Check for updates

Support for Repetitive Transactions and Ad Hoc Queries in System R

D. D. CHAMBERLIN, M. M. ASTRAHAN, W. F. KING,
R. A. LORIE, J. W. MEHL, T. G. PRICE,
M. SCHKOLNICK, P. GRIFFITHS SELINGER,
D. R. SLUTZ, B. W. WADE, and R. A. YOST
IBM Research Laboratory

System R supports a high-level relational user language called SQL which may be used by ad hoc users at terminals or as an embedded data sublanguage in PL/I or COBOL. Host-language programs with embedded SQL statements are processed by the System R precompiler which replaces the SQL statements by calls to a machine-language access module. The precompilation approach removes much of the work of parsing, name binding, and access path selection from the path of a running program, enabling highly efficient support for repetitive transactions. Ad hoc queries are processed by a similar approach of name binding and access path selection which takes place on-line when the query is specified. By providing a flexible spectrum of binding times, System R permits transaction-oriented programs and ad hoc query users to share a database without loss of efficiency.

System R is an experimental database management system designed and built by members of the IBM San Jose Research Laboratory as part of a research program on the relational model of data. This paper describes the architecture of System R, and gives some preliminary measurements of system performance in both the ad hoc query and the "canned program" environments.

Key Words and Phrases: relational database systems, compilation, performance measurements, transaction processing, query languages
CR Categories: 3.70, 4.12, 4.33, 4.6

INTRODUCTION

System R is an experimental database 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 [1], and SQL, its user interface, was described in [3]. Since these publications, System R has undergone certain architectural changes, and implementation of the prototype system is now essentially complete. The purpose of this paper is to bring up to date the previously published description of system architecture and to present some preliminary measurements of the performance of the prototype.

One of the basic goals of System R is to support two different types of processing against a database: (1) ad hoc queries and updates, which are usually

Authors' address: IBM Research Laboratory, K54/282, 5600 Cottle Road, San Jose, CA 95193.

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 [3] available in both these environments. These features include statements to query and update a database, to define and delete database objects such as tables, views, and indexes, and to control access to the database by various users.

An ad hoc user at a terminal may type SQL statements and view the result directly at the terminal as in the following examples:

```
SELECT NAME, SALARY FROM EMP WHERE JOB = 'PROGRAMMER';

UPDATE EMP SET SALARY = 9500 WHERE EMPNO = 501;
```

The same SQL statements may be embedded in a PL/I or COBOL program by prefixing them with $ 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 $ signs, as in the following example:

```
$UPDATE EMP SET SALARY = $X WHERE EMPNO = $Y;
```

Host-language variables in a SQL statement may be used in place of data values but not in place of table names or field names.

If a PL/I or COBOL program wishes to execute a SQL query and fetch the result, it does so by means of a "cursor." A cursor is defined by a LET statement, which associates the cursor name with a particular query. The cursor is readied for retrieval by an OPEN statement, which binds the values of any host-language variables appearing in search conditions 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 C1 BE
  SELECT NAME, SALARY INTO $X, $Y
  FROM EMP WHERE JOB = $Z;

$OPEN C1;  /* BINDS VALUE OF Z*/

$FETCH C1;  /* FETCHES ONE EMPLOYEE INTO X AND Y */

$CLOSE C1;  /* 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 SYR_CODE.

System R is based on a special multiuser access method called the Research Storage System (RSS), with facilities for locking, logging, recovery, and index maintenance. The description of the RSS is essentially unchanged since [1].

However, the Relational Data System (RDS) which runs on top of the access method is now 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-time 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 embedded 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,

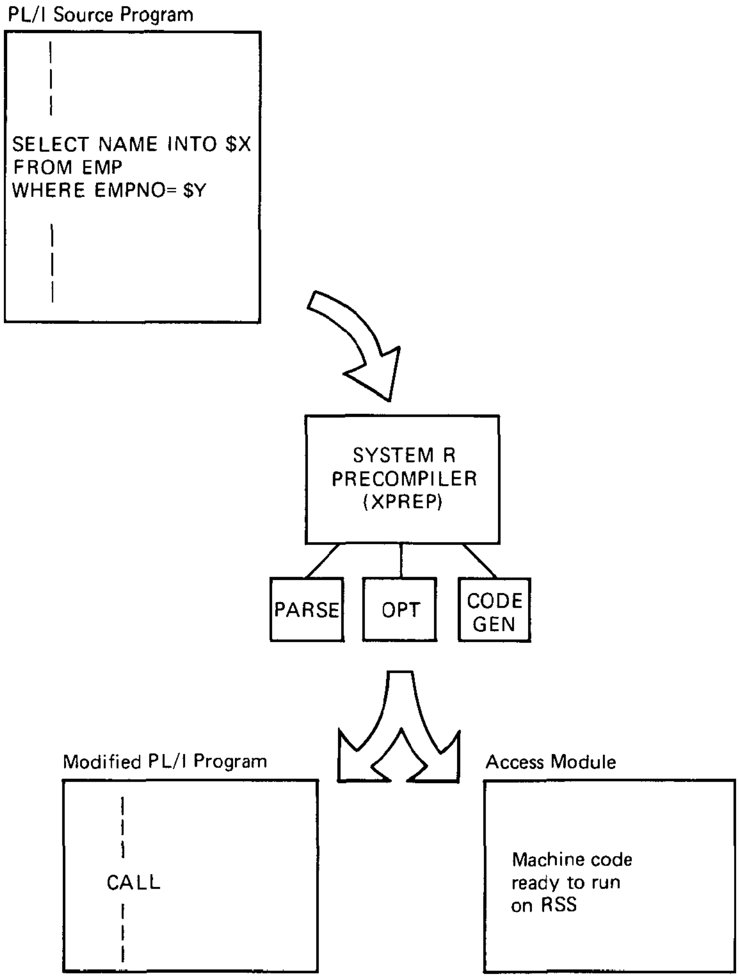

Fig. 1. Precompilation step.

the SQL statements are replaced by host-language calls to the access module. The access module is stored in the System R database to protect it from unauthorized modification. The precompilation step is illustrated in Figure 1.

