

Research Report

SYSTEM R: AN ARCHITECTURAL UPDATE

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
R. W. Wade
R. A. Yost

IBM Research Laboratory
San Jose, California 95193

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

IBM

Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

SYSTEM R: AN ARCHITECTURAL UPDATE

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
R. W. Wade
R. A. Yost

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: System R is an experimental database management system which 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 database compiler. Host language programs with imbedded SQL statements are compiled by System R, which replaces the SQL statements by calls to a machine-language access module. The compilation 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. RSS, on the other hand, is a low level DBMS, supporting simple record at a time operators, along with complete recovery and concurrency control subsystems.

CONTENTS

Foreword	1
Description of System R Features	1
The SQL Language	1
Ad Hoc Query and Host Language Support	4
Data Independence	5
Views and Authorization	6
Integrated Data Dictionary	7
Compilation	7
Dynamic Database Definition	8
Transaction Management	8
Architecture	8
Relational Data System	9
Precompilation	12
Optimization	14
Executing a Precompiled Program	19
Treatment of 'Non-optimizable' Statements	20
Operations on Temporary tables	21
Dynamically Defined Statements	21
The Research Storage System	24
Segments	24
Tables	26
Indices	28
Links	29
Sort	30
Transaction Management	30
System Checkpoint and Restart	31
Concurrency Control	32
Summary and Conclusions	36
References and Bibliography	37

FOREWORD

System R is an experimental database management system designed and built at 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 [6]. 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. This paper is derived in part from three papers: information on RSS from [1], RDS from [7], and other material from [2].

The paper is divided into three sections. The Introduction clarifies the goals of the system, and gives some examples. 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

Perhaps the greatest impediment to the use of computerized data management systems is the cost and complexity of understanding and installing such systems. At present, installation of a database management system requires a staff skilled in tele-communications, operating systems, data management and in the application area. System R is a relational database management system designed to address this problem. System R is designed to allow easy definition of databases and of the applications which use them while still providing the function and performance available in most commercially available systems. 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. An example relational database describing employees and offices in a company appears in Figure 1. Examples of the use of SQL follow, using this simple database.

q1: Find the names of employees in the Paris office.

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

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	Biasgen	7152
Paris	Portal	9123
London	Portal	3278
Bonn	Roever	1287
...		

Figure 1. A fragment of a relational data base.

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 mapping':

```
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: Assuming office managers appear in the EMPLOYEE table, find all employees who make more than their office manager.

```
SELECT EMP.NAME, EMP.SAL
FROM EMPLOYEE EMP, EMPLOYEE MGR, OFFICE
WHERE MGR.SAL < EMP.SAL
AND OFFICE.MANAGER = MGR.NAME
AND OFFICE.LOCATION = EMP.OFFICE
```

q7: Insert a new employee in the EMPLOYEE table.

```
INSERT INTO EMPLOYEE(NAME, OFFICE, JOB):
<'Wade', 'San Jose', 'Service'>
```

(sets SALARY field to null)

q8: Close the Paris office.

```
DELETE EMPLOYEE
WHERE OFFICE = 'Paris'
```

```
DELETE OFFICE
WHERE LOCATION = 'Paris'
```

q9: 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 database: (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 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 EMPLOYEE WHERE JOB = 'programmer'
```

```
UPDATE EMPLOYEE SET SALARY = 14500 WHERE NAME = 'Davis'
```

