A database machine based on the data distribution approach

by YAHIKO KAMBAYASHI

Kyushu University*
Fukuoka, Japan

ABSTRACT

Various VLSI circuits, each of which realizes a specific database operation, have been studied; and a VLSI database machine can be created by a collection of these circuits. Such a method is called the function distribution approach. The problems of this approach are that (1) the data transmission cost is very high and (2) some circuits become very slow when the data size exceeds the maximum size handled by the circuits. Since database systems handle a large number of data, we need to develop another approach that costs less for data transmission and has expandability. Because most database operations can be divided into operations on subsets of data, this paper proposes the data distribution approach. In this approach a subset of data is stored in a functional storage circuit, and each circuit can realize most database operations. The whole system can be viewed as a file system having functions for database operations. Compared with conventional file systems, the system has the following advantages: (1) frequent rebalancing is not required, and (2) parallel processing of database operations is realized. Three methods to realize functional storage circuits are described. Selection is made by cost, performance, and available VLSI technology. An organization of such circuits with efficient database processing is discussed in detail; it will be realized by technology in the near future.

*This paper was written when the author was at Kyoto University.
MOTIVATION FOR THE RESEARCH

Pipeline processing is very effective in computers as well as factories (A).

In order to handle a large amount of data, it is economical to move processors instead of data (B).

Instead of moving processors, (B) can be equivalently realized by changing the functions of processors. This method seems to be suitable for databases, since basic operations are rather simple (C).

BACKGROUND

As a result of the recent development of VLSI technology, hardware realization of various functions has been studied by many authors. Such research is especially important in the areas of databases, picture processing, inference systems, and similar areas, where current computer systems do not offer enough efficiency. A typical system organization is as follows: A system consists of a number of hardware components, each of which can perform one or more specific operations efficiently. To perform an operation on data, the data are transmitted to the component that can perform the required operation. This method will be termed the function distribution approach. In this approach, the computation time is determined mainly by the communication cost. This paper will introduce the concept of the data distribution approach, which is especially suitable for database systems. Organization of the system in this approach is also discussed.

Database machines using associative disk devices and bubble devices have been studied by many authors. In the near future there will be a VLSI-based database machine. For this purpose various circuits for database operations, such as sort, search, select, and join are separately discussed. A VLSI database machine can be created through a collection of these circuits by taking the function distribution approach. The approach has the following problems.

To realize a required operation, data must be transmitted to the circuit that can perform the operation, and the result has to be transmitted to some circuit or storage. If the operation is binary, usually three units of data transmission (two for input and one for output) are required. Because the number of pins of each VLSI chip is limited, the cost of data transmission is \( O(n) \) for transmitting \( n \) data. The problem can be summarized as follows: (1) the data transmission cost is very high, and (2) there is a control problem in data transmission.

There are two possible approaches to handle (1): (a) operations can be performed during the data transmission so that the effective time for the transmission is reduced; (b) a suitable approach can be found at less communication cost. There are many circuits taking the first approach, such as the up-down sorter\(^{10}\) and the parallel enumeration sorter,\(^{25}\) which produce serial sorted data immediately after the end of the serial input operation. The data distribution approach is an example of (b).

In the data distribution approach, the data are partitioned into small sets, each of which can be handled by one circuit. Each circuit can perform most of required operations as well as the storage function. Data transmission among components can be reduced. Such an approach is suitable for operations having the following two properties: (1) each operation can be divided into a set of operations each of which requires a subset of the whole data, and (2) each operation is simple.
Database operations satisfy the above properties. Restriction and projection operations can be realized by collecting the result of these operations to the subsets of data. For sorting and searching, the bucket sort can be used. For each bucket the upper and lower bounds of data to be sorted are determined (usually the intervals for buckets are disjoint), and buckets are numbered in increasing order. Then sort and search of the whole data can be realized by sort and search for each bucket that corresponds to one component circuit.

