The Effect of Index Partitioning Schemes on the Performance of Distributed Query Processing

Jörg Liebeherr, *Member, IEEE*, Edward R. Omiecinski, and Ian F. Akyildiz, *Senior Member, IEEE*

***Abstract*—The benefit of using indexes for processing conjunctive queries in a database system is well known. The use of indexes in distributed database systems is equally justified. In a distributed database environment a relation may be horizontally partitioned across the nodes of the system and indexes may be created for the fragment of the relation that resides at each node. However, as an alternative, one might construct each index on the entire relation, i.e., global indexes, and then partition each index between the nodes. An approach is presented for processing such an index partitioning scheme in response to a conjunctive range query. The performance of these schemes is evaluated in terms of the response time of a query and the utilization of processors, disk, and communication network while varying the number of nodes and query mix.**

***Index Terms*—Distributed database system, query processing, indexing scheme, partitioned global index, partial index, conjunctive queries, simulation, performance evaluation.**

I. Introduction

WITHIN the past ten years, query processing in distributed database systems has been a major area of research [2], [3], [7], [9], [11], [13]. Specific interest in distributed query processing for local area networks has also been popular [1], [12], [14], [18], [20]. Most of the research has been oriented to the optimization of multirelation queries, such as a join of two or more relations [12], [14], [16], [18], [19]. However, there are tradeoffs that are involved in processing single relation queries that have not as yet been explored. We examine these tradeoffs in the context of a locally distributed database system.

Intraquery parallelism as well as inter-query parallelism can provide improvements in response time for individual transactions [17]. For intraquery parallelism, a query optimizer would produce a query plan that could be executed in parallel by a number of processors. For interquery parallelism, several queries would be executed in parallel. In this paper we examine the tradeoff between intraquery and interquery parallelism for single relation queries that use secondary indexes.

In this work we consider only one type of query, which is a single relation conjunctive query. This type of query is one for which the access plan might use one or more indexes, i.e., if the selectivity of the attribute(s) is small [8]. A conjunctive query for a relation R with attributes $A_1, A_2, \cdots, A_N$ has the following form: $Term_1$ and $Term_2$ and $\cdots$ and $Term_M$ where $Term_i \equiv A_j <$ comparison operator $> \}$ $value_i$, $1 \leq j \leq N$ and $value_i \in domain(A_j)$. Range queries and exact match queries are special cases of conjunctive queries.

Since we are concerned with evaluating different index partitioning and processing schemes in our distributed database system, we will limit the access plans for the query to just those which use the index. In addition we are concerned only with secondary indexes. The index structure is the well known B+ tree [8]. We assume that the leaf nodes are linked together to allow efficient processing of a range query. We assume that the pages that comprise the index are stored on a secondary storage device, i.e., a disk, as well as the pages that store the tuples for the relation. In addition, the pages that store the indexes are disjoint from the pages that store the data. Since our intent is to compare different partitioning schemes, we divorce the query processing from the buffering scheme in that an access to an index block, other than the root, will cause a disk access.

There are two basic strategies used for processing conjunctive queries: the *single index method* and the *intersection method*. For a conjunctive query with M terms, assume that K of those terms have associated indexes:

1) *Single Index Method*. For the *single index method*, one of the corresponding indexes for the K terms (usually the one for the attribute with the smallest selectivity) would be used as the access path to locate the tuples that satisfy that term. The associated tuples would be read and each would be examined to see if they satisfy the additional terms in the query. If they do, then those tuples would be returned as the query result.
2) *Intersection Method*. For the *intersection method*, the K indexes would be searched to find the addresses of the tuples that satisfy each of the K terms individually, call these sets $Address_1, \cdots, Address_K$. Then the intersection of these sets would be performed, i.e., $Result = \cap_{i=1}^{K} Address_i$. Afterwards, the tuples whose addresses are contained in $Result$ are read. If there were additional terms in the query that did not have an associated index, then the tuples in $Result$ would be examined to see if they satisfy those terms, and only the qualifying tuples from $Result$ would be returned as the answer. Otherwise, all the tuples in $Result$ would be returned as the answer to the query.

Manuscript received April 18, 1990; revised November 5, 1990 and August 8, 1991.

J. Liebeherr was with the Computer Science Division, University of California–Berkeley, Berkeley, CA 94720, and he is now with the Computer Science Department, University of Virginia, Charlottesville, VA 22903.

E. Omiecinski is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

I. F. Akyildiz is with the School of Electrical Engineering, Georgia Institute of Technology, Atlanta, GA 30332.

IEEE Log Number 9207785.

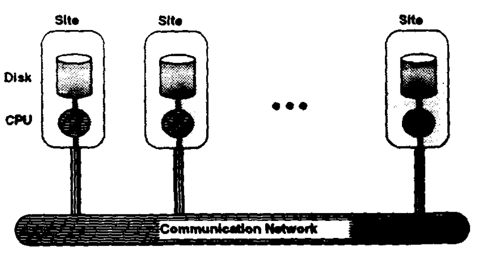


Fig. 1. Structure of the distributed database.

The paper is organized as follows. In Section II, we describe the use of indexes in a distributed database system. First we explain the classical partial index scheme. Then, we introduce a new scheme, called partitioned global index, for storing an index. In Section III, we show how a query is processed under the above mentioned index schemes. We develop a simulation model of the distributed database system in Section IV. For each index scheme a separate model is created. In Section V, we investigate the complexity of each index scheme analytically. In Section VI, sets of experiments are conducted where in each set the parameters such as the number of sites, the transmission capacity of the communication network, and the degree of complexity of a query are varied. We discuss the tradeoffs of the classical and the new index schemes. In Section VII, we discuss the conclusions of the obtained results.

II. Storage Organization

The distributed database system consists of several sites interconnected by a communication network as shown in Fig. 1. The sites operate as self-contained computer systems, i.e., each site has its own CPU and a disk drive that serves as secondary storage. The communication network allows broadcast messages that can be received by all sites. Since we are concerned only with the comparison of our partitioning schemes and their associated processing requirements, we limit our analysis to a single relation database. The single relation is horizontally partitioned across all sites without replication. The part of the relation at each site is referred to as a fragment. We make no assumptions about how the tuples from a relation may be distributed. For example, a round robin, hashed or range partitioning approach as discussed in [5], [6] may be used. Our only assumption is that the number of tuples at each site is approximately the same. For example, tuples from the employee relation may be partitioned as follows:

employee tuples where the age < 30 are stored at site 1,
employee tuples where the age 30 and 45 are stored at site 2,
employee tuples where the age > 45 are stored at site 3.

All data items are accessed indirectly using one or more indexes. A query can initiate execution at any site. If a secondary index was needed, the typical approach [6], [17] would be to construct a separate physical index for each fragment. Therefore, the fragment and its associated index are located at the same site. These indexes are referred to as partial indexes (*PI*) [17]. Fig. 2 illustrates the concept of partial indexes.

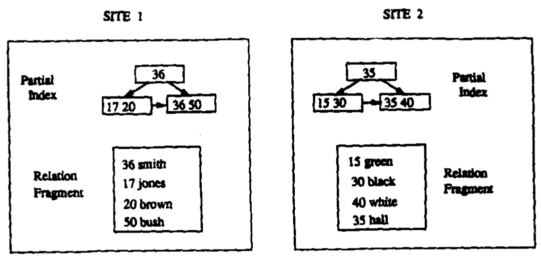


Fig. 2. Partial indexes.

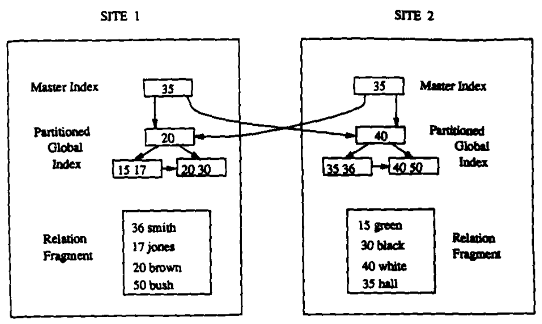


Fig. 3. Partitioned global index.