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

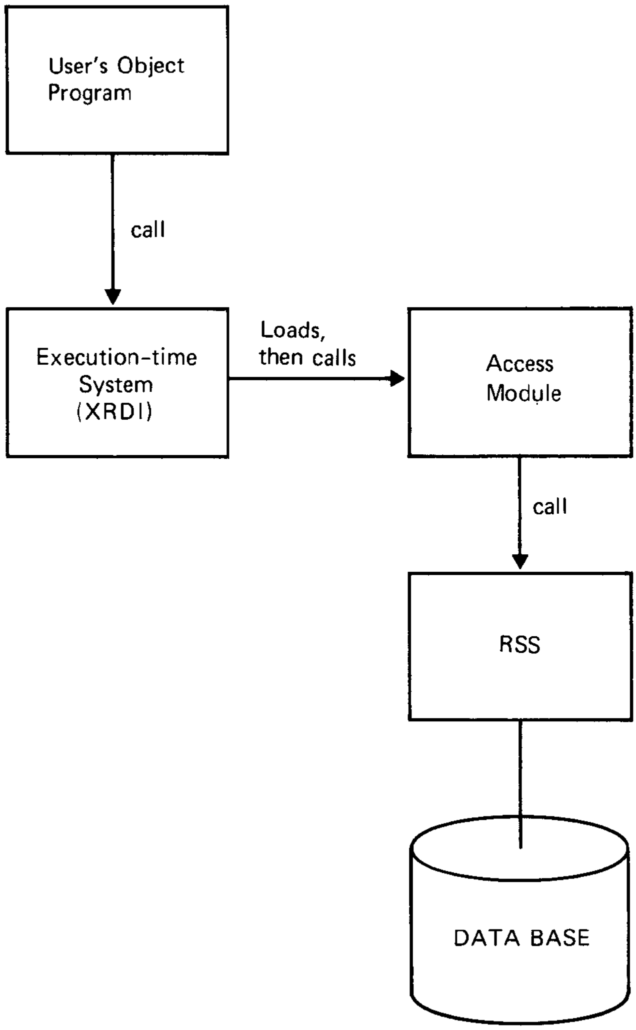

Fig. 2. Execution step.

in turn loads and invokes the access module for the program. The access module operates on the database by making calls to the RSS and delivers the result to the user's program. This process is illustrated in Figure 2.

The ad hoc user of System R is supported by a special program called the User-Friendly Interface (UFI), which controls dialogue 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 UFI's purpose is to execute SQL statements which are not known in advance. When a user enters an ad hoc SQL statement, the 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 UFI's access module to be dynamically generated for the new statement. The dynamically generated section of the access

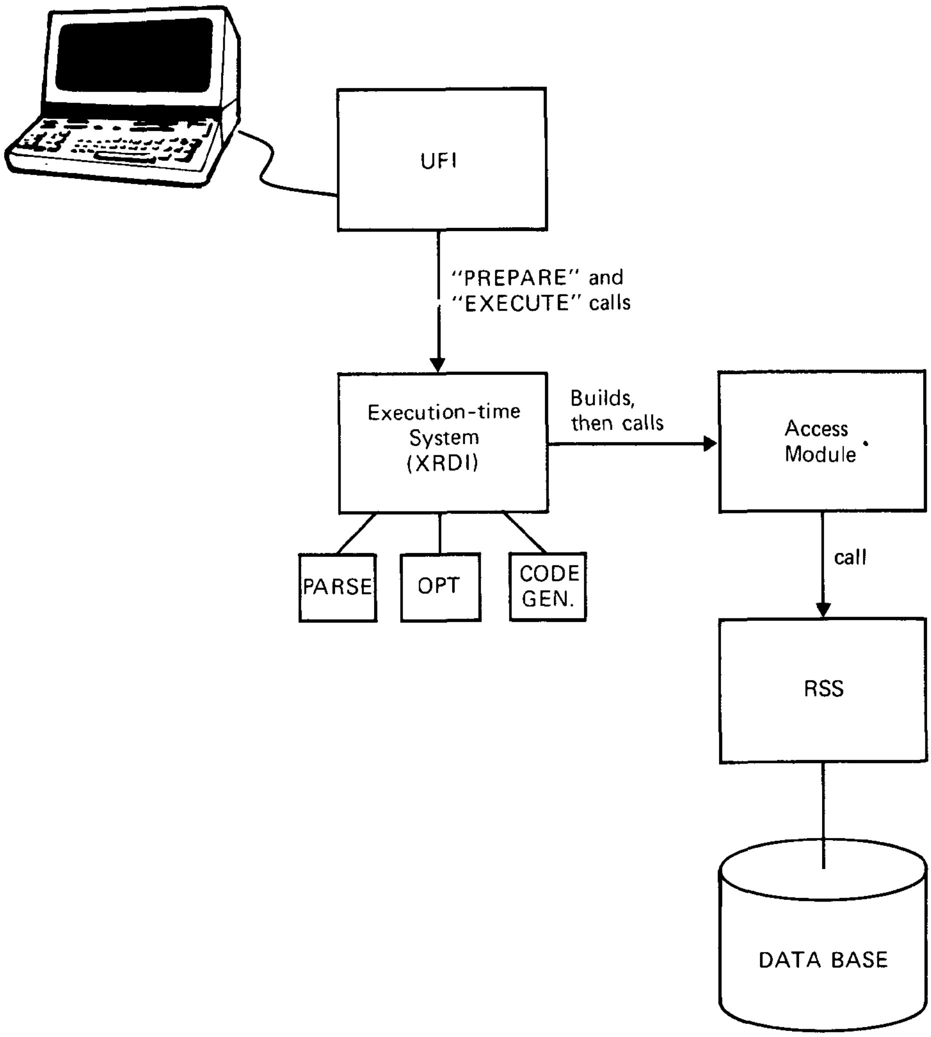

Fig. 3. Processing of ad hoc queries.

module contains machine-language code and is in every way indistinguishable from the sections which were generated by the precompiler. The interactions of the UFI with System R are illustrated in Figure 3.

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 the UFI, querying and updating the database and creating new tables and views. All these simultaneous activities are supported by the automatic locking subsystem built into the RSS described in [4].

First we examine in detail the two major functions of System R: precompilation and execution of a "canned program." Next, we examine how System R implements the special PREPARE and EXECUTE calls which are needed to support ad hoc users. Finally, we present a sample database and some measurements of the performance of System R in both a canned program and an ad hoc query environment.

PRECOMPILATION

When a PL/I or COBOL program with embedded SQL statements is presented to the System R precompiler, it scans the program to find the SQL statements (they are indicated by $) and replaces each SQL statement by a valid host-language CALL. In addition, each SQL statement is put through a three-step process in order to translate it to 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 EMP WHERE DEPT = $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 database objects.
    (b) A check is made 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 which 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 details of this access path selection process are given in [10]. The optimizer represents its chosen access path by structural modifications to the parse tree called Access Specification Language (ASL) [5], and by constructing the RSS control blocks to be used in processing the statement.

(3) *Code generation*: The code generator translates the ASL structures produced by the optimizer into a 370 machine-language routine which implements the chosen access path [6]. This machine-language routine is called a "section." When running, the section will access the database by using the RSS control blocks which were produced by the optimizer.