The same SQL statements may be imbedded 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 EMPLOYEE SET SALARY = $X WHERE NAME = $Y;
```

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'. The cursor 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 SYR_CODE.

DATA INDEPENDENCE

SQL allows data accesses and updates to be expressed without mentioning or implying the existence of specific access paths 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. It is difficult to imagine how a system without a high level language would allow programs to adapt to new storage structures.

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 search 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 which 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 productive. Secondly 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.

To illustrate the point that System R gives programs some independence from data reorganization as the system is 'tuned', suppose there are two transactions which must be run periodically against the database in Figure 1.

T1: List all employees at a given office.

T2: List all employees with a certain job.

When written in SQL, these two queries may be described without considering how or where the records are stored and without specifying the access paths to be used to locate the records which satisfy the query. The transactions will work correctly no matter how the data is organized. If the users expect to run T1 99% of the time, and T2 1% of the time, a database structure which results in high performance is: given an office name, there should be a fast way to find all employees in that office, (e.g. an index on the OFFICE field of the the EMPLOYEE table), and records of employees at a give office should be clustered together in secondary storage (to minimize I/O).

Suppose, however, that these estimates are wrong or that the use of the system changes so that T2 runs 99% of the time. Performance on the above database structure will be very poor, and the database will have to be restructured. Namely the OFFICE index may be dropped, and a clustering index on JOB should be created. An essential characteristic of a flexible system is that this restructuring not require that programs T1 and T2 be rewritten.

In general, since System R supports a very high level language, most database 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

some views may not be modifiable (do not allow insert, delete, or update).

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 which 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 department. Further one may qualify certain fields of the view as read only. This gives value dependent authorization at the granularity of a field. 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 DICTIONARY

SQL is an integrated data definition and data manipulation language. In System R the description of the database is stored in user visible 'system' tables which may be read using the SQL language. The creation of a table or an access path results in new entries in these system tables. Users defining tables and other objects are encouraged to include English text which describes the 'meanings' of the objects. 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 non-procedural 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, System 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 users 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 which are executed only once and retrieve or modify only a few records.

DYNAMIC DATABASE DEFINITION

Any of the following can be done by any 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 database 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 non-catastrophic restart is transparent to operations personnel.

The transaction notion is the key to a successful recovery philosophy. A complex update of the database may involve many SQL statements. If a complex update fails or if the system crashes during the operation, the state of the database will probably be confused. One must be able to 'undo' partially completed transactions. Once a transaction 'commits' its updates, they will never be undone.

Transactions also supply the key to concurrency control. If multiple transactions concurrently read and write the same data anomalies might 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, and the Research Storage System. These two components are described in the following sections.

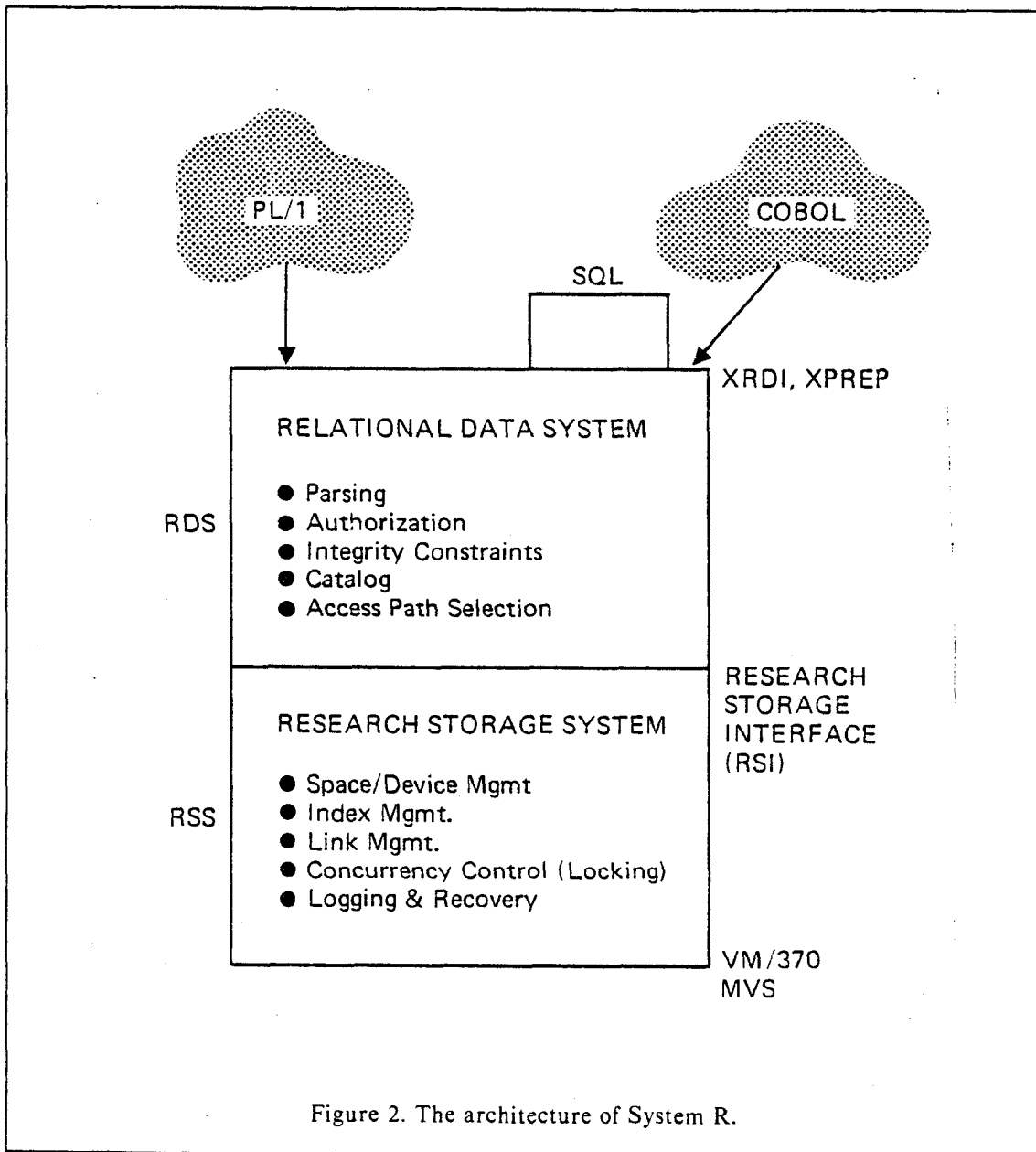
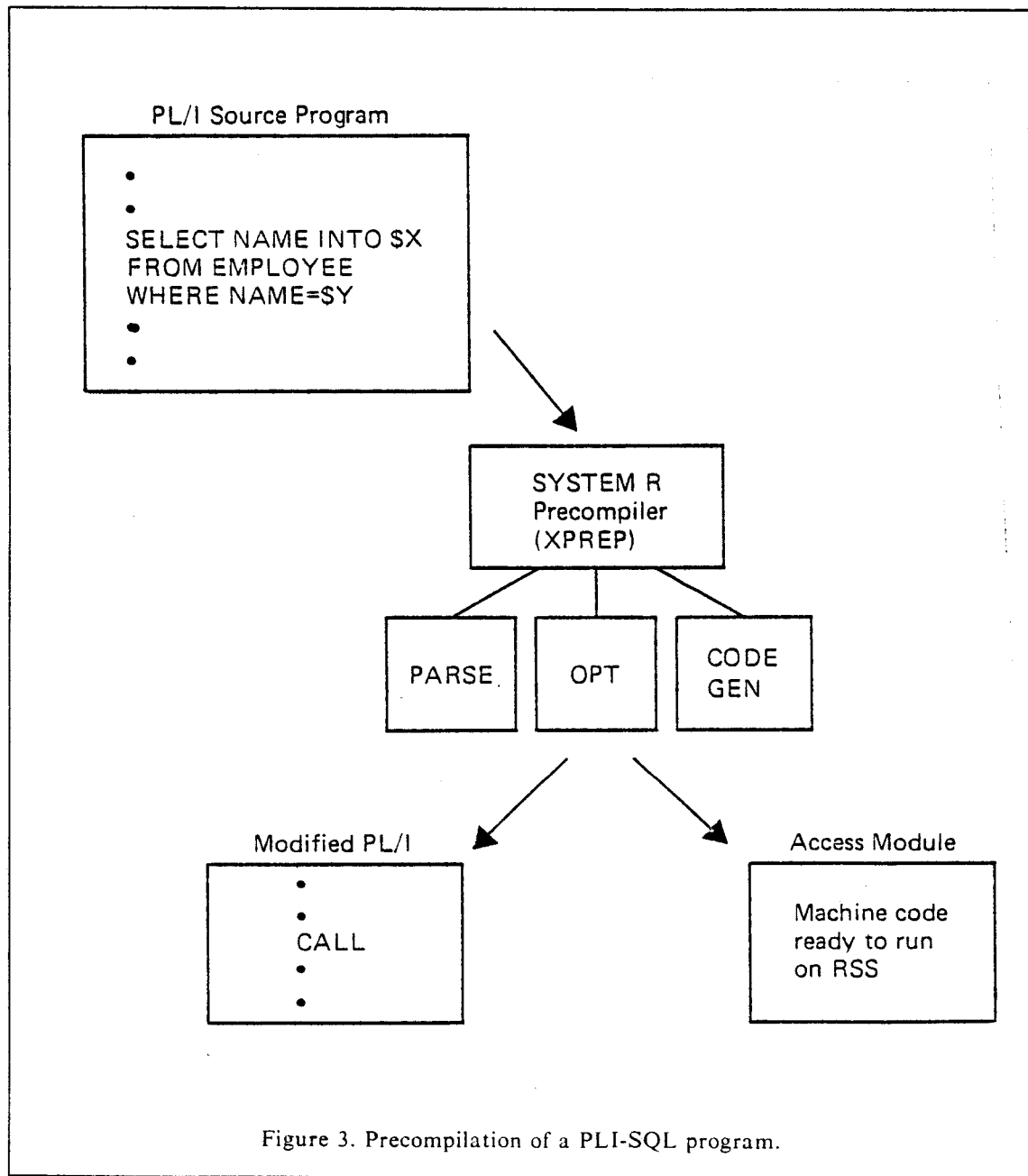


Figure 2. The architecture of System R.

RELATIONAL DATA SYSTEM

The 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-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 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 database to protect it from unauthorized modification. The precompilation step is illustrated in Figure 3.

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

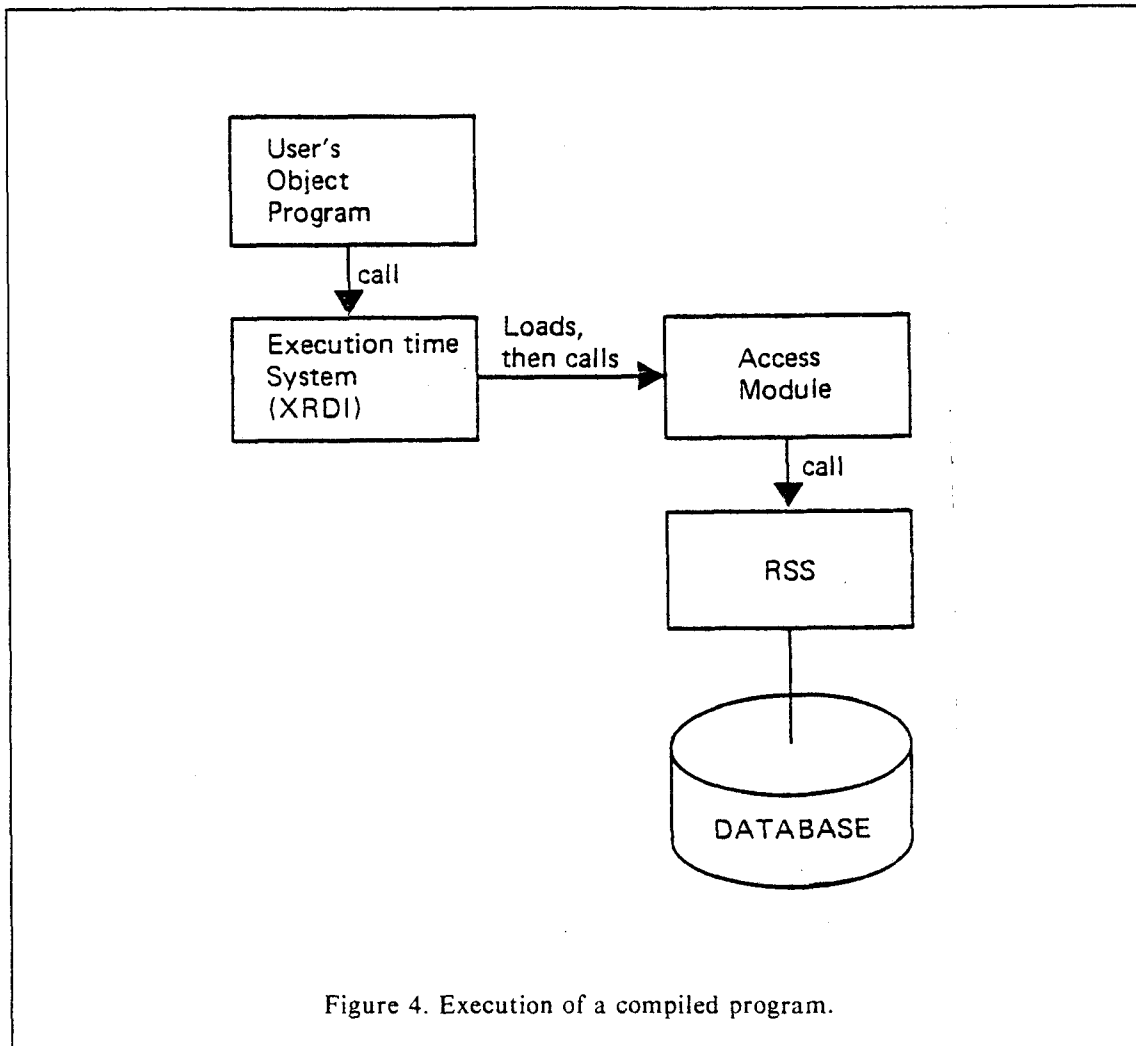
- (1) Much of the work of parsing, name-binding, access path selection, and authorization checking can be done once by the precompiler and thus removed from the process of running the canned program.
- (2) The access module, because it is tailored to one specific program, is much smaller and runs much 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 database by making calls to RSS, and delivers the result to the user's program. This process is illustrated in Figure 4.

The ad-hoc user of System R is supported by an SQL application program called the User-Friendly Interface (UFI), which controls dialog 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 UFI's purpose is to execute SQL statements which 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 UFI's access module 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 which 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 database 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 \$-signs) 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 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 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 design of this access path selection process is given briefly below, and in more detail in [16]. The optimizer represents its chosen access path by means of an ASL (Access Specification Language) [14] specification, 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 [15]. 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.

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 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. 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 statement are passed in with the AUXCALL.

If the SQL statement is an operation on a cursor, the situation is slightly more complex. The cursor is defined by a 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>. One may also UPDATE or DELETE the record currently addressed by a cursor. 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. 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.

