Sibyl: A relational database system with remote-access capabilities

by MANFRED RUSCHITZKA,
ANDREW CHOI,
and JOHN L. CLEVenger
University of California
Davis, California

ABSTRACT

The proliferation of inexpensive microprocessor systems and communications equipment has provided the general public with the ability to access remote database systems. It also makes off-loading of some of the query-processing load of such remote systems to microprocessor systems an attractive possibility, but problems concerning data portability and adequate software support need to be resolved. To demonstrate the feasibility of such a loose coupling of a microprocessor system with different brands of remote database systems, a relational database system capable of exchanging data with heterogeneous remote systems was designed and implemented. We describe the functionality of this operational system as well as the design and implementation of its major components.
INTRODUCTION

The maturing of database technology manifests itself in increasing numbers of very large databases that specialize—often nationwide—in specific areas of knowledge. The holdings of major libraries, weather data, news services, corporate information, and stock market reports represent examples of such domains. At the same time, the proliferation of inexpensive microprocessor systems and communications equipment increasingly provides the general public with the means to query such remote, specialized database systems. While some of these systems are capable of handling several thousand queries per minute (for example, high-performance airline reservation systems), such a rate cannot readily be exceeded with today's technology. Thus, increased access by the general public may severely saturate such remote databases. There are two approaches being taken to resolve this saturation problem: increasing the computational power and reducing the load of these systems. The former approach focuses on novel architectures for data storage devices, improved processing algorithms, and increased speed of devices. The latter approach deals with a spectrum of off-loading techniques ranging from tightly coupled distribution of the database functions over several systems to the delegation of individual sub-tasks to other systems.

Our approach at the Computer Systems Research Laboratory at the University of California at Davis is of the off-loading type. Specifically, we were interested in demonstrating that a microprocessor system that functions as a remote terminal of an interactive database system can itself be employed to handle a part of the generated load. Since microprocessors typically run in standalone mode and consequently tend to be lightly loaded, off-loading remote databases to them appears particularly attractive in terms of cost-effectiveness.

Over the past several years, a number of organizations have placed several thousand communicating microprocessor systems in the homes of employees. These systems provide additional computing power without requiring additional capacity from the central corporate facilities, but there is no unanimity yet on the effectiveness or desirability of such arrangements. Part of the reason is that contemporary microprocessor systems lack the sophisticated processing capabilities of the central facilities; data can be shipped, but processing is limited by the capabilities and the compatibility of the available software. In providing these software functions, however, special care must be taken to ensure the continuing uniformity, reliability, and integrity of the data.

To demonstrate the feasibility of our approach, we designed and implemented a self-contained, microprocessor-based relational database system, called Sibyl. An essential component of Sibyl is its transformer module, which allows it to exchange data with remote database systems of different brands. We describe the system's functional capabilities and characteristics in the next section and discuss design and implementation issues of its major components in the remainder of this article.

OVERVIEW OF SIBYL

Sibyl is a relational database system. The database consists of a collection of names relations (or tables) each of which consists of an arbitrary number of tuples (or rows). A tuple consists of an arbitrary number of attributes of varying types. The corresponding attributes in the rows of a table form a column. With this in mind, a user may specify operations on the data in the database in a tuple-relational calculus language. This query language is a subset of QUEL, the query language of INGRES. (Its syntax, which will be discussed later, is summarized in Figure 4.) In this language, selection, projection, and joining are provided for without any implementation restrictions. In Codd's terminology, Sibyl thus qualifies as a relationally complete system.

Sibyl is the product of continuing research effort, and its design and implementation are subject to constant change and enhancement. With minor exceptions, this article describes Version 1.0 as it existed in January 1983. This version runs on an IBM Personal Computer with 192 Kbytes of primary memory and uses the DOS operating system. An RS232 port, connected to a dial-up modem, serves as the communications link to remote database systems. The configuration also contains 320 Kbytes of diskette storage, and a 10-Mbyte Winchester disk. This capacity limits the size of the database that can be operated on at any one time. While DOS was enhanced by an interrupt-driven RS232 package written in assembly language, all other modules are written in Pascal. Sibyl consists of about 5000 lines of source code and its load module occupies 100 Kbytes of primary memory. Version 1.0 supported communications with only one brand of remote database system, INGRES, and an installation at the University of California at Berkeley was used for testing and demonstrations. The development effort amounted to one person-year.

