System R is an experimental data base management system that was designed to be unusually easy to use. System R supports a high-level relational user language called SQL, which may be used by ad hoc users at terminals or by programmers as an imbedded data sublanguage in PL/I or COBOL. This paper describes the overall architecture of the system, including the Relational Data System (RDS) and the Research Storage System (RSS).

RDS is a data base language compiler. Host language programs with imbedded SQL statements are compiled by System R, which replaces the SQL statements with calls to a machine-language access module. The compilation approach removes much of the work of parsing, name binding, and optimization from the path of a running program, enabling highly efficient support for repetitive transactions. In contrast, the RSS is a low-level DBMS, supporting simple record-at-a-time operators, but with rather sophisticated transaction management, recovery, and concurrency control.

System R: An architectural overview

by M. W. Blasgen, M. M. Astrahan, D. D. Chamberlin, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, I. L. Traiger, B. W. Wade, and R. A. Yost

System R is an experimental data base management system designed and built at the IBM San Jose Research Laboratory as part of a program of research in the relational model of data. The architecture of System R was first described in Reference 1, and SQL, its user interface, was described in Reference 2. Since these papers were published, System R has undergone certain architectural changes, and implementation of the prototype system is now essentially complete. The purpose of this paper is to describe the system—its goals, its design, and achievements—in a single report. More detailed reports describing specific aspects of the system are listed in the bibliography.

The paper is divided into three sections. The first section clarifies the goals of the system and introduces the features of the system.

The architecture is described in the second and third sections, covering, respectively, the two system layers: the Relational Data System and the Research Storage System.

Description of System R features

A major impediment to the widespread use of computerized data management systems is the cost and complexity of understanding and using such systems. System R is an experimental data base system which is easy to understand and use. The system adopts a relational data model and supports the language called SQL for defining, accessing, and modifying multiple views of stored tables. It provides a sophisticated authorization facility, and automatically handles systems functions such as recovery and concurrency control.

the SQL language

All access to data in System R is through SQL (formerly known as SEQUEL2), a relational data base language which is described in Reference 2. An example relational data base describing employees and offices in a company appears in Figure 1. Examples of the use of SQL follow, using this simple data base.

Figure 1 A fragment of a relational data base

EMPLOYEE

NAME	OFFICE	JOB	SALARY
SMITH	PARIS	SALES	15000
JONES	BONN	SALES	18000
CLARK	BOISE	SALES	12000
JONES	BOSTON	SERVICE	17000
KENT	PARIS	SERVICE	15000
DAVIS	LONDON	SERVICE	13000
JACOB	RIO	SALES	12000
...			

OFFICE

LOCATION	MANAGER	PHONE
SAN JOSE	BLASGEN	7152
PARIS	PORTAL	9123
LONDON	PORTAL	3278
BONN	ROEVER	1287
...		

Q1: Find the names of employees in the Paris office.

```
SELECT NAME
FROM EMPLOYEE
WHERE OFFICE = 'PARIS'
```

Q2: List all the different offices in the EMPLOYEE table.

```
SELECT UNIQUE(OFFICE)
FROM EMPLOYEE
```

Q3: Find the employees who work in an office managed by Roever. Using a "nested query," we obtain:

```
SELECT NAME, OFFICE, JOB
FROM EMPLOYEE
WHERE OFFICE IN
        (SELECT LOCATION
          FROM OFFICE
          WHERE MANAGER = 'ROEVER')
```

or alternatively we may "join" the tables:

```
SELECT NAME, OFFICE, JOB
FROM EMPLOYEE, OFFICE
WHERE EMPLOYEE.OFFICE = OFFICE.LOCATION
        AND MANAGER = 'ROEVER'
```

Q4: List all the offices and the average salary of employees in each.

```
SELECT OFFICE, AVG(SAL)
FROM EMPLOYEE
GROUP BY OFFICE
```

Q5: Print out a sorted list of employees in Paris, with their salaries.

```
SELECT NAME, SAL
FROM EMPLOYEE
WHERE OFFICE = 'PARIS'
ORDER BY NAME
```

Q6: Insert a new employee into the EMPLOYEE table.

```
INSERT INTO EMPLOYEE(NAME, OFFICE, JOB):
<'WADE', 'SAN JOSE', 'SERVICE'>
```

(sets SALARY to null)

Q7: Close the Paris office.

```
DELETE EMPLOYEE
WHERE OFFICE = 'PARIS'

DELETE OFFICE
WHERE LOCATION = 'PARIS'
```

Q8: Give a ten percent raise to the service people in Bonn.

```
UPDATE EMPLOYEE
SET SAL = SAL*1.1
WHERE JOB = 'SERVICE'
AND OFFICE = 'BONN'
```

ad hoc query and host language support

One of the basic goals of System R is to support two different types of processing against a data base: (1) ad hoc queries and updates, which are usually executed only once, and (2) canned programs, which are installed in a program library and executed hundreds of times. System R makes all the features of SQL available in both these environments. These features include statements to query and update a data base, to define and delete data base objects such as tables, views, and indexes, and to control access to the data base by various users.

