A History and Evaluation of System R

Donald D. Chamberlin
Morton M. Astrahan
Michael W. Blasgen
James N. Gray
W. Frank King
Bruce G. Lindsay
Raymond Lorie
James W. Mehl
Thomas G. Price
Franco Putzolu
Patricia Griffiths Selinger
Mario Schkolnick
Donald R. Slutz
Irving L. Traiger
Bradford W. Wade
Robert A. Yost

IBM Research Laboratory
San Jose, California

SUMMARY: System R, an experimental database system, was constructed to demonstrate that the usability advantages of the relational data model can be realized in a system with the complete function and high performance required for everyday production use. This paper describes the three principal phases of the System R project and discusses some of the lessons learned from System R about the design of relational systems and database systems in general.

1. Introduction

Throughout the history of information storage in computers, one of the most readily observable trends has been the focus on data independence. C.J. Date [27] defined data independence as "immunity of applications to change in storage structure and access strategy." Modern database systems offer data independence by providing a high-level user interface through which users deal with the information content of their data, rather than the various bits, pointers, arrays, lists, etc. which are used to represent that information. The system assumes responsibility for choosing an appropriate internal representation for the information; indeed, the representation of a given fact may change over time without users being aware of the change.

The relational data model was proposed by E.F. Codd [22] in 1970 as the next logical step in the trend toward data independence. Codd observed that conventional database systems store information in two ways: (1) by the contents of records stored in the database, and (2) by the ways in which these records are connected together. Different systems use various names for the connections among records, such as links, sets, chains, parents, etc. For example, in Figure 1(a), the fact that supplier Acme supplies bolts is represented by connections between the relevant part and supplier records. In such a system, a user frames a question, such as "What is the lowest price for bolts?", by writing a program which "navigates" through the maze of connections until it arrives at the answer to the question. The user of a "navigational" system has the burden (or opportunity) to specify exactly how the query is to be processed; the user's algorithm is then embodied in a program which is dependent on the data structure that existed at the time the program was written.

Relational database systems, as proposed by Codd, have two important properties: (1) all information is

Key words and phrases: database management systems, relational model, compilation, locking, recovery, access path selection, authorization
CR Categories: 3.50, 3.70, 3.72, 4.33, 4.6
Authors' address: D. D. Chamberlin et al., IBM Research Laboratory, 5600 Cottle Road, San Jose, California 95193.

represented by data values, never by any sort of "connections" which are visible to the user; (2) the system supports a very high-level language in which users can frame requests for data without specifying algorithms for processing the requests. The relational representation of the data in Figure 1(a) is shown in Figure 1(b). Information about parts is kept in a PARTS relation in which each record has a "key" (unique identifier) called PARTNO. Information about suppliers is kept in a SUPPLIERS relation keyed by SUPPNO. The information which was formerly represented by connections between records is now contained in a third relation, PRICES, in which parts and suppliers are represented by their respective keys. The question "What is the lowest price for bolts?" can be framed in a high-level language like SQL [16] as follows:

```
SELECT MIN(PRICE)
FROM   PRICES
WHERE  PARTNO IN
       (SELECT PARTNO
       FROM    PARTS
       WHERE   NAME = 'BOLT');
```

A relational system can maintain whatever pointers, indices, or other access aids it finds appropriate for processing user requests, but the user's request is not framed in terms of these access aids and is therefore not dependent on them. Therefore, the system may change its data representation and access aids periodically to adapt to changing requirements without disturbing users' existing applications.

Since Codd's original paper, the advantages of the relational data model in terms of user productivity and data independence have become widely recognized. However, as in the early days of high-level programming languages, questions are sometimes raised about whether or not an automatic system can choose as efficient an algorithm for processing a complex query as a trained programmer would. System R is an experimental system constructed at the San Jose IBM Research Laboratory to demonstrate that a relational database system can incorporate the high performance and complete function required for everyday production use.

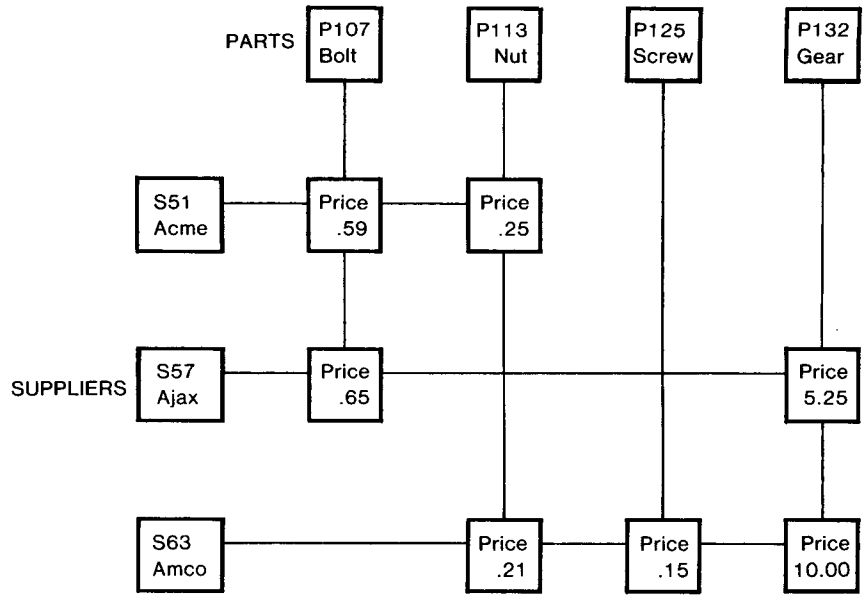

Fig. 1(a). A "Navigational" Database.

The key goals established for System R were:

(1) To provide a high-level, nonnavigational user interface for maximum user productivity and data independence.

(2) To support different types of database use including programmed transactions, ad hoc queries, and report generation.

(3) To support a rapidly changing database environment, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the database without stopping the system.

(4) To support a population of many concurrent users, with mechanisms to protect the integrity of the database in a concurrent-update environment.

(5) To provide a means of recovering the contents of the database to a consistent state after a failure of hardware or software.

(6) To provide a flexible mechanism whereby different views of stored data can be defined and various users can be authorized to query and update these views.

(7) To support all of the above functions with a level of performance comparable to existing lower-function database systems.

Throughout the System R project, there has been a strong commitment to carry the system through to an operationally complete prototype

PARTS

PARTNO	NAME
P107	Bolt
P113	Nut
P125	Screw
P132	Gear

SUPPLIERS

SUPPNO	NAME
S51	Acme
S57	Ajax
S63	Amco

PRICES

PARTNO	SUPPNO	PRICE
P107	S51	.59
P107	S57	.65
P113	S51	.25
P113	S63	.21
P125	S63	.15
P132	S57	5.25
P132	S63	10.00

Fig. 1(b). A Relational Database.

which could be installed and evaluated in actual user sites.

The history of System R can be divided into three phases. "Phase Zero" of the project, which occurred during 1974 and most of 1975, involved the development of the SQL user interface [14] and a quick implementation of a subset of SQL for one user at a time. The Phase Zero prototype, described in [2], provided valuable insight in several areas, but its code was eventually abandoned. "Phase One" of the project, which took place throughout most of 1976 and 1977, involved the design and construction of the full-function, multiuser version of System R. An initial system architecture was presented in [4] and subsequent updates to the design were described in [10]. "Phase Two" was the evaluation of System R in actual use. This occurred during 1978 and 1979 and involved experiments at the San Jose Research Laboratory and several other user sites. The results of some of these experiments and user experiences are described in [19–21]. At each user site, System R was installed for experimental purposes only, and not as a supported commercial product.[1]

This paper will describe the decisions which were made and the lessons learned during each of the three phases of the System R project.