The next section compares the function distribution approach with the data distribution approach introduced in this paper for realizing database machines. In the following section, definitions of relational operations are given. The next section discusses the organization of component circuits for database machines designed under the data distribution approach. The final section discusses one possible circuit configuration for realizing component circuits by using VLSI chips. A good database machine should be realized by a proper combination of the function distribution approach and the data distribution approach.

APPROACHES TO REALIZING DATABASE MACHINES

First we will summarize the approaches and the problems for database machine realization. In the function distribution approach,

1. Each component is designed to perform one function or a set of related functions.
2. The system consists of a storage and component circuits discussed above.
3. Data usually reside in the storage. When a specific operation is required, the data to be processed are transmitted to the component circuit that can perform the operation. The result and the data are transmitted to the storage or to other components for further processing.

In the data distribution approach,

1. Each component circuit can store data, and it realizes most required operations.
2. All data are divided into each component circuit so that there is a simple procedure for realizing an operation on all the data by performing corresponding operations at each component circuit separately. Figure 1 shows the organization of systems by these two approaches.

The properties of database operations are as follows:

1. Each operation is rather simple.
2. Each operation can be realized by operations applied to subsets of data.
3. The number of data to be handled by each operation varies from very small to very large.
4. Even if the volume of data is very large, usually, a query or a modification operation needs only a portion of the data, selected by some specified criteria.

Figure 1—Approaches to developing database machines

5. There are operations like join, sort, and search, which are time-consuming.

If the data distribution approach is taken most operations can be realized at a reduced transmission cost. As shown above, database operations are suitable for this approach. Previously known approaches are as follows: (1) a system consisting of components each of which can store at least one relation and in each of which database operations are performed; (2) the function distribution approach.

The first approach is usually used by intelligent disk-based systems. The data distribution approach can handle cases in which the number of data is large or component size is small, which occurs when VLSI circuits are used to realize each component.

Requirements for each component circuits are as follows:

1. Since the hardware approach is better than the software approach for a large number, n, of data, a circuit must be designed that can handle as many data as possible. A circuit requiring an area of $O(n^p)$ ($p > 2$) does not seem to be practical for replacing software.
2. The computation time should be determined by the volume of data, not by the circuit size.
3. A proper and efficient method must exist for handling a set of data whose number exceeds the maximum limit of the circuit capability.

There are hardware methods whose computation time is determined by the maximum number of data processed by the circuit. For example, the time required by a high-speed 64-bit multiplier is usually almost fixed, even if the inputs are 5-bit numbers. For arithmetic operations this problem is not serious, since the ratio of the most frequently used data volume to the maximum volume handled by the circuit is usually not high. As in database systems, when the volume of data varies very widely, the problem is serious. When the processing time is $O(m)$ or more ($m$ is the maximum number of data handled by the circuit), the circuit is very inefficient if the data size $n$ is much smaller than $m$. The problem is not so serious for circuits requiring $O(\log n)$ or $O(\sqrt{n})$ processing time.

For example, the joint circuit realized by the systolic approach always requires time determined by the circuit size, even if the data size is very small. For joining large relations a large circuit is needed, since if the relation size exceeds the bound determined by the hardware size, efficiency decreases.
very much. If we use a large circuit, however, we need a fixed amount of time to process relations, even if it is a small amount. Pipeline processing is very important in order to improve efficiency when the same operations are used repeatedly. The approach, however, increases the complexity of the data transmission control. In this approach an attempt will be made to increase efficiency by simultaneous processing of each component circuit, i.e., parallel processing.

BASIC OPERATIONS OF RELATIONAL DATABASES

A relation $R$ is defined as a finite set of tuples, each of which is a combination of domain values for the attribute set $R$, called a database schema. Figure 2(a) shows a relation STUDENT. NAME and DEPT are attributes, and STUDENT $= \{\text{NAME, DEPT}\}$. There are three tuples in STUDENT. The first tuple (Anderson, Computer Science) shows that Anderson studies at the computer science department.

For a tuple $t$ in $R$, $t[X]$ denotes the part of $t$ containing only values of attributes in $X(X \subseteq R)$. The following notations are used for basic relational operations.