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

The first stage of this selection is name resolution. The OPTIMIZER accumulates the names of tables and columns referenced in the query and looks them up in the System R catalogs to verify their existence and to retrieve information about them. This lookup determines, for example, that there is a table called EMPLOYEE, that it has a SALARY column which is the third stored column and of datatype INTEGER, etc.

The catalog lookup portion of the OPTIMIZER also obtains statistics about the referenced tables, and the access paths available on each of them. These will be used later in access path selection. After catalog lookup has obtained the datatype and length of each column, the OPTIMIZER rescans the SELECT-list and WHERE-tree to check for semantic errors and type compatibility in both expressions and predicate comparisons, (e.g. adding two character strings together is illegal).

View composition is also done if necessary. This replaces all references to view tables and columns with their underlying representations in the view definition. If the view definition contained predicate restrictions, these are ANDed to this query's WHERE tree. If these predicates contain subqueries, more query blocks will appear in the parse tree. After view composition every table reference in the parse tree is a stored table.

Finally the OPTIMIZER performs access path selection. (Reference [16] has a detailed discussion of this). The OPTIMIZER first determines the evaluation order among the query blocks in the statement. Then for each query block, the tables in the FROM list are processed. If there is more than one table in a block, permutations of the join order and of the method of joining are evaluated. The access paths that minimize total cost for the block are chosen from a tree of alternate path choices.

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:

$$\text{COST} = \text{PAGE FETCHES} + W \times (\text{RSI CALLS}).$$

This cost is a weighted measure of I/O (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.

During execution of the type-compatibility and semantic checking portion of the OPTIMIZER, each query block's WHERE tree of predicates is examined. The WHERE tree is initially considered to be in conjunctive normal form, and every conjunct is called a *boolean factor (BF)*. Boolean factors are notable because every record returned to the user must satisfy every boolean factor. A BF may be a single predicate or a subtree headed by an OR. It is either the whole WHERE tree or is connected to the tree root only by ANDs.

The RSS has a particularly efficient implementation for checking certain combinations of simple predicates, called 'search arguments,' or SARG's. Individual SQL predicates are said to be *sargable* if they can be put in the form: <column-name,scalar-comparison,literal-value>. An index is said to match a boolean factor if the boolean factor is a sargable predicate whose referenced column is the index key; e.g., an index on SALARY matches the predicate 'SALARY = 20000'. If an index matches a boolean factor, an access using that index is an efficient way to satisfy the boolean factor.

A sargable boolean factor is a BF composed of sargable predicates. A sargable BF can be put into disjunctive normal form and checked directly as RSS retrieval SARG's, eliminating the overhead of RSS calls to retrieve non-qualifying records. Several boolean factors may together be matched by an index; e.g., a NAME, LOCATION index matches NAME = 'Smith' AND LOCATION = 'San Jose'. BF's that are satisfied by an indexed access path or by incorporation into RSS SARG's are removed from the WHERE tree. Remaining predicates must be applied by the compiled program to test each record retrieved by an RSS call. Each boolean factor is entered into a predicate list along with a set of classifications recording whether it is eligible to become an RSS SARG, what kind of predicate it is, and which tables are referenced by the predicate.

During catalog lookup, the OPTIMIZER retrieves statistics on the tables in the query and on the access paths available on each table. The access paths considered are indexes and segment scans (see [16] and part 3 below on the RSS). Indexes retrieve records in key value order. A segment scan retrieves records in an RSS determined order but touches each data page only once. One index may be clustering, so that records having the same or adjacent key values are stored and on the same data page. The statistics kept are the following:

For each table T,

- NCARD(T), the cardinality of table T
- TCARD(T), the number of 4K pages occupied by T
- PCTPAGES(T), the percentage of pages in the segment containing pages
from this relation

for each column C on table T,

- HIGH2KEY(C), the second highest value for this column present in the table
- HIGH2KEY(C), the second lowest value for this column present in the table

and for each index I on table T,

- ICARD(I), the number of distinct keys in index I
- FIRSTICARD(I), the number of distinct keys in index I, considered as a one-column index only.
For single column indexes, ICARD = FIRSTICARD,
but for multi-column indexes, usually
FIRSTICARD << ICARD.
- NINDX(I), the number of pages in index I.
- CLUSTER(I), whether this is a clustering index or not

These 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 database operations and the locking bottleneck this 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 assigns a *selectivity factor* 'F' for each boolean factor in the predicate list. This selectivity factor very roughly corresponds to the expected fraction of records which will satisfy the predicate. The product of the selectivity factors with the table cardinalities produces QCARD, the expected cardinality of the query result.

For single tables, the cheapest access path is obtained by evaluating the cost for each available access path. For each such access path, a predicted cost is computed along with the ordering of the records it will produce. An order is 'interesting' with respect to a query if the query result must be grouped or sorted in that order. Scanning along the SALARY index in ascending order, for example, will produce some cost C and a record order of SALARY (ascending). To find the cheapest access plan for a single table query, we need only to examine the cheapest access path which produces records in each 'interesting' order and the cheapest 'unordered' access path. If it is a single table query, and if there are no GROUP BY or ORDER BY clauses in the query, then there will be no interesting orderings, and the cheapest access path is the one chosen. If there are GROUP BY or ORDER BY clauses, or joins, then the cost for producing that interesting ordering must be compared to the cost of the cheapest unordered path *plus* the cost of sorting QCARD records into the proper order. The cheapest of these alternatives is chosen as the plan for the query block.