2. Phase Zero: An Initial Prototype

Phase Zero of the System R project involved the quick implementation of a subset of system functions. From the beginning, it was our intention to learn what we could from this initial prototype, and then scrap the Phase Zero code before construction of the more complete version of System R. We decided to use the relational access method called XRM, which had been developed by R. Lorie at IBM's Cambridge Scientific Center [40]. (XRM was influenced, to some extent, by the "Gamma Zero" interface defined by E.F. Codd and others at San Jose [11].) Since XRM is a single-user access method without locking or recovery capabilities, issues relating to concurrency and recovery were excluded from consideration in Phase Zero.

An interpreter program was written in PL/I to execute statements in the high-level SQL (formerly SEQUEL) language [14, 16] on top of XRM. The implemented subset of the SQL language included queries and updates of the database, as well as the dynamic creation of new database relations. The Phase Zero implementation supported the "subquery" construct of SQL, but not its "join" construct. In effect, this meant that a query could search through several relations in computing its result, but the final result would be taken from a single relation.

The Phase Zero implementation was primarily intended for use as a standalone query interface by end users at interactive terminals. At the time, little emphasis was placed on issues of interfacing to host-language programs (although Phase Zero could be called from a PL/I program). However, considerable thought was given to the human factors aspects of the SQL language, and an experimental study was conducted on the learnability and usability of SQL [44].

One of the basic design decisions in the Phase Zero prototype was that the system catalog, i.e., the description of the content and structure of the database, should be stored as a set of regular relations in the database itself. This approach permits the system to keep the catalog up to date automatically as changes are made to the database, and also makes the catalog information available to the system optimzer for use in access path selection.

The structure of the Phase Zero interpreter was strongly influenced by the facilities of XRM. XRM stores relations in the form of "tuples," each of which has a unique 32-bit "tuple identifier" (TID). Since a TID contains a page number, it is possible, given a TID, to fetch the associated tuple in one page reference. However, rather than actual data values, the tuple contains pointers to the "domains" where the actual data is stored, as shown in Figure 2. Optionally, each domain may have an "inversion," which associates domain values (e.g., "Programmer") with the TIDs of tuples in which the values appear. Using the inversions, XRM makes it easy to find a list of TIDs of tuples which contain a given value. For example, in Figure 2, if inversions exist on both the JOB and LOCATION domains, XRM provides commands to create a list of TIDs of employees who are programmers, and another list of TIDs of employees who work in Evanston. If the SQL query calls for programmers who work in Evanston, these TID lists can be intersected to obtain the list of TIDs of tuples which satisfy the query, before any tuples are actually fetched.

The most challenging task in constructing the Phase Zero prototype was the design of optimizer algorithms for efficient execution of SQL statements on top of XRM. The design of the Phase Zero optimizer is given in [2]. The objective of the optimizer was to minimize the number of tuples fetched from the database in processing a query. Therefore, the optimizer made extensive use of inversions and often manipulated TID lists before beginning to fetch tuples. Since the TID lists were potentially large, they were stored as temporary objects in the database during query processing.

The results of the Phase Zero implementation were mixed. One strongly felt conclusion was that it is a very good idea, in a project the size of System R, to plan to throw away the initial implementation. On the positive side, Phase Zero demonstrated the usability of the SQL language, the feasibility of creating new tables and inversions "on the fly"

[1] The System R research prototype later evolved into SQL/Data System, a relational database management product offered by IBM in the DOS/VSE operating system environment.

and relying on an automatic optimizer for access path selection, and the convenience of storing the system catalog in the database itself. At the same time, Phase Zero taught us a number of valuable lessons which greatly influenced the design of our later implementation. Some of these lessons are summarized below.

(1) The optimizer should take into account not just the cost of fetching tuples, but the costs of creating and manipulating TID lists, then fetching tuples, then fetching the data pointed to by the tuples. When these "hidden costs" are taken into account, it will be seen that the manipulation of TID lists is quite expensive, especially if the TID lists are managed in the database rather than in main storage.

(2) Rather than "number of tuples fetched," a better measure of cost would have been "number of I/Os." This improved cost measure would have revealed the great importance of clustering together related tuples on physical pages so that several related tuples could be fetched by a single I/O. Also, an I/O measure would have revealed a serious drawback of XRM: Storing the domains separately from the tuples causes many extra I/Os to be done in retrieving data values. Because of this, our later implementation stored data values in the actual tuples rather than in separate domains. (In defense of XRM, it should be noted that the separation of data values from tuples has some advantages if data values are relatively large and if many tuples are processed internally compared to the number of tuples which are materialized for output.)

(3) Because the Phase Zero implementation was observed to be CPU-bound during the processing of a typical query, it was decided the optimizer cost measure should be a weighted sum of CPU time and I/O count, with weights adjustable according to the system configuration.

(4) Observation of some of the applications of Phase Zero convinced us of the importance of the "join" formulation of SQL. In our subsequent implementation, both "joins" and "subqueries" were supported.

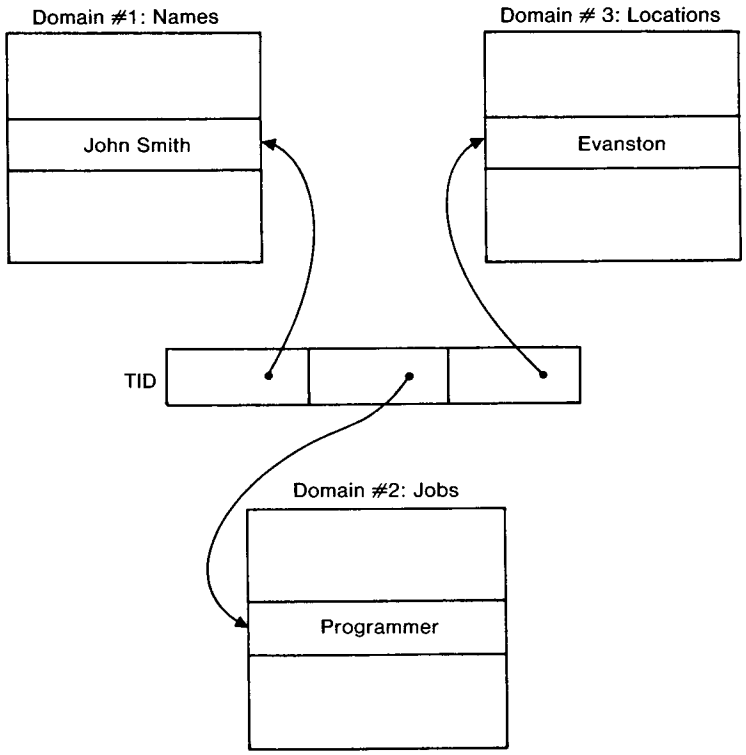

Fig. 2. XRM Storage Structure.

(5) The Phase Zero optimizer was quite complex and was oriented toward complex queries. In our later implementation, greater emphasis was placed on relatively simple interactions, and care was taken to minimize the "path length" for simple SQL statements.

3. Phase One: Construction of a Multiuser Prototype

After the completion and evaluation of the Phase Zero prototype, work began on the construction of the full-function, multiuser version of System R. Like Phase Zero, System R consisted of an access method (called RSS, the Research Storage System) and an optimizing SQL processor (called RDS, the Relational Data System) which runs on top of the RSS. Separation of the RSS and RDS provided a beneficial degree of modularity; e.g., all locking and logging functions were isolated in the RSS, while all authorization and access path selection functions were isolated in the RDS. Construction of the RSS was underway in 1975 and construction of the RDS began in 1976. Unlike XRM, the RSS was originally designed to support multiple concurrent users.