The overall structure of Sibyl is shown in Figure 1. The command interpreter is invoked from the DOS command processor. It distinguishes between two types of commands: query and communications commands. A query command is passed to the query parser, which translates it into an intermediate representation, a query tree, which is then passed to the query processor. The processor executes the query, relying on the relation manager for the maintenance and accesses...
to relations. The relation manager, in turn, depends on the storage manager for the physical representation of relations and their accesses. The storage manager makes use of DOS files. There are three communications commands: terminal emulation, transfer from INGRES, and transfer to INGRES. For each of these, the command interpreter invokes the transformer. Terminal emulation turns the IBM Personal Computer into a terminal, and can be used to connect to a remote INGRES installation via the RS232 port. If invoked for a transfer, the transformer transfers a relation to or from INGRES, transforming the relation to the format of the destination system in the process.

A Sibyl user typically will operate in local mode and occasionally ship a few relations to or from the remote INGRES system. In local mode, the query language provides the user with complete relational processing capabilities. In remote mode, up-to-date copies of relations at the remote system may be acquired and updated or new relations may be shipped back. The control over the remote database remains with its database manager; the Sibyl user has the same privileges as the interactive user has regarding the remote system.

STORAGE MANAGER

The storage manager is responsible for the allocation and deallocation of the physical storage of all relations. Between sessions, all relations are stored in DOS files. When being accessed, however, a relation may be stored in its entirety in primary memory or it may be stored in a DOS file. In the latter case, the relation is accessed through the buffer that the DOS file system maintains for that file. When a relation is accessed, an attempt is made to move the entire relation to primary memory to speed up the most typical accessing task, query processing. Users of the storage manager are not aware of the physical storage medium of a relation; the storage manager moves relations internally between primary and secondary storage to optimize performance, and selects the appropriate routines when it is invoked for an operation.

The allocation unit both in primary (or "core") memory and on secondary (or disk) storage is a page of 512 bytes. The ordered sequence of pages of a single relation is called a heap. The storage manager maintains its own buffer pool for the allocation of core heaps, but relies on the DOS file system for the allocation of disk heaps. A heap control block (HCB) serves to locate a heap. For core heaps the HCB contains a sequence of pointers into the buffer pool. Since a disk heap is implemented as a DOS file the DOS file control block is used as the corresponding HCB. In either case, HCBs are referred to by a heap name and initialized when the named heap is activated.

A relation is encoded as a sequence of tuples. The physical counterpart of a tuple is a record, and the heap of a relation may be viewed as a sequence of records identified by number. Since records are stored contiguously, individual records may cross page boundaries. Given a record number, the HCB, which contains the size of its records, can be used to determine the page (or pages) containing the specified record, and locate its offset within that page for an access.

The interface of the storage manager, which includes the following six routines, summarizes its externally invocable functions.

- initialization and termination of the storage manager
- activation and deactivation of a heap
- reading and writing of a record

Note that the storage manager does not support accesses to the physical representation of attributes. The notion of attributes is provided by the relation manager, which stores them as fields within records and relies on the storage manager for accessing the latter.

RELATION MANAGER

The relation manager maintains and operates on relations, their tuples, and their attributes. It depends on the storage manager for allocating and accessing the physical representations of relations (heaps) and their tuples (records).

The basic building block of a relation is an attribute. It is characterized by a type (integer, string [length], etc.). A type machine[k] is also provided as a type escape mechanism: It denotes a raw block of k bytes. Note that the type implicitly specifies the size of an attribute in bytes. When an attribute is part of a relation, it is stored in some tuple and the offset within this tuple must be known in order to access the attribute. An attribute index, which consists of both the type and the tuple offset (in bytes) of an attribute, has been provided as a type escape mechanism: It denotes a raw block of k bytes. Note that the type implicitly specifies the size of an attribute in bytes. When an attribute is part of a relation, it is stored in some tuple and the offset within this tuple must be known in order to access the attribute. An attribute index, which consists of both the type and the tuple offset (in bytes) of an attribute, has been provided for this purpose; it is a characteristic of a column of a relation.

A tuple is a collection of attributes of possibly different types. The physical representation of a tuple is a record, and the type of the latter is machine[s] where s is the sum of the attribute sizes. Records are read and written by routines in the storage manager. A tuple identifier (TID) serves for referenc-
ing a specific tuple within a relation. It may be thought of as an imaginary attribute (column) of every relation. The TIDs of a given relation are unique.