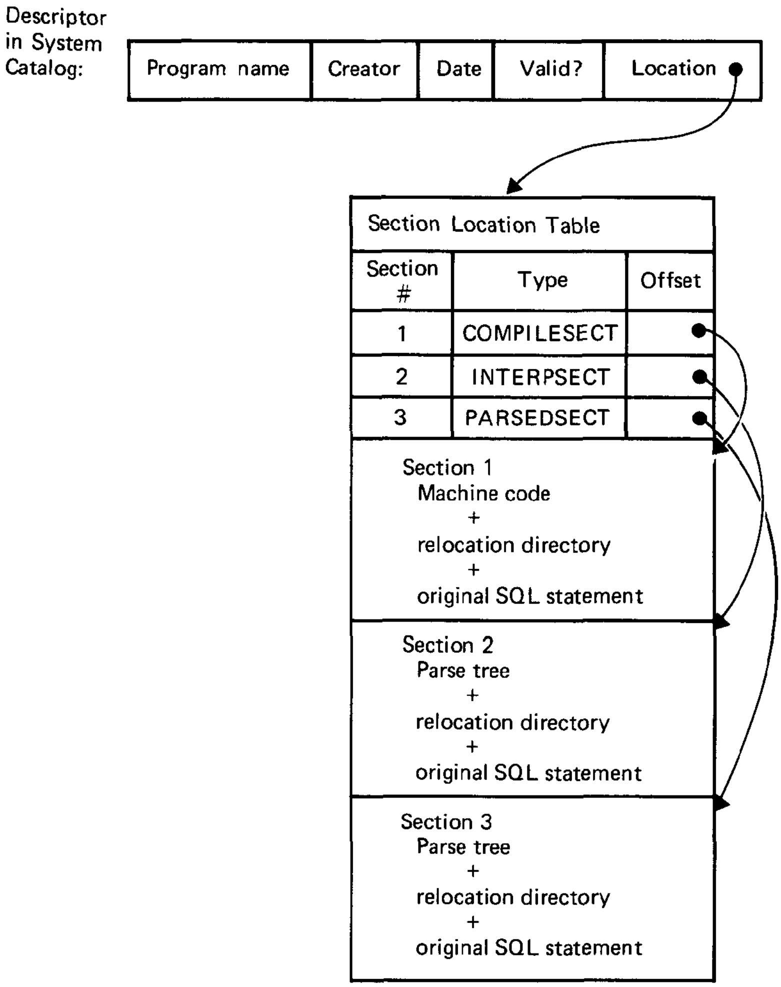

Fig. 4. Structure of an access module.

After all the SQL statements in a program have been translated into sections, the sections are collected together to form an access module. In the header of the access module is placed a Section Location Table which lists the relative byte offset of each section within the Access Module. Each section has a Relocation Directory which lists the offsets within the section of all internal pointers which must be relocated before the section can be used. In addition to machine-language code, each section holds the SQL statement from which it was originally constructed. This enables the section to be rebuilt if its original access path should become unavailable at some future time. The rebuilding process is similar to precompilation and is described later. The structure of an access module is shown in Figure 4. (Some of the entries in the access module, e.g., INTERPSECT, will be explained later in this paper.) When the access module is complete, it is stored in the System R database for later use. If the user who precompiled the program passes the authorization test for all SQL statements in the program, he receives the "RUN" privilege for the access module.

When the precompiler translates a SQL statement into a section, it must also replace the SQL statement in the user's PL/I or COBOL program by a CALL. The call invokes XRDI, the System R entry point used for executing a stored access module. The parameters of the call are the name of the access module, the section number within the access module, an operation code, and the addresses of the input and/or output variables to be used in processing the statement.

If the SQL statement under consideration is not an operation on a cursor, the construction of the call is straightforward. The operation code is AUXCALL, meaning simply, "execute the section." All host-program variables in the original SQL statements are passed in with the AUXCALL, as shown for the UPDATE statement in Figure 5.

If the SQL statement is an operation on a cursor, the situation is slightly more complex. The cursor is defined by an SQL statement of the form:

LET ⟨cursor name⟩ BE ⟨query⟩.

The basic operations on cursors are OPEN ⟨cursor name⟩, FETCH ⟨cursor name⟩, and CLOSE ⟨cursor name⟩. The LET statement does not result in a CALL since it is definitional in nature. In response to the LET statement, the System R precompiler produces a section for the indicated cursor containing machine code for opening, fetching, and closing the cursor. Then, in response to OPEN, FETCH, and CLOSE statements the precompiler generates CALLS on the appropriate section with the appropriate operation codes, as shown in Figure 5. The addresses of input variables are passed as parameters of the OPEN call since input values are always bound when a cursor is opened. Addresses of output variables are passed as parameters of the FETCH call, giving the target locations for the data to be fetched. No variables are involved in the CLOSE call.

After the System R precompiler has replaced all the SQL statements in the user's program by 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.

EXECUTING A PRECOMPILED PROGRAM

When a user invokes a program which 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 database into virtual memory, its internal pointers are adjusted using the relocation directory of each section, and then control is passed to the indicated section. On subsequent calls to the same access module, the authorization check, loading, and relocation steps are bypassed, and control passes directly to the indicated section. The machine-language code in the section examines the operation code of the call (e.g., OPENCALL or FETCHCALL) and proceeds to process the original SQL statement from which it was compiled, using as needed the host-program variables which were passed in with the call. When running under MVS, the access module is assigned a different storage protection key from the user's program in order to provide them with mutual protection.