The multiuser prototype of System R contained several important subsystems which were not present in the earlier Phase Zero prototype. In order to prevent conflicts which might arise when two concurrent users attempt to update the same data value, a locking subsystem was provided. The locking subsystem ensures that each data value is accessed by only one user at a time, that all the updates made by a given transaction become effective simultaneously, and that deadlocks between users are detected and resolved. The security of the system was enhanced by view and authorization subsystems. The view subsystem permits users to define alternative views of the database (e.g., a view of the employee file in which salaries are deleted or aggregated by department).

The authorization subsystem ensures that each user has access only to those views for which he has been specifically authorized by their creators. Finally, a recovery subsystem was provided which allows the database to be restored to a consistent state in the event of a hardware or software failure.

In order to provide a useful host-language capability, it was decided that System R should support both PL/I and Cobol application programs as well as a standalone query interface, and that the system should run under either the VM/CMS or MVS/TSO operating system environment. A key goal of the SQL language was to present the same capabilities, and a consistent syntax, to users of the PL/I and Cobol host languages and to ad hoc query users. The imbedding of SQL into PL/I is described in [16]. Installation of a multiuser database system under VM/CMS required certain modifications to the operating system in support of communicating virtual machines and writable shared virtual memory. These modifications are described in [32].

The standalone query interface of System R (called UFI, the User-Friendly Interface) is supported by a dialog manager program, written in PL/I, which runs on top of System R like any other application program. Therefore, the UFI support program is a cleanly separated component and can be modified independently of the rest of the system. In fact, several users improved on our UFI by writing interactive dialog managers of their own.

The Compilation Approach

Perhaps the most important decision in the design of the RDS was inspired by R. Lorie's observation, in early 1976, that it is possible to compile very high-level SQL statements into compact, efficient routines in System/370 machine language [42]. Lorie was able to demonstrate that SQL statements of arbitrary complexity could be decomposed into a relatively small collection of machine-language "fragments," and that an optimizing compiler could assemble these code fragments from a library to form a specially tailored routine for processing a given SQL statement. This technique had a very dramatic effect on our ability to support application programs for transaction processing. In System R, a PL/I or Cobol program is run through a preprocessor in which its SQL statements are examined, optimized, and compiled into small, efficient machine-language routines which are packaged into an "access module" for the application program. Then, when the program goes into execution, the access module is invoked to perform all interactions with the database by means of calls to the RSS. The process of creating and invoking an access module is illustrated in Figures 3 and 4. All the overhead of parsing, validity checking, and access path selection is removed from the path of the executing program and placed in a separate preprocessor step which need not be repeated. Perhaps even more important is the fact that the running program interacts only with its small, special-purpose access module rather than with a much larger and less efficient general-purpose SQL interpreter. Thus, the power and ease of use of the high-level SQL language are combined with the execution-time efficiency of the much lower level RSS interface.

Since all access path selection decisions are made during the preprocessor step in System R, there is the possibility that subsequent changes in the database may invalidate the decisions which are embodied in an access module. For example, an index selected by the optimizer may later be dropped from the database. Therefore, System R records with each access module a list of its "dependencies" on database objects such as tables and indexes. The dependency list is stored in the form of a regular relation in the system catalog. When the structure of the database changes (e.g., an index is dropped), all affected access modules are marked "invalid." The next time an invalid access module is invoked, it is regenerated from its original SQL statements, with newly optimized access paths. This process is completely transparent to the System R user.

SQL statements submitted to the interactive UFI dialog manager are processed by the same optimizing compiler as preprocessed SQL statements. The UFI program passes the ad hoc SQL statement to System R with a special "EXECUTE" call. In response to the EXECUTE call, System R parses and optimizes the SQL statement and translates it into a machine-language routine. The routine is indistinguishable from an access module and is executed immediately. This process is described in more detail in [20].

RSS Access Paths

Rather than storing data values in separate "domains" in the manner of XRM, the RSS chose to store data values in the individual rcords of the database. This resulted in records becoming variable in length and longer, on the average, than the equivalent XRM records. Also, commonly used values are represented many times rather than only once as in XRM. It was felt, however, that these disadvantages were more than offset by the following advantage: All the data values of a record could be fetched by a single I/O.

In place of XRM "inversions," the RSS provides "indexes," which are associative access aids implemented in the form of B-Trees [26]. Each table in the database may have anywhere from zero indexes up to an index on each column (it is also possible to create an index on a combination of columns). Indexes make it possible to scan the table in order by the indexed values, or to directly access the records which match a particular value. Indexes are maintained automatically by the RSS in the event of updates to the database.

The RSS also implements "links," which are pointers stored

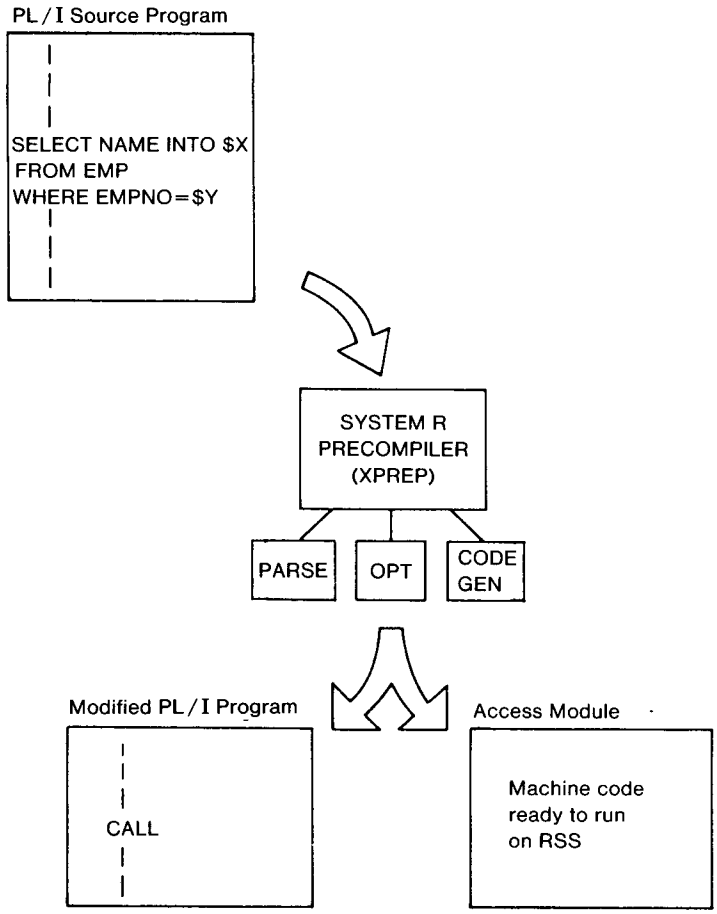

Fig. 3. Precompilation Step.

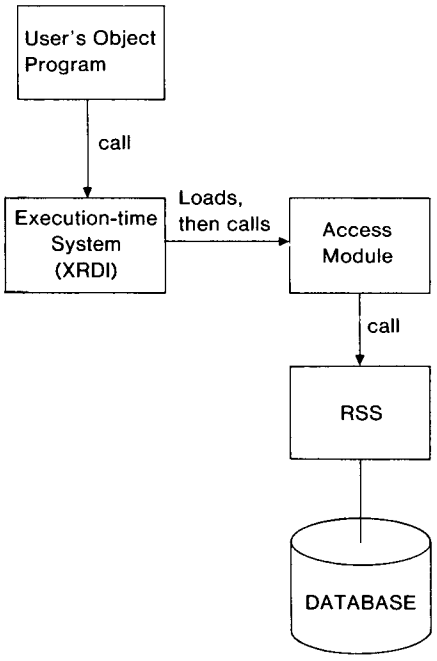

Fig. 4. Execution Step.

with a record which connect it to other related records. The connection of records on links is not performed automatically by the RSS, but must be done by a higher level system.

