System R: Relational Approach to Database Management

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

IBM Research Laboratory

System R is a database management system which provides a high level relational data interface. The system provides a high level of data independence by isolating the end user as much as possible from underlying storage structures. The system permits definition of a variety of relational views on common underlying data. Data control features are provided, including authorization, integrity assertions, triggered transactions, a logging and recovery subsystem, and facilities for maintaining data consistency in a shared-update environment.

This paper contains a description of the overall architecture and design of the system. At the present time the system is being implemented and the design evaluated. We emphasize that System R is a vehicle for research in database architecture, and is not planned as a product.

Key Words and Phrases: database, relational model, nonprocedural language, authorization, locking, recovery, data structures, index structures
CR categories: 3.74, 4.22, 4.33, 4.35

1. INTRODUCTION

The relational model of data was introduced by Codd [7] in 1970 as an approach toward providing solutions to various problems in database management. In particular, Codd addressed the problems of providing a data model or view which is divorced from various implementation considerations (the data independence problem) and also the problem of providing the database user with a very high level, nonprocedural data sublanguage for accessing data.

To a large extent, the acceptance and value of the relational approach hinges on the demonstration that a system can be built which can be used in a real environment to solve real problems and has performance at least comparable to today's existing systems. The purpose of this paper is to describe the overall architecture and design aspects of an experimental prototype database management system called System R, which is currently being implemented and evaluated at the IBM San Jose Research Laboratory. At the time of this writing, the design has been

Authors' address: IBM Research Laboratory, San Jose, CA 95193.

CONTENTS

---

completed and major portions of the system are implemented and running. However, the overall system is not completed. We plan a complete performance evaluation of the system which will be available in later papers.

The System R project is not the first implementation of the relational approach [12, 30]. On the other hand, we know of no other relational system which provides a complete database management capability—including application programming as well as query capability, concurrent access support, system recovery, etc. Other relational systems have focused on, and demonstrated, feasibility of techniques for solving various specific problems. For example, the IS/1 system [22] demonstrated the feasibility of supporting the relational algebra [8] and also developed optimization techniques for evaluating algebraic expressions [29]. Techniques for optimization of the relational algebra have also been developed by Smith and Chang at the University of Utah [27]. The extended relational memory (XRM) system [19] developed at the IBM Cambridge Scientific Center has been used as a single user access method by other relational systems [2]. The SEQUEL prototype [1] was originally developed as a single-user system to demonstrate the feasibility of supporting the SEQUEL [5] language. However, this system has been extended by the IBM Cambridge Scientific Center and the MIT Sloan School Energy Laboratory to allow a simple type of concurrency and is being used as a component of the Generalized Management Information System (GMIS) [9] being developed at MIT for energy related applications. The INGRES project [16] being developed at the University of California, Berkeley, has demonstrated techniques for the decomposition of relational expressions in the QUEL language into "one-variable

queries." Also, this system has investigated the use of query modification [28] for enforcing integrity constraints and authorization constraints on users. The problem of translating a high level user language into lower level access primitives has also been studied at the University of Toronto [21, 26].

Architecture and System Structure

We will describe the overall architecture of System R from two viewpoints. First, we will describe the system as seen by a single transaction, i.e. a monolithic description. Second, we will investigate its multiuser dimensions. Figure 1 gives a functional view of the system including its major interfaces and components.

The Relational Storage Interface (RSI) is an internal interface which handles access to single tuples of base relations. This interface and its supporting system, the Relational Storage System (RSS), is actually a complete storage subsystem in that it manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout, transaction recovery, and system recovery. Furthermore, it maintains indexes on selected fields of base relations, and pointer chains across relations.

The Relational Data Interface (RDI) is the external interface which can be called directly from a programming language, or used to support various emulators and other interfaces. The Relational Data System (RDS), which supports the RDI, provides authorization, integrity enforcement, and support for alternative views of data. The high level SEQUEL language is embedded within the RDI, and is used as the basis for all data definition and manipulation. In addition, the RDS maintains the catalogs of external names, since the RSS uses only system generated internal names. The RDS contains an optimizer which chooses an appropriate access path for any given request from among the paths supported by the RSS.

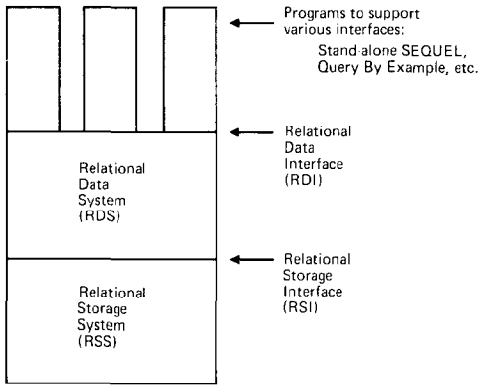

FIG. 1. Architecture of System R

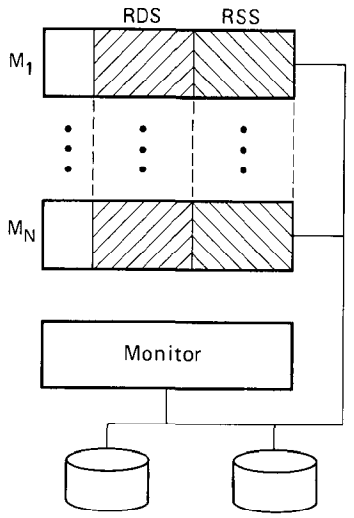

FIG. 2. Use of virtual machines in System R

The current operating system environment for this experimental system is VM/370 [18]. Several extensions to this virtual machine facility have been made [14] in order to support the multiuser environment of System R. In particular, we have implemented a technique for the selective sharing of read/write virtual memory across any number of virtual machines and for efficient communication among virtual machines through processor interrupts. Figure 2 illustrates the use of many virtual machines to support concurrent transactions on shared data. For each logged-on user there is a dedicated *database machine.* Each of these database machines contains all code and tables needed to execute all data management functions; that is, services are not reserved to a centralized machine.

The provision for many database machines, each executing shared, reentrant code and sharing control information, means that the database system need not provide its own multitasking to handle concurrent transactions. Rather, one can use the host operating system to multithread at the level of virtual machines. Furthermore, the operating system can take advantage of multiprocessors allocated to several virtual machines, since each machine is capable of providing all data management services. A single-server approach would eliminate this advantage, since most processing activity would then be focused on only one machine.

In addition to the database machines, Figure 2 also illustrates the Monitor Machine, which contains many system administrator facilities. For example, the Monitor Machine controls logon authorization and initializes the database machine for each user. The Monitor also schedules periodic checkpoints and maintains usage and performance statistics for reorganization and accounting purposes.

In Sections 2 and 3 we describe the main components of System R: the Relational Data System and the Relational Storage System.

2. THE RELATIONAL DATA SYSTEM

The Relational Data Interface (RDI) is the principal external interface of System R. It provides high level, data independent facilities for data retrieval, manipulation, definition, and control. The data definition facilities of the RDI allow a variety of alternative relational views to be defined on common underlying data. The Relational Data System (RDS) is the subsystem which implements the RDI. The RDS contains an optimizer which plans the execution of each RDI command, choosing a low cost access path to data from among those provided by the Relational Storage System (RSS).

The RDI consists of a set of operators which may be called from PL/I or other host programming languages. (See Appendix I for a list of these operators.) All the facilities of the SEQUEL data sublanguage [5] are available at the RDI by means of the RDI operator called SEQUEL. (A Backus-Naur Form (BNF) syntax for SEQUEL is given in Appendix II.) The SEQUEL language can be supported as a stand-alone interface by a simple program, written on top of the RDI, which handles terminal communications. (Such a stand-alone SEQUEL interface, called the User-Friendly Interface, or UFI, is provided as a part of System R.) In addition, programs may be written on top of the RDI to support other relational interfaces, such as Query by Example [31], or to simulate nonrelational interfaces.

Host Language Interface

The facilities of the RDI are basically those of the SEQUEL data sublanguage, which is described in [5] and in Appendix II. Several changes have been made to SEQUEL since the earlier publication of the language; they are described below.

The illustrative examples used in this section are based on the following database of employees and their departments:

```
EMP(EMPNO, NAME, DNO, JOB, SAL, MGR)
DEPT(DNO, DNAME, LOC, NEMPS)
```

The RDI interfaces SEQUEL to a host programming language by means of a concept called a *cursor*. A cursor is a name which is used at the RDI to identify a set of tuples called its *active set* (e.g. the result of a query) and furthermore to maintain a position on one tuple of the set. The cursor is associated with a set of tuples by means of the RDI operator SEQUEL; the tuples may then be retrieved, one at a time, by the RDI operator FETCH.

Some host programs may know in advance exactly the degree and data types of the tuples they wish to retrieve. Such a program may specify, in its SEQUEL call, the program variables into which the resulting tuples are to be delivered. The program must first give the system the addresses of the program variables to be used by means of the RDI operator BIND. In the following example, the host program identifies variables X and Y to the system and then issues a query whose results are to be placed in these variables:

```
CALL BIND('X', ADDR(X));
CALL BIND('Y', ADDR(Y));
CALL SEQUEL(C1,  'SELECT NAME:X,  SAL:Y
                  FROM EMP
                  WHERE JOB = ''PROGRAMMER'' ');
```

The SEQUEL call has the effect of associating the cursor C1 with the set of tuples which satisfy the query and positioning it just before the first such tuple. The optimizer is invoked to choose an access path whereby the tuples may be materialized. However, no tuples are actually materialized in response to the SEQUEL call. The materialization of tuples is done as they are called for, one at a time, by the FETCH operator. Each call to FETCH delivers the next tuple of the active set into program variables X and Y, i.e. NAME to X and SAL to Y:

```
CALL FETCH(C1);
```

A program may wish to write a SEQUEL predicate based on the contents of a program variable—for example, to find the programmers whose department number matches the contents of program variable Z. This facility is also provided by the RDI BIND operator, as follows:

```
CALL BIND('X', ADDR(X));
CALL BIND('Y', ADDR(Y));
CALL BIND('Z', ADDR(Z));
CALL SEQUEL(C1,  'SELECT NAME:X, SAL:Y
                  FROM EMP
                  WHERE JOB = ''PROGRAMMER''
                  AND DNO = Z');
CALL FETCH(C1);
```

Some programs may not know in advance the degree and data types of the tuples to be returned by a query. An example of such a program is one which supports an interactive user by allowing him to type in queries and display the results. This type of program need not specify in its SEQUEL call the variables into which the result is to be delivered. The program may issue a SEQUEL query, followed by the DESCRIBE operator which returns the degree and data types. The program then specifies the destination of the tuples in its FETCH commands. The following example illustrates these techniques:

```
CALL SEQUEL(C1,  'SELECT *
                  FROM EMP
                  WHERE DNO = 50');
```

This statement invokes the optimizer to choose an access path for the given query and associates cursor $C1$ with its active set.

```
CALL DESCRIBE(C1, DEGREE, P);
```

P is a pointer to an array in which the description of the active set of $C1$ is to be returned. The RDI returns the degree of the active set in DEGREE, and the data types and lengths of the tuple components in the elements of the array. If the array (which contains an entry describing its own length) is too short to hold the description of a tuple, the calling program must allocate a larger array and make another call to DESCRIBE.

