

[CHA84b] J. M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols" ACM Trans. on Computer Systems, Vol 2, No 3, August 1984, pp. 251-273.

[CHA85] J. S. Chang, "LAMBDA: A Distributed Database System for Local Area Networks," *Database Engineering*, Vol. 8, No. 2, June 1985, pp. 76-83.

[CHE84] A. L. P. Chen, and V. O. K. Li, "Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems," *IEEE Trans. on Computers*, Vol. C-33, No. 11, Nov. 1984, pp. 959-967.

[CHI84] D. M. Chiu, P. A. Bernstein, and Y. C. Ho, "Optimizing Chain Queries in a Distributed Database System," *SIAM J. of Comp.*, Vol. 13, No. 1, Feb. 1984, pp. 116-134.

[CHR72] N. Christofides and S. Eilon, "Algorithms for Large-scale Travelling Salesman Problems," *Operational Research Quarterly*, Vol. 23, No. 4, 1972, pp. 511-518.

[CHU69] W. W. Chu, "Multiple File Allocation in a Multiple Computer System," *IEEE Trans. on Computers*, Vol. C-18, No. 10, Oct. 1969, pp. 885-889.

[CHU82] W. W. Chu, and P. Hurley, "Optimal Query Processing For Distributed Database Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 9, Sept. 1982, pp. 835-850.

[CLA78] A. Claus, "A Simultaneous Enumeration Approach to the Traveling Salesman Problem," *Studies In Applied Math.*, Vol. 58, 1978, pp. 159-163.

[COD70] E. F. Codd, "A Relation Model of Data for Large Shared Data Banks," *CACM*, Vol. 13, 1970, pp. 377-387.

[DAN54] G. B. Dantzig, D. R. Fulkerson, and S. Johnson, "Solution of a Large-Scale Traveling-Salesman Problem," *Operations Research*, Vol. 2, 1954, pp. 393-410.

[DOW82] L. W. Dowdy and D. V. Foster, "Comparative Models of the File Assignment Problem," *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 287-313.

[ESW74] K. P. Eswaran, "Placement of Records in a File and File Allocation in a Computer Network," *Information Processing 74*, IFIPS, North Holland Publishing Co., 1974.

[FIS78] G. S. Fishman, "Principles of Discrete Event Simulation," *Wiley-Ingerscience, 1978. Management Science*, Vol. 18, No. 9, May 1972, pp. 465-491.

[GEO72] A. M. Geoffrion and R. E. Marsten, "Integer Programming: A Framework and State-of-art Survey," *Management Sci., Vol. 18, No. 9, pp. 465-491, May 1972.*

[GIL64] P. C. Gilmore and R. E. Gomory, "Sequencing A One State-Variable Machine: A Solvable Case of the Traveling Salesman Problem," *Operations Research*, Vol. 12, Sep.-Oct., 1964, pp. 655-679.

[GON62] *Solution to the Traveling Salesman Problem by Dynamic Programming on the Hypercube*, Tech. Rep. No. 18, O.R. Center, M.I.T., 1962.

[GOU81] M. G. Gouda and U. Dayal, "Optimal Semi-join Schedules for Query Processing in Local Distributed Database Systems," *Proc. ACM SIGMOD Conference*, May 1981, pp. 164-175.

[GUP68] J. N. D. Gupta, "Travelling Salesman Problem-A Survey of Theoretical Developments and Applications," *Opsearch*, Vol. 5, 1968, pp. 181-192.

[GUP78] J. N. D. Gupta, "A Search Algorithm for the Traveling Salesman Problem," *Compu. & Ops. Res.*, Vol. 5, 1978, pp. 243-250.

[HAR62] W. W. Hardgrave and G. L. Nemhauser, "On the Relation Between the Traveling-Salesman and the Longest-Path Problems," *Operations Research*, Vol. 10, No. 5, Jan.-Feb., 1962, pp. 647-657.

[HEL62] M. Held, and R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *SIAM*, Vol. 10, No. 1, 1962, pp. 196-210.

[HEL70] M. Held, and R. M. Karp, "The Traveling Salesman Problem and Minimum Spanning Trees," *Operations Research*, Vol. 18, No. 6, 1970, pp. 1138-1162.

[HEL71] M. Held and R. M. Karp, "The Traveling Salesman Problem and Minimum Spanning Trees: Part II," *Math. Prog.*, Vol. 1, pp. 6-25, North-Holland, 1971.

[HEV79a] A. R. Hevner, *Query Processing in Distributed Database Systems*, Ph.D. Dissertation, Computer Sciences Department, Purdue University, CS79-4, Dec. 1979.

[HEV79b] A. R. Hevner, "The Optimization Of Query Processing On Distributed Database Systems," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 177-187.

[HEV85] Alan R. Hevner, O. Qi Wu, and S. B. Yao, "Query Optimization on Local Area Networks," *ACM Transaction on Office Information*, Vol. 3, No. 1, Jan. 1985, pp. 35-62.

[HON76] S. Hong, "The Asymmetric Traveling Salesman Problem and Spanning Arborescences of Directed Graphs," *Proceedings of Symposium on Operations Research*, Heidelberg, Nov. 1976.

[HOU80] D. J. Houck Jr., J. C. Picard, M. Queyranne, and R. R. Vemuganti, "The Traveling Salesman Problem As A Constrained Shortest Path Problem: Theory and Computational Experience," *Opsearch*, Vol. 17, No. 2&3, 1980, pp. 93-109.

[IEEE83] Institute for Electrical and Electronic Engineers, "IEEE project 802, Local Area Network Standards," 1983.

[JUA84] J. Y. Juang and B. W. Wah, "Unified Window Protocols for Contention Resolution in Local Multiaccess Networks," *Proc. of 1984 IEEE INFOCOM*, San Francisco, CA, April 1984, PP. 97-104.

[KAR64] R. L. Karg and G. L. Thompson, "A Heuristic Approach to Solving Travelling Salesman Problems," *Management Science*, Vol. 10, No. 2, Jan., 1964, pp. 225-248.

[KAR79] R. M. Karp, "A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem," *SIAM J. Computing*, Vol. 8, No. 4, Nov., 1979, pp. 561-573.

[KER82] Larry Kerschberg, Peter D. Ting, and S. Bing Yao, "Query Optimization in Star Computer Networks," *ACM Transactions On Database Systems*, Vol. 7, No. 4, Dec. 1982, pp. 678-711.

[KRI84] R. Krishnamurthy and S. P. Morgan, "Distributed Query Optimization: An Engineering Approach," *Proc. Int'l Conf. on Data Engineering*, Los Angeles, CA, April 1984, pp. 220-227.

[KUM70] S. Kumar, "Optimal Path through k Specified Sets of Nodes," *Indian Journal of Mathematics*, Vol. 12, No. 1, pp. 25-30, Jan . 1970.

[KUN81] H. T. Kung and J. T. robinson, "Optimistic Methods for Concurrency Control," *ACM Trans. On Database Systems*, Vol. 6, No. 2, pp. 213-226, Jan . 1981.

[LIN65] S. Lin, "Computer Solutions of the Traveling Salesman Problem," *Bell System Tech. Journal*, Vol. 44, 1965, pp. 2245-2269.

[LIN84] B. G. Lindsey, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost, "Computation and Communication in R^*," *ACM Trans. On Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 24-38.

[LIT63] J. D. C. Little, K. G. Murty, D. W. Sweeney, and G. Karel, "An Algorithm for the Traveling Salesman Problem," *Operations Research*, Vol. 11, Nov.-Dec., 1963, pp. 972-989.

[LU85] Hongjun Lu, *Distributed Query Processing With Load Balancing In Local Area networks*, Ph.D. Dissertation, Computer Sciences Department, University of Wisconsin-Madison, Computer Sciences technical Report #624, 1985.

[MAC80] M. H. MacDougall, "SMPL - A Simple Portable Simulation Language," *Amdahl Technical Report*, April, 1980.

[MET76] R. F. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *CACM*, Vol. 19, No. 7, July 1976, pp. 395-404.

[MIL76] P. Miliotis, "Integer Programming Approaches to the Travelling Salesman Problem," *Math. Prog.*, Vol. 10, 1976, pp. 367-378.

[MIL78] P. Miliotis, "Using Cutting Planes to Solve the Symmetric Traveling Salesman Problem," *Math. Prog.*, Vol. 15, pp. 177-188, North-Holland, 1978.

[MOR86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Hovard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *CACM*, Vol. 29, No. 3, Mar. 1986, pp. 184-201.

[NGU81a] Gia Toan Nguyen and G. Sergeant, "Distributed Architecture and Decentralized Control for a Local Network Database System," *Proceedings of ACM International Computing Symposium*, London, March 1981.

[NGU81b] Gia Toan Nguyen, "Distributed Query Management for a Local Network Database System," *Proceedings of the Second International Conference on Distributed Computing Systems*, Paris, April 1981.

[PAN64] S. N. N. Pandit, "Some Observations On the Longest Path Problem," *Operations Research*, Vol. 12, No. 2, 1964, pp. 361-364.

[PAP79] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates," *JACM*, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[PAP82] C. H. Papadimitriou, "A Theorem in Database Concurrency Control," *JACM*, Vol. 29, No. 4, Oct. 1982, pp. 998-1006.

[PEE78] Richard Peebles and Eric Manning, "System Architecture for Distributed Data Management," *IEEE Computer*, Vol. 11, No. 1, Jan. 1978, pp. 40-47.

[POP83] G. J. Popek and G. Thiel, "Distributed Data Management Issues in the LOCUS System," *IEEE Data Engineering*, Vol. 6, No. 2, June 1983, pp. 63-68.

[RAM83] C. V. Ramamoorthy and B. W. Wah, "The Isomorphism of Simple File Allocation," *IEEE Trans. On Computers*, Vol. C-32, No. 3, March 1983, pp. 221-232.

[ROT80] J. Rothnie, P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)," *ACM Trans. On Database Systems*, Vol. 5, No. 1, March 1980, pp. 1-17.

[SAC84] G. M. Sacco, "Distributed Query Evaluation In Local Area Networks," *Proc. of Int'l Conf. On Data Engineering*, Los Angeles, CA, April 1984, pp. 510-516.

[SHO82] J. H. Shoch, Y. K. Dalal, D. D. Redell, and R. C. Crane, "Evolution of the Ethernet Local Computer Network," *IEEE Computer*, Vol. 15, No. 8, Aug. 1982, pp. 10-27.

[SRI70] S. S. Srivastava, S. Kumar, R. C. Garg and P. Sen, "Generalized Traveling-Salesman Problem through n Sets of Nodes," *Journal of the Canadian Operational Research Society*, pp. 97-101, 1970.

[STA84] William Stallings, "Local Networks," *ACM Computing Surveys*, Vol. 16, No. 1, Mar. 1984, pp. 3-42.

[STA85] William Stallings, *Data And Computer Communications*, Macmillan, 1985.

[STO77] M. Stonebraker and E. Neuhold, "A distributed Data Base Version of INGRES," *Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977, pp. 19-36.

[STU80] H. Sturgis, J. Michell, and J. Israel, "Issues In The Design And Use Of A Distributed File System," *ACM Operating System Review*, Vol. 14, No. 3, July 1980, pp. 55-69.

[SYK80] David J. Sykes, "The Economics Of Distributed Systems," *Digest of Papers, COMPCOM*, Fall 1980, pp. 8-15.

[THO79] R. H. Thomas, "A Majority Consensus Approach for Concurrency Control," *ACM Trans. On Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.

[ULL82] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, 1982.

[WAH83] B. W. Wah and J. Y. Juang, "An Efficient Protocol for Load Balancing on CSMA/CD Networks," *Proc. 8th Conference on Local Computer Networks*, Minneapolis, MN, Oct. 1983, pp. 55-61.

[WAH84a] B. W. Wah, "File Placement on Distributed Computer Systems," *IEEE Computer*, Vol. 17, No. 1, Jan. 1984, pp. 23-32.

[WAH84b] B. W. Wah and Y. N. Lien, "The File-Assignment And Query-Processing Problems In Local Multiaccess Networks," *Proc. of Int'l Conf. On Data Engineering*, Los Angeles, CA, April 1984, pp. 228-235.

[WAH85] B. W. Wah and Y. N. Lien, "Design of Distributed Databases On Local Computer Systems With A Multiaccess Network," *IEEE Trans. On Software Eng.*, Vol. SE-11, No. 7, July 1985, pp. 606-619.

[WAH85b] B. W. Wah and J. Y. Juang, "Resource Scheduling for Local Computer Systems with a Multiaccess Network ," *IEEE Trans. on Computers*, Vol. C-34, No. 12, Dec. 1985, pp. 1144-1157.

[WON77] E. Wong, "Retrieving Dispersed Data From SDD-1: A System for Distributed Databases," *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, May, 1977, pp. 217-235.

[YAO77] S. B. Yao, "Approximating Block Accesses In Database Organization," *CACM*, Vol. 20, No. 4, April 1977, pp. 260-261.

[YAO79] S. B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Trans. On Database Systems*, Vol. 4, No. 2, June 1979, pp. 133-155.

[YU82] C. T. Yu, C. C. Chang, and Y. C. Chang, "Two Surprising Results in Processing Simple Queries in Distributed Databases," *Proc. COMPSAC 82*, Nov. 82, pp. 377-384.

[YU83] C. T. Yu and C. C. Chang, "On The Design of A Query Processing Strategy In A Distributed Database Environment," *Proc. SIGMOD Conference*, Vol. 13, No. 4, 1983, pp. 30-39.

[YU84a] C. T. Yu and C. C. Chang, "Distributed Query Processing," *ACM Computing Surveys*, Vol. 16, No. 4, Dec. 1984. pp. 399-433.

[YU84b] C. T. Yu, Z. M. Ozsoyoglu, and C. C. Chang, "Optimization of Distributed Tree Queries," *Journal of Computer and System Sciences*, Vol. 29, No. 3, Dec. 1984. pp. 409-445.

