SEPTEMBER 1981 VOL. 4 NO. 1

a quarterly bulletin of the IEEE computer society technical committee on

Database Engineering

Contents

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in Database Engineering Technical Committee is open to IEEE Computer Society members, student members, and associate members. (Application form in this issue.)

Conference Report - SIGMOD '81

David Reiner
Arnie Rosenthal

Sperry Research Center
100 North Road
Sudbury, Mass. 01776

The 1981 ACM-SIGMOD International Conference on Management of Data was held at the University of Michigan, Ann Arbor, Michigan, from April 29 to May 1, 1981. Slightly over a third of the more than two hundred attendees were from universities, about a sixth were from research laboratories, and the remaining half represented various computer companies, software houses, and end users. Local arrangements were competently handled, and the conference ran smoothly.

The twenty-five papers presented ranged from fair to excellent. Most of the talks were good, although a few needed more rehearsal. In parallel with the presentation of papers, a number of well-attended panel sessions of somewhat variable quality were held. We describe below the conference highlights, focussing on the most important and controversial papers and discussions as we see them.

Several papers represented a movement to relate theory and practice. A paper presented by A. Motro and P. Buneman (University of Pennsylvania) on constructing "superviews" was rigorous in its mathematics and database theory, but was motivated by the real-world problem of producing a usable unified view of two databases while preserving their physical independence. From D. Maier and D.S. Warren (SUNY - Stony Brook) came a paper on extending standard relational database theory to include more computational power. The authors began in theoretical mode, and ended with a number of practical implementation considerations.

H. Sato (University of Tsukuba, Japan) gave a meticulously-structured paper on the derivability of data in a summary database, which seems very helpful in dealing with large summary databases with multiple users (or where the type of data collected may have changed with time).

Theoretical studies appeared to be moving in a more practical direction One common attitude seemed to be that theory needs to be simplified and assessed in terms of practical implementation. A major goal should then be to find assumptions which simplify the theoretician's task while still modelling most practical situations. For example, E. Sciore (SUNY - Stony Brook) presented a paper on real-world multivalued dependencies, in which he investigated how much complexity is actually needed in real-world situations. C.H. Papadimitriou (MIT) examined the expressive power and limitations of various locking primitives and proposed locking modes, as measured by their ability to implement different concurrency control principles.

P. Hawthorn (Lawrence Berkeley Laboratories) presented a well-received paper relating the design of a database machine to its intended target application. The paper showed that different database machine architectures are preferable for different types of processing, and gave a practical guide to the performance which may be expected from these architectures.

Several of the papers at the conference generated some controversy. W. Litwen (INRIA) proposed a new hashing algorithm, called "trie hashing", which relied on representing index levels very compactly, searching indices in main memory, and storing the records on disk, sorted by key. Although the method satisfied Knuth's definition of hashing, several attendees commented that the algorithm

was better viewed as a technique for age splitting and maintenance of a compressed sparse index, and that secondary storage utilization might be a problem. A paper by J.R. Jordan, J. Banerjee, and R. Batman (Sperry Univac) described an improved version of predicate locks, called "precision locks". These were claimed to yield maximum concurrency at low cost, and provoked spirited discussion.

P. Richard (INRIA) gave a paper on evaluating the size of relational algebra queries through a probabilistic model. The paper was quite thorough and may have closed off that particular area from a theoretical viewpoint. Obtaining the data necessary to apply the results could be quite difficult, however.

The design goals, architecture, and implementation of the evolving Cedar DBMS were described in a paper by M.R. Brown, R.G.G. Cattell, and N. Suzuki (Xerox Palo Alto). The system has several unusual aspects: It is accessible through personal computers (mostly Altos) on a local network, and it provides a low-level, data-independent interface which appears to be very useful for applications programming.

The first panel session, on the database aspects of VLSI design, attracted a large and interested crowd. Among the proposed systems was one at IBM Research, San Jose, which will integrate a DBMS, a system description system and a graphics system to help with VLSI design. The panel on commercial database machines covered the ICL Context Addressable File Store, the ADABAS Data Base Machine, the DAC Mega/Net, and Britton-Lee's Intelligent Database Machine. Several of the speakers were marketing-oriented and sometimes found it difficult to respond to questions about actual machine implementations.

A third panel session was on achieving high performance DBMS's in the 1980's. the theme of a few talks at this session seemed to be: "The way to achieve high performance is to buy our system," followed by a detailing of the system's highlights. Speakers who gave more general observations on current directions in commercial DBMS's were more appealing to the audience.

One good point was raised in discussion at this session. Users are becoming increasingly involved in the design of database applications, sometimes because MIS departments are not responsive to their needs. Different users have different perceptions of data, standards of database information content, concerns about data independence, and levels of attention to integrity constraints. When these users design or write applications from their varied points of view, it may be difficult to maintain the "conceptual unity" of the database (the unifying perspective which the database should bring to data access).

Additional panel sessions covered database administration, logical database design techniques, software, and services, and CAD/CAM data management needs.

Copies of the Conference Proceedings may be ordered prepaid from ACM Order Department, P.O. Box 64145, Baltimore, MD 21264. The price is $15 for ACM members and $20 for others, and the ACM order number is 472810. Next year's conference will be held in June, in Orlando, Florida.

A Project on Design Systems

Raymond Lorie

IBM Research
5600 Cottle Road
San Jose, Calif. 95193

This short note reports on work being done at the IBM Research Laboratory, in San Jose, CA., on the problem of managing the design data for complex VLSI circuits.

For many years design automation tools have been used for designing electronic circuits, but today the new technology known as Very Large Scale Integration (VLSI) poses new challenges. The electronic systems being designed comprise up to several hundred thousand gates. The turnaround time for building a prototype is extremely long; therefore the design must be extensively tested before the first chip is built. The possibility of improving the productivity of designing such chips hinges on our ability to provide adequate tools. Much attention is given to simulators, checkers, and compilers for high-level design languages, automatic placement and wiring programs, etc. Many of these tools have been used in the past, but are now being stressed to their limits because of the ever growing complexity of the components being designed. There are many phases in the design of VLSI circuits and for each phase large amounts of information must be stored: dataflow and algorithms at the architectural level, logic design, physical design, image data for masks, testing data, textual descriptions, etc. The structure of the information is generally complex, with many data elements of various types, and relationships between them. Even data created in different phases are inter-related: for example, there are relationships between the logic design and the corresponding physical design. In all phases of the design,

graphic terminals are used and it is well known that this mode of operations imposes additional requirements on the management of data. This quick analysis of the application requirements shows the need for a flexible repository of data.

