John F. Sowa

Conceptual Graphs for a Data Base Interface

Abstract: A data base system that supports natural language queries is not really natural if it requires the user to know how the data are represented. This paper defines a formalism, called conceptual graphs, that can describe data according to the user's view and access data according to the system's view. In addition, the graphs can represent functional dependencies in the data base and support inferences and computations that are not explicit in the initial query.

Introduction

Historically, data base systems evolved as generalized access methods. They addressed the narrow issue of enabling independent programs to cooperate in accessing the same data. As a result, most data base systems emphasize the questions of how data may be stored or accessed, but they ignore the questions of what the data base means to the people who use it or how it relates to the overall operations of a business enterprise. When a business converts from a manual system to a computerized system, the computer cannot adapt itself to the users' view of the world, and the people have to learn strange conventions to access their familiar data.

Before a computer can adapt itself to a person's world view, that view must be described in a formalism that the computer can process. The conceptual graphs defined in this paper provide a formal notation that serves as an intermediary between the human and the computer: the graphs describe the meaning of data according to the user's view, but they are also associated with procedures that can access the data according to the machine view. When a person asks a question in ordinary English or other natural language, the system would translate the question into a conceptual graph. Then the system could search for other graphs that describe the data base and are relevant to the original question. When it finds such graphs, it can use them to access the data and compute the answer.

Conceptual graphs are not intended as a means of storing data but as a means of describing data and the interrelationships. As a method of formal description, they have three principal advantages: First, they can support a direct mapping onto a relational data base as defined by Codd [1]; second, they can be used as a semantic basis for natural language; and third, they can support automatic inferences to compute relationships that are not explicitly mentioned. The first point has been shown by the TORUS system at the University of Toronto [2], which uses a representation similar to the one developed here. The second point has been investigated in a growing body of research in computational linguistics and artificial intelligence; for a survey of that work, see the article by Heidorn [3] and the collections of papers edited by Schank and Colby [4] or Bobrow and Collins [5]. The third point is the principal topic of this paper: Besides representing logical relations in a conceptual graph, the system must use the graphs to perform inferences that answer the original question.

Since relational data bases have a simpler logical structure than network or hierarchical systems, they are an important first step toward simplifying the user's interface. Relations are a good interface for a professional programmer, and they can also be used by nonprogrammers who are familiar with the data base conventions. Several query languages, such as SEQUEL [6], SQUARE [7], and Query-by-Example [8], have been designed for nonprogrammers who have been trained in using the data base. But casual users and even programmers who had not learned the conventions would require a considerable period of training before they could ask a question.

Figure 1 shows a sample relation in a form that might be presented to a user of a relational query language. The name of the relation is HIRE, and its domains are named EMPLOYEE, MANAGER, and DATE. Under the domain headings are the n-tuples for which the relation is true. A person familiar with the real world system could probably guess that the three domains represent an employee, the manager who originally hired the employee, and the date the employee was hired. But there is no information in the relation that excludes other interpretations, such as, for each manger, the date he first became a manger and the first employee he hired. For a

complex data base with dozens of relations, few users can correctly guess the meaning of every domain in every relation.

The meaning of a relation is called its *intension*, and the set of all the *n*-tuples stored in the data base is called its *extension*. The question of representing extensions, accessing them, and modifying them is the familiar one that all data base systems address. The question of representing intensions, however, tends to be ignored, largely because adequate formalisms and techniques for handling them have not been available. For a data base system, the three principal kinds of intensional information are the functional dependencies, the domain roles, and the constraints on domain values. In a data base relation, functional dependencies indicate which domains are permissible keys and which domains are dependent upon the keys; for the relation in Fig. 1, EMPLOYEE is the key domain, and the domains MANAGER and DATE are determined when EMPLOYEE is specified. The domain roles indicate how the domains are related; for Fig. 1, the MANGER of each *n*-tuple performs an act, HIRE, the EMPLOYEE is the one who is hired, and the DATE is when the particular act occurred. The constraints indicate permissible values; for Fig. 1, they would specify the expected form of a name or date, the requirement that no date of hire may precede the date the company was founded, and the constraint that no person may hire himself.