APPENDIX


```
/*****************************************************************
 *  NAME          : DDBLMN-SIM Simulator
 *  PURPOSE       : To simulate a DDBMS on a local multiaccess network
 *                  (DDBLMN)
 *  REFERENCE     : S. W. Wah and Y. N. Lien,
 *                  Design of Distributed Databases On Local Computer
 *                  Systems With A Multiaccess Network,
 *                  IEEE Trans. On Software Engineering,
 *                  vol. SE-11, No. 7, July 1985, pp. 606-619
 *
 *  STD INPUT     :
 *                  seed    seed to random number generator ( int )
 *                  length of execution time (second)
 *                  mean transaction arrival time  (0.1 second)
 *                  fraction of updated transaction (int 0-10)
 *                  time to release read lock ( 1  ANYTIME, 2  OBJ_END )
 *                                            ( 0  BT_END )
 *  STD OUTPUT    : DDBLMN SMPL information
 *  FILE OUTPUT   : a line containing following information is
 *                  output  to the file 'ddblmn.data'  at the end of
 *                  a simulation execution
 *                  average transaction processing time in second
 *                  average ( ratio of the time spent on bus to the total
 *                    processing time of TR ) on total transactions
 *                  average ( ratio of the time spent on disk to the total
 *                    processing time of TR ) on total transactions
 *                  average ( ratio of the time spent on cpu to the total
 *                    processing time of TR ) on total transactions
 *                  lost transaction per second
 *                  utilization of disk at site 1
 *
 *                  average queue length of disk at site 1
 *                  utilization of disk at site 2
 *                  average queue length of disk at site 2
 *                  . . . . . .
 *                  . . . . . .
 *                  utilization of disk at site SITE
 *                  average queue length of disk at site SITE
 *
 *                  utilization of network interface  at site 1
 *                  average queue length of network interface  at site 1
 *                  utilization of network interface  at site 2
 *                  average queue length of network interface  at site 2
 *                  . . . . . .
 *                  . . . . . .
 *                  utilization of network interface  at site SITE
 *                  average queue length of network-interface  at site SITE
 *
 *                  utilization of bus
 *                  average queue length of bus
 *
 *                  CPU-TIME in second  used in the simulation
 *
 *  COMPILING     : cc filename -lm
 *  DESCRIPTION   : See Chapter 6
 *  AUTHOR        : Yao Nan Lien,  EE,  Purdue University
 *  DATE          : August 6, 1986
 *****************************************************************/



#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "smpl.h"
      /*
       *    double   expotl(),
       *       uniform(),
       */
/*-----------------------------------------------------------------
 *
 *                System configuration  constants
 *-----------------------------------------------------------------*/

#define SITE            4
#define R               4
#define TR              100     /* tr pool size */
#define JA              4

#define TOKEN           10000   /* token pool size */

/*-----------------------------------------------------------------
 *      Following parameters is used in query generation
 *-----------------------------------------------------------------*/

#define QUERY_REL_HIT   8       /* hit ratio of queried relations  */
```

```
#define TARGET_REL_HIT  4       /* bit ratio of target relations   */
#define QUERY_ATT_HIT   4       /* bit ratio of queried attributes */
#define TARGET_ATT_HIT  4       /* bit ratio of target attributes  */
#define UPDATE_ATT_HIT  4       /* bit ratio of updated attributes */

/*-------------------------------------------------------------------
*
*               TIMING PARAMETERS  ( unit  1 micro second )
*
*------------------------------------------------------------------*/

#define PRE_TIME        1000.0  /* timing */
#define DELTA_TIME      100.0   /* tiny time used in state transition */
#define INSTRUCTION_TIME 1.0
#define CPU_TIME_PER_BIT 3.0/8* INSTRUCTION_TIME
                                /* time to process a bit */
#define BUS_TIME_PER_BIT 1.0    /* ave bus time to transmit a bit */
#define CONTENTION_TIME 50*BUS_TIME_PER_BIT
                                /* 50 bits for 1 contention slot */

#define DTIME_SEARCH    20000.0 /* Max disk search time */
#define DTIME_PER_BLOCK 1000.0  /* 1 ms disk transfer time per block */

/*-------------------------------------------------------------------
*               Data Format Constants
*------------------------------------------------------------------*/

#define BITS_OF_BYTE    8
#define BYTES_OF_BLOCK  1000
#define BLOCK_SIZE      BITS_OF_BYTE * BYTES_OF_BLOCK
#define PACKET_OVERHEAD 208 /* same as Ethernet specification */
/*-------------------------------------------------------------------
*               Flags and Other Constants
*------------------------------------------------------------------*/

#define VAXMAX          2147483647.0
#define INF             2147483647
#define MEG             1000000
#define DEVIATION       0.3

#define ERROR           INF
#define FAIL            -1
#define SUCC            1
#define NO              0
#define YES             1
#define QUEUED          1
#define BUSY            0
#define IDLE            0
#define NONIDLE         1

#define REJECT          0
#define GOT_LOCK        1
#define WAIVE           2
#define SAME            0
#define DIFF            1
#define VALID           0
#define INVALID         1
#define USED            2
#define FIRMED          2
#define LbeforeR        1
#define RbeforeL        -1

#define ANYTIME         1
#define GSJ_END         2
#define RT_END          0

#define BLOCKED         0
#define RUN             1       /* process state */
#define SLEEP           2       /* process state */

#define TM              0       /* process id */
#define NETWORK         TR+1    /* process id */

#define REMOTE          0       /* a remote query, regular  */
#define LOCAL_READ      1       /* needed rels are locally available */
#define LOCAL_QUERY     2       /* same as above and no remote update */

#define HOME_SITE       2
#define CHILD_SITE      1

#define HAD_BRO         1       /* site att had been broadcast */
#define WAIT_BRO        2       /* site att had not been broadcast */
#define NOT_BRO         0       /* site att had not been broadcast */

#define TRID            1       /* event queue data */
#define SITEID          2       /* event queue data */
#define EVENTID         3       /* event queue data */

/*-------------------------------------------------------------------
*               Transaction Phases
*------------------------------------------------------------------*/

#define LP              2
#define LP_SYN          3
```



```
#define GSJ             4
#define GSJ_SYN         5
#define RT              6
#define PP              7

/*-------------------------------------------------------------------
*               Lock Transaction States
*------------------------------------------------------------------*/

#define WANT_R          1
#define READING         2
#define HAD_READ        3
#define WANT_W          4
#define WRITING         5
#define WRITTEN         6
#define WANT_RW         7
#define READING_WANT_W  8
#define READING_WRITTING 9
#define HAD_RW          10

/*-------------------------------------------------------------------
*               EVENTs
*------------------------------------------------------------------*/

#define NEW_TR                  0
#define CONTEND_BUS             1
#define RELEASE_BUS             2
#define NEED_DISK               3
#define RELEASE_DISK            4
#define LP_FINISH               5
#define GSJ_FINISH              6
#define PP_FINISH               7
#define WAKE_UP                 8
#define NEED_BUS_TOKEN          9
#define RELEASE_BUS_TOKEN       10
#define STOP                    11

/*-------------------------------------------------------------------
*               Transaction Handler States
*------------------------------------------------------------------*/

#define PRE                     1       /* preprocessing        */
#define LP_INI                  2       /* LP initial           */
#define LP_PRE_LOCK             3       /* getting local lock   */
#define LP_WAIT_LOCK            4       /* waiting for locks    */
#define LP_DISK                 5       /* LP reading data using disk */
#define LP_SJ                   6       /* LP local semi-join   */
#define LP_SYN_EST_BRO          7       /* set site broad "finish"  */
#define LP_WAIT_SYN             8       /* waiting for end of LP sync */
#define LP_ALL_FINISH           9       /* all sites finished LP    */
#define LP_END                  10      /* LP phase is ending       */
#define LP_WAIT_END             11      /* waiting for the end of LP  */
#define GSJ_INI                 12      /* GSJ initial              */
#define GSJ_WAIT_LOCK           13      /* for those need lock only */
#define GSJ_WAIT_SJ             14      /* waiting for sj processing  */
#define GSJ_SJ_BRO              15      /* sj broadcast             */
#define GSJ_ALL_FINISH          16      /* all sites finished       */
#define GSJ_WAIT_SYN_END        17      /* waiting for the end of syn */
#define GSJ_WAIT_END            18      /* waiting for the end of GSJ */
#define RT_WAIT_START           19      /* waiting for the RT start   */
#define RT_START                20      /* start of RT              */
#define PP_WAIT_START           21
#define PP_START                22      /* start of PP              */
#define PP_PRE_LOCK             23      /* getting local write locks  */
#define PP_WAIT_LOCK            24
#define PP_WAIT_UPDATE          25      /* waiting for the end of RT  */
#define PP_UPDATE               26      /* start to updating          */
#define END                     27

/*-------------------------------------------------------------------
*               Some Input Variables (int)
*------------------------------------------------------------------*/

int     MAX_TR_OF_SITE  ;
int     SIMULATED_TR    ;
int     UPDATE_QUERY_RIT ;
int     mean_tr         ;
int     r_unlock_time   ;

/*-------------------------------------------------------------------
*               Some Queuing System Variables (int)
*------------------------------------------------------------------*/

int     event           ;
int     new_tr          ;
int     new_token       ;
int     bus             ;       /* bus to be reserved       */
int     finished_tr     ;       /* number of finished trans */
int     total_tr        ;       /* total number of trans    */

/*-------------------------------------------------------------------
*               Some Working Variables (int)
*------------------------------------------------------------------*/
```

```
int     I,J,K           ;
int     counter         ;
int     start_time      ;
int     end_time        ;
int     execution_time;
int     after           ;  /* timestamp for real execution time */
int     before          ;  /* timestamp for real execution time */
int     last_rt         ;
int     num_of_bro      ;
int     working_site    ;
int     minimum_site    ;
int     maximum_site    ;

/*-------------------------------------------------------------------------
*                   Some Accounting Variables
*------------------------------------------------------------------------*/

double  total_tr_time; /* total execution time of all tr       */
double  total_U_bus  ; /* total bus  busy ratio  of all tr     */
double  total_U_disk ; /* total disk busy ratio  on all sites  */
double  total_U_cpu  ; /* total cpu  busy ratio on all sites   */
double  ave_tr_time  ; /* average execution time of all tr     */
int     lost_customer; /* total number of lost customer        */
float   allowed_error; /* Allowable diff on ave tr time between 2 iter*/

/*-------------------------------------------------------------------------
*                System configuration  constants ( Double )
*------------------------------------------------------------------------*/

double  PACKET_SIZE_IN_BLOCK;
double  PACKET_TIME         ;
double  PACKET_SIZE         ;
double  CONTENTION_PACKET   ;
double  DATA_SIZE           ;  /* number of info  bit in a packet */
/*-------------------------------------------------------------------------
*                Working Variables ( Double )
*------------------------------------------------------------------------*/

double  X,Y,Z           ;
double  disk_time       ;
double  packet          ;
double  trace_time      ;
double  cc_time         ;   /* the time to execute cc_lock_get_bus */
double  iteration_error;

/*-------------------------------------------------------------------------
*
*                Integer arrays
*
*------------------------------------------------------------------------*/

int     blocked_by      [TOKEN+1];      /* who blocked this token */
                                        /* TR+1  by pguser != 0   */
int     blocked_by_lq   [TR+1];         /* # of block by local queries*/
int     blocked_site    [TR+1];         /* # of blocked sites in a tr */
int     bro_component   [TR+1]  [JA+1] [R+1]; /*NULL,  HAD_BRO */
int     bro_rel         [TR+1]  [R+1];        /*NULL,WAIT_BRO,HAD_BRO*/
int     bro_site_att    [TR+1]  [SITE+1] [JA+1];/* SAME*/
int     broadcasting_att [TR+1];
int     bus_token       [SITE+1];       /* bus_token to be reserved */
int     class_of        [TR+1];         /* REMOTE, LOCAL */
int     component_set   [TR+1]  [SITE+1] [JA+1] [R+1];  /*YES, NO*/
int     disk            [SITE+1];       /* disks to be reserved */
int     est_att_card    [TR+1]  [SITE+1] [JA+1]; /* est att card */
int     est_rel_card    [TR+1]  [SITE+1] [R+1]; /* estimated rel card */
int     est_slowest_site_of     [TR+1];
int     first_rt_site   [TR+1];         /* the first RT site of tr */
int     home_of         [TR+1];         /* the home site of  tr */
int     hv              [TR+1]  [SITE+1]; /* heuristic values  */
int     ini_rel_card    [R+1];          /* initial card of relations */
int     last_sj         [TR+1];         /* flag of last sj broadcast */
int     link_to_next_tr [TR+1];         /* the id of next tr */
int     local_lock      [SITE+1] [R+1];  /* local lock table  */
int     local_min_att   [TR+1]  [SITE+1];  /*  the minimum att */
int     lock_state      [TR+1]  [R+1];
int     lock_validity   [TR+1]  [SITE+1]; /* validity of held locks */
int     lost_tr         [SITE+1];       /* # of tr lost at site */
int     lptime          [TR+1]  [SITE+1]; /* proc time in LP phase */
int     min_hv          [TR+1];         /* smallest hv */
int     min_site        [TR+1];         /* site with smallest hv */
int     new_pg          [TR+1]  [TR+1];   /* newly updated pg */
int     new_relative    [TR+1]  [TR+1];   /* new relatives in pg */
                                        /* creat by get_local_lock */
int     not_lock_in_LP  [TOKEN+1];      /*YES, NO*/
int     num_in_gsj      [TR+1];         /* # of tb processing gsj */
int     num_in_lp       [TR+1];         /* # of tb in processing lp */
int     num_of_broad    [TR+1];         /* # of bro in extremum id */
int     num_of_tr_at    [SITE+1];       /* # of tr entering to site */
int     orphan          [TOKEN+1];      /* single child no home */
int     pg              [TR+1]  [TR+1];  /* precedence graph i --> j */
int     pguser          [SITE+1];       /* user of local_pg  */
                                        /* SAME : no user  */
int     picked_site     [TR+1];         /* the site randomly picked */
int     pptime          [TR+1]  [SITE+1]; /*  proc time in PP phase */
```



```
int     queried_att     [TR+1]  [R+1] [JA+1];   /* NO, YES, */
int     queried_rel     [TR+1]  [R+1];          /* NO, YES,*/
int     rel_card        [TR+1]  [R+1] [SITE+1]; /* relations cards */
int     related_site    [TR+1]  [SITE+1]; /* NULL,CHILD_SITE,HOME_SITE*/
int     relation_width  [R+1];
int     site_att_card   [TR+1]  [SITE+1] [JA+1]; /* card of site atts */
int     site_of         [TOKEN+1];   /* the site of a token
                                      * if tr_of[token]==0, it store
                                      * the link to the next empty cell
                                      */
int     site_rel_conf   [TR+1]  [SITE+1] [JA+1];        /* YES, NO*/
int     sjtime          [TR+1]  [SITE+1];  /*  proc time in GSJ phase */
int     target_jatt     [TR+1]  [R+1] [JA+1];
int     target_njatt    [TR+1]  [R+1];
int     target_rel      [TR+1]  [R+1];
int     th_state        [TR+1]  [SITE+1];
int     token_of        [TR+1]  [SITE+1]; /* token of tr at the site  */
int     tr_of           [TOKEN+1];      /* the id of next token */
int     tr_phase        [TR+1];         /* NULL,PRE,LP_GSJ,RT,PP*/
int     tr_rel_width    [TR+1]  [R+1];
int     trace_tr        [TR+1];
int     updated_jatt    [TR+1]  [R+1]   [JA+1];
int     updated_njatt   [TR+1]  [R+1];
int     updated_rel     [TR+1]  [R+1];
int     writing_rel     [TR+1]; /* assume at most 1 rel to be updated */

/*-------------------------------------------------------------------------
*
*                Float (double) Arrays
*
*------------------------------------------------------------------------*/

double  SELECTIVITY     [  R+1 ];               /* Testing argument */
double  time_of_tr      [ TR+1 ];               /* start time of a tr */
double  U_bus_tr        [ TR+1 ];               /* bus usage of tr */
double  U_disk_tr       [ TR+1 ];               /* disk usage of tr */
double  U_cpu_tr        [ TR+1 ];               /* cpu usage of tr */

/*-------------------------------------------------------------------------
*
*                Macro Definitions and some basic  definitions
*
*------------------------------------------------------------------------*/
#define pri(v)  ;
#define pre(v)  ;
#define out(v)  ;
#define ln      ;
#define loop(A,B,C)     for( A=B; A<=C; A++)
#define repeat(n)       loop(i,1,n)
#define RANDOM(range)   ceil((rand()/VAXMAX)*range)

#define for_all_token   loop ( token,   1,      TOKEN   )
#define for_all_tr      loop ( tr,      1,      TR      )
#define for_all_site    loop ( site,    1,      SITE    )
#define for_all_rel     loop ( rel,     1,      R       )
#define for_all_att     loop ( att,     1,      JA      )

#define for_all_related_site    for_all_site \
                                if(related_site[tr][site]>= CHILD_SITE )

#define for_all_related_i       repeat (SITE) \
                                if (related_site [tr][i] >= CHILD_SITE )

#define for_all_related_i_in_state(state)       repeat (SITE) \
                                if (related_site [tr][i] >= CHILD_SITE \
                                   && th_state [ tr] [ i] == state )

#define for_all_child_site      for_all_site \
                                if(related_site[tr][site]== CHILD_SITE )

#define for_all_child_i         repeat (SITE) \
                                if (related_site [tr][i] == CHILD_SITE )

#define for_all_token_of(tr)    for_all_token \
                                if ( tr_of [ token] == tr)

#define for_all_queried_rel_in_site for_all_rel \
                                if (  queried_rel [ tr] [rel] == 1 \
                                   && fap [ rel ] [ site ] == 1 )

#define for_all_queried_att_in_rel for_all_att \
                                if (queried_att[ tr][rel] [ att ] == 1 )

#define check_in_state(counter, state) {  counter=0, \
                                loop (I, 1,   SITE ) \
                                if ( th_state [ tr ][ I ] == state ) { \
                                counter=1,break, \
                                }
#define count(counter, range, match_value)      {  counter=0, \
                                loop (I, 1,  range ) \
                                   if ( th_state[tr][I] == match_value)\
                                      counter++,\
                                }
```

```
#define goto_state(s)           th_state[tr][site] = s
#define goto_phase(s)           tr_phase[tr] = s
#define go_sleep(tr, site, state) th_state[tr][site] = state
#define is_read_only            (writing_rel [ tr ] == 0 )
#define is_a_non_update_site(site ) (fap [writing_rel[tr]][ site ]== 0 )
#define RETURN_REJECT           request_status = REJECT;
#define return_reject           return (REJECT)
#define return_waive            return (WAIVE)
#define store_in_(site)         (fap [rel] [site] ==1)

#define schedule_B(s,t,token) { U_bus_tr [ tr_of [ token] ] += t; \
                                schedule( s, t, token);}
#define schedule_C(s,t,token) { U_cpu_tr [ tr_of [ token] ] += t; \
                                schedule(s,t,token);}
#define schedule_D(s,t,token) { U_disk_tr [ tr_of [ token] ] += t; \
                                schedule(s,t,token);}

#define the_rel_in_site_is_queried (queried_rel [ tr ] [ rel ] == 1 \
                                && fap [ rel ] [ site ] == 1)
#define this_att_is_queried     (queried_att [ tr ][ rel ][ att ] == 1\
                                && fap [ rel ] [ site ] == 1)
#define the_rel_in_site_i_is_queried (queried_rel [ tr ] [ rel ] == 1 \
                                && fap [ rel ] [ i ] == 1)

#define wake_up(token)             schedule_C( WAKE_UP, DELTA_TIME, token)
#define delayed_wake_up(token)     schedule(WAKE_UP,1.0*rand()/INF,token)
#define selective_wake_up(token) schedule(WAKE_UP,1.0*rand()/INF,token)

#define UNIFORM(x1,x2)  (x1 + (x2 - x1) * prob())
#define scatter(v)      UNIFORM((v)*(1.0-DEVIATION),(v)*(1.0+DEVIATION))
#define EXPNTL(x)       (-x * log ( prob() ))
#define PROB            ( rand()  / VAXMAX )
#define site_att_of_(tr, site, att)    site_rel_conf [tr] [site] [att]
#define bro_update(tr, site, att ) {loop(I,1,R) \
                        if (component_set [tr][site][att][I] == 1 ) \
                           bro_component [tr][att][I] = HAD_BRO, \
                     bro_site_att     [tr] [site] [att] = HAD_BRO;}
#define wake_up_brother(tr, state)       {loop(I, 1, SITE)\
                     if(th_state[tr][I] == state ) \
                        wake_up (token_of [ tr ] [ I ] );}
#define minimum_site(value, state) {int I,J;  \
                     I = minimum_site = INF, \
                     loop(J, 1, SITE ) \
                        if (th_state [ tr ] [ J ] == state \
                           && value [ tr] [ J ] < I )  \
                            minimum_site = J; \
                            }
#define maximum(value, state) {int I,J;  \
                     I = maximum_site = (- INF); \
                     loop(J, 1, SITE ) \
                        if (th_state [ tr ] [ J ] == state \
                           && value [ tr] [ J ] > I )  \
                            maximum_site = J, \
                            }
#define max(a,b)  ( a > b ? a : b )
#define min(a,b)  ( a < b ? a : b )
#define pres(s)  { prs(s),prs(TKTRS), out(token), out(tr), out(site ); }
#define prsERROR(s)  { prs(s),schedule ( STOP, 0.0, 0), }
#define lnprs(s)        {ln,prs(s); }
#define pri1(v,r1)      {int i, ln, \
                prs (v ); \
                repeat(r1) out ( v [i]);ln;}
#define pr22(v,r1,r2)   {int i,j; ln, \
                prs (v ),ln, \
                repeat(r1) { loop (j,1,r2) out ( v [i] [j]),ln,}}
#define pr2f(v,r1,r2)   {int i,j; ln, \
                prs (v ); \
                repeat(r1) { loop (j,1,r2) printf ("%4.2f ",v [i] [j]);ln,}}
#define prvec(v,r0,r1)  {int i; \
                prs (v ); \
                repeat(r1) out ( v [r0] [i]);ln,}
#define pr2(v,r0,r1)     {int i; ln, \
                prs (v ), \
                repeat(r1) out ( v [r0] [i]),ln;}
#define pr3(v,r0,r1,r2) {int i,j; ln, \
                prs (v ),ln, \
                repeat(r1) {loop (j, 1, r2 )out ( v [r0][i ][ j ] );ln,}}
#define pr4(v,r0,r1,r2,r3)       {int i,j,k, ln; \
                prs (v );ln, \
                repeat(r1) { \
                   loop(j,1,r2) {\
                   loop (k, 1, r3 )out ( v[r0] [i ][ j ][k] ),ln,}ln;}}

FILE *fp,  fclose().

/*-----------------------------------------------------------------------
 *
 *      Arrays with initial values
 *
 *-----------------------------------------------------------------------
 */

int relation_conf          [ R+1  ] [ JA+1 ] =    /* NO, YES */
{
```



```
{ 0, 0, 0, 0, 0},
{ 0, 0, 0, 1, 1},
{ 0, 0, 1, 1, 0},
{ 0, 1, 1, 1, 0},
{ 0, 1, 0, 0, 1}
};

int relation_nj_width    [ R+1  ] =   /* multiple of 40 bit*/

   { 0, 40, 200, 200, 200},

   /*
   *  relation width become 0  240 378 408 368
   */

int domain_width            [ JA+1 ] =   /* multiple of 8 bit*/

   { 0, 16, 8, 24, 40, },

int ini_rel_card_ratio    [ R+1  ] =   /* the existing tuples every 10 */

        /* The full is considered multiplication of 1/10 of each range */

   { 0, 1, 4, 3, 6, };

    /*
    *  initial cardinality become
    *  0 38707 40960 193536 30720
    */

int domain_range            [ JA+1 ] =   /* multiple of 40 */

   { 0, 80, 80, 160, 200},

double selectivity  [ R+1 ] [ JA+1 ] = /* Temporary set, to be change*/

   {
   { 0.0, 0.0, 0.0, 0.0, 0.0},
   { 0.0, 0.1, 0.3, 0.6, 0.9},
   { 0.0, 0.2, 0.4, 0.9, 0.7},
   { 0.0, 0.8, 0.6, 0.1, 0.3},
   { 0.0, 0.6, 0.2, 0.4, 0.5},
   };

int fap  [ R+1  ] [ SITE+1       ] =     /* NO, YES */

   {
   { 0, 0, 0, 0, 0},
   { 0, 1, 0, 1, 0},
   { 0, 0, 1, 1, 0},
   { 0, 1, 0, 0, 1},
   { 0, 0, 1, 0, 1},
   };

char DISK [ SITE +1 ] [ 8 ] = {
   "DISK 0 ",
   "DISK 1 ",
   "DISK 2 ",
   "DISK 3 ",
   "DISK 4 ",
   } ;

char BUS_TOKEN [ SITE +1 ] [ 12 ] = {

   "BUS_TOKEN 0 ",
   "BUS_TOKEN 1 ",
   "BUS_TOKEN 2 ",
   "BUS_TOKEN 3 ",
   "BUS_TOKEN 4 ",
   } ;

char BUS[ ] = "BUS",

/*************************************************************************
 *
 *   Macro definitions and global variables of DDBLMN-SMPL
 *
 *************************************************************************/

#define          POOLSIZE    10000/* size of element pool */
#define          NOFFACS     1000 /* no  of facilities  */
#define          NAMELENGTH    8  /* max length of fac  names */
#define          NIL          -1  /* end of list mark */
#define          NOTINIT      -2  /* list not initialized yet */
#define          INT         int  /* type of integer variables */
#define          FLOAT    double  /* type of real variables */
#define          Busy          1  /* status of facilities */
#define          FREE          0  /*     - " -              */

#define          PRI(k)       15[k]/* priority *//* queue elements */
#define          RTIME(k)     14[k]/* remaining event time */
#define          Event(k)     13[k]/* event no  */
#define          Token(k)     12[k]/* token no  */
#define          QLINK(k)     11[k]/* next elt in priority queue */
```

```
659
660 #define         BBTART(e)    15[e]/* busy start time *//* servers */
661 #define         BBUM(e)      14[e]/* busy total time */
662 #define         BBUMSQ(e)    16[e]/* busy sum of sq  */
663 #define         RELCNT(e)    13[e]/* release count */
664 #define         USRPRI(e)    12[e]/* user priority */
665 #define         USRTKN(e)    11[e]/* user token no  */
666
667 #define         TQBUM(f)     15[f+1]/* total queueing time  *//* facilities */
668 #define         PREEMPTC(f)  14[f+1]/* preempt count */
669 #define         NAME(f)      13[f+1]/* facility name */
670 #define         DCHAIN(f)    12[f+1]/* next in facility chain */
671 #define         RESQ(f)      11[f+1]/* first in queue */
672 #define         TLAST(f)     15[f]/* time of last change in qlength */
673 #define         QEXCNT(f)    14[f]/* queue exit count */
674 #define         QLEN(f)      13[f]/* queue length */
675 #define         NOFBusy(f)   12[f]/* number of busy servers */
676 #define         NOFSERV(f)   11[f]/* number of servers */
677 #define         MQLEN(f)     10[f]/* maximum queue length */
678
679 /*-------------------------------------- ------------------------------*/
680 /*      Static variables and arrays                                     */
681 /*--------------------------------------------- ------------------------*/
682
683 INT         empl_token,  empl_event,
684 INT         busy_free[NOFFACS],
685 FLOAT clock,    /* current simulation time */
686             start,                  /* start time */
687 INT         10[POOLSIZE],           /* list element pool */
688             11[POOLSIZE],
689             12[POOLSIZE],
690             13[POOLSIZE];
691 FLOAT       14[POOLSIZE],
692             15[POOLSIZE],
693             16[POOLSIZE];
694 char mdlname[60],       /* modelname */
695             *fnm[NOFFACS],          /* facility name pointers */
696             fname[NAMELENGTH * NOFFACS + 1],/* facility names */
697 INT         evl,                    /* event list head */
698             fdb,                    /* facility descriptor block */
699             avl,                    /* avail list head */
700             desc,                   /* descriptor chain */
701             na,                     /* next available fnm index */
702             Free,                   /* next available fname index */
703
704
705 char stringbuf[9],      /* buffer for int-to-string conversions */
706 char *str (),           /* forward declaration */
707
708 /*-------------------------------------- ------------------------------*/
709 /*  Macro funtions of DDSLAM-SMPL                                       */
710 /*--------------------------------------------- ------------------------*/
711
712 /*-------------------------------------- ------------------------------*/
713 /*      getelm - return free element i from the pool                    */
714 /*      prob - return random from Uniform[ 0, 1 )                       */
715 /*      putelm - return element i to pool                               */
716 /*      qlength - return queuelength at facility f                      */
717 /*      status - return facility status for device f                    */
718 /*               0 = Free, 1 = Busy                                     */
719 /*      time - return current simulation time                           */
720 /*-------------------------------------- ------------------------------*/
721
722 #define getelm(empl_i) { empl_i = avl; avl = QLINK(empl_i),/*get elem*/}
723 #define prob() ((1.0*rand())/INF)
724 #define putelm(empl_i) {QLINK (empl_i) = avl; avl = empl_i; }
725 #define qlength(f)  (QLEN (f))
726 #define status(f) (busy_free [f])
727 /*  status(f) ( (busy_free [f] == Busy  ? (Busy)  : (FREE)) */
728 #define time() (clock)
729
730 /*--  -------------------------------------- ------------------------*/
731 /*      cause - cause next event to happen  return event e and token tkn*/
732 /*-------------------------------------- ------------------------------*/
733
734 #define cause(e, tkn) {                                                 \
735         register    INT empl_i;                                         \
736         if (evl == NIL)                                                 \
737             err (5),            /* empty event list */                  \
738         empl_i = evl;                                                   \
739         evl = QLINK (empl_i);           /* unlink */                    \
740         tkn = empl_token = Token (empl_i),                              \
741         e = empl_event = Event (empl_i),                                \
742         clock = PRI (empl_i),                                           \
743         putelm (empl_i),                /* return to pool */            \
744                                                                         \
745         /* DDD */                                                       \
746         if (empl_tr > 0)                                                \
747             tracer (tkn, "CU", " ", empl_event),            \
748         if (test > 4)                                                   \
749             check (),                                                   \
760         /* EEE */                                                       \
761 }
762
```



```
783 /*-------------------------------------- ------------------------------*/
784 /*      schedule - schedule event e at token tkn to happen after time t */
785 /*-------------------------------------- ------------------------------*/
786
787 #define schedule(e, empl_t, tkn) {                                      \
788         register INT empl_i;                                            \
789         int t;                                                          \
790         t=(empl_t);                                                     \
791         /* DDD */                                                       \
792         if (test > 5)                                                   \
793             printf (">>>>< schedule  e=%d  t=%f  tkn=%d\n", e, t, tkn),  \
794                                                                         \
795         if (t < 0)                                                      \
796             err (4),            /* negative event time */           \
797         /* EEE */                                                       \
798         /*                                                             \
799         * save associated data into the element cell train, then        \
800         * link the train to the even list                               \
801         * even list is a virtual chain built on the top of element pool \
802         */                                                             \
803         getelm (empl_i);                                                \
804         Token (empl_i) = tkn,                                           \
805         Event (empl_i) = e,                                             \
806         RTIME (empl_i) = 0,                                             \
807         PRI (empl_i) = clock + t;                                       \
808         enlist (evl, empl_i),                                           \
809                                                                         \
810         /* DDD */                                                       \
811         if (empl_tr > 0)                                                \
812             tracer (tkn, "SC", " ", e),                 \
813         /* EEE */                                                       \
814 }
815
816 /*-------------------------------------- ------------------------------*/
817 /*      locate - return server reserved for token tkn at facility f.    */
818 /*               or NIL if tkn did not reserve any server               */
819 /*-------------------------------------- ------------------------------*/
820
821 #define locate(f, tkn, empl_i) { register    INT empl_i, empl_k,        \
822         empl_i = NIL;                                                   \
823         for (empl_k = f + 2,empl_i == NIL; ++empl_k)                    \
824             if (USRTKN (empl_k) == tkn)                                 \
825                 empl_i = empl_k - (f + 1),                              \
826 }
827
828 /*-------------------------------------- ------------------------------*/
829 /*      release - release token tkn from facility f                     */
830 /*-------------------------------------- ------------------------------*/
831
832 #define release(empl_f, tkn) {                                          \
833         register    INT empl_i,     /* server */                        \
834                     f,                                                  \
835                     empl_k,         /* dequeued entry -> queued event  */ \
836         f = empl_f,                                                     \
837         /* DDD */                                                       \
838         if (empl_tr > 0)                                                \
839             tracer (tkn, "release", fnm[NAME (f)], f),                  \
840         /* EEE */                                                       \
841                                                                         \
842         busy_free [ f ] = FREE ,                                        \
843         empl_i=f+2,                                                     \
844         USRTKN (empl_i) = NIL,                                          \
845         ++RELCNT (empl_i),                                              \
846         BBUM (empl_i) += clock - BSTART (empl_i),/* server statistics */ \
847         BBUMSQ (empl_i) +=                                              \
848                 (clock - BSTART (empl_i)) * (clock - BSTART (empl_i)),  \
849                                                                         \
850         if (QLEN (f) != 0) {/* dequeue and reschedule queued token */   \
851             if (QLEN (f) > MQLEN (f))                                   \
852                 MQLEN (f) = QLEN (f),                                   \
853             TQBUM (f) += QLEN (f) * (clock - TLAST (f)),                \
854             --QLEN (f),                                                 \
855             ++QEXCNT (f),                                               \
856             TLAST (f) = clock,                                          \
857             /*                                                          \
858             * take a queued token out of queue to serve                 \
859             * Normally, the first one  is picked (FIFO)                 \
860             * For bus, bus contension is simulated by randomly pick 1   \
861             * from the bus queue  THe priority can be played here       \
862             */                                                         \
863             empl_k=RESQ(f),                                             \
864             if (f == bus ) {                                            \
865                int pre_k,                                               \
866                I=RANDOM ( (QLEN (f) +1 ) ) ,                            \
867                if ( I ==  1 )                                           \
868                   RESQ (f) = QLINK (empl_k),                            \
869                else {   int P,                                          \
870                   loop (P, 1, (I-1) ) {                                 \
871                      pre_k = empl_k,                                    \
872                      empl_k = QLINK ( empl_k ),                         \
873                      }                                                  \
874                   QLINK (pre_k) = QLINK (empl_k),                       \
875                   }                                                     \
876                }                                                        \
```

```
        else { /* not a bus queue , pick the first one in queue */ \
          RESQ (f) = QLINK (empl_k).                               \
          }                                                        \
        /* empl_k = index of entry dequeued */                     \
        /* DDD */                                                  \
        if (empl_tr > 0)                                           \
              tracer (Token (empl_k), "dequeue", fnm[NAME (f)], f);\
        /* EEE */                                                  \
                                                                   \
        if (RTIME (empl_k) > 0) {/* resume, reserve faci.    */    \
              busy_free [ f ] = Busy ; /* for dequeued token */    \
              USRTKN (empl_i) = Token (empl_k);                    \
              USRPRI (empl_i) = PRI (empl_k).                      \
              BSTART (empl_i) = clock.                             \
                                                                   \
              /* DDD */                                            \
              if (empl_tr > 0)                                     \
                 tracer (Token (empl_k),"reserve",fnm[NAME (f)],f);\
              /* EEE */                                            \
        }                                                          \
                                                                   \
        PRI (empl_k)=clock+RTIME (empl_k),/* schedule next event */\
        enlist (evl, empl_k);     /* for dequeued token  */        \
                                                                   \
        /* DDD */                                                  \
        if (empl_tr > 0)                                           \
           tracer (Token (empl_k), "schedule", " ",        \       \
                   Event (empl_k));                                \
        /* EEE */                                                  \
      }                                                            \
}
/*------------------------------------------------------------------*/
/*      enlist - enter element elm to priority queue "list"
/*              queue is either event list (time, ascending order) or
/*              facility queue (normal priority, decending order)
/*------------------------------------------------------------------*/

#define enlist(list, elm) {                                        \
      register     INT pred, succ, exit,                           \
      FLOAT arg.                                                   \
                                                                   \
      /* DDD */                                                    \
      if (test > 6)                                                \
         printf (">>>> enlist:  list=%u  elm=%u\n", list, elm);  \ \
      /* EEE */                                                    \
                                                                   \
      arg = PRI (elm).                                             \
      succ = list,                                                 \
      exit = 0.                   /* initialize */                 \
                                                                   \
      while (exit == 0) {         /* find position */              \
            if (succ == NIL)      /* end of list */                \
                  exit = 1;                                        \
            else                                                   \
                  if (list == evl) {/* ascending order */          \
                        if (PRI (succ) > arg)                      \
                              exit = 1,/* correct place */         \
                  }                                                \
                  else                                             \
                        /* desc  order, normal case */             \
                        if (PRI (succ) < arg ||                    \
                            PRI (succ) == arg && RTIME (elm) > 0)  \
                           /*   -"-  , preempted transaction */    \
                              exit = 1;                            \
            if (exit != 1) {                                       \
                  pred = succ.                                     \
                  succ = QLINK (pred),/* advance */                \
            }                                                      \
      }                                                            \
                                                                   \
      QLINK (elm) = succ;         /* insert to list */             \
      if (succ == list)                                            \
            list = elm, /* first  */                               \
      else                                                         \
            QLINK (pred) = elm, /* somewhere in the middle  */     \
}

/*------------------------------------------------------------------*/
/*      enqueue - enter event ev in token tkn with priority pri  and
/*              remaining event time et to facility queue f
/*------------------------------------------------------------------*/

#define enqueue(f, tkn, pri, ev, te) {                             \
      register INT empl_i;                                         \
                                                                   \
      /* DDD */                                                    \
      if (test > 6)                                                \
         printf ("<<<< enqueue  f=%u  tkn=%u  pri=%f  ev=%u  te=%f\n", \
                        f, tkn, pri, ev, te).                      \
      if (empl_tr > 0)                                             \
            tracer (tkn, "queue", fnm[NAME (f)], f).               \
      /* EEE */                                                    \
```



```
                                                                   \
      TQSUM (f) += QLEN (f) * (clock - TLAST (f)),/* queue state */ \
      ++QLEN (f);                                                  \
      TLAST (f) = clock,                                           \
                                                                   \
      getelm (empl_i);                                             \
      Token (empl_i) = tkn,                                        \
      Event (e·p·_i) = ev;                                         \
      RTIME (e·pl_i) = te;                                         \
      PRI (empl_i) = pri,                                          \
                                                                   \
      enlist (RESQ (f), empl_i);    /* insert to pri-queue */      \
}

/*------------------------------------------------------------------*/
/*      cancel - cancel event with token number tkn
/*              return event ev and remaining time to event te
/*------------------------------------------------------------------*/

#define cancel(tkn, ev, te) {                                      \
      INT pred.                                                    \
      register INT  succ.                                          \
      for (succ = evl, succ != NIL && Token (succ) != tkn, succ =  \
           QLINK ( pred ) )                                        \
             pred = succ,                                          \
      if (succ == NIL) {                                           \
         pre ( The token have not schedule any event ).            \
         event = 0;                                                \
         }                                                         \
      else {                                                       \
                      /* old : err (6). token not in list */       \
                                                                   \
      if (succ == evl)          /* unlink */                       \
           evl = QLINK (succ);                                     \
      else                                                         \
           QLINK (pred) = QLINK (succ).                            \
                                                                   \
      ev = Event (succ).                                           \
      te = PRI (succ) - clock,                                     \
      putelm (succ),            /* return to pool */               \
