Performance of Catalog Management Schemes for Running Access Modules in a Locally Distributed Database System

Eui Kyeong Hong

Department of Computer Science and Statistics, Seoul City University
Dongdaemoon-Ku, Jeonnong-Dong 8-3, Seoul 130-743, Korea
(also with Center for Artificial Intelligence Research of KAIST)

Abstract

Catalog management schemes affect many aspects of distributed database systems such as site autonomy, query optimization, view management, authorization mechanism, and data distribution transparency. However, the performance comparison of various catalog management schemes has received relatively little attention. Embedded *read queries* to the catalogs in a form of data manipulation statements are assumed to be compiled into an access module, since the module is executed repeatedly with different parameters. *Update queries* to the catalogs are assumed to be interpreted due to their interactive nature. The performance of the catalog management schemes measured in terms of running access modules is investigated using simulation approach in a locally distributed database system. The three alternatives studied include the centralized catalogs, the fully replicated catalogs, and the partitioned catalogs.

1. Introduction

In the literature, most researches in distributed database systems have been concentrated on query optimization, concurrency control, recovery, and deadlock handling.

Although catalog management schemes are of great practical importance with respect to the site autonomy [14], query optimization [15], view management [1], authorization mechanism [22], and data distribution transparency [13], the performance comparison of various catalog management schemes has received relatively little attention [3, 18]. Even well-known distributed database management system (DDBMS) prototypes employ distinct catalog management schemes with one another. For example, SDD-1 [20] and Distributed Ingres [21] use the fully replicated catalogs, whereas R^* [13] uses the partitioned catalogs.

Catalogs are regarded as the system databases that contain information concerning various objects that are of interest to the database system itself. Relations, views, indexes, users, access modules (executable codes), and access privileges are examples of such objects. Catalogs map the user-specified objects in a query onto low-level object identifiers and object locations. Catalogs contain the definition of the schemas for database objects and the available access paths, e.g., indexes. They also supply statistics used in query optimization, and record the authorization of users to database objects as well as dependency information among database objects, for example, relationships of views to relations and of programs to indexes [13]. In a distributed database system environment, catalogs themselves become a distributed database, which should be distributed and managed efficiently.

One important design problem of computer systems is that of assigning files to possibly different sites in a computer network for query/update/execution purposes; this is commonly known as the *file allocation problem* [6]. The research results for the file allocation problem are not suitable to be applied to the catalog management schemes. The first reason is that queries may request data that reside in multiple files, and the second reason is that catalogs are, unlike user relations, always referenced whenever a query needs to be processed.

Cornell and Yu [4] addressed the relation assignment problem taking advantage of the knowledge on transaction characteristics and arrival frequencies to each site so that hardware resource requirement can be balanced. The research results for the relation assignment problem do not help due to the second reason mentioned earlier. In addition, catalogs are not used to join with other catalogs or user relations, since the contents of the catalogs only serve as metadata.

Chu [3] analyzed the performance of the different directory placement strategies in distributed databases by forming mathematical models. Chu considered communication cost, translation cost, and storage cost, but

**Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993**

unfortunately, queuing delay, concurrency control, and two-phase commit protocol are neglected. Matsushita et al. [18] improved the performance comparison of the three types of directory management schemes with concurrency control in mind by forming mathematical models. However, [18] did not consider I/O cost, CPU cost, two-phase commit protocol, and queuing delay. Moreover, [18] did not consider the partitioned catalogs and the centralized catalogs. In addition, in [3] and [18], a directory only means a listing of files available to the users of the networks, not catalogs used by queries in real distributed database systems.

The performance of various catalog architectures can be investigated from two viewpoints: query compilation (access module generation) and running access module. The performance of catalog management schemes from the viewpoint of access module generation has been addressed in [8, 9, 10]. Access module currently under execution is kept in the main memory and the corresponding catalog entries are locked in a shared mode to prevent any operation that might invalidate the access module, such as dropping an index, a privilege, a relation or view used by the access module. This means that the shared lock is maintained longer period, since query execution usually takes longer time than query compilation. The performance of queries, in particular, *update queries* to the catalogs, will be affected by the longer duration of holding the shared lock. In addition, the number of CPU cycles, disk accesses, and communication bandwidth needed by each access module also will be increased to a certain degree.

This paper reports on the simulation study of the performance of the three catalog management schemes – the centralized catalogs, the fully replicated catalogs, and the partitioned catalogs – from another viewpoint of running access module with CPU cost, I/O cost, communication cost, concurrency control, queuing delay, and two-phase commit protocol in mind for a locally distributed database system. The simulation results in a geographically distributed database system is described in [10].

The organization of the remainder of this paper is as follows. Section 2 presents the classification of queries according to the access pattern to the catalogs, catalog names and its functions, and query processing steps. Section 3 describes the alternative catalog management schemes that are examined in this paper. Section 4 presents the detailed simulation model and simulation parameters, and then performance results obtained are interpreted. Finally, conclusions appear in Section 5.

2. Classification of Queries

User queries usually access the catalogs prior to the access to the database regardless whether they are embedded queries or ad hoc ones. Conventionally, user queries have been classified into the read or update queries depending either on reading the database or on updating the database. In this paper, we use different semantics for the definition of read query and update

Table 2.1: Catalog names and functions.

Catalog Name	Function
SYSCATALOG	Indicates who created the relation and other statistics about the data in each relation. These statistics are used during access path optimization.
SYSCOLUMNS	Indicates the data type and length of each attribute.
SYSACCESS	Records the access modules created for user programs.
SYSINDEXES	Describes each index currently in the databse.
SYSSYNONYMS	Describes all synonyms currently in effect and indicates who defined them.
SYSTABAUTH	Records the access privileges owned by users.
SYSUSAGE	Records the dependencies that access modules have on database objects

query, since we are interested in evaluating the performance of catalog management schemes. A *read query* is defined to mean a query that causes only read accesses to catalogs. Note that the *read query* may include not only the conventional read query, but also the *update query* to the database. Analogously, the term *update query* is defined from the viewpoint of the update accesses to the catalogs. Catalog updating means the insertion or deletion of entries to or from the related catalogs. In SQL syntax [11], according to the classification described above, SELECT, INSERT, DELETE, and UPDATE commands are *read queries*. That is, *read queries* consist of data manipulation statements. *Update queries* include the statements such as CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, REVOKE, CREATE SYNONYM, DROP SYNONYM, and UPDATE STATISTICS statements, i.e., data definition and control statements.