**Projection:** $R[X] = \{t[X] | t \in R\}$

**Restriction:** $R[X \subseteq C] = \{t | t[X] \subseteq C, t \in R\}$

**θ-Join:** $R_1[X_1 \theta X_2]R_2 = \{t_1t_2 | t_1[X_1] \theta t_2[X_2], t_1 \in R_1, t_2 \in R_2\}$

Here, $X \subseteq R$, $X_1 \subseteq R_1$, $X_2 \subseteq R_2$, $C$ is a vector of constants and $\theta$ is a comparison operator ($=,$ $<$, $>$, etc.).

Projection of $R$ on $X$ is obtained by removing all attributes not in $X$. $R_3$ in Figure 2(d) is obtained by a projection from $R_1$ in Figure 2(c).

$$R_3 = R_1[\text{DEPT, BLDG}]$$

The restriction $R[X \subseteq C]$ shows the subrelation of $R$ consisting of tuples satisfying $X \subseteq C$. LOCATION and $R_2$ in Figures 2(b) and 2(d) have the following relationship:

$$R_2 = \text{LOCATION}[\text{BLDG} = \text{A}]$$

$R_1$ in Figure 2(c) is obtained by joining the two relations STUDENT and LOCATION in Figures 2(a) and 2(b).

$$R_1 = \text{STUDENT}[\text{DEPT} = \text{DEPT}] \text{ LOCATION}$$

Since the result of the join contains two identical columns, one of them is omitted. Such a join is called a natural join.

For two relations $R_1$ and $R_2$ defined on the same attribute set ($R_1 = R_2$), set operations can be defined. $R_1 \cup R_2$ is a relation consisting of all tuples in $R_1$ and $R_2$. $R_1 \cap R_2$ and $R_1 - R_2$ are also defined similarly.

Division is also known as a relational operator, which can be expressed by a combination of other operations.

There are aggregate functions, such as count, sum, and ave (average). The result of count is the number of different values. For example, $\text{COUNT(LOCATION[BUILDING])} = \text{COUNT}(\{\text{B, A}\}) = 2$. Sum takes the summation of values, and ave calculates average values.

Since contents in a relation can change, update operations, such as add, delete, and modify, are needed. In these operations, tuples are added, deleted, and modified (i.e., a part of a tuple is changed).

For efficient processing of some of the above operations, operations such as sort and search are needed. These operations are summarized in Figure 3, which includes operations not discussed above. This paper will discuss VLSI circuits to perform these operations effectively.

**Basic relational operations**
- Projection, Selection, Join, Division

**Set operations**
- Union, Intersection, Difference, Direct product

**Aggregate functions**
- Count, Sum, Average

**Update operations**
- Add, Delete, Modify

**Sort and search**
- Sort, Direct search, Sequential search

Figure 3—Major operations of databases
DESIGN OF A DATABASE MACHINE BY THE DATA DISTRIBUTION APPROACH

This section will discuss the organization of database machines based on the data distribution approach. Since the system will be realized by VLSI circuits, we assume that there are relations that cannot be contained in one component. Such a relation is divided into sets of tuples, so that each set can be stored in one component circuit called a functional storage circuit.

Before the organization of each functional storage circuit is discussed, methods to realize database operations will be discussed. They are classified as follows:

1. Operations that can be realized by local processing only: projection, selection, search, update.
2. Operations that can be realized by local processing and simple global processing: count, sum, average.
3. Operations that require data transmission among functional storage circuits: join, intersection, difference.
4. Operations that require reloading of the whole data: sort, division.

It is obvious that the projection, selection, search, and update operations can be realized at functional storage circuits. For aggregate functions, simple arithmetic operations on the results obtained by functional storage circuits are needed. To perform a join (or intersection, difference) operation on two relations stored in two sets of component circuits, joins must be realized on all possible combinations of the contents of two functional storage circuits (one from each relation). For sorted data the number of possible combinations will be reduced. To reduce the cost of operations in 3 and 4 above, we will use the bucket sort.