A relation is a set of tuples. A user views it as a table where the rows correspond to tuples and the columns represent the attributes of the relation. Sibyl maintains a system catalogue of its relations. This catalogue is contained in two files: one for the relation descriptors and one for the mapping of relation names into indices of the relation descriptor file. The left half of Figure 2, to be discussed later, shows the contents of a relation descriptor. Its function is analogous to that of any file directory entry in a contemporary file system. Most of its entries are self-explanatory, but the validity map requires explanation. For reasons of efficiency, the deletion of a tuple from a relation does not result in the deletion of the corresponding record. Instead, a valid-bit can be reset for the same effect. There is one valid-bit per record in the relation, and the sequence of valid-bits is called the validity map. Since the size of the validity map is proportional to the number of records (which may be quite large), the map is actually contained in a separate map-catalogue file.

When a relation is opened, the relation manager initializes a relation control block (RCB) for accessing it. The RCB format is illustrated in Figure 2. Its relation descriptor part has been described above, but the validity map field now contains a pointer to a data structure that has been initialized from the map-catalogue file. The RCB also contains an access control block for information that is relevant only while the relation is open. Heap name and HCB pointer refer to the heap that contains the relation. The current tuple identifier provides for random and sequential accessing of tuples, and the current tuple buffer contains the current (or most recent) tuple accessed. The master flag will be discussed below (see subrelations). When a relation is closed, its RCB is not automatically discarded. Instead, the RCBs active-flag is reset. This design permits the reopening of a relation without the overhead of accessing the system catalogue. At the end of a session, the status of all remaining inactive relations is updated in the catalogue and the modified-flag determines whether the relation must be written out. Sibyl also supports auxiliary relations, which never enter the system catalogue; they are created and opened like any other relation, but are destroyed prior to the termination of a session.

The high frequency of selection operations during query processing demands high performance of their executions (c.f. the marking operation in ZETA/TORUS\(^5\)). For this reason, Sibyl supports objects of the subrelation type. A subrelation consists of a subset of the tuples of another relation (the master relation) and requires no additional physical representation. A subrelation is defined by an RCB and thus has all of the characteristics of a relation. Its heap, however, is that of the master relation, and its validity map identifies its tuples as a subset of the master relation. The result of a selection can therefore be represented by a subrelation, thus avoiding the storage and copying costs otherwise affiliated with the creation of a new relation for the selected tuples. Figure 3 depicts the linkage of two subrelations to the heap of their master relation. (Version 1.0 supports only auxiliary subrelations.)

The storage manager can be invoked for a variety of operations on relations, tuples, and attributes after it is initialized. The termination procedure saves all inactive relations in the system catalogue. Relations are referred to by name. Operations on relations include creation and destruction in the system catalogue, opening and closing (including subrelations), and printing and displaying. For reasons of integrity, tuples are operated on in the current tuple buffers of the RCBs; they can be referenced by matching or sequential accessing, but cannot be passed in their entirety to and from the relation manager. Instead, attributes are the units of exchange to and

![Figure 2](image-url)

**Figure 2**—A relation control block consists of a relation descriptor and an access control block.

![Figure 3](image-url)

**Figure 3**—Linkage of subrelations \(S_1\) and \(S_2\) to the heap of their master relation \(M\).
from current tuple buffers. Attributes are referred to by attribute indices; they can be inserted, extracted, compared, operated on, inspected (for types and sizes), and displayed. A complete specification of the interface of the relation manager is contained in Reference 8.

**QUERY LANGUAGE AND PARSER**

Sibyl's query language is a subset of QUEL, the query language of INGRES. It is complete in the sense that its expressive power is equivalent to that of relational algebra. Figure 4 describes the syntax of Sibyl's query language. The "help" command lists all Sibyl relations or, if a relation is named, the names, types, and sizes of its attributes. The "range" command is a declarative command; it associates a relation variable with a named relation. The "create" command creates an empty, named relation whose attributes are specified by the format list. "Destroy" destroys the named relation and "print" displays the named relation on the console.

The remaining four commands take as input all relations named in the qualification part and in the target to produce one output relation, the target relation. The qualification part specifies a conjunctive list of clauses, which define the set of tuples in the target relation. Version 1.0 lacks union and complementing of clauses, but since the supported comparison operators include their individual complements, this represents a loss in convenience, but not in the scope of representative queries. The name of a target relation is specified by a relation name or a relation variable (if it is omitted in a relation)

```
<COMMAND> ::= help <REL-NAME> | help | range of <REL-VAR> is <REL-NAME> | create <REL-NAME> ( <FORMAT-LIST> ) | destroy <REL-NAME> | append <REL-NAME> ( <TARGET-LIST> ) <QUAL-PART> | retrieve ( <TARGET-LIST> ) <QUAL-PART> | retrieve <REL-NAME> ( <TARGET-LIST> ) <QUAL-PART> | delete <REL-VAR> <QUAL-PART> | replace <REL-VAR> ( <TARGET-LIST> ) <QUAL-PART>
```

The scriptural expression, the target relation is displayed on the console.