The catalogs are not same across distinct DBMSs, since the catalogs for a particular DBMS necessarily contain information that is interest to that DBMS. We believe the seven catalogs in Table 2.1 are the minimum which are needed for any distributed DBMS or centralized DBMS. We take the catalog names from SQL/DS [11]. Each of these catalogs has an index on the appropriate key values, so each access to catalog requires at most 2 I/Os, and after this first access only one I/O is required to get the catalog because the index of the catalog is already in the buffer. Also, some of the catalogs are linked to facilitate the direct access [16].

We postulate that *read queries* are used as embedded queries that are executed repeatedly with different parameters. That is, they are used in an application program. Each *read query* is compiled into a sequence of access structures, stored as a ***section*** of an access module for the application program containing the query. Each site storing the user relations referenced in a query stores in its access module the access structures to perform its portion(s) of the work for the query, called ***subsections*** [15].

An access module is generated by the seven steps: parsing, name resolution, catalog lookup, authorization checking, optimization, access module generation, and dependency recording. More details can be found in [15].

Update queries are assumed to be interpreted rather than compiled. That is, no optimization, access module generation, and dependency recording are performed [23]. Therefore, an *update query* is processed through five steps: parsing, name resolution, catalog lookup, authorization checking, and updating of related catalogs.

3. Catalog Management Schemes

Catalogs can be allocated in distributed databases in many different ways. The best allocation scheme varies with the database and the particular access pattern. The three basic alternatives are the centralized catalogs, the fully replicated catalogs, and the partitioned catalogs.

3.1 Centralized Catalogs (CC)

The whole catalogs are stored exactly at a single central site. The centralized catalogs may not enhance the advantages of distributed database systems, such as high reliability and availability, and the distribution of the processing load.

For a *read query* submitted from the central site, the access module is generated locally, and then distributed to the sites which store the relations referenced by the query. Dependency information is stored at the central site. During this time, the catalog entries corresponding user relations are locked in a shared mode at the central site. When a *read query* is submitted from any non-central site, the corresponding catalog entries are locked in a shared mode at the central site, and fetched to the requesting non-central site. When the access module is generated, it is stored at the relevant sites which store the relations referenced by the query. Dependency information is always recorded at the central site. Then shared locks are released at the central site. Acknowledgement messages are returned from the central site and the sites which store the relations referenced by the query to the requesting site.

An *update query* submitted from the central site can be processed only at the central site. During this time, corresponding catalog entries are locked in an exclusive mode. When an *update query* is requested from a non-central site, it is sent to the central site and processed there. An acknowledgement message is returned to the requesting site.

Stored access modules in the catalogs may be executed repeatedly with different parameters. Whenever a request is submitted from a terminal to run the stored access module, the associated access module is read from the catalog and kept in the main memory of the coordinator site (query site). If this site is the central site, the corresponding catalog entries are locked in a shared mode to prevent any operation that might invalidate the access module, such as dropping an index, a privilege, or a relation, used by the access module. If this site is not the central site, the lock request should be sent to the central site to lock the corresponding catalog entries in a shared mode. When the acknowledgement message indicating that the locks are granted is returned to the coordinator site, the remote subsections are invoked and executed.

3.2 Fully Replicated Catalogs with Unanimous Agreement (FRCUA)

The total catalogs are stored entirely at every site. This catalog architecture enhances locality of reference by satisfying more *read queries* locally. This reduces the response time for a *read query*, and reduces traffic on the communication network. A major restriction to using replication is that replicated copies must behave like a single copy. SDD-1 [20] and Distributed Ingres [21] use the fully replicated catalogs.

In this paper, unanimous agreement method (read-locks-one, write-locks-all) is taken to preserve the consistency of the fully replicated catalogs. To read catalogs, it suffices to set a read lock on any copy of the catalogs, so the local copy is locked; to update catalogs, write locks are required on all copies. The *update queries* are blocked until all of the replicas of the catalogs to be updated have been successfully locked. All locks are held until the transaction has successfully committed or aborted.

Fully Replicated Catalogs with Quorum Consensus (FRCQC) [5] is not considered in this paper, since this scheme shows the worst performance in almost all simulated situations [9]. Worse performance of FRCQC than FRCUA can be explained as follows: since a *read query* in FRCQC must collect a read quorum of r votes, it needs to access remote sites (randomly selected to be included in a read quorum) to retrieve catalog entries. *Read queries* enter several queues and use several physical resources at all sites which are the members of a read quorum. Hence the response time of *read queries* in FRCQC is worse than that of FRCUA. Furthermore, *read queries* in FRCQC increase the queuing delay of *update queries* in several queues, making the response time of *update queries* in FRCQC also higher than that of FRCUA. In addition, FRCQC requires too many messages for the case of *update queries* which delete the catalog entries [5].

An access module of *read query* requested from any site is generated locally, and then sent to the relevant sites which store the relations referenced by the query, and stored there. Dependency information also is stored at the same sites. During this time, corresponding catalog entries are locked in a shared mode at the local site.

An *update query* submitted from any site is broadcast to all sites, and updates of catalogs are performed at all sites. To treat the processing of an *update query* as a transaction, the centralized two-phase commit protocol [7] is employed. For *update queries*, deadlocks may occur when the write-locks-all is used. We think that any deadlock handling method would result in similar performance, since deadlocks occurred very infrequently in several our experiments. Thus we choose the deadlock prevention method based on distributed wound-wait locking algorithm [19].

Stored access modules in the catalogs may be executed repeatedly too. They are executed in the same way as in the case of CC, except that the required locks of access modules are set (in a shared mode) only at the coordinator site, since catalogs are fully replicated.

3.3 Partitioned Catalogs (PC)

Each site maintains its own catalogs for objects stored at that site. This scheme preserves individual site autonomy by avoiding centralized or global catalogs, and by storing catalog entries at the site where the object was created and at the site where it is stored. To improve performance without increasing complexity or compromising site independence, any site is allowed to cache any catalog entry which it has retrieved from the remote sites. A query referencing the same site object may use the locally cached catalog information. Locally cached copies of the remote catalog information are not kept up-to-date. This catalog management scheme is used in R^* [13, 15].

Update queries are implemented in R^* by a distributed recursive call to a single, common routine at the apprentice remote site. The call passes only the query statement and the userid (for authorization), and returns a completion code [23].