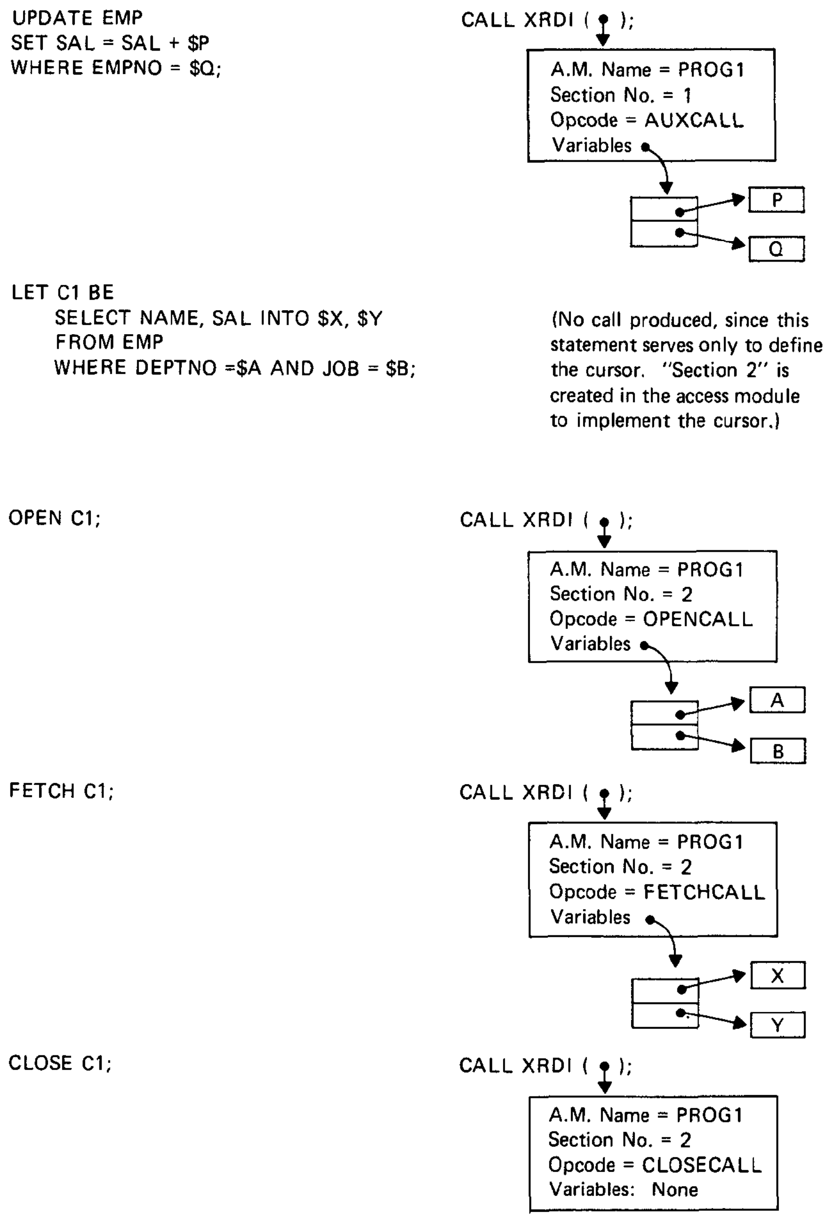

Fig. 5. Replacement of SQL statements by calls.

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 which are affected by the change. If the change involves dropping a table or revoking a necessary privilege, the access module is erased from the database. 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 database 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 may attempt to change the database in some way which would invalidate an access module while the access module is actually loaded and running. It would be undesirable if such a change were allowed to become effective while the running access module is in the middle of some operation. To prevent this from occurring, the "transaction" mechanism of System R is used. A programmer can declare transaction boundaries in his program by the BEGIN TRANSACTION and END TRANSACTION statements of SQL. Users are advised to end a transaction only when the database is in a "clean" and consistent state; that is, when one user-defined unit of work has completed and the next unit of work has not yet begun. While a transaction is in progress, the loaded access module protects itself by holding a lock on its own description in the system catalog tables. Therefore, any database change which would invalidate the access module (changing its description from "valid" to "invalid") must wait until the lock is released. At the end of each transaction, the running access module releases the lock on its own description, allowing any database changes which were waiting for the lock to proceed. At the beginning of the next transaction, the access module attempts to reacquire the lock on its own description. There are four possible outcomes.

(1) The description is still marked "valid," and the timestamp in the description is unchanged. In this case, execution of the access module proceeds normally.
(2) The description is gone. The access module has been destroyed by loss of an essential table or privilege. An appropriate code is returned to the user's program, which is unable to continue.
(3) The description is present but marked "invalid." This indicates that an index used by the access module has been dropped. The access module is regenerated on the spot, choosing a new access path to replace the missing index. The user program then continues without interruption.
(4) The description is marked "valid," but its timestamp has changed (indicating another user has caused a regeneration). The new (regenerated) access module is loaded into virtual memory and the user program continues.

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 fields and their data types. This information is conveyed in the form of the SQL parse tree for the CREATE TABLE statement. We refer to SQL statements which 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 field names) from the parse tree in the INTERPSECT.

OPERATIONS ON TEMPORARY TABLES

Occasionally a user may write a program which creates a temporary table in the database, processes the table in some way, then destroys the table at the end of the run. When such a program is precompiled, the System R optimizer is unable to choose an access path for processing the temporary table because it does not yet exist. Whenever the optimizer discovers during precompilation that some table referenced in an SQL statement does not exist, it places the parse tree for the SQL statement in a special section and labels it a "PARSEDSECT." This indicates that the normal process of parsing, optimization, and code generation was terminated after the parsing step.

At run time, when XRDI receives a call to execute the PARSEDSECT, it cannot give control directly to the section because it does not yet contain machine code. Instead, XRDI makes another attempt to invoke the optimizer on the parse tree in the PARSEDSECT. This time, since the temporary table is about to be operated on, it should be in existence. If optimization is successful, the code generator is invoked, a machine language routine is generated, and the PARSEDSECT changes into a COMPILESECT, which is immediately executed. However, if optimization fails because the indicated table still does not exist, a code is returned to the calling program indicating "nonexistent table."

The transformation of a PARSEDSECT into a COMPILESECT affects only the version of the access module which is held in virtual memory, not the version which is stored in the database.

DYNAMICALLY DEFINED STATEMENTS

Some programs may need to execute SQL statements which 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, which loads data into tables via SQL INSERT statements, but which 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 which supports this type of application is the PREPARE statement, which has the following 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 which 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, for example,

UPDATE EMP SET SALARY = ? WHERE EMPNO = ?

When the precompiler encounters a PREPARE statement in a program, it creates a special zero-length section in the access module called an INDEFSECT. In the user's program the PREPARE statement is replaced by a special call to XRDI with operation code = SETUPCALL, containing a pointer to the variable QSTRING.

At run time XRDI interprets the SETUPCALL as an instruction to accept a dynamically defined SQL statement and to pass it through the parser, optimizer, and code generator. The result is a new COMPILESECT or INTERPSECT, which replaces the INDEFSECT in the access module. (However, the INDEFSECT is replaced only in the virtual-memory copy of the access module, not in the copy which remains in the database.) 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 this EXECUTE statement into a normal AUXCALL on the indicated section, passing the addresses of $X and $Y as parameters of the call. The parameters passed at EXECUTE time are bound, in positional order, to the question marks in the SQL statement S1. (Note: These parameters may represent data values but not table names or column names.) The statement S1 may be executed many times, with different parameters, without reinvoking the optimizer. However, if the PREPARE S1 AS QSTRING statement is executed again, the contents of the section are discarded and a new COMPILESECT or INTERPSECT is constructed on the basis of the new contents of QSTRING.

If the prepared statement is a query, the COMPILESECT produced for it will look exactly like a COMPILESECT produced for a cursor. In other words, the two statements

LET C1 BE ⟨query⟩

and

PREPARE S1 AS QSTRING

will produce exactly the same section if the contents of QSTRING are the same as ⟨query⟩. Therefore, the operations on a "prepared" query are the same as the operations on a cursor: OPEN, FETCH, and CLOSE. Input variables may be included in an OPEN statement and the target variables listed in the FETCH statement, as in the following examples.