Having obtained a description of the tuples to be returned, the calling program may proceed to allocate a structure to hold the tuples and may specify the location of this structure in its FETCH command:

```
CALL FETCH(C1, Q);
```

Q is a pointer to an array of pointers which specify where the individual components of the tuple are to be delivered. If this "destination" parameter is present in a FETCH command, it overrides any destination which may have been specified in the SEQUEL command which defined the active set of $C1$.

A special RDI operator OPEN is provided as a shorthand method to associate a cursor with an entire relation. For example, the command

```
CALL OPEN(C1, 'EMP');
```

is exactly equivalent to

```
CALL SEQUEL(C1, 'SELECT * FROM EMP');
```

The use of OPEN is slightly preferable to the use of SEQUEL to open a cursor on a relation, since OPEN avoids the use of the SEQUEL parser.

A program may have many cursors active at the same time. Each cursor remains active until an RDI operator CLOSE or KEEP is issued on it. CLOSE simply deactivates a cursor. KEEP causes the tuples identified by a cursor to be copied to form a new permanent relation in the database, having some specified relation name and field names.

The RDI operator FETCH_HOLD is included for the support of interfaces which provide for explicit locking. FETCH_HOLD operates in exactly the same

way as FETCH except that it also acquires a "hold" on the tuple returned, which prevents other users from updating or deleting it until it is explicitly released or until the holding transaction has ended. A tuple may be released by the RELEASE operator, which takes as a parameter a cursor positioned on the tuple to be released. If no cursor is furnished, the RELEASE operator releases all tuples currently held by the user.

Query Facilities

In this section we describe only the most significant changes made to the SEQUEL query facilities since their original publication [5]. The changes correct certain deficiencies in the original syntax and facilitate the interfacing of SEQUEL with a host programming language. One important change deals with the handling of block labels. The following example, illustrating the original version of SEQUEL, is taken from [5]. (For simplicity, "CALL SEQUEL(. . .)" has been deleted from the next several examples.)

Example 1(a). List names of employees who earn more than their managers.

```
B1: SELECT NAME
    FROM   EMP
    WHERE SAL )
              SELECT SAL
              FROM   EMP
              WHERE EMPNO = B1.MGR
```

Experience has shown that this block label notation has three disadvantages:

(1) It is not possible to select quantities from the inner block, such as: "For all employees who earn more than their manager, list the employee's name and his manager's name."

(2) Since the query is asymmetrically expressed, the optimizer is biased toward making an outer loop for the first block and an inner loop for the second block. Since this may not be the optimum method for interpreting the query, the optimization process is made difficult.

(3) Human factors studies have shown that the block label notation is hard for nonprogrammers to learn [24, 25].

Because of these disadvantages, the block label notation has been replaced by the following more symmetrical notation, which allows several tables to be listed in the FROM clause and optionally referred to by variable names.

Example 1(b). For all employees who earn more than their managers, list the employee's name and his manager's name.

```
SELECT X.NAME, Y.NAME
FROM   EMP X, EMP Y
WHERE X.MGR = Y.EMPNO
AND    X.SAL > Y.SAL
```

Example 1(b) illustrates the SEQUEL notation for the JOIN operator of the relational algebra. The tables to be joined are listed in the FROM clause. A variable name may optionally be associated with each table listed in the FROM clause (e.g. X and Y above). The criterion for joining rows is given in the WHERE

clause (in this case, X.MGR = Y.EMPNO). Field names appearing in the query may stand alone (if unambiguous) or may be qualified by a table name (e.g. EMP.SAL) or by a variable (e.g. X.SAL).

In the earlier report [5], the WHERE clause is used for two purposes: it serves both to qualify individual tuples (e.g. "List the employees who are clerks") and to qualify groups of tuples (e.g. "List the departments having more than ten employees"). This ambiguity is now eliminated by moving group qualifying predicates to a separate HAVING clause. Queries are processed in the following order:

(1) Tuples are selected by the WHERE clause;
(2) Groups are formed by the GROUP BY clause;
(3) Groups are selected which satisfy the HAVING clause, as shown in the example below.

Example 2. List the DNOs of departments having more than ten clerks.

```
SELECT DNO
FROM   EMP
WHERE JOB = 'CLERK'
GROUP BY DNO
HAVING COUNT(*) > 10
```

Two more query features have been added to the ones described in [5]. The first allows the user to specify a value ordering for his query result.

Example 3 (Ordering). List all the employees in Dept. 50, ordered by their salaries.

```
SELECT *
FROM   EMP
WHERE DNO = 50
ORDER BY SAL
```

The other new feature, which is useful primarily to host language users of the RDI, allows a query to qualify tuples by comparing them with the current tuple of some active cursor:

Example 4 (Cursor reference). Find all the employees in the department indicated by cursor $C5$.

```
SELECT *
FROM   EMP
WHERE DNO = DNO OF CURSOR C5 ON DEPT
```

The evaluation of this reference to the content of cursor $C5$ occurs when the query is executed (by a SEQUEL call). Thereafter, moving the cursor $C5$ does not affect the set of tuples defined by the query. The optional phrase "ON DEPT" indicates to the optimizer that it can expect the cursor $C5$ to be positioned on a tuple of the DEPT table. This information may be useful in selecting an access path for the query.

Since elimination of duplicates from a query result is an expensive process and is not always necessary, the RDS does not eliminate duplicates unless explicitly requested to do so. For example, "SELECT DNO, JOB FROM EMP" may return duplicate DNO, JOB pairs, but "SELECT UNIQUE DNO, JOB FROM EMP" will return only unique pairs. Similarly, "SELECT AVG(SAL) FROM EMP" al-

lows duplicate salary values to participate in the average, while "SELECT COUNT (UNIQUE JOB) FROM EMP" returns the count only of different job types in the EMP relation.

Data Manipulation Facilities

The RDI facilities for insertion, deletion, and update of tuples are also provided via the SEQUEL data sublanguage. SEQUEL can be used to manipulate either one tuple at a time or a set of tuples with a single command. The current tuple of a particular cursor may be selected for some operation by means of the special predicate CURRENT TUPLE OF CURSOR. The values of a tuple may be set equal to constants, or to new values computed from their old values, or to the contents of a program variable suitably identified by a BIND command. These facilities will be illustrated by a series of examples. Since no result is returned to the calling program in these examples, no cursor name is included in the calls to SEQUEL.

Example 5 (Set oriented update). Give a 10 percent raise to all employees in Dept. 50.

```
CALL SEQUEL('UPDATE EMP
             SET SAL = SAL × 1.1
             WHERE DNO = 50');
```

Example 6 (Individual update).

```
CALL BIND('PVSAL', ADDR(PVSAL));
CALL SEQUEL('UPDATE EMP
             SET SAL = PVSAL
             WHERE CURRENT TUPLE OF CURSOR C3');
```

Example 7 (Individual insertion). This example inserts a new employee tuple into EMP. The new tuple is constructed partly from constants and partly from the contents of program variables.

```
CALL BIND('PVEMPNO', ADDR(PVEMPNO));
CALL BIND('PVNAME', ADDR(PVNAME));
CALL BIND('PVMGR' ADDR(PVMGR));
CALL SEQUEL('INSERT INTO EMP:
             ⟨PVEMPNO, PVNAME, 50, ''TRAINEE'', 8500, PVMGR⟩');
```

An insertion statement in SEQUEL may provide only some of the values for the new tuple, specifying the names of the fields which are provided. Fields which are not provided are set to the null value. The physical position of the new tuple in storage is influenced by the "clustering" specification made on associated RSS access paths (see below).

Example 8 (Set oriented deletion). Delete all employees who work for departments in Evanston.

```
CALL SEQUEL('DELETE EMP
             WHERE DNO =
                   SELECT DNO
                   FROM   DEPT
                   WHERE  LOC = ''EVANSTON'' ');
```

The SEQUEL assignment statement allows the result of a query to be copied into a new permanent or temporary relation in the database. This has the same effect as a query followed by the RDI operator KEEP.

Example 9 (Assignment). Create a new table UNDERPAID consisting of names and salaries of programmers who earn less than $10,000.

```
CALL SEQUEL('UNDERPAID(NAME, SAL) ←
        SELECT  NAME, SAL
        FROM    EMP
        WHERE   JOB = ''PROGRAMMER''
        AND     SAL < 10,000');
```

The new table UNDERPAID represents a snapshot taken from EMP at the moment the assignment was executed. UNDERPAID then becomes an independent relation and does not reflect any later changes to EMP.

Data Definition Facilities

System R takes a unified approach to data manipulation, definition, and control. Like queries and set oriented updates, the data definition facilities are invoked by means of the RDI operator SEQUEL. Many of these facilities have been described in [4] and [15].

The SEQUEL statement CREATE TABLE is used to create a new base (i.e. physically stored) relation. For each field of the new relation, the field name and data type are specified.[1] If desired, it may be specified at creation time that null values are not permitted in one or more fields of the new relation. A query executed on the relation will deliver its results in system determined order (which depends upon the access path which the optimizer has chosen), unless the query has an ORDER BY clause. When a base relation is no longer useful, it may be deleted by issuing a DROP TABLE statement.

