PERFORMANCE COMPARISON OF INDEX PARTITIONING SCHEMES FOR DISTRIBUTED QUERY PROCESSING

Jörg Liebeherr, Ian F. Akyildiz and Edward Omiecinski

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

ABSTRACT

The benefit of using indexes for processing 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. Two approaches are presented for processing such an index partitioning scheme in response to a range query and their performance is compared with the typical scheme. The performance of these schemes is evaluated in terms of the response time, system throughput, network utilization and disk utilization while varying the number of nodes and query mix.

Key Words: Distributed Database System, Performance Evaluation, Simulation, Query Processing

1. Introduction

Within the past ten years, query processing in distributed database systems has been a major area of research [2,3,6,8,11,13]. 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 multi-relation 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.

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

In this work we consider only one type of query, which is a single relation range query. This type of query is one for which the access plan might use one index, i.e., if the selectivity of the key is small [7]. A range query requests tuples from a relation whose key value is within the range of key values specified in the query. As a special case of range query, we allow a query to specify only one key value. If the key is unique then the range query is really an exact match query that would have only a single tuple as its result.

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. The index structure is the well known B+ tree [7]. We assume that the leaf nodes are linked together to allow efficient processing of a range query.

To process a query, the range of key values which appear in the query is used to search the index. The search begins at the root of the tree using the key value specified as the lower bound of the range. The search will always proceed to a leaf node that will contain the key value if there exists at least one tuple in the relation having that key value. From that leaf node, the key values which fall within the range specified in the query will be extracted along with the addresses of the tuples that have those values. If the greatest key value in the leaf node satisfies the query, then the next leaf node is examined, via the pointer which links together adjacent leaf nodes. The search ends when the current leaf node contains a key value greater than the upper bound of the range query. The result of searching the index is a set of tuple addresses. These addresses are then used to retrieve the set of tuples which satisfy the query. We assume that the pages (or nodes) which comprise the index are stored on a secondary storage device, i.e., a disk, as well as the pages which store the tuples for the relation. In addition, the pages that store the index 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.

The paper is organized as follows. In section 2 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 3 we describe the distributed database system under investigation. We show how a query is processed under the above mentioned index schemes. In section 4 we present the simulation model. In section 5 three series of experiments are conducted. In Section 6 we discuss the conclusions of the obtained results.

2. Storage Organization

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. This is reasonable in light of what has been stated. The single relation is horizontally partitioned across all sites, i.e., disk drives associated with each site. The part of the relation at each site is sometimes 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 [4,5] 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.

If a secondary index on the salary column was needed, the typical approach [5,17] would be to construct three physical indexes, one for each fragment. Therefore, the fragment and its associated index are located at the same site. These indexes are referred to as partial indexes [17]. Figure 1 illustrates the concept of partial indexes.

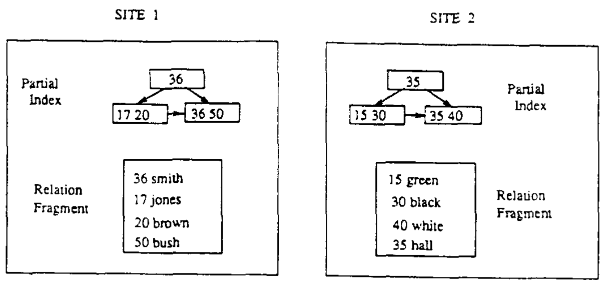


Figure 1: Partial Indexes

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. Figure 2 illustrates the concept of a partitioned global index.

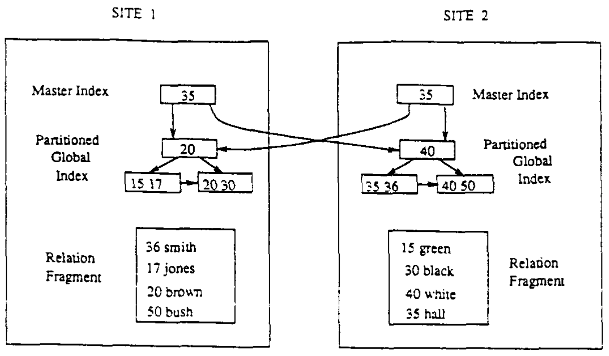


Figure 2: Partitioned Global Index

Intuitively, partial indexes look attractive from the standpoint of intra-query parallelism. That is, the indexes can be searched at each site in parallel. However, all sites must search their index to answer the range query, e.g., select employee where salary < 10K. Equally intuitively, partitioned global indexes look attractive from the standpoint of inter-query parallelism. That is, if the range query involves a limited range of key 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. In this work we investigate these schemes and quantify when one of these index schemes should perform better than the other.