OPEN S1 USING $A, $B;	(Precompiler produces OPENCALL with addresses of $A and $B as parameters.)
FETCH S1 INTO $X, $Y;	(Precompiler produces FETCHCALL with addresses of $X and $Y as parameters.)
CLOSE S1;	(Precompiler produces CLOSECALL.)

In addtion to OPEN, FETCH, and CLOSE, System R supports another operation called DESCRIBE on sections which contain a query. The syntax of a DESCRIBE statement is as follows:

DESCRIBE ⟨statement name⟩ INTO ⟨array⟩

The System R precompiler translates the DESCRIBE statement into a special DESCRIBECALL on the section corresponding to the indicated statement name. At run time, when XRDI receives the DESCRIBECALL, it returns into the indicated array a description of the field names and data types in the query result. The calling program can then use this information in formatting the query result for display at a terminal. A DESCRIBECALL on a section which does not contain a query returns a code indicating "no result."

SUMMARY OF SECTION TYPES AND CALL TYPES

The four basic steps in the processing of an SQL statement are parsing, optimization, code generation, and execution. The basic philosophy of System R is to perform as many of these steps as possible during precompilation, then to perform the remaining steps at run time. Depending on the nature of the statement, the break between precompile time and run time may occur at several different places in the processing of the statement, as shown in Figure 6. The mechanism which implements the early-binding philosophy of System R consists of four section types and six call types. The behavior of XRDI for each combination of section type and call type is summarized in Figure 7.

PERFORMANCE MEASUREMENT

In order to illustrate how System R might be used in an environment where ad hoc queries are mixed with repetitive transactions, the example database in Figure 8 was constructed. The PARTS table contains the description, quantity on hand, and quantity on order for a collection of parts identified by part

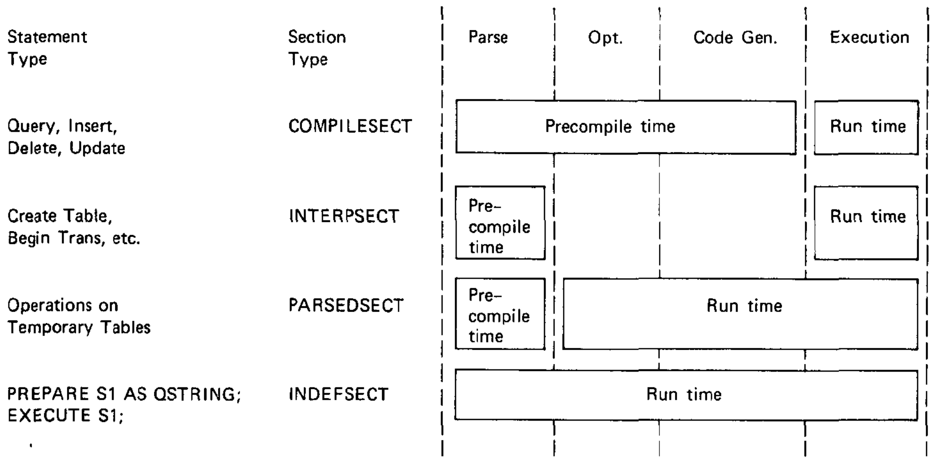

Fig. 6. Spectrum of binding times in System R.

numbers. The ORDERS table contains a set of outstanding orders for parts. The QUOTES table contains a set of price quotes for parts. Each price quote is identified by a particular supplier number and part number and the minimum and maximum quantities for which the quote applies. Typically, a given combination of supplier and part numbers may have several quotes: one for quantities from 1 to 100, another for quantities from 101 to 10,000, etc.

The following structural and statistical information completes our description of the example database.

(1) The total size of database (including data records but not indexes) equals 7.44 megabytes.
(2) The data values in the sample database were randomly generated according to the following rules:
   (a) The number of different part descriptions equals 1024.
   (b) The number of different supplier numbers equals 1000.
   (c) Each part number has exactly three outstanding orders and three price quotes from each of three different suppliers.
(3) Clustering method: The three tables are stored on disk in an interleaved fashion, ordered by PARTNO. Each PARTS record is followed by all the ORDERS and QUOTES for that part number, then by the next PARTS record, etc. Fifteen percent free space is preserved on each data page to allow for future insertions, which will also be clustered by PARTNO.

A two-part experiment was performed on the example database. The first part involved measurement of three example queries submitted via the User-Friendly Interface (UFI) of System R. For each query, the CPU time and number of I/Os were measured for each step in processing the query: parsing, optimization, code generation, and fetching of the answer set.

The second part of the experiment involved writing a PL/I program to process three types of "canned transactions" against the sample database. This program

<table>
<tr><th></th><th>COMPILESECT</th><th>INTERPSECT</th><th>PARSEDSECT</th><th>INDEFSECT</th></tr>
<tr><td>AUXCALL</td><td>Execute the machine code in the section.</td><td>Execute a standard routine controlled by the content of the section</td><td rowspan="2">Invoke the optimizer and code generator to convert the section into a COMPILESECT or INTERPSECT; then execute it.</td><td>(Not used)</td></tr>
<tr><td>OPENCALL</td><td>Execute the machine code in the section, with opcode =OPEN.</td><td>(Not used)</td><td>(Not used)</td></tr>
<tr><td>FETCHCALL</td><td>Execute the machine code in the section, with opcode =FETCH.</td><td>(Not used)</td><td>(Not used)</td><td>(Not used)</td></tr>
<tr><td>CLOSECALL</td><td>Execute the machine code in the section, with opcode =CLOSE.</td><td>(Not used)</td><td>(Not used)</td><td>(Not used)</td></tr>
<tr><td>SETUPCALL</td><td colspan="2">Throw away the current content of the section, and invoke the parser, optimizer, and code generator to build a new COMPILESECT or INTERPSECT from a new SQL statement.</td><td>(Not used)</td><td>Invoke the parser, optimizer, and code generator to convert a new SQL statement into a COMPILESECT or INTERPSECT.</td></tr>
<tr><td>DESCRIBE-CALL</td><td>Return a description of the answer set.</td><td>(Not used)</td><td>(Not used)</td><td>(Not used)</td></tr>
</table>

Fig. 7. Section types and call types.

accesses and updates the database by means of embedded SQL statements, as described in [3]. Measurements of CPU time and number of I/Os were made during precompilation and compilation of the transaction program, and for execution of each of the three transaction types. In addition, measurements were made of how the execution times for the three transactions are affected by

PARTS	PARTNO	DESCRIP	QOH	QOO
	CHAR(6)	CHAR(50)VAR	INTEGER	INTEGER

ORDERS	ORDERNO	PARTNO	SUPPNO	DATE	QTY
	CHAR(6)	CHAR(6)	CHAR(3)	CHAR(6)	INTEGER

QUOTES	SUPPNO	PARTNO	MINQ	MAXQ	PRICE
	CHAR(3)	CHAR(6)	INTEGER	INTEGER	INTEGER

Table	No. of Records	Avg. length of record (bytes, incl. header)	Indexes Maintained
PARTS	20,000	36	PARTNO DESCRIP
ORDERS	60,000	31	ORDERNO PARTNO SUPPNO
QUOTES	180,000	27	SUPPNO PARTNO

Fig. 8. Sample database.

database size, using a series of five databases structurally identical to the one described above, but with different numbers of records.