Stored access modules in the catalogs may be executed repeatedly too. They are executed in the same way as in the case of CC, except that the required locks of access modules are set (in a shared mode) at each apprentice site, since catalogs are partitioned. When the locks are granted, the subsections stored at the apprentice sites are read from the catalog and executed under the control of the coordinator site.

The partitioned catalogs can be divided further into three alternatives according to the caching method: no caching, incremental caching, and full caching [9]. From the viewpoint of query compilation (access module generation), three alternatives of partitioned catalogs show some performance differences [9]. However, *update queries* and stored access modules of *read queries* in the three partitioned catalogs alternatives are processed in the same manner. Thus, we do not further divide the partitioned catalogs according to the caching method.

4. Performance Study

Lu [17] measured the performance of eight different distributed two-way join methods in an experimental locally distributed computer system. He concluded that pipelined semijoin method is found to be preferable over a wide range of join queries. Since this paper intends to investigate the performance of catalog management schemes when the stored access modules are executed, rather than presenting yet another performance comparison of existing or new query processing methods, the pipelined semijoin method is employed in this study.

The primary performance index used throughout the paper is the response time. Response times are measured as the difference between when a terminal first submits a new query and when the query returns to the terminal following its successful completion. All response times are given in seconds. Several secondary performance indexes are used in analyzing the results of the experiments. Queuing delays measured in the queues (represented in Figure 4.1) are given in some cases. The relative importance of several costs are also given in some cases.

4.1 Simulation Background

For the simulation study, some assumptions are made as follows. These assumptions are believed to be reasonable, since these approaches are used in real prototypes, or their performance has been demonstrated good, or at the least they are simple to implement.

(1) An application program (access module) contains a *read query*.

(2) Concurrency control using the two-phase locking algorithm [7] is chosen for centralized catalogs and the partitioned catalogs.

(3) For *update queries* in the fully replicated catalogs, deadlocks may occur when the unanimous agreement (read-locks-one, write-locks-all) is used. The deadlock prevention method based on distributed wound-wait locking algorithm [19] is chosen.

(4) The centralized two-phase commit protocol [7] is chosen.

(5) Record-level locking is selected.

4.2 Simulation Model

The simulation model of distributed database system consists of a set of database sites (DB sites) connected by a high-speed local area network. In this study, the local area network is assumed to be a broadcast network. Figure 4.1 shows the detailed model of a DB site in the distributed database system. This model is an extended version of the model of Lu [17] and Carey [2]. The model of Lu has no concurrency control manager, since it is developed to investigate the

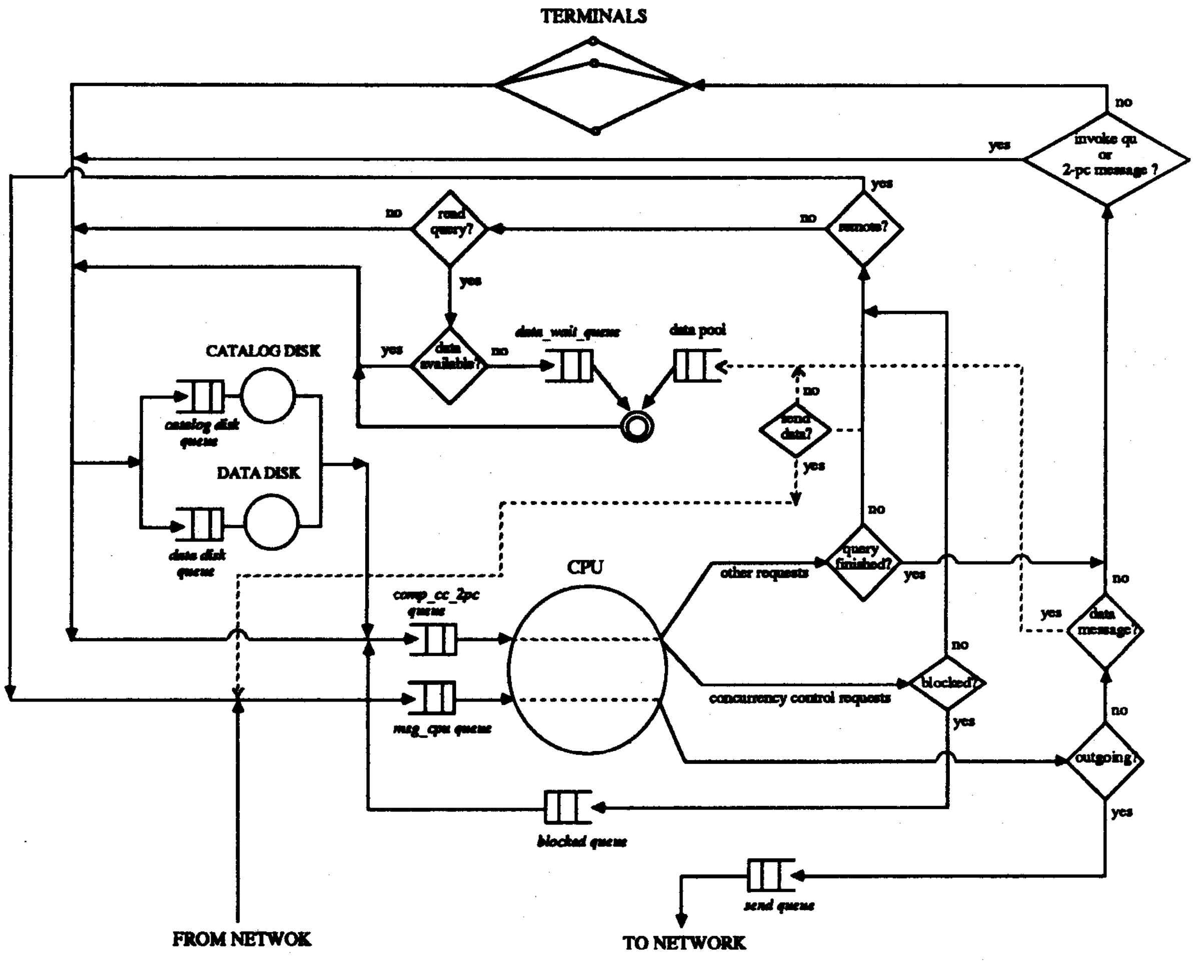

Figure 4.1: DB site model.

performance of only read query processing. Carey's model, on the other hand, captures the components related to concurrency control. Each DB site includes both terminals (users) and physical resources for storing and processing the data and messages such as CPU, disks, and communication network. Queries originate from terminals and to which the results are returned. Disks in a DB site are classified into two types: catalog disks for storing catalogs, and data disks for storing user relations. Several queues also are represented in the DB site model. The CPU serves requests from two queues:

(1) *comp_cc_2pc queue* for query interpretation steps (for *update queries*), running access module (for *read queries*), concurrency control, and two-phase commit processing.