An ad hoc user at a terminal may type SQL statements and view the result directly at a terminal, as in the examples above. Alternatively, the same SQL statements may be imbedded in a PL/I or COBOL program by prefixing them with dollar signs to distinguish them from host-language statements. SQL statements in PL/I or COBOL programs may contain host-language variables if the variable names are prefixed by dollar signs, as in the following example:

```
$UPDATE EMPLOYEE SET SALARY = $X WHERE NAME = $Y;
```

If a PL/I or COBOL program wishes to execute an SQL query and fetch the result, the answer set is readied for retrieval by an OPEN statement, which binds the values of any host-language variables appearing in the query. Then a FETCH statement is used repeatedly to fetch rows from the answer set into the designated program variables, as in the following example:

```
$LET PEOPLE BE
      SELECT NAME, SALARY
      INTO $X, $Y
      FROM EMPLOYEE
      WHERE JOB = $Z;

$OPEN PEOPLE; /*BINDS VALUE OF Z */
$FETCH PEOPLE; /* FETCHES ONE EMPLOYEE INTO X AND Y */
$CLOSE PEOPLE; /* AFTER ALL VALUES HAVE BEEN FETCHED */
```

After the execution of each SQL statement, a status code is returned to the host program in a variable called SYRCODE.

data independence

SQL allows data accesses and updates to be expressed without mentioning or implying the existence of specific access paths (access paths are techniques for finding the relevant data using, for example, an index on a particular column) or the physical layout of data. This has the advantage of making application programs simpler and also allows the data management system to choose an optimal strategy for evaluating the program.

For example, to determine if manager Portal has a service person in his office, one invokes the following query:

```
SELECT NAME
FROM EMPLOYEE, OFFICE
WHERE EMPLOYEE.OFFICE = OFFICE.LOCATION
AND OFFICE.MANAGER = 'PORTAL'
AND EMPLOYEE.JOB = 'SERVICE'
```

Since the language specifies only *what* is desired, and not *how* to obtain it, the system has several choices. For example, one strategy is to search EMPLOYEE looking for service people, and for each such entry, use the corresponding OFFICE to enter into the OFFICE table to see if that employee works for Portal. Another strategy involves first searching OFFICE to find what LOCATIONS Portal manages and then searching EMPLOYEE for service people at those locations. Other strategies involve sorting one or both tables.

The System R optimizer is responsible for selecting the strategy that minimizes the "cost" of carrying out an SQL statement. Cost is based on estimates of CPU and I/O requirements. Using an optimizer in this way has two benefits: First, the user need not be concerned with storage details, and thus may be more produc-

tive. Second, the user is prohibited from "taking advantage" of knowledge of such details. The second benefit allows the program to continue functioning as the underlying storage structures evolve with time.

In general, since System R supports a very high-level language, most data base structuring issues can be deferred until after the applications are written. This "install now, tune later" philosophy also eases application programming by deferring many performance decisions.

views and authorization

The result of any SQL query is itself a table. Such a table may be materialized immediately, or the definition may be stored as a view. Views may be used just like other tables except that certain views (involving, for example, join) cannot be modified.

Views extend the notion of data independence even further, permitting the user to be isolated not only from storage details (indexes, pointers) but also from the set of tables currently stored. If the structure of a table is changed (columns added or permuted or a table split into two tables) then a view may be defined that appears to users like the original table. Old programs can access the new data via the view.

Views also provide a powerful authorization mechanism. Rather than allowing users access to an entire table, one may define a view which is a row and column subset of the table and only allow access to that view. For example, one might allow managers to see only records in their own departments. Further, one may qualify certain columns of the view as read-only. In order to allow for either centralized or distributed control of access, a special privilege called "grant" is also included, which allows one to grant any subset of capabilities to other users. Each such operation may pass on the "grant" privilege as well.

integrated data base catalogs

SQL is an integrated data definition and data manipulation language. In System R the description of the data base is stored in user-visible "system" tables which may be read using the SQL language. The creation of a table results in new entries in these system tables. Users defining tables are encouraged to include English text that describes the "meanings" of the tables and their columns. Later, others may retrieve all tables with certain attributes or may browse among the descriptions of defined tables (if they are so authorized).

compilation

A major criticism of nonprocedural languages is that they have a great potential for execution inefficiency. If the use of SQL caused a large degradation in performance, System R would be of little interest. Therefore, the design has concentrated on performance as well as on function. To provide acceptable performance, Sys-

tem R supports the SQL language by *compiling* statements in SQL to machine code which contains calls to low-level accessing routines. As an example, during compilation of a user program, the user's authorization is checked for each SQL statement in the program. When the program is loaded for execution, one check is made to verify that the user's authorization to run the program remains in effect. No authorization checking is necessary on the execution of each SQL statement, so run-time overhead is minimized.

Experiments indicate that compilation is almost uniformly superior to interpretation, even for those SQL statements that are executed only once and retrieve or modify only a few records.

dynamic data base definition

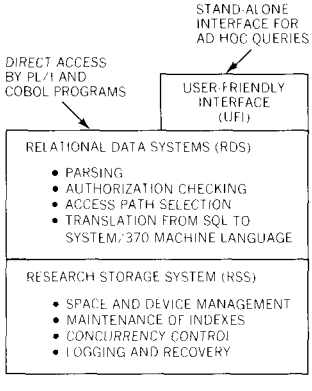
Any of the following can be done by an authorized user at any time without interrupting the normal operation of the system:

- Create and destroy tables
- Create and destroy indexes on tables
- Add a column to an existing table
- Install a new transaction
- Add users to the system
- Change the privileges held by various users
- Define or drop a view of existing data

transaction management

A major goal of System R is to provide a full set of capabilities for data base management in a realistic, operational environment. Only in this way can the viability of the architecture be assessed. In particular, System R supports multiple users concurrently accessing data and has complete facilities for transaction backout and system recovery. Recovery compensates for system failures as well as catastrophic failures of the magnetic media (e.g., disk head crash). Almost all recovery information is kept on disk and a noncatastrophic restart is transparent to operations personnel.