It is therefore not surprising that the design community's interest in database systems has grown rapidly. In the past, users have encountered difficulties in trying to use commercially available systems for storing engineering data because these systems were generally too rigid. Therefore engineers continued to use files with the inherent disadvantages of poor control, high redundancy, and poor or nonexistent data independence. In fact, data are very often converted from one file format to another to fit the need of individual applications. It is our conjecture that the emergence of relational systems should improve the situation drastically.

The relational model and systems based on it, like System R (1), INGRES (2), or SQL/DS (3), are much more flexible and easier to use than previously available systems. Only tables are specified and their specifications do not involve any implementation issues. New tables can be defined dynamically; new columns can be added easily to existing relations. There are no pointers linking together different rows in the same or different relations. Instead, all relationships between tuples are based on some mathematical relation (=,<,>, etc.) between data elements, and relational languages are used to exploit them. These features are particularly useful in applications as complex as VLSI design where it is impossible to know in advance all data types and relationships that are going to be used or how they are going to be used. It is precisely the purpose of our project to demonstrate how a relational system can be used to manage the VLSI

design data. (It should be noted that VLSI design is only an example and that our results should also benefit other engineering areas.) Our task is facilitated by some previous work done here at IBM San Jose Research, described in (4) and (5). TELL, for example, uses a relational system to implement the notion of hierarchical decomposition of a design. Designs are specified in terms of elements. Each of these elements is, in turn, designed in terms of simpler elements, and so on until one reaches the level of atomic elements. The designer, or a program, can study the data at a high level and require the details only when needed.

Although we are convinced that a relational system is most appropriate for our application, we also realize that systems like System R have been developed mainly for supporting data processing transactions; some enhancements could and should be made to support more efficiently a broader spectrum of applications. These enhancements are discussed in (6) and (7), together with the reasons for introducing them. Let us briefly mention the most important ones.

1. In engineering applications the user deals with objects that are more complex than single records or sets of homogeneous records. Quite frequently, an object is a hierarchy of tuples with a root tuple and one or several levels of dependent tuples that are part of the object. Even if such a structure is easily expressed relationally, by matching values, it cannot be manipulated as a single object. For example, in order to delete a single object the user would issue one delete for each tuple in the hierarchy. We are investigating the possibility of making such hierarchical structures known to the system so that a single delete operation could automatically cascade through the whole hierarchy. The same would be true for moving, copying and locking an object, or simply bringing an object

into main memory.

2. The creation and maintenance of these complex objects are generally handled interactively (through what we call **conversational transactions**) and may extend over a long time, maybe days or weeks. This is different from what happens in data processing applications where transactions only touch a few records and generally complete in less than a second. When transactions are so short, locks can be kept in volatile storage, waits can occur, transactions can be backed out in case of a deadlock. None of these techniques apply to conversational transactions and one way of handling the problem is to introduce the notion of a private database for each designer. Data are checked out of the public database, remain in the private database until a new consistent state of the design is reached, and then are sent back to the public database. A particular implementation could use a host processor to control the central (and shared) database and a series of satellites connected to it; each satellite would support a single user and his local, private database and terminals.

3. The objects often contain non-formatted data elements like text, strings, bit maps for images, etc. Such data elements may be too large to fit in main storage, but languages like SQL could be extended to support piecewise manipulation.

We believe that enhancements along these lines could substantially increase the value of relational systems for supporting engineering and design applications, in particular those related to VLSI design.

Acknowledgments

The author acknowledges the contribution of his colleagues who are most closely associated with the project: R. Haskin, W. Koenig, W. Plouffe and J. Rhyne. He is also endebted to P. Mantey for his constant support.

References

(1) Astrahan, M.M., et al., System R: a relational database management system, IEEE Computer, Vol 12, No 5 (1979) 42-48.

(2) Stonebraker, M. et.al., The design and implementation of INGRES, TODS, Vol 1, No 3 (1976) 189-222.

(3) SQL/DS IBM Program Product No. 5748-XXJ.

(4) Weller, D.L. and Williams, R., Graphic and relational database support for problem solving, Proc. ACM SIGGRAPH Conference, Computer Graphics, Vol 10, No 2 (1978) 183-189.

(5) Rhyne, J.R. and Dalpezzo, R.G., Distrubuted system design discipline: An application of TELL, Compcon IEEE, (February 1981) 328-332.

(6) Haskin, R.L. and Lorie, R.A., On extending the functions of a relational database system, IBM Research Report RJ3182 (July 1981).

(7) Lorie, R.A., Issues in databases for design applications, to appear in: Proceedings of IFIP WG 5.2, Working Conference on CAD Databases, Seeheim/Darmstadt, W. Germany (Sept 1981).

Transaction Flow in System-D

Kapali P. Eswaran

IBM Research
5600 Cottle Road
San Jose, Calif. 95193

Introduction

System-D is a distributed transaction processing system. The system consists of a set of computers interconnected by a relatively high speed local communications network. System-D emphasizes data integrity, availability, incremental growth and single-system image. In the System-D architecture, processes either in the same processor or in different processors exchange information solely by explicit messages. Sharing of variables between different processes are not permitted.

System Architecture

A resource, whether a process or a device, is a logical resource. Each process communicates with any other logical resource using datagram messages. A process may have one or more logical resource numbers associated with it and a logical resource number may be associated with one or more processes. A logical resource number is mapped to a mail box address. The address of a mail box is a (processor-id, box-id) pair. Messages can be sent by a process to any logical resource number it knows. A process may receive a message from any (local) mail box whose address it knows. There are no explicit acknowledgments provided by the system. The send and receive commands are synchronous with a time limit. For example, to receive a message, a process provides a mail box address and a time limit. The process gets control when there is a message in the mail box or when the time limit elapses.

In an interactive application processing environment of System-D, there are three distinct modules: the application, the data manager (a database system or a file system with an access path), and the storage manager (Input/Output supervisor). The application module, A, handles terminal interfaces and provides programming interface. The data manager module, D, understands and operates on records, relations, and access paths. The storage manager module, S, deals with concurrency (locking), commit and recovery in addition to fetching and storing pages. An application operates on a set of records obtained from a data manager module. An application process specifies the set of records that it is interested in by key association or by positional values. The data manager module computes the page numbers in which the desired records reside. The page numbers span all the disks attached to all the processors in the system. The data manager module issues a call to the local or remote storage manager module to fetch (or put) pages. All the changes made by an application are kept locally in the data manager module. The changes are sent to the storage manager by the D module only when the application commits.