All experiments were performed on an IBM 370 Model 168 under the VM/CMS operating system. Measurements were made using the timing facilities of VM, which report the cumulative virtual CPU time and number of I/Os performed by the user's virtual machine. These measurements do not include the "overhead" costs of VM in providing the user with a virtual machine (e.g., dispatching, virtual memory paging, channel program translation, etc.). Under VM, "overhead" includes about 2500 instructions per I/O. These costs are not included in our experiment because they are highly dependent on the operating system or data communication subsystem (e.g., VM/CMS, MVS/TSO, CICS) under which the application is executing.

All measurements were made with System R running in multiuser mode (locking subsystem enabled). Experience has shown that System R uses the same number of I/Os to perform equivalent functions in single-user and multiuser modes, but uses slightly less CPU time in single-user mode due to the disabling of the locking subsystem.

Table I. Query 1

English form:
Find the supplier number and price of all quotes for part number 010002 in quantities of 1000.

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

Access path chosen by optimizer:
PARTNO index on QUOTES table.

Cardinality of answer set: 3

Operation	CPU time (ms on 168)	Number of I/Os
Parsing	13.3	0
Optimization	40.0	9
Code generation	10.1	0
Open cursor	3.7	5
Fetch answer set (1.5 ms per answer record)	4.6	2
Close cursor	1.2	
Total (including all of above plus formatting answer set for display)	83.4	16

QUERY MEASUREMENTS

The three experimental queries, and their measured performance, are summarized in Tables I-III. In order to make these measurements reproducible, it was necessary to ensure that no query could benefit from data which remained in the system buffers from previous activities. (System R has a buffer space in virtual memory containing data recently fetched from the database. The buffer space is adjustable in size; in our experiment it was configured at twenty 4096-byte pages.) Therefore, before each query was measured, another query was run, which fetched at least 50 pages of data, none of which pertained to the query to be measured.

A striking fact about the measurements shown in the tables is the small amount of time involved in the "code generation" step. This is the processing step which could be avoided in a system which interprets a query immediately after choice of access path. We see that the cost of translating the chosen access path into executable code is quite small (zero I/Os and 19-42 percent of the CPU time required to parse and optimize the query). The payoff for this small investment is a small, efficient machine-language routine to process the query. This generated routine has a shorter path length and smaller working set than a general-purpose SQL interpreter because it is tailored to a specific query. Therefore it quickly pays off the cost of code generation as it fetches answer records from the database. Thus we see that the SQL compilation approach of System R has significant benefits for ad hoc query, as well as for a "canned transaction" environment.

For all the ad hoc queries the optimizer chose access paths using indexes. To illustrate the importance of the indexes, the three queries were rerun with no indexes on any of the tables. The CPU and I/O measurements for parsing, optimization, and code generation were essentially unchanged. For Queries 1, 2,

Table II. Query 2

English form:
Find the order number, part number, description, date and quantity for all parts ordered from supplier number 797 during 1975.
SQL form:
SELECT ORDERNO, ORDERS.PARTNO, DESCRIP, DATE, QTY
FROM ORDERS, PARTS
WHERE ORDERS.PARTNO = PARTS.PARTNO
AND DATE BETWEEN '750000' AND '751231'
AND SUPPNO = '797';
Access path chosen by optimizer:
Access ORDERS by SUPPNO index. For each qualifying ORDERS record, access corresponding PARTS record by index on PARTNO.
Cardinality of answer set: 7

Operation	CPU time (ms on 168)	Number of I/Os
Parsing	20.7	0
Optimization	73.2	9
Code generation	19.3	0
Open cursor	4.0	6
Fetch answer set (8.7 ms per answer record)	61.1	75
Close cursor	5.3	
Total (including all of above plus formatting answer set for display)	213.9	90

Table III. Query 3

English form:
For each supplier that supplies part number 010007, list the minimum and maximum quoted price for that part number.
SQL form:
SELECT SUPPNO, MIN(PRICE), MAX(PRICE)
FROM QUOTES WHERE PARTNO = '010007'
GROUP BY SUPPNO;
Access path chosen by optimizer:
Scan QUOTES by PARTNO index to get all quotes for part number 010007. Then sort these into SUPPNO order and scan sorted list to compute minima and maxima.
Cardinality of answer set: 3

Operation	CPU time (ms on 168)	Number of I/Os
Parsing	13.0	0
Optimization	40.3	9
Code generation	22.6	0
Open cursor (includes finding relevant quotes and sorting the list)	11.2	9
Fetch answer set (includes scanning sorted list for minima and maxima) (1.7 ms per answer record)	5.1	0
Close cursor	0.3	
Total (including all of above plus formatting answer set for display)	108.2	18

and 3 the total CPU times (measured in seconds) were 13.29, 30.64, and 9.11 and the numbers of I/Os were 3201, 26070, and 3205, respectively. The significantly higher values result, of course, from scanning all the tuples of each table involved in the query.

TRANSACTION MEASUREMENTS

The second part of our experiment involved preparing a PL/I program with embedded SQL statements to implement three types of "canned transactions" against the sample database. The program, named ORDERS, is included in the appendix. The ORDERS program differs from traditional database transaction programs in that its terminal interactions are handled by PL/I I/O rather than by a data communication subsystem. The program reads a transaction-type code from a terminal and then performs one of the following types of transactions.

Transaction Type N. (New order)	A new order has been placed. Enter the new order in the ORDERS table, and update the QOO field of the appropriate PARTS record.
Transaction Type A. (Arrival)	An existing order of parts has arrived. Access the ORDERS table to find the part number and quantity in the order, and update the appropriate PARTS record accordingly. Then delete the appropriate ORDERS record from the database.
Transaction Type Q. (Query)	Given a part number, look up the description, quantity on hand, and quantity on order of the given part and display them on a terminal.

These three transactions represent simple processes which might be expected to occur repeatedly, and which are therefore included in a precompiled program for maximum efficiency. An actual inventory control application would probably include a much larger collection of these "canned transactions."

The CPU time required for precompilation of the ORDERS program on System R was measured to be 2.22 virtual seconds on a 370/168 (approximately half of the CPU time required for compilation of the same program using the PL/I Optimizing Compiler).