The access paths made available by the RSS include (1) index scans, which access a table associatively and scan it in value order using an index; (2) relation scans, which scan over a table as it is laid out in physical storage; (3) link scans, which traverse from one record to another using links. On any of these types of scan, "search arguments" may be specified which limit the records returned to those satisfying a certain predicate. Also, the RSS provides a built-in sorting mechanism which can take records from any of the scan methods and sort them into some value order, storing the result in a temporary list in the database. In System R, the RDS makes extensive use of index and relation scans and sorting. The RDS also utilizes links for internal purposes but not as an access path to user data.

The Optimizer

Building on our Phase Zero experience, we designed the System R optimizer to minimize the weighted sum of the predicted number of I/Os and RSS calls in processing an SQL statement (the relative weights of these two terms are adjustable according to system configuration). Rather than manipulating TID lists, the optimizer chooses to scan each table in the SQL query by means of only one index (or, if no suitable index exists, by means of a relation scan). For example, if the query calls for programmers who work in Evanston, the optimizer might choose to use the job index to find programmers and then examine their locations; it might use the location index to find Evanston employees and examine their jobs; or it might simply scan the relation and examine the job and location of all employees. The choice would be based on the optimizer's estimate of both the clustering and selectivity properties of each index, based on statistics stored in the system catalog. An index is considered highly selective if it has a large ratio of distinct key values to total entries. An index is considered to have the clustering property if the key order of the index corresponds closely to the ordering of records in physical storage. The clustering property is important because when a record is fetched via a clustering index, it is likely that other records with the same key will be found on the same page, thus minimizing the number of page fetches. Because of the importance of clustering, mechanisms were provided for loading data in value order and preserving the value ordering when new records are inserted into the database.

The techniques of the System R optimizer for performing joins of two or more tables have their origin in a study conducted by M. Blasgen and

K. Eswaran [7]. Using APL models, Blasgen and Eswaran studied ten methods of joining together tables, based on the use of indexes, sorting, physical pointers, and TID lists. The number of disk accesses required to perform a join was predicted on the basis of various assumptions for the ten join methods. Two join methods were identified such that one or the other was optimal or nearly optimal under most circumstances. The two methods are as follows:

Join Method 1: Scan over the qualifying rows of table A. For each row, fetch the matching rows of table B (usually, but not always, an index on table B is used).

Join Method 2: (Often used when no suitable index exists.) Sort the qualifying rows of tables A and B in order by their respective join fields. Then scan over the sorted lists and merge them by matching values.

When selecting an access path for a join of several tables, the System R optimizer considers the problem to be a sequence of binary joins. It then performs a tree search in which each level of the tree consists of one of the binary joins. The choices to be made at each level of the tree include which join method to use and which index, if any, to select for scanning. Comparisons are applied at each level of the tree to prune away paths which achieve the same results as other, less costly paths. When all paths have been examined, the optimizer selects the one of minimum predicted cost. The System R optimizer algorithms are described more fully in [47].

Views and Authorization

The major objectives of the view and authorization subsystems of System R were power and flexibility. We wanted to allow any SQL query to be used as the definition of a view. This was accomplished by storing each view definition in the form of an SQL parse tree. When an SQL operation is to be executed against a view, the parse tree which defines the operation is merged with the parse tree which defines the view, producing a composite parse tree which is then sent to the optimizer for access path selection. This approach is similar to the "query modification" technique proposed by Stonebraker [48]. The algorithms developed for merging parse trees were sufficiently general so that nearly any SQL statement could be executed against any view definition, with the restriction that a view can be updated only if it is derived from a single table in the database. The reason for this restriction is that some updates to views which are derived from more than one table are not meaningful (an example of such an update is given in [24]).

The authorization subsystem of System R is based on privileges which are controlled by the SQL statements GRANT and REVOKE. Each user of System R may optionally be given a privilege called RESOURCE which enables him/her to create new tables in the database. When a user creates a table, he/she receives all privileges to access, update, and destroy that table. The creator of a table can then grant these privileges to other individual users, and subsequently can revoke these grants if desired. Each granted privilege may optionally carry with it the "GRANT option," which enables a recipient to grant the privilege to yet other users. A REVOKE destroys the whole chain of granted privileges derived from the original grant. The authorization subsystem is described in detail in [37] and discussed further in [31].

The Recovery Subsystem

The key objective of the recovery subsystem is provision of a means whereby the database may be recovered to a consistent state in the event of a failure. A consistent state is defined as one in which the database does not reflect any updates made by transactions which did not complete successfully. There are three basic types of failure: the disk media may fail, the system may fail, or an individual transaction may fail. Although both the scope of the failure and the time to effect recovery may be different, all three types of recovery require that an alternate copy of data be available when the primary copy is not.

When a media failure occurs, database information on disk is lost. When this happens, an image dump of the database plus a log of "before" and "after" changes provide the alternate copy which makes recovery possible. System R's use of "dual logs" even permits recovery from media failures on the log itself. To recover from a media failure, the database is restored using the latest image dump and the recovery process reapplies all database changes as specified on the log for completed transactions.

When a system failure occurs, the information in main memory is lost. Thus, enough information must always be on disk to make recovery possible. For recovery from system failures, System R uses the change log mentioned above plus something called "shadow pages." As each page in the database is updated, the page is written out in a new place on disk, and the original page is retained. A directory of the "old" and "new" locations of each page is maintained. Periodically during normal operation, a "checkpoint" occurs in which all updates are forced out to disk, the "old" pages are discarded, and the "new" pages become "old." In the event of a system crash, the "new" pages on disk may be in an inconsistent state because some updated pages may still be in the system buffers and not yet reflected on disk. To bring the database back to a consistent state, the system reverts to the "old" pages, and then uses the log to redo all committed transactions and to undo all updates made by incomplete transactions. This aspect of the System R recovery subsystem is described in more detail in [36].

When a transaction failure occurs, all database changes which have been made by the failing transaction must be undone. To accom-

plish this, System R simply processes the change log backwards removing all changes made by the transaction. Unlike media and system recovery which both require that System R be reinitialized, transaction recovery takes place on-line.

The Locking Subsystem

A great deal of thought was given to the design of a locking subsystem which would prevent interference among concurrent users of System R. The original design involved the concept of "predicate locks," in which the lockable unit was a database property such as "employees whose location is Evanston." Note that, in this scheme, a lock might be held on the predicate LOC = 'EVANSTON', even if no employees currently satisfy that predicate. By comparing the predicates being processed by different users, the locking subsystem could prevent interference. The "predicate lock" design was ultimately abandoned because: (1) determining whether two predicates are mutually satisfiable is difficult and time-consuming; (2) two predicates may appear to conflict when, in fact, the semantics of the data prevent any conflict, as in "PRODUCT = AIRCRAFT" and "MANUFACTURER = ACME STATIONERY CO."; and (3) we desired to contain the locking subsystem entirely within the RSS, and therefore to make it independent of any understanding of the predicates being processed by various users. The original predicate locking scheme is described in [29].

The locking scheme eventually chosen for System R is described in [34]. This scheme involves a hierarchy of locks, with several different sizes of lockable units, ranging from individual records to several tables. The locking subsystem is transparent to end users, but acquires locks on physical objects in the database as they are processed by each user. When a user accumulates many small locks, they may be "traded" for a larger lockable unit (e.g., locks on many records in a table might be traded for a lock on the table). When locks are acquired on small objects, "intention" locks are simultaneously acquired on the larger objects which contain them. For example, user A and user B may both be updating employee records. Each user holds an "intention" lock on the employee table, and "exclusive" locks on the particular records being updated. If user A attempts to trade her individual record locks for an "exclusive" lock at the table level, she must wait until user B ends his transaction and releases his "intention" lock on the table.