For handling joins, the System R optimizer chooses among two methods for performing 2-way joins. Reference [4] showed that for the System R context, one of these two methods is optimal. We first describe these methods, and then we discuss how they are extended for n-way joins. In joins involving two tables, one table is called the *outer* table, from which a record will be retrieved first, and the other is the *inner* table, from which records will be retrieved, possibly depending on the values obtained in the outer table record. A predicate which relates columns of two tables to be joined is called a *join predicate*. The columns referenced in a join predicate are called *join columns*.

The first join method, called the *nested loops* method, begins by opening a cursor on the outer table is opened and retrieving a record. For each outer table record obtained, a cursor is opened on the inner table to retrieve, one at a time, all the records of the inner table which satisfy the join predicate. The composite records formed by the outer table record/inner table record pairs comprise the result of this join. A clustering index which matches the join predicate is a particularly good path for the inner table.

The second join method, called *merging*, begins by opening cursors on two access paths which produce outer and inner table records in the join column order. An outer table record is retrieved. As in the nested loop join, inner table records are retrieved which match the outer table record on the join column value. As before, the composite records formed by the outer table record/inner table record pairs comprise the result of this join. This method differs from the nested loop method in that for each new join column value, the position of the first matching inner table record (if any) is marked by a placeholder. Thus when the outer table cursor is advanced, if the next outer record's join column value is the same as the previous outer record's, then the inner table cursor can be continued from the placeholder position. If the next outer record's join column value is greater than that of the previous outer record, then the inner table cursor is advanced from its current position. In doing this, the inner cursor may find no matches for that outer record, and this will cause an advance of the outer table cursor. This iterates until the next match of inner and outer join column values. It should be noted that all of the inner and outer records will be scanned at least once and that those passed over will never be rescanned during the rest of the join processing.

Merge joins require the outer and inner tables to be scanned in join column order. This implies that, along with the columns mentioned in ORDER BY and GROUP BY, columns of equi-join predicates (those of the form $Table1.column1 = Table2.column2$) also define 'interesting' orders. If there is more than one join predicate for a pair of tables, one of them is used as the join predicate and the others are treated as ordinary predicates. The merging method is only applied to equi-joins, although in principle it could be applied to other types of joins. If one or both of the tables to be joined has no indexes on the join column, it must be sorted into a temporary list which is ordered by the join column.

The costs for joins are computed from the costs of the scans on each of the tables and the cardinalities. The costs of the scans on each of the tables are computed using the cost formulas for single table access paths. Let $C\text{-outer}(\text{path1})$ be the cost of scanning the outer table via path1, and N be the cardinality of the outer table records which satisfy the eligible predicates. N is computed by:

$$N = (\text{product of the cardinalities of all tables } T \text{ of the join so far}) \times (\text{product of the selectivity factors of all eligible predicates}).$$

Let $C\text{-inner}(\text{path2})$ be the cost of scanning the inner table, applying all eligible predicates. Note that in the merge join this means scanning the contiguous group of the inner table which corresponds to one join column value in the outer table. Then the cost of a nested loop join is

$$C\text{-join}(\text{path1}, \text{path2}) = C\text{-outer}(\text{path1}) + N \times C\text{-inner}(\text{path2}) + \text{sort cost if r}$$

An n -way join access path consists of an ordered list of the tables to be joined, the join method used for each join, and a plan indicating how each table is to be accessed. If either the outer composite table or the inner table needs to be sorted before joining, then that is also included in the plan.

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 FETCH-CALL) and proceeds to process the original SQL statement from which it was compiled, using as needed the host-program variables which were 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 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 their changes are complete and consistent; i.e., when one user-defined unit of work has completed. 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 made by another concurrently running transaction which will 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 re-acquire 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.
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 'NON-OPTIMIZABLE' 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 will refer to SQL statements which can be handled in this way as 'non-optimizable' statements.

When the System R precompiler encounters a non-optimizable 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 labelled as an 'INTERPSECT', to distinguish it from a section containing machine code, which is labelled 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 PARSESECT 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 a 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, such as:

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

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 section may be executed many times, with different parameters, without re-invoking the System R 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 based on 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 are 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 addition 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.'

THE RESEARCH STORAGE SYSTEM