As an alternative to the partial index scheme, one could conceptually think of building an index for the entire relation, i.e., a global index, and then partitioning the index across the sites. Along with a given partition of the index, each site would have a small master index that indicates the partitions that are stored at each site. Fig. 3 illustrates the concept of a partitioned global index (*PGI*). Intuitively, partial indexes look attractive from the standpoint of intraquery parallelism. When using the *intersection method*, the indexes can be searched and the intersection performed at each site in parallel. However, all sites must search their respective indexes to answer the conjunctive query.

Equally intuitively, *PGI* looks attractive from the standpoint of inter-query parallelism. That is, if the conjunctive query involves a limited set of attribute values, only some of the sites will need to search their index allowing other sites to process different queries. However, as one can imagine, additional messages will be required for processing the tuple addresses found in the partitioned global indexes.

The index structure of *PGI* requires additional work for modifications to the data set. Note that in *PI* only one site would be responsible for handling the insertion of the tuple into its fragment as well as inserting the tuple's address in that site's index. In *PGI*, the number of sites that are involved in an update of a tuple is dependent on the number of *PGI* indexes. Since the value of the attribute determines the location of the index entry, in the worst case, all index entries and the tuple are stored at different sites. Additionally, updates of an indexed attribute may cause the transfer of an index entry to another site if the new value of the attribute is outside the range of index entries stored at the current site.

Addresses in a distributed database with *PGI* have the structure: < site, local address >. *Site* refers to the name of a site and *local address* denotes a logical local address. At a site, the physical address is obtained by a lookup into a table that contains the logical and physical address of a tuple.

The use of indexes in a distributed database has not been fully explored. Two parallel database system prototypes that use indexes are GAMMA [6] and BUBBA [4]. In GAMMA, partial indexes are used. In addition, a global index is used, if the associated data is also stored at the site where the partitioned index is stored. This is a restricted case of the partitioned global index scheme that we propose. In BUBBA [4], global clustered indexes and local clustered indexes are used, similar to XPRS [17].

In this work, we quantify the tradeoff between a *PI* and a *PGI* scheme when processing conjunctive queries in a database with infrequent update operations. In a previous study [10] we examined the tradeoffs between the two schemes when considering only range queries that utilize a single index. We note that each index adheres to the particular scheme under evaluation, i.e., all indexes are partial indexes or all indexes are partitioned global indexes.

III. Query Processing

In the last section, we described two different schemes of storing index and data blocks in the distributed database system, the classical partial index scheme (*PI*) and the new partitioned global index scheme (*PGI*), respectively. Each indexing scheme implies a different strategy of answering a query. The query processing strategies for both schemes are presented below.

A. Query Processing for Partial Indexes

For our purposes, we can think of a query as requesting tuples for a set of one or more ordered attribute values. In *PI*, the index at each site must be searched, when a query has requested a set of attribute values. However, attribute values that are stored in an index at a given site have their corresponding data records also stored at that site. This means that once a value has been found in an index block, it is assured that the tuples with that value are stored at the site where the index entry has been found. After accessing the data blocks, the tuples are sent to the site that initiated the query.

B. Query Processing for Partitioned Global Indexes

The index for the entire relation (i.e., global index) is partitioned across the sites. This is similar to the idea of range partitioning tuples as in GAMMA [5], [6], however in our case, the index is range partitioned and the data is partitioned according to some other method, e.g. round-robin or range partitioned on some other attribute. Each site is assumed to know the distribution condition of the index for all sites. We call this the master index. It requires a small amount of storage since it contains only one entry per site consisting of site address and attribute value. This scheme was illustrated in Fig. 3. When a query initiates execution at a site the master index is consulted and messages are sent to only those sites that have index entries for the desired set of attribute values. Each site that has index entries for the query receives a subset of the attribute values, i.e., exactly those attributes that appear in the index at that particular site. The site is requested to lookup the index entries for the attributes in the subset. Note that index entries and corresponding data entries are not necessarily stored at the same site. Therefore, once the index entry for an attribute has been found, possibly all sites have to be accessed to obtain the tuples with that value. A site that searches its index for a subset of attribute values obtains a list of addresses. Note that *PGI* guarantees that each index search returns at least one address (as long as the desired attribute value is present in the database). Once the lists of addresses are obtained they are sent to the site that initiated the query. After all addresses have arrived at the query-initiating site, messages are sent to those sites that store the tuples corresponding to the list of addresses. On reception of a message with addresses, a site accesses its data blocks, obtains the tuples and delivers them to the site that initiated the query,[1].

In the following we will give an example of processing a conjunctive query. We will explain how a query for the described database system is executed in *PI* and *PGI*. We will consider both strategies for processing conjunctive queries, i.e., the *single index method* and the *intersection method.*

C. Example

A distributed database system may consist of 5 sites $(SITE_1, SITE_2, \cdots, SITE_5)$. The relation REL contains 50 tuples $(REL = TUP_1, TUP_2, \cdots, TUP_{50})$ each tuple having a set of n attributes $(ATTR_1, ATTR_2, \cdots, ATTR_n)$. Let the database have an index for $ATTR_1$. For this example we assume that the values for $ATTR_1$ are unique, i.e., there are 50 different values for $ATTR_1$ with a range given by $(1, 2, \cdots, 50)$ and the value of $ATTR_1$ of a tuple is given by its index ($ATTR_1$ $[TUP_j] = j$, for $j = 1, 2, \cdots, 50$). For simplicity we assume that the values of the other attributes take values in the same range, i.e., $ATTR_i$ $[TUP_j] \in \{1, 2, \cdots, 50\}$, $i = 2, 3, \cdots, n, j = 1, 2, \cdots, 50$), without assuming uniqueness of the values. Let $SITE_1$ initiate the following query:

```
SELECT *
FROM    REL
WHERE   ATTR1 ≥ 24 AND ATTR1 ≤ 38
AND     ATTR2 ≥ 5  AND ATTR2 ≤ 13
AND     ATTR3 ≥ 42 AND ATTR3 ≤ 46.
```

When investigating the *intersection method* we assume that the database has indexes for all attributes $ATTR_1$, $ATTR_2$ and $ATTR_3$. Otherwise we assume an index only on $ATTR_1$ and apply the *single index method.*

1) *Partial Indexes (PI)*
Single Index Method

[1] As an alternative query processing strategy for *PGI* consider the following. After the index search, a site could determine to which sites the addresses refer and immediately send requests to these sites (without sending the addressing back to the site at which the query originated). However, this strategy showed worse performance than the strategy discussed above due to a large overhead for communication.

$SITE_1$ sends a broadcast message that contains the list of attribute values $24, 25, \cdots, 38$ together with the range conditions for $ATTR_2$ and $ATTR_3$ to all sites. Since $SITE_1$ itself does not know whether it has the index entries to some of the requested values, it starts to search its own index. All other sites start to search their index once the broadcast message is received from the communication network. When a site has scanned its index it holds a set of addresses of data items that match the attribute values. According to the specification of *PI*, these data items are stored at the same sites where the index entry was found. If all data items at one site are accessed the tuples are examined if they satisfy the given conditions for $ATTR_2$ and $ATTR_3$, i.e., $ATTR_2 = 5, 6, \cdots, 13$ and $ATTR_3 = 42, 43, \cdots, 46$. The remote sites (from the point of view of $SITE_1$, namely, $SITE_2, SITE_3, SITE_4, SITE_5$) send those data items that satisfy the conditions to $SITE_1$. $SITE_1$ waits until all data arrive and processes the data items.