4. Phase Two: Evaluation

The evaluation phase of the System R project lasted approximately 2½ years and consisted of two parts: (1) experiments performed on the system at the San Jose Research Laboratory, and (2) actual use of the system at a number of internal IBM sites and at three selected customer sites. At all user sites, System R was installed on an experimental basis for study purposes only, and not as a supported commercial product. The first installations of System R took place in June 1977.

General User Comments

In general, user response to System R has been enthusiastic. The system was mostly used in applications for which ease of installation, a high-level user language, and an ability to rapidly reconfigure the database were important requirements. Several user sites reported that they were able to install the system, design and load a database, and put into use some application programs within a matter of days. User sites also reported that it was possible to tune the system performance after data was loaded by creating and dropping indexes without impacting end users or application programs. Even changes in the database tables could be made transparent to users if the tables were read-only, and also in some cases for updated tables.

Users found the performance characteristics and resource consumption of System R to be generally satisfactory for their experimental applications, although no specific performance comparisons were drawn. In general, the experimental databases used with System R were smaller than one 3330 disk pack (200 Megabytes) and were typically accessed by fewer than ten concurrent users. As might be expected, interactive response slowed down during the execution of very complex SQL statements involving joins of several tables. This performance degradation must be traded off against the advantages of normalization [23, 30], in which large database tables are broken into smaller parts to avoid redundancy, and then joined back together by the view mechanism or user applications.

The SQL Language

The SQL user interface of System R was generally felt to be successful in achieving its goals of simplicity, power, and data independence. The language was simple enough in its basic structure so that users without prior experience were able to learn a usable subset on their first sitting. At the same time, when taken as a whole, the language provided the query power of the first-order predicate calculus combined with operators for grouping, arithmetic, and built-in functions such as SUM and AVERAGE.

Users consistently praised the uniformity of the SQL syntax across the environments of application programs, ad hoc query, and data definition (i.e., definition of views). Users who were formerly required to learn inconsistent languages for these purposes found it easier to deal with the single syntax (e.g., when debugging an application program by querying the database to observe its effects). The single syntax also enhanced communication among different functional organizations (e.g., between database administrators and application programmers).

While developing applications using SQL, our experimental users made a number of suggestions for extensions and improvements to the language, most of which were implemented during the course of the proj-

ect. Some of these suggestions are summarized below:

(1) Users requested an easy-to-use syntax when testing for the existence or nonexistence of a data item, such as an employee record whose department number matches a given department record. This facility was implemented in the form of a special "EXISTS" predicate.

(2) Users requested a means of seaching for character strings whose contents are only partially known, such as "all license plates beginning with NVK." This facility was implemented in the form of a special "LIKE" predicate which searches for "patterns" that are allowed to contain "don't care" characters.

(3) A requirement arose for an application program to compute an SQL statement dynamically, submit the statement to the System R optimizer for access path selection, and then execute the statement repeatedly for different data values without reinvoking the optimizer. This facility was implemented in the form of PREPARE and EXECUTE statements which were made available in the host-language version of SQL.

(4) In some user applications the need arose for an operator which Codd has called an "outer join" [25]. Suppose that two tables (e.g., SUPPLIERS and PROJECTS) are related by a common data field (e.g., PARTNO). In a conventional join of these tables, supplier records which have no matching project record (and vice versa) would not appear. In an "outer join" of these tables, supplier records with no matching project record would appear together with a "synthetic" project record containing only null values (and similarly for projects with no matching supplier). An "outer-join" facility for SQL is currently under study.

A more complete discussion of user experience with SQL and the resulting language improvements is presented in [19].

The Compilation Approach

The approach of compiling SQL statements into machine code was one of the most successful parts of the System R project. We were able to generate a machine-language routine to execute any SQL statement of arbitrary complexity by selecting code fragments from a library of approximately 100 fragments. The result was a beneficial effect on transaction programs, ad hoc query, and system simplicity.

In an environment of short, repetitive transactions, the benefits of compilation are obvious. All the overhead of parsing, validity checking, and access path selection are removed from the path of the running transaction, and the application program interacts with a small, specially tailored access module rather than with a larger and less efficient general-purpose interpreter program. Experiments [38] showed that for a typical short transaction, about 80 percent of the instructions were executed by the RSS, with the remaining 20 percent executed by the access module and application pro-

Example 1:

```
SELECT  SUPPNO, PRICE
FROM    QUOTES
WHERE   PARTNO = '010002'
AND MINQ<=1000 AND MAXQ>=1000;
```

Operation	CPU time (msec on 168)	Number of I/Os
Parsing	13.3	0
Access Path Selection	40.0	9
Code Generation	10.1	0
Fetch answer set (per record)	1.5	0.7

Example 2:

```
SELECT  ORDERNO,ORDERS.PARTNO,DESCRIP,DATE,QTY
FROM    ORDERS,PARTS
WHERE   ORDERS.PARTNO = PARTS.PARTNO
AND     DATE BETWEEN '750000' AND '751231'
AND     SUPPNO = '797';
```

Operation	CPU time (msec on 168)	Number of I/Os
Parsing	20.7	0
Access Path Selection	73.2	9
Code Generation	19.3	0
Fetch answer set (per record)	8.7	10.7

Fig. 5. Measurements of Cost of Compilation.

gram. Thus, the user pays only a small cost for the power, flexibility, and data independence of the SQL language, compared with writing the same transaction directly on the lower level RSS interface.

In an ad hoc query environment the advantages of compilation are less obvious since the compilation must take place on-line and the query is executed only once. In this environment, the cost of generating a machine-language routine for a given query must be balanced against the increased efficiency of this routine as compared with a more conventional query interpreter. Figure 5 shows some measurements of the cost of compiling two typical SQL statements (details of the experiments are given in [20]). From this data we may draw the following conclusions:

(1) The code generation step adds a small amount of CPU time and no I/Os to the overhead of parsing and access path selection. Parsing and access path selection must be done in any query system, including interpretive ones. The additional instructions spent on code generation are not likely to be perceptible to an end user.

(2) If code generation results in a routine which runs more efficiently than an interpreter, the cost of the code generation step is paid back after fetching only a few records. (In Example 1, if the CPU time per record of the compiled module is half that of an interpretive system, the cost of generating the access module is repaid after seven records have been fetched.)

A final advantage of compilation is its simplifying effect on the system architecture. With both ad hoc queries and precanned transactions being treated in the same way, most of the code in the system can be made to serve a dual purpose. This ties in very well with our objective of supporting a uniform syntax between query users and transaction programs.

Available Access Paths

As described earlier, the principal access path used in System R for retrieving data associatively by its value is the B-tree index. A typical index is illustrated in Figure 6. If we assume a fan-out of approximately 200 at each level of the tree, we can index up to 40,000 records by a two-level index, and up to 8,000,000 records by a three-level index. If we wish to begin an associative scan through a large table, three I/Os will typically be required (assuming the root page is referenced frequently enough to remain in the system buffers, we need an I/O for the intermediate-level index page, the "leaf" index page, and the data page). If several records are to be fetched using the index scan, the three start-up I/Os are relatively insignificant. However, if only one record is to be fetched, other access techniques might have provided a quicker path to the stored data.

Two common access techniques which were not utilized for user data in System R are hashing and direct links (physical pointers from one record to another). Hashing was not used because it does not have the convenient ordering property of a B-tree index (e.g., a B-tree index on SALARY enables a list of employees ordered by SALARY to be retrieved very easily). Direct links, although they were implemented at the RSS level, were not used as an access path for user data by the RDS for a two-fold reason. *Essential links* (links whose semantics are not known to the system but which are connected directly by users) were rejected because they were inconsistent with the nonnavigational user interface of a relational system, since they could not be used as access paths by an automatic optimizer. *Nonessential links* (links which connect records to other records with matching data values) were not implemented because of the difficulties in automatically maintaining their connections. When a record is updated, its connections on many links may need to be updated as well, and this may involve many "subsidiary queries" to find the other records which are involved in these connections. Problems also arise relating to records which have no matching partner record on the link, and records whose link-controlling data value is null.