System R currently relies on the user to specify not only the base tables to be stored but also the RSS access paths to be maintained on them. (Database design facilities to automate and adapt some of these decisions are also being investigated.) Access paths include images and binary links,[2] described in Section 3. They may be specified by means of the SEQUEL verbs CREATE and DROP. Briefly, images are value orderings maintained on base relations by the RSS, using multilevel index structures. The index structures associate a value with one or more Tuple Identifiers (*TID*s). A *TID* is an internal address which allows rapid access to a tuple, as discussed in Section 3. Images provide associative and sequential access on one or more fields which are called the *sort fields* of the image. An image may be declared to be UNIQUE, which forces each combination of sort field values to be unique in the relation. At most one image per relation may have the *clustering* property, which causes tuples whose sort field values are close to be physically stored near each other.

Binary links are access paths in the RSS which link tuples of one relation to

[1] The data types of INTEGER, SMALL INTEGER, DECIMAL, FLOAT, and CHARACTER (both fixed and varying length) are supported.

[2] Unary links, described in Section 3, are used for internal system purposes only, and are not exposed at the RDI.

related tuples of another relation through pointer chains. In System R, binary links are always employed in a value dependent manner: the user specifies that each tuple of Relation 1 is to be linked to the tuples in Relation 2 which have matching values in some field(s), and that the tuples on the link are to be ordered in some value dependent way. For example, a user may specify a link from DEPT to EMP by matching DNO, and that EMP tuples on the link are to be ordered by JOB and SAL. This link is maintained automatically by the system. By declaring a link frem DEPT to EMP on matching DNO, the user implicitly declares this to be a one-to-many relationship (i.e. DNO is a key of DEPT). Any attempts to define links or to insert or update tuples in violation of this rule will be refused. Like an image, a link may be declared to have the *clustering* property, which causes each tuple to be physically stored near its neighbor in the link.

It should be clearly noted that none of the access paths (images and binary links) contain any logical information other than that derivable from the data values themselves. This is in accord with the relational data model, which represents all information as data values. The RDI user has no explicit control over the placement of tuples in images and links (unlike the "manual sets" of the DBTG proposal [6]). Furthermore, the RDI user may not explicitly use an image or link for access to data; all choices of access path are made automatically by the optimizer.

The query power of SEQUEL may be used to define a view as a relation derived from one or more other relations. This view may then be used in the same ways as a base table: queries may be written against it, other views may be defined on it, and in certain circumstances described below, it may be updated. Any SEQUEL query may be used as a view definition by means of a DEFINE VIEW statement. Views are dynamic windows on the database, in that updates made to base tables immediately become visible via the views defined on these base tables. Where updates to views are supported, they are implemented in terms of updates to the underlying base tables. The SEQUEL statement which defines a view is recorded in a system maintained catalog where it may be examined by authorized users. When an authorized user issues a DROP VIEW statement, the indicated view and all other views defined in terms of it disappear from the system for this user and all other users.

If a modification is issued against a view, it can be supported only if the tuples of the view are associated one-to-one with tuples of an underlying base relation. In general, this means that the view must involve a single base relation and contain a key of that relation; otherwise, the modification statement is rejected. If the view satisfies the one-to-one rule, the WHERE clause of the SEQUEL modification statement is merged into the view definition; the result is optimized and the indicated update is made on the relevant tuples of the base relation.

Two final SEQUEL commands complete the discussion of the data definition facility. The first is KEEP TABLE, which causes a temporary table (created, for example, by assignment) to become permanent. (Temporary tables are destroyed when the user who created them logs off.) The second command is EXPAND TABLE, which adds a new field to an existing table. All views, images, and links defined on the original table are retained. All existing tuples are interpreted as having null values in the expanded fields until they are explicitly updated.

Data Control Facilities

Data control facilities at the RDI have four aspects: transactions, authorization, integrity assertions, and triggers.

A transaction is a series of RDI calls which the user wishes to be processed as an atomic act. The meaning of "atomic" depends on the level of consistency specified by the user, and is explained in Section 3. The highest level of consistency, Level 3, requires that a user's transactions appear to be serialized with the transactions of other concurrent users. The user controls transactions by the RDI operators BEGIN_TRANS and END_TRANS. The user may specify save points within a transaction by the RDI operator SAVE. As long as a transaction is active, the user may back up to the beginning of the transaction or to any internal save point by the operator RESTORE. This operator restores all changes made to the database by the current transaction, as well as the state of all cursors used by this transaction. No cursors may remain active (open) beyond the end of a transaction. The RDI transactions are implemented directly by RSI transactions, so the RDI commands BEGIN_TRANS, END_TRANS, SAVE, and RESTORE are passed through to the RSI, with some RDS bookkeeping to permit the restoration of its internal state.

The System R approach to authorization is described in [15]. System R does not require a particular individual to be the database administrator, but allows each user to create his own data objects by executing the SEQUEL statements CREATE TABLE and DEFINE VIEW. The creator of a new object receives full authorization to perform all operations on the object (subject, of course, to his authorization for the underlying tables, if it is a view). The user may then grant selected capabilities for his object to other users by the SEQUEL statement GRANT. The following capabilities may be independently granted for each table or view: READ, INSERT, DELETE, UPDATE (by fields), DROP, EXPAND, IMAGE specification, LINK specification, and CONTROL (the ability to specify assertions and triggers on the table or view). For each capability which a user possesses for a given table, he may optionally have GRANT authority (the authority to further grant or revoke the capability to/from other users).

System R relies primarily on its view mechanism for read authorization. If it is desired to allow a user to read only tuples of employees in Dept. 50, and not to see their salaries, then this portion of the EMP table can be defined as a view and granted to the user. No special statistical access is distinguished, since the same effect (e.g. ability to read only the average salary of each department) can be achieved by defining a view. To make the view mechanism more useful for authorization purposes, the reserved word USER is always interpreted as the user-id of the current user. Thus the following SEQUEL statement defines a view of all those employees in the same department as the current user:

```
DEFINE VIEW VEMP AS:
        SELECT *
        FROM   EMP
        WHERE DNO =
                SELECT DNO
                FROM   EMP
                WHERE NAME = USER
```

The third important aspect of data control is that of integrity assertions. The System R approach to data integrity is described in [10]. Any SEQUEL predicate may be stated as an assertion about the integrity of data in a base table or view. At the time the assertion is made (by an ASSERT statement in SEQUEL), its truth is checked; if true, the assertion is automatically enforced until it is explicitly dropped by a DROP ASSERTION statement. Any data modification, by any user, which violates an active integrity assertion is rejected. Assertions may apply to individual tuples (e.g. "No employee's salary exceeds $50,000") or to sets of tuples (e.g. "The average salary of each department is less than $20,000"). Assertions may describe permissible *states* of the database (as in the examples above) or permissible *transitions* in the database. For this latter purpose the keywords OLD and NEW are used in SEQUEL to denote data values before and after modification, as in the example below.

Example 10 (Transition assertion). Each employee's salary must be nondecreasing.

```
ASSERT ON UPDATE TO EMP: NEW SAL ≥ OLD SAL
```

Unless otherwise specified, integrity assertions are checked and enforced at the end of each transaction. Transition assertions compare the state before the transaction began with the state after the transaction concluded. If some assertion is not satisfied, the transaction is backed out to its beginning point. This permits complex updates to be done in several steps (several calls to SEQUEL, bracketed by BEGIN_TRANS and END_TRANS), which may cause the database to pass through intermediate states which temporarily violate one or more assertions. However, if an assertion is specified as IMMEDIATE, it cannot be suspended within a transaction, but is enforced after each data modification (each RDI call). In addition, "integrity points" within a transaction may be established by the SEQUEL command ENFORCE INTEGRITY. This command allows a user to guard against having a long transaction completely backed out. In the event of an integrity failure, the transaction is backed out to its most recent integrity point.

The fourth aspect of data control, triggers, is a generalization of the concept of assertions. A trigger causes a prespecified sequence of SEQUEL statements to be executed whenever some triggering event occurs. The triggering event may be retrieval, insertion, deletion, or update of a particular base table or view. For example, suppose that in our example database, the NEMPS field of the DEPT table denotes the number of employees in each department. This value might be kept up to date automatically by the following three triggers (as in assertions, the keywords OLD and NEW denote data values before and after the change which invoked the trigger):

```
DEFINE TRIGGER EMPINS
      ON INSERTION OF EMP:
            (UPDATE  DEPT
             SET     NEMPS = NEMPS + 1
             WHERE   DNO = NEW EMP.DNO)
```

```
DEFINE TRIGGER EMPDEL
      ON DELETION OF EMP:
            (UPDATE DEPT
             SET    NEMPS = NEMPS - 1
             WHERE  DNO = OLD EMP.DNO)
DEFINE TRIGGER EMPUPD
      ON UPDATE OF EMP:
            (UPDATE  DEPT
             SET     NEMPS = NEMPS - 1
             WHERE   DNO = OLD EMP.DNO;
             UPDATE  DEPT
             SET     NEMPS = NEMPS + 1
             WHERE   DNO = NEW EMP.DNO)
```

The RDS automatically maintains a set of catalog relations which describe the other relations, views, images, links, assertions, and triggers known to the system. Each user may access a set of views of the system catalogs which contain information pertinent to him. Access to catalog relations is made in exactly the same way as other relations are accessed (i.e. by SEQUEL queries). Of course, no user is authorized to modify the contents of a catalog directly, but any authorized user may modify a catalog indirectly by actions such as creating a table. In addition, a user may enter comments into his various catalog entries by means of the COMMENT statement (see syntax in Appendix II).

The Optimizer

The objective of the optimizer is to find a low cost means of executing a SEQUEL statement, given the data structures and access paths available. The optimizer attempts to minimize the expected number of pages to be fetched from secondary storage into the RSS buffers during execution of the statement. Only page fetches made under the explicit control of the RSS are considered. If necessary, the RSS buffers will be pinned in real memory to avoid additional paging activity caused by the VM/370 operating system. The cost of CPU instructions is also taken into account by means of an adjustable coefficient, H, which is multiplied by the number of tuple comparison operations to convert to equivalent page accesses. H can be adjusted according to whether the system is compute-bound or disk access-bound.

Since our cost measure for the optimizer is based on disk page accesses, the physical clustering of tuples in the database is of great importance. As mentioned earlier, each relation may have at most one clustering image, which has the property that tuples near each other in the image ordering are stored physically near each other in the database. To see the importance of the clustering property, imagine that we wish to scan over the tuples of a relation in the order of some image, and that the number of RSS buffer pages is much less than the number of pages used to store the relation. If the image is not the clustering image, the locations of the tuples will be independent of each other and in general a page will have to be fetched from disk for each tuple. On the other hand, if the image is the clustering image, each disk page will contain several (usually at least 20) adjacent tuples, and the number of page fetches will be reduced by a corresponding factor.