For each functional storage circuit the upper and lower bounds of sort key values are determined. For example, the first functional storage circuit stores tuples whose key values are contained in the intervals \([A,B]\), and the second functional storage circuit stores tuples in the interval \([C,D]\). In this case the tuple whose key value starts from \(D\) should be stored in the second functional storage circuit. We assume that a unique order number and a key interval are assigned to each functional storage circuit. These values are stored in the index circuit as shown in Figure 4. The following condition is satisfied by order \(o(s)\) and interval \(i(s)\) for functional storage circuit \(s\).

For any functional storages \(s\) and \(t\),

\[ j \leq k \text{ if } o(s) < o(t), \quad j \in i(s), \quad k \in i(t). \]

Since tuples are sorted in each functional storage circuit, all the tuples in the relation are sorted by retrieving functional storage circuits according to the ascending order of \(o(s)\).

Sorting of \(n\) data can be realized by \(O(n)\) steps by the above system. As shown below, overflow of a bucket will usually not increase the number of steps. Division can also be realized by sorting. For example, \(R(A,B) + S(B)\) can be realized by sorting by \(A\).
o(s) and i(s) values. Reorganization of the contents of the index circuit is not required as frequently as conventional file systems, since overflow tuples can usually be handled without reorganization (as discussed above).

**Parallel processing**

The index circuit can be duplicated in order to increase efficiency. If there are k indices, the expected sorting time becomes at most k times faster than one index case. Versions of such parallel bucket sorts are discussed by several authors. Most of the other operations can also be done at each functional storage circuit in parallel. For example, searching tuples satisfying some condition determined by values of each tuple can be realized at each functional storage circuit independently.

**Operations**

Various functions discussed in the previous section can be realized. In some cases a combination of operations can be realized by the maximum processing time required by each of these operations.

A merger is used to generate one sorted sequence from a set of sorted subsequences. When these subsequences are given in ascending order, the merger always takes the tuple with the smallest key value among tuples at the top of subsequences. Figure 5 shows an example of the merging of two subsequences. One application of the merger is discussed above. Another application is to sort by values different from the key. When such sorting is required, sort is first performed at each functional storage circuit, and then merging of these results is performed.

Join of two relations sorted in functional storages can be realized by the method discussed by Merrett et al.

For functional storage circuits, the structure shown in Figure 7 will be used. It consists of a tree part and a linear part, for the following reasons:

1. To store n data, we need 2n memory cells, since the original data in the functional storage circuit should be kept during an operation and an intermediate result must be also stored.

2. Since we need 2n memory cells, we will use a tree consisting of n cells and a linear arrangement of n cells, since most operations are suitable for one of the two structures.

This structure is also suitable to realize index circuits. The following methods realize index circuits and function storage circuits (Figure 6): (a) use of a microprocessor system, (b) use of a tree realized by assigning one processor to each level, and (c) use of storage cells integrated with processors.

Methods (a) and (b) can be realized by current technology. To realize a tree having n nodes, Method (a) requires only one processor, Method (b) requires O(\log n) processors, and Method (c) requires O(n) processing elements. Realization of a tree by log n processors is used for sorting circuits. The Method (a) is the most economical, and Method (c) realizes the fastest operations. The selection is determined by cost, performance, and available technology.

Method (a), shown in Figure 6(a), is simple; but the amount of communication between the processor and memory is large, which reduces the speed. In sorting of data, only one of the cells in each level is active at a time. Method (b) uses one processor to each level, which improves performance remarkably compared with method (a). Method (b), however, has the following problems:

1. Each processor has a different number of memory cells. This fact makes the embedding on the VLSI chip difficult. The processing time required for a processor with a larger number of memory cells is longer because of the address decoding time. It may cause a problem to synchronize all the processors.

2. There are still data communications between a processor and its memory, and it will make the system not so fast as Method (c).

Figure 7 shows the organization by the Method (c), where each rectangle corresponds to a storage with some processing capability.

Although by current technology Method (b) is the best choice, we will discuss the organization of the circuits by Method (c) in the next section. The author believes that such circuits can be realized in the near future.
ORGANIZATION OF A VLSI FUNCTIONAL STORAGE CIRCUITS

In this section the VLSI functional storage organized by Method (c) of the previous section is called a functional storage for short.