Intersection Method

As in the *single index method* $SITE_1$, the initiator of the query, broadcasts a message, this time containing three lists of attribute values $(24, 25, \cdots, 38)$, $(6, 7, \cdots, 13)$, $(42, 43, \cdots, 46)$. Each site then searches the index entries for the different indexes and obtains three lists of addresses. An intersection is done with the lists resulting in a single list of addresses for tuples that match the selection conditions of the query. The intersected list is used to access the data blocks. Once the tuples are retrieved they are sent back to $SITE_1$.

2) *Partitioned Global Indexes (PGI)*

Single Index Method

Here we assume that the index at $SITE_1$ contains entries for attribute values $1, 2, \cdots, 10$, the index at $SITE_2$ contains entries for $11, 12, \cdots, 20$, etc. Analyzing the same query as before, $SITE_1$ sends only messages to $SITE_3$ and $SITE_4$ requesting to lookup values $24, 25, \cdots, 29, 30$ for attribute $ATTR_1$ at $SITE_3$ and values $31, 32, \cdots, 37, 38$ at $SITE_4$. Only these sites start to search their index and obtain the addresses $(ADR_{24}, ADR_{25}, \cdots, ADR_{30})$ at $SITE_3$ and $(ADR_{31}, ADR_{32}, \cdots, ADR_{38})$ at $SITE_4$, respectively. We denote with ADR_r the address of the tuple with value $ATTR_1[TUP_r]$. Both $SITE_3$ and $SITE_4$ obtain the addresses and send them back to $SITE_1$. If $SITE_1$ has received both lists $(ADR_{24}, ADR_{25}, \cdots, ADR_{30})$ from $SITE_3$ and $(ADR_{31}, ADR_{32}, \cdots, ADR_{38})$ from $SITE_4$ it partitions the list of addresses according to the first component of each address, which is the site identifier. It then sends requests for data to sites that are referred to in the addresses. The range conditions for $ATTR_2$ and $ATTR_3$ are also sent to those sites. The sites, which receive the data request, access the data blocks, filter those tuples that do not meet the conditions for $ATTR_2$ and $ATTR_3$, and send the tuples back to $SITE_1$.

Intersection Method

Here, all lists of addresses are sent back to the querying site. At this time, the intersection of addresses can be done. The intersection yields the addresses of those tuples that satisfy all conditions from the query. The obtained address list is then partitioned according to the sites that have data items and the sublists are sent to the remote sites. After this step, the query answering strategy proceeds as described for the *single index method*.

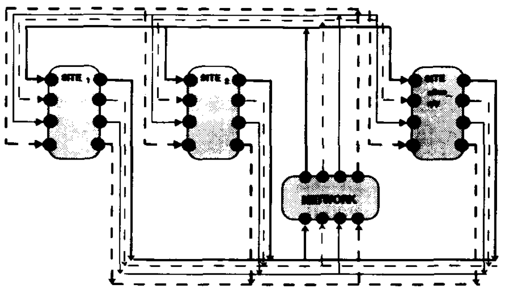


Fig. 4. Model of the distributed database system.

In the following section we develop a simulation model that implements the distributed database systems as described above. Two models are described, featuring either a *PI* or a *PGI* scheme.

IV. Simulation Model

The simulation model has been developed using the RESQ2 software package [15]. In the following we describe the parameters that characterize the simulation model. We did not include components of a database system in our model that are not influenced by the selection of a particular indexing scheme. Concurrency control, recovery mechanisms, buffer replacement strategies and logging were found not to be affected, if the strategy for handling conjunctive queries is changed.

A. Distributed Database System

A global view of the implementation of the distributed database system is given in Fig. 4. The simulation model consists of two types of subsystems, a site and the network. The number of sites is denoted by the parameter *SitesQty*. Each site contains an independent working CPU and a disk. The CPU in the model processes four different classes of requests. It generates a list of attribute values (*cpu_query*), it processes lists of attribute values before the index blocks are accessed (*cpu_ref*), it processes lists of addresses to obtain data items (*cpu_adr*) and it processes the incoming data before returning it to the user (*cpu_data*). We assume that each request to the CPU takes an exponentially distributed amount of time with mean value S_{CPU}. Considering a processor speed of 4 *MIPS*, we set $S_{CPU} = 5$. Incoming requests are served in a First-Come First-Served manner. However, processing of attribute or address lists may need more than one access to the disk. In this case, the list is re-queued at the CPU after the disk access to process the remaining part of the list. We assume that one disk access is required for each data item. For index retrieval, one disk access is assumed to yield up to

IndexPerBlock addresses at once. The size of a data record is given by 128 $Bytes$, addresses (and attribute values) are 4 $Bytes$ long. Disk accesses necessary to obtain addresses from the index blocks (*disk_ref*) and data records from the data blocks (*disk_adr*) need an exponentially distributed time period with mean value S_{DISK}. The time that a transaction waits before a new query is issued (*think_time*) is exponentially distributed with mean value S_{think}. The flow of control is modeled by messages that contain the information needed for processing and routing in the distributed system, such as: originating site, destination site, message type and attributes, addresses, or tuples.

As mentioned before, we assume that the database consists of only one relation since we are only interested in single relation queries. Tuples are uniformly distributed over all sites. The number of distinct attribute values in the index, denoted by *NumAttr*, is assumed to be 1% of the total number of tuples in the relation. Therefore, given *NumAttr*, the total number of tuples *NumberTuples* is obtained by

$$NumberTuples = NumAttr \cdot 100. \qquad (1)$$

A query requests a uniformly distributed list of attributes with maximum length value *MaxAttr*. The number of addresses that are found for one index entry is denoted by *TupPerAttr*. In the simulation model a uniformly distributed number of addresses with maximum *MaxTupPerAttr* is stored with an index entry. Attribute values are uniformly distributed over all sites. For our purposes, the attribute values are integers with range $[1 : NumAttr]$. The range of attribute values having the index entries stored at a site $SITE_i$ is computed from

$$\left[(i-1)\cdot\left\lceil\frac{NumAttr}{SitesQty}\right\rceil + 1 : i\cdot\left\lceil\frac{NumAttr}{SitesQty}\right\rceil\right] \quad \text{for } i = 1, 2, \cdots, SitesQty. \qquad (2)$$

If *NumAttr* is not an integral multiple of *SitesQty*, then $SITE_{SitesQty}$ may have fewer index entries. The range of attribute values requested for a given attribute is computed with two random variables X_{low} and X_{size}. X_{low} is *uniform* $[1 : NumAttr]$ distributed and indicates the lowest attribute value. X_{size} gives the number of attributes requested and follows also a *uniform* $[1 : MaxAttr]$ distribution. Therefore, the range of a query with a single attribute is given by

$$[X_{\text{low}} : (X_{\text{low}} + X_{\text{size}} - 1) \bmod NumAttr]. \qquad (3)$$

For conjunctive queries, the above computations are performed separately for each attribute appearing in the query. The number of attributes occurring in a query is given by the parameter *NoIndex*.