In general, our experience showed that indexes could be used very efficiently in queries and transactions which access many records,

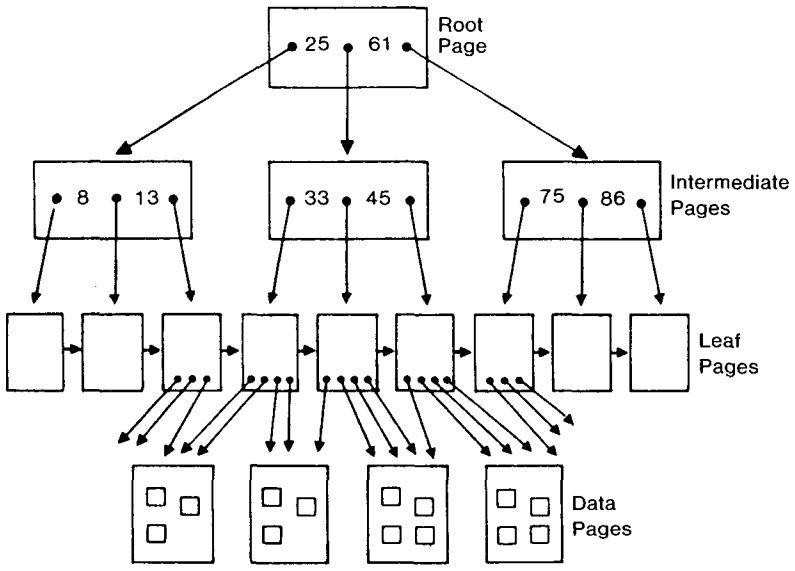

Fig. 6. A B-Tree Index.

but that hashing and links would have enhanced the performance of "canned transactions" which access only a few records. As an illustration of this problem, consider an inventory application which has two tables: a PRODUCTS table, and a much larger PARTS table which contains data on the individual parts used for each product. Suppose a given transaction needs to find the price of the heating element in a particular toaster. To execute this transaction, System R might require two I/Os to traverse a two-level index to find the toaster record, and three more I/Os to traverse another three-level index to find the heating element record. If access paths based on hashing and direct links were available, it might be possible to find the toaster record in one I/O via hashing, and the heating element record in one more I/O via a link. (Additional I/Os would be required in the event of hash collisions or if the toaster parts records occupied more than one page.) Thus, for this very simple transaction hashing and links might reduce the number of I/Os from five to three, or even two. For transactions which retrieve a large set of records, the additional I/Os caused by indexes compared to hashing and links are less important.

The Optimizer

A series of experiments was conducted at the San Jose IBM Research Laboratory to evaluate the success of the System R optimizer in choosing among the available access paths for typical SQL statements. The results of these experiments are reported in [6]. For the purpose of the experiments, the optimizer was modified in order to observe its behavior. Ordinarily, the optimizer searches through a tree of path choices, computing estimated costs and pruning the tree until it arrives at a single preferred access path. The optimizer was modified in such a way that it could be made to generate the complete tree of access paths, without pruning, and to estimate the cost of each path (cost is defined as a weighted sum of page fetches and RSS calls). Mechanisms were also added to the system whereby it could be forced to execute an SQL statement by a particular access path and to measure the actual number of page fetches and RSS calls incurred. In this way, a comparison can be made between the optimizer's predicted cost and the actual measured cost for various alternative paths.

In [6], an experiment is described in which ten SQL statements, including some single-table queries and some joins, are run against a test database. The database is artificially generated to conform to the two basic assumptions of the System R optimizer: (1) the values in each column are uniformly distributed from some minimum to some maximum value; and (2) the distribution of values of the various columns are independent of each other. For each of the ten SQL statements, the ordering of the predicted costs of the various access paths was the same as the ordering of the actual measured costs (in a few cases the optimizer predicted two paths to have the same cost when their actual costs were unequal but adjacent in the ordering).

Although the optimizer was able to correctly order the access paths in the experiment we have just described, the magnitudes of the predicted costs differed from the measured costs in several cases. These discrepancies were due to a variety of causes, such as the optimizer's inability to predict how much data would remain in the system buffers during sorting.

The above experiment does not address the issue of whether or not a very good access path for a given SQL statement might be overlooked because it is not part of the optimizer's repertoire. One such example is known. Suppose that the database contains a table T in which each row has a unique value for the field SEQNO, and suppose that an index exists on SEQNO. Consider the following SQL query:

```
SELECT * FROM T WHERE SEQNO IN
(15, 17, 19, 21);
```

This query has an answer set of (at most) four rows, and an obvious method of processing it is to use the SEQNO index repeatedly: first to find the row with SEQNO = 15, then SEQNO = 17, etc. However, this access path would not be chosen by System R, because the optimizer is not presently structured to consider multiple uses of an index within a single query block. As we gain more experience with access path selection, the optimizer may grow to encompass this and other access paths which have so far been omitted from consideration.

Views and Authorization

Users generally found the System R mechanisms for defining views and controlling authorization to be powerful, flexible, and convenient. The following features were considered to be particularly beneficial:

(1) The full query power of SQL is made available for defining new views of data (i.e., any query may be defined as a view). This makes it possible to define a rich variety of views, containing joins, subqueries, aggregation, etc., without having to learn a separate "data definition language." However, the view mechanism is not completely transparent to the end user, because of the restrictions described earlier (e.g., views involving joins of more than one table are not updateable).

(2) The authorization subsystem allows each installation of System R to choose a "fully centralized policy" in which all tables are created and privileges controlled by a central administrator; or a "fully decentralized policy" in which each user may create tables and control access to them; or some intermediate policy.

During the two-year evaluation of System R, the following suggestions were made by users for improvement of the view and authorization subsystems:

(1) The authorization subsystem could be augmented by the concept of a "group" of users. Each group would have a "group administrator" who controls enrollment of new members in the group. Privileges could then be granted to the group as a whole rather than to each member of the group individually.

(2) A new command could be added to the SQL language to change the ownership of a table from one user to another. This suggestion is more difficult to implement than it seems at first glance, because the owner's name is part of the fully qualified name of a table (i.e., two tables owned by Smith and Jones could be named SMITH.PARTS and JONES.PARTS). References to the table SMITH.PARTS might exist in many places, such as view definitions and compiled programs. Finding and changing all these references would be difficult (perhaps impossible, as in the case of users' source programs which are not stored under System R control).

(3) Occasionally it is necessary to reload an existing table in the database (e.g., to change its physical clustering properties). In System R this is accomplished by dropping the old table definition, creating a new table with the same definition, and reloading the data into the new table. Unfortunately, views and authorizations defined on the table are lost from the system when the old definition is dropped, and therefore they both must be redefined on the new table. It has been suggested that views and authorizations defined on a dropped table might optionally be held "in abeyance" pending reactivation of the table.

The Recovery Subsystem

The combined "shadow page" and log mechanism used in System R proved to be quite successful in safeguarding the database against media, system, and transaction failures. The part of the recovery subsystem which was observed to have the greatest impact on system performance was the keeping of a shadow page for each updated page. This performance impact is due primarily to the following factors:

(1) Since each updated page is written out to a new location on disk, data tends to move about. This limits the ability of the system to cluster related pages in secondary storage to minimize disk arm movement for sequential applications.

(2) Since each page can potentially have both an "old" and "new" version, a directory must be maintained to locate both versions of each page. For large databases, the directory may be large enough to require a paging mechanism of its own.

(3) The periodic checkpoints which exchange the "old" and "new" page pointers generate I/O activity and consume a certain amount of CPU time.