Figure 2 The architecture of System R



The transaction notion is the key to a successful recovery philosophy. A transaction is a user-defined unit of work which may involve many SQL statements. If the system crashes during processing of a transaction, data in the data base may not be in a consistent state at the time of the crash. Therefore, the system must be able to "undo" partially completed transactions. Once a transaction terminates, its updates are committed and are no longer subject to being backed out in the event of a crash.

Transactions also supply the key to concurrency control. If multiple transactions concurrently read and write the same data, anomalies may arise. System R uses a locking protocol such that (1) the system itself never gets confused because of concurrent access to a data item by two or more transactions, and (2) the user can control the extent to which his transaction is isolated from the

effects of other transactions. The locking subsystem handles queuing and deadlock detection.

architecture

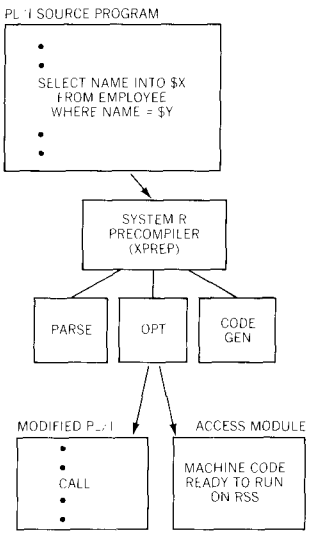
The System R architecture is shown in Figure 2. Functions are divided between two subsystems, the Relational Data System (RDS) and the Research Storage System (RSS). These two components are described in the following sections.

Relational Data System

RDS is split into two distinct functions: (1) a precompiler, called XPREP, which is used to precompile host-language programs and install them as "canned programs" under System R, and (2) an execution system, called XRDI, which controls the execution of these "canned programs" and also executes SQL statements for ad hoc terminal users.

When an application programmer has written a PL/I or COBOL program with imbedded SQL statements, his first step is to present the program to the System R precompiler, XPREP. XPREP finds the SQL statements in the program and translates them into a machine-language "access module." In the user's program, the SQL statements are replaced by host-language calls to the access module. The access module is stored in the System R data base to protect it from unauthorized modification. The precompilation step is illustrated in Figure 3.

Figure 3 Precompilation of a PL/I-SQL program



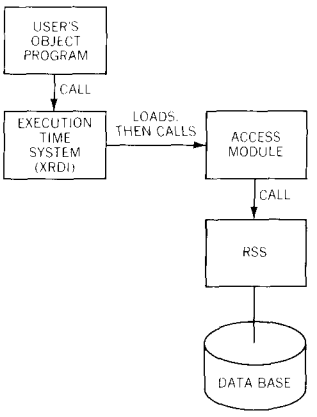
The advantages gained for canned programs by the precompilation step are twofold:

1. Much of the work of parsing, name-binding, access path selection, and authorization checking can be done once by the precompiler and thus removed from the process of running the canned program.
2. The access module, because it is tailored to one specific program, is much smaller and runs much more efficiently than a generalized SQL interpreter.

After precompilation, the user's program contains pure PL/I or COBOL and can be compiled using a conventional language compiler.

When a "canned program" is run on System R, it makes calls to XRDI, which in turn loads and invokes the access module for the program. The access module operates on the data base by making calls to RSS and delivers the result to the user's program. This process is illustrated in Figure 4.

Figure 4 Execution of a compiled program



The ad hoc user of System R is supported by an application program called the User-Friendly Interface (UFI), which controls dia-

logue management and the formatting of the display terminal. The UFI has an access module of its own, but its access module is not complete because the purpose of UFI is to execute SQL statements that are not known in advance. When a user enters an ad hoc SQL statement, UFI passes the statement to XRDI by means of special "PREPARE" and "EXECUTE" calls which will be described later. The effect of these calls is to cause a new "section" of the access module of UFI to be dynamically generated for the new statement. The dynamically generated part of the access module contains machine-language code and is in every way indistinguishable from the parts that were generated by the precompiler.

System R permits many users to be active simultaneously, performing a variety of activities. Some users may be precompiling new programs while others are running existing "canned programs." At the same time, other users may be using UFI, querying and updating the data base and creating new tables and views. All these simultaneous activities are supported by the automatic locking subsystem built into the RSS.

precompilation

When a PL/I or COBOL program with imbedded SQL statements is presented to the System R precompiler, it scans the program to find the SQL statements (they are indicated by dollar signs) and replaces each SQL statement with a valid host-language CALL. In addition, each SQL statement is put through a three-step process in order to translate it into a machine-language routine. The three steps are as follows:

1. Parsing: The parser checks the SQL statement for syntactic validity and translates it into a conventional parse-tree representation. The parser also returns to the System R precompiler two lists of host program variables found in the SQL statement: a list of input variables (values to be furnished by the calling program and used in processing the statement) and a list of output variables (target locations for data to be fetched by the statement). For example, if the SQL statement being parsed were as follows:

   ```
   SELECT NAME, SALARY INTO $X, $Y
   FROM EMPLOYEE WHERE OFFICE = $A AND JOB = $B
   ```

   the input variables would be A and B, and the output variables would be X and Y.