(2) *msg_cpu queue* for message processing.

A query enters the *comp_cc_2pc* queue whenever query interpretation steps, concurrency control, two-phase commit processing, and execution of an access module need to consume CPU time, whereas a query enters the *catalog disk queue* or *data disk queue* in order to read/update catalogs or user relations from/to disk, or write two-phase log record to disk. If the result of a concurrency control request is that the query must be suspended, it enters the *blocked queue* until it is able to proceed. In addition to the use of the communication lines, the messages also consume some CPU time. Hence all incoming and outgoing messages enter the *msg_cpu queue* to receive CPU service. The outgoing messages then enter the *send queue* to be served by the network. Two queues connected to CPU are assumed to be served alternatively. Each of disk has its own queue. Requests in the *catalog disk queue*, *data disk queue*, and *send queue* in a DB site are served in FCFS (First-Come, First-Served), whereas two CPU queues are served in a round-robin fashion.

Each catalog architecture has been implemented as a simulation program using the concurrent simulation language Path Pascal [12]. Path Pascal provides features for parallel processing of processes.

The various components of the model are used by *read queries* as follows [17]. To model pipelined semijoin query execution, the model includes the pipelined flow of data and the possibility that a *read query* may have to block awaiting the arrival of data page. The dotted lines represent the flow of intermediate results generated during access module execution. We assume that a stored access module of a *read query* is represented as a sequence of query units. The difference between subsection described in Section 2.3 and query unit is as follows. The subsection performs its portion of the work for the access module. Thus, a subsection may access more than one relations stored at one site. On the other hand, each query unit accesses at most one relation, and passes accumulated data to the next query unit in the access module. In this paper, therefore, a subsection may constitute more than or equal to one query unit. Even if the coordinator site does not store any user relation referenced by the access module, the access structure in the coordinator site is assumed to be a query unit. In this case, the query unit will simply receive the data transferred from remote site(s).

When a stored access module needs to access at least one remote site, the remote query unit is called. Before accessing user relations, corresponding catalog entries are locked in a shared mode at related site(s) according to the catalog management schemes. The first query unit in the pipelined semijoin begins execution by accessing the outer relation. The other query units will enter the *data_wait_queue*, and start their processing after receiving the first data page from their predecessors in the pipeline. For each data page received, a query unit will cycle through CPU and data disks several times. Upon finishing this process, the query unit then checks the local *data pool*. If the next data page is already in the *data pool*, the query unit gets the data page and continues executing. Otherwise, the query unit will block awaiting the arrival of the next input data page. If its predecessors have completed and at least one predecessor has been processed at a remote site and this query unit is the last one in the pipeline, then the centralized two-phase commit protocol is initiated by the coordinator site (query site). Note that a *read query* in this paper may include not only the conventional read query, but also the update query to the database (user relations). When all of the predecessors have completed and they are located at the same site and this query unit is the last one in the pipeline, the query unit will terminate without the two-phase commit protocol. After each cycle through CPU and data disks, a query unit produces a certain amount of intermediate result data. When one page of result data has been accumulated, the data page will either be sent to a remote site via the communication link or placed in the local *data pool* depending on the processing site of the successor in the pipeline. If an incoming message is a data message, it is directed to the data pool. Otherwise it is routed to the CPU.

Update queries are processed by accessing the catalog disk without need to access the data disks at all. This detailed model seems to be sufficient to investigate the performance of catalog management schemes in run time case.

4.3 Simulation Parameters

Table 4.1 lists the model associated simulation parameters used in this paper. We will describe several simulation parameters which need some additional explanation. *rq_ratio* parameter means the ratio of the number of *read queries* to the sum of the number of *read* and *update queries*, while *local_ratio* is defined as the ratio of the number of queries which access the local objects only to the number of total queries. Here, queries include both *read queries* and *update queries*. Each site has two types of disks: one for storing catalogs (catalog disk), and the other for storing user relations (data disk). *num_cata_disks* specifies the number of catalog disks, while *num_data_disks* gives the number of data disks in a DB site. *num_reads* specifies the mean number of cycles through CPU and data disks for the query unit [17]. For the first query unit, this gives the number of data pages it reads from the data disks. For the second query unit, *num_reads* specifies the mean number of processing cycles corresponding to the receipt of one data page from the predecessor query unit in the pipeline, since its execution is dependent upon the intermediate result data from the predecessor. *result_frac* specifies the fraction of a result page generated by each page processed.

Table 4.2 gives the values of the simulation parameters. The system consists of from 3 to 15 sites. The number of terminals at each site is varied from 5 to 40. Both the read query ratio and local query ratio are varied from 0 to 1. *num_reads* is set to 20 pages for the first query unit, and 5 pages for the second query unit. With a *result_frac* of 0.2, the mean number of result data pages for the first query unit is 4. Thus the total number of reads for the second query unit is also 20 (= 4 * 5). Broadcast on the local area network is assumed to have the effective transfer rate of 4M bits/sec, whereas its nominal transfer rate is 24M bits/sec (3M bytes/sec). We assume that a given *read query* has a 50 percent chance of referencing two relations, and a 50 percent opportunity of using one relation. Also, 50 percent of remote *read queries* reference relations at two remote sites, and 50 percent of remote *read queries* access relations at one remote site. The relation collection (catalog entries) accessed by a query (*read* or *update*) is randomly chosen from the associated catalog entries.

Among the parameters in Table 4.2, *num_sites*, *num_terms*, *rq_ratio*, and *local_ratio* are varied over some ranges. When their fixed values are referenced in the experiments *num_sites* = 5, *num_terms* = 15,

Table 4.1: Simulation parameters.