The query processor interprets query commands that are passed to it in the form of query trees by the query parser. The utility commands "help," "create," "destroy," and "print" are interpreted by invoking the corresponding procedures in the relation manager. The commands "append," "retrieve," "delete," and "replace" constitute the actual query-processing commands; each scans all relations referenced in its qualification part and target list, and compiles a result relation. This common task is implemented by one shared procedure, the query processing nucleus.

```
<COMMAND> ::= help <REL-NAME> | help | range of <REL-VAR> is <REL-NAME> | create <REL-NAME> ( <FORMAT-LIST> ) | destroy <REL-NAME> | append <REL-NAME> ( <TARGET-LIST> ) <QUAL-PART> | retrieve ( <TARGET-LIST> ) <QUAL-PART> | retrieve <REL-NAME> ( <TARGET-LIST> ) <QUAL-PART> | delete <REL-VAR> <QUAL-PART> | replace <REL-VAR> ( <TARGET-LIST> ) <QUAL-PART>
```

For illustration purposes, consider the sample query which searches through the two relations nut(NUM, WT, COLOR) and bolt(CODE, WT, COLOR) and adds to the relation match(NNR, BCD) those pairs of nut numbers and bolt codes that satisfy the four clauses. Assume that the type of the attributes NUM, WT, CODE, NNR, and BCD was specified as integer and the type of COLOR was specified as string[8] when their respective relations were created. The query tree of this sample query is illustrated in Figure 5. The command node contains the command, the target relation name, and four lists: the target list and one list each for 0VCs, 1VCs, and MVCs. To improve query-processing performance, the query tree also contains several enhancements[11] that are not shown in Figure 5.

**THE QUERY PROCESSOR**

The query processor interprets query commands that are passed to it in the form of query trees by the query parser. The utility commands "help," "create," "destroy," and "print" are interpreted by invoking the corresponding procedures in the relation manager. The commands "append," "retrieve," "delete," and "replace" constitute the actual query-processing commands; each scans all relations referenced in its qualification part and target list, and compiles a result relation. This common task is implemented by one shared procedure, the query processing nucleus.

The nucleus is a generalized framework for query-processing primitives whose invocation sequences may be altered to effect alternate heuristic strategies. Invoked with a query tree, the nucleus operates on it, and generates tuples in a result-relation whose format corresponds to the target list in
SAMPLE QUERY: append to match (NNR=nut.NUM, BCD=bolt.CODE) where 5 < 4+2 and bolt.WT < 4 and bolt.COLOR = nut.COLOR

The query tree. It may invoke itself recursively with a modified query tree, but the result-relation does not change (i.e., every invocation appends tuples to the same result-relation). The initial Sibyl nucleus makes use of four primitives for evaluating 0VCs, selection, subtree generation for dissection, and result generation, respectively. Before discussing the nucleus in more detail, we briefly describe these four primitives:

- **function eval0VC(0VC):** boolean; `eval0VC` evaluates the passed 0VC and returns its value (true or false)
- **procedure select(1VC, subrelation):** `select` is being passed a 1VC and the name of an empty subrelation of the relation referenced in the 1VC; it selects, from the relation referenced in the 1VC, the tuples satisfying the 1VC and returns them in into the subrelation
- **function treegen(relation, tuple):** query tree; `treegen` returns a newly created query tree, which is a copy of the query tree of the current invocation of the nucleus, except that all references to attributes of the passed relation are replaced by the corresponding attribute values of the passed tuple
- **procedure resgen;** `resgen` appends tuples to the result relation as specified by the target list in the query tree. When `resgen` is invoked, any attribute in the target list either references a relation in which all tuples qualify (this may have been assured by a prior selection and relation name substitution) or represents a value that has been substituted in a preceding dissection. In general, `resgen` appends the target list projection of the cross product of the relations referenced in target list attributes. If all target list attributes have been substituted by their values, the cross product projection degenerates into a single tuple consisting of these target list values.