2. Optimization: The System R optimizer is then invoked with the parse tree as input. The optimizer performs several tasks:

   a. First, using the internal catalogs of System R, it resolves all symbolic names in the SQL statement to internal data base objects.

b. A check is made to ensure that the current user is authorized to perform the indicated operation on the indicated table(s).
c. If the SQL statement operates on one or more user-defined views, the definitions of the views (stored in parse tree form) are merged with the SQL statement to form a new composite SQL parse tree that operates on real stored tables.
d. The optimizer uses the system catalogs to find the set of available indexes and certain other statistical information on the tables to be processed. This information is used to choose an access path and an algorithm for processing the SQL statement. The design of this access path selection process is given briefly below and in more detail in Reference 3. The optimizer represents its chosen access path by means of an ASL (Access Specification Language)[4] specification and by construction of the RSS parameter lists to be used in processing the statement.

3. Code generation: The code generator translates the ASL structures produced by the optimizer into a System/370 machine-language routine that implements the chosen access path.[5] This machine-language routine is called a "section." When running, the section will access the data base by using the RSS parameter lists that were produced by the optimizer.

After all the SQL statements in a program have been translated into sections, the sections are collected together to form an access module. In addition to machine-language code, each section holds the SQL statement from which it was originally constructed, thus enabling the section to be rebuilt if its original access path should become unavailable at some future time. When the access module is complete, it is stored in the System R data base for later use.

After the System R precompiler has replaced all the SQL statements in the user's program with calls to XRDI, the program contains pure PL/I or COBOL, and it may be compiled using one of the conventional language compilers. The resulting object program is now ready to be run on System R.

optimization

Before proceeding to discuss how this program is executed, we describe in more detail how the access path selection portion of the optimizer works.

A *query block* is represented by a SELECT list, a FROM list, and a WHERE tree, containing, respectively, the list of items to be retrieved, the table(s) referenced, and the Boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one

operand that is itself a query. For each query block, an optimal access path is selected by the optimizer.

The optimizer carries out name resolution and view composition if necessary. View composition replaces all references to view tables and columns with their underlying definitions in terms of actual stored tables. If the view definition contained predicate restrictions, these are ANDed to the WHERE tree of this query. After view composition every table reference is a stored table.

Finally, the optimizer performs access path selection. (Reference 3 has a detailed discussion of this operation.) The optimizer examines both the predicates in the query and the access paths available on the tables referenced by the query and formulates a cost prediction for each access plan, using the following cost formula:

COST = PAGE FETCHES + W * (RSI CALLS).

This cost is a weighted measure of I/O operations (pages fetched) and CPU used (instructions executed). W is an adjustable weighting factor between I/O and CPU. RSI CALLS is a predicted number of records returned from the RSS to be used in evaluating this query. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus, the choice of a minimum cost path to process a query involves an attempt to minimize total resources required.

To compute the estimated costs, statistics are maintained in the System R catalogs and come from several sources. Initial table loading and index creation initialize these statistics. They are then updated periodically by an UPDATE STATISTICS command, which can be run by any authorized user. System R does not update these statistics at every INSERT, DELETE, or UPDATE because of the extra data base operations and the locking bottleneck it would create. Continuous maintenance of statistics would tend to serialize access to a table for users that modify the table contents.

Using these statistics, the optimizer develops estimates of the cost of carrying out the statement using a variety of access paths (indexes, sorting, and scanning). The cheapest access path is then selected.

executing a precompiled program

When a user invokes a program that has been precompiled on System R, the normal facilities of the operating system are used to load and start the object program. System R first becomes aware of the program when it makes its first call to XRDI. On the first such call, XRDI checks the authority of the current user to invoke the indicated access module, and checks that the access module is still valid. If these checks are successful, the access

module is loaded from the data base into virtual memory, and control is passed to the indicated section. On subsequent calls to the same access module, control passes directly to the indicated section. The machine language code in the section processes the original SQL statement from which it was compiled, using as needed the host-program variables which are passed with the call.

Since all name binding, authorization checking, and access path selection are done during the precompilation step, the resulting access module is dependent on the continued existence of the tables it operates on, the indexes it uses as access paths, and the privileges of its creator. Therefore, whenever a table or index is dropped or a privilege is revoked, System R automatically performs a search in its internal catalogs to find access modules that are affected by the change. If the change involves dropping a table or revoking a necessary privilege, the access module is erased from the data base. However, if the change involves merely dropping an index used by the access module, it will be possible to regenerate the access module by choosing an alternative access path. In this case, the access module is marked "invalid." When the access module is next invoked, the invalid marking is detected and the access module is regenerated automatically. The original SQL statement contained within each section is once again passed through the parser, the optimizer, and the code generator to produce a new section based on the currently available access paths. The newly regenerated access module is stored in the data base and also loaded into virtual memory for execution. The user's source program is not affected in any way, and the user is unaware of the regeneration process except for a slight delay during the initial loading of his access module.

It is possible that a user might attempt to change the data base in some way that would invalidate an access module while the access module was actually loaded and running. It would be undesirable if such a change were allowed to become effective while the running access module was in the middle of some operation. To prevent this from occurring, the loaded access module protects itself by holding a lock on its own description in the system catalog tables. Therefore, any data base change made by another concurrently running transaction that will invalidate the access module (changing its description from "valid" to "invalid") must wait until the lock is released.

treatment of "nonoptimizable" statements

For certain types of SQL statements, no significant choice of access path is required. These statements include those which create and drop tables and indexes, begin and end transactions, and grant and revoke privileges. The process of creating a new table, for example, involves placing a description of the table in the system catalogs. Since this process takes place essentially the same way for each new table, it is possible to build into System R

a standard routine for creating tables. It is then unnecessary to generate new machine code in an access module whenever a new table is to be created. Instead, the standard program is invoked and given the name of the table to be created and a list of its columns and their data types. This information is conveyed in the form of the SQL parse tree for the CREATE TABLE statement. We will refer to SQL statements that can be handled in this way as "nonoptimizable" statements.