This section of the paper is concerned with the Research Storage System or RSS, a low level DBMS which provides underlying support for System R. The RSS was designed as a desirable target for the SQL compiler. As a result, 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 Relational Data System 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 database 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 which 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 which can be found in other systems, both relational and non-relational, such as the use of index and pointer chain structures. The areas which 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, multi-point recovery for in-process transactions, a novel and efficient technique for system checkpoint and restart, multiple levels of isolation from the actions of other concur-

rent users, and automatic locking at the granularity of segments, tables, pages, and/or single records. The next several sections describe all of these RSS functions, and include a sketch of the implementation.

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. A special segment is dedicated to the storage of transaction logs for backing out or redoing the changes made by individual transactions.

Several types of segments are supported, each with its own combination of functions and overhead. For example, one type is intended for storage of shared data, and has provisions for concurrent access, transaction backout, and for recovery of the segment's contents to a previous state. Another segment type is intended for low overhead storage of temporary tables, and has no provision for either concurrent access or segment recovery. A maximum length is associated with each segment, and is chosen by a user during initialization of the system.

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 equal-sized *pages*, which are referenced and formatted by various components of the RSS. Physical page slots in the disk extents are allocated to segments dynamically upon first reference, by checking and modifying bit maps associated with the disk extents. Physical page slots are freed when access path structures are destroyed or when the contents of a segment are destroyed. This dynamic allocation scheme allows for the definition of many large sized segments, to accommodate large intermediate results and growing databases. Facilities are provided to cluster pages on physical media so that sequential or localized access to segments can be handled efficiently.

The RSS maintains a page map for each segment, which is used to map each segment page to its location on disk. Such a map is maintained as a collection of equal-sized *blocks*, which are allocated statically. A page request is handled by allocating space within a main memory buffer shared among all concurrent users. In fact two separate buffers are managed, one for the page map blocks and one for the segment pages themselves. Both pages and blocks 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 provides a novel technique to handle segment recovery, by associating with each recoverable segment *two* page maps, called current and backup. When the OPEN_SEGMENT operator is issued, to make the segment available for processing, these page maps have identical entries. When a component of the RSS later requests access to a page, with intent to update (after suitable locks have been acquired), the RSS checks whether this is the first update to the page since the OPEN or since the last SAVE_SEGMENT operation. If so, a new page slot is allocated nearby on disk, the page is accessed from its original disk location, and the current page map is then modified to point to the new page slot. When the page is later replaced from the buffer, it will be directed to the new location, while the backup page and backup page map are left

When the SAVE_SEGMENT operator is issued, the disk pages bound to segments are brought up to date, by storing through all buffer pages which have been updated. Both page maps are then scanned, and any page which has been modified since the last save point has its old page slot released. Finally the backup page map entries are set equal to the current page map entries, and the cycle is complete.

With this technique, the RESTORE_SEGMENT operator is relatively simple, since the backup page map points to a complete, consistent copy of the segment. The current page map is simply set equal to the backup one, and newly allocated page slots are released. The SAVE_SEGMENT and RESTORE_SEGMENT functions are useful for recovering a previous version of private data, and also for support of system checkpoint and restart, as explained below. However, the effect of restoring a segment of public data may be to undo changes made by several transactions, since each of them may have modified data since the segment was last saved. Another mechanism is therefore used to back out only those changes made by a single transaction, and is explained below.

Note that this recovery scheme depends on the highly stylized management of two page maps per segment, and on an ability to control when pages are stored through from main memory to disk. These particular requirements led to the decision to handle our own storage management and I/O for RSS segments, rather than relying on the automatic paging of virtual memory in the operating system. See Reference [13].

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 database reload and without immediate modification to existing records.

Two field types are supported: fixed length and variable length. For both field types, a special protocol is used at the RSI to generate an undefined value. This feature has a number of uses, but a particularly important one is that when the user adds new fields to an existing table, values for those fields in each existing record are treated as undefined until they are explicitly updated.

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 cursor or *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. The access paths which are supported include a value-determined ordering of records through use of an index, an RDS-determined ordering of records through use of a link (see below for discussions of indexes and links), and an RSS-determined ordering of records in a table. For all of these access paths, the RDS may attach a search argument (SARG) to each NEXT operation. The search argument may be any disjunctive normal form expression where each atomic expression has 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 SARGs because of the performance advantage due to 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. TID's 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 database reorganization.