After precompiling the ORDERS program, we examined the resulting access module to determine its size and the access paths which had been selected. The ORDERS program contains nine SQL statements. Therefore, its access module contains nine sections and a Section Location Table (SLT). The size of the access module is summarized in Table IV.

Next, measurements were made of the CPU time and number of I/Os used in executing each of the three transaction types on System R in multiuser mode. When the *first* transaction of a user session is executed (independent of its type), an additional cost is incurred for loading the access module into virtual memory, and this cost was measured separately. For this part of the experiment, we desired to measure the sensitivity of transaction execution times to the size of the database. Therefore, we loaded five databases of different sizes and ran 100 transactions of each type on each database. (Of course, the ORDERS program was separately precompiled in each database.) Each database was structurally

Table IV. Contents of Access Module

Section number	SQL statement	Size of section (bytes)	Access path selected
(SLT)		254	
1	BEGIN TRANSACTION	70	
2	INSERT INTO ORDERS	751	
3	UPDATE PARTS	1321	Index on PARTNO
4	SELECT FROM ORDERS	1319	Index on ORDERNO
5	DELETE ORDERS WHERE CURRENT OF C1	648	Established cursor position
6	UPDATE PARTS	1449	Index on PARTNO
7	SELECT FROM PARTS	1423	Index on PARTNO
8	END TRANSACTION	68	
9	RESTORE TRANSACTION	72	

Note. Total size of access module: 7375 bytes.

Table V. Five Experimental Databases

Database	Number of PARTS records	Number of ORDERS records	Number of QUOTES records	Total size of database in megabytes (not including indexes)
1	5,000	15,000	45,000	1.86
2	20,000	60,000	180,000	7.43
3	40,000	120,000	360,000	14.87
4	70,000	210,000	630,000	26.02
5	100,000	300,000	900,000	37.17

identical to the one described above, the only difference being the total number of records of each type. The sizes of the five databases are summarized in Table V.

For each database, a "script" was created consisting of 300 part numbers randomly selected with a uniform distribution over all the part numbers in the database. Using the script, 100 transactions of each type were executed on the 300 random part numbers (Type N = new order for a given part; Type A = arrival of order for a given part; Type Q = query on a given part). The three types of transactions were mixed together in random order. Since the records in the database are physically clustered by part number, the random sequence of part numbers in the script is uncorrelated with the physical placement of records. This precaution eliminates any spurious effects due to a transaction accessing pages which were left in the system buffers by previous transactions.

For each transaction type, in each database, the average CPU time and average number of I/Os were measured over the 100 executions of the transaction. The CPU time measurement includes time spent in the PL/I portion of the ORDERS program, as well as time spent in System R. However, the I/O counts include only database accesses (not interactions with the terminal). The costs of the three transaction types are summarized in Table VI and in Figure 9. We observe that since all the transactions access the database directly via an index, they are

Table VI. Execution Cost of Transactions

	Database size (megabytes)				
Type	1.86	7.43	14.87	26.02	37.17
A. Average CPU Time (ms *on* 168)					
N	17.0	17.4	18.4	19.7	21.4
A	18.6	18.9	20.2	21.2	23.0
Q	12.3	12.3	12.8	12.1	12.8
B. Average Number of I/Os					
N	10.0	10.5	11.7	16.0	18.3
A	9.5	9.7	11.5	14.7	17.6
Q	3.3	3.8	4.9	5.6	6.0

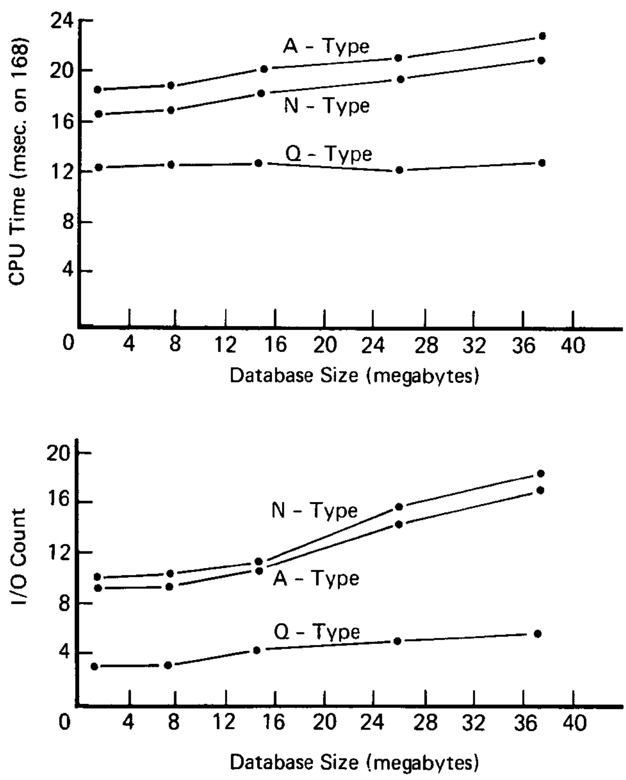

Fig. 9. Execution cost of transactions.

relatively insensitive to the size of the database (a twentyfold increase in database size causes only a 26 percent increase in CPU time and an 85 percent increase in number of I/Os).

In addition to the transaction costs listed in Table VI, two other costs were measured. The cost of loading the access module, which occurs at the first transaction of a session, is 28.1 milliseconds of CPU time and 6 I/Os, independent of the size of the database. In addition, System R automatically takes a "checkpoint" after approximately every 7000 transactions (frequency of checkpoint is an adjustable parameter). Checkpoints involve certain internal system bookkeeping and are not visible to users. The cost of a checkpoint depends on the size of the database and on the activity since the last checkpoint. For our 37.17-megabyte database, the cost of a checkpoint is approximately 117 milliseconds of CPU time and 100 I/Os, representing an average cost per transaction of about 0.02 millisecond and 0.014 I/O.

SUMMARY AND CONCLUSIONS

We have described the architecture of System R, which 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 a small, machine-language access module which executes 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.
(6) The multiuser locking subsystem 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 database at the same time.

We have also described an example database and shown how it might be used both by ad hoc query users and by transaction programs. Some preliminary performance measurements were made on the database using an IBM 370 Model 168 under the VM/370 operating system. The results of our measurements support the following conclusions.

(1) Ad hoc queries, including joins of more than one table, can be parsed, optimized, and executed in substantially less than one virtual second if their answer sets are small and the appropriate indexes are available.
(2) The process of generating machine-language code to execute a query adds a small increment (typically about one-third) to the cost of access path selection for the query.
(3) The access modules resulting from compilation of simple transactions contain about 1000–1500 bytes of code and control blocks per SQL statement.
(4) For simple transactions which are compiled in advance and which are supported by appropriate indexes, System R can process several transactions per second on a 370 Model 168.
(5) When a query or transaction is supported by an index, its performance is relatively insensitive to the size of the database (e.g., in our transaction

experiment, a twentyfold increase in database size caused an average increase of only 26 percent in CPU time and 85 percent in I/O count).