We also indicate that updates are not addressed here, e.g., inserting a new tuple in the relation. In the partial index approach 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 the partitioned global index approach, at most two sites would be responsible for inserting the tuple in the fragment and its address in the appropriate index. The problem of maintaining indexes of approximately equivalent size would be common to both indexing schemes.

3. System Description

The distributed database system consists of several sites interconnected by a communication network as shown in Figure 3. The sites operate as self-contained computer systems, i.e., each site has its own CPU and a disk drive which serves as secondary storage.

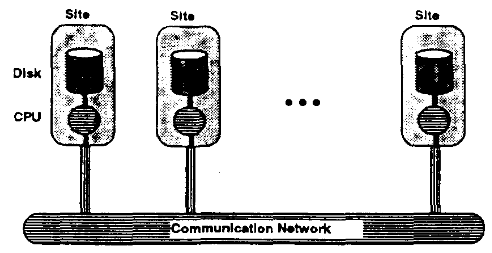


Figure 3: Structure of the Distributed Database

The communication network is an Ethernet-type local area network, thus allowing broadcast messages which can be received by all sites. Database items, i.e., tuples from each relation, are equally distributed over all sites. That is, a relation is horizontally fragmented without replication. For our purposes we consider the case of accessing only one relation. All data items are accessed indirectly with an index. A query can initiate execution at any site. Each transaction issues only one query at a time. The list of key values generated by a query is a range of consecutive key values belonging to a secondary key.

We distinguish two different schemes of storing index and data blocks in the distributed database system, the partial index scheme and the partitioned global index scheme, respectively. Each scheme follows a different policy of answering a query. Three policies are described, *Send-None* for the partial index scheme, *Send-Forward* and *Send-Back* for the partitioned global index scheme. The names of the policies correspond to the way each policy handles addresses of tuples which are available after index retrieval.

a) PARTIAL INDEX

Send-None.

For our purposes, we can think of a query as requesting tuples for a set of one or more ordered key values. In the partial index scheme the index at each site must be searched, when a query has requested a set of key values. However, key 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 key has been found in an index block it is assured that the tuples with that key are stored at the site where the index entry has been found. No address list has to be transmitted to other sites (thus the name: *Send-None*). The partial index scheme with the *Send-None*-policy is the one typically implemented in a distributed database system.

b) PARTITIONED GLOBAL INDEX

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 [4,5], 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 key 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 a key value. This scheme was illustrated in Figure 2. When a query initiates execution at a site the master index is consulted and messages are sent to only those sites which have index entries for the desired set of key values. Each site which has index entries for the query receives a subset of the key values with exactly those keys which appear in the index at that particular site. The site is requested to lookup the index entries for the keys 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 a key has been found, possibly all sites have to be accessed to obtain the tuples with that key value (a key may yield a set of addresses). A site which searches its index for a subset of key values obtains a list of addresses. Once the lists of addresses are obtained we may think of two strategies of

Send-Forward.

Each site which obtained a list of addresses from the index search determines to which sites the addresses refer and immediately sends requests to the sites appearing in the set of addresses. The site which receives a request retrieves the tuples from secondary storage using the disk address part from the address and delivers them to the site which initiated the query. Since key values from a query are ordered, *Send-Forward* can easily be implemented by just assigning each site a range of key values that appear in the index.

Send-Back.

The procedure of obtaining the addresses corresponding to a list of keys is the same as for *Send-Forward*. However, once the addresses are obtained *Send-Back* sends the addresses back to the site which initiated the query. After all addresses have arrived at the query-initiating site, messages are sent to those sites which 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 which initiated the query.

In the following example we will explain how a query for the described database system is processed for each of the considered policies.