Besides representing intensions, the system must use them to provide a more natural interface and to check the plausibility of new information that is being added. This paper defines conceptual graphs as an intensional formalism and shows how they might be used to meet the following requirements:

1. *Familiar conventions* A person who knows the forms and procedures of a business enterprise should be able to ask questions about it without having to learn the peculiarities of the computer system.
2. *Automatic inference* The system should infer relations that are not stored explicitly in the data base.
3. *Naturalness* The intensional formalism should be close enough to the semantics of natural language to support convenient dialogue and prompting facilities.
4. *Semantic integrity* The domain constraints should help to keep the data base an accurate reflection of the real world.

These are requirements for the user's interface; the physical implementation must also satisfy other criteria, such as speed and reliability. By separating the conceptual graphs that describe the meaning of data from the system that stores and accesses data, the two problems can be addressed independently. Efficient storage allocation or means of recovery after a system crash must be supported by the underlying data base system; what the data mean should be described by an intensional formalism at the interface to the data.

HIRE	EMPLOYEE	MANAGER	DATE
	Robert Lee	John Brown	3/1/74
	Mary Smith	John Brown	9/7/70
	Tom Jones	Mary Smith	8/8/75

Figure 1 The HIRE relation.

Figure 2 Examples of concepts.

To meet these requirements, conceptual graphs are a network of concepts and conceptual relations that describe the domain roles. Certain conceptual graphs, the conceptual schemata and working graphs, have a superimposed network of functional dependencies that are mapped to the data base. To answer a user's question, the system assembles a working graph that has the appropriate domain roles together with functional dependencies that determine the answer. The next several sections of this paper define these structures formally, present an algorithm for computing the working graphs, and give an example of how the system would process a typical question.

Conceptual graphs

In the theory of conceptual graphs, the basic primitive is called a *concept*. It is represented by a box containing a *sort label*, which identifies the type of concept. For readability, sort labels are written as English words in upper case letters, but they could just as well be numbers or computer addresses.

Formally, a concept is an undefined primitive. Informally, it is a symbol that could represent anything that anyone might ever think of—an entity, action, or property in the real world, an abstraction, fantasy, or mathematical function. Some concepts are shown in Fig. 2. For those aspects of the world recorded in the data base, at least one concept is defined for every data base domain. Some concepts are more general than others: the sort label PERSON marks a more general concept than EMPLOYEE, and EMPLOYEE is more general than MANAGER. To represent the levels of generality, the sort labels are ordered.

Definition There is a set S whose members are called *sort labels*, with a partial ordering $\leq$ defined upon S. The function *sort* maps concepts into sort labels. If a and b are concepts for which $sort\ (a) \leq sort\ (b)$, then a is said to be a *subsort* of b.

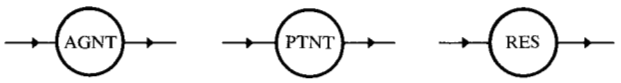

Figure 3 Examples of conceptual relations.

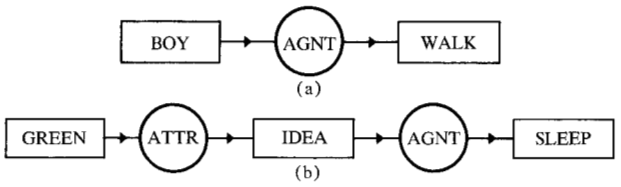

Figure 4 The conceptual graph (a) is well-formed while the graph at (b) is ill-formed.

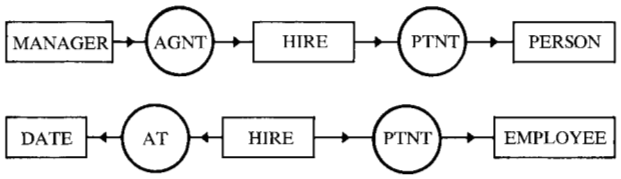

Figure 5 Two well-formed conceptual graphs.