                                                                   \
      /* DDD */                                                    \
      if (empl_tr > 0)                                             \
           tracer (tkn, "cancel", " ", ev),              \         \
      /* EEE */                                                    \
} }
/*------------------------------------------------------------------
*                 Static variable in DDBLMN SMPL
*------------------------------------------------------------------*/


INT empl_tr    = 0,   /* trace indicator: 0=off, 1,2=on  */
    test       = 0,   /* test option 0 .9,higher test, more output*/
    inputi[10] =   /* I/P on cmd line, can be used by user */

          { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

INT nl = POOLSIZE,      /* no  of pool elements */
    nm = NOFFACS.       /* no  of facilities */


/*------------------------------------------------------------------
*
*      bro_set(token) --- set up all broadcast component sets
*
*,                 CALLED BY : new_trans ()
*------------------------------------------------------------------
*/
bro_set(token)
int token ;
{
register int tr, rel,site,att, home_site.

home_site = site_of [ token ];
tr        =   tr_of [ token ];

for_all_rel if (queried_rel [ tr ] [ rel ] == 1 )
   bro_rel      [tr] [rel] = WAIT_BRO,
   else bro_rel [tr] [rel] = NULL,
for_all_site    if ( related_site [ tr ] [ site ] >= CHILD_SITE ) {

   for_all_att  {
      if (site_rel_conf[tr][site][att] == 1 )
```

```
1024            bro_site_att       [tr] [site] [att] = WAIT_BRO;
1025            else bro_site_att  [tr] [site] [att] = NULL;
1026        for_all_rel
1027           if (   relation_conf [rel] [att]        == 1
1028               && fap           [rel] [site]       == 1
1029               && queried_att   [tr]  [rel]  [att] == 1
1030               && site_rel_conf [tr]  [site] [att] == 1
1031              )
1032              component_set [tr] [site] [att] [rel]  = 1;
1033           else
1034              component_set [tr] [site] [att] [rel]  = 0;
1035        }
1036     }
1037  for_all_att for_all_rel  bro_component  [tr] [att] [rel]  = NULL;
1038
1039  }
1040
1041  /*-------------------------------------------------------------------
1042  *
1043  *      Main Simulation program :  sim ()
1044  *
1045  *-------------------------------------------------------------------
1046  */
1047  sim ()
1048  {
1049  register int tr,site,token, r;
1050  int      iteration, start_time, end_time;
1051  int      event,
1052  FILE     * fp3,  * fopen (),
1053
1054  start_time  = timestamp ();
1055  smpl ( "DDBLMN");
1056  initial ();
1057  for_all_site
1058     disk [site] = facility ( DISK  [site], 1 );
1059  for_all_site
1060     bus_token [site] = facility ( BUS_TOKEN  [site], 1 );
1061  bus = facility ( BUS, 1 );
1062  set_q ();
1063  schedule ( NEW_TR, 1.0, 0 );
1064
1065
1066  fprintf ( stderr, " Minimum number of transactions to be generated? \n" );
1067  scanf   ("%d", &SIMULATED_TR ),
1068
1069  fprintf ( stderr, " Allowed iteration error ? \n" );
1070  scanf   ("%f", &allowed_error ),
1071
1072  fprintf ( stderr, " Mean transaction arriving interval ?  in second \n" );
1073  scanf   ("%d", &mean_tr ),
1074
1075  fprintf ( stderr, " Fraction of updated transaction?  0 - 10 \n" ),
1076  scanf   ("%d", &K ),
1077  UPDATE_QUERY_RIT = 10 - K;
1078
1079  fprintf ( stderr, " time to release read lock?  0  LP, 1  non - LP \n" ),
1080  scanf   ("%d", &r_unlock_time ),
1081
1082  fprintf ( stderr, " PACKET SIZE in block/10 ? \n" ),
1083  scanf   ("%d", &K ),
1084  PACKET_SIZE_IN_BLOCK = 1.0 * K/10.
1085  PACKET_SIZE = PACKET_SIZE_IN_BLOCK * BLOCK_SIZE ,
1086  CONTENTION_PACKET = 1.0 * 50. /PACKET_SIZE,
1087  PACKET_TIME = PACKET_SIZE * BUS_TIME_PER_BIT,
1088
1089  fprintf ( stderr, " Maximum # of TR entering each site? \n" ),
1090  scanf   ("%d", &MAX_TR_OF_SITE ),
1091
1092  fprintf ( stderr, " Redundant or non   redundant materialization? 1/0 \n" ),
1093  scanf   ("%d", &K ),
1094  if ( K = 0 )
1095     fap [ 1 ][ 3 ] = fap [ 2 ][ 3 ] = fap [ 3 ][ 4 ] = fap [ 4 ][ 4 ] = 0.
1096
1097  fp = fopen ("ddblmn data", "w"),
1098  fclose ( fp ),
1099  DATA_SIZE   = PACKET_SIZE - PACKET_OVERHEAD.
1100
1101  /*
1102  * check if tr had been terminated
1103  */
1104
1105  while (     finished_tr     <=   SIMULATED_TR
1106         ||   iteration_error >    allowed_error
1107        ){
1108
1109     cause ( event, token ),
1110
1111     tr   = tr_of [ token ] ,
1112     site = site_of [ token ],
1113     my_trace ( token, tr, site, event ),
1114
1115     if (  tb_state [tr] [site] ==    NULL
1116        &&    event             !=    NEW_TR
1117        &&    event             !=    STOP
```



```
1118        )
1119        continue,
1120
1121     switch ( event ) {
1122
1123        case ( NEW_TR ) :
1124           new_trans (),
1125           break,
1126
1127        case ( NEED_BUS_TOKEN )
1128           if ( need_bus_token ( token ) == 0 ) return,
1129           if ( reserve ( bus_token [ site ], token, 0 ) == BUSY ) {
1130              schedule_C ( CONTEND_BUS, DELTA_TIME, token ),
1131              }
1132           break,
1133
1134        case ( CONTEND_BUS )
1135           if ( reserve ( bus, token, 0 ) == BUSY ) {
1136              schedule_B ( RELEASE_BUS_TOKEN, DELTA_TIME, token ),
1137              get_bus ( token ),
1138              }
1139           break,
1140
1141        case ( RELEASE_BUS_TOKEN )
1142           release ( bus_token [ site ], token ),
1143           break,
1144
1145        case ( RELEASE_BUS ) :
1146           release ( bus, token ),
1147           release_bus ( token ),
1148           break,
1149
1150        case ( NEED_DISK ) :
1151
1152           /*   check if the lock still valid or not */
1153           if (    tb_state [ tr ] [ site ]           == LP_PRE_LOCK
1154              &&  valid_check ( tr, site, LP_WAIT_LOCK ) == 0
1155              ) return ( 0 ),
1156
1157           if ( reserve ( disk[site], token, 0 ) == BUSY ) {
1158              get_disk ( token ),
1159              }
1160           break,
1161
1162        case ( RELEASE_DISK )
1163           release ( disk[site], token ),
1164           release_disk ( token ),
1165           break,
1166
1167        case ( LP_FINISH )
1168           finish ( token ),
1169           break,
1170
1171        case ( GSJ_FINISH )
1172           finish ( token ),
1173           break,
1174
1175        case ( PP_FINISH ) :
1176           finish ( token ),
1177           break,
1178
1179        case ( WAKE_UP )
1180           waken_up ( token ),
1181           break,
1182
1183        case ( STOP )
1184           goto stop,
1185        }
1186     }
1187
1188  /**********/
1189     stop
1190  /**********/
1191
1192  end_time = timestamp (),
1193  pri ( end_time - start_time ), ln,
1194
1195  report (),
1196  }
1197  /*-------------------------------------------------------------------
1198  *
1199  *      need_bus_token() ---
1200  *
1201  *-------------------------------------------------------------------
1202  */
1203  need_bus_token(token)
1204  register int token,
1205  {
1206  register int i,tr,rel,site,att, j,k,
1207  int result,state,
1208  site = site_of [ token ],
1209  tr   = tr_of   [ token ],
1210
1211  result = 1,
```

```
1212  state =  th_state [ tr ] [ site ] ;
1213
1214  switch ( state) {
1216
1216          /*-------------------------------------------------
1217          *                  LP phase
1218          *-------------------------------------------------*/
1219
1220          case LP_SYN_EST_BRO :
1221          case LP_SJ :
1222              if (valid_check (tr, site, LP_WAIT_LOCK) == 0) result =0;
1223                break;
1224
1226          /*-------------------------------------------------
1226          *                  GSJ phase
1227          *-------------------------------------------------*/
1228
1229          case GSJ_INI :
1230              if ( not_lock_in_LP [ token ] == 1
1231                && valid_check (tr, site, GSJ_WAIT_LOCK)== 0) result =0;
1232                break;
1233
1234          /*-------------------------------------------------
1235          *                  PP phase
1236          *-------------------------------------------------*/
1237
1238          case PP_PRE_LOCK :
1239              if (valid_check (tr, site, PP_WAIT_LOCK) == 0) result =0;
1240                break;
1241    }
1242  return (result);
1243  }
1244  /*-------------------------------------------------------------------
1245  *
1246  *     get_bus() --- get the bus
1247  *
1248  *-------------------------------------------------------------------
1249  */
1260  get_bus(token)
1261  register int token;
1262  {
1263  register int tr,site,att;
1264
1265  site = site_of [ token ];
1266  tr   = tr_of   [ token ];
1267
1268  switch ( th_state [tr][site] ) {
1269
1280          /*-------------------------------------------------
1281          *                  LP phase
1282          *-------------------------------------------------*/
1283
1284          case PRE :
1286
1286               if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
1287                  schedule_B ( RELEASE_BUS, cc_time, token ),
1288                  break,
1289                  /* !! state had been changed by  solve_pg_conflict */
1270                  }
1271               schedule_B ( RELEASE_BUS, 3 * PACKET_TIME, token ),
1272               break,
1273
1274          case LP_SJ :
1276
1276               /*
1277               * only one child site, home site is empty
1278               * transmit joined relations to home site
1279               * calculate relation size
1280               */
1281               packet = size_of_rt_bro ( tr, site ),
1282               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1283               break,
1284
1286          case LP_SYN_EST_BRO :
1288
1287               if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
1288                  schedule_B ( RELEASE_BUS, cc_time, token ),
1289                  break,
1290                  /* !! state had been changed by  solve_pg_conflict */
1291                  }
1292               pre ( 1ST_HV_BROAD ),
1293               schedule_B ( RELEASE_BUS, PACKET_TIME, token ),
1294               break,
1296
1296          case LP_WAIT_END :
1297
1298               if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
1299                  schedule_B ( RELEASE_BUS, cc_time, token ),
1300                  break,
1301                  /* !! state had been changed by  solve_pg_conflict */
1302                  }
1303               pre ( An Alternate Est Slowest site 1ST HV BROAD ),
1304               schedule_B ( RELEASE_BUS, PACKET_TIME, token ),
1305               break,
```



```
1306
1307          case LP_ALL_FINISH :
1308               pre ( Synchronization of LP ),
1309
1310               if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
1311                  schedule_B ( RELEASE_BUS, cc_time, token ),
1312                  break,
1313                  /* !! state had been changed by  solve_pg_conflict */
1314                  }
1315               goto_state ( LP_END ),
1316               packet = ( 2 * CONTENTION_PACKET ) * num_of_broad [ tr ] ,
1317               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1318               break,
1319
1320          /*-------------------------------------------------
1321          *                  GSJ phase
1322          *-------------------------------------------------*/
1323
1324          case GSJ_INI :
1325
1326               if (  pguser[ site]                    == tr
1327                  && cc_lock_get_bus ( tr, site ) == FAIL
1328                  ) {
1329                  schedule_B ( RELEASE_BUS, cc_time, token ),
1330                  break,
1331                  /* !! state had been changed by  solve_pg_conflict */
1332                  }
1333               packet = ( 2 * CONTENTION_PACKET ) * num_of_broad [ tr ] ,
1334               schedule_B ( RELEASE_BUS, packet * PACKET_TIME,  token ),
1335               break,
1336
1337          case GSJ_SJ_BRO :
1338
1339               att = broadcasting_att [ tr ],
1340               K = min (  site_att_card [ tr  ] [ site ] [att] ,
1341                          (0.5 * domain_range [ att ] )    ),
1342               packet = K * domain_width  [ att ] / DATA_SIZE,
1343               packet = ( packet < 1 ) ? 1   packet,
1344               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1345               /* at least one message is needed */
1346               break,
1347
1348          case GSJ_ALL_FINISH :
1349
1350               packet = ( 2 * CONTENTION_PACKET ) * num_of_broad [ tr ] ,
1351               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1352               break,
1353
1354          case GSJ_WAIT_END :
1356
1356               packet = ( 2 * CONTENTION_PACKET ) * num_of_broad [ tr ] ,
1357               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1358               break,
1359
1360          /*-------------------------------------------------
1361          *                  RT phase
1362          *-------------------------------------------------*/
1363
1364          case RT_START :
1366
1368               packet = size_of_rt_bro ( tr, site ),
1367               packet = ( packet < 1 ) ? 1   packet,
1368               schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
1369               break,
1370
1371          /*-------------------------------------------------
1372          *                  PP phase
1373          *-------------------------------------------------*/
1374
1376          case PP_START :
1378
1377               /* a local query broadcast blocking message */
1378               schedule_B ( RELEASE_BUS, 2 * PACKET_TIME, token ),
1379               break,
1380
1381          case PP_PRE_LOCK :
1382
1383               if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
1384                  schedule_B ( RELEASE_BUS, cc_time, token ),
1385                  break,
1386                  /* !! state had been changed by  solve_pg_conflict */
1387                  }
1388               if ( class_of [ tr ] == LOCAL_QUERY )  {
1389                  /* broadcasting blocking message for the local query */
1390                  schedule_B ( RELEASE_BUS, PACKET_TIME, token ),
1391                  break,
1392                  }
1393               /*
1394               * Count the size of updated attributes
1396               * The cardinality is assumed the cross product
1396               * of all updated_att
1397               *
1398               * There is a problem on the copy selection
1399               * i e  from which site to get the updated relation to
```

```
            * count the cardinality? The different copies may reach a
            * different result  We had ignore the join-restriction in
            * RT and PP phase  That also make diff copies different
            * Here we always use the first site that contains the
            * updated relation
            */

            { int  bit_per_tuple, card, dsize, rel;

            dsize = 0,
            rel = writing_rel [ tr ] ,
            for_all_site
               if ( fap [ rel ] [ site ]  == 1 ) break,

            card = 1;
            for_all_att
               if ( updated_jatt [ tr ] [ rel ] [ att ] == 1)
                  card *= site_att_card [ tr ] [ site ] [ att ];

            bit_per_tuple = 0;
            for_all_att
               if ( updated_jatt [ tr ] [ rel ] [ att  ] == 1 )
                  bit_per_tuple += domain_width[ att ],
            if ( updated_njatt [ tr ] [ rel ] == 1 )
               bit_per_tuple += relation_nj_width [ rel ];

            packet = ( card * bit_per_tuple ) / DATA_SIZE;
            }
            packet=max(1,packet),    packet=min(100,packet);
            schedule_B ( RELEASE_BUS, packet * PACKET_TIME, token ),
            break,
        }
    }
/*-------------------------------------------------------------------
*
*       release_bus() --- after using bus
*
*--------------------------------------------------------------------
*/
release_bus(token)
register int token,
{
register int i,tr,j,rel,site,att,

site = site_of [ token ];
tr   = tr_of   [ token ];

switch  (th_state [tr][site] ) {

        /*-----------------------------------------------------------
        *                   LP phase
        *-----------------------------------------------------------*/

             /*
             * The following 3 state is for trans whose
             * lock had been withdrawn
             * due to a pg conflict
             */

        case LP_WAIT_LOCK

               for_all_related_i blocked_site [ tr ] --,
               wake_up_brother ( tr, LP_WAIT_LOCK );
               break,

        /*-----------------------------------------------------------
        *                   GSJ phase
        *-----------------------------------------------------------*/

        case GSJ_WAIT_LOCK

               blocked_site [ tr ] --,
               wake_up ( token ),
               break,

        case PP_WAIT_LOCK

               blocked_site [ tr ] --,
               wake_up ( token );
               break,

        case PRE


             goto_phase ( LP );
             before = timestamp (),
             orphan_test ( token );

             /* check if the site contains a site relation or not */
             i = 0,
             for_all_att
                if ( site_rel_conf [tr][site][att] == 1 ) i ++,
             if ( i == 0 ){
                go_sleep ( tr, site, PP_WAIT_START ),
```



```
                child_gen ( tr ),
                dbsize ( tr, site );
                return;
                }

             goto_state ( LP_INI ),
             num_in_lp [ tr] ++,
             child_gen ( tr ),
             dbsize ( tr, site );
             pre ( Home ?lock ),

        case LP_INI       /* entry point of a new child */

             if ( need_lock_in_LP ( token )== 0 ) {
                schedule_C ( NEED_DISK, 1000 * INSTRUCTION_TIME, token),
                return,
                }

             goto_state ( LP_PRE_LOCK ),
             wake_up ( token ),
             break,

        case LP_SJ :
             pre ( SKIP GSJ PHASE  GOTO RT PHASE );

             /*
             *  the only one child site, home site is empty
             *  transmit relations to home site
             */

             cc_lock (  tr, site );
             goto_state ( RT_START ),
             last_rt = 0,
             goto orphan_cutin,
             break,

        case LP_SYN_EST_BRO
             pre ( ESTIMATE SLOWEST SITE Broadcast HV ),

             cc_lock ( tr, site ),

             if ( r_unlock_time == ANYTIME ) {
                cc_unlock ( tr, site ),
                resume_by (  tr, site );
                }

             /*  need check if all have finished or not */
             go_sleep ( tr, site, LP_WAIT_END ),
             num_in_lp [tr] --,

             if ( num_in_lp [tr]  != 0  ) return,

             num_of_broad [ tr ] = 1,
             goto_state ( LP_ALL_FINISH ),
             blocked_site [ tr ] = 0,

        case LP_ALL_FINISH

             pres ( LP_ALL_FINISH ),
             schedule_C ( NEED_BUS_TOKEN, DELTA_TIME ,token ),
             break,

        case LP_WAIT_END
             pre ( Alternate ESTIMATE SLOWEST SITE Broadcast HV ),

             cc_lock ( tr, site );
             if ( r_unlock_time == ANYTIME ) {
                cc_unlock ( tr, site ),
                resume_by (  tr, site ),
                }
             break,

        case LP_END     /* start GSJ phase */
             pres (**   LP Phase END !!);
             cc_lock ( tr, site );
             if ( r_unlock_time == ANYTIME ) {
                cc_unlock ( tr, site );
                resume_by (  tr, site ),
                }
             goto_phase ( GSJ );
             goto_state ( LP_WAIT_END ),

             I = pick_min_site ( tr, site, LP_WAIT_END ),
                                          /* I is est min site */

             wake_up ( token_of [ tr] [ I ] ) ,

             break,

        case GSJ_INI
             pre ( ** ONE GSJ PHASE   A MIN SITE S IDENTIFIED ),

             /*
```

```
1688            * Had Identified the min_site,
1689            * lock those relations not locked in LP phase
1690            * start to broadcast joining attribute
1691            */
1692
1693            if ( not_lock_in_LP [ token ] == 1 )
1694               cc_lock (  tr, site ),
1695
1696            schedule_C ( NEED_DISK, DELTA_TIME, token );
1697            goto_state ( GSJ_SJ_BRO ),
1698            break,
1699
1700         case GSJ_SJ_BRO :
1701            pre ( AN ATTRIBUTE IS BROADCAST ),
1702
1703            /*
1704            * update broadcast table, bro_component, bro_site_att
1705            * wake up blocked processes
1706            * if ( no other att to be broadcast )
1707            *    goto_state ( GSJ_WAIT_END ),
1708            * otherwise go_sleep ( tr, site, GSJ_WAIT_SYN_END );
1709            */
1710            dbwire ( tr, site ),
1711            bro_update ( tr , site, local_min_att [tr] [ site]  );
1712            materialization ( tr, site, local_min_att [ tr ] [ site] ),
1713
1714            if ( (     r_unlock_time            ==    ANYTIME
1715                 &&    not_lock_in_LP [ token ] ==    1
1716                 )
1717               ||      r_unlock_time            ==    GSJ_END
1718               ) {
1719
1720               cc_unlock ( tr, site ),
1721               resume_by ( tr, site ),
1722               not_lock_in_LP [ token ] = 0;
1723               }
1724
1725            /*
1726            *  check to see if all site attributes had been broadcast
1727            */
1728
1729            j = 0;
1730            for_all_att
1731               if ( bro_site_att [tr] [site] [ att ] == WAIT_BRO ) {
1732                  j ++,
1733                  break,
1734                  }
1735
1736            if ( j == 0 ) { /* no att to be bro in the site */
1737               pre ( LAST_SJ? ),
1738               go_sleep ( tr, site, GSJ_WAIT_END ),
1739               count ( j, SITE, GSJ_WAIT_SJ ),
1740
1741               /* check if this is the last sj, set the flag */
1742
1743               if ( j == 0 && blocked_site [ tr ] != 0 )
1744                  last_sj [ tr ]= 1,
1745               }
1746            else go_sleep ( tr, site, GSJ_WAIT_SYN_END ),
1747
1748            local_min_att [tr] [ site] = 0,
1749
1750            /*
1751            *  wake up those sites had broadcast all their atts
1752            *  They need perform semi   join too
1753            *  Their states are not changed
1754            */
1755
1756            num_in_gsj [ tr ] = 0,
1757            for_all_related_i_in_state ( GSJ_WAIT_END ) {
1758               if ( i != site ) {
1759                  double F,
1760                  pre ( ''' WAKEUP sites in GSJ_WAIT_END ),
1761                  num_in_gsj [ tr] ++,
1762                  F = 1000 * INSTRUCTION_TIME,
1763                  schedule_C ( NEED_DISK,F,token_of[tr][i]),
1764                  }
1765               }
1766
1767            /* wake up those site haven't bro all site atts */
1768
1769            wake_up_brother ( tr, GSJ_WAIT_SJ ), /* may be empty */
1770            break,
1771
1772         case  GSJ_ALL_FINISH :
1773            pre ( ALL SITE FINISH SEMI_JOIN ''),
1774
1775            /*
1776            *  This will not be the last sj, since at least one left
1777            *  wake up all processes in GSJ_WAIT_SYN_END state
1778            *  initiate another GSJ phase
1779            */
1780
1781            go_sleep ( tr, site, GSJ_WAIT_SYN_END ),
```



```
1782            picked_site [ tr ] = 0,
1783            min_bv      [ tr ] = INF,
1784            min_site    [ tr ] = 0,
1785
1786            K = pick_min_site ( tr, site, GSJ_WAIT_SYN_END ),
1787            if ( K == 0 ) {
1788
1789               pre ( no site in GSJ_WAIT_SYN_END ),
1790
1791               if ( blocked_site [ tr] == 0 ) {
1792                  pre ( ERROR ),
1793                  schedule ( STOP, 0.0, token ),
1794                  }
1795               else return,
1796               }
1797
1798            if ( site == 0 )
1799               return_token_id ( token ), /* use a dummy token  */
1800            wake_up ( token_of [ tr] [ K ] ),
1801            break,
1802
1803         case GSJ_WAIT_END :
1804
1805            pre ( END OF GSJ PHASE ),
1806            /* This is the end of GSJ phase, the past bro is for syn */
1807
1808            goto_phase ( RT ),
1809            wake_up_brother ( tr , GSJ_WAIT_END ),
1810            break,
1811
1812
1813         /*-------------------------------------------------
1814         *                  RT phase
1815         *------- ---------- - ------ ----- --- -- - ------ */
1816
1817         case RT_START
1818
1819            pre ( ONE SITE BROADCAST ITS RELATIONS ),
1820
1821            count ( last_rt, SITE, RT_WAIT_START ),
1822   orphan_cutin
1823            for_all_queried_rel_in_site
1824
1825               if ( bro_rel [tr] [ rel] != HAD_BRO ) {
1826                  rel_card [ tr ] [ rel ] [ home_of [ tr ] ] =
1827                     rel_card [ tr ] [ rel ] [ site ] ,
1828                  bro_rel [tr] [ rel] = HAD_BRO,
1829                  }
1830
1831            cc_unlock ( tr, site ),
1832            /*
1833            *  assume no change in       dbwire ( tr, site ),
1834            *  include broadcast relation to home site , update db
1835            */
1836            resume_by ( tr, site ),    /* wake up blocked processes */
1837
1838            if (        orphan [ token ] == 1    ) {
1839
1840               terminate_site ( token, tr, site ),
1841               terminate_tr   ( tr ),
1842               return,
1843               }
1844
1845            if (        is_read_only
1846                ||      is_a_non_update_site ( site )
1847               )
1848
1849               terminate_site ( token, tr, site ),
1850
1851            else  go_sleep ( tr, site, PP_WAIT_UPDATE ),
1852
1853            /*
1854            *    RT materialize,
1855            */
1856
1857            for_all_related_i_in_state ( RT_WAIT_START ) {
1858
1859               I = 0,
1860
1861               for_all_rel
1862
1863                  if (        queried_rel [ tr] [rel] == 1
1864                        &&    fap [ rel ] [ i ]       == 1
1865                        &&    bro_rel [ tr ] [ rel ]  == WAIT_BRO
1866                     ) {
1867
1868                     I = 1 ,
1869                     break,
1870                     }
1871
1872               if ( I == 0 ) {
1873                  last_rt ++,
1874
1875                  if (        is_read_only
1876
```

```
                ||       is_a_non_update_site ( i )
                )

                terminate_site ( token_of [ tr ] [ i ] , tr, i );
            else  go_sleep ( tr, i, PP_WAIT_UPDATE );
            }
        }

        if ( last_rt == 0 ) {
            goto_phase ( PP );
            wake_up ( token_of [ tr ] [ home_of [ tr ] ] );
            /*
            *  wakeup home site for PP
            */
            }
        else {
            for_all_related_i_in_state ( RT_WAIT_START )
                break;
            wake_up ( token_of [ tr ] [ i ] );
            }
        break;

    /* ------------------------------------------------------
    *               PP phase
    *------------------------------------------------------*/

    case PP_START :

        pre ( LOCAL QUERY BROAD AN BLOCKING MESSAGE read only );

        cc_unlock       ( tr, site );
        resume_by       ( tr, site );
        terminate_site  ( token, tr, site );
        terminate_tr    ( tr );
        break;

    case PP_PRE_LOCK

        pre ( UPDATED DATA IS BROADCAST );

        cc_lock         (  tr, site );
        cc_unlock       ( tr, site ).
                           /* omit w-lock, unlocked immediately */
        resume_by       (  tr, site );
        terminate_site ( token, tr, site );

        if ( class_of [ tr ] == LOCAL_QUERY )  {
           terminate_tr ( tr );
           break;
           }

        /*
        * unlocked since each site had gotton updated value
        * nobody can read the data before the new value is written
        * to disk
        */

        count ( j, SITE, PP_WAIT_UPDATE );
        if ( j == 0 ) terminate_tr ( tr );
        else wake_up_brother ( tr, PP_WAIT_UPDATE );
        break;
      }
}
/*      ------------------------------------------------------
*
*     get_disk() --- get the use of disk
*
*       ------------------------------------------------------
*/

#define DTIME_PER_BIT (DTIME_PER_BLOCK/(BITS_OF_BYTE * BYTES_OF_BLOCK))

get_disk(token)
register int token;
{
register int rel,site,att, tr;
int i,j,k;
int dsize;

site = site_of [ token ];
tr   = tr_of   [ token ];

switch ( th_state [tr][site] ) {

        /*--------------------------------------------------
        *             LP phase
        *--------------------------------------------------*/

        case LP_INI
        case LP_PRE_LOCK

            { int  bit_per_tuple,
            goto_state ( LP_DISK ),
```



```
        /*
        * count the disk_access_time, which is proportional
        * to relation card  ( all tuple should be scaned )
        * All involved relations is counted
        */

        if ( valid_check ( tr, site,  LP_WAIT_LOCK ) == 0 )
            return,

        dsize = 0;
        for_all_queried_rel_in_site {
            dsize +=
                ( rel_card [tr] [rel] [site] * relation_width [rel]).
            lptime[tr][site] = dsize * CPU_TIME_PER_BIT.
            }
        }
        break,

    /*--------------------------------------------------
    *             GSJ phase
    *--------------------------------------------------*/

    case GSJ_BJ_BRO :

        /*
        * count the disk_access_time, which is proportional
        * to att card   it is assumed  projected attributes are
        * stored as a independent unit  Only bro  att  is scanned
        * in GSJ  phase, no need to scan entire relation
        * Any relation containing broad attribute is picked
        */

        for_all_queried_rel_in_site

            if ( relation_conf [ rel ][broadcasting_att [ tr ]]==1 )
                break, /* need only one rel with bro attribute */

        dsize =
            site_att_card [ tr ][ site ][ broadcasting_att [tr ]]
            * domain_width [ broadcasting_att [ tr ] ].
        sjtime [tr] [site] = dsize * CPU_TIME_PER_BIT,
        break,

    case GSJ_WAIT_SJ :
    case GSJ_WAIT_END :

        { int  bit_per_tuple;
        /*
        * count the disk_access_time, which is proportional
        * to either the card of relations or projected att card
        * Depends on the operating mode
        * for ANYTIME or GSJ_END only projected attributes in
        * all involved relations is counted
        * for RT_END, the entire involved relations should be
        * scanned since no projected attribute exist
        * The cardinality of original relations are used
        *
        * since only values of the joining attribute is needed
        * by the processor, count such attributes only
        */

        dsize = 0;
        att = broadcasting att [ tr ] .
        for_all_queried_rel_in_site {
           switch ( r_unlock_time ) {

              case ANYTIME
              case GSJ_END
                           dsize += (site_att_card [tr][site][att]
                                  * domain_width [ att ]),
                           break,

              case RT_END
                           dsize +=( ini_rel_card[rel] *
                                domain_width [ att ]).
                           break,
              }
           }

        sjtime[tr][site]= dsize * CPU_TIME_PER_BIT,
        }
        break,

    /*--------------------------------------------------
    *             RT phase
    *--------------------------------------------------*/

    case RT_START

        { int  bit_per_tuple;

        /*
        * All unbroadcast relations are broadcast
        * The semi - joins on the original relations are assume
```

```
1964              * performed at the time a relation is broadcast
1965              * Relation width need not to be the original
1966              * since temporary copies of queried relation is used
1967              * But the cardinality is original since no semi - join is
1968              * is performed before
1969              */
1970
1971             dsize = 0;
1972             for_all_queried_rel_in_site
1973
1974                if ( bro_rel [ tr ] [ rel ] == WAIT_BRO )
1975                   dsize += ( ini_rel_card [rel]
1976                              * tr_rel_width [ tr ] [ rel ] ) ;
1977             }
1978             break;
1979
1980          /*-------------------------------------------------
1981          *                PP phase
1982          *-------------------------------------------------*/
1983
1984          case PP_START :
1985
1986             /* pp_time had been calculated in get_bus */
1987
1988             { int  bit_per_tuple;
1989
1990             /*
1991             * All relations is read for post-processing
1992             * The result needs not be written back
1993             * The retrieved and updated target can be get here
1994             * there is no need to read disk again
1995             * Needed tuples can be filtered out at the same time
1996             * Assume their size will not excess the memory size
1997             */
1998
1999             dsize = 0;
2000
2001             if ( class_of [ tr ] != REMOTE )
2002                dbsize ( tr, site );
2003
2004             for_all_rel /* all rels are bro to home S */
2005
2006                if ( queried_rel [ tr ] [ rel ] == 1 )
2007                   dsize += ( rel_card [tr][rel][site]
2008                              * relation_width  [ rel ] ) ;
2009             }
2010             break;
2011
2012          case PP_UPDATE :
2013
2014             { int  bit_per_tuple;
2015             /*
2016             * All updated relations is counted
2017             * to relation card  ( all tuple should be scaned )
2018             */
2019
2020             dsize = 0;
2021             for_all_rel
2022                if (   fap [ rel ] [ site ] == 1
2023                    && writing_rel [  tr  ] == rel
2024                   ){
2025                dsize += ( rel_card         [tr][rel][site]
2026                           * relation_width  [rel]          );
2027                pptime [ tr ] [ site ] =  dsize * CPU_TIME_PER_BIT;
2028
2029                /*
2030                * because entire tuple should be scanned in disk, so
2031                * used original width, instead of useful attribute only
2032                */
2033
2034                }
2035             }
2036             break;
2037       }
2038    disk_time = UNIFORM(0,DTIME_SEARCH) +  ( DTIME_PER_BIT * dsize);
2039    schedule_D(RELEASE_DISK, disk_time, token);
2040    }
2041    /*-------------------------------------------------------------------
2042    *
2043    *     release_disk() --- after use of disk
2044    *
2045    *------------------------------------------------------------------
2046    */
```



```
2047    release_disk(token)
2048    register int token;
2049    {
2050    register int i,tr,rel,site,att, j,k;
2051
2052    site = site_of [ token ];
2053    tr   = tr_of   [ token ];
2054
2055    switch ( th_state [tr][site] ) {
2056
2057          /*-------------------------------------------------
2058          *                LP phase
2059          *-------------------------------------------------*/
2060
2061          case LP_DISK :
2062
2063               if ( valid_check ( tr, site, LP_WAIT_LOCK ) == 0 )
2064                  return;
2065
2066               if ( class_of [tr] != REMOTE )
2067                  goto local_query;
2068
2069               if ( not_lock_in_LP [token] == 1 ) {
2070                  go_sleep ( tr, site, LP_WAIT_END );
2071                  return;
2072                  }
2073
2074               if ( orphan[token] == 1 ) {
2075                  schedule_C
2076                     ( NEED_BUS_TOKEN, 1.0 * lptime[tr][site], token );
2077                  goto_state ( LP_SJ );
2078                  return;
2079                  }
2080
2081               if ( set_slowest_site_of [tr] == 0 )
2082                  set_slowest_site ( tr );
2083
2084               if ( site != set_slowest_site_of [tr] ) {
2085                  go_sleep ( tr, site, LP_WAIT_SYN );
2086                  schedule_C
2087                     ( LP_FINISH  ,1.0 * lptime[tr][site], token );
2088                  return;
2089                  }
2090
2091               goto_state (  LP_SYN_EST_BRO );
2092
2093    local_query
2094               schedule_C ( LP_FINISH, 1.0 * lptime[tr][site], token );
2095               break;
2096
2097          /*-------------------------------------------------
2098          *                GSJ phase
2099          *-------------------------------------------------*/
2100
2101          case GSJ_SJ_BRO :
2102
2103               schedule_C ( NEED_BUS_TOKEN        , DELTA_TIME, token );
2104               break;
2105
2106          case GSJ_WAIT_SJ :
2107
2108               schedule_C ( GSJ_FINISH, 1.0 * sjtime[tr][site], token );
2109               break;
2110
2111          case GSJ_WAIT_END :
2112
2113               schedule_C ( GSJ_FINISH, 1.0 * sjtime[tr][site], token );
2114               break;
2115
2116          /*-------------------------------------------------
2117          *                RT phase
2118          *-------------------------------------------------*/
2119
2120          case RT_START :
2121
2122               schedule_C ( NEED_BUS_TOKEN, DELTA_TIME, token );
2123               break;
2124
2125          /*-------------------------------------------------
2126          *                PP phase
2127          *-------------------------------------------------*/
2128
2129          case PP_START :
2130
2131               schedule_C
2132                  ( PP_FINISH, 1.0 * pptime [ tr ] [site ], token );
2133               break;
2134
2135          case PP_UPDATE :
2136
2137               terminate_site ( token, tr, site );
2138               count ( ), SITE, PP_UPDATE );
2139
2140               if ( j == 0 ) { /* This is the last site */
```

```
                terminate_tr ( tr );
                }
            break;
        }
}
/*-----------------------------------------------------------------------
 *
 *     finish() --- after use of disk
 *
 *-----------------------------------------------------------------------
 */
finish(token)
int token;
{
int i,j,k,rel,site,att, tr;

site = site_of [ token ];
tr   = tr_of   [ token ];

switch ( tb_state [tr][site] ) {

        /*-----------------------------------------------
         *                LP phase
         *-----------------------------------------------*/
        case LP_DISK   /* This is a local query */

             if ( valid_check ( tr, site, LP_WAIT_LOCK ) == 0 ) {
                return;          /* stay in the state LP_WAIT_LOCK */
                }
             goto_state ( PP_WAIT_START );
             wake_up ( token );

             break;

        case LP_SYN_EST_BRO :

             if ( valid_check ( tr, site, LP_WAIT_LOCK ) == 0 ) {
                return;          /* stay in the state LP_WAIT_LOCK */
                }
             schedule ( NEED_BUS_TOKEN, 3 * DELTA_TIME, token );
             break;

        case LP_WAIT_SYN

             if ( valid_check ( tr, site, LP_WAIT_LOCK ) == 0 ) {
                return;
                }
             num_in_lp [ tr] --;
             go_sleep ( tr, site,LP_WAIT_END );

             /*  need check if all have finished */

             if ( num_in_lp [tr]  != 0  ) {
                resolve_local_dl ( token );
                return;
                }

             if ( syn_steps ( tr, LP_WAIT_END ) == 0 ) {
                pre ( NO SITE IN state LP_WAIT_END );
                return;
                }

             goto_state ( LP_ALL_FINISH );
             blocked_site [ tr ] = 0;
             schedule ( NEED_BUS_TOKEN, 200 * INSTRUCTION_TIME, token );
             break;

        /*-----------------------------------------------
         *                GSJ phase
         *-----------------------------------------------*/

        case GSJ_WAIT_SJ

             num_in_gsj [ tr] --;
             go_sleep ( tr, site, GSJ_WAIT_SYN_END );

             /*
              *  check if all the site had finished sj
              *  this include those sites in GSJ_WAIT_END
              *  no need to wait those blocked sites  here
              */

             if ( num_in_gsj [tr]  != 0  ) return;

             /* get the number of broadcast for synchronization */

next_gsj     if ( syn_steps ( tr, GSJ_WAIT_SYN_END ) == 0 ) {

                pre ( NO SITE IN state GSJ_WAIT_SYN_END );
                if ( blocked_site [ tr] == 0 ) {
                   pre ( ERROR );
                   schedule ( STOP, 0 0, token );
                   }
                else return;
```



```
                }

             goto_state ( GSJ_ALL_FINISH );
             blocked_site [ tr ] = 0;
             schedule ( NEED_BUS_TOKEN, 100 * INSTRUCTION_TIME, token );
             break;

        case GSJ_WAIT_END

             num_in_gsj [ tr ] --;

             if ( num_in_gsj [ tr ] == 0 )
                if ( last_sj [ tr ] == 1 )  {
                                        /** count number of broadcast */
                syn_steps ( tr , GSJ_WAIT_END );
                schedule
                   ( NEED_BUS_TOKEN, 100 * INSTRUCTION_TIME, token );
                }

             else {
                /*
                 *  creat a dummy token to trigger the next GSJ
                 *  Since the state of the site in GSJ_WAIT_END is
                 *  not to be changed, otherwise it will run into trouble
                 */

                pre ( Get a Pseudo token );
                token = get_token_id ();
                site  = site_of [ token ] = 0;
                tr_of [ token ] = tr;
                goto next_gsj;
                }
             break;

        /*-----------------------------------------------
         *                PP phase
         *-----------------------------------------------*/

        case PP_START

             if ( class_of [ tr ] != REMOTE ) {
                int block_someone;

                /*
                 * check to see if there are some tr blocked by
                 * this tr. If yes, broadcast a blocking message and
                 * a unblocking message. Since one blocking message is
                 * sufficient to tell every tr, there is no need to
                 * broadcast more than one blocking message
                 * Since there is difficulty of determine the time of
                 * first blocking message ( electing procedure are
                 * virtually simulated ), it is broadcast on the time
                 * of unlock.
                 */

                /* Determine if there are transactions blocked by tr */

                block_someone  = 0;

                repeat ( TR )
                   if (   (            i                         != tr )
                       && (   (       pg [ tr ][ i ]             ==  1 )
                           || (       new_relative [ i ] [ tr ]  ==  1 )
                          )
                      ) {
                      block_someone   = 1;
                      break;
                      }

                if ( block_someone   == 1 ) {

                   if ( is_read_only ) {
                      double F;
                      F = 10 * TR * INSTRUCTION_TIME;
                      schedule ( NEED_BUS_TOKEN, F, token );
                      return;
                      }
                   else goto pp_pre_lock;
                   }

                else { /* did not block some body else */

                   if ( is_read_only ) {
                      cc_unlock ( tr, site );
                      resume_by ( tr, site ); /* redundant resume */
                      terminate_site ( token, tr, site );
                      terminate_tr ( tr );
                      return;
                      }
                   else goto pp_pre_lock;
                   }

                } /* is a local query */
```

```
            if ( is_read_only ) {
                terminate_site ( token, tr, site );
                terminate_tr ( tr ),
                return,
                }

pp_pre_lock
            goto_state ( PP_PRE_LOCK ),

    case PP_PRE_LOCK  /* entry point of waken process */

        switch (   get_local_lock ( token, tr , site )    ) {

          ' case REJECT
                go_sleep ( tr, site, PP_WAIT_LOCK );
                pre ( BLOCKED !! ),
                blocked_site[ tr] ++;
                break,

            case GOT_LOCK
                if ( class_of [ tr ] != REMOTE ) {
                                /* Determine if tr blocked others */
                    int block_someone,
                    block_someone  = 0;
                    repeat ( TR )
                      if (  (           i                  !=   tr )
                          && (   ( pg [ tr ][ i ]          ==    1 )
                              || ( new_relative [ i ] [ tr ] ==   -1 )
                              )
                         ) {
                         block_someone   = 1;
                         break,
                         }

                    if ( block_someone   == 1 )  /* block someone */
                       goto broad,
                    else {
                       /* did not block any body, lock unlock
                       * NO need to broad blocking message
                       * cpu time is ignored
                       * notice  no disk read is needed
                       * since r_lock can protect the read set to be
                       * altered
                       *
                       * the following check is a redundancy checking
                       * since pg should have not be changed right
                       * after it get the lock for get_local_lock in
                       * the same function code
                       * the code is questionable  No test yet
                       */

                       if ( cc_lock_get_bus ( tr, site ) == FAIL ){
                          double F;
                          F = 20 * INSTRUCTION_TIME + cc_time,
                          schedule ( RELEASE_BUS, F, token ),
                          return,
                          }

                       cc_lock ( tr, site ),
                       cc_unlock ( tr, site );
                       resume_by ( tr, site ),
                       terminate_site ( token, tr, site ),
                       terminate_tr ( tr ),
                       break,
                       }
                    }          /* Local Query */

broad               schedule ( NEED_BUS_TOKEN, DELTA_TIME, token ),
                    break,

              case WAIVE
                    terminate_site ( token, tr, site ),
                    terminate_tr ( tr ),
                    break,
                }
            break,
            }
}
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*
*      waken_up() ---
*
*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/
waken_up(token)
register int token,
{
register int i,tr,rel,site,att, j,k,
site = site_of [ token ],
tr   = tr_of   [ token ],

switch ( th_state [tr][site] ) {

        /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```