The RSS stores and accesses records within tables, and maintains pointer chains to implement the links described below. Each record is stored as a contiguous sequence of field values within a single page. Field lengths are also included for variable length fields. A prefix is stored with the record, for use within the RSS. The prefix contains such information as the table identifier, the pointer fields (TID's) for link structures, the number of stored data fields, and the number of pointer fields. These numbers are employed to support dynamic creation of new fields and links to existing relations, without requiring immediate access or modification to the existing records. Records are found only on pages which have been reserved as data pages. Other pages within the segment are reserved for the storage of index or internal catalog entries. A given data page may contain records from more than one table, so that extra page accesses can be avoided when records from different tables are accessed together. When a scan is executed on a table (rather than an index or link), an internal scan is generated on all non-empty data pages within the segment containing that table. Each such data page is touched once, and the prefix of each record within the page is checked to see if it belongs to the table.

The implementation of record identifier access is a hybrid scheme, which combines the speed of a byte address pointer with the flexibility of indirection. Each record identifier is a concatenation of a page number within the segment, along with a byte offset from the bottom of the page. The offset denotes a special entry or 'slot' which contains the byte location of record in that page. This technique allows efficient utilization of space within data pages, since space can be compacted and records moved with only local changes to the pointers in the slots. The slots themselves are never moved from their positions at the bottom of each data page, so that existing TID's can still be employed to access the records. In the rare case when a record is updated to a longer total value and insufficient space is available on its page, an overflow scheme is provided to move the record to another page. In this case the TID points to a tagged overflow record which is used to reference the other page. If the record overflows again, the original overflow record is modified to point to the newest location. Thus, a record access via a TID almost always involves a single page access, and never involves more than two page accesses (plus possible accesses to the page map blocks).

In order to tune the database 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. Another use would be to cluster records of one table near particular records of another table, because of matching values in some of the fields. This clustering rule would result in high performance for relational join operations, as well as for the support of hierarchical and network applications.

A variation on a table, used primarily for sorting, is a LIST. A list is an ordered collection of records which can only be accessed sequentially.

INDICES

An *index* in the RSS is an access path which 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 particular 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 which 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 hierarchic structure, in the style of B-trees and of Key Sequenced Data Sets in IBM's VSAM access method. Each page is a node within the hierarchy, and contains an ordered sequence of index entries, with free space kept at the end. For non-leaf nodes, an entry consists of a <sort value, pointer> pair. The pointer addresses another page in the same structure, which may be either a leaf page or another non-leaf page. In either case, the target page contains entries for sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values, along with an ascending list of TID's for records having exactly those sort values. The leaf pages are chained in a doubly-linked list, so that sequential access can be supported from leaf to leaf. Pages are searched using binary search when possible, otherwise with a particularly efficient variant of square root (or quadratic) search which we call slide search.

In order to handle variable length, multi-field 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 search. This encoding eliminates the need for costly padding of each field and slow field-by-field comparison [5].

LINKS

A *link* in the RSS is an access path which is used to relate records in one table to records in another table. Links are supported by pointer chains. A binary link provides a path from single records (parents) in one table to sequences of records (children) in another table. The RSS allows a user to determine which records will be children under a given parent, and the relative order of children under a given parent, through the CONNECT and DISCONNECT operators. Operators are then available to scan the children of a parent or go directly from a child to its parent along a given link. In general, a record in the parent table may have no children, and a record in the child table may have no parent. Also, records in a table may be parents and/or children in an arbitrary number of different links. The only restriction is that a given record can appear only once within a given link.

Links are intended to be used in System R to connect child records to a parent, based on value matches in one or more fields. With such a structure, the RDS could access records in one table, say the Employee table, based on matching the Department Number field in a record of the Department table. This function is useful for supporting one to many join operations. However, in the current implementation, the RDS does not take advantage of the existence of links in access path selection. The link provides direct access to the correct Employee records

from the Department record (and vice-versa), while use of an index may involve access to several pages in the index. An advantage is gained over indexes when the child records have been clustered on the same page as the parent, so that no extra pages are touched using the link, while several pages may be touched in a large index.

Another possible use of links is to provide reasonably fast associative access to a table, without an extra index. In the above example, if the Department table has an index on Department Number, then the RDS could gain associative access to Employee records for a given value of Department Number by using the Department table index and the binary link - even if the Department record is not being referenced by the SQL user.

Links are maintained in the RSS by storing TID's in the prefix of records. New links can be defined at any time. When a new link is defined, a portion of the prefix is assigned to hold the required entries. This operation does not require access to any of the existing records, since new prefix space for an existing record is formatted only when the record is connected to the link. When necessary, the prefix length is enlarged through the normal mechanisms used for updates and new data fields. An existing link can be dropped at any time. When this occurs, each record in the corresponding table(s) is accessed by the RSS, in order to invalidate the existing prefix entries, and make the space available for subsequent link definitions.