B. The Communication Network

The network has a bandwidth of 10 Mbit/s as in Ethernet. A data packet is assumed to have a maximum size of 1 $kByte$. The setup time for a packet, i.e., the time to packetize data and perform network access functions, is assumed to be exponentially distributed with mean S_{nw_setup}. A message may need to be split into several data packets. The number of tuples (attribute values, addresses) that can be transmitted in a single packet is denoted by *TupPerPacket* (*AdrPerPacket*). The data record size is assumed to be 128 $Byte$, yielding a value of $TupPerPacket$ of 8. Since attribute values and addresses are assumed to have a length 4 $Byte$, *AdrPerPacket* is set to 256. Note that the last packet corresponding to a particular message may not be completely filled. The overall transmission time of a packet with attribute values or addresses (*nw_ref, nw_adrbk, nw_adr*) and a data packet (*nw_data*) are assumed to be exponentially distributed with mean values S_{nw_ref}, S_{nw_adr}, S_{nw_adrbk} or S_{nw_data}, respectively. Overhead information of a packet is assumed to be constant and therefore included in the setup time of the packet. The total transmission delay of a packet consists of a fixed part, the setup time, and a variable part that accounts for the transmission delay. Naturally, the transmission delay is dependent on the amount of data transmitted in a packet. With the given network bandwidth of 10 Mbit/s the transmission delay for a data record is given by 0.1 ms, for a single attribute value (or address) 0.003 ms. The total time to transmit a packet containing attribute values (or addresses) and data is then calculated by

$$\begin{aligned} S_{nw\_ref} &= S_{nw\_adrbk} = S_{nw\_adr} \\ &= S_{nw\_setup} + \max(AdrPerPacket, \\ &\qquad \text{remaining addresses}) \cdot 0.003 \text{ ms} \\ S_{nw\_data} &= S_{nw\_setup} + \max(TupPerPacket, \\ &\qquad \text{remaining data records}) \cdot 0.1 \text{ ms}. \end{aligned} \qquad (4)$$

1) **PI**
The network model for query processing with *PI* transmits messages containing a list of attribute values (*nw_ref*) and messages containing a list of tuples (*nw_data*), as shown in Fig. 5.[2] Broadcasting of messages is modeled by generating multiple copies of a message (at node nw_fork). Note that messages are, if necessary, split into several packets. Splitting messages into several packets is done under exclusive possession of the network channel. For that reason mutual exclusion is guaranteed by nodes $P(mutex)$ and $V(mutex)$.

2) **PGI**
Fig. 6 shows the simulation model of the network for *PGI*. For *PGI* we send lists of addresses over the network. We distinguish two different types of messages that contain lists of addresses, i.e., addresses sent back to the site that started the query (*nw_adrbk*) and addresses that are sent to retrieve data (*nw_adr*). The four types of messages are stored in separate queues contending for the network server that simulates the propagation delay of a message.

C. The Site

Although the major components of the model are similar for both indexing schemes the different ways a query is handled lead to a conceptually different flow control. Therefore, we dis-

[2]The symbols used to denote elements of the model are explained in the Appendix.

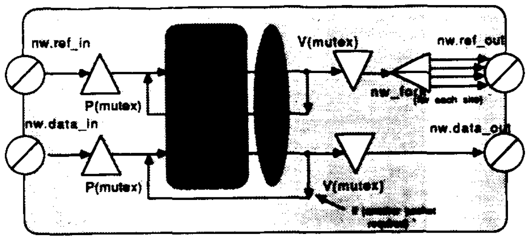


Fig. 5. Network model for *PI*.

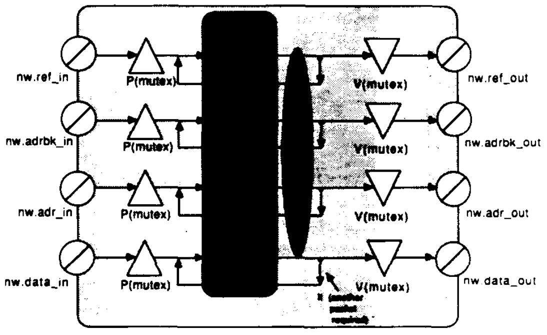


Fig. 6. Network model for *PGI*.

cuss the models separately for *PI* and *PGI*. The description of the model is simplified compared to the actual implementation.

1) **PI**

The model of a site with *PI* indexes is given in Fig. 7. The major components can be identified in Fig. 7 as dark shaded areas representing the CPU and the disk. The CPU consists of four waiting queues and one server. The disk has two queues, one for current index block accesses (*disk_ref*), one for data block accesses (*disk_adr*). Processing of a query is implemented in the site model as follows. A query starts off from node *think* (top of Fig. 7) where a transaction issues a query. The query enters queue *cpu_query* and eventually enters the server of the CPU. Here a list of attribute values for the range query is determined. At node *fork1*, the list is replicated, one copy going to the local *cpu_ref* queue, the other leaving the site to the network submodel. A node receives lists of attribute values locally and from remote sites. If multiple indexes are used, each list is partitioned into sublists, each referring to a list of attribute values for a single index. For each index used, one separate list of attribute values is produced (at *forkind*). One disk access is required for *IndexPerBlock* attributes. This is represented by a visit of queues *cpu_ref* and *disk_ref*. According to the description of *PI*, all addresses are forwarded to *cpu_adr*, and data block access starts. For queries that use multiple indexes the intersection of the obtained address list is done prior to accessing the data blocks (at *joinind*). Each data item requires a separate data block access, each time consuming CPU and disk time. Once all data items are accessed, they are sent to the site that started the query. Local data is sent directly

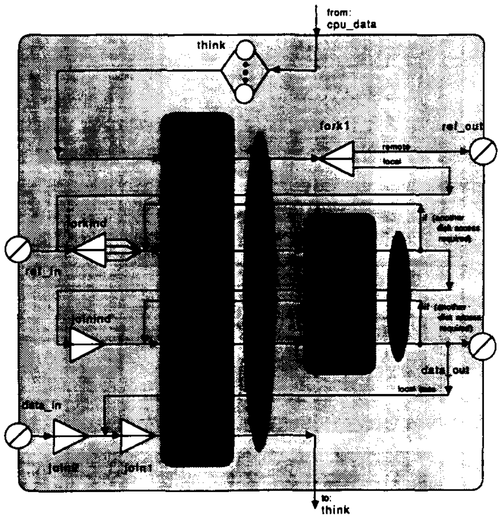


Fig. 7. Site Model for PI.

to node *join1* where messages that left node *fork* in an earlier phase of the process wait on each other before any processing can proceed. Data items from remote sites wait on each other at *join2*. Remote and local data items are collected at *join1*. The complete set of data items is then queued at *cpu_data* and finally, transferred back to *think*.

2) **PGI**

The model of a site implementing a *PGI* scheme is given in Fig. 8. We will limit our discussion of the *PGI* model to differences to the model with a *PI* scheme. Node *forkind* generates a separate list of attribute values for each attribute occurring in the conjunctive query. Since the location of index blocks to attribute values is known, the list of attribute values is now partitioned into sublists, one sublist for each site that has an index block to at least one attribute value in the list. For each sublist a separate message is generated at node *fork1*. Note that the number of created messages is dependent on the values of the random variables X_{low} and X_{size}. Sublists referring to remote sites are transmitted over the network. A sublist referring to the local site is directly forwarded to queue *cpu_ref*.

The process of retrieving addresses from an attribute value is exactly as described for *PI*. Different from *PI*, the addresses are sent back to the site that started the query. There, the lists of addresses to a query wait on each other at nodes *join1* and *joinind*. If all corresponding lists have arrived, the complete list of addresses to data items found for a particular index of one query is available. The site now partitions the list according to the sites that are referred to in the list (at *fork2*). Each created sublist now proceeds to the site that has the data items corresponding to its list of addresses. Data access is as described for *PI*. Having retrieved all data the list of data items is sent to the querying site. Eventually, all data items arrive at *join2* and proceed to *cpu_data*.