```
        *                  LP phase
        *-----------------------------------------------------*/

        case LP_WAIT_LOCK

             before = timestamp (),
/*           blocked_site [ tr ] - ,    */
             goto_state ( LP_PRE_LOCK ) ,

        case LP_PRE_LOCK       /* entry point of a waken process */

             if ( get_local_lock ( token, tr, site ) == REJECT ) {

                pre ( BLOCKED ! ),
                go_sleep ( tr, site, LP_WAIT_LOCK ),
                blocked_site [ tr ] ++,
                return,
                }

             if ( class_of [ tr ] != REMOTE ) {

                if ( cc_lock_get_bus ( tr, site ) == FAIL ){
                   pre ( ERROR in obtained locks   pg conflict ),
                   schedule ( STOP, 0 0, token ),
                   }

                cc_lock ( tr, site );
                }

             after = timestamp (),
             execution_time = after - before + 20 * INSTRUCTION_TIME,
             schedule_C ( NEED_DISK, 1 0 * execution_time, token ),
             break,

        case LP_WAIT_END .

             if ( est_slowest_site_of [ tr ] == site ) {

                schedule C ( NEED_BUS_TOKEN, DELTA_TIME, token ),
                /* est_slowest_site of [ tr ] = INF, */
                return,
                }

        /*-----------------------------------------------------
        *                  GSJ phase
        *-----------------------------------------------------*/

        case GSJ_INI : /* the site had been picked as min_site */

             goto_phase ( GSJ ) ,

gsj_ini :
             schedule_C ( NEED_BUS_TOKEN, DELTA_TIME, token ),
             break,

        case GSJ_WAIT_LOCK

             if ( get_local_lock ( token, tr, site ) == REJECT ) {

                pre ( BLOCKED ! );
                go sleep ( tr, site, GSJ_WAIT_LOCK ),
                blocked_site [ tr ] ++,
                return,
                }

             if ( blocked_site [tr] == 0 )
                goto gsj_ini,
             else go sleep ( tr, site, GSJ_INI ),

             break,

        case GSJ_WAIT_SJ

             num_in_gsj [ tr] ++,
             schedule_C ( NEED_DISK, DELTA_TIME, token ),
             break,

        case GSJ_WAIT_END

             if ( last_sj [ tr] == 0 ) {

                /*
                * This is a site had bro all its site att
                * In this case, the site is waked up to perform
                * semi joins with the remaining part the DB
                */

                num_in_gsj [ tr] ++,
                schedule_C ( NEED_DISK, DELTA_TIME, token ),
                }
             else {
                /*
                * this is the last sj, go to RT phase
                *   check if this is the first site then do it
```

```
        *   otherwise stay at RT_WAIT_START state
        */

        if ( site == home_of [ tr ] ) {

            go_sleep ( tr, site, PP_WAIT_START ),
                    /* pseudo broadcast its relations */
            for_all_rel
                if ( the_rel_in_site_is_queried )
                    bro_rel [ tr ] [ rel ] = HAD_BRO ,
            return,
            }

        if ( first_rt_site [ tr] == 0 )
            first_rt ( tr ),

        if ( site != first_rt_site [ tr ] ) {
            go_sleep ( tr, site, RT_WAIT_START ),
            return,
            }

        goto_state ( RT_START ),
        goto_phase ( RT );
        schedule_C ( NEED_DISK, DELTA_TIME, token );
        }
    break,

/*-------------------------------------------------------------
*               RT phase
*------------------------------------------------------------ */

case RT_WAIT_START      /* non home site only */

    goto_state ( RT_START ),

    /*
    *   Need some time to do remaining semi - join
    */

    schedule_C
        ( NEED_DISK, DELTA_TIME + INSTRUCTION_TIME, token ),
    break,

/*-------------------------------------------------------------
*               PP phase
*------------------------------------------------------------ */

case PP_WAIT_UPDATE      /* non home site only */

    goto_state ( PP_UPDATE ),

    /*
    * assume no time is need to prepare data for updating disk
    */

    schedule_C ( NEED_DISK, DELTA_TIME, token ),
    break,

case PP_WAIT_START

    goto_phase ( PP ),
    goto_state ( PP_START ),
            /* get value from disk to process */
    schedule_C ( NEED_DISK, 3 * DELTA_TIME, token ),
    break,

case PP_WAIT_LOCK

    goto_state ( PP_PRE_LOCK ),
    blocked_site [ tr ] --,
    finish ( token ),
    break,
    }
}
/*-------------------------------------------------------------
*
*   cc_lock () ---  set pg permanently in LP, GSJ, PP
*                   copy new_pg to pg
*
*-------------------------------------------------------------
*/

cc_lock ( tr, site )
register int tr, site,
{
register int i, j, tmp,  rel,
int  trj,
int old_pg [ TR + 1 ] ;
double enter_time,

pre ( cc_lock ),

enter_time = timestamp (),
        /* save old pg info for check validity for other transactions */
```



```
repeat ( TR )
    old_pg [ i ] = pg [ tr ] [ i ] ,

loop ( i, 1, TR )
    loop ( j, 1, TR )
        pg [ i ] [ j ] = new_pg [ i ] [ j ] ,

/*
* invalidated those temporary re-locks which is conflicting
* with locks used by this trans and is hold by a trans
* which had no precedence relationship with current trans before
* but is preceded  by it now
*
* Don't care about directive precedence relationship
* sonce no conflict may exit between them
*
* This should be moved to the time after the bus is released
*/

loop ( i, 1, SITE )            /* check site by site to save time */

    if ( i !=  site
        &&   pguser [ i ]        != NULL
        &&   pguser [ i ]        != tr
        &&   pg[tr] [pguser[i]] == LbeforeR
        &&   old_pg [pguser[i]] == 0

        &&   lock_validity [pguser[i]] [ i ]  == VALID
        ) {

        /* found a trans in site i  that is newly preceded by tr */

        trj = pguser [ i ] ;

        /* to see if the locks obtained in this site by trj is invalid */

        for_all_rel   if ( queried_rel [ tr ] [ rel ] == 1 )

            if (
                (
                    (     lock_state [ tr ] [ rel ] == READING
                      || lock_state [ tr ] [ rel ] == READING_WANT_W
                    )
                    &&
                    (     local_lock [ i  ] [ rel ] == WRITING
                      || local_lock [ i  ] [ rel ] == READING_WRITING
                    )
                )
                ||
                (
                    (     lock_state [ tr ] [ rel ] == WRITING
                      || lock_state [ tr ] [ rel ] == READING_WRITING
                    )
                    &&
                    (     local_lock [ i  ] [ rel ] == READING
                      || local_lock [ i  ] [ rel ] == READING_WANT_W
                    )
                )
            )

            {
            lock_validity [ trj ] [ i ] = INVALID,
            break,
            }
    }

cc_time = timestamp () - enter_time,
return ( SUCC ),
}

/*-------------------------------------------------------------
*
*      cc_lock_get_bus () ---  set lock permanently in LP, GSJ, PP
*                      copy local_lock to lock_state, reset flag pguser
*
*-------------------------------------------------------------
*/

cc_lock_get_bus ( tr, site )
register int tr, site,
{
register int i, j, tmp,  rel,
int  trj;
int old_pg [ TR + 1 ] ;
double enter_time,

pre ( cc_lock_get_bus ),

enter_time = timestamp (),
        /* save old pg info for check validity for other transactions */
repeat ( TR )
    old_pg [ i ] = pg [ tr ] [ i ] ,

/*
```

```
2705  * Save pg info in order to be restored at the end
2706  * New information should be stored at new_pg at this moment
2707  * It can only be stored into pg after release bus
2708  * This overhead is due to that cc_lock_get_bus uses pg, not new_pg
2709  */
2710
2711  loop ( i, 1, TR )
2712     loop ( j, 1, TR )
2713        new_pg [ i ] [ j ] = pg [ i ] [ j ] ,
2714
2715  repeat ( TR )
2716     old_pg [ i ] = pg [ tr ] [ i ] ;
2717
2718  loop ( j, 1, TR ) {
2719
2720     if ( new_relative [ tr ] [ j ] == 1 ) {
2721
2722        if ( pg [ tr ] [ j ] == - 1 ) {
2723
2724           new_relative [ tr ] [ j] ,
2725
2726           if ( not_lock_in_LP [ token_of [ tr ] [ site ] ] == 1 )
2727              th_state [ tr ] [ site ] = LP_WAIT_END,
2728
2729           solve_pg_conflict ( tr, site ),
2730           cc_time = timestamp () - enter_time,
2731           return ( FAIL ),
2732           }
2733
2734        transitive ( tr , j ),
2735        new_relative [ tr ] [ j ] = USED,
2736        }
2737
2738     if ( new_relative [ tr ] [ j ] == - 1 ) {
2739
2740        if ( pg [ tr ] [ j ] == 1 ) {
2741
2742           new_relative [ tr ] [ j] ,
2743           solve_pg_conflict ( tr, site ),
2744           cc_time = timestamp () - enter_time,
2745           return ( FAIL ),
2746           }
2747
2748        transitive ( j , tr ),
2749        new_relative [ tr ] [ j ] = -USED,
2750        }
2751     }
2752
2753  /*
2754  * update lock_state, pguser, be careful, may be called more than once
2755  * in every phase including LP phase
2756  */
2757
2758  switch ( tr_phase [ tr ] ) {
2759
2760     case LP
2761                 /* Confirm locks for those sites had obtained locks */
2762
2763                 for_all_related_i {
2764
2765                    if (  pguser              [ i ]          == tr
2766                       && lock_validity    [ tr ] [ i ]    == VALID
2767                       ){
2768
2769                       pre ( READING ),
2770                       lock_validity [ tr ] [ i ]    = FIRMED,
2771                       pguser [ i ] = NULL,
2772
2773                       for_all_rel  {
2774                          lock_state[tr][rel] =
2775                          ( lock_state[tr][rel] < local_lock [ i ] [ rel ])
2776                          ? local_lock [ i ] [ rel ]
2777                            lock_state[tr][rel]   ,
2778                          local_lock [ i ] [ rel ] = NULL,
2779                          }
2780
2781                       local_resume ( i ),
2782                       }
2783                    }
2784                 break,
2785
2786     case GSJ
2787
2788                 pre ( READING ),
2789                 pre ( Return local pg ),
2790
2791                 for_all_rel
2792                    lock_state[tr][rel] = local_lock [ site ] [ rel ],
2793
2794                 pguser [ site ] = NULL,
2795
2796                 for_all_rel local_lock [ site ] [ rel ] = NULL,
2797                 local_resume ( site ),
2798                 break,
```



```
2799
2800     case PP
2801
2802                 pre ( WRITING ),
2803
2804                 for_all_rel
2805                    lock_state[tr][rel] = local_lock [ site ] [ rel ],
2806
2807                 pguser [ site ] = NULL,
2808                 for_all_rel local_lock [ site ] [ rel ] = NULL,
2809                 local_resume ( site ),
2810                 break,
2811     }
2812
2813  /*
2814  * restore pg back to old pg
2815  * and store new pg into new_pg
2816  */
2817
2818  loop ( i, 1, TR ) loop ( j, 1, TR ) {
2819     tmp = new_pg [ i ] [ j ] ,
2820     new_pg [ i ] [ j ] = pg [ i ] [ j ] ,
2821     pg [ i ] [ j ] = tmp,
2822     }
2823
2824  cc_time = timestamp () - enter_time,
2825  return ( SUCC ),
2826  }
2827
2828  /*-------------------------------------------------------------------
2829  *
2830  *     cc_new_tr () ---  update BTT under different cases
2831  *
2832  *                       CALLED BY   PRE on RELEASE_BUS
2833  *-------------------------------------------------------------------
2834  * NEW_TRANSACTION
2835  * the lock_state store the lock requirement of each trans to
2836  * each relation  The WANT_W is covered by the WANT_R  Each update
2837  * is associated with a R          NO WAIVE MAY OCCURRED !!!!
2838  * This routine is designed for general case with waive possibility
2839  */
2840
2841  cc_new_tr ( tr )
2842  register int tr,
2843  {
2844  register int i, rel,
2845  register int NEED_R, NEED_W,
2846  pre ( cc_new_tr ()),
2847
2848  loop ( i, 1, TR ) {
2849
2850     new_relative  [ tr ] [ i  ] = 0,
2851
2852     if ( new_relative  [ i  ] [ tr ] > 0 )
2853        new_relative  [ tr ] [ i  ] = - 1,
2854
2855     if ( new_relative  [ i  ] [ tr ] < 0 )
2856        new_relative  [ tr ] [ i  ] = 1,
2857     }
2858
2859  for_all_rel {
2860
2861     NEED_R  = ( queried_rel [ tr ] [ rel ] == 1    ),
2862     NEED_W  = ( writing_rel [ tr ]         == rel  ),
2863
2864     if ( NEED_R  == 1 && NEED_W == 1 ) lock_state [ tr ][ rel ] = WANT_RW,
2865     if ( NEED_R  == 1 && NEED_W == 0 ) lock_state [ tr ][ rel ] = WANT_R,
2866     if ( NEED_R  == 0 && NEED_W == 1 ) lock_state [ tr ][ rel ] = WANT_W,
2867     if ( NEED_R  == 0 && NEED_W == 0 ) lock_state [ tr ][ rel ] = NULL,
2868     }
2869
2870  loop ( i, 1, TR   )  pg [ tr ] [ i  ] =
2871                       pg [ i  ] [ tr ] = 0,
2872  loop ( i, 1, SITE )  lock_validity [ tr ] [ i ] = VALID,
2873
2874  }
2875
2876  /*-------------------------------------------------------------------
2877  *
2878  *     cc_unlock () ---  unlock  update BTT,  lock_state
2879  *                       pg is not updated since the precedence relation
2880  *                       can not be changed before termination
2881  *
2882  *       CALLED by LP_SYN_EST_BRO, GSJ_SJ_BRO, RT_START, PP_PRE_LOCK
2883  *                 on RELEASE_BUS
2884  *
2885  *-------------------------------------------------------------------
2886  * Different responses for different operating modes
2887  * AYTIME   the second copy of relations are stored after LP
2888  * GSJ_END  only projected attributes are stored as second copies
2889  * RT_END   No second at all
2890  * LP  phase, unlock those unneeded r locks ( read_only, locked in LP )
2891  * GSJ phase, unlock those unneeded r locks ( read_only, locked in GSJ )
2892  * RT phase, unlock those unneeded r locks (read_only, locked by orphan )
```

```
2893  * PP  phase, unlock all r-locks and w-locks
2894  * In each phase, the set of locked relations are identified first
2895  *         since a copy of the relation is assumed copied into a second
2896  *         place, it can be unlocked immediately after initial broadcast
2897  *         or semi-join broadcast
2898  */
2899
2900  cc_unlock ( tr, site )
2901  register int tr, site;
2902  {
2903  register int i, rel, att;
2904  int trj, k;
2905  int to_be_unlock [ R + 1 ] ;
2906
2907  switch ( tr_phase [ tr ] ) {
2908
2909      case LP
2910
2911              if ( r_unlock_time == ANYTIME ) {
2912
2913                 /*
2914                 * be careful, there may be some sites haven't get lock yet
2915                 * there may be some sites haven't finish yet if called lat
2916                 * time    when called later, careful too.
2917                 *
2918                 * save all the lock status into the vector to_be_unlock
2919                 * eliminate those locks still used by some sites
2920                 * release those locks no body use it any more
2921                 */
2922
2923                 for_all_rel
2924                    to_be_unlock  [ rel ] = lock_state [ tr ] [ rel ];
2925
2926                 for_all_related_i
2927
2928                    if (  tb_state [ tr ] [ i ] == LP_PRE_LOCK
2929                       || tb_state [ tr ] [ i ] == LP_WAIT_LOCK
2930                       || tb_state [ tr ] [ i ] == LP_DISK
2931                       || tb_state [ tr ] [ i ] == LP_SJ
2932                       )
2933
2934                       for_all_rel                 *
2935
2936                          if ( queried_rel [ tr ] [ rel ] == 1
2937                             && fap [ rel ] [ i ] == 1
2938                             )
2939                             to_be_unlock [ rel ] = 0;
2940
2941                 for_all_rel
2942
2943                    if ( to_be_unlock [ rel ] == READING )
2944                       lock_state [ tr ] [ rel ] --;
2945                 }
2946              break;
2947
2948      case GSJ
2949
2950           if ( r_unlock_time == ANYTIME ) {
2951
2952              /*
2953              * check if the relations locked by this site
2954              * are not locked by other site
2955              * if not, unlock them
2956              */
2957
2958              for_all_rel
2959                 to_be_unlock  [ rel ] = lock_state [ tr ] [ rel ];
2960                                                    .
2961              /*
2962              * The following piece of codes is redundant since once
2963              * not_lock_in_LP is set, the relation will not exist here
2964              * for the current  implementation
2965              * ( only one copy in the system )
2966              */
2967
2968              for_all_related_i
2969
2970                 if (          i != site
2971                    && not_lock_in_LP[ token_of [ tr ][ i ]] == 1
2972                    )
2973
2974                    for_all_rel
2975
2976                       if (  queried_rel [ tr ] [ rel ] == 1
2977                          && fap [ rel ] [ i ] == 1
2978                          )
2979                          to_be_unlock [ rel ] = 0;
2980
2981              for_all_queried_rel_in_site
2982
2983                 if (  relation_conf [rel][ broadcasting_att [ tr ] ] == 1
2984                    && to_be_unlock [ rel ] == READING
2985                    ){
2986
```