We make a distinction between a module and an agent. A module is a function that exists in a node and is associated with a mail box. One or more processes in that node may perform that function and share the mail box. Such processes are called agents. For example, node N1 and N2 may have D modules; i.e. N1 and N2 each has at least one process that can perform the data manager function. If there are 3 processes (schedulable tasks) in N1 that share the D module mail box and perform the data manager function, then there are three D agents in N1.

Distributed Database System

A limited database system capability is provided in the present phase of the

prototype. There are two kinds of datasets: sequential datasets and database datasets. Sequential datasets are lockable by dataset names and are mainly used by the editor applications. Database datasets provide random access to sets of records by hashing or B-tree indexing on key values. The database itself is partitioned; i.e., a piece of information is resident on one and only one disk (however, all information is mirrored on two disks for availability reasons).

Data Integrity

Multiple transactions run concurrently in System-D which provides full data integrity. The effect of running multiple concurrent transactions is equivalent to running transactions according to some serial schedule. The system also guarantees that the serial schedule is the same in all the processors. Changes made by committed transactions are never lost, and no trace of aborted transactions is left on the database. The storage manager has the responsibility for data integrity. Shared and exclusive page locks are provided and enforced by the storage manager. A lock request is accompanied by a time limit. If the lock cannot be acquired within the time limit, the request is aborted. Thus, there is no explicit global deadlock detection mechanism.

System Design

Whenever a transaction gets started, say in node N, the system binds the transaction to a D module mail box. During the binding process, the system checks to see if the load on the local D module is 'light' (as indicated by the number of messages in the mail box). If so, the transaction is bound to the local D module. If not, the transaction is bound to the D module in node i+1 where i was the last node with D module to which a transaction in N was bound. This algorithm

attempts to bind transactions to local data managers as much as possible and when not possible, heuristically attempts to balance the load on the system network without incurring any message overhead. Once the A-D binding is done, the transaction communicates to that mail box for any database functions. Note that the mail box may be shared by one or more D agents and that the transaction is not bound to a single D agent. Each request of the transaction may be done by different D agents sharing the same mail box. However a request is completely processed by a single D agent.

As mentioned earlier, the page numbers span the entire network of computers. A D agent has the knowledge to compute the page number to obtain the required piece of information. The page number indicates the S module mail box to which the page number request should be sent. A page is obtained either in a shared or exclusive mode.

When a transaction makes a database request for the first time, it informs the D module the maximum amount of time the transaction expects to execute before commiting. When a D agent makes the first page-request for this transaction from a S module, it passes on this time value to that S module. The S module which services the first page-request is called the master for this transaction. The master initiates aborting of this transaction if the transaction does not commit within the specified time limit with the presumption that there is communication failure or the node in which the transaction and/or the D module has failed. We denote the master S module for a transaction T as MS(T). Whenever the D module for T, D(T), makes subsequent page-requests for T, it informs the various S modules the identification of MS(T). The other S modules are called the slaves with respect to MS(T). The main difference between the master and slave S modules is

that only the master can initiate the termination of a transaction.

Associated with each transaction T, there is a time limit T(w) which is the maximum amount of time that T is willing to wait for a lock. If a lock cannot be granted within the time T(w), the termination of T is initiated. The S module for T which times-out is called the termination initiator. The initiator communicates to D(T) that there has been a time-out and that the transaction is to terminated (the presumption is that there has been a deadlock). D(T) sends the list of slaves to MS(T) and asks that T be aborted. MS(T) asks all the slaves to abort T which involves releasing all the locks held by T and erasing existence T from the memory (from the Transaction Control Blocks). MS(T) also aborts the transaction. MS(T) then communicates to D(T) that the transaction has been aborted. D(T) does the necessary clean up and informs the transaction that it has probably been involved in a deadlock and that it has been aborted. The transaction may start again either immediately or after a time delay.

When a transaction wants to end, the end-transaction command is sent to D(T). D(T) initiates the commit procedure. At this time, we recall that all the changes made by T are kept locally in D(T). The change file containing all the modifications (to pages in the master and slave nodes) is sent by D(T) to MS(T). MS(T) is the commit-coordinator for this transaction. MS(T) records the information in a stable storage. When this is done, MS(T) notifies D(T) that the commit is complete as far as D(T) is concerned. D(T) then notifies the transaction. MS(T) then sends commit message to the slaves and begins making its own database changes. Each of the other slaves records its updates stably and in such a way that it knows the order in which transactions committed at that site. Each slave, then, acknowledges entry into the committed state to the MS(T). After

this each slave makes the database changes (applies the changes in the change file to the database). When MS(T) has received all of the acknowledgments, and has made all its own updates, it discards its log (the change file), returns the transaction to idle state (releases all the locks), and sends a message to the slave sites that this has occurred. Upon receipt of such a "finish" message, and when done updating its own database, a site can dispose of its local transaction log and enter the idle state. Note however, that as in other log management schemes, the log for a transaction cannot be disposed of until all previous transactions have been returned to the idle state.

Conclusion

System-D has implemented the notion of 'change files'. The granularity of locking in System-D is a page. Time-outs are used to "detect" live locks and dead locks. It is claimed that the scheme outlined will work in all possible failure modes (failure of the various sites, communication links failure, loss of messages etc.). A new commit procedure has been sketched. Readers are encouraged to compare this with standard commit procedures. They are also encouraged to verify that the commit scheme works and to propose various recovery schemes. The recovery scheme used in System-D and a detailed account of the commit procedure with a proof will be the subject of another paper. Discussions on change files, granularity of locking, time-out schemes, and commit procedures are welcome.

This note describes the work being done by the following members of the project: Sten Andler, Ignatius Ding, Kapali Eswaran, Carl Hauser, Won Kim, Jim Mehl and Tom Neuman. This note is being expanded and is being submitted as a journal paper authored jointly by the above individuals.

Special-Purpose Processors for Text Retrieval

Roger Haskin

IBM Research
5600 Cottle Road
San Jose, Calif. 95193

Introduction

With a rapidly increasing amount of computer-readable text existing as a side effect of electronic document preparation, there is more interest in methods that allow text to be searched and retrieved by content. Retrieval by content is useful in both small-system environments such as the automated office (i.e. 'electronic filing cabinets') and in large ones (i.e. classical online search services).