EXAMPLE:

A distributed database system may consist of 5 sites ($SITE_1, SITE_2, \ldots, SITE_5$). The relation *REL* contains 50 tuples ($REL = \{TUP_1, TUP_2, \ldots, TUP_{50}\}$) each tuple having a set of n attributes ($ATTR_1, ATTR_2, \ldots, ATTR_n$). Let the database have an index for $ATTR_1$. For this example we assume that the values for $ATTR_1$ are unique key values, i.e., there are 50 different values for $ATTR_1$ with a range given by (1,2,...,50) and the value of $ATTR_1$ of a tuple is given by its index ($ATTR_1\,[TUP_j] = j$, for $j = 1,2,...,50$). Let $SITE_1$ initiate the following query:

```
SELECT *
FROM REL
WHERE ATTR1 ≥ 24 AND ATTR1 ≤ 38
```

We now discuss the execution of the query under *Send-None*, *Send-Forward*, and *Send-Back*.

i) **Send-None**

$SITE_1$ sends a broadcast message which contains the list of keys $\{24, 25,..., 38\}$ to all sites. Since $SITE_1$ itself does not know whether it has the index entries to some of the requested key values it starts to search its own index for all key values. 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 key values. According to the specification of the partial index scheme these data items are stored at the same sites where the index entry was found. For example, assume that the index entry for the key with value $ATTR_1 = 24$ is found at $SITE_3$. The address retrieved from the index entry refers to data stored at $SITE_3$. Therefore, each site which has an index entry for the list of key values starts to access its own data blocks to obtain the data items. If all data items at one site are accessed the remote sites (from the point of view of $SITE_1$, namely, $SITE_2, SITE_3, SITE_4, SITE_5$) send the data items to $SITE_1$. $SITE_1$ waits until all data arrive and processes the data items.

ii) **Send-Forward**

Here we assume that the index at $SITE_1$ contains entries for key values $\{1, 2,..., 10\}$, the index at $SITE_2$ contains entries for key values $\{11, 12,..., 20\}$, etc.. Analyzing the same query as before $SITE_1$ sends only messages to $SITE_3$ and $SITE_4$ requesting to lookup key values $\{24, 25,..., 29, 30\}$ at $SITE_3$ and key values $\{31, 32,..., 37, 38\}$ at $SITE_4$. Only these sites start to search their index and obtain the addresses ($ADR_{24}, ADR_{25}, \ldots, ADR_{30}$) at $SITE_3$ and ($ADR_{31}, ADR_{32}, \ldots, ADR_{38}$) at $SITE_4$, respectively. We denote with ADR_r the address of the tuple with value $ATTR_1\,[TUP_r]$. ADR_r has two components ADR_r = (*site identification,local address*) with the first component giving the site at which the data item is stored, the second component giving the address on the disk at that site. Since addresses in a site's index may refer to tuples at any site, $SITE_3$ and $SITE_4$ analyze the addresses to determine at which site the corresponding tuples are stored. They send a message to each site $SITE_i$ which has at least one address with $SITE_i$ as the first component. A site which receives a message with a list of addresses accesses its local disk to obtain the data. The data is then sent back to $SITE\,1$, the initiator of this query.

iii) **Send-Back**

As in *Send-Forward*, messages are sent to request the addresses for key values $\{24, 25,..., 29, 30\}$ from $SITE_3$ and $\{31, 32,..., 37, 38\}$ from $SITE_4$. Both $SITE_3$ and $SITE_4$ obtain the addresses and send them back to $SITE_1$. If $SITE_1$ has received both lists ($ADR_{24}, ADR_{25}, \ldots, ADR_{30}$) from $SITE_3$ and ($ADR_{31}, ADR_{32}, \ldots, ADR_{38}$) from $SITE_4$ it partitions the list of addresses according to the first component of each address. It then sends requests for data to sites which are named in the addresses. The sites, which receive the data request, access the data blocks, and send the tuples back to $SITE\,1$.

4. Simulation Model

The simulation model has been developed using the RESQ2 software package [14]. In the following we describe the parameters which characterize the simulation model. A complete discussion of the simulation model can be found in [10].

4.1. Distributed Database System

A global view of the implementation of the distributed database system is given in Figure 4.