A possible alternative technique for recovering from system failures would dispense with the concept of shadow pages, and simply keep a log of all database updates. This design would require that all updates be written out to the log before the updated page migrates to disk from the system buffers. Mechanisms could be developed to minimize I/Os by retaining updated pages in the buffers until several pages are written out at once, sharing an I/O to the log.

The Locking Subsystem

The locking subsystem of System R provides each user with a choice of three levels of isolation from other users. In order to explain the three levels, we define "uncommitted data" as those records which have been updated by a transaction that is still in progress (and therefore still subject to being backed out). Under no circumstances can a transaction, at any isolation level, perform updates on the uncommitted data of another transaction, since this might lead to lost updates in the event of transaction backout.

The three levels of isolation in System R are defined as follows:

Level 1: A transaction running at Level 1 may read (but not update) uncommitted data. Therefore, successive reads of the same record by a Level-1 transaction may not give consistent values. A Level-1 transaction does not attempt to acquire any locks on records while reading.

Level 2: A transaction running at Level 2 is protected against reading uncommitted data. However, successive reads at Level 2 may still yield inconsistent values if a second transaction updates a given record and then terminates between the first and second reads by the Level-2 transaction. A Level-2 transaction locks each record before reading it to make sure it is committed at the time of the read, but then releases the lock immediately after reading.

Level 3: A transaction running at Level 3 is guaranteed that successive reads of the same record will yield the same value. This guarantee is enforced by acquiring a lock on each record read by a Level-3 transaction and holding the lock until the end of the transaction. (The lock acquired by a Level-3 reader is a "share" lock which permits other users to read but not update the locked record.)

It was our intention that Isolation Level 1 provide a means for very quick scans through the database when approximate values were acceptable, since Level-1 readers acquire no locks and should never need to wait for other users. In practice, however, it was found that Level-1 readers did have to wait under certain circumstances while the physical consistency of the data was suspended (e.g., while indexes or pointers were being adjusted). Therefore, the potential of Level 1 for increasing system concurrency was not fully realized.

It was our expectation that a tradeoff would exist between Isolation Levels 2 and 3 in which Level 2 would be "cheaper" and Level 3 "safer." In practice, however, it was observed that Level 3 actually involved less CPU overhead than Level 2, since it was simpler to acquire locks and keep them than to acquire locks and immediately release them. It is true that Isolation Level 2 permits a greater degree of

access to the database by concurrent readers and updaters than does Level 3. However, this increase in concurrency was not observed to have an important effect in most practical applications.

As a result of the observations described above, most System R users ran their queries and application programs at Level 3, which was the system default.

The Convoy Phenomenon

Experiments with the locking subsystem of System R identified a problem which came to be known as the "convoy phenomenon" [9]. There are certain high-traffic locks in System R which every process requests frequently and holds for a short time. Examples of these are the locks which control access to the buffer pool and the system log. In a "convoy" condition, interaction between a high-traffic lock and the operating system dispatcher tends to serialize all processes in the system, allowing each process to acquire the lock only once each time it is dispatched.

In the VM/370 operating system, each process in the multiprogramming set receives a series of small "quanta" of CPU time. Each quantum terminates after a preset amount of CPU time, or when the process goes into page, I/O, or lock wait. At the end of the series of quanta, the process drops out of the multiprogramming set and must undergo a longer "time slice wait" before it once again becomes dispatchable. Most quanta end when a process waits for a page, an I/O operation, or a low-traffic lock. The System R design ensures that no process will ever hold a high-traffic lock during any of these types of wait. There is a slight probability, however, that a process might go into a long "time slice wait" while it is holding a high-traffic lock. In this event, all other dispatchable processes will soon request the same lock and become enqueued behind the sleeping process. This phenomenon is called a "convoy."

In the original System R design, convoys are stable because of the protocol for releasing locks. When a process *P* releases a lock, the locking subsystem grants the lock to the first waiting process in the queue (thereby making it unavailable to be reacquired by *P*). After a short time, *P* once again requests the lock, and is forced to go to the end of the convoy. If the mean time between requests for the high-traffic lock is 1,000 instructions, each process may execute only 1,000 instructions before it drops to the end of the convoy. Since more than 1,000 instructions are typically used to dispatch a process, the system goes into a "thrashing" condition in which most of the cycles are spent on dispatching overhead.

The solution to the convoy problem involved a change to the lock release protocol of System R. After the change, when a process *P* releases a lock, all processes which are enqueued for the lock are made dispatchable, but the lock is not granted to any particular process. Therefore, the lock may be regranted to process *P* if it makes a subsequent request. Process *P* may acquire and release the lock many times before its time slice is exhausted. It is highly probable that process *P* will not be holding the lock when it goes into a long wait. Therefore, if a convoy should ever form, it will most likely evaporate as soon as all the members of the convoy have been dispatched.

Additional Observations

Other observations were made during the evaluation of System R and are listed below:

(1) When running in a "canned transaction" environment, it would be helpful for the system to include a data communications front end to handle terminal interactions, priority scheduling, and logging and restart at the message level. This facility was not included in the System R design. Also, space would be saved and the working set reduced if several users executing the same "canned transaction" could share a common access module. This would require the System R code generator to produce reentrant code. Approximately half the space occupied by the multiple copies of the access module could be saved by this method, since the other half consists of working storage which must be duplicated for each user.

(2) When the recovery subsystem attempts to take an automatic checkpoint, it inhibits the processing of new RSS commands until all users have completed their current RSS command; then the checkpoint is taken and all users are allowed to proceed. However, certain RSS commands potentially involve long operations, such as sorting a file. If these "long" RSS operations were made interruptible, it would avoid any delay in performing checkpoints.

(3) The System R design of automatically maintaining a system catalog as part of the on-line database was very well liked by users, since it permitted them to access the information in the catalog with exactly the same query language they use for accessing other data.

5. Conclusions

We feel that our experience with System R has clearly demonstrated the feasibility of applying a relational database system to a real production environment in which many concurrent users are performing a mixture of ad hoc queries and repetitive transactions. We believe that the high-level user interface made possible by the relational data model can have a dramatic positive effect on user productivity in developing new applications, and on the data independence of queries and programs. System R has also demonstrated the ability to support a highly dynamic database environment in which application requirements are rapidly changing.

In particular, System R has illustrated the feasibility of compiling a very high-level data sublanguage, SQL, into machine-level code. The

result of this compilation technique is that most of the overhead cost for implementing the high-level language is pushed into a "precompilation" step, and performance for canned transactions is comparable to that of a much lower level system. The compilation approach has also proved to be applicable to the ad hoc query environment, with the result that a unified mechanism can be used to support both queries and transactions.

The evaluation of System R has led to a number of suggested improvements. Some of these improvements have already been implemented and others are still under study. Two major foci of our continuing research program at the San Jose laboratory are adaptation of System R to a distributed database environment, and extension of our optimizer algorithms to encompass a broader set of access paths.

Sometimes questions are asked about how the performance of a relational database system might compare to that of a "navigational" system in which a programmer carefully hand-codes an application to take advantage of explicit access paths. Our experiments with the System R optimizer and compiler suggest that the relational system will probably approach but not quite equal the performance of the navigational system for a particular, highly tuned application, but that the relational system is more likely to be able to adapt to a broad spectrum of unanticipated applications with adequate performance. We believe that the benefits of relational systems in the areas of user productivity, data independence, and adaptability to changing circumstances will take on increasing importance in the years ahead.

Acknowledgments

From the beginning, System R was a group effort. Credit for any success of the project properly belongs to the team as a whole rather than to specific individuals.

The inspiration for constructing a relational system came primarily from E. F. Codd, whose landmark paper [22] introduced the relational model of data. The manager of the project through most of its existence was W. F. King.