TABLE I

Parameter	Description	Distribution
SitesQty	Number of sites	—
NumAttr	Total number of attribute values in database	—
X_{size}	Number of attribute values for a query	$uniform[1 : MaxAttr]$
X_{low}	Lowest attribute value of a query	$uniform[1 : MaxAttr]$
IndexPerBlock	Maximum number of index entries per disk block	—
TupPerAttr	Number of tuples found for one attribute value	$uniform[1 : MaxTupPerAttr]$
TupPerPacket	Maximum number of tuples fitting in a data packet	—

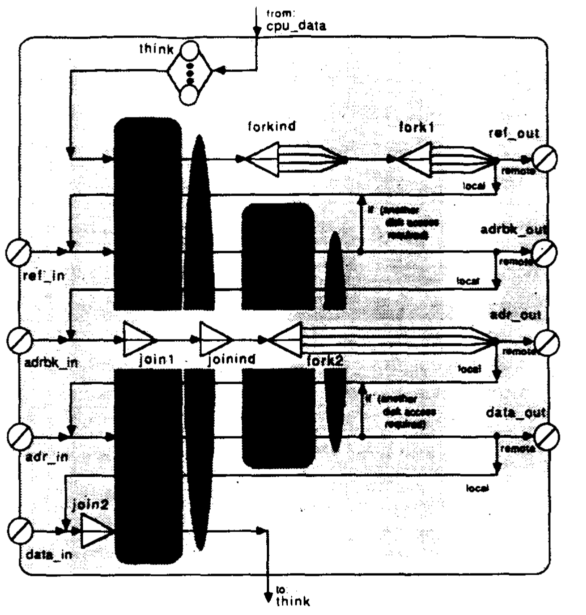


Fig. 8. Site Model for *PGI.*

V. Analysis of I/O and Communication Overhead

The description of the index access schemes and their respective query processing strategies in the previous sections indicate a tradeoff between disk processing overhead for *PI* and communication overhead for *PGI.* In this section we will quantify the overhead of the presented schemes. The notation of the parameters used in this section and their distribution is summarized in Table I.

To simplify the analysis we substitute the uniformly distributed parameters X_{size} and *TupPerAttr* by constants X'_{size} and $TupPerAttr'$:

$$X'_{\text{size}} \equiv const$$
$$TupPerAttr' \equiv const.$$

Furthermore, we assume that address lists always fit into a single packet.

A. Disk Processing Overhead

The main advantage of the new *PGI* over *PI* is the selected search of index entries. *PI* has to perform index accesses for all attribute values generated by a query at each site of the distributed system. If the number of sites is large compared to

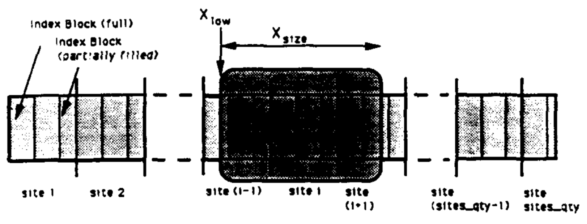


Fig. 9. Distribution of index entries (partitioned global index).

the number of attribute values, then most index block accesses are not necessary. Denoting the number of index accesses by α_{PI} and α_{PGI} for the number of required index block accesses for a range query in *PI* and *PGI*, respectively, we obtain

$$\alpha_{PI} = \left\lceil \frac{X'_{\text{size}}}{IndexPerBlock} \right\rceil \cdot SitesQty \tag{5}$$

$$\left\lceil \frac{X'_{\text{size}}}{IndexPerBlock} \right\rceil \le \alpha_{PGI} \le \left\lceil \frac{X'_{\text{size}}}{IndexPerBlock} \right\rceil + \left\lceil \frac{X'_{\text{size}}}{NumAttr/SitesQty} \right\rceil. \tag{6}$$

The second term of the sum in (6) accounts for the access of index blocks that are not completely filled with index entries. Note that this term does not exceed $SitesQty$. The equations consider only accesses to the leaf nodes of the $B+$ tree where the index entries are kept. Since the leaf nodes are assumed to be linked together in order to process a range query efficiently (see section 1) the number of neglected accesses to inner nodes of the $B+$ tree is small. Fig. 9 illustrates the derivation of (6). Each site except the last site ($site_{SitesQty}$) keeps $\lceil \{NumAttr/SitesQty\} \rceil$ index blocks. Thus, the last index block of each site may not be filled completely. Note that the number of index blocks that have to be accessed is dependent on the value of X_{low}, i.e., the starting point of the range of desired attributes. The number of disk accesses required to retrieve the data tuples is the same for either indexing scheme.

B. Communication Overhead

Communication between the sites takes place at different stages of the processing of a query. To separate the communication demand of each stage, we distinguish the following phases during which communication takes place:

Phase 1: Transfer of list of attribute values to sites for index retrieval.

Phase 2: Transfer of address lists back to querying site.
Phase 3: Transfer of address lists to sites that hold the data items.
Phase 4: Transfer of data items to querying site.

We now discuss the communication demand of each indexing scheme during the various phases of processing a query.

1) **PI**

During *Phase 1* only one message is transmitted, since we assume broadcasting capabilities of the communication network. *Phase 2* does not apply to a *PI* scheme. No communication is required during *Phase 3*, since the index and data tuples are always stored at the same site. During *Phase 4* all data items are sent to the site that started the query. The total number of tuples for a query is given by

$$X'_{\text{size}} \cdot TupPerAttr'$$

Assuming that data items are distributed evenly over the sites, the number of sites that have index and data entries for a query is given by

$$\min\{SitesQty, X'_{\text{size}}\}.$$

The maximum number of tuples fitting into one data packet is given by *TupPerPacket*. Additionally, the last packet transmitted by each site may not be filled completely. Hence, the number of data packets sent during *Phase 4*, denoted by $\beta^{(4)}_{PI}$, is computed by

$$\beta^{(4)}_{PI} = \left\lceil \frac{X'_{\text{size}} \cdot TupPerAttr'}{TupPerPacket \cdot \min\{SitesQty, X'_{\text{size}}\}} \right\rceil \cdot \min\{SitesQty, X'_{\text{size}}\} \cdot Fact \quad (7)$$

where *Fact* normalizes the number of packets since no site sends packets to itself. *Fact* is obtained by

$$Fact = \frac{SitesQty - 1}{SitesQty}. \quad (8)$$

2) **PGI**

During *Phase 1*, a packet with attribute values has to be transmitted only to those sites holding index entries for the attribute values. Let $\beta^{(1)}_{PGI}$ denote the mean number of packets to be transmitted during *Phase 1*. Note that $\beta^{(1)}_{PGI}$ is dependent on the value of X_{low} (see Fig. 9). Its value is determined by

$$\left\lceil \frac{X'_{\text{size}} \cdot SitesQty}{NumAttr} \right\rceil \cdot Fact \le \beta^{(1)}_{PGI} \le \left(\left\lceil \frac{X'_{\text{size}} \cdot SitesQty}{NumAttr} \right\rceil + 1 \right) \cdot Fact. \quad (9)$$

The number of packets transmitted during *Phase 2* is given by the number of sites that received the list with attribute values during *Phase 1*:

$$\beta^{(2)}_{PGI} = \beta^{(1)}_{PGI}. \quad (10)$$

In *Phase 3*, each site with index entries sends a packet with an address list to those sites that are referenced in the addresses. The number of sites that receive an address list is limited by the number of remote sites. We obtain:

$$\begin{aligned}\beta^{(3)}_{PGI} &= \min\{SitesQty, \text{total number of addresses found}\} \cdot Fact \\ &= \min\{SitesQty, X'_{\text{size}} \cdot TupPerAttr'\} \cdot Fact.\end{aligned} \quad (11)$$

Each site that receives an address list during *Phase 3* finds

$$\frac{X'_{\text{size}} \cdot TupPerAttr' \cdot Fact}{\beta^{(3)}_{PGI}} \quad (12)$$

tuples. The number of packets in *Phase 4* is therefore calculated as

$$\beta^{(4)}_{PGI} = \left\lceil \frac{X'_{\text{size}} \cdot TupPerAttr' \cdot Fact}{\beta^{(2)}_{PGI} \cdot TupPerPacket} \right\rceil \cdot \beta^{(3)}_{PGI} \quad (13)$$

TABLE II
EXAMPLE PARAMETERS

NumAttr	4000
X'_{size}	60
$TupPerAttr'$	5
IndexPerBlock	32
TupPerPacket	8

The equations derived in this section are simplified since they do not account for the distribution of parameters X_{size} and *TupPerAttr*. Additionally, the uniform distribution of the tuples over the sites was simplified by just distributing the tuples evenly over all sites. However, the expressions provide some insight into the degree of complexity of both disk processing and communication overhead. We now present an example that applies the above results.