```
2987                    /*
2988                    * The r-locked rel contains the bro att, could be
2989                    * a candidate for unlocking
2990                    */
2991
2992                    k = 0;       /* check to see if all att had been bro */
2993
2994                    for_all_att
2995                       if (    relation_conf [ rel ] [ att ]       == 1
2996                          &&   queried_att [ tr ] [ rel ] [ att ] == 1
2997                          ){
2998                                /* check if the att had been broadcast */
2999                          if ( bro_component [tr][ att ][ rel ] == WAIT_BRO){
3000
3001                             k = 1;
3002                             break;
3003                             }
3004
3005                          }
3006
3007                    if ( k == 0 ) lock_state [ tr ] [ rel ] --;
3008
3009                    }           /* had r_locked */
3010              }                 /* ANYTIME */
3011
3012           if ( r_unlock_time == GSJ_END ) {
3013
3014              /*
3015              * consider those non_target relations only
3016              * check if all the atts had been broadcast
3017              * if yes, unlock them
3018              */
3019
3020              for_all_queried_rel_in_site
3021
3022                 if ( target_rel[tr][rel] == 0 ){
3023
3024                    if(relation_conf [rel][ broadcasting_att [tr]] == 1
3025                       && lock_state [ tr ] [ rel ] == READING
3026                       ){
3027
3028                    /*
3029                    * The r-locked rel contains the bro att, could be a
3030                    * candidate for unlocking
3031                    */
3032
3033                    k = 0;  /* check to see if alle att had been bro */
3034
3035                    for_all_att
3036                       if (  relation_conf [ rel ] [ att ]       == 1
3037                          && queried_att [ tr ] [ rel ] [ att ] == 1 ){
3038
3039                                /* check if the att had been bro */
3040                          if(bro_component[tr][att][rel]== WAIT_BRO ){
3041                             k = 1;
3042                             break;
3043                             }
3044                          }
3045
3046                    if ( k == 0 ) lock_state [ tr ] [ rel ] --;
3047
3048                    }           /* had r_locked */
3049                 }              /* non   target */
3050              }                 /* GSJ_END */
3051
3052           break;
3053
3054   case RT
3055
3056           for_all_rel
3057
3058              if (     lock_state [ tr ] [ rel ] == READING
3059                 &&    bro_rel    [ tr ] [ rel ] == HAD_BRO
3060                 )
3061
3062                 lock_state [ tr ] [ rel ] --;
3063           break;
3064
3065   case PP
3066
3067           /*
3068           * From now on, this transaction has nothing to do with
3069           * cc any more  It can be treated as having been terminated
```

```
3070                */
3071
3072                if ( class_of [ tr ] == LOCAL_QUERY ) {
3073
3074                   int t_lock;
3075
3076                   t_lock = lock_state [ tr ] [ writing_rel [ tr ] ];
3077
3078                   for_all_rel
3079                      lock_state [ tr ] [ rel ] = NULL;
3080                   lock_state [ tr ] [ writing_rel [ tr ] ] = t_lock;
3081                   }
3082
3083                else {
3084
3085                   for_all_rel     lock_state [ tr ] [ rel ] = NULL;
3086                   loop ( i, 1, TR ) pg [ i ][ tr ] = pg [ tr ] [ i ] =  0;
3087                   }
3088
3089                break;
3090          }
3091
3092       pre ( cc_unlock );
3093       }
3094
3095       /*-------------------------------------------------------------------------
3096       *
3097       *     child_gen () ---  generate child processes
3098       *                       set up necessary information
3099       *                       CALLED BY : release_bus PRE
3100       *
3101       *-------------------------------------------------------------------------
3102       */
3103       child_gen ( tr )
3104       register int tr;
3105       {
3106       register int j, rel, site, att, token;
3107       int home_site, home_token;
3108
3109       pre ( child_gen );
3110       home_site  = home_of  [tr];
3111       home_token = token_of [tr] [ home_site ];
3112
3113       for_all related_site  {/* something in home site should also be set */
3114
3115          if ( site != home_site ) {
3116
3117             if ( ( token = get_token_id ()) == 0 ) {
3118                pre ( Token pool full  '' );
3119                return ( ERROR );
3120                }
3121
3122             site_of  [ token ] = site;
3123             tr_of    [ token ] = tr;
3124             token_of [ tr ] [ site ] = token;
3125             th_state [ tr ] [ site ] = LP_INI;
3126             num_in_lp [ tr] ++;
3127             }
3128
3129          else token = home_token;
3130
3131          local_min_att [ tr ] [ site  ] = 0;
3132          min_bv        [ tr ]           = INF;
3133          bv            [ tr ] [ site  ] = INF;
3134          orphan        [ token ] = 0;
3135
3136          /* obsolete codes
3137          loop ( j, 1, TR ) {
3138             local_pg   [ site ] [ tr ] [ j  ] = 0;
3139             local_pg   [ site ] [ j  ] [ tr ] = 0;
3140             }
3141          for_all_rel  local lock is set in LOCAL_PRE_LOCK
3142          local_lock    [ site  ] [ tr    ] [ rel   ] = 0;
3143          */
3144
3145          /* those array need to be filled here */
3146
3147          not_lock_in_LP [ token ] = 0;
3148          for_all_att
3149             if ( site_rel_conf[tr][site][att] == 1 )
3150                bro_site_att  [ tr ] [ site ] [ att ] = WAIT_BRO;
3151
3152          /*
3153          * set up local relation and att cardinality, it should be reduced
3154          * by the local semi-join using the sam estimation tech in
3155          * GSJ  Temporary set to ini_rel_care, then reduced in dbsize ()
3156          */
3157
3158          for_all_rel
3159                rel_card    [ tr ] [ rel  ] [ site ] = ini_rel_card [rel];
3160          for_all_att
3161             site_att_card [ tr ] [ site ] [ att  ] = 0;
3162
3163          if ( site != home_site )
```



```
3164             release_bus ( token );            /* skip to that point */
3165          }
3166       }
3167
3168
3169       /*-------------------------------------------------------------------------
3170       *
3171       *     consistency () --- Check the consistency of a lock request
3172       *                        from transaction tri at site `site'
3173       *                        This is the implementation of procedure
3174       *
3175       *                          cc - minimumlock
3176       *
3177       *            CALLED BY
3178       *-------------------------------------------------------------------------
3179       * check r-w conflicts first then check w-w conflict for all the
3180       * requested locks needed by the tri in the site
3181       * The new precedence relations are record in the vector new_relative
3182       */
3183
3184       consistency ( token, tr, site )
3185       int token;
3186       int tr;
3187       register int site;
3188
3189       {
3190       int  request_status;
3191       int  trj, rel;
3192       int  flag;
3193       int  j;
3194       register int tri;
3195       FILE     *fp1, *fopen ();
3196
3197       tri = tr;
3198       request_status = GOT_LOCK;
3199
3200       /* check for the r - w conflict first */
3201
3202       loop ( trj, 1, TR )       if ( tri != trj ) {
3203
3204          if (           pg        [ tri ] [ trj ] == 0
3205             &&      new_relative [ tri ] [ trj ] == 0
3206             ){                              /* no precedence relation before */
3207
3208             /*
3209             * The second condition is for resolving the inconsistency caused
3210             * by different sites requesting locks in different time
3211             * The new_relative setting my be inconsistent since the
3212             * situations are different
3213             */
3214
3215             if ( rw_conflict ( tri, trj, site ) == 1 ) {
3216                                                /* has r-w conflict */
3217                flag = 0;
3218                for_all_rel if ( the_rel_in_site_is_queried ) {
3219
3220                     switch ( lock_state [ trj ] [ rel ] ) {   /* classify */
3221
3222                      /* the following condition is eliminated to prevent
3223                      * the two diff trans from setting new_relative to -1
3224                      * such that deadlock to each other, this may cause some
3225                      * conflict, but it is more close to real execution
3226                      */
3227
3228                      case WRITING
3229                      case READING_WRITING    : flag = 1; goto next; break;
3230                      case HAD_READ           :
3231                      case WRITTEN            :
3232                      case HAD_RW             : flag = 2 ;            break;
3233                      }/* switch */ }
3234       next   :
3235
3236       switch ( flag ) { /* process for diff situation */
3237
3238                      case 0          /* all locks haven't beed used by  trj */
3239
3240                              pre ( case 0 );
3241                              if ( new_relative [ tri] [ trj ] == - 1
3242                                || new_relative [ tri] [ trj ] == - USED ) {
3243                                 pre ( ERROR in new_relative 179 );
3244                                 pri ( new_relative [ tri] [ trj] ) ;
3245                                 schedule ( STOP, 0 0, token );
3246                                 }
3247                              else if ( new_relative [ tri ][ trj ] != USED )
3248                                 new_relative [ tri] [ trj ] = 1;
3249
3250                              /* transitive ( tri, trj ); */
3251                              break;
3252
3253                      case 1          /* some locks is being used by trj  */
3254                                      /* this is a redundant test */
3255
3256                              pre ( case 1 );
3257                              if ( new_relative [ tri] [ trj ] ==  1
```

```
3268                            || new_relative [ tri] [ trj ] ==  USED ) {
3269                            pre ( ERROR in new_relative 190 ),
3270                            pri ( new_relative [ tri] [ trj] ) ,
3271                            schedule ( STOP, 0 0, token );
3272                            }
3273
3274                         else if( new_relative[ tri ][ trj ] != -USED ){
3275                            new_relative [ tri] [ trj ] = - 1,
3276                            }
3277                         blocked_by [ token ] = trj ;
3278                         RETURN_REJECT,
3279                         break;
3280
3281                case 2 :        /*
3282                                * all locks had been released by trj
3283                                * transitive ( trj, tri ),
3284                                */
3285
3286                         pre ( case 2 ),
3287                         if ( new_relative [ tri] [ trj ] ==  1 ) {
3288                            pre ( ERROR in new_relative 204 ),
3289                            pri ( new_relative [ tri] [ trj] ) ,
3290                            schedule ( STOP, 0 0, token ),
3291                            }
3292
3293                         else if( new_relative [ tri ] [ trj ] != USED)
3294                            new_relative [ tri] [ trj ] = - 1,
3295
3296                         break,
3297                }                       /* switch */
3298            }                           /* has r-w conflict */
3299        }                               /* no precedence relation before */
3300
3301    else if ( (     pg              [ tri ] [ trj ]     ==   RbeforeL
3302              ||    new_relative [ tri ] [ trj ]     ==   RbeforeL
3303              )
3304           &&     rw_conflict ( tri,   trj, site ) ==   1
3305           ) {
3306
3307           blocked_by [ token ] = trj;
3308           RETURN_REJECT,
3309           }
3310
3311    } /* loop trj */
3312
3313 if ( request_status == REJECT )
3314    return_reject,
3315
3316 if (  writing_rel [ tri ] == 0
3317    || tr_phase    [ tr ]  != PP
3318    ){                          /* read lock, set lock state locally */
3319
3320    for_all_rel
3321
3322       if (          the_rel_in_site_is_queried
3323
3324           && (  local_lock [ site ] [ rel ] == WANT_R
3325              || local_lock [ site ] [ rel ] == WANT_RW
3326              )
3327          )
3328
3329       local_lock [ site ]  [ rel ] ++  ,
3330
3331    return ( GOT_LOCK ),
3332    }
3333
3334 /* check for the w-w conflict then */
3335
3336 loop ( trj, 1, TR )
3337
3338    if ( tri != trj ) {
3339
3340       if (    pg [ tri ] [ trj ]              ==   0
3341          &&   new_relative [ tri ] [ trj ]    ==   0
3342          ){
3343                                   /* no precedence relation before */
3344
3345       if ( ww_conflict ( tri, trj ) == 1 ) {
3346                                   /* transitive ( tri, trj ), */
3347          pre ( ww_conflict ),
3348          if ( new_relative [ tri] [ trj ] == - 1 ) {
3349             pre ( ERROR in new_relative 242 ),
3350
3351             schedule ( STOP, 0 0, token ),
3352             }
3353
3354          else if ( new_relative [ tri ] [ trj ] != USED )
3355             new_relative [ tri] [ trj ] = 1,
3356
3357          lock_state [ tr ] [ writing_rel [tr ]  ] ++,
3358          return_waive,
3359          }
3360       }                           /* no precedence relation before */
3361
```



```
3362    else if (     (  pg    [ tri ] [ trj ] == RbeforeL    /*  tri <- trj */
3363                  || new_relative [ tri ] [ trj ] == RbeforeL
3364                  )
3365             &&(     lock_state [ trj ] [writing_rel [tr]] == WRITING
3366                  || lock_state [ trj ] [writing_rel [tr]]
3367                                             == READING_WRITING
3368               )
3369             )
3370       {
3371
3372       blocked_by [ token ]          =   trj,
3373       request_status                =   REJECT,
3374       new_relative [ tri ] [ trj ] =   -1,
3375       }
3376
3377    else if ( ( pg  [ tri ] [ trj ] ==  LbeforeR  /*  tri -> trj */
3378              || pg  [ tri ] [ trj ] ==  LbeforeR
3379               )
3380
3381          &&(   ww_conflict ( tri, trj ) == 1 )
3382
3383          &&(   lock_state [ trj ] [ writing_rel [tr] ] == WRITING
3384             || lock_state [ trj ] [ writing_rel [tr] ] == READING_WRITING
3385             || lock_state [ trj ] [ writing_rel [tr] ] == WRITTEN
3386             || lock_state [ trj ] [ writing_rel [tr] ] == HAD_RW
3387             ) ) {
3388
3389          lock_state   [ tr  ] [ rel ] ++,
3390          new_relative [ tri ] [ trj ] = 1,
3391          return_waive,
3392          }
3393
3394    }                                 /* loop trj */
3395
3396 if ( request_status == REJECT )
3397    return_reject,
3398
3399 rel =  writing_rel [ tr ],
3400
3401 if ( rel == WANT_W || rel == READING_WANT_W )
3402    local_lock [ site ]  [ rel ] ++,
3403
3404 return ( GOT_LOCK ),
3405 }
3406
3407 /*-----------------------------------------------------------------------
3408 *
3409 *     dbsize () --- change the size of DB after a semi   join
3410 *
3411 *            CALLED BY : FINISH ( LP_SYN_EBT_BRO, LP_WAIT_SYN )
3412 *                        RELEASE_BUS ( GSJ_SJ_BRO, GSJ_WAIT_END )
3413 *-----------------------------------------------------------------------
3414 *
3415 * Calculate real and estimated cardinalities of atts and rels in all
3416 * phase.   Cardinalities are assumed reduced by  selectivities
3417 *
3418 * new cardinality = old cardinality  * selectivity of broadcasting att
3419 *
3420 * The estimated cardicalities is assumed uniformly distributed between
3421 * +- ( DEVIATION )% of real values
3422 * The original selectivity, which is equivalent to relative selectivity,
3423 * of each att is kept
3424 * In LP phase, the cards of all atts in one domain are the same,
3425 * since they can be fully reduced in a site
3426 * In GSJ phase, the unbroadcast components of the broadcasting att are
3427 * abstracted out, each site take those abstracted components that are
3428 * not  in the site  to reduce the cards of all atts and relations
3429 */
3430
3431 dbsize ( tr, site )
3432 register int tr, site,
3433 {
3434 register int  i, rel, att, j,
3435 int effect_component  [ R + 1 ],
3436 int local_effect_component  [ R + 1 ],
3437
3438 pre ( dbsize ),
3439
3440 switch ( tr_phase [ tr ] ) {
3441
3442    case LP
3443                         /*
3444                         *  Cards of all rels reduced by all applicable
3445                         *  selection effect on it
3446                         */
3447
3448          loop ( j, 1, SITE ) {
3449
3450             for_all_rel (
3451                rel_card     [ tr ] [ rel ] [ , ] =
3452                set_rel_card [ tr ] [ rel ] [ j ] = 0,
3453                }
3454
3455             for_all_att (
```

```
3446                site_att_card [ tr ] [ j ] [ att ] =
3447                est_att_card  [ tr ] [ j ] [ att ] = 0.
3448                }
3449             }
3450
3451          for_all_related_i   for_all_rel
3452
3453             if (     the_rel_in_site_i_is_queried      )
3454
3455                est_rel_card [ tr ] [ rel ] [ i ] =
3456                rel_card     [ tr ] [ rel ] [ i ] =
3457                ini_rel_card [ rel ].
3458
3459          for_all_related_i {
3460
3461             double site_att_select,
3462             double est_site_att_select ,
3463
3464             /*
3465             * build up the site_att_card ( selectivity )
3466             * and the estimated attribute cardinalities
3467             */
3468
3469             for_all_att   if ( site_rel_conf [ tr ][ i ][ att ] == 1 ){
3470
3471                site_att_select     =
3472                est_site_att_select = 1.0,
3473
3474                for_all_rel  /* see each relation if they contains att */
3475
3476                   if (  queried_att [ tr  ] [ rel ] [att] == 1
3477                      &&       fap    [ rel ] [ i   ]      == 1
3478                      ) {
3479
3480                      site_att_select *= scatter(selectivity[rel][att]),
3481                      est_site_att_select *= selectivity[rel][att].
3482                      }
3483
3484                if ( site_rel_conf [ tr ] [ i ] [ att ] == 1 ) {
3485
3486                   site_att_card [ tr ] [ i ] [ att ]
3487                      = domain_range [ att ] * site_att_select,
3488                   est_att_card  [ tr ] [ i ] [ att ]
3489                      = domain_range [ att ] * est_site_att_select,
3490                   }
3491
3492                for_all_rel  /* set up relation card */
3493
3494                   if (  queried_att [ tr  ] [ rel ] [ att ] == 1
3495                      &&       fap    [ rel ] [ i ]         == 1
3496                      ) {
3497
3498                      rel_card [ tr ] [ rel ] [ i ] =
3499                      rel_card [ tr ] [ rel ] [ i ] *
3500                         site_att_select / selectivity[ rel ][ att ],
3501                      est_rel_card [ tr ] [ rel ] [ i ] =
3502                      est_rel_card [ tr ] [ rel ] [ i ] *
3503                         est_site_att_select/selectivity[ rel ][ att ],
3504                      }
3505
3506                } /* for all att */
3507             } /* for_all_related_i */
3508          break,
3509
3510    case GSJ
3511
3512          att = broadcasting_att [ tr ],
3513                   /* figure out the component that in the bro att */
3514          for_all_rel {
3515
3516             effect_component [ rel ] =
3517                      component_set [ tr ][ site ][ att ] [ rel ],
3518             if ( bro_component [ tr] [ att] [ rel ] == HAD_BRO )
3519                effect_component [ rel ] = 0,
3520             }
3521
3522          for_all_related_i {
3523
3524             if ( i != site && site_rel_conf [ tr] [ i ] [ att] == 1 ) {
3525                                /*  not the min_site */
3526
3527                for_all_rel {
3528                            /*
3529                            *  set up local effective component
3530                            *  strip out the components that exist here
3531                            */
3532                   local_effect_component [rel]= effect_component [rel],
3533
3534                   if ( fap [ rel ] [ i ] == 1 )
3535                      local_effect_component [ rel ] = 0,
3536
3537                   }        /* all rel */
3538
3539                /*
```



```
3540                *  start to calculate new card of all atts and rels
3541                *  count the effect of each bro component one by one
3542                *  rel is the component in broadcast att
3543                *  j is the indices of local relations
3544                */
3545
3546                for_all_rel
3547
3548                   if (      local_effect_component [ rel ]     == 1
3549                      &&     queried_att [ tr ] [ rel ] [ att ] == 1
3550                      ) {
3551
3552                      site_att_card    [ tr ] [ i ] [ att ] =
3553                         site_att_card [ tr ] [ i ] [ att ]
3554                            * scatter ( selectivity [ rel ] [ att ] ),
3555
3556                      loop ( j, 1, R )
3557                         if (          fap    [ j  ] [ i ]       == 1
3558                            &&    queried_att [ tr ] [ j ] [att] == 1
3559                            )
3560                            rel_card [ tr ] [ j ] [i ] =
3561                               rel_card [ tr ] [ j ] [i ]
3562                               * scatter ( selectivity [ rel ] [ att ]),
3563
3564                      } /*  if  */
3565                   } /* non min_site */
3566                } /* for_all_related_i */
3567          break,
3568
3569    case BT
3570             /*
3571             * assume the multi-att joining effect is not considered
3572             */
3573          break,
3574
3575    case PP          /*
3576                     * for Local Queries only
3577                     */
3578          for_all_queried_rel_in_site
3579
3580             for_all_queried_att_in rel
3581
3582                loop ( j, 1, R )
3583
3584                   if ( j != site && queried_att [tr][ j ][ att ] == 1 )
3585
3586                      rel_card [ tr ] [ rel ] [ site ]
3587                         *= selectivity [ j ] [ att ] ,
3588
3589                      pptime [ tr ] [ site ] =
3590                           rel_card [ tr ] [ rel ] [ site ]
3591                           * CPU_TIME_PER_BIT,
3592          break,
3593    } /* switch */
3594 }
3595 /*-----------------------------------------------------------------------
3596 *
3597 *     est_slowest_site () --- estimate slowest site
3598 *
3599 *          if haven't  called yet, slowest_site[tr] = 0
3600 *
3601 *-----------------------------------------------------------------------
3602 * assume the site with the largest total relation cardinality is
3603 * the slowest site
3604 *
3605 * To prevent the system from deadlock due to the unavailability
3606 * of local_pg to the other transactions which hold locks needed by
3607 * the estimated slowest site. The estimated slowest site should be
3608 * changed after timeout or the detection of deadlock.
3609 * The blocked site may have higher priority of broadcasting "unfinished"
3610 * message to reduced the number of induced messages.
3611 *
3612 * To reduce the programming difficulty, we choose a simple strategy
3613 * to resolve the deadlock. Once the estimated site gets blocked, it
3614 * should change the right to another site that is not blocked.
3615 */
3616 est_slowest_site ( tr )
3617 register int tr,
3618 {
3619 register int i, j, rel, site, token ,
3620 int k, att,  slowest_site, max_time, size, est_card,
3621 int flag,
3622 prs ( slowest?),
3623
3624 max_time = -VAXMAX,
3625 slowest_site = 0,
3626 flag = 0;
3627
3628 /***********/
3629    avoid
3630 /***********/
3631
3632 for_all_token_of ( tr ){
3633    site = site_of[token],
```

```
    if (  (                     site               !=       home_of  [tr]
          ||    tb_state [tr] [ site ]             !=       PP_WAIT_START
          )
       && (     flag      ==      0
          ||    site      !=      slowest_site
          )
       ) {

       J = O;

       for_all_queried_rel_in_site
              J += ( est_rel_card [ tr ] [ site ] [rel]
                  * tr_rel_width [ tr ] [ rel ]);

       if ( J > max_time ) {
          max_time =  J;
          slowest_site = site;
          }
       }
    }

if ( tb_state [ tr ] [ slowest_site ] == LP_WAIT_END ) {

   flag = 1;
   goto avoid;            /* avoid select a site already in LP_WAIT_END */
   }

est_slowest_site_of [ tr ] = slowest_site,
}

/*----------------------------------------------------------------------
*
*     first_rt () --- get the first rt site ( smallest site number )
*                          CALLED BY : WAKE_UP  GBJ_WAIT_END
*
*----------------------------------------------------------------------
*/
first_rt ( tr )
int tr,
{
int site,
pre ( first_rt ),
for_all_child site  break,
first_rt_site [ tr ] = site ;
return ( site );
}

/*----------------------------------------------------------------------
*
*     get_local_lock () ---
*
*              CALLED BY    WAKE_UP  * WAIT_LOCK
*  ---------------------------------------------------------------------
*/

get_local_lock ( token, tr, site )
int token,
register int tr, site,
{
register int i, rel, j,

pre ( get_local_lock ()  );

/*
* There is a difficulty of determining the new precedence order between
* two transactions that has no precedence relationship before in
* distributed manner
* Sometimes two possibility exist ( i -> j, or j -> i ), a site does not
* know if other sites can get use of local pg or not, therefore
* generated possibilities is only a subset of all possibilities
*
* The straight forward solution is make later transaction always
* later in precedence relationship
*
*/

/* test if there exist active or pending transactions */

if (  pguser [ site ] != NULL ) {
   pre ( local BTT is not available Blocked by ),
   blocked_by [ token ] = TB + 1 ;
   return_reject,
   }

for_all_rel
   local_lock [ site ][rel] = lock_state [tr][rel],

working_site = site,
pguser [ site ] = tr;

switch ( consistency ( token, tr , site ) ) {

    case GOT LOCK
```



```
                          lock_validity [ tr ] [ rel ] = VALID,
                          pre ( GOT LOCK ),
                          lnpre ( LOCK_STATUS ),ln,
                          return ( GOT_LOCK );
                          break.

    case REJECT
                          pguser [ site ] = NULL,
                          lock_validity [ tr ] [ rel ] = INVALID,
                          pre ( REJECT );
                          local_resume ( site ),
                          lnpre ( LOCK_STATUS );ln,
                          blocked_by [ token ] =
                             ( blocked_by [token]== 0 )?1 blocked_by [ token ],
                          return_reject;
                          break,

    case WAIVE
                          pguser [ site ] = NULL;
                          pre ( WAIVE ),
                          local_resume ( site ),
                          lnpre ( LOCK_STATUS ),ln,
                          return_waive,
                          break,
    }
}


/*----------------------------------------------------------------------
*
*     get_related_site(tr) --- figure out the related site of tr
*                                  and set up necessary information
*
*----------------------------------------------------------------------
*/

get_related_site(tr) /* find out the set of relate sites for tr */
register int tr.
{
register int rel,site,att,

   /* test for if it's local read  */

   class_of [tr] = LOCAL_READ,
   for_all_rel
      if(queried_rel [tr] [rel] == 1 && fap [rel] [home_of  [tr]] == 0){
         class_of [tr] = REMOTE,
         break,
         }

   /*  test for if a local transaction  */
   /*   updating a remote copy          */

   if(class_of [tr] == LOCAL_READ && writing_rel [ tr ] != 0) {
      rel = writing_rel [ tr ];
      for_all_site  /* check other site if a copy is there */
      if (site != home_of  [tr] && store_in_(site) ) {
            class_of [tr] = LOCAL_QUERY,
            break,
            }
         }

   /*  set up related sites and test if it's a local read */

   for_all_site {
      related_site [tr] [site] = 0, /* initialize */
      for_all_att
         site_att_of_(tr, site ,att)=0,
      }
   if (class_of [tr] != REMOTE) {
      pre(LOCAL      ),
      }

   for_all_site {
      if(site == home_of  [tr] )
          related_site [tr] [site] = HOME_SITE,
      else  for_all_rel
         if(queried_rel [tr] [rel] ==1 && fap [rel] [site] ==1)
            related_site [tr] [site] = CHILD_SITE,

      /* set up site relation configuration */

      if(related_site[tr][site] >= CHILD_SITE){
         for_all_rel
            for_all_att
               if ( relation_conf [rel] [att] == 1 &&
                     fap [rel] [site] == 1 &&
                     queried_att [tr] [rel] [att] == 1 )
                  site_att_of_(tr ,site ,att) = 1,
         }
      }
   }
/*----------------------------------------------------------------------
*
```

```
3822  *     get_token_id() ---  Get a token from the token pool
3823  *
3824  *-------------------------------------------------------------------
3825  */
3826
3827  get_token_id()              /* get a token_id from token_pool */
3828                              /* (tr_of, site_of ) */
3829  {
3830  int token;
3831
3832  if (new_token == INF ) {
3833     printf ("token pool full \n");
3834     schedule (STOP, 0.0 , 0);
3835     return(0);
3836     }
3837  else{
3838     token=new_token;
3839     new_token=site_of [new_token];
3840     return(token);
3841     }
3842  }
3843
3844  /*-------------------------------------------------------------------
3845  *
3846  *       get_tr_id()
3847  *
3848  *-------------------------------------------------------------------
3849  */
3850
3851  get_tr_id()                 /* get a tr_id from tr_pool */
3852                              /* (home_of ) */
3853  {
3854  int tr;
3855
3856     if (new_tr == INF ) {
3857        printf ("tr_id pool full \n");
3858        /* temporary delayed the arrival rate of new transactions
3859         * It should report to me  in real execution
3860         */
3861        schedule ( NEW_TR , expnti (10000000.0*mean_tr) , 0 );
3862        return(0);
3863        }
3864     else{
3865        tr=new_tr;
3866        new_tr=link_to_next_tr [new_tr];
3867        return(tr);
3868        }
3869     }
3870
3871  /*-------------------------------------------------------------------
3872  *
3873  *     hv_of_site () --- get the heuristic value of a token
3874  *                       put into hv array
3875  *                       hv is assume the size * selectivity
3876  *                       each site scans unbro att and calculates their
3877  *                       h v
3878  *                       choose the minimum one as the site h v
3879  *                       put the id of min att of the site in
3880  *                       local_min_att [ tr] [site], need changed
3881  *                       CALLED by min_site_gen ()
3882  *
3883  *-------------------------------------------------------------------
3884  */
3885  hv_of_site ( token, tr, site )
3886  int token;
3887  register int tr, site;
3888  {
3889  register int  j, att;
3890  int h;
3891
3892  if ( not_lock_in_LP [ token] == 1 ) {
3893
3894     if ( get_local_lock ( token, tr, site )  == REJECT ){
3895
3896        go_sleep ( tr, site, OBJ_WAIT_LOCK );
3897        blocked_site [ tr ] ++;
3898        return ( INF );
3899        }
3900     }
3901
3902  h = INF;
3903
3904  for_all_att
3905
3906     if ( bro_site_att [ tr ] [ site ] [ att ] == WAIT_BRO ) {
3907
3908        I = site_att_card [ tr ] [ site ] [ att ] ;
3909        j =  I * I * domain_width [ att ] / domain_range [ att ] ;
3910
3911        if ( j < h ) {
3912           h = j;
3913           local_min_att [ tr ] [ site ]  = att;
3914           }
3915        }
```