Since a concept is not a set, one concept cannot be a subset of another concept. Yet subsorts and subsets are closely related: If a is a subsort of b, as the concept EMPLOYEE is a subsort of the concept PERSON, then the set of all things to which a applies is a subset of the things to which b applies. Since a concept is an intensional symbol, not an extensional set of things, the principle of extensionality does not hold: Two concepts with different sort labels are distinct, even if they represent exactly the same things in the data base. Even if every person mentioned in the data base happens to be an employee, the meaning of employee includes additional relationships beyond those that are true for persons in general. Following are some examples of the ordering of sort labels:

$$\text{MANAGER} < \text{EMPLOYEE} < \text{PERSON} < \text{ANIMAL} < \text{ENTITY}$$
$$\text{HIRE} < \text{ACT} < \text{EVENT}, \quad \text{DATE} < \text{TIME}$$

Ordering symbols other than $\leq$ are defined in the obvious way; i.e., $x < y$ if and only if $x \leq y$ and $x \neq y$.

Definition The concept c is called a *common subsort* of the concepts a and b if $sort\ (c) \leq sort\ (a)$ and $sort\ (c) \leq sort\ (b)$.

FIXED-BINARY is a common subsort of FIXED and BINARY. Two concepts may have many common subsorts: LION, TIGER, and JAGUAR are all common subsorts of FELINE and WILD-ANIMAL. Not all pairs of concepts have common subsorts: NUMBER and EMPLOYEE have no common subsort [9].

Connections between concepts are represented by *conceptual relations*, which are written as labeled circles having one or more links. The links are numbered consecutively, starting with 1; for the special case of dyadic conceptual relations, an arrowhead pointing towards the circle indicates link 1, and an arrowhead pointing away indicates link 2.

The examples in Fig. 3 show some common conceptual relations. The relations AGNT and PTNT have been adapted from case grammar [10]: AGNT (or AGENT) links a concept representing an animate entity to a concept of an action that the entity is performing; PTNT (or PATIENT) links an action to an entity that is being acted upon. Besides linguistic cases, conceptual relations can represent mathematical or computational notions: RES (or RESULT) links a concept representing a function to a concept representing the result of the function. As with sort labels, the choice of labels for conceptual relations has no formal significance in the theory, but a readable set should be chosen for a given application [11].

Definition A *conceptual graph* is a finite, connected, undirected, bipartite graph with nodes of one type called *concepts* and nodes of the other type called *conceptual relations*. A conceptual graph may consist of a single concept, but it cannot have conceptual relations with unattached links.

In the operations defined upon conceptual graphs, links may be attached or detached from concepts, but they are permanently bound to conceptual relations. Only dyadic conceptual relations are used in the examples in this paper, but the definitions and theorems allow arbitrarily many links.

Definition A conceptual relation has a certain number of *links*, which may be attached to concepts. If a conceptual relation has n links for some integer $n \geq 1$, it is called *n-adic*, and its links are numbered $1, \cdots, n$.

Formation rules

Not all combinations of concepts and conceptual relations are meaningful; the data base designer must have a way of declaring certain combinations well-formed and other combinations ill-formed. Figure 4(a) shows a well-formed conceptual graph, which represents the phrase "boy walking." The graph in Fig. 4(b) is an ill-formed combination, taken from Chomsky's famous example "Colorless green ideas sleep furiously."

Well-formed conceptual graphs are like well-formed formulas in symbolic logic or grammatical sentences in

English. They are not necessarily true or even plausible, but they rule out some nonsensical combinations. To distinguish the well-formed conceptual graphs, there are formation rules that generate all of the well-formed graphs but none of the ill-formed ones. The data base designer must prime the system with a starting set of conceptual graphs, which are all well-formed by definition. Every other well-formed graph is generated by repeated applications of four basic rules.

Assumption The system has a collection of conceptual graphs that are defined as *well-formed.* Every graph consisting of a single concept is well-formed. All other well-formed conceptual graphs are obtained by repeated application of the following rules:

1. *Copy* An exact copy of any well-formed conceptual graph is well-formed.
2. *Detach* All connected graphs that remain when any conceptual relation is removed from a well-formed conceptual graph are also well-formed.
3. *Restrict* If a is a concept in a well-formed conceptual graph v, then for any sort label $s \leq sort\ (a)$, the graph obtained by substituting s for the sort label of a is well-formed.
4. *Join* Let a be a concept in a well-formed conceptual graph v and b be a concept in a well-formed conceptual graph w, where v and w may be the same graph. Then if $sort\ (a) = sort\ (b)$, v and w may be joined to form a well-formed conceptual graph u by deleting a from v and attaching to b all the links of conceptual relations in v that had previously been attached to a.