Conventional online search systems consist of disk storage units holding the text and any necessary indices, and a mainframe processor for searching. The cost of searching this text is proportional to the cost per instruction of the search processor. The cost per bit of disk storage has been dropping much faster than the cost per instruction of mainframe processors- a trend that promises to continue. Thus, the cost of searching text relative to that of storing it online is increasing.

This cost/performance problem is already a factor limiting the growth of text search systems. Typical commercially available systems are at the point where neither cost nor response time can be allowed to rise significantly. Representative numbers for one system are an average search time of several minutes during prime shift hours, and an average cost per simple query of $25 (this can go as high as $100 for more complex queries). Taking advantage of the dramatic decreases in the expense of online storage, both to increase the maximum

feasible size of the database in large-scale systems, and to lower the entry price for smaller-scale users, is going to require a radically new architecture for search processors, one in which the cost of searching the text rises at most linearly with the cost of storing it.

1. Search Strategies

Because of the relative expense of searching by comparing the text against a pattern, most conventional systems use indexing or file inversion to accomplish the same end. A dictionary of possible query terms is maintained, with each term's dictionary entry containing pointers to its occurrences in the database. The granularity of resolution of these pointers varies from system to system. They can point to comparitively large sections of the database (such as an entire document), to a small area (such as an individual occurrence of the term), or to some physical entity (such as a disk track).

Some systems rely entirely on indexing, allowing no direct comparison of the text data against query patterns. This requires that the proximity operations supported by the query language (i.e. finding words occurring in the same sentence) correspond to the granularity of the index. If the granularity is coarse (such as the document), precision and recall are adversely affected (i.e. queries don't retrieve the desired documents). If it is fine (such as the word), it becomes more expensive to store and consult the index. A word-level index can range in size from 50% to 300% of the size of the text itself [BiNeTr78]. Maintaining it is expensive; adding a new document requires on the order of one modification to the index for each unique term in the document. Usually, indexing is only appropriate in situations where updates are either infrequent or where they can be saved and processed in a batch during off-shift hours.

An alternative to indexing is to directly scan the text, comparing it

against a pattern generated from one or more users' queries. This, of course, can result in poor response time for large databases, depending upon the implementation (whether all disk drives are searched sequentially or in parallel; whether it is possible to search more than one track on a cylinder simultaneously, etc.). The amount of parallelism is a central architectural concern in a search-intensive system, and will be dealt with in more detail later.

It is possible to combine indexing and full searching in a strategy called 'partial inversion'. Here, a relatively coarse index is maintained (i.e. resolving to the cylinder, track, or document). Although the number of terms in the index remains constant as the granularity becomes more coarse, the number of pointers for each term decreases, and the index size can be reduced to around 20% to 30% of the size of the text. During query processing, the index is first consulted to restrict the search to regions of the text potentially satisfying the query. Next, these candidate regions are full searched to determine whether they actually satisfy the query. The advantage of partial inversion over full inversion is that it allows choosing the indexing level to optimize storage overhead and processing efficiency rather than to achieve a desired level of precision and recall. By varying the granularity of the index, the workload can be balanced between the index processing and text searching subsystems based upon their relative speeds.

2. Special-Purpose Hardware for Text Searching

An interesting question is whether, by using partial inversion, it is possible to confine full-text searching to a small enough area to make it reasonable to use a general purpose computer to perform the search. However, even assuming that such a processor could keep up with typical disk transfer rates (around one microsecond per character), and assuming that indexing will narrow the search to

a very optimistic 0.1% of the database, a query on a 50-billion character database (comparable to the size of on-line legal retrieval systems) will completely saturate the processor for almost a minute. Devoting this time to each query would limit the number of simultaneous users to an unrealistically low number, or (stated differently) would raise the cost per query unacceptably.

To find ways of getting acceptable response time for a reasonable number of users at tolerable cost, research has been and is being done into the design of special hardware to augment text search systems. Machines to perform both index processing and full-text searching have been proposed and analyzed.

Index-Processing Hardware

Two architectures that have been proposed for index processors are the Batcher merge network ([Stell75]) and the tree merger ([Hollaar76]). In both of these, queries to be processed are converted into trees of Boolean operations (i.e. AND, inclusive OR) between lists of postings (pointers to term occurrences). Stellhorn's merger processes the search tree one node at a time, using a Batcher network to perform the node operation on the two input lists, and producing intermediate results in memory. Hollaar's tree merger assigns one sequential processor to each node of the search tree. The output of a node's processor is connected directly to an input of the processor of the next higher level node. The fact that lists are merged sequentially rather than in parallel (as in the Batcher network) is balanced by the fact that all nodes are merged in parallel (rather than one at a time) and no intermediate results (that must be stored in and later read from buffers) are generated. Extensive modelling and simulation of both have been done ([Hurl76], [Miln76], [Huang80]). There is still disagreement over which represents the best solution, which is difficult to resolve since there is no working prototype of either system.

Index processing involved two basic steps: consulting a dictionary to find postings lists for each term involved in a query, and then merging these lists as specified by the search tree to obtain the result. Full-text searching can be broken up similarly: scanning the raw text data for occurrences of terms involved in the query (term matching), and keeping track of these occurrences to determine when an instance of the search expression has been detected (query resolution).

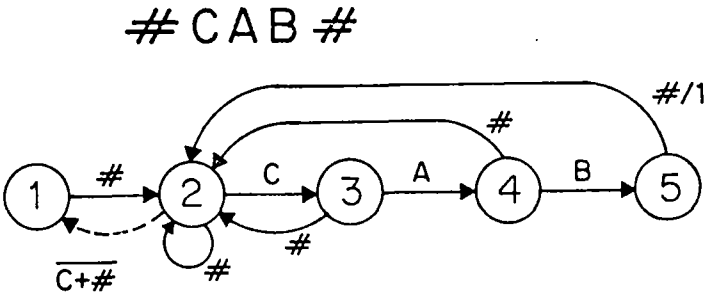
While the term matcher must run essentially at disk speeds, the query resolver need only process items as fast as the term matcher finds them. However, the processing required in response to each term hit can be rather complex, and can in fact depend upon details of the query language. For these reasons, proposed implementations of query resolvers usually use conventional processors (such as microcomputers) rather than custom hardware. Query resolution thus becomes a programming problem rather than an architecture problem, and therefore has generated substantially less interest than term matching. Interested readers are referred to Chapters 1 and 4 of [Hask80b] for a brief survey of query resolution.