Parameter Name	Description
num_sites	number of sites in a distributed database system
num_terms	number of terminals in a DB site
think_time	mean think time for the terminals
rq_ratio	read query ratio
local_ratio	local query ratio
num_rels	number of user relations in a DB site
disk_time	mean access time for a disk page
page_size	size of a disk page
num_cata_disks	number of disks storing catalogs
num_data_disks	number of disks storing user relations
msg_cpu_time	per-message cost such as message packing and task switching
msg_byte	per-byte cost of message needed for transferring one byte over the local area network
query_msg_size	size of a query message in bytes
data_msg_size	size of a data message in bytes
st_msg_size	size of short messages such as *prepare*, and *ready* in bytes
lock_time	CPU time needed to perform concurrency control, and deadlock handling (only for FRCUA)
num_reads	mean number of cycles through CPU and disks for query unit
result_frac	fraction of a result data page generated be each page processing
page_cpu_time	CPU time needed for processing a data page
parse_cpu	CPU time needed for parsing per relation
name_cpu	CPU time needed for name resolution per relation
cata_cpu	CPU time needed for catalog lookup per relation
auth_cpu	CPU time needed for authorization checking per relation
name_io	number of I/Os needed for name resolution per relation
cata_io	number of I/Os needed for catalog lookup per relation
auth_io	number of I/Os needed for authorization checking per relation
amodule_io	number of I/Os needed for reading an access module
avg_wrt_io	mean number of I/Os which an *update query* writes

Table 4.2: Parameters settings for the simulation.

Parameter Name	Range
num_sites	3 - 15 sites
num_terms	5 - 40 terminals
think_time	3 - 21 seconds
rq_ratio	0.0 - 1.0
local_ratio	0.0 - 1.0
num_rels	100 relations
disk_time	25 milliseconds
page_size	4096 bytes
num_cata_disks	1 disk
num_data_disks	2 disks
msg_cpu_time	16 milliseconds
msg_byte	0.002 milliseconds
query_msg_size	2048 bytes
data_msg_size	4096 bytes
st_msg_size	1024 bytes
lock_time	1 millisecond
num_reads	20, 5
result_frac	0.2, 0.2
page_cpu_time	8 milliseconds
parse_cpu	10 milliseconds
name_cpu	2 milliseconds
cata_cpu	5 milliseconds
auth_cpu	2 milliseconds
name_io	1
cata_io	3
auth_io	1
amodule_io	1
avg_wrt_io	2

rq_ratio = 0.8, and *local_ratio* = 0.8 are implied. *think_time*, *num_reads* and *page_cpu_time* parameters are exponentially distributed. Other parameters represent constant service time requirements rather than stochastic ones for simplicity.

4.4 Experiments and Results

In this section, we present the performance of the three catalog architecture alternatives – the centralized catalogs, the fully replicated catalogs with unanimous agreement, and the partitioned catalogs – under various simulation conditions. Four experiments are performed using a detailed simulation model, and their performance is discussed and analyzed.

4.4.1 Experiment 1: Varying Number of Sites

Our first experiment examines the impact of the varying the number of sites on the performance of the three catalog architecture alternatives. The number of sites is varied between 3 and 15. Figures 4.2 and 4.3, respectively, show the response time results obtained for *read queries* and *update queries*.

For *read queries*, Figure 4.2 indicates that PC performs the best, followed by FRCUA, followed by CC. As the number of sites increases, the performance of CC for *read queries* degrades significantly. The explanation lies in the fact that even if the central site does not store any user relation referenced by a *read query*, the *read query* should access the central site to lock the associated catalog entries in a shared mode, leading to the congestion at the central site. The performance of FRCUA for *read queries* is little affected by the increase of the number of sites compared to the query compilation case [9]. This is due to the fact that each *read query* accesses catalog disk once to fetch the stored access module, and then cycles through CPU and data disks several times to execute the access module. Consequently, the queuing delay in the *catalog disk queue* decreases considerably compared to the query compilation case where no data disks are needed. The performance of PC for both *read* and *update queries* is relatively insensitive to the increase of the number of sites.

The relative ordering that we obtained for *update queries*, however, is different than that of *read queries*. The ordering between FRCUA and CC is reversed. Figure 4.3 shows that CC outperforms FRCUA, but it performs worse than PC. Since updates need to be performed at all sites, the performance of FRCUA for *update queries* is the worst. For FRCUA, the centralized two-phase commit protocol is also required to treat an *update query* as a transaction. On the other hand, any *update query* in CC and PC can be processed without the two-phase commit protocol. This is due to the fact that in CC the updates to the catalogs are performed at the central site which stores entire catalogs, and in PC the updates to the catalogs can be performed at one site (local or remote site) where the catalog information to be updated is currently stored. As the number of sites increases, the response time of CC for *update queries* also increases, since the updates to the catalogs are performed at the central site which stores entire catalogs, resulting in the increase of queuing delays. Another minor factor is as follows. Read queries referencing remote user relations need to access the remote sites. As the number of sites increases, the probability that the central site is not included in the collection of the remote sites accessed by a *read query* also increases. As described before, the central site should be accessed by any *read query* to lock the associated catalog entries in a shared mode. Thus requirements of *read queries* to access the central site are increased as the number of sites increases, causing CC to experience more queuing delays in the *msg_cpu queue* and *catalog disk queue* at the central site.

4.4.2 Experiment 2: Varying Number of Terminals

Experiment 2 is conducted to see how the load increase on each site affects the performance of the three catalog management schemes. The level of resource and catalog entries contention is controlled here by varying the number of terminals. The number of terminals is varied from 5 to 40. Figure 4.4 shows the response time of *read queries*, and Figure 4.5 is for the *update queries*.

For *read queries*, PC has slightly better performance than CC, and FRCUA in turn performs marginally better than PC. Minor differences among the catalog management schemes are due to the fact that any *read query* in CC needs to access the central site to lock catalog entries in a shared mode to prevent any operation that might invalidate the access modules, even when the central site does not store any user relations accessed by the *read query*. For *update queries*, the ordering of the performance of CC and FRCUA is reversed because of the same reasons described in Experiment 1. As a result, FRCUA presents the worst performance, followed by CC, and followed by PC.

Tables 4.3 and 4.4, respectively, give the relative weight of CPU cost, disk cost, communication cost, and queuing delay (accumulated in all queues) of *read* and *update queries* when the number of terminals is 40. In fact, queuing delay has the highest weight for two query types, and for all the catalog management schemes examined. Tables 4.5 and 4.6 summarize the relative ratio of each queuing delay (associated with physical resources in Figure 4.1) of *read* and *update queries*, respectively, when the number of terminals is 40. *Read queries* have the highest queuing delay in the *data disk queue*, whereas *update queries* have the highest weight in the *blocked queue*. As anticipated, queuing delay in the *send queue* turns out to be negligible for both *read queries* and *update queries*. Note that *update queries* do not access data disks at all.