Example: Let the parameters of a distributed database system be given as in Table II.

The value of *SitesQty* is varied from 10 to 100. Table III shows the overhead of disk processing. The total number of data packets required for one query is shown in Fig. 10. Note that *PGI* has a significant communication overhead compared to *PI*.

VI. EXPERIMENTS

In this section we discuss the experiments conducted with the simulation model that was developed according to the specification in Section III. In each experiment we varied a parameter of the model and compared the performance for the different index schemes and query processing strategies. The following parameters are varied in different sets of experiments:

I) Number of sites (*SitesQty*).
II) Number of indexes in a query (*NoIndex*).
III) Transmission capacity of the communication network (S_{nw_setup}, S_{nw_ref}, S_{nw_adrbk}, S_{nw_adr}, S_{nw_data}).
IV) Number of disks per site.

TABLE III
DISK PROCESSING OVERHEAD

SitesQty	10	20	30	40	50	60	70	80	90	100
α_{PI}	20	40	60	80	100	120	140	160	180	200
α_{PGI}	[2,3] [a]	[2,3]	[2,3]	[2,3]	[2,3]	[2,3]	[2,4]	[2,4]	[2,4]	[2,4]

[a] [.] denotes the range of values.

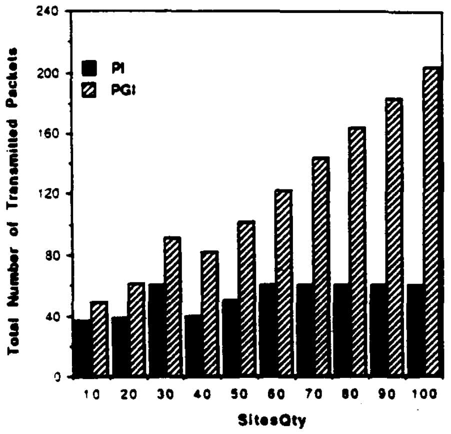


Fig. 10. Total communication overhead.

TABLE IV
BASIC PARAMETERS

SitesQty	16
NoQuery	3 *per site*
NoIndex	3
S_{think}	3 s
S_{CPU}	5 ms
S_{DISK}	30 ms
S_{nw_setup}	5 ms
$S_{nw\_ref}, S_{nw\_adrbk}, S_{nw\_adr}$	see (4)
S_{nw_data}	see (4)
NumAttr	25 $\cdot$*SitesQty*
MaxAttr	20 per query
MaxTupPerAttr	10 per index entry
TupPerPacket	8
AdrPerPacket	128

The basic parameters for the simulation model are specified in Table IV. These parameters remain unchanged throughout all experiments, if they do not denote the parameter that is varied for a particular experiment. We assume that the distributed database is homogeneous, i.e., the parameters of the components for all sites are the same.

Note that the number of tuples in the database is dependent on the number of sites (*SitesQty*). Thus, if we add new sites to the distributed system, we simultaneously increase the size of the global database. By this, we avoid obtaining a lightly loaded system when sites are added to the distributed system. We present the following performance measures:

- *Mean Response Time*, which is the average time a message carrying the information for a particular query needs from leaving node *think* to entering it again. The mean response time is also referred in the literature as cycle time, turnaround time, residence time, sojourn time, etc.
- *Utilization*, which is the fraction of time that a particular device is busy.

In the following sections we describe our simulation results and observations. The run time of the simulation was chosen such that the 95% confidence interval are less than 1% of the mean values for utilization and less than 5% for the mean response time. We will use the following abbreviations for the different query processing strategies:

PI-SI :	Partial index with single index method.
PI-IM :	Partial index with intersection method.
PGI-SI:	Partitioned global index with single index method.
PGI-IM:	Partitioned global index with intersection method.

A. Experiment I

In this experiment we study the performance of the query processing strategies if the number of sites *SitesQty* is varied between 2–16. The mean response time of a query for the different strategies is depicted in Fig. 11. *PGI* shows superior performance compared to *PI* for both the single index method and the intersection method. Note the performance difference between the two indexing schemes if the single index method is used. Fig. 12–14 depict the utilization of the resources, i.e, CPU, communication network and disk. As can be seen in Fig. 12, the CPU is not a critical resource of the system. CPU utilization is low for all strategies. In Fig. 13 we see that the utilization of the communication network increases faster with the number of sites for *PGI* than for *PI*. Note that the network utilization for *PGI-IM* reaches saturation at *SitesQty = 12*. Fig. 14 shows a high utilization of the disk for all query processing strategies. The decrease of disk and CPU utilization for *PGI-IM* at *SitesQty = 16* is explained by the fact that the system's bottleneck has migrated from the disk of each site to the communication network.

B. Experiment II

In this experiment we investigate the sensitivity of query processing to the variation of the complexity of a query.[3] We achieve this by varying the parameter *NoIndex* between 1 and 5, with *NoIndex = 1* being a nonconjunctive query. Recall that the single index method (SI) uses only one index for data retrieval. The mean response time for all query processing strategies is given in Fig. 15. *PI-SI* shows very poor performance for higher values of *NoIndex*. *PGI-IM*

[3] The simulations for *Experiment II* were run with parameter *NumQuery* = 5.

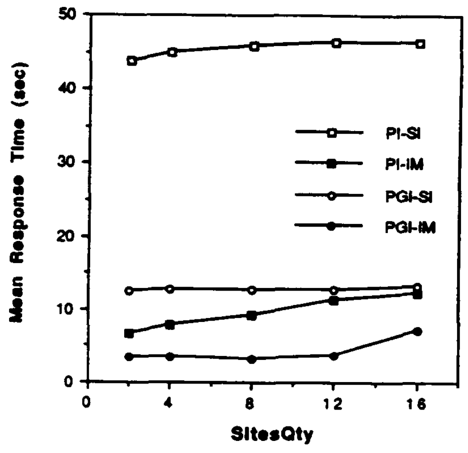


Fig. 11. Mean response time.

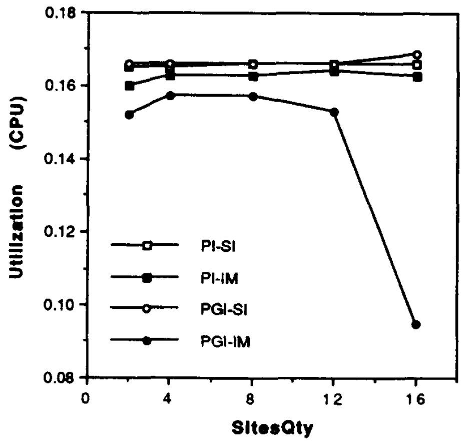


Fig. 12. CPU utilization.

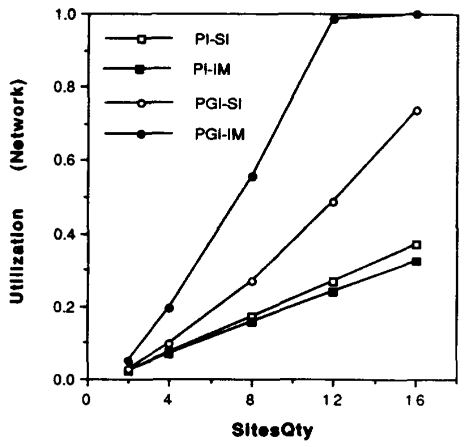


Fig. 13. Network utilization.