In addition to the authors of this paper, the following people were associated with System R and made important contributions to its development:

M. Adiba	M. Mresse
R.F. Boyce	J.F. Nilsson
A. Chan	R.L. Obermarck
D.M. Choy	D. Stott Parker
K. Eswaran	D. Portal
R. Fagin	N. Ramsperger
P. Fehder	P. Reisner
T. Haerder	P.R. Roever
R.H. Katz	R. Selinger
W. Kim	H.R. Strong
H. Korth	P. Tiberio
P. McJones	V. Watson
D. McLeod	R. Williams

References

1. Adiba, M.E., and Lindsay, B.G. Database snapshots. IBM Res. Rep. RJ2772, San Jose, Calif., March 1980.

2. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.

3. Astrahan, M.M., and Lorie, R.A. SEQUEL-XRM: A Relational System. Proc. ACM Pacific Regional Conf., San Francisco, Calif., April 1975, p. 34.

4. Astrahan, M.M., et al. System R: A relational approach to database management. *ACM Trans. Database Syst.1*, 2 (June 1976) 97–137.

5. Astrahan, M.M., et al. System R: A relational data base management system. *IEEE Comptr. 12*, 5 (May 1979), 43–48.

6. Astrahan, M.M., Kim, W., and Schkolnick, M. Evaluation of the System R access path selection mechanism. Proc. IFIP Congress, Melbourne, Australia, Sept. 1980, pp. 487–491.

7. Blasgen, M.W., Eswaran, K.P. Storage and access in relational databases. *IBM Syst. J. 16*, 4 (1977), 363–377.

8. Blasgen, M.W., Casey, R.G., and Eswaran, K.P. An encoding method for multifield sorting and indexing. *Comm. ACM 20*, 11 (Nov. 1977), 874–878.

9. Blasgen, M., Gray, J., Mitoma, M., and Price, T. The convoy phenomenon. *Operating Syst. Rev. 13*, 2 (April 1979), 20–25.

10. Blasgen, M.W., et al. System R: An architectural overview. *IBM Syst. J. 20*, 1 (Feb. 1981), 41–62.

11. Bjorner, D., Codd, E.F., Deckert, K.L., and Traiger, I.L. The Gamma Zero N-ary relational data base interface. IBM Res. Rep. RJ1200, San Jose, Calif., April 1973.

12. Boyce, R.F., and Chamberlin, D.D. Using a structured English query language as a data definition facility. IBM Res. Rep. RJ1318, San Jose, Calif., Dec. 1973.

13. Boyce, R.F., Chamberlin, D.D., King, W.F., and Hammer, M.M. Specifying queries as relational expressions: The SQUARE data sublanguage. *Comm. ACM 18*, 11 (Nov. 1975), 621–628.

14. Chamberlin, D.D., and Boyce, R.F. SEQUEL: A structured English query language. Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich., May 1974, pp. 249–264.

15. Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, authorization, and locking in a relational database system. Proc. 1975 Nat. Comptr. Conf., Anaheim, Calif., pp. 425–430.

16. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. and Develop. 20*, 6 (Nov. 1976), 560–575 (also see errata in Jan. 1977 issue).

17. Chamberlin, D.D. Relational database management systems. *Comptng. Surv. 8*, 1 (March 1976), 43–66.

18. Chamberlin, D.D., et al. Data base system authorization. In *Foundations of Secure Computation*, R. Demillo, D. Dobkin, A. Jones, and R. Lipton, Eds., Academic Press, New York, 1978, pp. 39–56.

19. Chamberlin, D.D. A summary of user experience with the SQL data sublanguage. Proc. Internat. Conf. Data Bases, Aberdeen, Scotland, July 1980, pp. 181–203 (also IBM Res. Rep. RJ2767, San Jose, Calif., April 1980).

20. Chamberlin, D.D., et al. Support for repetitive transactions and ad-hoc queries in System R. *ACM Trans. Database Syst. 6*, 1 (March 1981), 70–94.

21. Chamberlin, D.D., Gilbert, A.M., and Yost, R.A. A history of System R and SQL/data system (presented at the Internat. Conf. Very Large Data Bases, Cannes, France, Sept. 1981).

22. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.

23. Codd, E.F. Further normalization of the data base relational model. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 33–64.

24. Codd, E.F. Recent investigations in relational data base systems. Proc. IFIP Congress, Stockholm, Sweden, Aug. 1974.

25. Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.

26. Comer, D. The ubiquitous B-Tree. *Compting. Surv. 11*, 2 (June 1979), 121–137.

27. Date, C.J. *An Introduction to Database Systems.* 2nd Ed., Addison-Wesley, New York, 1977.

28. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48–68.

29. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. On the notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.

30. Fagin, R. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 262–278.

31. Fagin, R. On an authorization mechanism. *ACM Trans. Database Syst. 3*, 3 (Sept. 1978), 310–319.

32. Gray, J.N., and Watson, V. A shared segment and inter-process communication facility for VM/370. IBM Res. Rep. RJ1579, San Jose, Calif., Feb. 1975.

33. Gray, J.N., Lorie, R.A., and Putzolu, G.F. Granularity of locks in a large shared database. Proc. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428–451.

34. Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. Proc. IFIP Working Conf. Modelling of Database Management Systems, Freudenstadt, Germany, Jan. 1976, pp. 695–723 (also IBM Res. Rep. RJ1654, San Jose, Calif.).

35. Gray, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course*, Goos and Hartmanis, Eds., Springer-Verlag, New York, 1978, pp. 393–481 (also IBM Res. Rep. RJ2188, San Jose, Calif.).

36. Gray, J.N., et al. The recovery manager of a data management system. IBM Res. Rep. RJ2623, San Jose, Calif., June 1979.

37. Griffiths, P.P., and Wade, B.W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 242–255.

38. Katz, R.H., and Selinger, R.D. Internal comm., IBM Res. Lab., San Jose, Calif., Sept. 1978.

39. Kwan, S.C., and Strong, H.R. Index path length evaluation for the research storage system of System R. IBM Res. Rep. RJ2736, San Jose, Calif., Jan. 1980.

40. Lorie, R.A. XRM—An extended (N-ary) relational memory. IBM Tech. Rep. G320-2096, Cambridge Scientific Ctr., Cambridge, Mass., Jan. 1974.

41. Lorie, R.A. Physical integrity in a large segmented database. *ACM Trans. Database Syst. 2*, 1 (March 1977), 91–104.

42. Lorie, R.A., and Wade, B.W. The compilation of a high level data language. IBM Res. Rep. RJ2598, San Jose, Calif., Aug. 1979.

43. Lorie, R.A., and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. and Develop. 23*, 3 (May 1979), 286–298.

44. Reisner, P., Boyce, R.F., and Chamberlin, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. AFIPS Nat. Comptr. Conf., Anaheim, Calif., May 1975, pp. 447–452.

45. Reisner, P. Use of psychological experimentation as an aid to development of a query language. *IEEE Trans. Software Eng. SE-3*, 3 (May 1977), 218–229.

46. Schkolnick, M., and Tiberio, P. Considerations in developing a design tool for a relational DBMS. Proc. IEEE COMPSAC 79, Nov. 1979, pp. 228–235.

47. Selinger, P.G., et al. Access path selection in a relational database management system. Proc. ACM SIGMOD Conf., Boston, Mass., June 1979, pp. 23–34.

48. Stonebraker, M. Implementation of integrity constraints and views by query modification. Tech. Memo ERL-M514, College of Eng., Univ. of Calif. at Berkeley, March 1975.

49. Strong, H.R., Traiger, I.L., and Markowsky, G. Slide Search. IBM Res. Rep. RJ2274, San Jose, Calif., June 1978.

50. Traiger, I.L., Gray J.N., Galtieri, C.A., and Lindsay, B.G. Transactions and consistency in distributed database systems. IBM Res. Rep. RJ2555, San Jose, Calif., June 1979.