Fig. 7(a) shows a basic organization for a functional storage proposed in this paper for data size \( n = 7 \). We assume that each cell can store one tuple. If the given relation has more than \( n \) tuples, it is distributed to more than one functional storage.

It consists of a tree part and a shift register part. To handle \( n \) tuples there are \( 2n \) storage cells and \( O(n) \) connections; thus the circuit consists of \( O(n) \) elements. Since the height of the tree is \( O(\log n) \), the area required for the circuit is \( O(n \log n) \), although the coefficient part can be minimized by a proper embedding of the circuit (see Figure 7(b)).

We will show how database operations are realized by a functional storage using very simple examples (\( n = 7 \)). In the following, \( T \) and \( S \) stand for a tree and a shift register, respectively, which show the part mainly used by the operation. There are two input/output terminals for the circuit. The terminal for the shift register part is denoted by \( I/O(S) \), and the terminal for the tree part is denoted by \( I/O(T) \). To use the tree part for fast access of data, data should be arranged in ascending or descending order. In the following examples, the ascending order is used for simplicity.

Initial data loading (S): A sequence of data is supplied from \( I/O(S) \). The sequence starts from \( L \) (loading) and ends at \( E \) (end of data). The data sequence can contain \( B \) (blank). Figure 8 shows an example when 9B7531 is supplied. The first \( L \) sets the operation of each shift register cell so that only shift operation is realized. After six steps we have the situation shown in Fig. 8(b). Here \( E \) is supplied from the input terminal. In this case, instead of the data's being shifted, the con-

![Diagram](image-url)
The contents of the shift register part are copied by corresponding tree nodes and \( L \) is replaced by \( B \). The result is shown in Figure 8(c). If the data size is 7, \( L \) is shifted out; and if the data size is over 7, a proper warning signal is created. The initial loading can be used instead of the sort in the following cases:

1. Put blanks in the sequence of data in order to handle the increase of data easily.
2. The data are required not to be sorted. For unsorted data we cannot usually use the tree part for an efficient search. It can be used when the data are clustered by related key values, etc.

Single tuple retrieval (T): If we want to retrieve the data whose key value is 5 in Figure 8(c), we put 5 from I/O(T). This value is compared with the contents of the node. Since 5 is smaller than 7, the left son, the node containing 3, is examined; 5 is larger than 3, and the right son is examined, which contains a tuple whose key value is 5. The tuple is retrieved by traversing the path in the opposite direction (see Figure 8(d)). The changed data on the path are recovered by copying data from the corresponding shift register cells.

Multiple data retrieval (S): If all the tuples are required, the output terminal for the shift register is used. After shifting out (Figure 8(e)), all shift register cells become empty. The data are recovered by copying values contained in the tree nodes.

Replacement of blanks (T,S): If a blank node is a parent of a nonblank node in the tree part, the search mechanism of the tree will not work. Such a tree is called improper. An improper tree may be produced by addition or deletion of a tuple. In a proper tree every subtree must satisfy the condition that every node of the subtree is blank if the root of the subtree is blank. There are two methods of handling the problem:

1. Exchange of blank and nonblank values at the tree part: By a proper exchange of values, an improper tree is converted into a proper tree. Figure 9(c) shows an exam-

![Diagram](image_url)
ple of an improper tree. In this case exchange of data between the blank node and one of its sons generates a proper tree (Figure 9(d)).

2. Blank suppression (S): The shift register part can be used to remove all blanks contained in the sequence by shifting nonblank values to fill blank values. For the circuit shown in Figure 9(a), the situation shown in Fig. 9(e) is obtained. By copying values in shift register cells, blanks in the trees are erased except blanks at the right side end.

Addition of one tuple (T,S): When a tuple whose key is 8 must be inserted in the circuit in Figure 8(c), just apply the same operation as single-tuple retrieval; since the left son of 9 is B, the tuple is stored here. When there is no blank cell, we must use the shift register part, as shown in Figure 9(a), (b), (c), and (d).