Comparatively more attention has been paid to term matching, and several architectures have been suggested in the literature. Searchers using arrays of comparators to match terms have been proposed by Stellhorn [Stell74b], Foster and Kung [FoKu80], and Mules and Warter [MuWar79]. These matchers accept data at typical disk rates, requiring one matcher for each track being searched at any one time. The first two of these matchers share similar drawbacks. They cannot handle embedded variable-length don't cares (EVLDC's - patterns with a specified prefix and suffix and an unspecified middle). More importantly, they are extremely wasteful of hardware. Stellhorn's matcher, for example, uses a fixed

NxM array of cells, where N is the maximum term length allowed and M is the maximum number of terms allowed. The length of most search terms is under eight characters, but N must be chosen to allow reasonably long terms (at least 16 characters). This results in half the cells being unused. A similar argument applies to the choice of the maximum number of terms. Also, the comparator cells are typically limited in the comparisons they can perform, usually allowing only exact match or don't care. It is often required to support other functions, such as matching any punctuation or matching any alphanumeric. However, the additional logic necessary to implement these functions must be duplicated in each of the NxM comparators in the array.

Mules and Warter's matcher is a very clever design - it detects many types of keyboarding errors in the data (insertions, deletions, and transpositions), and matches terms in spite of them. This matcher was designed for searching a small, noninverted database. It is debatable how useful the error-detection feature is when searching is used in conjunction with indexing; presumably the index processor will only point the text searcher to areas known to contain correct spellings. Also, this searcher is designed to handle a small number of terms (16) of limited length (16 characters). It is not clear that it could be economically scaled up to handle the number of terms required in a large system.

Figure 1



STATE	INPUT #	A	B	C	D	....
1	2	1	1	1	1	...
2	2	1	1	3	1	...
3	2	4	1	1	1	...
4	2	1	5	1	1	...
5	2/1	1	1	1	1	....

Copeland [Cop78] and Mukhopahhyay [Muk78] proposed matchers based upon networks of match cells. Mukhopadhyay's design included both comparator cells and cells for functions such as character counting and Boolean operations. Instead of using an array of cells, they are connected as required to detect the pattern. Presumably, the connections must be reconfigured to detect different patterns. This was not discussed in [Muk78]. Reconfigurable networks have many gates and are difficult to build in LSI, so a large matcher with hundreds of cells might be quite expensive.

Finite State Automaton

A more promising architecture for term matching uses the concept of a finite-state automaton (FSA). Figure 1 shows the state diagram and associated table necessary to match the word '#CAB#', where '#' indicates a word break. When a '#' is seen, the FSA makes a transition to state 2. If the next input character is 'C', the FSA goes to state 3; on any other input the FSA goes either to state 2 (on a word break) or state 1 (for anything else). When the trailing '#' is seen, a match is signalled and the FSA returns to state 2. The state behavior can also be specified by the table shown in Figure 1. Each row represents a state, and the entry in each column corresponds to the next state if the input character shown at the top of the column is received. The FSA is a significant improvement over cellular matchers. It has less logic wasted due to term length variations, and since separate comparison logic is not required for each character in each term, it can be sophisticated as to the types of matching it will perform.

[OSI77a] describes a searcher based upon an FSA model. It is designed to search an entire disk sequentially; all queries arriving during one pass over the disk are batched and processed concurrently during the next pass. All terms

to be searched for in one batch are collected, and a state table is built. Batches of terms can become quite large, and storing the state table in an array as shown in Figure 1 results in its occupying a prohibitive amount of memory. However, since most transitions out of a state correspond to a mismatch (the '1' entries in Figure 1), it is possible to condense the state table substantially. The OSI design does this by breaking the state set into two classes. States having only one match transition leading out of them are called 'sequential' states, and those having more than one (i.e. when there is more than one valid alternative successor character) are called 'index' states. Sequential states have only one successor, which is stored in the following location in memory. No successor address is required, so a sequential state can be stored in fewer bits than an index state. Since typically only 10% of the states are index states, using this table encoding results in a substantial memory savings.

Basically, sequential state words contains the code of the next match character. If the next input character compares successfully, the next sequential state is entered. On a mismatch, a default transition to an idle state is taken. Index states, for which more than one successful match alternative are possible, require more sophisticated processing. Rather than indexing into a vector of next state addresses, one for each possible input character (wasteful of memory) or comparing against a list of alternative successful match characters (requiring multiple comparators or iteratively cycling one comparator), this design has a bit vector stored in each index state word indicating which input characters are being looked for in that state. If the input character code is k, the kth bit of the vector is tested. Bit k being set indicates a match. To find the address of the successor state, bits 0 thru k-1 are examined, and a count is made of how many are set. This count is added to a base address stored in the state word. The result is the address of the next state. If the input character's bit was clear,

the default transition to an idle state is taken.

Large input alphabets imply long bit vectors; storing text in 8-bit EBCDIC requires a 256 bit vector in each index state. In addition to the amount of memory required, engineering a leading-ones counter for long vectors is difficult; a fast counter requires many gates. To overcome this, [OSI77a] proposed a modified FSA that processes each 8-bit character as two 4-bit nibbles. Nibble processing requires making two state transitions per input character, increasing the number of memory accesses. However, the bit vector size is reduced to 16 bits, allowing the leading-ones counter to be speeded up, or even built as a ROM.

Certain pathological types of patterns greatly increase the complexity of the FSA state table. This can be illustrated most simply with an example. Figure 2 shows a state diagram to match the string 'ANAS'. Suppose the input string is 'BANANAS'. When the first 'ANA' has been matched, the FSA will be in state 3. When the following 'N' is seen, the FSA must 'backtrack' via the transition to state two to enable successful recognition of the succeeding 'AS' rather than taking the normal mismatch transition to state 0. Similarly, if a mismatch occurs when the input is 'A', the FSA must go to state 1 rather than 0, since the 'A' could be the start of 'ANAS'. The point of this example is that some terms can have a prefix that is embedded within another (or, in this case, the same) term. Considerable processing is necessary to detect these and include the necessary backtrack transitions in the table.