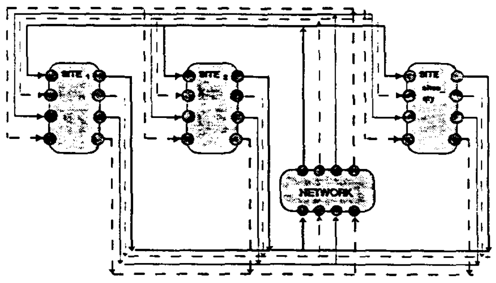


Figure 4: Model of the Distributed Database System

The simulation model consists of two types of subsystem, 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. All sites are connected to the communication network in the same way. The CPU in the model processes four different classes of requests. It generates a list of key values, it processes lists of key values before the index blocks are accessed, it processes lists of addresses to obtain data items and it processes the incoming data before returning it to the user. Note that the number of different message types which are sent between sites and the network is dependent on the implemented policy. We assume that the time each class of requests to the CPU takes to process is exponentially distributed with mean value S_{CPU}. Incoming requests are served in a First-Come First-Served manner. However, processing of key lists and address lists may need more than one access to the disk. In this case, the process working on a list is re-queued at the CPU after the disk access has been finished 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 *Byte*, addresses (and key values) are 4 *Bytes* long. Disk accesses necessary to obtain addresses from the index blocks and data records from the data blocks need an exponentially distributed time period with mean value S_{DISK}. Once a transaction has completed a query it returns to the user until a new query is initiated. The time 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 which traverse the system and are processed at the service units of the system. Each message contains the information needed for processing and routing in the distributed system, such as: originating site, destination site, message type, information needed for a specific type, etc..

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 key values in the index, denoted by *NumKeys*, is assumed to be 1% of the total number of tuples in the relation. Therefore, given *NumKeys* total number of tuples *NumberTuples* is obtained by:

$$NumberTuples = NumKeys \cdot 100 \tag{1}$$

A query requests a list of keys with a uniformly distributed length with maximum value *MaxKeys*. The number of addresses which are found for one index entry, denoted by *TupPerKey* is assumed to have an upper limit. In the simulation model a uniformly distributed number of addresses with maximum *MaxTupPerKey* is stored with an index entry. Key values are uniformly distributed over all sites regardless of the implemented index scheme, i.e., whether the location of key values are known or not. If locations of key values are known (*Send-Forward, Send-Back*) the range of the key values are assumed to be divided among the sites in such a way that each site has the same number of index entries. For our purposes the key values are integers with range *[1: NumKeys]*. The range of key values sites having the index entries stored at a site $SITE_i$ - given a database system with number of sites *SitesQty* - is computed from:

$$\left[(i-1) \cdot \left\lceil \frac{NumKeys}{SitesQty} \right\rceil + 1 \;:\; i \cdot \left\lceil \frac{NumKeys}{SitesQty} \right\rceil \right] \tag{2}$$

$$\text{for } i = 1, 2, \ldots, SitesQty$$

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

$$\left[X_{low} \;:\; (X_{low} + X_{size} - 1) \bmod NumKeys \right] \tag{3}$$

Once the values for X_{low} and X_{size} are known the set of sites and the number of keys at one site can be determined.

4.2. The Communication Network

The Ethernet-type network has a bandwidth of 10 *Mbit/sec*. 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}. When a list of addresses or a list of data items has to be transmitted on the network it is regarded as a message which is only divided into several packets if the message does not fit into one packet. The number of tuples which can be transmitted in a single packet is denoted by *TupPerPacket*. With the given maximum packet size and the data record size of 128 *Byte TupPerPacket* is set to 8. Up to 256 key values or addresses (each with size of 4 *Bytes*) can be transmitted in one packet. Therefore, splitting of lists of key values or addresses is not required since the database considered here does not generate key value lists (address lists) of that size. The overall transmission times of a packet with key values or addresses (*nw_ref, nw_adr*) and a data packet (*nw_data*) are assumed to be exponentially distributed with mean values S_{nw_ref} for a packet containing a list of key values, S_{nw_adr} for a packet containing a list of addresses and S_{nw_data} for a packet containing a list of data items. 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 which 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/sec* the transmission delay for a data record is given by 0.1 *ms*, for a single key value (or address) 0.003 *ms*. The total time to transmit a packet containing addresses (key values) and

$$S_{nw\_ref} = S_{nw\_adr} = S_{nw\_setup} + \tag{4}$$
$$+ \; (addresses\ to\ be\ transmitted) \cdot 0.003ms$$

$$S_{nw\_data} = S_{nw\_setup} + \tag{5}$$
$$+ \; MAX(TupPerPacket,\ data\ records\ to\ be\ transmitted) \cdot 0.1ms$$

5. Experiments

In this section we discuss the experiments conducted with the simulation model described in the previous section. 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 series of experiments:

(I) number of sites (*SitesQty*)

(II) transmission capacity of the communication network (S_{nw_setup}, S_{nw_ref}, S_{nw_adr}, S_{nw_data})

(III) number of disks per site

The basic parameters for the simulation model are specified in Table 1. These parameters remain unchanged throughout all experiments if they do not denote the parameter which is varied for a particular experiment. Note that we assume that the distributed database is homogeneous, i.e., the components for all sites are the same.