When the System R precompiler encounters a nonoptimizable statement in a user program, it places the parse tree of the statement directly into the section of the access module rather than invoking the optimizer and code-generator. The resulting section is labeled as an "INTERPSECT," to distinguish it from a section containing machine code, which is labeled a "COMPILESECT."

At run time, when XRDI receives a call to execute a given section, it examines the label on the section. If it is a COMPILESECT, XRDI gives control directly to the section. If it is an INTERPSECT, XRDI determines the statement type by examining the root of the parse tree, then invokes the appropriate standard routine. The standard routine obtains its necessary inputs (e.g., table and column names) from the parse tree in the INTERPSECT.

dynamically defined statements

Some programs may need to execute SQL statements that were not known at the time the program was precompiled. An example of such a program is the "User-Friendly Interface" of System R, which allows users to type ad hoc SQL statements at a terminal, then executes them and displays the results. Another example is a general-purpose bulk loader program that loads data into tables via SQL INSERT statements but that does not know at precompilation time the name of the table to be loaded, or the number and data types of its columns.

The SQL language feature that supports this type of application is the PREPARE statement, which is an executable statement having the syntax:

PREPARE <statement-name> AS <variable>

For example, a programmer might write:

```
PREPARE S1 AS QSTRING;
```

This indicates to System R that, at run time, the character-type variable QSTRING will contain an SQL statement that should be optimized and associated with the name S1. QSTRING may contain any kind of SQL statement, and the SQL statement may have "parameters" indicated by question marks, such as:

```
UPDATE EMPLOYEE SET SALARY = ? WHERE NAME = ?
```

When the precompiler encounters a PREPARE statement in a program, it creates a section in the access module called an INDEFSECT.

A call to an INDEFSECT causes a dynamically defined SQL statement to pass through the parser, optimizer, and code generator. The result is a new COMPILESECT or INTERPSECT, which replaces the INDEFSECT in the access module. The dynamically defined statement is now ready to be executed like any other SQL statement.

After writing PREPARE S1 AS QSTRING, the programmer will want to execute the statement he has prepared. If the prepared statement was not a query, the programmer may use the following syntax:

EXECUTE <statement-name> [USING <variable-list>]

For example:

EXECUTE S1 USING $X, $Y

The precompiler will translate the EXECUTE statement into a call on the indicated section, passing the addresses of $X and $Y as parameters of the call. The section may be executed many times, with different parameters, without reinvoking the System R optimizer.

If the prepared SQL statement was a query (a SELECT statement), the COMPILESECT produced for it will look exactly like a COMPILESECT produced by the precompiler. Therefore, the program may proceed to fetch the result of the query using OPEN and FETCH statements much like those used with a query that was defined at precompile time. Details of this process are described more fully in Reference 6.

The Research Storage System

We now discuss the RSS, a low-level DBMS that provides underlying support for System R. The RSS supports the Research Storage Interface (RSI), which provides simple, record-at-a-time operators on base tables. Operators are also supported for data recovery, transaction management, and data definition. Calls to the RSI require explicit use of data areas called segments and access paths called indexes and links, along with the use of RSS-generated, numeric identifiers for data segments, tables, access paths, and records. The RDS handles the selection of efficient access paths to optimize its operations, and maps symbolic table names to their internal RSS identifiers. The RSI is a navigational interface and supports an object called a *scan* which can move from record to record along a specified access path.

In order to facilitate gradual data base integration and tuning of access paths, the RSS permits new stored tables or new indexes to be created at any time, or existing ones destroyed, without quiescing the system and without dumping and reloading the data. One can also add new fields to existing tables, or add or delete pointer chain paths across existing tables. This facility, coupled with the ability to retrieve any subset of fields in a record, provides a degree of data independence at a low level of the system, since existing access modules that execute RSI operations on records will be unaffected by the addition of new fields or access paths.

As a point of comparison, the RSS has many functions that can be found in other systems, both relational and nonrelational, such as the use of index and pointer chain structures. The areas that have been emphasized and extended in the RSS include dynamic definition of new data types and access paths, as described above, dynamic binding and unbinding of disk space to data segments, multiple levels of isolation from the actions of the other concurrent users, and automatic locking at the granularity of segments, tables, pages, or single records.

segments

In the RSS, all data is stored in a collection of logical address spaces called *segments*, which are employed to control physical clustering. Segments are used for storing user data, access path structures, internal catalog information, and intermediate results generated by the RDS. All the records of any table must reside within a single segment chosen by the RDS. However, a given segment may contain several tables.

The RSS has the responsibility for mapping logical segment spaces to physical extents on disk storage and for supporting segment recovery. Within the RSS, each segment consists of a sequence of 4096-byte *pages*. Disk space for pages is allocated dynamically, and pages are the transfer unit from disk to virtual storage. A page request is handled by allocating space within a virtual storage buffer shared among all concurrent users. Pages are fixed in their buffer slots until they are explicitly freed by RSS components. Freeing a page makes it available for replacement, and when space is needed, the buffer manager replaces whichever freed page was least recently used.

The RSS handles segment recovery by a novel technique that is described in Reference 7.

tables

The main data object of the RSS is the n-ary relation, alternatively called a table, which consists of a time-varying number of records, each containing n fields. A new table can be defined at any time within any segment chosen by the RDS. An existing table and its associated access path structures can be dropped at any time,

with all storage space made reusable. Even after a table is defined and loaded, new fields may be added on the right, without a data base reload and without immediately modifying existing records.