Table 4.3: Relative importance of *read queries* (*num_terms* = 40).

	CC	FRCUA	PC
CPU cost	5.8%	4.8%	4.8%
disk cost	10.2%	10.6%	10.5%
communication cost	0.2%	0.2%	0.2%
queuing delay	83.8%	84.5%	84.6%

Table 4.4: Relative importance of *update queries* (*num_terms* = 40).

	CC	FRCUA	PC
CPU cost	2.6%	8.0%	1.7%
disk cost	6.2%	14.5%	8.0%
communication cost	0.2%	0.3%	0.1%
queuing delay	91.1%	77.1%	90.3%

Table 4.5: Relative importance of queuing delays of *read queries* (*num_terms* = 40).

	CC	FRCUA	PC
comp_cc_2pc queue	15.1%	12.6%	12.1%
msg_cpu queue	4.7%	0.9%	0.3%
blocked queue	3.3%	2.8%	1.9%
catalog disk queue	0.3%	1.8%	0.2%
data disk queue	76.6%	81.9%	85.5%
send queue	0.0%	0.0%	0.0%

Table 4.6: Relative importance of queuing delays of *update queries* (*num_terms* = 40).

	CC	FRCUA	PC
comp_cc_2pc queue	12.2%	23.0%	8.4%
msg_cpu queue	4.2%	8.5%	0.1%
blocked queue	74.8%	38.4%	89.7%
catalog disk queue	8.7%	30.1%	1.7%
data disk queue	0.0%	0.0%	0.0%
send queue	0.0%	0.0%	0.0%

Figure 4.6 shows the associated queuing delay of *read queries* in the *data disk queue*. For *update queries*, Figure 4.7 presents the associated queuing delay in the *blocked queue*. The unit used here is milliseconds, whereas the unit used in the response time curves is second. In CC, the queuing delay in the *blocked queue* is obtained from the central site, because total catalogs are stored only at the central site. Similarly, the queuing delay of PC for *update queries* (Figure 4.7) in the *blocked queue* is obtained from one site, where the catalog information to be updated is currently stored. On the other hand, the curve of FRCUA for *update queries* (Figure 4.7) is obtained by dividing the total queuing delay in the blocked queue from all sites by the number of sites. Other queuing delays represent the average values obtained from the related sites, even though the amount of resources used may differ, and thus the amount of queuing delays incurred may differ between the coordinator site and apprentice site(s). In addition, query processing may show some serial execution pattern between the coordinator site and apprentice site(s).

4.4.3 Experiment 3: Different Read Query Ratio

To investigate the effects of the read query ratio on the performance, this experiment is conducted in a way that the read query ratio is varied from 0 to 1. The read query ratio becomes 0 when the number of *read queries* is 0. Reversely, the read query ratio becomes 1 when all queries generated are *read queries*.

Figures 4.8 and 4.9 show the response time results of both *read* and *update queries* for the alternatives of the catalog architecture. For low read query ratio, FRCUA for *read queries* has inferior performance to CC. However, as the read query ratio increases, the curves are reversed, with FRCUA providing better performance than CC. In Figure 4.8, PC outperforms other catalog management schemes for all read query ratio ranges. Even when the read query ratio is 1, where all queries generated by terminals are *read queries*, CC has the highest response time among all of the catalog management schemes. This is due to the fact that any *read query* in CC still needs to access the central site to lock the associated catalog entries in a shared mode even when the central site has no user relations referenced by this *read query*. The improvement of response time of FRCUA for *read queries* for low read query ratio range is mainly attributed to the decrease of queuing delay in the *catalog disk queue* as the read query ratio increases. As the read query ratio increases, queuing delay in the *data disk queue* increases since more accesses to data disks are needed by query units in the access modules of *read queries*. Thus the performance of all of the catalog management schemes degrades, as the read query ratio is increased.

Figure 4.9 shows that PC for *update queries* is insensitive to the different read query ratio, and has the best performance among the three catalog management schemes. CC follows PC, and FRCUA has the worst performance. The response time results of FRCUA and CC are significantly improved as the read query ratio increases. The explanation lies in the fact that queuing delay in the *catalog disk queue* is considerably decreased. In CC, all *update queries* are processed at the catalog disk of the central site where whole catalogs are stored, and in FRCUA all *update queries* need to access the catalog disk at all sites. In addition, FRCUA needs the centralized two-phase commit protocol.

4.4.4 Experiment 4: Different Local Query Ratio

Finally, Experiment 4 is performed to identify how the performance of the catalog management schemes behaves when the local query ratio is varied. The local query ratio is varied from 0 to 1. Local query ratio of 1 means that all queries access the only relations stored at local site. Here, queries include both *read queries* and *update queries*.

Figures 4.10 and 4.11 summarize the response times of *read* and *update queries*. For *read queries*, CC provides the worst performance, since all *read queries* generated from any site need to access the central site to lock the associated catalog entries for all local query ratio ranges. Thus, even when the local query ratio is 1 where all queries reference only local objects, CC is still noticeably worse than FRCUA and PC in its performance. Figure 4.10 shows that the performance of all of the catalog management schemes for *read queries*

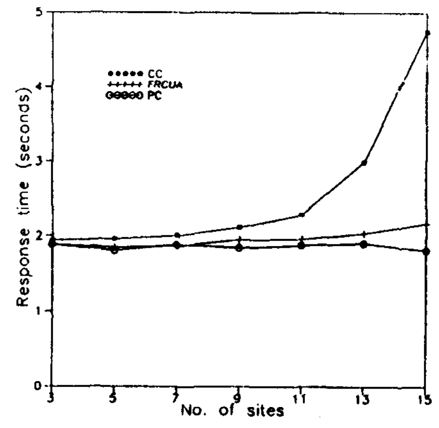

Figure 4.2: Response time of *read queries*.

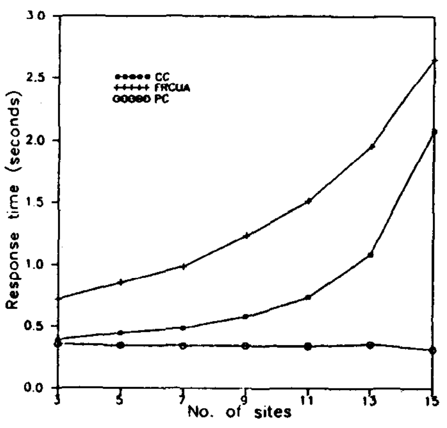

Figure 4.3: Response time of *update queries*.