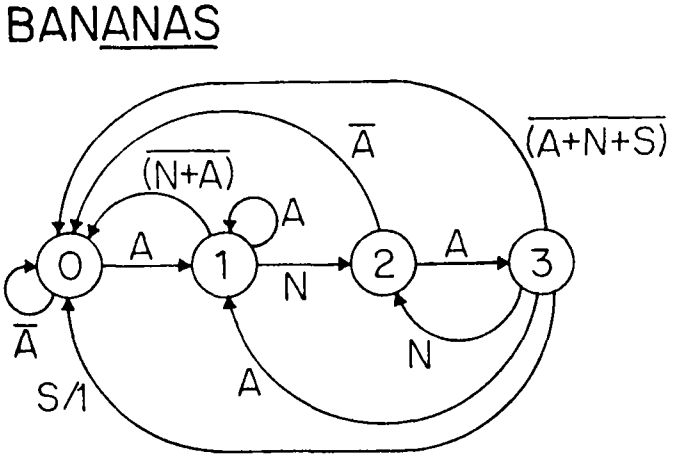


Figure 2

The OSI machine handles backtracking to the idle state by including special logic to automatically execute transitions to one state if the input is both a mismatch and a word separator, and another state on any other mismatch. If all terms start on word boundaries, most have only one match transition and can be held in sequential states. However, if even one term is not restricted to starting on a word boundary, all other states in the table must have backtrack transitions leading to that term, and thus must be index states. To circumvent this problem, the OSI machine assigns all such terms to a second, identical FSA. All states in the second FSA are index states, but most of the states in the main FSA will be sequential states. A similar problem exists with continuous word phrases (CWP's) - terms containing more than one word in sequence. Yet a third FSA handles these.

A few other details regarding the implementation bear mentioning. The state table memory chosen for the OSI matcher had an access time chosen to match the 'average sustained worst-case' state processing speed. The design called for relatively fast 100ns. bipolar memory. The state memory word is also quite wide (85 bits for the 6-bit character version and 39 bits for the 4-bit nibble version). Since the FSA must be replicated at least once per disk drive, it would be desirable to build it as a VLSI circuit. However, the wide memory word and the required fast cycle time complicate a VLSI implementation using present technology.

Nondeterministic Finite State Automaton

An alternative to the FSA-based machine is the nondeterministic FSA, or NFSA, which is a finite state machine that can be in more than one state at a given time. Such a machine is described in [Hask80a], and in more detail in [Hask80b]. In the standard FSA approach, it was necessary to include many extra states and

transitions to 'backtrack' after a mismatch and to handle terms with prefixes embedded within other terms in the table. Using the NFSA model, it is possible to circumvent this problem. One state is active for each term of which the start has been seen and for which a match is currently possible. This state table can be generated directly from the list of terms. In the simplest embodiment, each character in a term has one state in the table, and each state has one outbound transition to a successor corresponding to the next character in the term. Each transition is associated with a single token (either a specific character or a class of characters such as numerics or delimiters), and the next input character is compared against the token for each active state. If the character matches the token, the successor state is entered. As for the FSA, there is a distinguished 'idle' state that is handled slightly differently: it has an outbound transition corresponding to the start of each term, leading to the state for the first character in that term. To ensure that all startup transitions occur, the idle state is always active (one can think of the idle state as having a transition to itself labelled with a token matching any character). It should be emphasized that what is being described is a method of transforming search patterns into a class of nondeterministic state tables with certain desirable properties. The machine described in [Hask80a] handles this class of tables, but cannot handle arbitrary nondeterministic tables.

The obvious way to implement the NFSA would use one processor per active state, each interpreting a central state table. Aside from the obvious problems with memory contention and starting the processors, one other problem exists: what to do if there are more active states than there are processors to handle them. Fortunately, the solution to the latter suggests the solution to the others.

It is possible to define a compatibility relation between two states such

that they are compatible iff they cannot simultaneously be active. [Hask80b] gives rules for determining compatibility by inspection of the state table. For a given table, one can determine in advance the largest set I of mutually incompatible states. If $I \geq N$, the number of available processors, an arbitrary input string can be matched without ever having more active states than processors. However, one can take further advantage of compatibility testing. By partitioning the state table into blocks of mutually compatible states, it is possible to assign each block to a processor. Each processor matches only states from its block, thus there can only be one active state per processor. The partitioning also ensures that a processor will be inactive before a transition is taken to a state in its block from one not in the block (i.e. states in another block or the idle state). This allows processors to dispatch themselves. Finally, since the blocks of the partition are disjoint, each processor can be connected to a dedicated memory containing its block, eliminating state table memory contention.

[Hask80a] describes such a machine. It consists of a Match Controller, which acts as the interface between the processors and all external components, and the processors themselves, called Character Matchers (CM's). Each CM contains the comparison and sequencing logic necessary to perform state transitions, and also contains the state table memory for its block of the table. It contains a Startup Table, used to detect transitions from the idle state into states in its block, and a Fork Table, controlling transitions from states in its block to states in other blocks. The CM's are interconnected in a ring; this topology restricts inter-block state transactions to occur only between connected CM's.

Three obvious questions exist regarding the practicality of this architecture: the amount of computation required to partition the state table, the num-

ber of CM's required to match the desired number of terms, and whether the limited interconnectivity among CM's is a problem. Results reported in [Hask80b] show, for randomly chosen tables of terms, that the PDP-11 used could partition tables at the rate of about 10 ms. per term, that a table of N terms can almost always be partitioned among (N/16)+5 CM's, and that in only one out of 240 trials was it impossible to assign a table to the minimal number of CM's without violating the neighbor-connection restriction.

The main advantage of the NFSA over the FSA is that the NFSA is more amenable to a VLSI implementation. The state word is small, so internal data paths are narrow. The CM's have a limited memory requirement, small pinout, and regular interconnection topology. The design is expandable- a matcher for any term batch size can be built by configuring it with the right number of CM's. Finally, the required memory speed is achievable using MOS LSI (about 600 nsec. for a 1.2 MHz disk data rate). Physical layout of an NMOS CM is currently underway at the University of Utah under the direction of Prof. L. A. Hollaar, and while definitive results are not yet available, it appears that a CM for 8-bit characters with 128 states will occupy a chip area 180 mils on a side.

Summary

Architectures for text search processors have evolved greatly in the past several years. They are now to the point where it is feasible to build and market a hardware searcher, and in fact at least one company (Datafusion) is doing so. The availability of a small VLSI searcher would lower the cost to the point where they could be included in word processors and other personal computers. This would enable the pipe dream of an 'electronic file cabinet' to finally become a reality.