SitesQty	24
S_{think}	3 sec
S_{CPU}	5 *ms*
S_{DISK}	30 *ms*
S_{nw_setup}	5 *ms*
S_{nw_ref}, S_{nw_adr}	see equation (4)
S_{nw_data}	see equation (5)
NumKeys	25 · *SitesQty*
MaxKeys	20 *per query*
MaxTupPerKey	10 *per index entry*
TupPerPacket	8

Table 1. Basic Parameters

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.
- *Mean Queue Length* which is the average number of messages containing key values, addresses or tuples waiting to be processed at a particular resource of the system.
- *Throughput* which is the average number of messages leaving a particular device (resource) per unit of time.

In the following sections we describe our simulation results and observations:

5.1. Experiment I

In this experiment we study how the policies perform if the number of sites (*SitesQty*) is varied between 4 - 24. The mean response time for all policies is depicted in Figure 5.

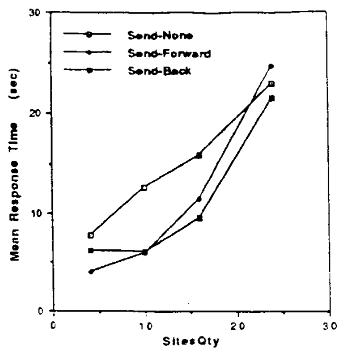


Figure 5: Mean Response Time

Send-Back shows the best performance. *Send-Forward* performs better than *Send-None* when the number of sites is small. However, the mean response time increases slower for *Send-None* when the number of sites becomes larger. The cause of this tradeoff will be clear when we investigate bottleneck situations, i.e., the resource with the highest utilization in the system. Bottleneck study is important since it limits the entire system performance. In Figures 6, 7 and 8 we give the utilization values of disk, CPU and the communication network for *Send-None, Send-Forward* and *Send-Back*, respectively.

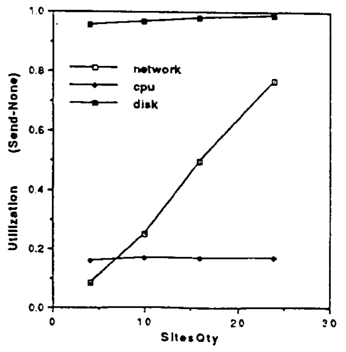


Figure 6: Utilization (*Send-None*)

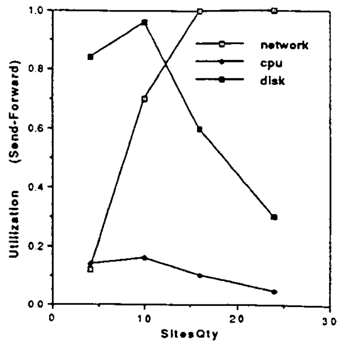


Figure 7: Utilization (*Send-Forward*)

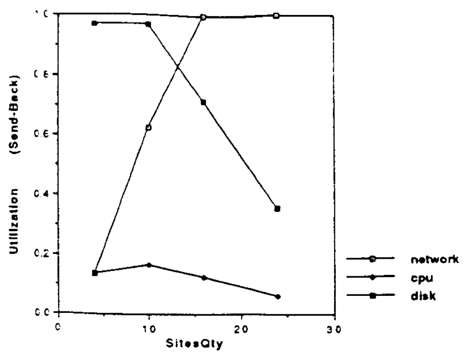


Figure 8: Utilization (*Send-Back*)

As it can be seen in Figure 6 disk utilization is high (> 0.9) under *Send-None* even when the number of sites is small. Since the disk and the CPU work together to process a disk access, the throughput of the CPU is limited by the throughput of the disk. Since the CPU service time is less than the disk service time (5 *ms* for CPU; 30 *ms* for disk), the CPU utilization is low for all cases. The communication network utilization for *Send-None* increases linearly with the increasing number of sites. The utilization of the resources under *Send-Forward* and *Send-Back* given in Figures 7 and 8 shows a completely different behavior. The utilization of the communication network increases faster with the number of sites. With the increase of the network's utilization we observe that the utilization of the disk decreases for both *Send-Forward* and *Send-Back*. This is explained by the fact that the system's bottleneck is migrated from the disk of each site to the communication network. Since the network is highly utilized, a queue of untransmitted messages builds up, thus keeping the disk idle. *Send-Back* has less communication overhead than *Send-Forward*. Therefore, the decrease of the disk's utilization due to untransmitted messages for increasing number of sites is slower. This explains the shorter mean response time of a query in *Send-Back* if more sites are added to the distributed system.