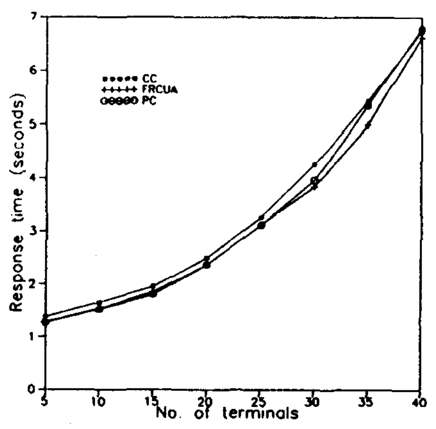

Figure 4.4: Response time of *read queries*.

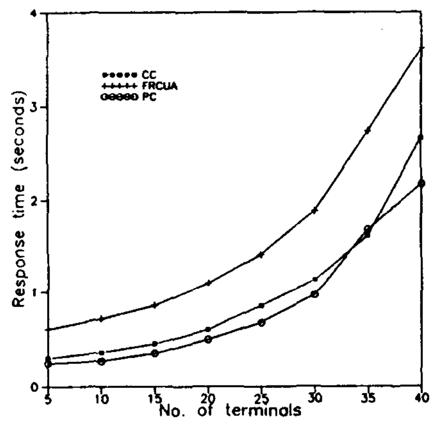

Figure 4.5: Response time of *update queries*.

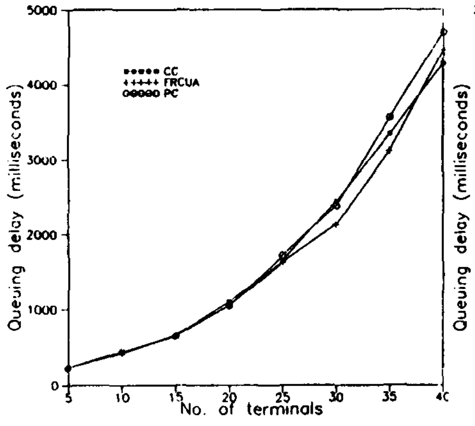

Figure 4.6: Queuing delay of *read queries* in the *data disk queue*.

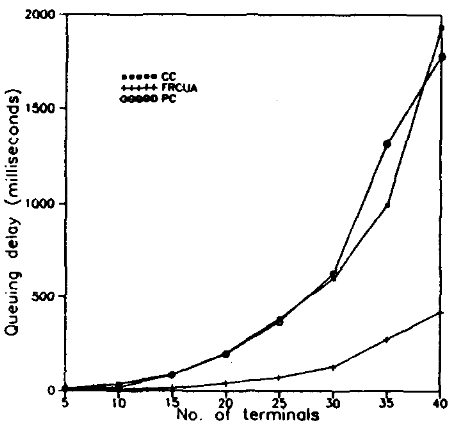

Figure 4.7: Queuing delay of *update queries* in the *blocked queue*.

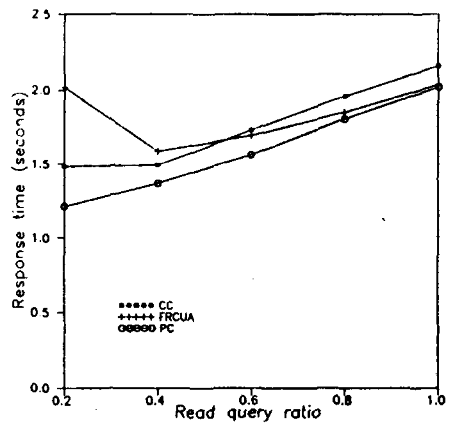

Figure 4.8: Response time of *read queries*.

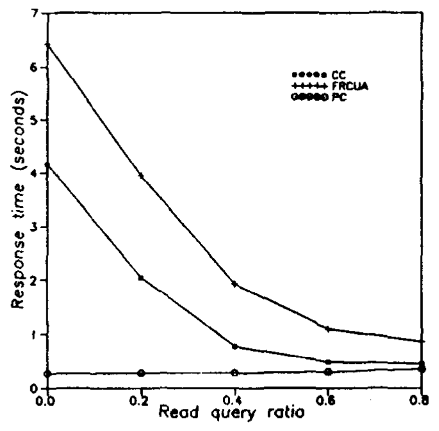

Figure 4.9: Response time of *update queries*.

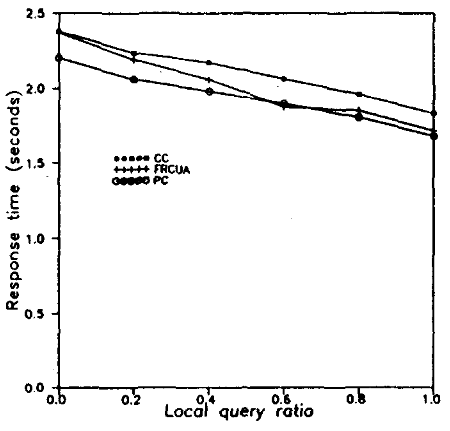

Figure 4.10: Response time of *read queries*.

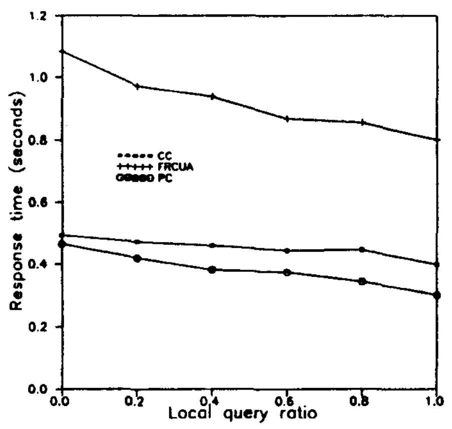

Figure 4.11: Response time of *update queries*.

is improved as the local query ratio increases. The reason is as follows. When a *read query* is generated from a terminal at a site, its corresponding access module is invoked at the local site, and its execution is initiated. As the local query ratio increases, the probability that the query units of an access module be processed at the same site increases. Then the probability that the data messages generated at one site be transmitted to another site to be joined (via pipelined semijoin method) decreases. Therefore, the queuing delay in the *msg_cpu queue* also is reduced.