```
3916
3917  hv [ tr ] [ site ] = h;
3918  return ( h );
3919  }
3920
3921  /*-------------------------------------------------------------------
3922  *
3923  *     initial() --- Initialize the simulator
3924  *
3925  *-------------------------------------------------------------------
3926  */
3927
3928  initial()                   /* initialization  */
3929  {
3930  register int i,tr,rel,site,att,j,k,token;
3931  FILE    *fp, *fopen();
3932
3933  fprintf (stderr, " Seed ?" );
3934  scanf("%d", &K);
3935  srand(K);
3936  link_token_id();
3937  link_tr_id();
3938
3939  /* reset data base */
3940
3941  /*   withing tr, rel, att    */
3942
3943  for_all_token blocked_by [ token ] = 0;
3944  for_all_site {
3945
3946     pguser [ site ] = 0;
3947     lost_tr [ site ] = lost_customer = 0;
3948     for_all_tr th_state  [tr][site] = 0;
3949     for_all_rel local_lock [ site ] [ rel ] = 0;
3950     }
3951
3952  total_tr = finished_tr = 0;
3953  total_tr_time = ave_tr_time = 0.0;
3954  total_U_bus = total_U_bus = total_U_disk = total_U_cpu = 0.0;
3955  for_all_tr {
3956
3957     U_bus_tr[tr] = U_disk_tr[ tr] = U_cpu_tr [ tr ] = 0.0;
3958     class_of [tr] =0;
3959
3960     loop (j,1,TR) {
3961        pg [tr][j] = 0;
3962        new_relative [ tr ] [ j ] = 0;
3963        }
3964
3965     for_all_site {
3966
3967        lock_validity [ tr ] [ site ] = VALID;
3968        tr_phase  [tr] = NULL;
3969        related_site [tr] [site] = NULL;
3970        bro_rel [tr] [site] = NULL;
3971
3972        for_all_att {
3973
3974           site_rel_conf  [tr] [site] [att] =0;
3975           bro_site_att  [tr] [site] [att] =NULL;
3976
3977           for_all_rel
3978
3979              component_set  [tr] [site] [att] [rel] =0;
3980           }
3981        }
3982
3983     for_all_rel {
3984
3985        queried_rel [tr] [rel] =
3986        target_rel [tr] [rel] =
3987        updated_rel [tr] [rel] =
3988        target_jatt [tr] [rel] =
3989        updated_jatt [tr] [rel] =0;
3990
3991        lock_state [tr] [rel] = 0;
3992
3993        for_all_att {
3994
3995           queried_att [tr] [rel] [att] =
3996           target_jatt [tr] [rel] [att] =
3997           updated_jatt [tr] [rel] [att] =0;
3998
3999           }
4000        }
4001     }
4002
4003  /* set up initial relation cardinality */
4004
4005  for_all_rel {
4006
4007     ini_rel_card [rel] =
4008     ini_rel_card_ratio [rel];
4009     k=5;
```

```
    for_all_att{

       if (relation_conf [rel] [att] ==1){
          ini_rel_card [rel] *= (domain_range [att]/k);
          k += 7;
          }
       }

    ini_rel_card [rel] /= 10,
    }

for_all_rel {

    relation_width [rel] =relation_nj_width [rel];

    for_all_att

      relation_width [rel] +=
          (relation_conf [rel] [att] ==1)?domain_width [att]:0;
    }
}


/*-----------------------------------------------------------------------
*
*      is_covered () --- check if the component set of the site attribute
*                        is covered by the broadcast component set
*
*-----------------------------------------------------------------------
*/
is_covered(tr,site,att)
int tr,site,att;
{
register int rel;
int cover,

cover = 1,
for_all_rel
    if (component_set [tr] [site] [att] [rel] == 1
         && bro_component [tr] [att] [rel] == NULL) {
         cover=0,
         break,
         }
return(cover),
}

/*-----------------------------------------------------------------------
*
*      link_token_id()
*
*-----------------------------------------------------------------------
*/

link_token_id()          /* set up the token_id pool */
{
int i,
    repeat(TOKEN) {
       tr_of  [i] = 0;
       site_of [i] = i+1,
       }
    tr_of [TOKEN] = INF,
    new_token = 1,
    }

/*-----------------------------------------------------------------------
*
*      link_tr_id()
*
*-----------------------------------------------------------------------
*/

link_tr_id()             /* set up the tr_id pool */
{
int i,
    repeat(TR) {
       home_of  [i] = 0;
       link_to_next_tr [i] = i+1;
       }
    link_to_next_tr [TR] = INF,
    new_tr = 1;
    }
/*-----------------------------------------------------------------------
```



```
*
*      local_resume () --- resume a process which had been blocked by
*                          not available of pg
*
*-----------------------------------------------------------------------
*/

local_resume ( site )
register int site
{
register int i, rel, oldest_tr,
double oldest_time,

pre ( local_resume );
pguser [ site ] = NULL,

for_all_rel
    local_lock [ site ] [ rel ] = NULL,

oldest_tr   = 0,
oldest_time = 1 0 * VAXMAX,

loop ( i, 1, TR )

    if (     blocked_by [ token_of [ i ] [ site ] ] == ( TR + 1 )
        &&   time_of_tr [ i ] < oldest_time
        ) {

        oldest_time = time_of_tr [ i ] ;
        oldest_tr = i  ,
        }

if ( oldest_tr != 0 ) {

    blocked_by [ token_of [ oldest_tr ] [ site ] ] = 0,
    blocked_site [ oldest_tr ] --,
    wake_up ( token_of [ oldest_tr ] [ site ] ),
    }
}
/*-----------------------------------------------------------------------
*
*      materialization () --- Eliminate redundant site attributes
*                             for all site
*                             called by release_bus   GSJ_SJ_BRO
*
*-----------------------------------------------------------------------
*/
materialization ( tr, site, bro_att )
register int tr, site;
register int bro_att,
{
register int i, j, att,
pre ( materialize ),

for_all_related_i
    if (                i                  !=      site
        && (  tb_state [ tr ] [ i ]        ==      GSJ_WAIT_SJ
           || tb_state [ tr ] [ i ]        ==      GSJ_WAIT_SYN_END
           || tb_state [ tr ] [ i ]        ==      GSJ_WAIT_LOCK
           )
        )

    if (  bro_site_att[tr][i][bro_att]  == WAIT_BRO
        && is_covered ( tr, i, bro_att ) == 1
        ) {
        bro_site_att [ tr ] [ i] [bro_att ] = HAD_BRO,
        pre ( 1 att killed ),
        pre ( An att is drawn out );

        /*
        *  check to see if the site had out of site attribute
        */

        j = 0;
        for_all_att

           if ( bro_site_att [tr] [i] [ att ] == WAIT_BRO ) {
              j ++,
              break,
              }

        if ( j == 0 ) { /* no att to be bro in the site */

           if ( tb_state [ tr ] [ i ] == GSJ_WAIT_LOCK )
              blocked_site [ tr ] --,

           go_sleep ( tr, i , GSJ_WAIT_END ),
           not_lock_in_LP [ token_of [ tr ] [ i ] ] == 0,
           }
        }
}

/*-----------------------------------------------------------------------
*
```

```
4187 *      min_site_estimate () --- return estimated min_site
4188 *
4189 *              Notice that each site should used its own estimated
4190 *              value to contend the min_est_site in the simulation
4191 *              even though its hv is known, otherwise, conflicts may
4192 *              occur. Diff  TH with diff  results
4193 *              When broadcast, the estimated min site can broadcast
4194 *              real min att in its site
4195 *
4196 *          CALLED BY : pick_min_site () in LP
4197 *-- ----------------------------------------------------------------
4198 */
4199 min_site_estimate ( tr, site )
4200 register int tr, site;
4201 {
4202 register int i, att;
4203 int min_est_hv, est_min_site, est_card, est_hv;
4204
4205 prs ( min_site_estimated );
4206
4207 min_est_hv = INF;
4208 est_min_site = 0;
4209
4210 for_all_related_i_in_state ( LP_WAIT_END ) {
4211
4212    for_all_att      if ( site_rel_conf [ tr ] [ i ] [ att ] == 1 ){
4213
4214       est_hv = est_att_card [tr ] [ i ] [ att ]
4215              * est_att_card[tr ] [ i ] [ att ]
4216              * domain_width [ att ]
4217              / domain_range [ att ] ;
4218
4219       if ( est_hv < min_est_hv ) {
4220          min_est_hv = est_hv;
4221          est_min_site = i;
4222          }
4223       }
4224    }
4225 return ( est_min_site );
4226 }
4227
4228 /*-- ----------------------------------------------------------------
4229 *
4230 *      min_site_gen () --- generate min_site of tr
4231 *
4232 *          CALLED BY : pick_min_site,  min_site_estimate
4233 *-- ----------------------------------------------------------------
4234 */
4235 min_site_gen ( tr, state )
4236 register int tr, state;
4237 {
4238 register int i;
4239
4240 prs ( min_site_gen );
4241
4242 min_hv [ tr ] = INF;
4243
4244 for_all_related_i_in_state ( state ) {
4245
4246    hv_of_site ( token_of [ tr ] [ i ], tr , i );
4247    if ( hv [ tr ] [ i ] < min_hv [ tr ] ) {
4248       min_hv [ tr ] = hv [ tr ] [ i ];
4249       min_site [ tr ] = i;
4250       }
4251    }
4252
4253 broadcasting_att [ tr ] = local_min_att [ tr] [ min_site [ tr] ];
4254 }
4255
4256 /*-- ----------------------------------------------------------------
4257 *
4258 *      need_lock_in_LP () --- a new transaction comes in
4259 *                             CALLED by  LP_INI on RELEASE_BUS
4260 *-- ----------------------------------------------------------------
4261 *    check if the relations in the site need to do local semi-join
4262 *    or not. if yes, read locks are required in LP phase, otherwise
4263 *    wait till GBJ phase to lock. In LP phase, disk access is still
4264 *    needed for obtaining the H.V. All broadcast semi-join are buffered
4265 *    in the site until the time to broadcast is come
4266 *
4267 *    set the array not_lock_in_LP [ token ] 1  not locked, 0  locked
4268 *    Return the result of the token ( for saving another check )
4269 *    Condition   only one relation in the site
4270 *    though more than one rels is
4271 *    also possible, however, we ignore it for simplicity
4272 *
4273 *    if the only relation is also stored in other site, the condition of
4274 *    whether they are locked or not should also be checked
4275 *    Here we assume they are all locked in LP if it is stored in other
4276 *    sites to save execution time and programming effort
4277 */
4278
4279
4280 need_lock_in_LP ( token )
```



```
4281 int token;
4282 {
4283
4284 register int i, tr,  rel, site, att;
4285 int j, k, one_rel ;
4286
4287 site = site_of [ token ];
4288 tr   = tr_of   [ token ];
4289 prs ( LPlock?);ln;
4290
4291 j = 0;
4292 for_all_queried_rel_in_site {
4293    j ++;
4294    one_rel = rel;
4295    }
4296
4297 if ( j == 1 ) {
4298
4299    for_all_related_i      if ( i != site )
4300       if ( fap [ one_rel ] [ i ] == 1 ) return ( 1 );
4301
4302       /* this site will need a redundant get_local_lock in this case */
4303
4304    not_lock_in_LP [ token ] = 1;
4305    return ( 0 );
4306    }
4307
4308 else return ( 1 );
4309 }
4310
4311 /*-- ----------------------------------------------------------------
4312 *
4313 *      needed_lock(tr) --- figure out the needed locks of transaction tr
4314 *
4315 *-- ----------------------------------------------------------------
4316 */
4317
4318 needed_lock(tr)          /*  find out the needed locks of tr */
4319 register int tr;
4320 {
4321 register int rel;
4322
4323 for_all_rel
4324    if (queried_rel[tr][rel] == 1 ) {
4325       lock_state[tr][rel] = WANT_R;
4326       if (updated_rel[tr][rel] == 1 )
4327          lock_state[tr][rel] = WANT_RW;
4328          }
4329    else if (updated_rel[tr][rel] == 1)
4330          lock_state[tr][rel]=WANT_W;
4331    else lock_state [ tr ] [ rel ] = NULL;
4332 }
4333
4334 /*-- ----------------------------------------------------------------
4335 *
4336 *      new_trans() --- a new transaction comes in
4337 *
4338 *-- ----------------------------------------------------------------
4339 */
4340
4341 new_trans()      /* A new transaction come in  */
4342 {
4343 register int tr, rel,site,att,token;
4344 int i,k;
4345
4346 /*  ACTIONS */
4347
4348 site = RANDOM (SITE);
4349 if (num_of_tr_at [site] >= MAX_TR_OF_SITE ) {
4350    prs ( SITE FULL! CUSTOMER LEAVE !! );
4351    lost_tr[site]++;
4352    schedule ( NEW_TR , expntl (100000.0*mean_tr) , 0 );
4353    return (0);
4354    }
4355 else {
4356
4357    num_of_tr_at [site] ++;
4358    tr = get_tr_id();
4359    if (tr == 0 ) return (0);
4360    token = get_token_id();
4361    tr_of [token] = tr;
4362    site_of [token] = site;
4363    home_of [tr] = site;
4364    token_of [tr] [site] = token;
4365    query_gen(tr);
4366    get_related_site(tr);
4367    needed_lock(tr);
4368
4369    /* DATA BASE CHANGE */
4370
4371    time_of_tr [ tr ] = clock;
4372    total_tr ++;
4373    U_bus_tr [ tr ] = U_disk_tr [ tr ] = U_cpu_tr [ tr ] = 0;
4374    /*
```

```
4375      *  fp=fopen("ddblmn.data","a");
4376      *  fprintf(fp, "%7 Of NEWIR %d\n", clock, tr);
4377      *  fclose(fp);
4378      */
4379      tr_phase [ tr ] = PRE;
4380      th_state  [tr][site] = PRE;
4381      cc_new_tr(tr);
4382
4383      if ( class_of [ tr ]  == REMOTE ) {
4384         schedule_C ( NEED_BUS_TOKEN , PRE_TIME  , token );
4385         bro_set(token);
4386         last_sj[ tr]                = 0;
4387         first_rt_site [ tr ]        = 0;
4388         set_slowest_site_of [ tr ]= 0;
4389         num_in_lp [ tr ]            = 0;
4390         num_in_gsj [ tr ]           = 0;
4391         blocked_site [ tr ]         = 0;
4392         first_rt_site [ tr ]        = 0;
4393         min_site [ tr ]             = 0;
4394         }
4395      else {
4396         goto_state ( LP_PRE_LOCK);
4397         goto_phase ( LP );
4398         wake_up ( token );
4399         }
4400
4401      /******* initialize site relation card  ********/
4402
4403      for_all_rel for_all_site
4404         if ( queried_rel [ tr ] [ rel ] == 1
4405              && fap [ rel ] [ site ]     == 1 )
4406              rel_card [ tr] [ rel ] [ site ] = ini_rel_card [rel] ;
4407
4408      schedule ( NEW_TR , expntl (100000 0 * mean_tr) , 0 );
4409
4410      }
4411      /*
4412      * calculate effective relation width ( sum of queried atts )
4413      */
4414      for_all_rel if ( queried_rel [ tr ] [ rel ] == 1 ) {
4415         tr_rel_width [ tr] [ rel ] = 0;
4416         for_all_queried_att_in_rel
4417            tr_rel_width [ tr] [ rel ] += domain_width[ att ];
4418         if (target_njatt [ tr ] [ rel ] ==1 )
4419            tr_rel_width [ tr] [ rel ] += relation_nj_width [ rel ];
4420         }
4421
4422   /* set up effective relation width for each relation */
4423
4424   for_all_rel if ( queried_rel [ tr ] [ rel ] ==1 ) {
4425      tr_rel_width [ tr ] [ rel ] = 0;
4426      for_all_att if ( queried_att [ tr ] [ rel ] [ att ]  == 1 )
4427         tr_rel_width [ tr ] [ rel ] += domain_width [ att ];
4428      if ( target_njatt [ tr ] [ rel ] == 1)
4429        tr_rel_width [ tr ] [ rel ] += relation_nj_width [ rel ];
4430      }
4431
4432   return(token);
4433   }
4434
4435   /*-------------------------------------------------------------------------
4436   *
4437   *     orphan_test () ---  test if the related site is a single kid
4438   *
4439   *-------------------------------------------------------------------------
4440   * If home site has some queried relations, the condition is set to false
4441   * since semi-join process is needed in this case
4442   * when this is an orphan site, the site will perform join in the site
4443   * then sent the results directly, no PP phase is needed
4444   */
4445
4446   orphan_test ( token )
4447   int token;
4448   {
4449   int j, site, tr;
4450
4451   tr = tr_of [ token ];
4452
4453   /* pre ( orphan_test ); */
4454
4455   j = 0;
4456   for_all_related_site j ++;
4457
4458   if ( j == 1 ) orphan [ token ] = 1;
4459   else          orphan [ token ] = 0;
4460   }
4461
4462   /*-------------------------------------------------------------------------
4463   *
4464   *     pick_min_site () --- pick the  min_site in GSJ
4465   *                          CALLED by GSJ_ALL_FINISH  RELEASE_BUS
4466   *
4467   *              pick brothers one by one until the min_site is picked
4468   *              check if need lock and get the lock, if not, restart
```



```
4469   *-------------------------------------------------------------------------
4470   */
4471   pick_min_site ( tr, site, state )
4472   register int tr, site;
4473   int state;
4474   {
4475   register int i, size;
4476   int k;
4477   int broadcast_site;
4478   int alive [ SITE + 1];
4479
4480   pre ( pick_min_site );
4481
4482   min_site_gen ( tr, state );          /*  start the election process */
4483
4484   if ( state == LP_WAIT_END ){
4485
4486      /*
4487      *  First min site does not need election
4488      *  count the number of alive sites and set up alive vector
4489      */
4490
4491      size = 0;
4492      for_all_related_i_in_state ( state ) {
4493         size ++;
4494         alive [ i ] = 1;
4495         }
4496
4497      num_of_broad [ tr ] = 1;
4498      broadcast_site = min_site_estimate ( tr, site );
4499      goto mark3;                      /* skip the election steps */
4500      }
4501
4502   num_of_broad[ tr ] = 0;
4503
4504   /**********/
4505     mark1 :
4506   /**********/
4507
4508   size = 0;              /*  start the election process */
4509
4510   for_all_related_i_in_state ( state ) {
4511                          /* count alive sites for min_site election */
4512     size ++;
4513     alive [ i ] = 1;
4514     }
4515
4516   else alive [ i ] = 0;
4517
4518   /*********/
4519   mark2
4520   /*********/
4521
4522   if ( size == 0 ) return ( 0 );
4523
4524   num_of_broad[ tr ] ++;
4525
4526   k = RANDOM ( size );            /* this is a broadcast */
4527
4528   /* find out who is the site k to broadcast its bv */
4529
4530   for_all_related_i_in_state ( state ) if ( alive [i] == 1 )  {
4531      k --;
4532      if ( k == 0 ) break;
4533      }
4534
4535   broadcast_site = i;
4536
4537   /**********/
4538     mark3
4539   /**********/
4540
4541   if ( broadcast_site != min_site [ tr ]){
4542                                  /* non min site bro bv */
4543      if ( bv [ tr][ broadcast_site] == min_bv [ tr ]) {
4544
4545         /*
4546         * the broadcast value equal to the min - bv, change the
4547         * min_site election
4548         */
4549
4550         min_site [ tr ] = i;
4551         broadcasting_att[tr] = local_min_att [tr][broadcast_site];
4552         goto mark4;  /* set the site as min - site, skip other steps */
4553         }
4554
4555      for_all_related_i_in_state ( state )
4556
4557         if ( alive [ i ] == 1 ){
4558
4559            if ( bv [ tr][ i] >= bv [ tr ][broadcast_site]) {
4560
4561               /* shut off sites with worse bv */
4562               alive [i];
```

```
4663              alive [ i ] = 0 ;
4664              go_sleep ( tr, i , GSJ_WAIT_SJ );
4665              }
4666           }
4667
4668     goto mark2; /* Broadcast hv is not the min hv, repeat the election */
4669     }
4670
4671  /* Had picked up a the min_site */
4672
4673  /**********/
4674     mark4
4675  /**********/
4676
4677  if( not_lock_in_LP [ token_of [ tr] [ broadcast_site ] ] == 1 )
4678
4679     if ( get_local_lock(token_of[tr][broadcast_site], tr, broadcast_site)
4680
4681        == REJECT ){
4682
4683        go_sleep (  tr, broadcast_site , GSJ_WAIT_LOCK ),
4684        alive [ broadcast_site ] = 0;
4685        blocked_site [ tr ] ++,
4686        site --,
4687        goto mark1, /* The elected site didn't get the lock, re-elect */
4688        }
4689
4690  for_all_related_i_in_state ( state )
4691
4692     go_sleep ( tr, i , GSJ_WAIT_SJ );
4693  th_state [ tr] [ broadcast_site] = GSJ_INI,
4694
4695  return ( broadcast_site );
4696  }
4697
4698  /*------ ------------------------------ ---------- --------- ----------
4699  *
4700  *      query_gen(tr) ---- query generation for a new transaction tr
4701  *
4702  *-------------------------------------------------------------------
4703  */
4704
4705  query_gen(tr)              /*  generate a query */
4706  register int tr;
4707  {
4708  register int i,j,k,rel,site,att,
4709  int a,b,c,
4710  int num_of_update,
4711
4712
4713  /*  initialization  */
4714
4715     num_of_update = writing_rel [ tr ] = 0,
4716
4717     for_all_rel {
4718
4719        queried_rel    [tr] [rel] =
4720        target_rel     [tr] [rel] =
4721        updated_rel    [tr] [rel] =
4722        target_njatt   [tr] [rel] =
4723        updated_njatt [tr] [rel] = 0,
4724
4725        for_all_att {
4726
4727           queried_att   [tr] [rel] [att] =
4728           target_jatt   [tr] [rel] [att] =
4729           updated_jatt [tr] [rel] [att] = 0,
4730
4731           }
4732        }
4733
4734  /*
4735  *  set up the query
4736  *  number of updated relation in each query is assume at most one
4737  */
4738
4739     for_all_rel {
4740
4741        if ( RANDOM(10) > QUERY_REL_HIT ) { /* queried attribute */
4742
4743           queried_rel [tr] [rel] =1;
4744
4745           if ( RANDOM(10) > TARGET_REL_HIT ) { /* target rel */
4746
4747              target_rel [tr] [rel] =1,
4748
4749              } /* target rel */
4750
4751           /* pick queried, target j att*/
4752
4753           for_all_att   {
4754              if ( relation_conf [rel] [att] == 1) {
4755
4756                 if (  RANDOM(10) > QUERY_ATT_HIT )
```



```
4757                    queried_att [tr] [rel] [att] =1,
4758
4759                 if (  target_rel [tr] [rel] ==1
4760                    && RANDOM(10) > TARGET_ATT_HIT
4761                    ){
4762
4763                    target_jatt [tr] [rel] [att] =1,
4764
4765
4766                    } /* target att */
4767                 } /* rel_conf */
4768              } /* for_all_att */
4769
4770           /* Assume entire rel is targeted if it's a target relation */
4771
4772           if (   target_rel [tr] [rel] ==1
4773               && RANDOM(10) > TARGET_REL_HIT
4774              ){
4775
4776              target_njatt [tr] [rel] =1,
4777
4778              }
4779           } /* if queried relation */
4780        } /* loop R */
4781
4782     /*
4783     *  Test for the existence of target_rel and queried_rel
4784     *  At least one should be exist
4785     */
4786
4787     K=0;
4788     for_all_rel {
4789        if (target_rel [tr] [rel] ==1) { K++,   break;  }
4790        }
4791
4792     if (K==0) {
4793        K = 1;
4794        rel=RANDOM(R);
4795        target_rel [tr] [rel] =queried_rel [tr] [rel] =1;
4796        }
4797
4798  /* Generate Update realtion and attributes */
4799     if ( RANDOM(10) > UPDATE_QUERY_HIT ) {
4800        /* Pick up one target relation to update */
4801        /* Currently, K is the number of target relations */
4802        int picked;
4803
4804        picked = RANDOM (K);
4805        for_all_rel {
4806           if ( target_rel [tr] [rel] == 1 ) {
4807
4808              if ( picked  == 1 ) {
4809                 updated_rel [tr] [rel] = 1;
4810                 writing_rel [tr]       = rel;
4811
4812                 for_all_att { /* pick updated_atts */
4813
4814                    if ( RANDOM(10) > UPDATE_ATT_HIT)
4815                       updated_jatt [tr] [rel] [att] = 1;
4816                    }
4817                 /*  Test for the existence of update_att */
4818                 K=0;
4819                 for_all_att
4820                       if (target_jatt [tr] [rel] [att] ==1)
4821                          { K++;   break;  }
4822                 if ( K==0 || RANDOM(10) > UPDATE_QUERY_HIT )
4823                    updated_njatt [tr] [rel] =1;
4824                 break;
4825                 }
4826              else picked -- ;
4827              }
4828           }
4829
4830     for_all_rel {
4831
4832        /*
4833        *  Test for the existence of target_att
4834        *  At least one should be exist
4835        */
4836
4837        if ( target_rel [tr] [rel] == 1 ) {
4838
4839           i=0;
4840           for_all_att
4841              if ( target_jatt [tr] [rel] [att] == 1 ) { i++;  break; }
4842
4843           if (i==0) target_njatt [tr] [rel] = 1;
4844
4845           }
4846
4847        /*
4848        *  Test for the existence of queried_att
4849        *  At least one should be exist
```

```
4751        */
4752
4753        if ( queried_rel [tr] [rel] == 1 ) {
4754
4755           i=0;
4756           for_all_att
4757              if (queried_att [tr] [rel] [att] ==1) { i++; break; }
4758
4759           if (i==0) {
4760              j=RANDOM(R);
4761              k=1;
4762              while ( j > 0 ) {
4763                 att = ( ( k -1 ) % JA ) +1;
4764                 if ( relation_conf [rel] [ att ] == 1 ) j--;
4765                 k++;
4766                 }
4767              queried_att [tr] [rel] [att] =1;
4768              }
4769           }
4770        }
4771
4772     /*
4773     *  Test for the existence of joining att
4774     *  Either zero or more than one should be exist
4775     */
4776
4777     for_all_att {
4778
4779        i=0;
4780        for_all_rel if ( queried_att [tr] [rel] [att] == 1 ) i++;
4781
4782        if ( i == 1 ) {
4783           register int i;
4784           i=0;
4785           while ( i == 0) {
4786              i = RANDOM(R);
4787              if (relation_conf [i] [att] == 0) i=0;
4788              }
4789           queried_att [tr] [i] [att] = queried_rel [tr] [i] = 1;
4790           }
4791        }
4792     }
4793
4794  /* ------------------------------------------------------------------------
4795  *
4796  *      resolve_local_dl () ---  Resolving possible local deadlock
4797  *
4798  *             CALLED BY   LP_WAIT_SYN on finish
4799  *
4800  * if the estimated slowest site is blocked by somebody else, the
4801  * local-pg  held by the TH may cause deadlock problems
4802  * Changing the est_slowest site selection to prevent it
4803  *
4804  * Check if the estimated slowest site is blocked, if YES then do it
4805  * Assign the est_slowest_site the  one of the site in state LP_DISK
4806  * if no such site, then choose a site in LP_WAIT_END, wake it up
4807  * est_slowest_site [ tr ] should be changed
4808  */
4809
4810  resolve_local_dl ( token )
4811  int token;
4812  {
4813  register int KK, tr;
4814  int i, j, rel, site, att;
4815
4816  site = site_of [ token ];
4817  tr   = tr_of   [ token ];
4818  pre ( resolve_local_dl );
4819
4820  KK =  est_slowest_site_of [ tr ] ;
4821
4822  if (                          KK   >   0
4823     &&                         KK   <   INF
4824     &&  th_state [ tr ] [ KK ]   ==   LP_WAIT_END
4825     ) {
4826
4827      /*
4828      * SYN had been broadcast, do locks alone
4829      * unlock unneeded locks. ( read_only )
4830      */
4831
4832      if ( cc_lock_get_bus ( tr, site ) == FAIL ) {
4833         pre ( FAIL in resolve_local_dl and cc_lock_get_bus );
4834         return ( FAIL );
4835         }
4836
4837      cc_lock ( tr, site );
4838      if ( r_unlock_time == ANYTIME ) {
4839         cc_unlock ( tr, site );
4840         resume_by (  tr, site );
4841         }
4842
4843      return;
4844      }
```