Deletion of a tuple (S): Deletion of a tuple is very easy, since we need only replace it by blank symbol B, then remove B by using the shift register cells.

Sort (S,T): Sorting is realized by a hardware version of the bubble sort. Tuples are given from the I/O(S), and larger values are shifted to the right. In order to perform sort, the sequence starts from S (Sort). When S passes in the cell, cell operation becomes as shown in Figure 10. If the key value sorted in a shift register cell is a and the corresponding tree node stores b, the new values for the tree node and the shift register cell to the right are c and d, respectively, where

\[ c = \min(a, b) \]
\[ d = \max(a, b) \]

for descending order.

S is considered to be larger than blank, and blank is considered to be larger than any value.

Figure 11 shows an example when 63714 is an input. Shift register cells are initialized by S, which contains (1) the definition of the key and (2) the definition of the ordering, ascending or descending. In Figure 11(g), S is shifted out. In Figure 11(k) all the tuples are sorted at the tree part. In Figure 11(l) values in tree nodes are duplicated, and the whole result can be sequentially retrieved from I/O(S). Other operations can be also applied to the result.

The circuit can simulate the operation of the up-down counter developed by Lee et al. and Kikuno et al. The advantage of the counter is that after the input is finished we can start to get the sorted result, although the result does not remain in the circuit. Any time after Figure 11(f) we can start to get the output. Figure 12 shows the case in which the retrieval operation starts from Figure 11(g). First, values in tree cells and shift register cells are exchanged. Figure 13 shows a basic operation, where c and d satisfy the same condition as Figure 10. Details of the operation are omitted here.

Addition of tuples, merge (S,T): By using the sorting function, a set of tuples can be added very easily. This operation is the merge operation for sorted tuples.

Deletion of tuples, set subtraction (S): A sequence of tuples to be deleted is given from I/O(S). The top of the sequence is D (delete) in order to set the cell operation, and the last symbol is E. These tuples are shifted to the right, and the sequence is examined to determine whether the values at a shift register cell and the corresponding tree cell are equivalent. If they are, the value in the tree node is replaced by B (blank). The blank removal operation is applied after the deletion.

Intersection (S): Intersection is almost the same as deletion, except that tuples replaced by Bs are the results of intersection. Each storage cell contains a tuple and a binary value to indicate the result. The binary values of all cells are initially 0. Let S₁ be the set stored in the functional storage and S₂ be the set given from the outside. The input is given from I/O(S). The sequence of tuples in S₂ starts from I (intersection) and ends at E. I contains the information on which part of the tuples is to be compared. Tuples in S₂ are shifted to the right, and at each step, values at each tree cell and the corresponding shift register cells are examined. If these are equivalent, the binary values at both cells are set to 1. After E passes the cell corresponding to the rightmost tree cell containing a tuple, the results is obtained as binary values. The binary value for a tuple in S₁ ∩ S₂ is 1. By the above method we require that \( n > |S₁| + |S₂| \). Another method requiring \( n \geq \max(|S₁|, |S₂|) \) is as follows. After all values in S₂ are given to the functional storage, tree cell values and shift register cell values are exchanged. Then values in shift registers are shifted to the left. At the terminal I/O(S) the binary values are examined, and tuples whose binary values are 0 are erased. In this case the result is shifted out from I/O(S). This method can be also used for deletion. In this case tuples in S₁ that are not in S₁ ∩ S₂ can be detected.

Join (S): It is known that any query can be converted into tree queries. For a tree query there is an efficient procedure for joins using semijoins. The basic operation of a semijoin is intersection of two sets contained in the join attributes. Thus the above intersection procedure can be used for semijoin. We assume that S₁ and S₂ are stored in two different functional storages and the intersection is performed by the functional storage containing S₁. The result must be transmitted to the functional storage containing S₂. Since S₂ is stored in the functional storage, we need only transmit the binary values for
Figure 11—An example of sorting
S₂ that indicate the result. In this way the cost of data transmission can be reduced.