Figure 4.11 shows that FRCUA for *update queries* has the worst performance for all local query ratio ranges because of the same reasons described in Experiment 1. However, its performance is improved as the local query ratio increases. This is due to the fact that more *read queries* can be processed locally as the local query ratio increases, leading to the reduction of queuing delay in the *msg_cpu queue*. As one would expect, the performance of CC for *update queries* is relatively little affected by the change of local query ratio. Its minor improvement is attributed to the fact that queuing delay in the *msg_cpu queue* is decreased, since *read queries* need to access only local site (and the central site). The performance of PC for *update queries* is also slightly improved because the CPU time needed to message processing and communication time are reduced. An access module holds the shared locks to the associated catalog entries until its completion. During this period, any *update query* which wishes to lock the same catalog entries in an exclusive mode must await the release of the locks. If some of the query units of an access module are invoked from the remote sites and processed there, shared locks are held for longer periods, leading to the increase of the queuing delay in the *blocked queue*. Thus, the queuing delay of *update queries* in the *blocked queue* turns out to be strongly correlated to the queuing delay in the *msg_cpu queue*.

5. Conclusions

In this paper, the performance of the three catalog architectures measured in terms of running access modules is examined using simulation in a locally distributed database system. The effect of several important parameters on the relative performance of the three catalog management schemes is studied and analyzed.

The three catalog management schemes considered include the centralized catalogs (CC), the fully replicated catalogs with unanimous agreement (FRCUA), and the partitioned catalogs (PC). For *read queries*, in no situation does CC become the catalog management of choice. This is due to the fact that all *read queries* generated from any site need to access the central site to lock the associated catalog entries in a shared mode. It is found that there is only one situation in which FRCUA for *read queries* outperforms PC: the higher number of terminals. Except this case, the performance of PC for *read queries* is better than FRCUA.

For *update queries*, FRCUA has the worst performance for all simulation conditions because of the inherent needs to access all sites in a distributed database system. FRCUA also needs the centralized two-phase commit protocol to treat an *update query* as a transaction. PC performs best, since updates to the catalogs can be performed at one site (local or remote site) where the catalog information to be updated is currently stored. CC has in-between performance.

This paper has concentrated primarily on the performance issue of the catalog management schemes. The main performance metric used to compare the performance among several catalog architectures is the response time. In certain cases, however, achieving higher availability would be a major concern rather than achieving higher performance. It would be useful to study the performance–availability tradeoffs raised by the catalog management schemes.

References

[1] E.Bertino, et al., "View Management in Distributed Database Systems," IBM Research Report RJ3851, San Jose, Calif., April 1983.

[2] M.J.Carey and M.Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," Proc. 14th Int'l Conf. on VLDB, LA, Calif., Sept. 1988, pp.13-25.

[3] W.W.Chu, "Performance of File Directory Systems for Data Bases in Star and Distributed Networks," Proc. AFIPS, 1976, pp.577-587.

[4] D.W.Cornell and P.S.Yu, "On Optimal Site Assignment for Relations in the Distributed

Database Environment," IEEE Trans. on Software Eng. SE-15,8 (Aug. 1989), pp.1004-1009.

[5] D.Daniels and A.Z.Spector, "An Algorithm for Replicated Directories," ACM Annual Symp. on Principles of Dist. Comput., 1983, pp.104-113.

[6] L.W.Dowdy and D.V.Foster, "Comparative Models of the File Assignment Problem," ACM Computing Surveys 14,2 (June 1982), pp.287-313.

[7] J.Gray, "Notes on Data Base Operating Systems," Operating Systems: An Advanced Course, Lecture Notes in Computer Science:60, Edited by: R.Bayer, et al., Springer-Verlag, 1979.

[8] E.K.Hong, "Modeling and Evaluation of Three Catalog Management Schemes in a Distributed Database System," Jour. of Korea Information Science Society 16,2 (March 1989), pp.178-189 (in Korean).

[9] E.K.Hong and J.W.Cho, "Performance Evaluation of Catalog Management Schemes in Distributed Database Systems," Information Systems 16,2 (1991), pp.125-144.

[10] E.K.Hong, "Performance Evaluation of Catalog Architectures in Distributed Database Systems," Ph.D. Thesis, Dept. of Computer Science, KAIST, Seoul, Korea, Feb. 1991.

[11] IBM Corporation, SQL/Data System Concepts and Facilities, IBM Reference Manual GH24-5013-2, File No. S370/4300-50, Aug. 1983.

[12] R.B.Kolstad, et al., "Path Pascal User Manual," 2nd Edition, Univ. of Illinois-Champaign, Dept. of Computer Science, Nov. 1984.

[13] B.G.Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager," IBM Research Report RJ2914, San Jose, Calif., Aug. 1980.

[14] B.G.Lindsay and P.G.Selinger, "Site Autonomy Issues in R^*: A Distributed Database Management System," IBM Research Report RJ2927, San Jose, Calif., Sept. 1980.

[15] G.M.Lohman, et al., "Query Processing in R^*," IBM Research Report RJ4272, San Jose, Calif., April 1984.

[16] G.M.Lohman, Personal communication, 1987.

[17] H.Lu, "Distributed Query Processing with Load Balancing in Local Area Networks," Ph.D. Thesis, Computer Sciences Technical Report #624, Univ. of Wisconsin-Madison, Dec. 1985.

[18] Y.Matsushita, et al., "Cost Evaluation of Directory Management Schemes for Distributed Database Systems," Proc. ACM SIGMOD Conference, 1980, pp.117-124.

[19] D.J.Rosenkrantz, et al., "System Level Concurrency Control for Distributed Database Systems," ACM Trans. on Database Systems 3,2 (June 1978), pp.178-198.

[20] J.B.Rothnie,Jr., et al., "Introduction to a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems 5,1 (March 1980), pp.1-17.

[21] M.Stonebraker and E.Neuhold, "A Distributed Data Base Version of INGRES," 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp.19-36.

[22] P.F.Wilms and B.G.Lindsay, "A Database Authorization Mechanism Supporting Individual and Group Authorization," IBM Research Report RJ3137, San Jose, Calif., May 1981.

[23] P.F.Wilms, et al., ""I wish I were over there": Distributed Execution Protocols for Data Definition in R^*," IBM Research Report RJ3892, San Jose, Calif., May 1983.