As a consequence from the first experiment, we conclude that *Send-Back* outperforms the other policies. *Send-Forward* shows good performance only for smaller number of sites. As demonstrated, the communication network is the bottleneck when *Send-Forward* or *Send-Back* are used.

5.2. Experiment II

In *Experiment I* we have seen for policies *Send-Forward* and *Send-Back* that the communication network with the given transmission capacity ($10\,Mbit/sec$) is not able to process the number of messages which is required if the partitioned global index scheme is used. Since improvements in communication technology will provide faster networks in the near future (up to $150\,Mbit/sec$ with optical fiber technology) it is a matter of high interest to study the presented query processing schemes for networks with 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 by:

$$S_{nw\_ref} = S_{nw\_adr} = nw\_speed \cdot \cdot (S_{nw\_setup} + (addresses\ to\ be\ transmitted) \cdot 0.003\ ms) \tag{6}$$

$$S_{nw\_data} = nw\_speed \cdot \cdot (S_{nw\_setup} + MAX(8,\ data\ records\ to\ be\ transmitted) \cdot 0.001 ms) \tag{7}$$

The results for the mean response time of a query are shown in Figure 9.

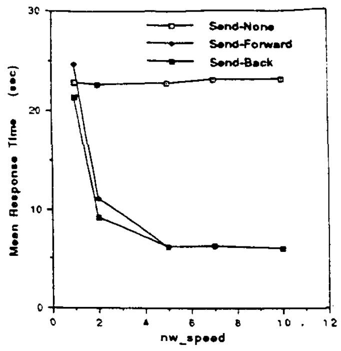


Figure 9: Mean Response Time

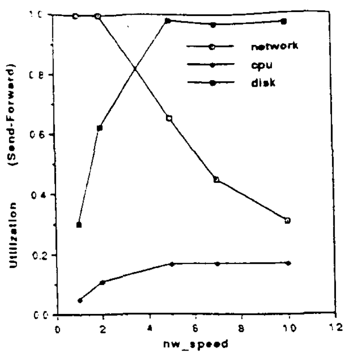


Figure 10: Utilization (*Send-Forward*)

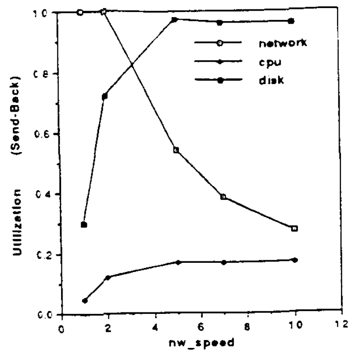


Figure 11: Utilization (*Send-Back*)

The mean response time for *Send-Forward* and *Send-Back* decreases fast if the transmission capabilities of the communication network are improved. For a network with a capacity of 50 *Mbit/sec* (*nw_speed* = 5) the mean response time of a query can be reduced to one third of the response time obtained for *Send-None*. Note that the values for *Send-Forward* and *Send-Back* do not improve for values *nw_speed* > 5. Note further that *Send-Forward* never performs better than *Send-Back*. For *Send-None* we observe that the mean response time is not affected by increasing the transmission capacity of the network. Since the system under *Send-None* is disk bound the result comes to no surprise. The performance of the system in this case is limited by the disk and increasing the network speed does not improve the response time of a query. The following Figures (Figures 10, 11) giving the utilization of the resources for *Send-Forward* and *Send-Back* explain why the mean response time for *Send-Forward* and *Send-Back* does not decrease beyond a certain threshold.

Summarizing we conclude that fast communication networks reduce the response time of a query significantly if the strategies developed for the partitioned global index a used.

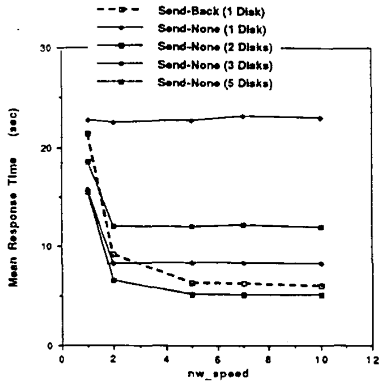
5.3. Experiment III.