SORT

The RSS contains a sort component which sorts records from a table according to an RDS provided order specification. The sort component can sort all of a table, or any row (specified by SARGs) or column subset of a table, one or more fields, either ascending or descending by field. Sort is carried out using a sort-merge algorithm, using QUICKSORT as the internal sort of the records on a page, and a peerage tournament to carry out the merge. The output of sort is an ordered LIST.

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 marked by the `BEGIN_TRANSACTION` and `END_TRANSACTION` operat Various resources are assigned to transactions by the RSS, using the locking techniques described below. A transaction recovery scheme is provided which allows a transaction to be backed out to the beginning of the transaction.

Transaction recovery occurs when the `RESTORE_TRANSACTION` is issued, when the end-user issues a cancel, or when the RSS initiates the procedure to handle deadlock. The effect is to undo all the changes made by that transaction to recoverable data. Those changes include

all the record and index modifications caused by INSERT, DELETE, and UPDATE operations, all the link modifications caused by CONNECT and DISCONNECT operations, and even all the declarations for defining new tables, indexes and links. Finally, all locks on recoverable data which 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 recoverable objects, along with the operation object identification. Modifications to index structures are not logged, since their values can be determined from data values and index catalog information.

During transaction recovery, the log entries for the transaction are read in last-in-first-out order. Special routines are employed to undo all the listed modifications for incomplete transactions, and to redo modifications for transactions which completed after the crash.

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

The RSS actually supports a more general recovery scheme involving multiple save points and incremental backout as described in [17], but the RDS does not support this feature at this time.

SYSTEM CHECKPOINT AND RESTART

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

In the RSS, system recovery mechanisms have been developed which use disk storage to recover in the event of a 'soft' failure which causes the contents of main memory to be lost, but which 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 that disk storage is destroyed, and is oriented toward less frequent checkpoints.

The recovery mechanisms are heavily dependent on the segment recovery functions described above, and also on the availability of transaction logs. A Monitor Machine has the responsibility for scheduling checkpoints, based on parameters set during system startup. When a checkpoint is required, the Monitor quiesces all activity within the RSS, at a point of physical

consistency: transactions may still be in progress, but may not be executing an RSI operation. The technique for halting RSS activity is to acquire a special RSS lock in exclusive mode, which every activation of the RSS code acquires in share mode before executing an RSI operation, and releases at the end of the operation. The Monitor records the current status of the system in a checkpoint log record and then issues the SAVE_SEGMENT operator to bring disk copies of all relevant segments up to date. Finally the RSS lock is released, and transactions are allowed to resume.

When a soft failure occurs, the RESTORE_SEGMENT operator is used to restore the contents of all saved segments. Recall that the restore function is a relatively simple one, involving the setting current page map values equal to the backup page map values, and the releasing of pages allocated since the save point. The log segment is saved more frequently than normal data segments, effectively at the end of each transaction, and contains 'after' values as well as 'before' values of modified data. Therefore transactions completing after the last database save can be redone automatically. In addition, the transaction logs are used to back out transactions which were ongoing at the checkpoint and did not complete, in order that a consistent database state is reached.

Our tape-oriented recovery scheme is an extension of the above one. In order to recover in the event of lost disk data, some technique is required to get a sufficient copy of data and log information to tape, and to guarantee that the log of subsequent modifications is not lost by a 'hard' crash. The technique we have chosen is to periodically archive the contents of the database disks to tape, and to maintain dual logs [17].

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 insure 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 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 insure that data updated by some ongoing transaction, say T1, is not updated by another, say T2. Otherwise, the backout of transaction T1 will undo T2's update, and violate our principle of isolated backout.

At the *physical* level of pages, locking techniques are required to insure 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 could 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, TID's and key value intervals and holding them 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 which can be tailored to future research applicati

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.

In order to experiment with various lock protocols, the RSS supported multiple levels of consistency which control the 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 at the RDI.) 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 may be accessed, and one may read different values for the same data item during the same 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. On the other hand, this level may be entirely satisfactory for gathering statistical information from a large database, when exact results are not required.

The HOLD option can be used during read operations to insure against lost updates or dirty data values.

In a transaction with Level 2 consistency, the user is assured that every item read is 'clean', i.e. that the transaction which 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, or that associative access will yield the same item. 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. Transactions running at Level 2 consistency may use the HOLD option during read operations preceding updates, to insure against lost updates.