APPENDIX

The following is a pseudocode form of the PL/I program which was used in the performance measurements described in this paper.

```
ORDERS: PROCEDURE;

 /***********************************************************
 *                                                          *
 *    INTERACTIVE PROCESSING OF 3 TRANSACTION TYPES:        *
 *       'N' = NEW ORDER                                    *
 *       'A' = ARRIVAL OF ORDER                             *
 *       'Q' = QUERY SUPPLY OF A GIVEN PART                 *
 *                                                          *
 ***********************************************************/

    (A declaration of the System R return code structure,
     containing SYR_CODE and SYR_MESSAGE, must be copied
     into the program from a macro library.)

    DECLARE   PARTNO        CHARACTER(6);
    DECLARE   DESCRIP       CHARACTER(50) VARYING;
    DECLARE   QOH           BIN FIXED(31);
    DECLARE   QOO           BIN FIXED(31);
    DECLARE   ORDERNO       CHARACTER(6);
    DECLARE   SUPPNO        CHARACTER(3);
    DECLARE   QTY           BIN FIXED(31);
    DECLARE   DATE          CHARACTER(6);
    DECLARE   TRANTYPE      CHARACTER(1);

    GETNEXTTRANS:

    Read TRANTYPE from terminal: N | A | Q or Z to quit;

    IF TRANTYPE = 'N' THEN
       Read ORDERNO, PARTNO, SUPPNO, DATE, QTY from terminal;
    ELSE IF TRANTYPE = 'A' THEN
       Read ORDERNO from terminal;
    ELSE IF TRANTYPE = 'Q' THEN
       Read PARTNO from terminal;
    ELSE IF TRANTYPE='Z' THEN STOP;
    ELSE
    DO;
       Write 'INVALID TRANSACTION TYPE' on terminal;
       GO TO GETNEXTTRANS;
    END;

$BEGIN TRANSACTION;
IF SYR_CODE¬=0 THEN CALL TROUBLE('BEGIN TRANS');

IF TRANTYPE='N' THEN
DO;
     /* NEW ORDER */
     $INSERT INTO ORDERS:<$ORDERNO,$PARTNO,$SUPPNO,$DATE,$QTY>;
     IF SYR_CODE¬=0 THEN CALL TROUBLE('INSERT');
```

```
        $UPDATE PARTS SET QOO=QOO+$QTY WHERE PARTNO=$PARTNO;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('UPDATE');
END;
ELSE IF TRANTYPE='A' THEN
DO;
        /* ARRIVAL */
        $LET C1 BE SELECT PARTNO,QTY INTO $PARTNO,$QTY
                FROM ORDERS WHERE ORDERNO=$ORDERNO;
        $OPEN C1;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('OPEN CURSOR');
        $FETCH C1;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('FETCH');
        $DELETE ORDERS WHERE CURRENT OF C1;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('DELETE');
        $CLOSE C1;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('CLOSE');
        $UPDATE PARTS SET QOH=QOH+$QTY, QOO=QOO-$QTY
                WHERE PARTNO=$PARTNO;
        IF SYR_CODE¬=0 THEN CALL TROUBLE('UPDATE');
END;
ELSE IF TRANTYPE='Q' THEN
DO;
        /* QUERY */
        $SELECT DESCRIP,QOH,QOO INTO $DESCRIP,$QOH,$QOO
                FROM PARTS WHERE PARTNO=$PARTNO;
        IF SYR_CODE = 0 THEN
                Write DESCRIP, QOH, QOO on terminal;
        ELSE IF SYR_CODE = 100 THEN
                Write 'THERE IS NO SUCH PART' on terminal;
        ELSE CALL TROUBLE ('SELECT');
END;

$END TRANSACTION;
IF SYR_CODE¬=0 THEN CALL TROUBLE('END TRANS');

GO TO GETNEXTTRANS;
    TROUBLE: PROCEDURE(STMT);

        DECLARE STMT CHARACTER(12) VARYING;

        Write 'TROUBLE ENCOUNTERED' on terminal;

        Write TRANTYPE, STMT, ORDERNO, PARTNO, SYR_CODE,
                SYR_MESSAGE on terminal;

        $RESTORE TRANSACTION;
        GO TO GETNEXTTRANS;

    END TROUBLE;

END ORDERS;
```

ACKNOWLEDGMENT

The authors of this paper wish to thank all the IBM employees, visiting students and faculty members, and IBM postdoctoral fellows who made important contributions to the architecture and implementation of System R.

REFERENCES

(Note. References [2, 7-9, 11-13] are not cited in the text.)

1. ASTRAHAN, M.M., ET AL. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97-137.
2. CHAMBERLIN, D.D. Relational database management systems. *Comput. Surv. 8*, 1 (March 1976), 43-66.
3. CHAMBERLIN, D.D., ET AL. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976), 560-575.
4. GRAY, J.N., LORIE, R.A., PUTZOLU, G.R., AND TRAIGER, I.L. Granularity of locks and degrees of consistency in a shared database. Res. Rep. RJ1654, IBM Res. Lab., San Jose, Calif., 1975.
5. LORIE, R.A., AND NILSSON, J.F. An access specification language for a relational database system. Res. Rep. RJ2218, IBM Res. Lab., San Jose, Calif., Apr. 1978.
6. LORIE, R.A., AND WADE, B.W. The compilation of a very high level language. Res. Rep. RJ2008, IBM Res. Lab., San Jose, Calif., May 1977.
7. MCGEE, W.C. The information management system IMS/VS. *IBM Syst. J. 16*, 2 (1977), 84-168.
8. MCLEOD, D., AND MELDMAN, M. RISS—A generalized minicomputer relational database management system. In *Proc. AFIPS 1975 NCC*, vol. 44, AFIPS Press, Arlington, Va., pp. 397-402.
9. MYLOPOULOS, J., SCHUSTER, S., AND TSICHRITZIS, D. A multi-level relational system. In *Proc. AFIPS 1975 NCC*, vol. 44, AFIPS Press, Arlington, Va., pp. 403-408.
10. SELINGER, P. G., ASTRAHAN, M.M., CHAMBERLIN, D.D., LORIE, R.A., AND PRICE, T.G. Access path selection in a relational database management system. In *Proc. ACM SIGMOD 1979 Int. Conf. Management of Data*, 1979, pp. 23-24.
11. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189-222.
12. TODD, S.J.P. The Peterlee relational test vehicle—A system overview. *IBM Syst. J. 15*, 4 (1976), 285-308.
13. ZLOOF, M.M. Query by example: A database language. *IBM Syst. J. 16*, 4 (1977), 324-343.

Received May 1979; revised April 1980; accepted June 1980