The nucleus framework lends itself to experimentation with different query-processing heuristics and their performance evaluation. A skeleton version that is structured according to clause types is shown in Figure 6. Its algorithm is similar to INGRES's decomposition, but reduction is not implemented. The nucleus is invoked with a query tree and generates a result relation which is global to all recursive invocations. It first evaluates all 0VCs; if any one is false, it implies that the result relation should be empty, and the nucleus returns. (On recursive invocations, this step will always be skipped, since all 0VCs have been dealt with.) If the query tree contains 1VCs, the appropriate selections are performed, the affected relations are substituted in the query tree by their respective subrelations, and the 1VCs are discarded from the query tree. A selection resulting in an empty subrelation indicates that no tuples qualify at this level, and the nucleus returns in such a case.

When MVCs are present, any one referenced relation may be chosen for dissection. The Version 1.0 strategy chooses the smallest. For every tuple of the chosen relation, the dissection step calls the nucleus recursively with a modified query tree returned by `treegen`. Thereafter, it returns; the final step, generating result tuples, cannot be performed until all MVCs in the query tree (if there are any) have been reduced to 1VCs and these 1VCs have been processed. Then, qualifying tuples

```plaintext
procedure nucleus (query tree);
   (evaluate 0VC's);
   for each 0VC var0 in the query tree:
      if not eval0VC(var0) then return,
      otherwise, remove the clause var0 from the query tree;
   (evaluate 1VC's);
   for each 1VC var1 in the query tree:
      select(var1, sub1);
      if sub1 is empty, return,
      otherwise, substitute in the query tree sub1 for the relation referenced in var1 and remove the 1VC var1 from the query tree;
   (evaluate MVC's);
   if the query tree contains any MVC's:
      choose a relation varR that is referenced in any of the MVC's;
      for each tuple varT of varR:
         nucleus(treegen(varR, varT));
   return;
   (append to result relation)
   resgen;
   return;
end nucleus;
```

Figure 6—Skeleton version of the query-processing nucleus
are appended to the result relation. It follows that the final step is only executed when the nucleus operates on a deepest recursive level; that is, when the MVC step, which calls the nucleus recursively and returns, is skipped.

The commands “append,” “retrieve,” “delete,” and “replace” are implemented using the query processing nucleus. The “append” command is implemented by invoking the nucleus directly with the query tree and the command’s target relation as the result relation. For the “retrieve” command, the query processor first creates the specified target relation and then passes it to the nucleus as its result relation. For the “delete” command, the nucleus is invoked with an auxiliary result-relation with only one attribute, TID, and a corresponding target list in the query tree. Upon return, the result relation contains the TIDs of the tuples to be deleted in the target relation, and the query processor uses it to perform the actual deletions. For the “replace” command, the nucleus is invoked to generate an auxiliary result-relation of replacement tuples concatenated with the TIDs of the tuples to be replaced in the target relation. The query processor subsequently uses the TIDs to locate and replace the corresponding tuples in the target relation.

THE TRANSFORMER

The transformer provides for Sibyl’s capability of transferring relations to and from remote database systems of a different brand, INGRES. It makes use of a communications package that includes interrupt-driven input-outout routines for the RS232 port and interactive dial-up and remote-command procedures for invoking INGRES with a specific database at the remote system. To the remote system Sibyl appears to be a terminal; the transformer sends and receives the same type of information and interactive user of the remote INGRES system would type and have displayed.

The transformer can be invoked for three functions: terminal emulation, relation transfer from INGRES to Sibyl, and relation transfer from Sibyl to INGRES. The terminal emulator turns the IBM Personal Computer into a terminal, and can be used to dial-up and log into a remote INGRES system. (It is also used to log out.) After being connected, the user may invoke either of the two transfer functions. The transformer then prompts the Sibyl user for the names of the Sibyl and the INGRES relations. When it terminates, a transformed copy of the source relation has been created at the destination system. The transformation is reversible if the attribute types of the attributes of the shipped relation are supported by the destination system. (In Sibyl Version 1.0, shipments were therefore restricted to integer and string attributes.

The transformer’s function may be viewed as a generalized transformation or translation of data from a source to a destination format. For such a translation, the description of these formats (the source and destination schemata) must be available; it may be built into the translation algorithm or it may be specified independently (e.g., in schema files). The generalization of the transformer in Sibyl stems from the fact that the schema of a remote relation must be acquired from the remote system. To do so, it requires a description of the format (the meta-schema) of requesting the schema of the remote relation (e.g., a certain command with specific parameters). In Sibyl, the schema of a relation R(A1,A2,...An) consists of the names, types, and lengths of its attributes A1,A2,...An. It is readily available in the system catalogue. INGRES, on the other hand, maintains this information in a relation named attribute on the remote system, and the transformer must send an appropriate query command to obtain it.