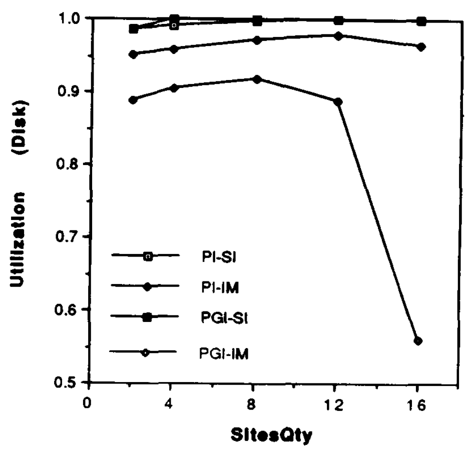


Fig. 14. Disk utilization.

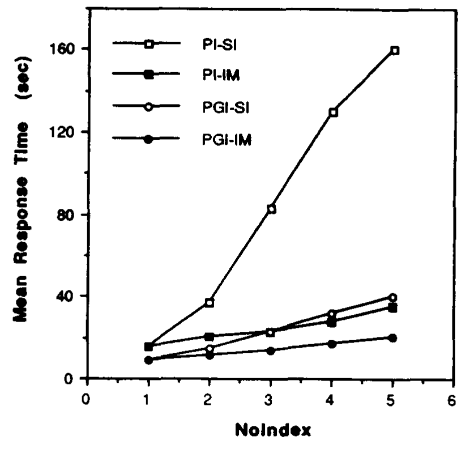


Fig. 15. Mean response time.

appears to be the best strategy for complex conjunctive queries. The disadvantage in *PI-IM* of accessing the index blocks for all attribute values at each site becomes obvious when we investigate the composition of the disk's utilization. The workload of the disk consists of index accesses to retrieve index blocks and data access for retrieval of data blocks. Figs. 16 and 17 decompose the utilization of the disk into index and data accesses. We see that as the number of indexes increases, the disk for *PI-IM* spends most of the time accessing index blocks. For *NoIndex* = 5, although the disk is busy almost all the time, it uses only about 1/3 of its activation for data retrieval. Here, the disadvantage of doing index searches at each site, as required for *PI*, becomes obvious. On the contrary, Fig. 17 reveals that although the total utilization of the disk decreases with increasing values of *NoIndex*, due to congestion of the network, the absolute amount of time spent for data retrieval in *PGI-IM* is more than in *PI-IM*.

C. Experiment III

In the previous experiments we saw that the performance differences of query processing of *PGI* and *PI* are dependent on the transmission speed of the communication network and the speed of the disk. Since improvements in communication technology will provide faster networks in the near future (up to 150 Mbit/s with optical fiber technology), it is of high interest to study our query processing schemes for networks with

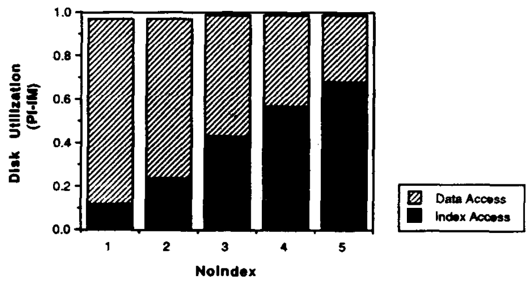


Fig. 16. Disk utilization (*PI, intersection method*).

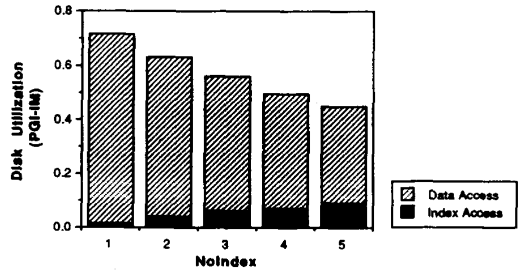


Fig. 17. Disk utilization (*PGI, intersection method*).

a higher transmission capacity. In this series of experiments, we increase the transmission speed and the setup time of the network gradually by increasing the parameter *nw_speed*. The transmission of a packet is then computed from (4) by

$$S'_i = nw\_speed \cdot S_i,$$
$$i \in \{nw\_ref, nw\_adrbk, nw\_adr, nw\_data\}. \quad (14)$$

The results for the mean response time of a query are shown in Fig. 18. Fig. 19 depicts the decrease of network utilization for increased transmission capacity of the network. In Fig. 18 we see that only *PGI-IM* is able to take advantage of a faster communication network. For all other strategies, we observe that the mean response time is not affected by increasing the transmission capacity of the network. The performance of the system is then limited by the disk, and increasing the network speed does not improve the response time of a query.

D. Experiment IV

In this section we investigate the performance of the *PI-IM* if disk drives are added to each site. We present results for 1, 2, 3 and 5 disk drives. As in *Experiment III* the parameter that is varied in this experiment is *nw_speed*. Thus, we are able to answer the question if and how much an index scheme with the *PI-IM*-strategy can benefit from a faster communication network if the I/O-capabilities are improved. Fig. 20 plots the results for the mean response time. For comparison, we included the results from *PGI-IM* in *Experiment III* (dashed line). We see that a site with multiple disks benefits from an upgraded communication network only to a certain extent. Fig. 20 shows that for a network with $nw\_speed > 2$ *PGI-IM* provides a better mean response time than *PI-IM*, even if the latter system has 2 disks available at each site.

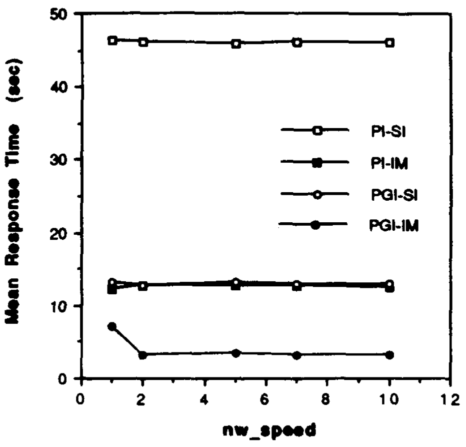


Fig. 18. Mean response time.

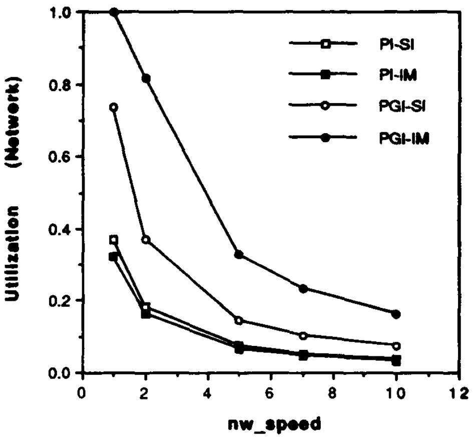


Fig. 19. Network utilization.

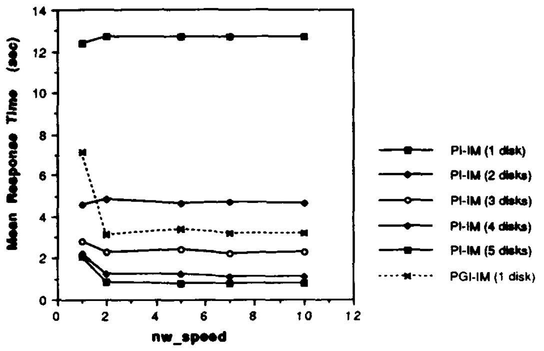


Fig. 20. Mean response time.

VII. Conclusion

We introduced a new indexing scheme called partitioned global indexes (*PGI*) for a locally distributed database system. The new scheme builds a global index for the entire relation and partitions the index across the sites. We also presented a strategy for processing such an index. In order to evaluate the

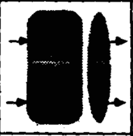
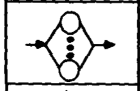
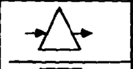
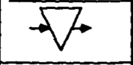
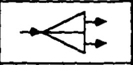
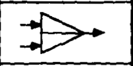
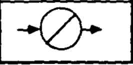
Symbol	Interpretation
	single server with multiple classes
	delay server
	mutual exclusion (start)
	mutual exclusion (end)
	fork node (generates multiple copies of a job)
	join node (merges jobs)
	interface node between subsystems

Fig. 21.

performance of the new scheme, we developed a simulation model. The simulation results were compared to the classical scheme, called partial indexes (*PI*), in which corresponding index and data entries are stored at the same site. We investigated analytically the advantages and disadvantages of the indexing schemes when processing conjunctive queries. Analysis and simulation experiments showed the tradeoffs between the new and the classical scheme.