Pseudo-operations and composite operations (T,S): By using binary values we can indicate tuples satisfying some conditions. More than one condition can be indicated by permitting more than one binary value for each cell. Operations that do not change tuples are called pseudo-operations. Binary values can be used to realize more than one operation. For example, sorting and intersection can be realized by modifying the sorting operation.

There are two modes in the functional storage. The first mode keeps tuples with the same key values, and the second mode erases duplicated keys. We can also specify the first key, second key, etc., for sorting tuples.

In Section 2 we discussed the facts that (1) if the computation time is O(log n), it is not serious, even if n is the circuit size; and (2) if the computation time is O(n), n should be the data volume and not the circuit size. Functional storage satisfies these conditions. For operations using the tree part, the computation is O(log m) where m is the circuit size. For operations using the shift register part, the computation time is O(n) where n is the data size. Functional storage can also be used as an index circuit, as discussed in the previous section.

We can further generalize functional storage in order to improve efficiency by adding (1) a bus line for the shift register part, (2) a calculation capability to the tree part (aggregate functions can be realized) and (3) fast internal sort capability.
ACKNOWLEDGMENT

The author is grateful to Professor Shuzo Yajima, Mr. Hiroto Yasuura, and Mr. Naofumi Takagi for their discussion. The work is supported in part by a grant from the Ministry of Education of Japan.

REFERENCES

Artificial intelligence

James R. Miller, Track Chair

The 1980s have seen artificial intelligence (AI) moving out of the laboratory and into the marketplace. This movement has been especially clear in 1984: Practical systems are in use daily on a wide range of hardware and in such diverse domains as database retrieval, VLSI chip design, geological exploration, and computer system configuration. At the same time, the topics under exploration in research laboratories are building a better understanding of what would be required to build increasingly powerful and intelligent computer systems. The six sessions in this track bring a survey of all these areas to NCC.

AI and the computer industry — The purpose of two sessions in this track is to consider how AI is changing the computer industry itself. Part of this focus is addressed in the session “Expert Systems in the Computing Industry,” which examines how expert system technology—the development of computer systems that capture some significant part of human expertise in a complex technical domain—is being applied to problems central to the computer science community. To provide a balanced perspective on this topic, the presentations in this session discuss these systems from the perspective of both the system builder and the end user.

The last few years have also seen the growth of companies developing tools for building AI systems. The session on “Tools for Commercial AI Systems” describes tools under development in two areas: computers and programming environments that are especially well-suited for AI system development; and high-level software tools designed to relieve system developers from much of the effort of building the basic architecture of the AI system, allowing them to focus on the problem at hand. As in the previous session, the presentations will discuss these tools from the perspectives of system builders as well as users.

AI application areas — Two sessions have been designed to take a careful look at application areas that have been studied increasingly by the AI community. These sessions are intended to show how artificial intelligence workers approach a problem, what aspects of problems are easy and hard, and what results might be expected in the short term and in the long term. One of these, “Knowledge-Based Training Systems,” examines a number of systems that are applying AI techniques to problems in training and education—what a system must know about the domain being taught, about the educational process, and about the student in order to truly help the student acquire a body of knowledge. The second, “AI Techniques for Signal Interpretation,” considers how AI techniques are being applied to problems that have traditionally been attacked by complex numerical methods, such as seismic exploration and speech signals. This approach requires transforming the basic signals under analysis to a symbolic representation of the phenomena responsible for those signals. While constructing this representation is not a simple task, its richness allows much more powerful analyses to take place and provides a depth of analysis not possible before.

AI and natural language — A final pair of sessions focuses on one of the oldest dreams of human-computer interaction—being able to communicate with a computer in natural language. While most natural-language understanding systems to date have been designed to provide a convenient interface to a structured database system, the presentations in the session “Natural-Language Interfaces to Software Systems” consider the use of natural language for a wide range of purposes. The second session, “Intelligent Aids to Document Preparation,” discusses the use of computer systems in one of the most time-consuming parts of any job—the generation of documents. Both general-purpose and domain-specific systems are being developed to this end, and the presentations focus on the underlying structure of these systems, how they are used, and what they can achieve.