The optimizer begins by classifying the given SEQUEL statement into one of several statement types, according to the presence of various language features such as join and GROUP BY. Next the optimizer examines the system catalogs to find the set of images and links which are pertinent to the given statement. A rough decision procedure is then executed to find the set of "reasonable" methods of executing the statement. If there is more than one "reasonable" method, an expected cost formula is evaluated for each method and the minimum-cost method is chosen. The parameters of the cost formulas, such as relation cardinality and number of tuples per page, are obtained from the system catalogs.

We illustrate this optimization process by means of two example queries. The first example involves selection of tuples from a single relation, and the second involves joining two relations together according to a matching field. For simplicity we consider only methods based on images and relation scans. (A relation scan in the RSS accesses each of the pages in a data segment in turn (see Section 3), and selects those tuples belonging to the given relation.) Consideration of links involves a straightforward extension of the techniques we will describe.

Example 11 will be used to describe the decision process for a query involving a single relation:

Example 11. List the names and salaries of programmers who earn more than $10,000.

```
SELECT NAME, SAL
FROM   EMP
WHERE JOB = 'PROGRAMMER'
AND    SAL > 10,000
```

In planning the execution of this example, the optimizer must choose whether to access the EMP relation via an image (on JOB, SAL or some other field) or via a relation scan. The following parameters, available in the system catalogs, are taken into account:

- R relation cardinality (number of tuples in the relation)
- D number of data pages occupied by the relation
- T average number of tuples per data page (equal to R/D)
- I image cardinality (number of distinct sort field values in a given image)
- H coefficient of CPU cost ($1/H$ is the number of tuple comparisons which are considered equivalent in cost to one disk page access).

An image is said to "match" a predicate if the sort field of the image is the field which is tested by the predicate. For example, an image on the EMP relation ordered by JOB (which we will refer to as an "image on EMP.JOB") would match the predicate JOB = 'PROGRAMMER' in Example 11. In order for an image to match a predicate, the predicate must be a simple comparison of a field with a value. More complicated predicates, such as EMP.DNO = DEPT.DNO, cannot be matched by an image.

In the case of a simple query on a single relation, such as Example 11, the optimizer compares the available images with the predicates of the query, in order to determine which of the following eight methods are available:

Method 1: Use a clustering image which matches a predicate whose comparison-

operator is '='. The expected cost to retrieve all result tuples is $R/(T \times I)$ page accesses (R/I tuples divided by T tuples per page).

Method 2: Use a clustering image which matches a predicate whose comparison operator is not '='. Assuming half the tuples in the relation satisfy the predicate, the expected cost is $R/(2 \times T)$.

Method 3: Use a nonclustering image which matches a predicate whose comparison operator is '='. Since each tuple requires a page access, the expected cost is R/I.

Method 4: Use a nonclustering image which matches a predicate whose comparison-operator is not '='. Expected cost to retrieve all result tuples is $R/2$.

Method 5: Use a clustering image which does not match any predicate. Scan the image and test each tuple against all predicates. Expected cost is $(R/T) + H \times R \times N$, where N is the number of predicates in the query.

Method 6: Use a nonclustering image which does not match any predicate. Expected cost is $R + H \times R \times N$.

Method 7: Use a relation scan where this relation is the only one in its segment. Test each tuple against all predicates. Expected cost is $(R/T) + H \times R \times N$.

Method 8: Use a relation scan where there are other relations sharing the segment. Cost is unknown, but greater than $(R/T) + H \times R \times N$, because some pages may be fetched which contain no tuples from the pertinent relation.

The optimizer chooses a method from this set according to the following rules:

1. If Method 1 is available, it is chosen.
2. If exactly one among Methods 2, 3, 5, and 7 is available, it is chosen. If more than one method is available in this class, the expected cost formulas for these methods are evaluated and the method of minimum cost is chosen.
3. If none of the above methods are available, the optimizer chooses Method 4, if available; else Method 6, if available; else Method 8. (Note: Either Method 7 or Method 8 is always available for any relation.)

As a second example of optimization, we consider the following query, which involves a join of two relations:

Example 12. List the names, salaries, and department names of programmers located in Evanston.

```
SELECT NAME, SAL, DNAME
FROM   EMP, DEPT
WHERE EMP.JOB = 'PROGRAMMER'
AND    DEPT.LOC = 'EVANSTON'
AND    EMP.DNO = DEPT.DNO
```

Example 12 is an instance of a join query type, the most general form of which involves restriction, projection, and join. The general query has the form:

> Apply a given restriction to a relation R, yielding R1, and apply a possibly different restriction to a relation S, yielding S1. Join R1 and S1 to form a relation T, and project some fields from T.

To illustrate the optimization of join-type queries, we will consider four possible methods for evaluating Example 12:

Method 1 (use images on join fields): Perform a simultaneous scan of the image

on DEPT.DNO and the image on EMP.DNO. Advance the DEPT scan to obtain the next DEPT where LOC is 'EVANSTON'. Advance the EMP scan and fetch all the EMP tuples whose DNO matches the current DEPT and whose JOB is 'PROGRAMMER'. For each such matching pair of DEPT, EMP tuples, place the NAME, SAL, and DNAME fields into the output. Repeat until the image scans are completed.

Method 2 (sort both relations): Scan EMP and DEPT using their respective clustering images and create two files W1 and W2. W1 contains the NAME, SAL, and DNO fields of tuples from EMP which have JOB = 'PROGRAMMER'. W2 contains the DNO and DNAME fields of tuples from DEPT whose location is 'EVANSTON'. Sort W1 and W2 on DNO. (This process may involve repeated passes over W1 and W2 if they are too large to fit the available main memory buffers.) The resulting sorted files are scanned simultaneously and the join is performed.

Method 3 (multiple passes): DEPT is scanned via its clustering image, and the DNO and DNAME fields (a subtuple) of those DEPT tuples which have LOC = 'EVANSTON' are inserted into a main memory data structure called W. If space in main memory is available to insert a subtuple (say S), it is inserted. If there is no space and if S.DNO is less than the current highest DNO value in W, the subtuple with the highest DNO in W is deleted and S inserted. If there is no room for S and the DNO in S is greater than the highest DNO in W, S is discarded. After completing the scan of DEPT, EMP is scanned via its clustering image and a tuple E of EMP is obtained. If E.JOB = 'PROGRAMMER', then W is checked for the presence of the E.DNO. If present, E is joined to the appropriate subtuple in W. This process is continued until all tuples of EMP have been examined. If any DEPT subtuples were discarded, another scan of DEPT is made to form a new W consisting of subtuples with DNO value greater than the current highest. EMP is scanned again and the process repeated.

Method 4 (*TID* algorithm): Using the image on EMP.JOB, obtain the *TID*s of tuples from EMP which satisfy the restriction JOB = 'PROGRAMMER'. Sort them and store the *TID*s in a file W1. Do the same with DEPT, using the image on DEPT.LOC and testing for LOC = 'EVANSTON', yielding a *TID* file W2. Perform a simultaneous scan over the images on DEPT.DNO and EMP.DNO, finding the *TID* pairs of tuples whose DNO values match. Check each pair (*TID*1, *TID*2) to see if *TID*1 is present in W1 and *TID*2 is in W2. If they are, the tuples are fetched and joined and the NAME, SAL, and DNAME fields placed into the output.

These methods should be considered as illustrative of the techniques considered by the optimizer. The optimizer will draw from a larger set of methods, including methods which use links to carry out the join.

A method cannot be applied unless the appropriate access paths are available. For example, Method 4 is applicable only if there are images on EMP.DNO and EMP.JOB, as well as on DEPT.DNO and DEPT.LOC. In addition, the performance of a method depends strongly on the clustering of the relations with respect to the access paths. We will consider how the optimizer would choose among these four methods in four hypothetical situations. These choices are made on the basis of cost formulas which will be detailed in a later paper.

Situation 1: There are clustering images on both EMP.DNO and DEPT.DNO, but no images on EMP.JOB or DEPT.LOC. In this situation, Method 1 is always chosen.

Situation 2: There are unclustered images on EMP.DNO and DEPT.DNO, but no images on EMP.JOB or DEPT.LOC. In this case, Method 3 is chosen if the entire working file W fits into the main memory buffer at once; otherwise Method 2 is chosen. It is interesting to note that the unclustered images on DNO are never used in this situation.

Situation 3 :There are clustering images on EMP.DNO and DEPT.DNO, and unclustered images on EMP.JOB and DEPT.LOC. In this situation, Method 4 is always chosen.

Situation 4: There are unclustered images on EMP.DNO, EMP.JOB, DEPT.DNO, and DEPT.LOC. In this situation, Method 3 is chosen if the entire working file W fits into the main memory buffer. Otherwise, Method 2 is chosen if more than one tuple per disk page is expected to satisfy the restriction predicates. In the remaining cases, where the restriction predicates are very selective, Method 4 should be used.

After analyzing any SEQUEL statement, the optimizer produces an Optimized Package (OP) containing the parse tree and a plan for executing the statement. If the statement is a query, the OP is used to materialize tuples as they are called for by the FETCH command (query results are materialized incrementally whenever possible). If the statement is a view definition, the OP is stored in the form of a Pre-Optimized Package (POP) which can be fetched and utilized whenever an access is made via the specified view. If any change is made to the structure of a base table or to the access paths (images and links) maintained on it, the POPs of all views defined on that base table are invalidated, and each view must be reoptimized from its defining SEQUEL code to form a new POP.

When a view is accessed via the RDI operators OPEN and FETCH, the POP for the view can be used directly to materialize the tuples of the view. Often, however, a query or another view definition will be written in terms of an existing view. If the query or view definition is simple (e.g. a projection or restriction), it can sometimes be *composed* with the existing view (i.e. their parse trees can be merged and optimized together to form a new OP for the new query or view). In more complex cases the new statement cannot be composed with the existing view definition. In these cases the POP for the existing view is treated as a formula for materializing tuples. A new OP is formed for the new statement which treats the existing view as a table from which tuples can be fetched in only one way: by interpreting the existing POP. Of course, if views are cascaded on other views in several levels, there may be several levels of POPs in existence, each level making reference to the next.