```
4845  if (                          KK   >   0
4846     &&                         KK   <   INF
4847     &&  th_state [ tr ] [ KK ]   ==   LP_WAIT_LOCK
4848     ) {
4849
4850     j = 0;
4851
4852     for_all_related_i_in_state ( LP_DISK ) {
4853        j = i;
4854        break;
4855        }
4856
4857     if ( j > 0 ) {
4858        est_slowest_site_of [ tr ] = j;
4859        return;
4860        }
4861
4862     for_all_related_i_in_state ( LP_WAIT_END ) break;
4863     est_slowest_site_of [ tr ] = i;
4864     th_state [ tr ] [ i ] = LP_SYN_EST_BRO;
4865     num_in_lp [ tr ] ++;
4866     finish ( token_of [ tr ] [ i ] );
4867     }
4868
4869  return;
4870  }
4871  /*-------------------------------------------------------------------------
4872  *
4873  *      resume_by () --- resume all blocked processes by the token
4874  *
4875  *-------------------------------------------------------------------------
4876  */
4877
4878  resume_by ( tr, site )
4879  register int  tr, site;
4880  {
4881  register int i;
4882  pre ( resume_by );
4883
4884  loop ( i, 1, TOKEN )  {
4885
4886     int trofi;
4887     int siteofi;
4888
4889     trofi   = tr_of   [ i ] ;
4890     siteofi = site_of [ i ] ;
4891
4892     if (  trofi != tr
4893        && (                pg [ tr ] [ trofi ]   == 1
4894           ||     new_relative [ trofi ] [ tr ]   == -1
4895           ||     new_relative [ trofi ] [ tr ]   == UNUSED
4896           )
4897        && (      th_state [ trofi ] [ siteofi ] == LP_WAIT_LOCK
4898           ||     th_state [ trofi ] [ siteofi ] == GSJ_WAIT_LOCK
4899           ||     th_state [ trofi ] [ siteofi ] == PP_WAIT_LOCK
4900           )
4901        && blocked_by [ i ] != ( TR + 1 )
4902        && blocked_by [ i ] != 0
4903        ) {
4904
4905        blocked_by [ i  ] = 0;
4906        blocked_site [ trofi ] - ;
4907        delayed_wake_up ( i );
4908        }
4909     }
4910  }
4911
4912  /* ------------------------------------------------------------------------
4913  *
4914  *      return_token_id()
4915  *
4916  *-------------------------------------------------------------------------
4917  */
4918
4919  return_token_id(token)          /* return a token_id to token_id_pool */
4920                                  /* (tr_of and site_of) */
4921  int token;
4922  {
4923
4924     if(token > TOKEN) {
4925        printf(" TOKEN NUMBER TOO BIG \n ");
4926        return (ERROR);
4927        }
4928     tr_of    [token] = 0;
4929     site_of  [token] = new_token;
4930     new_token        = token;
4931     }
4932
4933  /*-------------------------------------------------------------------------
4934  *
4935  *      return_tr_id(tr)
4936  *
4937  *-------------------------------------------------------------------------
4938  */
```

```
4939
4940  return_tr_id(tr)                  /* return a tr_id to tr_id-pool */
4941                                    /* (home_of and link_to_next_site_tr) */
4942  int tr;
4943  {
4944
4945     if(tr > TR) {
4946        printf(" TR NUMBER TOO BIG \n ");
4947        return (ERROR);
4948        }
4949     link_to_next_tr [tr] = new_tr;
4950     home_of             [tr] = 0;
4951     new_tr                   = tr;
4952     }
4953
4954  /*-----------------------------------------------------------------------
4955  *
4956  *      rw_conflict () --- check r-w conflict between tri <- trj for all
4957  *                         locks requested by transaction tri from site 'site'
4958  *      ww_conflict () --- check w-w conflict between tri <-> trj for
4959  *                         THE locks requested by transaction tri
4960  *
4961  *              CALLED BY . consistency ()
4962  * ----------------------------------------------------------------------
4963  * condition of r-w conflict : trj needs a read lock which is being or is
4964  *                             going to be written by trj
4965  * condition of w-r conflict : trj needs a write lock which is being or
4966  *                             is going to be read by trj
4967  * condition of w-w conflict : trj needs THE write lock on 'rel' which is
4968  *                             being, is going to be, or had been written
4969  *                             by trj
4970  */
4971  rw_conflict ( tri, trj, site )
4972  register int tri,  site,
4973  int trj;
4974  {
4975  register int rel, tr;
4976  /* prs ( rw_conflict ); */
4977
4978  tr = tri;
4979
4980  for_all_rel     if ( the_rel_in_site_is_queried ) {
4981
4982        if (  (   lock_state  [ tri ] [ rel ] == WANT_R
4983               || lock_state  [ tri ] [ rel ] == WANT_RW )
4984
4985           && (   lock_state  [ trj ] [ rel ] == WANT_RW
4986               || lock_state  [ trj ] [ rel ] == READING_WANT_W
4987               || lock_state  [ trj ] [ rel ] == READING_WRITING
4988               || lock_state  [ trj ] [ rel ] == WANT_W
4989               || lock_state  [ trj ] [ rel ] == WRITING ))
4990
4991           return ( 1 );
4992
4993        if (       updated_rel [ tri ] [ rel ] == 1
4994
4995           && (   lock_state  [ tri ] [ rel ] == WANT_W
4996               || lock_state  [ tri ] [ rel ] == READING_WANT_W )
4997
4998           && (   lock_state  [ trj ] [ rel ] == WANT_R
4999               || lock_state  [ trj ] [ rel ] == READING
5000               || lock_state  [ trj ] [ rel ] == WANT_RW
5001               || lock_state  [ trj ] [ rel ] == READING_WANT_W
5002               || lock_state  [ trj ] [ rel ] == READING_WRITING ))
5003
5004           return ( 1 );
5005        }
5006
5007  return ( 0 );
5008  }
5009
5010  ww_conflict ( tri, trj )
5011  int tri, trj;
5012  {
5013  register int rel;
5014  /* prs ( ww_conflict ); */
5015
5016  rel = writing_rel [ tri ];
5017
5018  if (  (   lock_state  [ tri ] [ rel ] == WANT_W
5019         || lock_state  [ tri ] [ rel ] == WANT_RW )
5020
5021     && (   lock_state  [ trj ] [ rel ] == WANT_RW
5022         || lock_state  [ trj ] [ rel ] == READING_WANT_W
5023         || lock_state  [ trj ] [ rel ] == READING_WRITING
5024         || lock_state  [ trj ] [ rel ] == HAD_RW
5025         || lock_state  [ trj ] [ rel ] == WANT_W
5026         || lock_state  [ trj ] [ rel ] == WRITING
5027         || lock_state  [ trj ] [ rel ] == WRITTEN ))
5028
5029     return ( 1 );
5030
5031  return ( 0 );
5032  }
```



```
5033  /*-----------------------------------------------------------------------
5034  *
5035  *      size_of_rt_bro () ---  decide which relation  to be broadcast
5036  *                             figure out the # of packets to be broadcast
5037  *                             return the # of Packets
5038  *
5039  *-----------------------------------------------------------------------
5040  */
5041  size_of_rt_bro ( tr, site )
5042  register int tr, site;
5043  {
5044  register int  rel, att;
5045  int tuple_width, rt_size;
5046
5047  prs ( size_of_rt_bro );
5048  rt_size = 0;
5049
5050  for_all_rel if ( the_rel_in_site_is_queried
5051                   && bro_rel [ tr ] [ rel ] == WAIT_BRO ) {
5052
5053      tuple_width = 0;
5054
5055      for_all_att if ( queried_att [ tr] [ rel ] [ att ] == 1 )
5056         tuple_width += domain_width [ att ];
5057
5058      if ( target_pjatt [tr][rel] == 1 )
5059         tuple_width += relation_pj_width [ rel ];
5060
5061      rt_size += rel_card [ tr ] [ rel ] [ site ] * tuple_width;
5062      }
5063
5064  return ( rt_size/DATA_SIZE );
5065  }
5066  /*-----------------------------------------------------------------------
5067  *
5068  *     solve_pg conflict () - withdraw locks of a trans due to pg conflict
5069  *
5070  *              CALLED BY   cc_lock_get_bus
5071  *-----------------------------------------------------------------------
5072  * all the sites of the transaction are reset to the state WAIT_LOCK if
5073  * they are involved in this lock bounce
5074  * all the status and the database are reset here
5075  */
5076
5077  solve_pg_conflict ( tr, site )
5078  int tr, site;
5079  {
5080  int i, j, k, rel, event;
5081  double  remain_time;
5082  prs ( solve_pg_conflict );
5083
5084  switch ( tr_phase [ tr ] ) {
5085
5086     case LP
5087          K =  get_slowest_site_of [ tr ] ;
5088          if (  K > 0
5089             && K < INF
5090             && tb_state [ tr ] [ K ] == LP_WAIT_END
5091             ) {
5092
5093             /*
5094             * locking broadcast has been made
5095             * This TR should go ahead alone, other sites will not
5096             * have to follow it
5097             */
5098
5099             if ( not_lock_in_LP [ token_of [ tr ] [ site ] ] == 0 ){
5100                                       /* release all resources in hand */
5101                pqueer  [ site ] = NULL;
5102                local_resume ( site );
5103                blocked_site [ tr ] --;             /* add the site back */
5104                num_in_lp [ tr ] --;                /* add the site back */
5105                tb_state [ tr ] [ site ] = LP_WAIT_LOCK;
5106                loop ( I, 1, TR )
5107                   new_relative [ tr ] [ I ] = pg [ tr ] [ I ];
5108
5109                /*
5110                * should be careful, if other sites who have using
5111                * some locks based on tb obsolete new_relative may not
5112                * know what had been changed in new_relative
5113                * Therefore, the obtained lock may have no way to be
5114                * withdrawn. Should have some way to control this
5115                */
5116
```

```
5116
5117                schedule ( WAKE_UP, 0.0, token_of [ tr ] [ site ] ),
5118                                        /* redo lock request */
5119              }
5120           }
5121
5122        else {
5123
5124           for_all_related_i {
5125
5126              if (    not_lock_in_LP ( token_of [ tr ] [ i ] ) == 0
5127                 &&   tb_state [ tr ] [ i ]     !=    LP_WAIT_LOCK
5128                 ){
5129
5130                 cancel ( token_of [ tr ] [ i ], event, remain_time ),
5131
5132                 if (     event == RELEASE_BUS_TOKEN
5133                       || event == RELEASE_DISK
5134                    )
5135                    schedule ( event, 0.0, token_of [ tr ] [ i ] ),
5136
5137                 pguser  [ i ] = NULL,
5138                 local_resume ( i ),
5139                 blocked_site [ tr ] ++,        /* add the site back */
5140
5141                 if ( tb_state [ tr ] [ i ] > LP_WAIT_SYN )
5142                    num_in_lp [ tr ] ++,        /* add the site back */
5143
5144                 tb_state [ tr ] [ i ] = LP_WAIT_LOCK,
5145                 }
5146
5147              loop ( I, 1, TR )
5148                 new_relative [ tr ] [ I ] = pg [ tr ] [ I ] ,
5149
5150              est_slowest_site_of [ tr ] = 0,
5151
5152              /*
5153              * To enforce the algo return the slowest site estimation
5154              * The algo is changed to accommodate the pg-conflict
5155              * the other statistic data such as rel_card,
5156              * est_att_card, est_rel_card, lp_time, etc are to be
5157              * recalculate automatically
5158              *
5159              * local locks are ignored ( will be overwritten )
5160              */
5161              }
5162           }
5163        break,
5164
5165     case GSJ
5166
5167        tb_state [ tr ] [ site ] = GSJ_WAIT_LOCK,
5168        blocked_site [ tr ] ++,
5169        pguser  [ i ] = NULL,
5170        local_resume ( i ),
5171
5172        /*
5173        *  dbsize haven't been changed in this stage
5174        *  num_in_gsj has nothing to do here
5175        */
5176        break,
5177
5178     case PP
5179
5180        tb_state [ tr ] [ site ] = PP_WAIT_LOCK,
5181        pguser   [ i ]           = NULL,
5182        local_resume ( i ),
5183        break,
5184
5185     }
5186  return,
5187  } ;
5188
5189  /*-------------------------------------------------------------------
5190  *
5191  *     syn_steps () ---    Count the number of broadcasts to synchronized
5192  *                         in LP and GSJ phase
5193  *                         Identify the slowest site
5194  *                         Figure out the slowest site first, then randomly
5195  *                         pick sites until the slowest site is picked
5196  *
5197  *              CALLED BY   LP_FINISH on RELEASE_BUS when all finished
5198  *-------------------------------------------------------------------
5199  */
5200  syn_steps ( tr, state )
5201  register int tr,
5202  int state,
5203  {
5204  register int i, size,
5205  int k, slowest_time,
5206  int alive [ SITE + 1 ],
5207  int broadcast_site,
5208
5209  prs ( syn_steps ),
```



```
5210
5211  size = 0,
5212  for_all_related_i_in_state ( state ) {
5213     size ++,
5214     alive [ i ] = 1,
5215     }
5216
5217  else alive [ i ] = 0,
5218
5219  num_of_broad [ tr ]= 0,
5220
5221  /*
5222  * find out who is the slowest site , then see how many steps to
5223  * identify the slowest site
5224  */
5225
5226  switch ( state ) {
5227
5228          case   LP_WAIT_END
5229
5230                 maximum ( lptime, LP_WAIT_END ),
5231                 slowest_time = lptime[ tr ] [ maximum_site ],
5232                 break,
5233
5234          case   GSJ_WAIT_SYN_END
5235
5236                 maximum ( sjtime, GSJ_WAIT_SYN_END ),
5237                 slowest_time = sjtime[ tr ] [ maximum_site ],
5238                 break,
5239
5240          case   GSJ_WAIT_END
5241
5242                 maximum ( sjtime, GSJ_WAIT_END ),
5243                 slowest_time = sjtime[ tr ] [ maximum_site ],
5244                 break,
5245        }
5246  /* the slowest site is the maximum_site now */
5247
5248  num_of_broad [ tr ]= 0,
5249
5250  /*********/
5251     mark :
5252  /*********/
5253
5254  if ( size == 0 ) return ( 0 ),
5255  num_of_broad [ tr ] ++,
5256  k = RANDOM ( size ), /* the kth site get the bus to broad unfinished */
5257
5258  /* pick the  k - th site */
5259
5260  for_all_related_i_in_state ( state )  {
5261     if ( alive [ i ] == 1 ) k --,
5262     if ( k == 0 ) break,
5263     }                          /* the site i get the bus */
5264
5265  broadcast_site = i,
5266  alive [ broadcast_site ] = 0,
5267
5268  if ( broadcast_site != maximum_site ){
5269                             /* find out who haven't finished */
5270     for_all_related_i_in_state ( state )
5271
5272        if (     alive [ i ]  ==  1
5273           &&              i   !=  maximum_site ) {
5274
5275           switch ( state ) {
5276
5277                    case   LP_WAIT_END
5278
5279                           if ( lptime [ tr][ i] <= slowest_time ) {
5280                              size --,
5281                              alive[i]= 0,
5282                              }
5283                           break,
5284
5285                    case   GSJ_WAIT_SYN_END
5286                    case   GSJ_WAIT_END
5287
5288                           if ( sjtime [ tr][ i] <= slowest_time ) {
5289                              size --,
5290                              alive[i]= 0,
5291                              }
5292                           break,
5293                 }
5294           }
5295     goto mark,
5296     }
5297
5298  return ( num_of_broad [ tr] ),
5299  }
5300
5301  /*-------------------------------------------------------------------
5302  *
5303  *     terminate_site () ---  terminate a site
```

```
*
*                       unlock
*                       change BTT
*                       wake up blocked processes
*                       delete the transaction, return resources  back
*-------------------------------------------------------------------------
*/

terminate_site ( token, tr, site )
int token;
register int tr, site;
{
register int j, rel, att;

/* pre ( terminate_site ),*/
pre# ( ONE SITE IS TERMINATED !!),

th_state      [ tr ] [ site ] = NULL;
related_site  [ tr ] [ site ] = NULL;
return_token_id ( token );
num_of_tr_at [site] --  ;
token_of      [ tr ] [ site ] = 0;
th_state      [ tr ] [ site ] = NULL;
local_min_att [ tr ] [ site ] = 0;
bv            [ tr ] [ site ] = INF;

for_all_att
   bro_site_att  [ tr    ] [ site  ] [ att   ] = NULL;
}

/*-------------------------------------------------------------------------
*
*     terminate_tr () ---  terminate a transaction
*
*                       unlock
*                       change BTT
*                       wake up blocked processes
*                       delete the transaction, return resource  back
*-------------------------------------------------------------------------
*/

terminate_tr ( tr )
int tr;
{
int i, j, k, rel, site, att, token;
double  used_time,
double  ave,
/* pre ( terminate_tr !!!!!!!!!!!!),*/
pre# ( ONE TRANSACTION IS TERMINATED !!!!!!!!!!),ln,

num_of_tr_at [ home_of [ tr ] ] --,

for_all_site
   th_state    [ tr ][site] = NULL,
tr_phase       [ tr ]       = NULL,
last_mj        [ tr ]       = 0,
first_rt_site [ tr ]        = 0,
set_slowest_site_of [ tr ] = 0,
num_in_lp      [ tr ]       = 0,
num_in_gmj     [ tr ]       = 0,
blocked_site   [ tr ]       = 0,
first_rt_site [ tr ]        = 0,
min_site       [ tr ]       = 0,

used_time     =  ( clock - time_of_tr [ tr ]),
total_tr_time += used_time,
total_U_bue   += U_bue_tr [ tr ]/used_time,
total_U_disk += U_disk_tr [ tr ]/used_time,
total_U_cpu   += U_cpu_tr [ tr ]/used_time,


finished_tr ++,
ave = total_tr_time / finished_tr ;
iteration_error =  fabs ( ave - ave_tr_time ) / ave ;

if (     finished_tr      >=    SIMULATED_TR
   &&    iteration_error  <     allowed_error
   )
   #schedule ( STOP, 0 0, 0 );

#ifdef TRACE
fp = fopen ("ddblmn_data","a"),
fprintf(fp,"%4 2f %3 3f %3 3f %3 3f %4 4f \n ",
    total_tr_time/(finished_tr*MEG),
    total_U_bue  /finished_tr,
    total_U_disk /(finished_tr * SITE ) ,
    total_U_cpu  /(finished_tr * SITE ),
    lost_customer * MEG/clock ),
fclose (fp),
#endif

ave_tr_time = ave,
```



```
U_bue_tr  [ tr ] =
U_disk_tr [ tr ] =
U_cpu_tr  [ tr ] = 0,
home_of   [ tr ] = 0,

for_all_site
   related_site [ tr ][ site ] = 0,

return_tr_id ( tr );

for_all_rel  {
   lock_state     [ tr  ] [ rel ] = NULL;
   queried_rel    [ tr  ] [ rel ] = NULL;
   updated_rel    [ tr  ] [ rel ] = NULL;
   target_rel     [ tr  ] [ rel ] = NULL;
   updated_pjatt  [ tr  ] [ rel ] = NULL;
   target_pjatt   [ tr  ] [ rel ] = NULL;
   bro_rel        [ tr  ] [ rel ] = NULL;

   for_all_att {
      queried_att  [ tr ] [ rel ] [ att ] = 0,
      target_jatt  [ tr ] [ rel ] [ att ] = 0,
      updated_jatt [ tr ] [ rel ] [ att ] = 0,
      bro_component[ tr ] [ att ] [ rel ] = NULL,
      }
   }

for_all_site

   for_all_att  {
      site_rel_conf     [tr] [site] [att] = 0;
      bro_site_att      [tr] [site] [att] = NULL,
      for_all_rel
        component_set [tr] [site] [att] [rel]  = 0,
      }

loop ( j, 1, TR ) {

   new_relative  [ tr ] [ j  ] = 0,
   new_relative  [ j  ] [ tr ] = 0,
   pg [ tr ] [ j  ] =
   pg [ j  ] [ tr ] = 0;
   }
}

/*-------------------------------------------------------------------------
*
*     timestamp () --- get the system time
*
* ------------------------------------------------------------------------
*/
timestamp ()
{
struct rusage buffer,
int i,
/* pre ( timestamp ), */

getrusage ( 0, &buffer ),
i = buffer ru_utime tv_sec,
return ( i ),
}
/*-------------------------------------------------------------------------
*
*     transitive  () --- create transitive closure for a new precedence
*                        relation  Not using matrix multiplication
*                        because some operations is wasted
*                        This strategy should be reevaluated if conflicts
*                        among transactions are too much
*
*     downward_pg () --- add edges of all successors to the new father
*                        and all his ancestors the procedure is called
*                        recursively
*
*             CALLED BY    cc_min
* ------------------------------------------------------------------------
*/
#define precede(i,j) { \
                        if (  pg [ i ] [ j ] == -1 \
                           || pg [ j ] [ i ] == 1   \
                           ) \
                           preERROR ( in pg transitive downward precede ), \
                           pg [i][j] =  1 ,  \
                           pg [j][i] = -1 ,\
                        }
#define succeed(i,j) { \
                        if (  pg [ i ] [ j ] ==  1 \
                           || pg [ j ] [ i ] == -1   \
                           ) \
                           preERROR ( in pg transitive downward succeed ), \
                           pg [i][j] =  -1 , \
                           pg [j][i] =   2 ,\
                        }
#define for_all_predecessor_of(self) loop ( ancestor, 1, TR ) \
        if ( ancestor != self && pg  [ self ][ ancestor ] == -1 )
```

```
#define for_all_successor_of(self) loop ( descendant, 1, TR ) \
        if ( descendant != self && pg  [ self ][ descendant ] ==  1 )

transitive ( tri, trj )
register int tri, trj;
{
int ancestor, descendant, i;
pre ( transitive closure );

for_all_predecessor_of ( tri )
   precede ( ancestor, trj );

for_all_successor_of ( trj ) {

   for_all_predecessor_of ( tri )

      precede ( ancestor, descendant );
      precede ( tri, descendant );
   }
}

/* ther is no need of recursive down to the subtree of trj
* since all the descendant in the subtree had been recorded in pg [trj]
* it's good enough to make a link all all the ancestor of tri to
* all the descendant of trj
*/

/*-----------------------------------------------------------------------
*
*    valid_check ()  ---  check the validity of holding locks
*                         this is the implementation of procedure
*
*                              precedencecheck
*
*-----------------------------------------------------------------------
* if invalid return 0 and change to false_state  reset local BTT
*/

valid_check ( tr, site,  false_state )
register int tr, site;
int  false_state;
{
int i;

pre ( valid? );

if ( lock_validity [ tr ] [ site ] == VALID ||
     lock_validity [ tr ] [ site ] == FIRMED
   )
   return ( 1 );

go_sleep ( tr, site, false_state );
pg[tr [ site ] = NULL;
local_resume ( site );
blocked_site [ tr ] --;

return ( 0 );
}


/*************************************************************************
*
*               Modified empl simulation subsystem
*
*************************************************************************/

/*-----------------------------------------------------------------------*/
/*      main - main program                                              */
/*-----------------------------------------------------------------------*/

main (argc, argv)
int   argc;
char *argv[];

{
      char *str;
      int   cmd;

      /* DDD */
      cmd = 0;

      while (--argc > 0 && (++argv)[0] == '-')
            for (str = argv[0] + 1; *str != '\0'; str++)
                  switch (*str) {

                        case 't':
                              inputi[++cmd] = 't';
                              empl_tr = 1;
                              break;

                        case 'd':
                              inputi[++cmd] = 'd';
```



```
                              test = 16;
                              break;

                        default:
                              sscanf (argv[cmd + 2], "%d", &inputi[++cmd]);
                  }
      /* EEE */
      sim ();

      /* DDD */
      if (test > 0)
            check ();
      /* EEE */

      exit (0);
}


/*************************************************************/
/*                                                           */
/*      user interface (in abstract data object empl)        */
/*                                                           */
/*************************************************************/


/* DDD */
/*-----------------------------------------------------------------------*/
/*      visitcnt - determine number of completions at node               */
/*-----------------------------------------------------------------------*/

visitcnt (node)
INT node;
{
      return (RELCNT (node + 2));
}
/* EEE */

/* DDD */
/*-----------------------------------------------------------------------*/
/*      check - print complete data dump                                 */
/*-----------------------------------------------------------------------*/

check () {
      int   f, d, q;

      printf ("\n\n");
      printf ("****** DATA STRUCTURE DUMP ******                ");
      printf ("empl_event=%d  empl_tr=%d    test=%d  evl=%d\n%43efdb\
=%d   avl=%d  dec=%d  empl_token=%d \n\n",
                  empl_event, empl_tr, test, evl, "", fdb, avl, dec,
                  empl_token);
      printf ("      PRI      RTIME                    \
Event Token QLINKL  empl_event loc \n");
      printf ("  TQSUM     PREEMPTC              NAME   DCHAIN  \
RESQ    fac descr loc \n");
      printf ("  TLAST      QEXCNT                QLEN   NOFBusy NOFSERV\n");
      printf ("    BSTART     BSUM                  RELCNT  USRPRI \
USRTKN   dev descr loc \n");
      printf ("      PRI      RTIME                    Event   Token \
QLINKL    queue el  loc \n\n");

      for (f = evl; f != NIL; f = QLINK (f))
            printf ("              %10f %10f %20d %4d %4d   evf=%4d\n",
                  PRI (f), RTIME (f), Event (f), Token (f), QLINK (f), f);
      printf ("\n");

      for (f = dec; f != NIL; f = DCHAIN (f)) {/* all devices */
            printf ("%10f %10f %20s %4d %4d   f=%4d\n",
                  TQSUM (f), PREEMPTC (f), fom[NAME (f)], DCHAIN (f), RESQ (f), f);
            printf ("%10f %10f %20d %4d %4d\n ",
                  TLAST (f), QEXCNT (f), QLEN (f), NOFBusy (f), NOFSERV (f));

            for (d = f + 2; d < f + 2 + NOFSERV (f); ++d)
                  printf ("  %10f %10f %20d %4d %4d   d=%4d\n",
                        BSTART (d), BSUM (d), RELCNT (d), USRPRI (d), USRTKN (d), d);

            for (q = RESQ (f); q != NIL; q = QLINK (q))
                  printf ("    %10f %10f  %20d %4d %4d    q=%4d\n",
                        PRI (q), RTIME (q), Event (q), Token (q), QLINK (q), q);
            printf ("\n\n");
      }
}