We demonstrated that *PGI* provides significant performance advantages for query processing in a distributed environment. Even if only one index is used for processing conjunctive queries (single index method), the more efficient index retrieval technique makes *PGI* superior to *PI* by reducing the workload on the disk. This allows the disk to spend more time for data access. We showed that the performance difference between *PI* and *PGI* increases with the complexity of a query. However, it showed that the performance of *PGI* that applies the intersection method is limited by an increased overhead for communication. If a communication network with more transmission capabilities is used, the processing time of a query can be reduced significantly under *PGI-IM*. Since new communication technologies with a high bandwidth (> 50 Mbit/s) were introduced in the late 1980's and will find their way into the market in the 1990's the superiority of using high speed networks makes *PGI* attractive for future use.

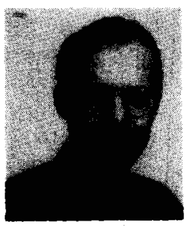
We should reiterate that our work assumes a uniform distribution of data values to sites and that updates have not been included. To increase the applicability of our work, these assumptions should be relaxed in any further study.

APPENDIX
GRAPHICAL SYMBOLS

Fig. 21 shows the graphical symbols that are used to describe the simulation models in Section IV.

REFERENCES

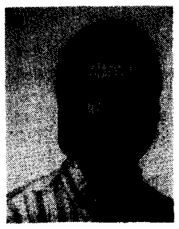
[1] P. Agrawal, D. Bitton, K. Guh, C. Liu, and C. Yu, "A case study for distributed query processing," *Proc. Int. Symp. Databases in Parallel & Distributed Syst.*, 1988, pp. 124–130.
[2] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query processing in a system for distributed databases SDD-1," *ACM TODS*, vol. 6, no. 4, pp. 602–625, 1981.
[3] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems.* New York: McGraw Hill, 1984.
[4] G. Copeland and J. Keller, "A comparison of high-availability media recovery techniques," *Proc. ACM SIGMOD*, 1989, 89–109.
[5] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "Gamma—A high performance dataflow database machine," *Proc. VLDB Conf.*, 1986, pp. 228–237.
[6] D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A performance analysis of the gamma database machine," *Proc. ACM SIGMOD Conf.*, 1988, pp. 350–360.
[7] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in relational database systems," *Proc. ACM SIGMOD Conf.*, 1978, pp. 169–180.
[8] G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases.* Reading, MA: Addison Wesley, 1989.
[9] S. Lafortune and E. Wong, "A state transition model for distributed query processing," *ACM TODS*, vol. 11, no. 3, pp. 294–322, 1986.
[10] J. Liebeherr, I. F. Akyildiz, and E. Omiecinski, "Performance comparison of index partitioning schemes for distributed query processing," *24th Hawaii Int. Conf. Syst. Sci.*, Koloa, HI, Jan. 1991, pp. 317–323.
[11] G. Lohman, C. Mohan, L. Haas, B. Lindsay, P. Selinger, P. Wilms, and D. Daniels, "Query processing in R^*," *Query Processing in Database Systems*, W. Kim, D. Batory, and D. Reiner, Eds. New York: Springer Verlag, 1985, pp. 31–47.
[12] H. Lu and M. Carey, "Some experimental results on distributed join algorithms in a local network," *Proc. VLDB Conf.*, 1985, pp. 292–304.
[13] L. Mackert and G. Lohman, "R^* Optimizer validation and performance evaluation for distributed queries," *Proc. VLDB Conf.*, 1986, pp. 149–159.
[14] W. Perrizo, J. Lin, and W. Hoffman, "Algorithms for distributed query processing in broadcast local area networks," *IEEE Trans. Knowledge Data Eng.*, vol. 1, pp. 215–225, 1989.
[15] C. H. Sauer, E. A. MacNair, J. F. Kurose, "The Research Queueing Package Version 2," *IBM Research Division*, Yorktown Heights, NY, 1982.
[16] A. Segev, "Optimization of join operations in horizontally partitioned database systems," *ACM TODS*, vol. 11, no. 1, pp. 48–80, 1986.
[17] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," *Proc. VLDB Conf.*, pp. 318–330, 1988.
[18] X. Wang and W. Luk, "Parallel join algorithms on a network of workstations," *Proc. Int. Symp. Databases in Parallel & Distributed Syst.*, 1988, pp. 87–95.
[19] H. Yoo and S. Lafortune, "An intelligent search method for query optimization by semijoins," *IEEE Trans. Knowledge Data Eng.*, vol. 1, pp. 226–237, 1989.
[20] C. Yu, K. Guh, W. Zhang, M. Templeton, D. Brill, and A. Chen, "Algorithms to process distributed queries in fast local networks," *IEEE Trans. Comput.*, vol. C-36, pp. 1153–1164, 1987 .



Jorg Liebeherr (S'88–M'92) was born in Cologne, Federal Republic of Germany in 1961. He received the Diplom-Informatiker degree in 1988 from the University of Erlangen-Nürnberg, Nürnberg, Germany, and the Ph.D. degree in computer science in 1991 from the Georgia Institute of Technology, Atlanta, GA. He was a Postdoctoral fellow in the Department of Electrical Engineering and Computer Science at the University of California–Berkeley.

He is an Assistant Professor in the Computer Science Department of the University of Virginia, Charlottesville. His research interests are computer networks, and distributed multimedia systems.

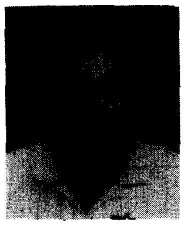
He is a member of ACM (SIGCOMM and SIGMETRICS).



Edward R. Omiecinki received the B.S. and M.E. degrees in electrical engineering, in 1973 and 1974, respectively. from the University of Florida, Gainesville. He received the Ph.D. degree in computer science in 1984 from Northwestern University, Evanston, IL.

He worked for General Dynamics and Texas Instruments from 1975 to 1977. He joined the Computer Science Department at North Dakota State University in 1983 as an Assistant Professor. In 1986, he joined the School of Information and Computer Science (which is now the College of Computing) at Georgia Institute of Technology, where he is currently an Assistant Professor. His research interests include database systems and parallel algorithms.

Dr. Omiecinski is a member of the ACM.



Ian F. Akyildiz (M'86–SM'89) received the B.S., M.S., and Doctor of Engineering degrees in Computer Engineering from the University of Erlangen-Nürnberg, Germany, in 1978, 1981, and 1984, respectively.

Currently, he is an Associate Professor in the College of Computing, Georgia Institute of Technology. He is the co-author of the textbook *Analysis of Computer Systems* (Teubner Verlag 1992) and has published more than 75 technical papers in refereed journals and conference proceedigs. He is an Associate Editor for the IEEE TRANSACTIONS ON COMPUTERS, AN Associate Editor for *Computer Networks* and *ISDN Systems Journal.* He guest edited special issues on "Performance of Parallel and Distributed Simulation" for the *ACM Transactions on Modeling and Simulation*; "Teletraffic Issues in ATM Networks" for *Computer Networks and ISDN Systems Journal* as well as the special issue on "Networks in Metropolitan Area" for the IEEE JOURNAL FOR SELECTED AREAS IN COMMUNCATIONS. His current research interests are in telecommunications systems, computer networks, performance evaluation, parallel simulation, and computer security.

Dr. Akyildiz is a National Lecturer for the ACM and a member of ACM (SIGMETRICS, SIGOPS, and SIGCOMM).