Modifying Cursors

A number of issues are raised by the use of the insertion, deletion, and update facilities of System R. When a modification is made to one of the tuples in the active set of a cursor, the modification may change the ordinal position of the tuple or even disqualify it entirely from the active set. It should be noted here that a

user operating at Level 3 consistency is automatically protected against having his cursors affected by the modifications of other users. However, even in Level 3 consistency, a user may make a modification which affects one of his own active cursors.

If the cursor in question is open on a base relation, the case is simple: the modification is done and immediately becomes visible via the cursor. Let us consider a case in which the cursor is not on a base relation, but rather on the result of a SEQUEL query. Suppose the following query has been executed:

```
SELECT *
FROM   EMP
WHERE DNO = 50
ORDER BY SAL
```

If the system has no image ordered on SAL, it may execute this query by finding the employees where DNO = 50 and sorting them by SAL to create an ordered list of answer tuples. Along with this list, the system will keep a list of the base relations from which the list was derived (in this case, only EMP). The effect resembles that of performing a DBTG KEEP verb [6] on the underlying base relations: if any tuple in an underlying relation is modified, the answer list is marked "potentially invalid." Now any fetch from this list will return a warning code since the tuple returned may not be up to date. If the calling program wishes to guarantee accuracy of its results, it must close its cursor and reevaluate the query when this warning code is received.

Simulation of Nonrelational Data Models

The RDI is designed in such a way that programs can be written on top of it to simulate "navigation oriented" database interfaces. These interfaces are often characterized by collections of records connected in a hierarchic [17] or network [6] structure, and by the concept of establishing one or more "current positions" within the structure (e.g. the currency indicators of DBTG). In general our strategy will be to represent each record type as a relation and to represent information about ordering and connections between records in the form of explicit fields in the corresponding relations. In this way all information inserted into the database via the "navigational" interface (including information about orderings and connections) is available to other users who may be using the underlying relations directly. One or more "current positions" within the database may then be simulated by means of one or more RDI cursors.

We will illustrate this simulation process by means of an example. Suppose we wish to simulate the database structure shown in Figure 3, and wish to maintain a "current position" in the structure. The hierarchical connections from DEPT to

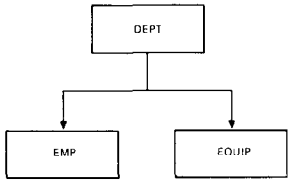

FIG. 3. Example of a hierarchic data structure

EMP and from DEPT to EQUIP may be unnamed in a hierarchic system such as IMS [17], or they may represent named set types in a network oriented system such as DBTG [6].

At database definition time, a relation is created to simulate each record type. The DEPT relation must have a sequence-number field to represent the ordering of the DEPT records. The EMP and EQUIP relations must have, in addition to a sequence-number field, one or more fields which uniquely identify their "parent" or "owner" records (let us assume the key of DEPT is DNO). If a record had several "owners" in different set types, several "owner's key" fields would have to appear in the corresponding relation.

Also at database definition time, a view definition is entered into the system which will represent the "currently visible" tuples of each relation at any point in time. The view definitions for our example are given below:

```
DEFINE VIEW VDEPT AS
       SELECT *
       FROM   DEPT
       ORDER BY ⟨sequence field⟩
DEFINE VIEW VEMP AS
       SELECT *
       FROM   EMP
       WHERE DNO = DNO OF CURSOR C1 ON DEPT
       ORDER BY ⟨sequence field⟩
DEFINE VIEW VEQUIP AS
       SELECT *
       FROM   EQUIP
       WHERE DNO = DNO OF CURSOR C1 ON DEPT
       ORDER BY ⟨sequence field⟩
```

The definitions of VEMP and VEQUIP call for tuples of EMP and EQUIP which have the same DNO as cursor $C1$; furthermore they promise that, when these views are used, cursor $C1$ will be active on the DEPT relation. These view definitions are parsed and optimized, and stored in the form of POPs. During this optimization process, any direct physical support for the hierarchy (such as a link from DEPT to EMP by matching DNO) will be discovered.

At run time, when a position is to be established on a DEPT record, the cursor $C1$ is opened on the view VDEPT. If the "current position" then moves downward to an EMP record, the view VEMP is opened. The exact subset of EMP tuples made available by this view opening depends on the location of the cursor $C1$ in the "parent" relation. If the "current position" moves upward again to DEPT, the view VEMP is closed, to be reopened later as needed. Any insertion, deletion, or update operations issued against the hierarchy are simulated by SEQUEL INSERT, DELETE, and UPDATE operations on the corresponding relations, with appropriate sequence-number and parent-key values generated, if necessary, by the simulator program. At the end of the transaction, all cursors are closed.

Following this general plan, it is expected that hierarchic oriented or network oriented interfaces can be simulated on top of the RDI. It should be particularly noted that no parsing or optimization is done in response to a command to move the "current position"; the system merely employs the POP for the view which was

optimized at database definition time. For any connections which are given direct physical support in the form of a binary link, the optimizer will take advantage of the link to provide good performance. The system is also capable of simulating connections which have no direct physical support, since the optimizer will automatically find an appropriate access path.

3. THE RELATIONAL STORAGE SYSTEM

This section is concerned with the Relational Storage System or RSS, the database management subsystem which provides underlying support for System R. The RSS supports the RSI which provides simple, tuple-at-a-time operators on base relations. Operators are also supported for data recovery, transaction management, and data definition. (A list of all RSI operators can be found in Appendix III.) Calls to the RSI require explicit use of data areas called segments and access paths called images and links, along with the use of RSS-generated, numeric identifiers for data segments, relations, access paths, and tuples. The RDS handles the selection of efficient access paths to optimize its operations, and maps symbolic relation names to their internal RSS identifiers.

In order to facilitate gradual database integration and retuning of access paths, the RSI has been designed so that new stored relations or new indexes can 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 relations, or add or delete pointer chain paths across existing relations. This facility, coupled with the ability to retrieve any subset of fields in a tuple, provides a degree of data independence at a low level of the system, since existing programs which execute RSI operations on tuples will be unaffected by the addition of new fields.

As a point of comparison, the RSS has many functions which can be found in other systems, both relational and nonrelational, 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, multipoint recovery for in-process transactions, a novel and efficient technique for system checkpoint and restart, multiple levels of isolation from the actions of other concurrent users, and automatic locking at the level of segments, relations, and single tuples. The next several subsections 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 tuples of any relation must reside within a single segment chosen by the RDS. However, a given segment may contain several relations. A special segment is dedicated to the storage of transaction logs for backing out 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 recovery of the segment's contents to a previous state. Another segment type is intended for low overhead storage of temporary relations, and has no provision for either concurrent access or segment recovery. A maximum length is associated with each segment; it 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 requested.

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 intact.

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 operation 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. How-

ever, the effect of restoring a segment of public data segment may be to undo changes made by several transactions, since each of them may have modified data since the segment was last saved. An entirely different mechanism is therefore used to back out only those changes made by a single transaction, and is explained below.

Note that our recovery scheme depends on the highly stylized management of two page maps per segment, and on our 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.

Relations

The main data object of the RSS is the n-ary *relation*, which consists of a time-varying number of tuples, each containing n fields. A new relation can be defined at any time within any segment chosen by the RDS. An existing relation and its associated access path structures can be dropped at any time, with all storage space made reusable. Even after a relation is defined and loaded, new fields may be added on the right, without a database reload and without immediate modification to existing tuples.

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 relation, values for those fields in each existing tuple are treated as undefined until they are explicitly updated.

Operators are available to INSERT and DELETE single tuples, and to FETCH and UPDATE any combination of fields in a tuple. One can also fetch a sequence of tuples along an access path through the use of an RSS cursor or *scan*. Each scan is created by the RSS for fetching tuples on a particular access path through execution of the OPEN_SCAN operator. The tuples 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 tuples through use of an image, an RDS determined ordering of tuples through use of a link (see below for discussions of images and links), and an RSS determined ordering of tuples in a relation. For all of these access paths the RDS may attach a search argument 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 '$=$', '$\neq$', '$<$', '$>$', '$\leq$', or '$\geq$'.

Associated with every tuple of a relation is a *tuple identifier* or *TID*. Each tuple identifier is generated by the RSS, and is available to the RDS as a concise and efficient means of addressing tuples. *TID*s are also used within the RSS to refer to tuples 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 tuple deletions and are reassigned during database reorganization.

The RSS stores and accesses tuples within relations, and maintains pointer chains to implement the links described below. Each tuple 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 tuple for use within the RSS. The prefix contains such information as the relation 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 tuples. Tuples 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 tuples from more than one relation, so that extra page accesses can be avoided when tuples from different relations are accessed together. When a scan is executed on a relation (rather than an image or link), an internal scan is generated on all nonempty data pages within the segment containing that relation. Each such data page is touched once, and the prefix of each tuple within the page is checked to see if it belongs to the relation.

The implementation of tuple identifier access is a hybrid scheme, similar to one used in such systems as IDS [11] and RM [20], which combines the speed of a byte address pointer with the flexibility of indirection. Each tuple 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 the tuple in that page. This technique allows efficient utilization of space within data pages, since space can be compacted and tuples 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 tuples. In the rare case when a tuple is updated to a longer total value and insufficient space is available on its page, an overflow scheme is provided to move the tuple to another page. In this case the *TID* points to a tagged overflow record which is used to reference the other page. If the tuple overflows again, the original overflow record is modified to point to the newest location. Thus, a tuple 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 tuple 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 tuples of a given relation with respect to some criterion such as a value ordering on one or more fields. Another use would be to cluster tuples of one relation near particular tuples of another relation, 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.

Images

An *image* in the RSS is a logical reordering of an n-ary relation with respect to values in one or more sort fields. Images combined with scans provide the ability to scan relations along a value ordering, for low level support of simple views. More importantly, an image provides associative access capability. The RDS can rapidly fetch a tuple from an image by keying on the sort field values. The RDS can also

open a scan at a particular point in the image, and retrieve a sequence of tuples or subtuples with a given range of sort values. Since the image contains all the tuples and all the fields in a relation, the RDS can employ a disjunctive normal form search argument during scanning to further restrict the set of tuples which is returned. This facility is especially useful for situations where SEQUEL search predicates involve several fields of a relation, and at least one of them has image support.

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

The RSS maintains each image through the use of a multipage index structure. An internal interface is used for associative or sequential access along an image, and also to delete or insert index entries when tuples are deleted, inserted, or updated. The parameters passed across this interface include the sort field values along with the *TID* of the given tuple. In order to handle variable length, multifield indexes efficiently, a special encoding scheme is employed on the field values so that the resulting concatenation can be compared against others for ordering and search. This encoding eliminates the need for costly padding of each field and slow field-by-field comparison.