The transformer operates in two phases: the schema phase and the data phase. On a transfer from INGRES to Sibyl, the transformer is given the name of the requested INGRES relation, but not the names, types, or lengths of its attributes. The latter are obtained during the schema phase, and the data phase performs the relation transfer. Figure 7 illustrates the major operations. During the schema phase, the transformer sends to INGRES the commands

\[
\text{retrieve (a.attname, a.attfrmt, a.attfrml) where a.attrelid = " < requested relation name >"}
\]

which searches the relation attribute for the names, the types, and the lengths of all attributes of the requested relation, and sends this schema information in tabular form. The transformer receives it, transforms it into Sibyl objects, stores them for subsequent use, and creates a compatible Sibyl relation by calling the storage manager. During the subsequent data phase, the transformer sends INGRES commands for displaying the requested relation. Upon its arrival in tabular form, it is first copied into a DOS file. After the complete transfer of the requested relation, each of its tuples is read back from this file, transformed into Sibyl attributes by making use of the previously stored schema information, and appended to the destination relation by calling the storage manager. When the last tuple has been processed, the Sibyl copy of the remote

![Transformer diagram](image-url)
relation is complete, and the transformer invocation terminates.

Relation transfers from Sibyl to INGRES are handled by the transformer in a similar fashion. The schema information is now obtained from the relation manager, transformed, and sent to INGRES as part of a “create” command. Subsequently, individual tuples are processed by obtaining their individual attributes from the relation manager, transforming them into ASCII representation, and embedding them into “append” commands that are shipped to INGRES.

CONCLUSION

To demonstrate the feasibility of loosely coupling a microprocessor system with heterogeneous remote database systems, a relational database system, called Sibyl, was designed and implemented at the Computer Systems Research Laboratory at the University of California at Davis. This relationally complete system runs on an IBM Personal Computer with a 10-Mbyte Winchester disk. In addition to its function as a self-contained database system with its own interactive query language, Sibyl provides the capability of transforming relations to and from the data format of a remote system and transmitting the transformed data over a dial-up line. In this paper, we described Version 1.0 of Sibyl essentially as it existed in January 1983. An INGRES system running on a VAX/UNIX configuration at the Berkeley campus served as the remote database system for this version.

The paper provides an overview of the major Sibyl components: the storage manager, the relation manager, the query parser, the query processor, and the transformer. We emphasized those features that have a substantial effect on the performance or capabilities of the system. The concept of subrelations, as provided by the relation manager, permits drastic cuts both in memory requirement and execution time for nontrivial queries. The nucleus of the query-processing algorithm permits experimentation with different heuristics. The loose coupling with the remote INGRES system is a consequence of the data transformation capability of the transformer. Control over the INGRES database remains with its manager; Sibyl has no more privileges than an interactive INGRES user.

Sibyl is the product of continuing research effort and thus is subject to continuous enhancement. We have added FRAMIS16 to the repertoire of supported remote database systems; connected our IBM Personal Computers with an Ethernet cable, which we use to access relations; and modified the DOS file system to support Sibyl more efficiently. As for current and future research, we are addressing the design of a unified, syntax-driven transformer to deal with an open-ended set of remote systems. (Currently, two separate transformers are used for INGRES and FRAMIS interactions.)

Using data translation techniques,15 the syntax-driven transformer will make use of one syntax file for each remote system. This file contains the syntactic description of the remote data format as well as the syntax for connection commands and relation accesses. A second area concerns measurements of the performance of different query-processing heuristics. In particular, we are investigating the use of selectivity factors in the heuristics. We are focusing on verifying the conjecture that selectivity factors of relations measured during recently executed queries are a good indicator of the selectivity factors that will result from future queries involving those relations.

In any event, the operational first version of Sibyl demonstrates that microprocessors can be used cost effectively to provide a complete relational processing capability that permits the acquisition of data from and the integration of results in a remote, large-scale database system without qualitatively introducing additional problems regarding the integrity, the reliability, and the uniformity of its data.

ACKNOWLEDGMENT

Research sponsored by IBM Corporation and by Program MICRO of the State of California.

REFERENCES