Operators are available to INSERT and DELETE single records, and to FETCH and UPDATE any combination of fields in a record. One can also fetch a sequence of records along an access path through the use of an RSS *scan*. Each scan is created by the RSS for fetching records on a particular access path through execution of the OPEN_SCAN operator. The records along the path may then be accessed by a sequence of NEXT operations on that scan. A scan may employ an *index*, which gives direct access and value-ordering according to one or more of the columns of a table (e.g., to retrieve all employees in a given department). A table may have as many indexes as desired. The RSS also provides a scan through the physical pages on which the data is stored, delivering records in a system-determined order. For all of these access paths, the RDS may attach a search argument (SARG) to each NEXT operation. The search argument may be any predicate involving atomic expressions of the form <field number, operator, value>. The value is an explicit byte string provided by the RDS, and the operator is "=", "¬=", "<", ">", "<=", or ">=". The RDS optimizer attempts, whenever possible, to place SQL predicates into RSS search arguments because of the performance advantage resulting from reduced interface crossing.

Associated with every record of a table is a record identifier called a TID. Each record identifier is generated by the RSS and is available to the RDS as a concise and efficient means of addressing records. TIDs are also used within the RSS to refer to records from index structures and to maintain pointer chains. However, they are not intended for end users above the RDS, since they may be reused by the RSS after record deletions, and are reassigned during data base reorganization.

In order to tune the data base to particular environments, the RSS accepts hints for physical allocation during INSERT operations in the form of a tentative TID. The new record will be inserted in the page associated with that TID if sufficient space is available. Otherwise, a nearby page is chosen by the RSS. Use of this facility enables the RDS to cluster records of a given table with respect to some criterion such as a value ordering on one or more fields.

indexes

An *index* in the RSS is an access path that provides a view of a table ordered with respect to values in one or more sort fields. Indexes combined with scans provide the ability to scan tables along a value ordering for low-level support of simple views. More importantly, an index provides associative access capability. By keying on the sort field values, the RDS can rapidly fetch a record using an index. The RDS can also open a scan at a particu-

lar point in the index and retrieve a sequence of records with a given range of sort values. The RDS can employ a disjunctive normal form search argument (a SARG) during scanning to further restrict the set of records that is returned. This facility is especially useful for situations where SQL search predicates involve several fields of a table and at least one of them has index support.

A new index can be defined at any time, on any combination of fields in a table. Furthermore, each of the fields may be specified as ascending or descending. Once defined, an index is maintained automatically by the RSS during all INSERT, DELETE, and UPDATE operations. An index can also be dropped at any time.

Each index is composed of one or more pages within the segment containing the table. A new page can be added to an index when needed, as long as one of the pages within the segment is marked as available. The pages for a given index are organized into a balanced hierarchical structure, in the style of B trees and of Key Sequenced Data Sets in IBM's Virtual Storage Access Method (VSAM).

In order to handle variable-length, multifield indexes efficiently, a special encoding scheme is employed on the field values so that the resulting concatenation can be compared against others for ordering and searching. This encoding eliminates the need for padding of each field and field-by-field comparison.[8]

sort

The RSS contains a sort component that sorts records from a table according to an RDS-provided order specification. The sort component can sort all of a table, or any row or column subset of a table, by one or more fields, in either ascending or descending order. Sort is carried out using a sort-merge algorithm, using a rapid internal sort of the records on a page, followed by a merge of the internally sorted pages.

transaction management

A *transaction* at the RSS is a sequence of RSI calls issued in behalf of one user. It also serves as a unit of consistency and recovery, as will be discussed below. An RSS transaction is delimited by the BEGIN TRANSACTION and END TRANSACTION commands. Various resources are assigned to transactions by the RSS, using the locking techniques described below. A transaction recovery scheme that allows a transaction to be backed out to the beginning of the transaction is provided.

Transaction recovery occurs when the RESTORE TRANSACTION command is issued by the RDS, or when the RSS initiates the procedure to handle deadlock. The effect is to undo all the changes made by the transaction, including all record and index modifications caused by INSERT, DELETE, and UPDATE operations, and all

the declarations for defining new tables and indexes. Finally, all locks on recoverable data that have been obtained are released.

The transaction recovery function is supported through the maintenance of time-ordered lists of log entries, which record information about each change to recoverable data. The entries for each transaction are chained together and include the old and new values of all modified objects. Modifications to index structures are not logged since their values can be determined from data values and index catalog information.

The log entries themselves are stored in a dedicated segment used as a ring buffer. This segment is treated as a simple linear byte space, with entries spanning page boundaries.

The RSS provides functions to recover the data base to a consistent state in the event of a system crash. By a consistent state we mean a set of data values that would result if a set of transactions had completed and no other transactions were in progress. At such a state all indexes and pointers are correct at the RSS level, and all user-defined semantics on data values are valid.

In the RSS, the system recovery mechanisms use disk storage to recover in the event of a "soft" failure that causes the contents of main memory to be lost but that does not damage secondary storage. This recovery technique is oriented toward frequent checkpoints and rapid recovery. A similar mechanism uses tape storage to recover in the relatively infrequent case where disk storage is destroyed, and is oriented toward less frequent checkpoints.

concurrency control

Since System R is a concurrent user system, locking techniques must be employed to solve various synchronization problems, both at the logical level of objects like tables and records and at the physical level of pages.