Each index is composed of one or more pages within the segment containing the relation. 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 [3] and of Key Sequenced Data Sets in IBM's VSAM access method [23]. Each page is a node within the hierarchy and contains an ordered sequence of index entries. For nonleaf 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 nonleaf 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 tuples 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.

Links

A *link* in the RSS is an access path which is used to connect tuples in one or two relations. The RDS determines which tuples will be on a link and determines their relative position, through explicit CONNECT and DISCONNECT operations. The RSS maintains internal pointers so that newly connected tuples are linked to previous and next twins, and so that previous and next twins are linked to each other when a tuple is disconnected. A link can be scanned using a sequence of OPEN SCAN and NEXT operations, with the optional search arguments described above.

A unary link involves a single relation and provides a partially defined ordering of tuples. Unary links can be used to maintain tuple ordering specifications which are not supported by the RSS (i.e. not value ordered). Another use is to provide an efficient access path through all tuples of a relation without the time overhead of an internal page scan.

The more important access path is a binary link, which provides a path from single tuples (parents) in one relation to sequences of tuples (children) in another relation. The RDS determines which tuples 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 tuple in the parent relation may have no children, and a tuple in the child relation may have no parent. Also, tuples in a relation may be parents and/or children in an arbitrary number of different links. The only restriction is that a given tuple can appear only once within a given link. Binary links are similar to the notion of an owner coupled set with manual membership found in the DBTG specifications for a network model of data [6].

The main use of binary links in System R is to connect child tuples to a parent based on value matches in one or more fields. With such a structure the RDS can access tuples in one relation, say the Employee relation, based on matching the Department Number field in a tuple of the Department relation. This function is especially important for supporting relational join operations, and also for supporting navigational processing through hierarchical and network models of data. The link provides direct access to the correct Employee tuples from the Department tuple (and vice versa), while use of an image may involve access to several pages in the index. A striking advantage is gained over images when the child tuples have been clustered on the same page as the parent, so that no extra pages are touched using the link, while three or more pages may be touched in a large index.

Another important feature of links is to provide reasonably fast associative access to a relation without the use of an extra index. In the above example, if the Department relation has an image on Department Number, then the RDS can gain associative access to Employee tuples for a given value of Department Number by using the Department relation image and the binary link—even if the Department tuple is not being referenced by the end user.

Links are maintained in the RSS by storing *TID*s in the prefix of tuples. New links can be defined at any time. When a new link is defined for a relation, a portion of the prefix is assigned to hold the required entries. This operation does not require access to any of the existing tuples, since new prefix space for an existing tuple is formatted only when the tuple 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 tuple in the corresponding relation(s) is accessed by the RSS, in order to invalidate the existing prefix entries and make the space available for subsequent link definitions.

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. In general, an RSS transaction consists of those calls generated by the RDS to execute all RDI operators in a single System R transaction, including the calls required to perform such RDS internal functions as authorization, catalog access, and integrity checking. An RSS transaction is marked by the START_TRANS and

END_TRANS operators. Various resources are assigned to transactions by the RSS, using the locking techniques described below. Also, a transaction recovery scheme is provided which allows a transaction to be incrementally backed out to any intermediate save point. This multipoint recovery function is important in applications involving relatively long transactions when backup is required because of errors detected by the user or RDS, because of deadlock detected by the RSS, or because of long periods of inactivity or system congestion detected by the Monitor.

A transaction save point is marked using the SAVE_TRANS operator, which returns a save point number for subsequent reference. In general, a save point may be generated by any one of the layers above the RSS. An RDI user may mark a save point at a convenient place in his transaction in order to handle backout and retry. The RDS may mark a save point for each new set oriented SEQUEL expression, so that the sequence of RSI calls needed to support the expression can be backed out for automatic retry if any of the RSI calls fails to complete.

Transaction recovery occurs when the RDS or Monitor issues the RESTORE_TRANS operator, which has a save point number as its input parameter, 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 since the given save point. Those changes include all the tuple and image 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 relations, images, and links. In order to aid the RDS in continuing the transaction, all scan positions on recoverable data are automatically reset to the tuples they were pointing to at the time of the save. Finally, all locks on recoverable data which have been obtained since the given save point 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 code and object identification. Modifications to index structures are not logged, since their values can be determined from data values and index catalog information.

At each transaction save point, special entries are stored containing the state of all scans in use by the transaction, and the identity of the most recently acquired lock. 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 back to the recorded save point, and also to restore the scans and release locks acquired after the save point.

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. Entries are also archived to tape to support audits and database reconstruction after system failure.

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 relations and tuples 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 thus 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 tuples with each tuple accessed through its tuple 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 relation or a different tuple in the same relation, a problem could occur at the physical level if one transaction follows a pointer to a tuple on some page while the other transaction updates a second tuple on the same page and causes a data compaction routine to reassign tuple locations.

One basic decision in establishing 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, relations, *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 tuples. 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 applications.

Another basic decision in formulating System R was to automate all of the locking functions, both logical and physical, so that users can access shared data and delegate some or all lock protocols to the system. 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 relations.

In order to provide reasonable performance for a wide spectrum of user requirements, the RSS supports multiple levels of consistency which control the isolation of a user from the actions of other concurrent users (see also [13]). 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 violate integrity constraints, and 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. 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 still require use of 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 tuple identifier, but even to sequences of items and to items accessed associatively. For example, if the RDS employs an image on the Employee relation, 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 SEQUEL predicate and obtained by any search strategy, against insertions into or deletions from the set. Similarly, if the RDS employs an image to access the unique tuple where Name = 'Smith', and no such tuple 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 tuples, since other transactions are logically serialized with the given one. As an example of this last point, consider a situation where two or more related data items are periodically updated, such as the mean and variance of a sequence of temperature measurements. With Level 3 consistency, a reader is assured of reading a consistent pair—rather than, say, a new variance and an old mean. 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.

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 tuples, such as when they have been inserted or updated. Simi-

larly, 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 in handling the case of 'Smith' described above. In both of these cases the RSS must also acquire physical locks on one or more pages, which are held at least during the execution of each RSI operation, in order to 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 in every activation of the RSS, which maintains a collection of queue structures called *gates* in shared, read/write memory. Some of these gates are numbered and are associated by convention with such resources as the table of buffer contents, or the availability of the database for processing. However, in order to handle locks on a potentially huge set of objects like the tuples themselves, the RSS also includes a named gate facility. Internal components can request a lock by giving an eight-character name for the object, using such names as a tuple identifier, index value, or page number. If the named resource is already locked it will have a gate. If not, then a named gate will be allocated from a special pool of numbered gates. The named 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 duration of a lock is also used for scheduling purposes, such as to select a transaction for backout when deadlock is detected.

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. If a tuple is inserted or updated by a transaction at any consistency level, then an exclusive lock must be held on the tuple (or some superset) until the transaction has ended. If a tuple is deleted, then an exclusive lock must be held on the *TID* of that tuple for the duration of the transaction, in order to guarantee that the deletion can be undone correctly during transaction backout. For any of these cases, as well as for the ones described below, an additional lock is typically set on the page itself to prevent conflict of transactions at the physical level. However, these page locks are released at the end of the RSI call.

In the case of a transaction with Level 3 consistency, share locks must be maintained on all tuples and index values which are read, for the duration of the transaction, to insure repeatability. For transactions with Level 2 consistency, read accesses require a share lock with immediate duration. Such a lock request is enqueued behind earlier exclusive lock requests so that the user is assured of reading clean data. The lock is then released as soon as the request has been granted, since reads do not have to be repeatable. Finally, for transactions with Level 1 consistency, no locks are required for read purposes, other than short locks on pages to insure that the read operation is correct.

Data items can be locked at various granularities, to insure that various applications run efficiently. For example, locks on single tuples are effective for transactions which access small amounts of data, while locks on entire relations 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 [13]. The basic idea of the scheme is that separate locks are associated with each granularity of object, such as segment, relation, and tuple. If the RDS requests a lock on an entire segment in share or exclusive mode, then every tuple of every relation in the segment is implicitly locked in the same mode. If the RDS requests a lock on a single relation, 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 relation. 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 tuples, through automatic acquisition of intent locks on the segment and relation, before a lock is acquired on the tuple 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. The detection is done by the Monitor, on a periodic basis, by looking for cycles in a user-user matrix. The selection of a victim is based on the relative ages of transactions in each deadlock cycle, as well as on the durations of the locks. In general the RSS selects the youngest transaction whose lock is of short duration, i.e. being held for the duration of a single RSI call, since the partially completed call can easily be undone. If none of the locks in the cycle are of short duration, then the youngest transaction is chosen. This transaction is then backed out to the save point preceding the offending lock request, 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.)

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 been completed, and no other transactions were in progress. At such a state all image and link pointers are correct at the RSS level, and more importantly all user defined integrity assertions on data values are valid at the RDS level, since the RDS guarantees all integrity constraints at transaction boundaries.

In the RSS, special attention has been given to reduce the need for complete database dumps from disk to tape to accomplish a system checkpoint. The database dump technique has several difficulties. Since the time to copy the database to tape may be long for large databases, checkpoints may be taken infrequently, such as overnight or weekly. System restart is then a time consuming process, since many database changes must be reconstructed from the system log to restore a recent database state. In addition, before the checkpoint is performed, all ongoing transactions must first be completed. If any of these are long, then no new transactions are allowed to initiate until the long one is completed and the database dump is taken.

In the RSS, two system recovery mechanisms have been developed to alleviate these difficulties. The first mechanism uses disk storage to recover in the event of a "soft" failure which causes the contents of main memory to be lost; it is oriented toward frequent checkpoints and rapid recovery. The second mechanism uses tape storage to recover in the relatively infrequent case that disk storage is destroyed; it is oriented toward less frequent checkpoints. In both mechanisms, checkpoints can be made while transactions are still in progress.

The disk oriented recovery mechanism is heavily dependent on the segment recovery functions described above, and also on the availability of transaction logs. The 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 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 of current page map values equal to the backup page map values and the releasing of pages allocated since the save point. The log segment, which is saved more frequently than normal data segments, is effectively saved 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, but before the last log save, can be redone automatically. In addition, the transaction logs are used to back out transactions which were incomplete at the checkpoint and cannot be redone, 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. The technique we have chosen is to have the Monitor schedule certain checkpoints as "long" rather than standard short ones. A long checkpoint performs the usual segment save operations described above, but also initiates a process which copies the saved pages from disk to tape. Thus the checkpoint to tape is incremental.