References

[BiNeTr78] Bird, R. M., Newsbaum, J. B., and Trefftzs, J. L., 'Text File Inversion: An Evaluation,' Proc. Fourth Non-Numeric Workshop, Syracuse, N. Y., Aug. 1978, pp. 42-50.

[Cop78] Copeland, G. P., 'String Storage and Searching for Database Applications: Implementation on the INDY Backend Kernel,' Proc. Fourth Non-Numeric Workshop, Syracuse, N. Y., Aug. 1978, pp. 8-17.

[FoKu80] Foster, M. J., and Kung, H. T., 'Design of Special Purpose VLSI Chips', Computer, Jan. 1980, Vol. 13, No. 1, (ISSN 0018-9162), pp. 26-40.

[Hask80a] Haskin, R. L., 'Hardware for Searching Very Large Text Databases,' Proc. Fifth Workshop on Computer Architecture for Non-Numeric Processing, Pacific Grove, Calif., Mar. 1980, pp. 49-56.

[Hask80b] Haskin, R. L., 'Hardware for Searching Very Large Text Databases,' Ph.D. Thesis, University of Illinois, Urbana, Aug. 1980.

[Hollaar76] Hollaar, L. A., 'An Architecture for the Efficient Combining of Linearly Ordered Lists,' Second Workshop on Computer Architecture for Non-Numeric Processing, Jan. 1976.

[Hua80] Huang, H. M., 'On the Design and Scheduling of an Index Processing System for Very Large Databases,' Ph.D. Thesis, University of Illinois, Urbana, Aug. 1980.

[Hurl76] Hurley, B. J., 'Analysis of Computer Architectures for Information Retrieval,' M. S. Thesis, University of Illinois, Urbana, May 1976.

[Miln76] Milner, J. M., 'An Analysis of Rotational Storage Access Scheduling in a Multiprogrammed Information Retrieval System,' Ph.D. Thesis, University of Illinois, Urbana, Sept. 1976.

[Muk78] Mukhopadhyay, A., 'Hardware Algorithms for Non-Numeric Computation,' Proc. Fifth Symposium on Computer Architecture, Palo Alto, Calif., Apr. 1978, pp. 8-16.

[MuWar79] Mules, D. W., and Warter, P. J., 'A String Matcher for an 'Electronic File Cabinet' Which Allows Errors and Other Approximate Matches,' Department of Electrical Engineering, University of Delaware, Newark, Apr. 1979.

[OSI77] 'High-Speed-Text-Search Design Contract Interim Report,' OSI:R77-002, Operating Systems Incorporated, Woodland Hills, Calif., Jan. 1977, pp. 2-69 - 2-101.

[Stell74] Stellhorn, W. H., 'A Processor for Direxts Scanning of Text,' presented at the First Nonnumeric Workshop, Dallas, Texas, Oct. 1974.

[Stell75] Stellhorn, W. H., 'A Specialized Computer for Information Retrieval,' Ph.D. Thesis, University of Illinois, Urbana, 1975.

How to Get Even with Database Conference Program Committees

Frank Manola

Computer Corporation of America
575 Technology Square
Cambridge, Mass. 02139

You say your last twelve papers submitted to database conferences have been rejected? You say your last paper came back with the referee's suggestion that you use it to line a bureau drawer? You say the paper before that was rejected by Computerworld as not being theoretical enough? Don't despair! This paper offers a number of sure-fire methods for driving those program committee members and referees who have been making your life miserable right up the wall! All of these methods have been tested in practice, and the author, who has served on a number of program committees and as referee for various publications and conferences, will vouch for their effectiveness. (Note: This paper is primarily intended for those who wish to get even by actually writing a paper. There are, of course, other methods of getting even. These are covered in other papers by the author [1,2,3]).

As you might expect, there are a number of aspects to be considered in getting even by submitting a paper. These are covered in the following sections, with the most important aspects first.

Submission of the Paper

The tactics for submission of "get even" papers are extremely important -- considerably more so than actually writing it (this also appears to be the case for many "serious" papers). Useful ideas on paper submission are:

1. Submit the complete text of your Master's thesis or Ph.D. dissertation to a conference, with no cuts whatsoever (the paper should be at least 65 pages long). Be sure to leave in the abstract the line about "submitted in partial fulfillment of the requirements for the degree of ..." to create the maximum amount of annoyance. In addition to the mental damage this may cause, the program committee members will have to carry the paper to the program committee meeting, and may rupture themselves.

2. As a followup to idea #1, submit a "revised version" (of equal length or preferably longer) just before the deadline, so that the referee will have to read both versions. Be sure not to indicate where the revisions are, so the referee will have to look for them. Also, make sure that any revisions you actually make are as trivial as possible, to ensure that the maximum amount of effort will be wasted in reading the "revised version".

3. Never submit your paper to one conference or publication at a time. Instead, submit your paper to several conferences and publications simultaneously. Try to see to it that some of the same people are on the various program committees and editorial boards involved. This has a number of potential benefits, including: (a) the paper may be accepted by several conferences or publications, making all concerned look like idiots; (b) even if the paper is rejected, you can force them to reject it several times; (c) the overlap of people involved increases the chance that they will realize what you've done, and thus be really irritated. (Note: In applying this idea, remember that your goal is to get even with these people, not to get your paper published. Thus, it's bad form to acutally change the form of the paper when you submit it to different places simultaneously. Really rub it in!)

Organization and Physical Aspects

The organization of your paper, and its physical appearance, can make an important contribution towards getting even. Important ideas to keep in mind concerning these aspects are:

4. Be sure that the copies of your paper submitted to the conference are as illegible as possible (running them through a copying machine several times with the "lighter copy" switch on is a good starting point). Include lots of hand-written equations and other comments. If possible, use non-standard paper so that copying is difficult. The goal here is to create as much physical effort for the program chairman as possible in creating and distributing copies of your paper, and as much eye strain as possible for the referees.

5. Continuing along the lines of idea #4, when stapling your paper, either (a) use weak staples so the paper falls apart if you look at it, or (b) use staples the size of the rivets on the George Washington Bridge so the paper can't be copied without the aid of a demolition team to open the staples. A side effect of the latter option is that the holes created may prevent the automatic feed on a copying machine from working (or, even better, may jam the machine!), thus adding to any copying effort involved.

6. Don't under any circumstances number the pages of your paper. Be sure, however, to use page numbers in the text of the paper. This forces the referee to number the pages, and try to maintain consistency with the text, always an annoying job.