For the highest consistency level, called Level 3, the user sees the logical equivalent of a single user system. 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 answer will occur every time within the same transaction. Thus, the RDS can effectively lock a set of items defined by a SQL predicate and obtained by any search strategy, against insertions into or deletions from the set. Similarly, if the RDS employs an index to access the unique record where Name = 'Smith', and no such record exists, then the same nonexistence result is assured for subsequent accesses.

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 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, a old balance of one and new balance of the other. Although one could use the HOLD option to handle this particular problem, many such associations may not be understood in a more complex database environment, even by relatively experienced programmers.

It has been a surprise to us that the degree 3 consistency lock protocol is no more expensive than the degree 2 protocol. In several surprising cases degree 3 is cheaper than degree 2! For that reason most users elect degree 3 consistency (the default). Degree 1 consistency is no longer supported by System R.

The RSS components set locks automatically, in order 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 p which are held at least during the execution of each RSI operation, in order to insure that data and index pages are accessed and maintained correctly.

The RSS employs a single lock mechanism to synchronize access to all objects. This synchronization is handled by a set of procedures which maintain a collection of queue structures called *gates* in shared, read/write memory. Internal components can request a lock by giving an eight-character name for the object, using such names as a record identifier, index value or page number. If the resource is already locked it has a gate. If not, then a gate is allocated. The gate will be deallocated when its queue becomes empty.

An internal request to lock an object has several parameters: the name of the object, the mode of the lock (such as shared, exclusive or various other modes mentioned below), and an indication of lock duration, so that the RSS can quickly release all locks held for a single RSI call, or all locks held for the entire transaction.

The choice of lock duration is influenced by several factors, such as the type of action requested by the user, and the consistency level of the transaction.

Data items can be locked at various granularities, to insure that various applications run efficiently. For example, locks on single records are effective for transactions which access small amounts of data, while locks on entire tables or even entire segments are more reasonable for transactions which 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. The basic idea of the scheme is that separate locks are associated with each granularity of object, such as segment, table and record. If the RDS requests a lock on an entire segment in share or exclusive mode, then every record of every table in the segment is implicitly locked in the same mode. If the RDS requests a lock on a single table, say in exclusive mode, but does not wish exclusive access to the entire segment, then the RSS first generates an automatic request for a lock in *intent-exclusive* mode on the segment, before requesting an exclusive lock on the table. This intent-exclusive lock is compatible with other intent locks, but incompatible with share and exclusive locks. The same protocol is extended to include locks on individual records, through automatic acquisition of intent locks on the segment and table, before a lock is acquired on the record in share or exclusive mode.

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 using the transaction recovery scheme described above. (To simplify the code, special provisions are made for transactions which need locks and are already backing up.) Reference [10] 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 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.

The RSS, on the other hand, is a low level database management system with complete locking, recovery, and transaction management subsystems. The locking subsystem, 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 database at the same time.

REFERENCES AND BIBLIOGRAPHY

- [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 Database Management.' *ACM Transactions on Database Systems*, Vol. 1, No. 2, June 1976 (pp. 97-137.)
- [2] M. M. Astrahan, et. al., 'System R, A Relational Database Management System', *IEEE Computer Magazine*, May 1979.
- [3] M. W. Blasgen, et. al., 'The Convoy Phenomenon', *ACM Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 20-25. To appear in SIGOPS transactions.
- [4] Blasgen, M.W. and K. P. Eswaran, 'Storage and Access in Relational Data Bases', *IBM Systems Journal*, Vol 16, No. 4, 1977.
- [5] Blasgen, M.W., K. P. Eswaran, and R. G. Casey, 'An Encoding Method for Multifield Sorting and Indexing', *CACM*, Vol. 20, No. 11. November 1977.
- [6] 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*, Vol. 20, No. 6, Nov. 1976.
- [7] D. D. Chamberlin, et. al., 'Support for Repetitive Transactions and Ad-Hoc Query in System R', *IBM Research Report: RJ 2551*, 1979.
- [8] Gray, J.N., V. Watson, 'A Shared Segment and Interprocess Communication Facility for VM/370', *IBM San Jose Research Laboratory Report: RJ 1579*, May 1975.
- [9] Gray, J.N., 'Notes on Data Base Operating Systems', *Operating Systems - An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller editors, Springer Verlag, 1978 pp.393-481. Also *IBM Research Report: RJ 2188*.
- [10] Gray, J.N., R.A. Lorie, G.F. Putzolu, 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, 1976, pp. 365-394. Also *IBM Research Report: RJ 1606*.

- [11] P. P. Griffiths and B. W. Wade, 'An Authorization Mechanism for A Relational Database System', ACM Transactions on Database Systems, Vol 1, No. 3, 1976.
- [12] Eswaran, K. P., J.N. Gray, R.A. Lorie, I.L. Traiger, 'On the Notions of Consistency and Predicate Locks in a Relational Database System,' CACM, Vol. 19, No. 11, Nov. 1976, pp. 624-634.
- [13] Lorie, R.A., 'Physical Integrity in a Large Segmented Database.' ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, pp. 91-104
- [14] R. A. Lorie and J. F. Nilsson. 'An Access Specification Language for a Relational Database System.' IBM J. of Research and Development, Vol. 23, No. 3, May 1979, pp.286-298.
- [15] R. A. Lorie and B. W. Wade. 'The Compilation of a Very High Level Language.' Research Report RJ2008, IBM Research Laboratory, San Jose, CA., May 1977.
- [16] 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.' Proceedings of the 1979 SIGMOD Conference.
- [17] Gray, J.N., P.R. McJones, M.W. Blasgen, R.A. Lorie, T.G. Price, G.F. Putzolu, I.L. Traiger, 'The Recovery Manager of a Data Management System', IBM Research Report, 1979.
- [18] Strong, H.R., I.L. Traiger, G. Markoski, 'Slide Search', IBM Research Report: RJ 2274, 1878.