4. SUMMARY AND CONCLUSION

We have described the overall architecture of System R and also the two main components: the Relational Data System (RDS) and the Relational Storage System (RSS). The RSS is a concurrent user, data management subsystem which provides underlying support for System R. The Relational Storage Interface (RSI) has operations at the single tuple level, with automatic maintenance of an arbitrary number of value orderings, called *images*, based on values in one or more fields. Images are implemented through the use of multilevel index structures. The

RSS also supports efficient navigation from tuples in one relation to tuples in another, through the maintenance of pointer chain structures called *links*. Images and links, along with physical scans through RSS pages, constitute the access path primitives which the RDS employs for efficient support of operators on the relational, hierarchical, and network models of data. Furthermore, to facilitate gradual integration of data and changing performance requirements, the RSS supports dynamic addition and deletion of relations, indexes, and links, with full space reclamation, and the addition of new fields to existing relations—all without special utilities or database reorganization.

Another important aspect of the RSS is full support of concurrent access in a multiprocessor environment, through the use of gate structures in shared, read/write memory. Several levels of consistency are provided to control the interaction of each user with others. Also locks are set automatically within the RSS, so that even unsophisticated users can write transactions without explicit lock protocols or file open protocols. These locks are set on various granularities of data objects, so that various types of application environments can be accommodated.

In the area of recovery, transaction backout is provided to any one of an arbitrary number of user specified save points, to aid in the recovery of long application programs. Backout may also be initiated by the RSS during automatic detection of deadlock. A new recovery scheme is provided at the system level, so that both checkpoint and restart operations can be performed efficiently.

The RDS supports the Relational Data Interface (RDI), the external interface of System R, and provides the user with a consistent set of facilities for data retrieval, manipulation, definition, and control. The RDI is designed as a set of operators which may be called directly from a host program. It is expected that programs will be written on top of the RDI to implement various stand-alone relational interfaces and other, possibly nonrelational, interfaces.

The most important component of the RDS is the optimizer, which makes plans for efficient execution of high level operations using the RSS access path primitives. Of great importance in optimizing queries is the method by which tuples are arranged in physical storage. The RDS provides the RSS with clustering hints during insert operations, so that the tuples of a relation are physically clustered according to some value ordering, or placed near associated tuples along a binary link. Given the cluster properties of stored relations, the optimizer uses an access path strategy with the main emphasis on reducing the number of I/O operations between main memory and on-line, direct access storage.

In addition to the optimizer, the RDS contains components for various other functions. The authorization component allows the creator of a relation or view to grant or revoke various capabilities. The integrity system automatically enforces assertions about database values, which are entered through SEQUEL commands. A similar mechanism is employed to trigger one or more database actions when a given action is detected. The SEQUEL language may also be used to define any query as a named view. The access plan to materialize this view is selected by the optimizer, and can be stored away as a Pre-Optimized Package (POP) for subsequent execution. POPs are especially important for the support of transactions which are run repetitively, since they avoid much of the overhead usually associated with a high level of data independence.

APPENDIX I. RDI OPERATORS

Square brackets [ ] are used below to indicate optional parameters.

```
Operators for data definition and manipulation:

    SEQUEL ( [ <cursor name>,] <any SEQUEL statement> )

    FETCH ( <cursor name> [, <pointers to I/O locations>] )

    FETCH_HOLD ( <cursor name> [, <pointers to I/O locations> ] )

    OPEN ( <cursor name>, <name of relation or view> )

    CLOSE ( <cursor name> )

    KEEP ( <cursor name>, <new relation name>,
           <list of new field names> )

    DESCRIBE ( <cursor name>, <degree>, <pointers to I/O
               locations> )

    BIND ( <program variable name>, <program variable address> )


Operators on transactions and locks:

    BEGIN_TRANS ( <transaction id>, <consistency level> )

    END_TRANS

    SAVE ( <save point name> )

    RESTORE ( <save point name> )

    RELEASE ( <cursor name> )
```

APPENDIX II. SEQUEL SYNTAX

The following is a shortened version of the BNF syntax for SEQUEL. It contains several minor ambiguities and generates a number of constructs with no semantic support, all of which are (hopefully) missing from our complete, production syntax. Square brackets [ ] are used to indicate optional constructs.

```
statement  ::=  query
              |  dml-statement
              |  ddl-statement
              |  control-statement

dml-statement  ::=  assignment
                  |  insertion
                  |  deletion
                  |  update

query  ::=  query-expr [ ORDER BY  ord-spec-list ]

assignment  ::=  receiver <- query-expr

receiver  ::=  table-name [ ( field-name-list ) ]

insertion  ::=  INSERT  INTO  receiver  :  insert-spec

insert-spec  ::=  query-expr
                |  literal
                |  constant

field-name-list  ::=  field-name
                    |  field-name-list  ,  field-name
```

```
deletion  ::=  DELETE table-name [ var-name ]  [ where-clause ]

update  ::=  UPDATE  table-name [ var-name ] set-clause-list
                [ where-clause ]

where-clause  ::=  WHERE  boolean
                 |  WHERE  CURRENT [ TUPLE ] OF
                       [ CURSOR ] cursor-name

set-clause-list  ::=  set-clause
                    |  set-clause-list  ,  set-clause

set-clause  ::=  SET  field-name  =  expr
               |  SET  field-name  =  ( query-expr )

query-expr  ::=  query-block
               |  query-expr  set-op  query-block
               |  ( query-expr )

set-op  ::=  INTERSECT  |  UNION  |  MINUS

query-block  ::=  select-clause  FROM  from-list
                        [ WHERE boolean ]
                        [ GROUP BY  field-spec-list
                          [ HAVING boolean ] ]

select-clause  ::=  SELECT [ UNIQUE ] sel-expr-list
                  |  SELECT [ UNIQUE ] *

sel-expr-list  ::=  sel-expr
                  |  sel-expr-list  ,  sel-expr

sel-expr  ::=  expr [ : host-location ]
            |  var-name  .  *  |  table-name     *

from-list  ::=  table-name [ var-name ]
             |  from-list  ,  table-name [ var-name ]

field-spec-list  ::=  field-spec
                    |  field-spec-list  ,  field-spec

ord-spec-list  ::=  field-spec [ direction ]
                  |  ord-spec-list  ,  field-spec [ direction ]

direction  ::=  ASC  |  DESC

boolean  ::=  boolean-term
            |  boolean  OR  boolean-term

boolean-term  ::=  boolean-factor
                |  boolean-term  AND  boolean-factor

boolean-factor  ::=  [ NOT ] boolean-primary

boolean-primary  ::=  predicate
                    |  ( boolean )

predicate  ::=   expr  comparison  expr
               |  expr  BETWEEN  expr  AND  expr
               |  expr  comparison  table-spec
               |  < field-spec-list >  =  full-table-spec
               |  < field-spec-list >  [ IS ]  IN  full-table-spec
               |  IF  predicate  THEN  predicate
               |  SET ( field-spec-list )  comparison
                        full-table-spec
               |  SET ( field-spec-list )  comparison
                        SET ( field-spec-list )
               |  table-spec  comparison  full-table-spec

full-table-spec  ::=  table-spec
                    |  (  entry  )
                    |  constant

table-spec  ::=  query-block
               |  ( query-expr )
               |  literal

expr  ::=  arith-term
         |  expr  add-op  arith-term
```

```
arith-term  ::=  arith-factor
               |  arith-term  mult-op  arith-factor

arith-factor  ::=  [ add-op ] primary

primary  ::=  [ OLD | NEW ]  field-spec
            |  set-fn  ( [ UNIQUE ] expr )
            |  COUNT  (  *  )
            |  constant
            |  (  expr  )

field-spec  ::=  field-name
               |  table-name  .  field-name
               |  var-name  .  field-name

comparison  ::=  comp-op
               |  CONTAINS
               |  DOES NOT CONTAIN
               |  [  IS  ]  IN
               |  [  IS  ]  NOT IN

comp-op  ::=  =  | ¬ =  |  >  |  >=  |  <  |  <=

add-op  ::=  +  |  -

mult-op  ::=  *  |  /

set-fn  ::=  AVG  |  MAX  |  MIN  |  SUM  |  COUNT  |  identifier

literal  ::=  (  lit-tuple-list  )
            |  (  entry-list  )
            |  lit-tuple

lit-tuple-list  ::=  lit-tuple
                   |  lit-tuple-list  ,  lit-tuple

lit-tuple  ::=  <  entry  >
              |  <  entry-list  >

entry-list  ::=  entry  ,  entry
              |  entry-list  ,  entry

entry  ::=  [ constant ]

constant  ::=  quoted-string
             |  number
             |  host-location
             |  NULL
             |  USER
             |  DATE
             |  field-name  OF CURSOR  cursor-name
                    [ ON  table-name ]

table-name  ::=  name

image-name  ::=  name

link-name  ::=  name

asrt-name  ::=  name

trig-name  ::=  name

name  ::=  [ creator  . ] identifier

creator  ::=  identifier

user-name  ::=  identifier

field-name  ::=  identifier

var-name  ::=  identifier

cursor-name  ::=  identifier

host-location  ::=  identifier

integer  ::=  number
```

```
ddl-statement  ::=  create-table
                  | expand-table
                  | keep-table
                  | create-image
                  | create-link
                  | define-view
                  | drop
                  | comment

create-table  ::=  CREATE  [ perm-spec ]  [ share-spec ]  TABLE
                     table-name  :  field-defn-list

perm-spec  ::=  PERMANENT  |  TEMPORARY

share-spec  ::=  SHARED  |  PRIVATE

field-defn-list  ::=  field-defn
                    | field-defn-list  ,  field-defn

field-defn  ::=  field-name  ( type [ , NONULL ] )

type  ::=  CHAR  ( integer )
         | CHAR ( * )
         | INTEGER
         | SMALLINT
         | DECIMAL ( integer , integer )
         | FLOAT

expand-table  ::=  EXPAND  TABLE  table-name  ADD
                       FIELD  field-defn

keep-table  ::=  KEEP  TABLE  table-name

create-image  ::=  CREATE  [ image-mod-list ]  IMAGE  image-name
                       ON  table-name  ( ord-spec-list )

image-mod-list  ::=  image-mod
                   | image-mod-list  image-mod

image-mod  ::=  UNIQUE
              | CLUSTERING

create-link  ::=  CREATE  [ CLUSTERING ]  LINK  link-name
                     FROM  table-name ( field-name-list )
                     TO  table-name ( field-name-list )
                     [ ORDER BY ord-spec-list ]

define-view  ::=  DEFINE  [ perm-spec ]  VIEW  table-name
                  [ ( field-name-list ) ]  AS  query

drop  ::=  DROP  system-entity  name

comment  ::=  COMMENT ON  system-entity  name  :  quoted-string
            | COMMENT ON FIELD  table-name . field-name
                  :  quoted-string

system-entity  ::=  TABLE  |  VIEW  |  ASSERTION
                  | TRIGGER  |  IMAGE  |  LINK

control-statement  ::=  asrt-statement
                      | enforcement
                      | define-trigger
                      | grant
                      | revoke

asrt-statement  ::=  ASSERT  asrt-name [ IMMEDIATE ]
                        [ ON asrt-condition ]  :  boolean

asrt-condition  ::=  action-list
                   | table-name  [ var-name ]

action-list  ::=  action
                | action-list  ,  action

action  ::=  INSERTION OF  table-name [ var-name ]
           | DELETION OF  table-name [ var-name ]
           | UPDATE OF  table-name [ var-name ]
                [ ( field-name-list ) ]
```

```
enforcement  ::=  ENFORCE  INTEGRITY
                | ENFORCE ASSERTION  asrt-name

define-trigger  ::=  DEFINE TRIGGER  trig-name
                       ON trig-condition  :  ( statement-list )

trig-condition  ::=  action
                   | READ OF  table-name [ var-name ]

statement-list  ::=  statement
                   | statement-list  ;  statement

grant  ::=  GRANT [ auth ] table-name  TO  user-list
                      [ WITH GRANT OPTION ]

auth  ::=  ALL RIGHTS ON
         | operation-list  ON
         | ALL BUT  operation-list  ON

user-list  ::=  user-name
              | user-list  ,  user-name
              | PUBLIC

operation-list  ::=  operation
                   | operation-list  ,  operation

operation  ::=  READ
              | INSERT
              | DELETE
              | UPDATE [ ( field-name-list ) ]
              | DROP
              | EXPAND
              | IMAGE
              | LINK
              | CONTROL

revoke  ::=  REVOKE [ operation-list ON ] table-name
                    FROM  user-list
```