At the *logical* level, such classic situations as the Lost Update problem must be handled to ensure that two concurrent transactions do not read the same value and then try to write back an incremented value. If these transactions are not synchronized, the second update will overwrite the first, and the effect of one increment will be lost. Similarly, if a user wishes to read only "clean" or committed data, not "dirty" data which has been updated by a transaction still in progress and which may be backed out, then some mechanism must be invoked to check whether or not the data is dirty. For another example, if transaction recovery is to affect only the modifications of a single user, then mechanisms are needed to ensure that data updated by some ongoing transaction T1 is not updated by another transaction T2. Otherwise, the backout of transaction T1 will undo T2's update and violate the principle of isolated backout.

At the *physical* level of pages, locking techniques are required to ensure that internal components of the RSS give correct results. For example, a data page may contain several records, and each record is accessed through its record identifier, which requires following a pointer within the data page. Even if no logical conflict occurs between two transactions, because each is accessing a different table or a different record in the same table, a problem can occur at the physical level if one transaction follows a pointer to a record on some page, while the other transaction updates a second record on the same page and causes a data compaction routine to reassign record locations.

One basic decision in System R was to handle both logical and physical locking requirements within the RSS, rather than splitting the functions across the RDS and RSS subsystems. Physical locking is handled by setting and holding locks on one or more pages during the execution of a single RSI operation. Logical locking is handled by setting locks on such objects as segments, tables, TIDs and key value intervals and holding them either until they are explicitly released or to the end of the transaction. The main motivation for this decision is to facilitate the exploration of alternative locking techniques. (One particular alternative has already been included in the RSS as a tuning option, whereby the finest level of locking in a segment can be expanded to an entire page of data, rather than single records. This option allows pages to be locked for both logical and physical purposes by varying the duration of the lock.) Other motivations are to simplify the work of the RDS and to develop a complete, concurrent user RSS that can be tailored to future research applications.

For situations detected by the end user or RDS where locking large aggregates is desirable, the RSS also supports operators for placing explicit share or exclusive locks on entire segments or tables.

The RSS supports multiple levels of consistency that control the degree of isolation of a user from the actions of other concurrent users. When a transaction is started at the RSI, one of three consistency levels must be specified. (These same consistency levels are also reflected to the end user at the SQL level.) Different consistency levels may be chosen by different concurrent transactions. For all of these levels, the RSS guarantees that any data modified by the transaction is not modified by any other, until the given transaction ends. This rule is essential to our transaction recovery scheme, where the backout of modifications by one transaction does not affect modifications made by other transactions.

The differences in consistency levels occur during read operations. Level 1 consistency offers the least isolation from other

users but causes the lowest overhead and lock contention. With this level, "dirty data" (data which has been updated by a still-running transaction) may be read (but not, of course, updated) by a second transaction. It is clear that execution with Level 1 consistency incurs the risk of reading data values that in some sense never appeared if the transaction which set the data values is later backed out. Yet, this level may be entirely satisfactory for gathering statistical information from a large data base when exact results are not required.

In a transaction with Level 2 consistency, the user is assured that every item read is "clean," i.e., that the transaction that established the value has ended and is therefore not subject to backout. However, no guarantee is made that subsequent access to the same item will yield the same values. At this consistency level, it is possible for another transaction to modify a data item any time after the given Level 2 transaction has read it. A second read by the given transaction will then yield the new value, since the item will become clean again when the other transaction terminates.

For the highest consistency level, called Level 3, the user sees the logical equivalent of a single-user system. No user running Level 3 can tell that the other users are concurrently accessing and modifying the data base. Every item read is clean, and subsequent reads yield the same values, subject, of course, to updates by the given user. This repeatability feature applies not only to a specific item accessed directly by record identifier but even to sequences of items and to items accessed associatively. For example, if the RDS employs an index on the Employee table, ordered by Employee Name, to find all employees whose names start with "B," then the same set of names will be returned if the access is repeated later in the same transaction. Thus, the RDS can effectively lock a set of items defined by an SQL predicate and obtained by any search strategy against insertions into or deletion from the set. Similarly, if the RDS employs an index to access the record where Name = "Smith," and no such record exists, then the same nonexistence result is ensured for subsequent accesses within the same transaction.

Level 3 consistency eliminates the problem of lost updates and also guarantees that one can read a logically consistent version of any collection of records, since other transactions are logically serialized with the given one. As an example of this last point, consider a situation where two or more related data items are updated together, such as the source and target of a funds transfer. With Level 3 consistency, a reader is assured of reading a consistent pair, rather than, say, an old balance of one and a new balance of the other.

It has been a surprise to us that the Level 3 consistency lock protocol is no more expensive than the Level 2 protocol. In sev-

eral cases Level 3 is cheaper than Level 2. For that reason most users elect Level 3 consistency (the default).

The RSS components set locks automatically to guarantee the logical functions of these various consistency levels. For example, in certain cases the RSS must set locks on records, such as when they have been inserted or updated. Similarly, in certain cases the RSS must set locks on index values or ranges of index values, even when the values are not currently present in the index, such as to handle the case of "Smith" described above. In both of these cases the RSS must also acquire physical locks on one or more pages, which are held at least during the execution of each RSI operation, in order to ensure that data and index pages are accessed and maintained correctly.

Data items can be locked at various granularities to ensure that various applications run efficiently. For example, locks on single records are effective for transactions that access small amounts of data, whereas locks on entire segments are more reasonable for transactions that cause the RDS to access large amounts of data. In order to accommodate these differences, a dynamic lock hierarchy protocol has been developed so that a small number of locks can be used to lock both few and many objects.