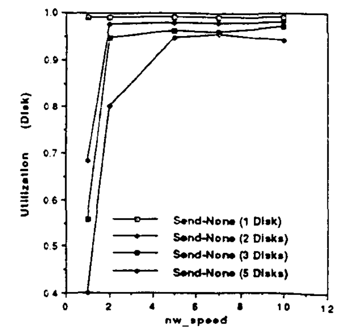
In the previous experiment we compared the performance of the query processing strategies for increased network capacity. Since *Send-None* was disk bound increasing the network speed did not improve the response time of a query. In this experiment we investigate solely the behavior of *Send-None* and show how the performance can be improved if disk drives are added to each site. We present results for 1, 2, 3 and 5 disk drives. As in *Experiment II* the parameter which 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 *Send-None*-policy can benefit from a faster communication network if the I/O-capabilities are improved. Figure 12 plots the results for the mean response time:

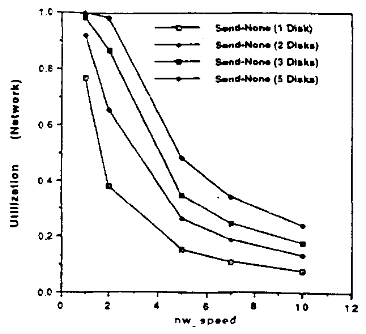
For comparison we included the results from *Send-Back* from *Experiment II* (dashed line). We see that a site with multiple disks benefits from an upgraded communication network. However, the response time does (relatively) not improve as much as for *Send-Back*. If the network speed is increased beyond a factor *nw_speed* = 5 the response time does not improve for either multiple disk system. Note that the speed-up until saturation is reached is approximately proportional to the number of sites added, i.e., with 2 Disks at each site we achieve a mean response time twice faster than with one Disk at each site, ... Figure 12 shows that for a network with *nw_speed* > 2 the *Send-Back*-policy gives a better mean response time than a partial index scheme with the *Send-None* even if the latter system has 3 disks available at each site. We now discuss the utilization of the critical resources disk and network, i.e., disk and network. Figures 13 and 14 show the utilization of the disk and the network for each system. Note that the values given for the disk refer to each single disk of a multiple disk station.

In Figure 13 we observe that if more disks are added to each site, the utilization of the disk drives becomes less. However, the utilization approaches a saturation point fast when the network speed is increased. The chart for the utilization of the network (Figure 14) shows the opposite behavior. For a low network speed the utilization of the network is high and decreases when the network speed is increased. Note that the utilization of the network becomes higher as more disks are added to the sites. Adding disks increases the processing power. If processing of queries is accelerated then the load on the network will be higher.

Comparing the results from Figures 10 (*Send-Forward*) and 11 (*Send-Back*) in *Experiment II* with Figures 13 and 14 we see that for a high speed network environment the utilization of the resources of *Send-Forward* (and *Send-Back*) having one disk at each site is about equal to the utilization of resources of *Send-None* having multiple disks at each site. Additionally, the response time of *Send-Forward* and *Send-Back*







(single disk) compared to *Send-None* (> 3 disks) is approximately the same. Since adding disk drives at each site involves considerable costs we conclude that for high speed networks the partitioned global index scheme with either policy *Send-Forward* or *Send-Back* is superior to the partial index scheme with *Send-None*.

6. Conclusions

We introduced a new indexing scheme called partitioned global indexes 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 two policies, *Send-Forward* and *Send-Back*, for processing such an index. In order to evaluate the performance of the new scheme we developed a simulation model. The simulation results were compared to the classical scheme, called partial indexes, in which corresponding index and data entries are stored at the same site. We referred to the query processing strategy of the partial index scheme as *Send-None*. The simulation experiments showed the tradeoffs between the new and the classical scheme. The results can be summarized as follows:

- Query processing strategies for a partitioned global index scheme in a distributed database system, i.e., *Send-Forward* and *Send-Back*, have the advantage of reducing the time spent to do index searches and, thus, reduce the workload on the disk. The amount of disk accesses required for index retrieval in the partial index scheme is considerably larger than in the new scheme. However, the developed policies for the partitioned global index suffer from a larger communication overhead. The communication overhead of *Send-Back* was shown to increase linearly if sites are added to the distributed database system, the overhead of *Send-Forward* increases faster than linearly. Therefore, if the bandwidth of the underlying communication network is small the partitioned global index scheme may not perform much better than the old scheme (*Experiment I*).
- If a communication network with more transmission capabilities is used, the processing time of a query can be reduced significantly under *Send-Forward* or *Send-Back* (*Experiment II*). 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 the partitioned global index scheme attractive for future use.
- The performance of the policy for the partial index, *Send-None*, scheme can be improved if each site in the distributed database system has multiple disk drives available. However, *Experiment III* demonstrated that in a high speed network environment a large number of disk drives need to be added to outperform the policies for the partitioned global index scheme having just a single disk drive.