APPENDIX III. RSI OPERATORS

The RSI operators are oriented toward the use of formatted control blocks. Rather than explain the detailed conventions of these control blocks, we list below an approximate but hopefully readable form for the operators. Square brackets [ ] are used to indicate optional parameters.

```
Operators on segments:

    OPEN_SEGMENT ( <segid> )

    CLOSE_SEGMENT ( <segid> )

    SAVE_SEGMENT ( <segid> )

    RESTORE_SEGMENT ( <segid> )


Operators on transactions and locks:

    START_TRANS ( <consistency level> )

    END_TRANS

    SAVE_TRANS, RETURNS ( <saveid> )

    RESTORE_TRANS ( <saveid> )

    LOCK_SEGMENT ( <segid>, <mode: SHARE or EXCLUSIVE or SIX> )

    LOCK_RELATION ( <segid>, <relid>, <mode, as above> )

    RELEASE_TUPLE ( <segid>, <tid> )
```

```
Operators on tuples and scans:
    FETCH ( <segid>, <relid>, <identifier: tid or scanid or imageid,
            key values>, <field list>, <pointers to I/O locations>
            [, HOLD] )
    INSERT ( <segid>, <relid>, <pointers to I/O locations>
             [, <nearby tid> ] ),   RETURNS ( <tid> )
    DELETE ( <segid>, <relid>, <identifier, as above> )
    UPDATE ( <segid>, <relid>, <identifier, as above>,
             <field list>, <pointers to I/O locations> )
    OPEN_SCAN ( <segid>, <path: relid or imageid or linkid>,
                <start-point: key values for image, or tid for link,
                or scanid for link> ),
                  RETURNS ( <scanid> )
    NEXT ( <segid>, <scanid>, <field list>, <pointers to I/O locations>
           [, <search argument>] [, HOLD] )
    CLOSE ( <segid>, <scanid> )
    PARENT ( <child segid>, <linkid>, <identifier for new tuple, as
             above>, <field list>, <pointers to I/O locations>
             [, HOLD] )
    CONNECT ( <child segid>, <linkid>, <identifier for new tuple, as
              above>, <neighbor relid>, <neighbor tid>,
              <location: BEFORE or AFTER> )
    DISCONNECT ( <child segid>, <linkid>, <identifier for child, as
                 above> )

Operators for data definition:
    CREATE ( <segid>, <object type: REL or IMAGE or LINK >, <specs> ),
               RETURNS ( <object identifier: relid or imageid or linkid> )
    DESTROY ( <segid>, <object identifier, as above> )
    CHANGE ( <segid>, <object identifier, as above>,
             <new specs> )
    READSPEC ( <segid>, <object identifier, as above>,
               <pointer to I/O location> )
```

ACKNOWLEDGMENTS

The authors wish to acknowledge many helpful discussions with E.F. Codd, originator of the relational model of data, and with L.Y. Liu, manager of the Computer Science Department of the IBM San Jose Research Laboratory. We also wish to acknowledge the extensive contributions to System R of Phyllis Reisner, whose human factors experiments (reported in [24, 25]) have resulted in significant improvements in the SEQUEL language.

REFERENCES

1. ASTRAHAN, M.M., AND CHAMBERLIN, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.
2. ASTRAHAN, M.M., AND LORIE, R.A. SEQUEL-XRM: A relational system. Proc. ACM Pacific Conf., San Francisco, Calif., April 1975, pp. 34–38.
3. BAYER, R., AND MCCREIGHT, E.M. Organization and maintenance of large ordered indexes. *Acta Informatica 1* (1972), 173–189.
4. BOYCE, R.F., AND CHAMBERLIN, D.D. Using a structured English query language as a data definition facility. Res. Rep. RJ 1318, IBM Res. Lab., San Jose, Calif., Dec. 1973.
5. CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structured English query language. Proc. ACM SIGFIDET Workshop, Ann Arbor, Mich., May 1974, pp. 249–264.
6. CODASYL DATA BASE TASK GROUP. April 1971 Rep. (Available from ACM, New York.)
7. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
8. CODD, E.F. Relational completeness of data base sublanguages. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, G. Forsythe, Ed., Prentice-Hall, Engelwood Cliffs, N.J., 1971, pp. 65–98.
9. DONOVAN, J.J., FESSEL, R., GREENBERG, S.S., AND GUTENTAG, L.M. An experimental VM/370 based information system. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 549–553. (Available from ACM, New York.)
10. ESWARAN, K.P., AND CHAMBERLIN, D.D. Functional specifications of a subsystem for data base integrity. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 48–68. (Available from ACM, New York.)
11. Feature analysis of generalized data base management systems. CODASYL Systems Committee Tech. Rep., May 1971. (Available from ACM, New York.)
12. GOLDSTEIN, R.C., AND STRNAD, A.L. The MACAIMS data management system. Proc. ACM SIGFIDET Workshop on Data Description and Access, Houston, Tex., Nov. 1970, pp. 201–229.
13. GRAY, J.N., LORIE, R.A., PUTZOLU, G.R., AND TRAIGER, I.L. Granularity of locks and degrees of consistency in a shared data base. Proc. IFIP Working Conf. on Modelling of Data Base Management Systems, Freudenstadt, Germany, Jan. 1976, pp. 695–723.
14. GRAY, J.N., AND WATSON, V. A shared segment and inter-process communication facility for VM/370. Res. Rep. RJ 1579, IBM Res. Lab., San Jose, Calif., Feb. 1975.
15. GRIFFITHS, P.P., AND WADE, B.W. An authorization mechanism for a relational data base system. Proc. ACM SIGMOD Conf., Washington, D.C., June 1976 (to appear).
16. HELD, G.D., STONEBRAKER, M.R., AND WONG, E. INGRES: A relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 409–416.
17. Information Management System, General Information Manual. IBM Pub. No. GH20-1260, IBM Corp., White Plains, N.Y., 1975.
18. Introduction to VM/370. Pub. No. GC20-1800, IBM Corp., White Plains, N.Y., Jan. 1975.
19. LORIE, R.A. XRM—An extended (n-ary) relational memory. IBM Scientific Center Rep. G320-2096, Cambridge, Mass., Jan. 1974.
20. LORIE, R.A., AND SYMONDS, A.J. A relational access method for interactive applications. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, G. Forsythe, Ed., Prentice-Hall, Engelwood Cliffs, N.J., 1971, pp. 99–124.
21. MYLOPOULOS, J., SCHUSTER, S.A., AND TSICHRITZIS, D. A multi-level relational system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 403–408.
22. NOTLEY, M.G. The Peterlee IS/1 System. IBM UK Scientific Center Rep. UKSC-0018, March 1972.
23. Planning for Enhanced VSAM under OS/VS. Pub. No. GC26-3842, IBM Corp., White Plains, N.Y., 1975.
24. REISNER, P. Use of psychological experimentation as an aid to development of a query language. Res. Rep. RJ 1707, IBM Res. Lab., San Jose, Calif., Jan. 1976.
25. REISNER, P., BOYCE, R.F., AND CHAMBERLIN, D.D. Human factors evaluation of two data base query languages: SQUARE and SEQUEL. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 447–452.

26. SCHMID, H.A., AND BERNSTEIN, P.A. A multi-level architecture for relational data base systems. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 202–226. (Available from ACM, New York.)
27. SMITH, J.M., AND CHANG, P.Y. Optimizing the performance of a relational algebra database interface. *Comm. ACM 18*, 10 (Oct. 1975), 568–579.
28. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. Proc. ACM SIGMOD Conf., San Jose, Calif., May 1975, pp. 65–78.
29. TODD, S. PRTV: An efficient implementation for large relational data bases. Proc. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 554–556. (Available from ACM, New York.)
30. WHITNEY, V.K.M. RDMS: A relational data management system. Proc. Fourth Internat. Symp. on Computer and Information Sciences, Miami Beach, Fla., Dec. 1972, pp. 55–66.
31. ZLOOF, M.M. Query by Example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431–437.

Received November 1975; revised February 1976