To illustrate the formation rules, Fig. 5 presents two conceptual graphs that are assumed to be well-formed. The first graph may be read as "A manager hiring a certain person," and the second as "An employee being hired at a certain date."

Since both graphs in Fig. 5 have a concept with sort label HIRE, they may be joined by deleting one of the concepts with label HIRE and attaching the two dangling links to the corresponding concept in the other graph. This operation produces the graph in Fig. 6.

Since two concepts can only be joined when they have identical sort labels, the sort label PERSON in Fig. 6 would have to be restricted to EMPLOYEE, as in Fig. 7, before it could be joined to the other concept labeled EMPLOYEE.

Since Fig. 7 now has two concepts with identical sort labels, they may be joined by deleting one of them and attaching the dangling link to the other one.

According to the detach rule, one of the two copies of PTNT in Fig. 8 may be removed to form the graph in Fig. 9. This graph may be read "A manager hiring an employee at a certain date."

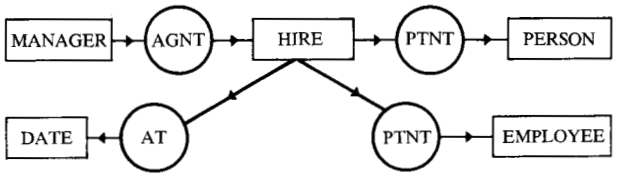

Figure 6 A join of the graphs in Fig. 5.

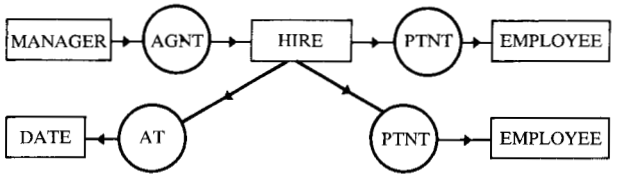

Figure 7 A restriction of the graph in Fig. 6.

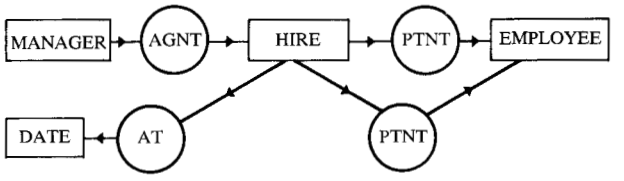

Figure 8 A join of two concepts in the same graph.

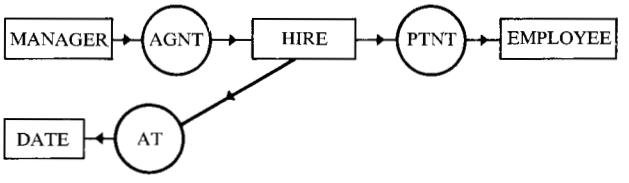

Figure 9 Final graph obtained by detachment.

With an appropriate set of starting graphs, the formation rules generate graphs that may be considered "grammatically correct." But grammar rules are not rules of inference: formation rules generate syntactically well-formed combinations; inference rules generate combinations that are true if the assumptions are true. In defining the rules of a formal system, a logician has various options for assigning a construct either to the formation rules or to the rules of inference. Sorted logic [12, 13] differs from the standard predicate calculus by incorporating sorts into the formation rules; as a result, it often has simpler formulas and shorter proofs. The sort labels on concepts make the formation rules more

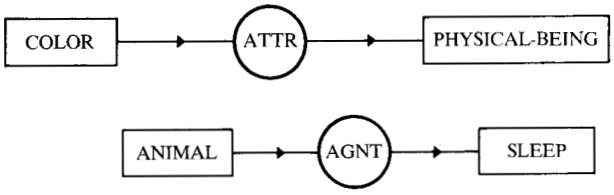

Figure 10 Sample starting graphs.

Figure 11 Graph derived from Fig. 10.

complex, but they sharply reduce the number of alternatives to be considered in performing inferences.

To show how the formation rules impose constraints on a derivation, take the two graphs in Fig. 10 as a starting set. Since ANIMAL < PHYSICAL-BEING, the sort label PHYSICAL-BEING in the first graph can be restricted to ANIMAL. Then the two graphs can be joined on ANIMAL to form the graph in Fig. 11.