7. Use as few diagrams as possible. Remember that each diagram potentially

reduces the length of the paper by 1000 words. If you must use diagrams, use the same type of figure (box, circle, etc.) and connector (arrows, etc.) throughout the diagram, whether the things represented are of the same type or not. Alternatively, use different types of figures and connectors without explaining what the differences mean. Put the diagrams at the back of the paper so that the referee has to turn back and forth while reading the paper. Another useful trick is to skip a number while numbering the figures, so that the referee can speculate as to what might have been on the omitted figure.

8. Throughout the paper, use sentences that are grammatical only according to the rules of a foreign language. Combine sentences that have fewer than 20 words apiece. To make reading the paper even more challenging, remove every other punctuation mark, and organize the entire paper into no more than six paragraphs.

9. Never proof read the paper. Force the referee to wade through your typographical errors. This is a particularly neat trick when the typo actually changes the meaning of the word involved.

Content

While obviously the least important consideration in a "get even" paper, nevertheless content can make an important contribution. Consider the following suggestions:

10. In the reference list, cite every paper you ever wrote, whether relevant or not. Cite several papers of yours which are "in progress" and appear highly relevant, but which you have no intention of actually writing (see the reference

list of this paper for examples). Also, cite "personal communications" from several prominent researchers in database management, whether you've ever communicated with them or not. For background or tutorial information on the particular system or approach you talk about in your paper, always refer to six or seven university technical reports, vendor's manuals, or other sources the referee probably won't have. Do not under any circumstances: (a) provide the necessary background as part of your paper, or (b) refer to papers in easily obtainable sources such as Communications of the ACM or IEEE Transactions. Remember that the idea is to fill the reference list with papers which are either irrelevant, unobtainable, or both.

11. Discuss a wide variety of topics in the same paper. For example, your paper might discuss abstract data types, VLSI, conceptual models, distributed databases, ADA, artificial intelligence, denotational semantics, and relational normalization theory. Suggest that there's an important relationship among these topics which your paper discusses (but don't actually discuss it), or which you assume the reader knows. Suggest in a footnote that Godel's proof and the Second Law of Themodynamics are also relevant.

12. Be as abstract as possible. Never say anything in English which can be stated as a formula. Never use concrete examples, but instead use x', y's, and so on (for example, never say something like "the EMPLOYEE relation is related to the DEPARTMENT relation"; say instead "relation A is related to relation B"). To add extra complexity, reverse a few inequalities (or arrowheads if you have diagrams) "by mistake". This can cause a great deal of confusion if the referee is not alert. (Use of strange notation, super-superscripts, etc. will be covered in my forthcoming paper entitled "How to Get Even with Publishers of Database

Conference Proceedings").

13. If your're writing a paper on a particular system, be sure not to include any information on the status of the work. An even neater trick is to say in the paper that the paper is about a project you haven't even done yet! You may cause the referee to have a stroke. You may also get a free design review from the referee.

14. In writing papers on Languages, always provide the format syntax of the language (preferably in an appendix, using a metalanguage no one has seen before). Omit describing the semantics, and never include any examples using the language. This forces the reader to try to deduce the meaning from the syntax, always an interesting exercise. (The opposite approach is also effective: provide only a few syntax fragments to illustrate the language, and make numerous outrageous claims for the capabilities of your language. The claims will then be impossible to refute).

15. If you have a very formal paper, be sure to omit any motivating, introductory, or concluding material. In particular, English text separating consecutive theorems is to be avoided. This forces the referee to try to determine what the paper is about.

16. Never introduce only one new idea in a paper; instead, introduce several new ideas which are interrelated. This creates the maximum amount of confusion. For example, suppose you are writing a paper on a new update protocol for distributed databases. Never simply describe the new protocol. Instead, embed the new protocol in a new DBMS architecture, and describe the new protocol using a

new abstract specification technique (which should, of course, be as formal as possible). In this way, it is impossible to understand any one of the ideas without understanding all of them!

Conclusion

The author believes that this paper has made an important contribution to the database literature by collecting in one place a number of ideas which previously have been applied only sporadically. The paper will be able to contribute numerous other ideas along the lines of those presented here, and welcomes such contributions. With our concerted efforts, we can turn the art of making referees and program committees miserable into a true science!

Acknowledgements

The author would like to thank Diane Smith and Ronnie Rosenberg, both of CCA, for their help in collecting the ideas discussed in this paper.

References

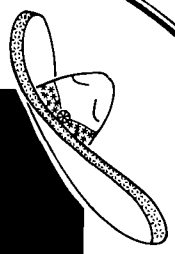
1. F. Manola, "How to Smuggle Your Unrefereed Paper into a Database Conference as a Panel Discussion Session", paper in progress, 1981.

2. F. Manola, "Mumbling and Assumed Foreign Accents in Database in Database Conference Presentations", paper in progress, 1981.

3. F. Manola, "Suggestions for Unreadable Visual Aids for use in Database Conferences", paper in progress, 1981.

CALL FOR PAPERS

1982 INTERNATIONAL CONFERENCE ON Very Large Data Bases



Mexico City, Mexico
8-10 September 1982

TOPICS

Technical papers are sought on topics including but not limited to:

- Database system design and evolution
- Distributed databases
- Database system interfaces
- Database control
- Database system implementation
- Structures for unconventional databases
- Applications

Proposals for possible panel topics are also requested.

HOW TO SUBMIT

- Papers should not exceed 5000 words in length.
- All papers must be submitted in English.
- A 100 word abstract must be included.
- Submit five (5) copies of paper and abstract.
- Deadline for receipt is 8 March 1982.
- Send submissions to the U.S. Program Chairperson:

DR. DENNIS MCLEOD
Computer Science Department
University of Southern California
SAL-200
Los Angeles, CA 90007

OTHER IMPORTANT INFORMATION

- Notification of acceptance will be mailed by first week of May 1982.
- Final drafts of papers are due by 14 June 1982.
- Accepted papers will be published in a Proceedings, available at the Conference.
- For further information on the Conference, please contact either:

U.S. CONFERENCE CHAIRPERSON
DR. VINCENT Y. LUM
IBM Research Laboratory
5600 Cottle Road
San Jose, CA 95193

LATIN AMERICAN CHAIRPERSON
DR. ALFONSO CARDENAS
Computer Science Department
Boelter Hall Room 3731
U.C.L.A.
Los Angeles, CA 90024

Sponsored by

IEEE COMPUTER SOCIETY

ASSOCIATION FOR COMPUTING MACHINERY