Since locks are requested dynamically, it is possible for two or more concurrent activations of the RSS to deadlock. The RSS has been designed to check for deadlock situations when requests are blocked and to select one or more victims for backout if deadlock is detected. Each time a transaction waits, a matrix of who is waiting for whom is examined, and deadlock cycles (if any) are detected. The selection of a victim is based on the relative ages of transactions in each deadlock cycle. In general, the RSS selects the youngest transaction as the victim. This transaction is then backed out. Reference 9 has a more complete discussion of the System R lock manager.

Summary and conclusions

We have described the architecture of System R, including the Relational Data System and the Research Storage System. The RDS supports a flexible spectrum of binding times, ranging from precompilation of "canned transactions" to on-line execution of ad hoc queries. The advantages of this approach may be summarized as follows:

1. For repetitive transactions, all the work of parsing, name binding, and access path selection is done once at precompilation time and need not be repeated.

2. Ad hoc queries are compiled on line into small machine-language routines that execute more efficiently than an interpreter.
3. Users are given a single language, SQL, for use in ad hoc queries as well as in writing PL/I and COBOL transaction programs.
4. The SQL parser, access path selection routines, and machine language code generator are used in common between query processing and precompilation of transaction programs.
5. When an index used by a transaction program is dropped, a new access path is automatically selected for the transaction without user intervention.

The RSS is a low-level data base management system that provides multiple paths for accessing data including sequential scans, indexes, sorting, and pointer chains, and, in addition, provides service for locking, recovery, and transaction management. The locking facility, for example, allows some users to be running transaction programs, others to be precompiling new programs, and others to be running ad hoc queries and updates, all on the same data base at the same time with predictable consequences.

CITED REFERENCES

1. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: A relational approach to data base management," *ACM Transactions on Database Systems* **1**, No. 2, 97–137 (June 1976).
2. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* **20**, No. 6, 560–575 (November 1976).
3. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," 23–24, *Proceedings of the 1979 SIGMOD Conference*.
4. R. A. Lorie and J. F. Nilsson, "An access specification language for a relational data base system," *IBM Journal of Research and Development* **23**, No. 3, 286–298 (May 1979).
5. R. A. Lorie and B. W. Wade, *The Compilation of a Very High-Level Language*, Research Report RJ2008, IBM Corporation, San Jose, CA (May 1977).
6. D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost, *Support for Repetitive Transactions and Ad Hoc Query in System R*, Research Report RJ2551, IBM Corporation, San Jose, CA (May 1979).
7. R. A. Lorie, "Physical integrity in a large segmented database," *ACM Transactions on Database Systems* **2**, No. 1, 91–104 (March 1977).
8. M. W. Blasgen, K. P. Eswaran, and R. G. Casey, "An encoding method for multifield sorting and indexing," *Communications of the ACM* **20**, No. 11, 874–878 (November 1977).
9. J. N. Gray, R. A. Lorie, G. F. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," *Modeling in Data Base Management Systems*, G. M. Nijssen, editor, North Holland, pp. 365–394 (1976). (Also as IBM Research Report RJ1606.)

BIBLIOGRAPHY

M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, P. Tiberio, I. L. Traiger, B. W. Wade, and R. A. Yost, "System R, A relational database management system," *IEEE Computer* **13**, No. 5, 43-48 (May 1979).

M. M. Astrahan et al., *A History and Evaluation of System R*, Research Report RJ2843, IBM Corporation, San Jose, CA (June 1980).

M. W. Blasgen and K. P. Eswaran, "Storage and access in relational data bases," *IBM Systems Journal* **16**, No. 4, 363-377 (1977).

M. W. Blasgen, J. N. Gray, M. Mitoma, and T. G. Price, "The convoy phenomenon," *ACM Operating Systems Review* **13**, No. 2, 20-25 (April 1979).

D. D. Chamberlin, "Relational data base management systems," *Computing Surveys* **8**, No. 1, 43-66 (March 1976).

D. D. Chamberlin, *A Summary of User Experience with the SQL Data Sublanguage*, Research Report RJ2737, IBM Corporation, San Jose, CA (April 1980).

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "On the notions of consistency and predicate locks in a relational database system," *Communications of the ACM* **19**, No. 11, 624-634 (November 1976).

J. N. Gray, "Notes on data base operating systems," in *Operating Systems—An Advanced Course*, R. Bayer, R. M. Graham, G. Seegmuller, editors, Springer-Verlag, New York, 393-481 (1978).

J. N. Gray, P. R. McJones, M. W. Blasgen, R. A. Lorie, T. G. Price, G. R. Putzolu, and I. L. Traiger, *The Recovery Manager of a Data Management System*, Research Report RJ2623, IBM Corporation, San Jose, CA (August 1979).

J. N. Gray and V. Watson, *A Shared Segment and Interprocess Communication Facility for VM/370*, Research Report RJ1579, IBM Corporation, San Jose, CA (May 1975).

P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Transactions on Database Systems* **1**, No. 3, 242-255 (September 1976).

H. R. Strong, I. L. Traiger, and G. Markowsky, *Slide Search*, Research Report RJ2274, IBM Corporation, San Jose, CA (June 1978).

M. W. Blasgen is with IBM at 10215 Fernwood Road, Bethesda, MD 20034; J. N. Gray is with TANDEM in Cupertino, CA; W. F. King is with IBM Corporate Headquarters, Old Orchard Road, Armonk, NY 10504; the other authors are located at the IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.

Reprint Order No. G321-5140.