References

1. Agrawal, P., Bitton, D., Guh, K., Liu, C. and Yu, C., "A Case Study for Distributed Query Processing," *Proc. of Int. Symposium on Databases in Parallel & Distributed Systems*, 1988, 124-130.

2. Bernstein, P., Goodman, N., Wong, E., Reeve, C. and Rothnie, J., "Query Processing in a System for Distributed Databases SDD-1," *ACM TODS, 6, 4, 1981*, 602-625.

3. Ceri, S. and Pelagatti, G., "Distributed Databases: Principles and Systems", *McGraw Hill, 1984.*

4. DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K. and Muralikrishna, M., "Gamma - A High Performance Dataflow Database Machine," *Proc. of VLDB Conference*, 1986, 228-237.

5. DeWitt, D., Ghandeharizadeh, S. and Schneider, D., "A Performance Analysis of the Gamma Database Machine," *Proc. of ACM SIGMOD Conference*, 1988, 350-360.

6. Epstein, R., Stonebraker, M. and Wong, E., "Distributed Query Processing in Relational Database Systems," *Proc. of ACM SIGMOD Conference*, 1978, 169-180.

7. Gardarin, G. and Valduriez, P., "Relational Databases and Knowledge Bases", *Addison Wesley, Reading*, MA, 1989.

8. Lafortune, S. and Wong, E., "A State Transition model for Distributed Query Processing," *ACM TODS*, 11, 3, 1986, 294-322.

9. Liebeherr, J., Omiecinski, E., and Akyildiz, I. F., "Index Partitioning Schemes for a Locally Distributed Database System", *Technical Report*, GIT-ICS-89-40, Georgia Institute of Technology, Oct. 1989.

10. Liebeherr, J., Omiecinski, E., and Akyildiz, I. F., "The Effect of Index Partititoning Schemes on the Performance of Distributed Query Processing", *Technical Report*, GIT-ICS-90-20, Georgia Institute of Technology, April 1990.

11. Lohman, G., Mohan, C., Haas, L., Lindsay, B., Selinger, P., Wilms, P. and Daniels, D., "Query Processing in R*," *Query Processing in Database Systems*, eds., Kim, W., Batory, D. and Reiner, D., Springer Verlag, 1985, 31-47.

12. Lu, H. and Carey, M., "Some Experimental Results on Distributed Join Algorithms in a Local Network," *Proc. of VLDB Conference*, 1985, 292-304.

12. Mackert, L. and Lohman, G., "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proc. of VLDB Conference*, 1986, 149-159.

13. Perrizo, W., Lin, J. and Hoffman, W., "Algorithms for Distributed Query Processing in Broadcast Local Area Networks," *IEEE TKDE, 1, 2*, 1989, 215-225.

14. Sauer, C. H., MacNair, E. A., Kurose, J. F., "The Research Queueing Package Version 2," *IBM Thomas J. Watson Research Center*, Yorktown Heights, New York 10598.

15. Segev, A., "Optimization of Join Operations in Horizontally Partitioned Database Systems," *ACM TODS*, 11, 1, 1986, 48-80.

16. Stonebraker, M., Katz, R., Patterson, D. and Ousterhout, J., "The Design of XPRS," *Proc. of VLDB Conference*, 1988, 318-330.

17. Wang, X. and Luk, W., "Parallel Join Algorithms on a Network of Workstations," *Proc. of Int. Symposium on Databases in Parallel & Distributed Systems*, 1988, 87-95.

18. Yoo, H. and Lafortune, S., "An Intelligent Search Method for Query Optimization by Semijoins," *IEEE TKDE, 1, 2*, 1989, 226-237.

19. Yu, C., Guh, K., Zhang, W., Templeton, M., Brill, D. and Chen, A., "Algorithms to Process Distributed Queries in Fast Local Networks," *IEEE TC*, C-36, 10, 1987, 1153-1164.