This graph can be further restricted to form graphs for the sentences "A brown beaver sleeps" or "A purple cow sleeps." But since IDEA is not a subsort of ANIMAL nor of PHYSICAL-BEING, there is no way of deriving "A green idea sleeps." The formation rules thus eliminate nonsensical things like green ideas, but they allow conceivable, nonexistent things like purple cows. The rules for handling subsorts impose the same kinds of constraints as the semantic markers used by Katz and Fodor [14]. The partial ordering of subsorts, however, is more general than semantic markers: besides binary distinctions, a partial ordering may include arbitrary trees and lattices.

The rules presented so far place no restrictions upon the starting set of well-formed conceptual graphs: any combination of symbols that anyone might ever think of could be represented as a conceptual graph. In setting up a query facility, the data base designer would select a set of concepts for all the domains in the data base, auxiliary concepts for real world characteristics related to those domains, and other concepts for functions that might be applied to values in the domains. Since few data base designers are trained linguists, a practical system would have to be primed with a basic set of concepts for common English words, a set of conceptual relations for linguistic cases and mathematical relations, and a set of tools and questionnaires for automating the task of defining conceptual graphs. Much work remains to be done before the definition of a language can be reduced to filling out a questionnaire, but the purpose of this paper is to present a formalism that may help to systematize that job.

Derived formation rules

The basic formation rules operate on one concept at a time; derived formation rules are sequences of basic operations that do a complex derivation in one step. There are two reasons for having the derived rules: theoretically, they can simplify the definitions and shorten the proofs; and practically, the combined operations can eliminate intermediate computations and improve system performance. The first derived rule is projection, which extracts a subgraph from a conceptual graph and then restricts some of the concepts in it. Another derived rule is the join of two graphs on a common projection, which allows the graphs in Fig. 5 to form the graph in Fig. 9 in a single step. A special case of this rule is maximal join, where the common projection is as large as possible; maximal joins are important in the algorithm for answering a data base query.

Definition A well-formed conceptual graph v is a *projection* of a well-formed conceptual graph w if v can be derived from w by zero or more applications of detachment and zero or more applications of restriction, but no application of join.

Each detachment reduces the size of the resulting graph by at least one conceptual relation. It may also cause the graph to become disconnected, and thereby create several well-formed conceptual graphs, each with fewer concepts and conceptual relations than the original. Each restriction leaves the number of concepts and relations unchanged, but it makes the graph more specialized. None of the formation rules allow a restriction to be undone to return to the original, more general graph.

Theorem If v is a connected subgraph of a well-formed conceptual graph w, then v is a projection of w that can be derived from w solely by the rule of detachment.

Proof Apply the rule of detachment to each conceptual relation of w that is not in v. All the graphs that remain are, by definition, projections of w. Since v is connected and none of its conceptual relations were detached, v must be wholly contained within one of those projections. That projection cannot contain any conceptual relation not in v since all of them were detached. Furthermore, it cannot contain any concept not in v since each such concept would have to be attached to a concept of v by some conceptual relation not in v. Therefore, that projection must be v.

If two isomorphic graphs were drawn carefully on transparent plastic sheets, one graph could be overlaid on the other with a perfect match: all concepts, conceptual relations, and links would line up exactly. The next theorem shows that a projection of a graph can be over-

laid on some subgraph of the original; the matching conceptual relations would be identical, but some or all of the concepts in the original would correspond to subsorts in the projection.

Theorem If v is a projection of w, then there is an isomorphism ϕ that maps v onto a connected subgraph w' of w: if a is any concept of v, then a is a subsort of $\phi(a)$; if r is any conceptual relation of v, then $r = \phi(r)$; and if the ith link of r is attached to a_i, then the ith link of $\phi(r)$ is attached to $\phi(a_i)$.