/*-----------------------------------------------------------------------*/
/*      cmdinput - return i th additional (integer) input value          */
/*                 given as command line parameter i=2  (i=1,2,...,9)    */
/*-----------------------------------------------------------------------*/

int   cmdinput (i)
int   i;
{
      return (inputi[i]);
}
```

```
5680
5681  /*-------------------------------------------------------------------*/
5682  /*      checkcmd - check for the command as an input argument        */
5683  /*                                                                   */
5684  /*-------------------------------------------------------------------*/
5685
5686  checkcmd (cmd)
5687  int   cmd;
5688  {
5689        int   i;
5690
5691        for (i = 1; i <= 10; i++)
5692              if (cmdinput (i) == cmd)
5693                    return (1);
5694
5695        return (0);
5696  }
5697
5698  /*-------------------------------------------------------------------*/
5699  /*      expntl - return random from Exp( x1 )                        */
5700  /*-------------------------------------------------------------------*/
5701
5702  FLOAT expntl (x1)
5703  FLOAT x1;
5704  {
5705        return (-x1 * log (prob ()));
5706  }
5707
5708  /*-------------------------------------------------------------------*/
5709  /*      random - return integer random from Equiprob( i1, i2 )       */
5710  /*-------------------------------------------------------------------*/
5711
5712  INT random (i1, i2)
5713  INT i1, i2;
5714  {
5715        return (i1 + (i2 - i1 + 1) * prob ());/* chopping is used */
5716  }
5717  /* EEE */
5718
5719  /*-------------------------------------------------------------------*/
5720  /*      report - user interface to reporter                          */
5721  /*-------------------------------------------------------------------*/
5722
5723  report (num)
5724  INT num;
5725  {
5726        reporter (num);
5727  }
5728
5729  /* DDD */
5730  /*-------------------------------------------------------------------*/
5731  /*      reset - reset facility and queue statistics                  */
5732  /*-------------------------------------------------------------------*/
5733
5734  reset (flag, num)
5735  INT flag, num;
5736  {
5737        INT i, j;
5738
5739        if (flag)
5740              reporter (num);
5741
5742        i = dec;
5743        while (i > NIL) {                   /* each facility */
5744              MQLEN (i) = QEXCNT (i) = PREEMPTC (i) = TQSUM (i) = 0;
5745              for (j = i + 2; j < i + 2 + NOFSERV (i); ++j)
5746                    RELCNT (j) = BSUMSQ (j) = BSUM (j) = 0;
5747              i = DCHAIN (i);
5748        }
5749        start = clock;
5750  }
5751  /* EEE */
5752
5753
5754  /*-------------------------------------------------------------------*/
5755  /*      smpl - initialize simulation subsystem called mdl            */
5756  /*-------------------------------------------------------------------*/
5757
5758  smpl (mdl)
5759  char  mdl[];                        /* name of model */
5760  {
5761        INT i;
5762        clock = 0;
5763        start = 0;                    /* initialize counters and arrays, etc   */
5764        smpl_event = 0;
5765        avl = NIL;
5766        fdb = 0;
5767        avl = NOTINIT;
5768        dec = NIL;
5769        for (i = 1; i <= nl; ++i)
5770              l1[i] = l2[i] = l3[i] = l4[i] = l5[i] = l6[i] = 0;
5771        strcpy (mdlname, mdl);
5772        na = 0;
5773        for (i = na; i < nm; ++i)
```



```
5774              fnm[i] = NULL;
5775        Free = 0;
5776  }
5777
5778
5779  /*-------------------------------------------------------------------*/
5780  /*      testoption - return testoption given as 2nd value in command line */
5781  /*-------------------------------------------------------------------*/
5782
5783  int   testoption () {
5784              return (test);
5785  }
5786
5787
5788
5789  /*-------------------------------------------------------------------*/
5790  /*      uniform - return random from Uniform( x1, x2 )               */
5791  /*-------------------------------------------------------------------*/
5792
5793  FLOAT uniform (x1, x2)
5794  FLOAT x1, x2;
5795  {
5796        return (x1 + (x2 - x1) * prob ());
5797  }
5798
5799
5800  /*********************************************************************/
5801  /*                                                                   */
5802  /*      internal routines, not callable from user routines           */
5803  /*                                                                   */
5804  /*********************************************************************/
5805
5806
5807
5808  /*-------------------------------------------------------------------*/
5809  /*      copy - copy string s1 to string s2, at most NAMELENGTH chars */
5810  /*             return length of string copied (include '\0' )        */
5811  /*-------------------------------------------------------------------*/
5812
5813              INT copy (s1, s2)
5814  register char    *s1, *s2;
5815  {
5816        register    INT i;
5817
5818        if (s2 == mdlname)          /* no restrictions on length */
5819              while (*s2++ = *s1++);
5820        else {
5821              for (i = 1; (*s2++ = *s1++) && i <= NAMELENGTH; ++i);
5822              if (i > NAMELENGTH)
5823                    *--s2 = '\0';
5824        }
5825
5826        return (i);
5827  }
5828
5829
5830
5831  /*-------------------------------------------------------------------*/
5832  /*      err - print error message n                                  */
5833  /*-------------------------------------------------------------------*/
5834
5835  /*              err (n) */
5836  err (n)
5837              INT n;
5838  {
5839      static   char *msg[] = {
5840              "<<<<<< 0  not in use",
5841              "<<<<<< 1  EMPTY ELEMENT POOL",
5842              "<<<<<< 2  EMPTY NAME POOL",
5843              "<<<<<< 3  FACILITY DEFINED AFTER QUEUE/SCHEDULE OPERATION",
5844              "<<<<<< 4  NEGATIVE Event TIME",
5845              "<<<<<< 5  PREEMPTED Token NOT IN Event LIST",
5846              "<<<<<< 6  RELEASE OF IDLE/UNOWNED FACILITY"
5847        },
5848
5849        printf ("\n\n**************\n");
5850        printf ("DOBLMN SMPL EXECUTION ERROR DETECTED AT TIME %.10f\n", clock);
5851        printf ("% 48s\n", msg[n]);
5852        printf ("**************\n\n");
5853        reporter (0);
5854        if (test > 0)
5855              check ();
5856
5857        exit (n);
5858  }
5859
5860
5861  /*-------------------------------------------------------------------*/
5862  /*      reporter - generate simulation report                        */
5863  /*-------------------------------------------------------------------*/
5864
5865              reporter (num)
5866              INT num;
5867  {
```

```
5868        INT i, j, k, nr, last, cnt,
5869        FLOAT t, b, u, q, x, xsum, v,
5870
5871        struct tms  buff,           /* buffer area for times */
5872        static char facname[40],  /* work area */
5873
5874  /* Output Results */
5875
5876  loop ( i, 1, SITE ) lost_customer += lost_tr [ i ],
5877  fp = fopen ("ddblmn data","a");
5878  fprintf(fp,"%4 2f %3 3f %3 3f %3 3f %4 4f ",
5879     total_tr_time/(finished_tr*MEG),
5880     total_U_bus  /finished_tr,
5881     total_U_disk /(finished_tr * SITE ) ,
5882     total_U_cpu  /(finished_tr * SITE ),
5883     lost_customer * MEG/clock ),
5884        printf ("\f         DDBLMN-SIM     S I M U L A T I O N    R E P O R T \n\n\n\n");
5885        printf ("MODEL  %-44sTIME       %12f\n ", mdlname, clock);
5886
5887        t = clock - start,
5888        times (&buff),
5889        q = buff.tms_utime / 60.  /* power 60 Hz */
5890        printf ("%46s     INTERVAL  %12f\n", "", t),
5891        printf ("%46s      CPU-TIME  %12f\n\n", "", q),
5892
5893        printf ("%41s%-21s%-24s%-26s\n", " ", "MEAN Busy", "QUEUE LENGTH",
5894                "      OPERATION  COUNTS"),
5895        printf ("%-14s%-16s%-13s%-11s%-11s%-11s%-9s%-30s\n\n", "FACILITY", "COMP  RATE",
5896                "UTIL ", "PERIOD", "VARIANCE", "MEAN", "MAX",
5897                " RELEASE   PREEMPT  QUEUE"),
5898
5899        cnt = 0,
5900        xsum = 0.0,
5901        last = NIL,
5902        while (last != dec) {      /* facility descriptors in defined */
5903              i = dec;             /* order (reverse of linkage)      */
5904              while (DCHAIN (i) != last)
5905                    i = DCHAIN (i),/* i=last device not reported yet  */
5906
5907              v = b = q = u = 0.0,
5908              nr = 0,
5909              for (j = i + 2, j < i + 2 + NOFSERV (i), ++j) {/* sum busy periods */
5910                    b += BSUM (j),
5911                    nr += RELCNT (j),/* and counts       */
5912                    v += BSUMSQ (j),
5913              }
5914
5915              if (t > 0) {        /* not clock ! */
5916                    u = b / t,    /* utility */
5917                    if (QEXCNT (i) > 0)
5918                          q = TQSUM (i) / t,/* queue length */
5919              }
5920
5921              if (nr > 1) {
5922                    v = (nr * v - b * b) / (nr * (nr - 1)),
5923                    b /= nr,
5924              }
5925              else
5926                    if (nr == 1) {
5927                          v = 0.0,
5928                          b /= nr,
5929                    }
5930                    else
5931                          v = 0.0,
5932
5933              copy (fam[NAME (i)], facname),
5934              k = NOFSERV (i),
5935
5936              if (k > 1)
5937                    sprintf (facname, "%20s(%s)", facname, str (k)),
5938
5939              if (nr > 0)
5940                    x = nr / t,
5941              else
5942                    x = 0.0,
5943
5944              printf("%-14s%9f     %8f %12f %10f %9f %9d  %9d%8d  %7d \n"
5945                          , facname, x, u, b, v, q, (int) MQLEN (i), nr,
5946                          (int) PREEMPTC (i), (int) QEXCNT (i)),
5947
5948              fprintf(fp,"%4 3f %4 3f ", u, q ),
5949              if (cnt < num)
5950                    xsum = xsum + x,
5951
5952              cnt++,
5953
5954              last = i,
5955        }
5956
5957        if (num > 0)
5958              printf ("\n\nNode Completion Rate  %8f Average %f\n", xsum, xsum / num),
5959
5960  fprintf(fp,"%8 2f\n", 1.0* buff.tms_utime / 60),
5961  fclose ( fp ),
```



```
5962  }
5963
5964
5965  /*-------------------------------------------------------------------- */
5966  /*      str - return m as a string                                     */
5967  /*-------------------------------------------------------------------- */
5968
5969        char *str (m)
5970              INT m
5971  {
5972        sprintf (stringbuf, "%9d", m),
5973
5974        if (test > 8)
5975              printf ("<<>> str   m=%d  stringbuf=%s\n", m, stringbuf),
5976
5977        return (stringbuf),
5978  }
5979
5980
5981  /*-------------------------------------------------------------------- */
5982  /*      tracer -  print trace message for token j, transaction w1, object w2
5983  /*                and event or facility k                               */
5984  /*-------------------------------------------------------------------- */
5985
5986  tracer (j, w1, w2, k)
5987  INT j, k,
5988  char  w1[], w2[],
5989  {
5990        static      INT pgn = 0,
5991                    lno = 61,
5992        static char w3[20], s[30],
5993
5994        if (lno > 60 && smpl_tr > 1) {    /* event trace */
5995              ++pgn,
5996              lno = 6,
5997              printf ("\f%-42s%-9s %3u\n\n\n\n", "DDBLMN-SMPL Event TRACE", "PAGE", pgn),
5998              printf ("%-18s %-20s %-20s\n\n", "---TIME ---", "-----OPERATION   -- ",
5999                                "      TRANS     Token"),
6000        }
6001
6002        if (j > NIL)
6003              strcpy (w3, str (j)),
6004        else
6005              strcpy (w3, ""),
6006
6007        printf ("\n%7.0f ", clock),
6008        sprintf (s, "%s%s", w1  " ", w2),
6009        sprintf (s, "%-13s%8s", s, str (k)),
6010        printf ("%-13s%8s\n", s, w3),
6011        ++lno,
6012  }
6013
6014  /*-------------------------------------------------------------------- */
6015  /*      reserve - reserve facility f for token tkn with priority pri
6016  /*                return 0 if reserved, 1 if queued                     */
6017  /*-------------------------------------------------------------------- */
6018
6019  /*                INT reserve (f, tkn, priority) */
6020  INT reserve (f, tkn, priority)
6021  register    INT f, tkn,
6022  FLOAT priority,
6023  {
6024        register    INT i,
6025
6026        /* DDD */
6027        if (smpl_tr > 0)
6028              tracer (tkn, "reserve", fam[NAME (f)], f),
6029        if (test > 4)
6030              printf ("<<>< reserve2  f=%u  tkn %u  priority %f\n",
6031                          f, tkn, priority),
6032        /* EEE */
6033
6034        if (busy_free [f] == Busy) {/* it is busy */
6035              enqueue (f, tkn, priority, smpl_event, 0.0),
6036              return (1),
6037        }
6038        else {                      /* find free server, reserve it */
6039              busy_free [f]  = Busy,
6040              i= f+2,
6041              USRTKN (i) = tkn,
6042              USRPRI (i) = priority,
6043              BSTART (i) = clock,
6044              return (0),
6045        }
6046  }
6047
6048  /*-------------------------------------------------------------------- */
6049  /*      set_q -  set up element queue at the begining                   */
6050  /*-------------------------------------------------------------------- */
6051
6052  set_q()
6053  {
6054  register    INT j,
6055
```

```
6066
6067    for (j = fdb; j <= nl - 1; ++j)/* first call, build avail chain */
6068        QLINK (j) = j + 1;
6069    avl = fdb;              /* prevent further allocations */
6070    fdb = NIL;
6071
6072    /* DDD */
6073    if (test > 0) check ();
6074    /* EEE */
6075 }
6076 /*------------------------------------------------------------------*/
6077 /*      facility - define facility named name with n servers        */
6078 /*                 return facility number f                         */
6079 /*------------------------------------------------------------------*/
6080
6081 INT facility( name, n )
6082 char *name ;
6083 INT  n ;
6084 {
6085     INT f, i ;                          /* facility, servers */
6086     if ( fdb+n+1 > nl ) err( 1 ) ;      /* pool empty or short */
6087     if ( fdb == NIL  ) err( 3 ) ;       /* pool not dynamic any more */
6088
6089     f=fdb ; fdb += n+2 ;                /* allocate n+2 elements */
6090     DCHAIN( f ) = dec ; dec=f ;         /* link to descriptor chain */
6091     NOFSERV( f) = n ;                   /* no  of servers */
6092     RESQ( f ) = NIL ;
6093     for ( i=f+2 ; i < f+2+n ; ++i ) USRTKN( i )=NIL ;    /* initialize */
6094
6095     if ( na >= nm ) err( 2 ) ;          /* facility pool empty */
6096
6097     fnm[ na ] = &fname[ Free ] ;
6098     Free += copy ( name, fnm[ na ]) ;   /* set facil. name */
6099
6100     if ( test>6 ) printf( "><>< facility: fnm[%u]=%s (%s)\n",na,fnm[na],fnm[0]) ;
6101
6102     NAME( f ) = na++ ;
6103
6104     if ( empl_tr>0 ) tracer( NIL, "FA", fnm[ NAME( f ) ], f ) ;
6105
6106     return( f ) ;
6107 }
6108 my_trace()
6109 {
6110 }
6111 /*
6112  bp
6113 *-------------------------------------------------------------------
6114 *
6115 *                  Cross Reference of Functions and Macros
6116 *
6117 *-------------------------------------------------------------------
6118
6119  cancel                  960 984 5123
6120
6121  cause                   734 1109
6122
6123  cc_lock                 1268 1287 1298 1310 1327 1383 1525 1534 1560 1569 1594 1816 2380  2387
6124                          2444 2449 2599 2607 2689 2697 4825 4826 4830
6125
6126  cc_lock_get_bus         1268 1287 1298 1310 1327 1383 2380 2444 2689 2697 4825 4826
6127
6128  cc_new_tr               2841 2846 4374
6129
6130  cc_unlock               1537 1562 1671 1820 1731 1806 1817 2318 2388 2900 3092 4832
6131
6132  check                   461 749 1154 1222 1231 1239 1876 2063 2167 2177 2185  5513  5879  5810
6133                          5868 5837 6046
6134
6135  check_in_state          461
6136
6137  checkcmd                5868
6138
6139  child_gen               1496 1502 3103 3109
6140
6141  cmdinput                5857 5874
6142
6143  consistency             3177 3718
6144
6145  copy                    5796 5916 6070
6146
6147  dbsize                  1496 1503 1810 2002 3414 3421
6148
```



```
6139  enlist                  778 869 886 891 952
6140
6141  enqueue                 931 936 6017
6142
6143  err                     306 326 737 768 1066 1069 1070 1072 1076 1079 1082 1089 1092 1108 3928
6144                          5371 5374 5818 6058 6059 6067
6145
6146  est_slowest_site        3688
6147
6148  expntl                  3864 4346 4401 5664
6149
6150  facility                1058 1060 1061 6053 6072
6151
6152  finish                  276 1106 1168 1172 1176 2162 2687 3946 4869 5369 5370 5373  5861  5862
6153                          5863 5864
6154
6155  first_rt                349 2630 2631 2633 3666 3669 3671 4380 4386 5464 5559
6156
6157  get_bus                 1137 1260 1266 1287 1298 1310 1327 1383 2380 2444 2689 2697 4825 4826
6158
6159  get_disk                1168 1848
6160
6161  get_local_lock          2340 2434 2480 3683 3689 3887 4672
6162
6163  get_related_site        3736
6164
6165  get_token_id            2262 3117 3820 4363
6166
6167  get_tr_id               3844 4361
6168
6169  getelm                  722 773 946
6170
6171  hv_of_site              3878 4239
6172
6173  initial                 1056 3921
6174
6175  is_covered              4034 4149
6176
6177  link_token_id           3929 4067
6178
6179  link_tr_id              3930 4076
6180
6181  local_resume            2781 2797 2809 3731 3741 4093 4099 5096 5131 5163 5175 5528
6182
6183  locate                  791
6184
6185  main                    580 595 1341 1342 1422 1909 1939 1944 3480 3482 3902  4007  4022  4209
6186                          4210 4410 4420 5049 5074 5123 5646
6187
6188  max                     526 1428
6189
6190  materialization         1092 1612 4133
6191
6192  min                     526 1428
6193
6194  min_site_estimate       4170
6195
6196  min_site_gen            4228 4233 4476
6197
6198  my_trace                1113 6080
6199
6200  need_bus_token          1120 1203
6201
6202  need_lock_in_LP         1508 4273
6203
6204  needed_lock             4311 4360
6205
6206  new_trans               1124 4334
6207
6208  orphan_test             1488 4439
6209
6210  pick_min_site           1677 1686 4484 4473
6211
6212  precede                 5458 5480 5485 5491 5492
6213
6214  prob                    499 501 723 5667 5697 5778
6215
6216  putelm                  724 743 980
6217
6218  qlength                 725
6219
6220  query_gen               4368 4598
6221
6222  random                  5694
6223
6224  release                 802 809 1079 1142 1146 1147 1153 1164 1439 2047 3167
6225
6226  release_bus             1147 1439 3167
6227
6228  release_disk            1164 2047
6229
6230  repeat                  425 438 441 448 532 536 538 541 544 547 560 2294 2363 2611  2701  2716
6231                          4080 4078
6232
```

bp

```
*
*        Cross Reference of Important Global Variables and Constants
*
```

char 619 627 636 694 706 706 5648 5661 5741 5796 5821 5864 5951 5970 5974 6054

stringbuf 706 5964 5967 5969

class_of 343 1388 1822 2001 2066 2276 2349 2442 3072 3764 3767 3774 3778 3790 3951 4376

component_set 344 606 1032 1034 3610 3972 4042 5413

counter 284 461 464 466 469

disk 302 322 345 412 486 1068 1167 1168 1183 1184 1848 2038 2039 2047 3947 3960 4366 5366 5381 5863

disk_time 322 2038 2039

domain_range 595 1341 3480 3482 3902 4007 4210

domain_width 580 1342 1422 1909 1939 1944 3902 4022 4209 4410 4420 5049

end_time 286 1060 1192 1193

end_time 286 1060 1192 1193

est_att_card 346 3440 3481 4207 4208

est_rel_card 347 3436 3448 3494 3495 3639

est_slowest_site_of 348 2081 2084 2469 3666 4381 4813 4851 4866 5080 5143 5366

event 272 583 741 747 967 968 1051 1109 1113 1116 1117 1121 5073 5123 5126 5126 5128 5618 5617 5620 5746 6017

execution_time 287 2463 2464

fap 458 475 479 489 491 493 609 1028 1095 1412 1764 2022 2937 2977 3470 3486 3627 3660 3786 3796 3807 4293 4398

finished_tr 278 1105 3945 5369 5370 5373 5861 5862 5863 5864

first_rt_site 349 2630 2633 3671 4380 4386 5364 5369

allowed_error 308 1070 1106 5374

home_of 350 1726 1788 2520 3110 3628 3766 3777 3795 4079 4366 4943 5348 5383

hv 361 361 1683 3132 3133 3878 3910 4196 4200 4207 4212 4213 4235 4239 4240 4241 4536 4552 5321

ini_rel_card 352 584 1943 1976 3152 3460 4000 4001 4007 4012 4399

ini_rel_card_ratio 584 4001

iteration 328 1060 1069 1106 5371 5374

iteration_error 328 1106 5371 5374

last_rt 290 1527 1721 1773 1784

last_wj 363 1644 2247 2602 4379 5363

link_to_next_tr 364 3689 4080 4082 4942

local_lock 366 2340 2434 2480 2666 2668 2686 2668 2776 2778 2778 2792 2796 2806 2808 3307 3308 3312 3386 3683 3689 3713 3887 3942 4103 4672

local_min_att 366 1611 1612 1648 3131 3906 4246 4544 5320

lock_state 357 2661 2662 2681 2682 2774 2775 2777 2792 2806 2864 2865 2866 2867 2924 2944 2969 3007 3026 3046 3058 3062 3078 3079 3080 3086 3213 3340 3348 3349 3366 3367 3368 3369 3372 3713 3984 4318 4320 4323 4324 4976 4976 4978 4979 4980 4981 4982 4988 4989 4991 4992 4993 4994 4996 5011 5012 5014 5015 5018 5017 5018 5019 5020 5391

lock_validity 358 2638 2672 2766 2770 2872 3721 3729 3960 5621 5622

lost_tr 359 3940 4344 5868

lptime 360 1883 2078 2087 2094 5223 5224 5272

maximum_site 294 519 523 5224 5230 5236 5261 5266

mean_tr 266 1073 3864 4345 4401

min_hv 381 1683 3132 4236 4240 4241 4536

min_site 362 1577 1884 1886 4192 4196 4198 4201 4214 4218 4228 4233 4242 4246 4388 4464 4473 4475 4491 4634 4643 5380

minimum_site 293 511 512 518

new_pg 363 2618 2713 2819 2820

new_relative 364 2297 2366 2720 2724 2736 2738 2742 2749 2860 2862 2863 2866 2868 3198 3234 3236 3236 3237 3240 3241 3260 3261 3262 3263 3267 3268 3270 3271 3272 3276 3277 3286 3324 3331 3332 3337 3338 3346 3367 3373 3966 4887 4888 5100 5141 5418 5419

new_token 274 3825 3831 3832 4066 4922 4923

new_tr 273 1124 2841 2846 3849 3868 3869 4083 4334 4374 4942 4944

not_lock_in_LP 366 1230 1693 1616 1622 2069 2726 2971 3140 3886 4174 4297 4670 5094 5119

num_in_gwj 367 1656 1661 2214 2223 2244 2246 2496 2610 4383 5367

num_in_lp 368 1601 1643 1646 2188 2193 3126 4382 4866 5097 5136 5366

num_of_bro 291 369 1316 1333 1360 1366 1547 4490 4496 4617 5212 5241 5248 5291

num_of_broad 369 1316 1333 1360 1366 1547 4490 4496 4617 5212 5241 5248 5291

num_of_tr_at 370 4342 4360 5317 5348

orphan 371 1486 1526 1722 1738 2074 3134 4439 4461 4462

packet 323 1281 1282 1316 1317 1333 1334 1342 1343 1344 1360 1361 1366 1367 1368 1367 1368 1428 1428 1429

pg 383 372 373 1326 2296 2366 2445 2604 2612 2616 2633 2634 2636 2636 2638 2643 2694 2702 2713 2716 2722 2729 2740 2743 2766 2771 2789 2794 2807 2819 2820 2821 2870 2871 3086 3197 3284 3323 3346 3360 3361 3706 3716 3728 3739 3939 3964 4100 4886 5070 5076 5094 5100 5130 5141 5162 5174 5420 5421 5467 5468 5460 5461 5462 5466 5466 5468 5469 5470 5473 5476 5527 5972 5977 5979

pguser 373 1326 2633 2634 2636 2636 2638 2643 2766 2771 2794 2807 3706 3716 3728 3739 3939 4100 5094 5130 5162 5174 5527

picked_site 375 1682

pptime 376 2027 2132 3682

queried_att 377 458 459 490 1029 2996 3037 3469 3487 3642 3661 3673 3677 3808 3988 4409 4419 4620 4650 4760 4760 4773 4782 5048 5400

queried_rel 378 454 455 488 492 1017 1723 1763 1880 1902 1933 1972 2006 2647 2861 2936 2978 2981 3020 3671 3638 3788 3798 3978 4286 4317 4397 4407 4417 4612 4636 4668 4746 4782 5392

r_unlock_time 266 1080 1536 1661 1670 1614 1617 1934 2911 2960 3012 4831

rel_card 347 352 379 584 1726 1727 1882 1943 1976 2007 2026 3162 3474 3436 3448 3449 3460 3491 3492 3494 3476 3663 3664 3679 3683 3699 4000 4001 4007 4012 4399 5064

related_site 380 435 436 439 442 446 449 1020 3113 3767 3786 3796 3799 3803 3862 4389 4449 5316 5388

relation_conf 583 1027 1904 2983 2995 3024 3038 3808 4006 4022 4647 4767 4780

relation_nj_width 672 1424 4017 4412 4422 5062

relation_width 381 1882 2006 2026 4017 4021

selectivity 599 3473 3474 3493 3496 3647 3656 3680

site_att_card 382 1340 1417 1908 1938 3154 3439 3479 3645 3648 3901

site_of 348 383 1014 1112 1208 1286 1444 1866 2062 2081 2094 2167 2253 2412 2469 3122 3626 3666 3832 4062 4280 4366 4381 4809 4813 4851 4858 4866 4922 5080 5143 5366

site_rel_conf 387 603 1023 1030 1491 3142 3462 3477 3617 3967 4206 5410

sjtime 388 1910 1949 2108 2113 5229 5240 5236 5236 5281

start_time 285 1060 1064 1193

start_time 285 1060 1064 1193

target_jatt 389 3989 4621 4866 4713 4736 5401

target_njatt 390 3981 4411 4421 4618 4869 4737 5061 5396

target_rel 391 3022 3979 4613 4640 4662 4665 4682 4688 4699 4731 5394

tb_state 392 443 483 488 471 473 509 514 521 1116 1163 1212 1268 1447 1868 2080 2420 2727 2928 2929 2930 2931 3125 3629 3649 3941 4142 4143 4189 4373 4586 4617 4840 4867 4890 4891 4892 5083 5098 5120 5134 5160 5173 5314 5319 5361

time_of_tr 410 4111 4114 4304 5362

token_of 393 461 610 1880 1883 1700 1779 1788 1794 2726 2971 3111 3124 4110 4120 4122 4173 4239 4367 4570 4672 4859 5092 5110 5119 5123

```
6609                    5318
6610
6611  total_tr          277 300 3945 3946 4366 5363 5370 5861
6612
6613  total_U_bus       301 3947 5364 5862
6614
6615  total_U_cpu       303 3947 5366 5864
6616
6617  total_U_disk      302 3947 5365 5863
6618
6619  total_tr_time     300 3946 5363 5370 5861
6620
6621  tr_of             394 452 481 483 485 1015 1111 1209 1256 1445 1856 2053 2158 2264  2418
6622                    3123 4061 4064 4281 4364 4444 4810 4882 4921
6623
6624  tr_phase          396 472 2758 2907 3300 3423 3961 4372 5077 5362
6625
6626  tr_rel_width      396 1976 3640 4408 4410 4412 4418 4420 4422
6627
6628  trace_time        324
6629
6630  trace_tr          397
6631
6632  updated_jatt      398 1416 1421 3990 4622 4708 5402
6633
6634  updated_pjatt     399 1423 3982 4616 4716 5395
6635
6636  updated_rel       400 3980 4319 4322 4614 4702 4986 5393
6637
6638  working_site      292 3716
6639
6640  writing_rel       401 474 475 1410 2023 2862 3076 3080 3299 3340  3348  3349  3366  3367
6641                    3368 3369 3382 3774 3775 4608 4703 5009
6642
6643  */
```


VITA

Yao-Nan Lien was born in Taichung, Taiwan, Republic of China, on February 15, 1951. He graduated from The First N.C.O School of Chinese Army in January 1969, and served as a sergeant in Chinese Army from 1969 to 1975. He began his college education at the National Cheng Kung University in September 1975, and received a B.S. degree in Electrical Engineering in June 1979. From July 1979 to July 1980, he was a teaching assistant at the National Taiwan Institute of Technology.

He attended Purdue University for his graduate study in August 1980. Since then he had been a research assistant in the School of Electrical Engineering. He received a M.S. degree in Electrical Engineering in December 1981. He is an instructor in the Department of Computer and information Science, the Ohio State University, since January 1986.

He is a student member of the IEEE computer society, ACM, and AAAI.