Proof Since v may be derived from w, there must be a finite sequence of applications of detachment and restriction, $A^1, A^2, \cdots$, leading from w to v. Construct two series of well-formed conceptual graphs $v^1, v^2, \cdots$, and $w^1, w^2, \cdots$, and a series of isomorphisms between them $\phi^1, \phi^2, \cdots$. Let $v^1 = w^1 = w$, and let ϕ^1 be the identity mapping from v^1 to w^1. Then if the ith rule A^i was detachment, perform that rule on both v^i and w^i to derive v^{i+1} and w^{i+1}, and let $\phi^{i+1} = \phi^i$. Or if A^i restricts some concept a to one of its subsorts b, then apply A^i to v^i to derive v^{i+1}, let $w^{i+1} = w^i$, and let $\phi^{i+1}(b) = \phi^i(a)$, but let ϕ^{i+1} have the same value as ϕ^i for all other concepts and conceptual relations in v^{i+1}. At each stage in the derivation, ϕ^{i+1} will be an isomorphism from v^{i+1} to w^{i+1} satisfying the conditions of the theorem if the conditions held for i. Since they hold for 1, they must, by induction, hold for all i. The conditions must therefore hold for the last members of the series, which are v, w', and ϕ.

This theorem implies that a projection of a conceptual graph can be overlaid on some subgraph of the original. That subgraph is called a projective origin of the projection. A given projection may have more than one possible projective origin. Suppose, for example, that the concept c was a common subsort of several different concepts in a graph v; then the concept c by itself would be a projection of v, and every concept in v of which c was a subsort would be a projective origin of c.

Definition If v is a projection of w, then a subgraph of w that is isomorphic to v under the conditions of the preceding theorem is called a *projective origin* of v in w.

The definition of projection would allow the detachments and restrictions to be applied in any order. The next theorem shows that the same projection could be derived in a standard order that first applies all detachments and then applies all restrictions.

Theorem If v is a projection of w, then v can be derived from w by first detaching conceptual relations to form a projective origin of v in w and then performing a series of restrictions on concepts of the projective origin to derive v.

Proof Since a projective origin of v in w is a connected subgraph of w, it must be a well-formed conceptual graph that is derivable from w solely by detachment. Then to derive v, restrict each concept of the projective origin to the concept in v to which it is mapped by the isomorphism.

Definition If v and w are well-formed conceptual graphs, u is a projection of v, and u is a projection of w, then u is called a *common projection* of v and w.

Theorem If u is a common projection of v and w, then the projective origin of u in v is isomorphic to the projective origin of u in w.

Proof Two graphs that are isomorphic to u must be isomorphic to each other.

If two graphs have a common concept, the join rule allows them to be combined by merging the two common concepts. That simple join on a common concept can be extended to a join on a common projection. If two graphs v and w have a common projection u, then the projective origins of u in v and in w are subgraphs of v and w that are isomorphic. Therefore, v and w can be overlaid with the two projective origins matched up exactly. Each concept in the two subgraphs can then be restricted to the common subsort in their common projection u.

Theorem If u is a common projection of v and w, then v and w may be *joined on the common projection* u to form a well-formed conceptual graph by the following steps:

1. Let v' be a projective origin of u in v, and let w' be a projective origin of u in w.
2. Restrict each concept of v' and w' to the sort label of the corresponding concept of u.
3. Detach all conceptual relations of v'.
4. Join each concept of v' to the corresponding concept of w'.

The concepts and conceptual relations in the resulting graph are the union of all those in u, those in $v - v'$, and those in $w - w'$.

Proof To prove that the resulting conceptual graph is well-formed, it is necessary to show that the same graph could be obtained from v and w by applying only the basic formation rules. First, the acts of restricting concepts in v' and w' to their corresponding concepts in u are legal because each concept in a projection is a subsort of the corresponding concept in any of its projective origins. Second, the act of detaching all the conceptual relations of v' at once produces the same collection of well-formed graphs obtained by detaching them one at a

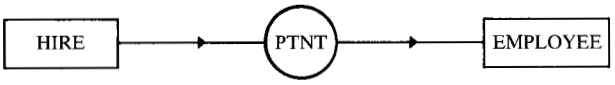

Figure 12 A maximal common projection of graphs in Fig. 5.

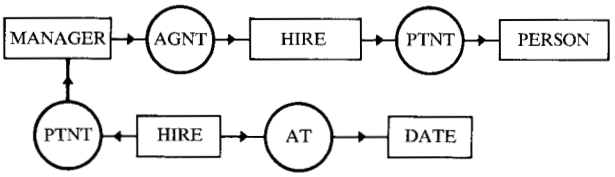

Figure 13 A maximal join with a different kernel.

time. Finally, the acts of joining the concepts from v' and w' are legal because they have been restricted to identical subsorts. The derivation is therfore equivalent to a sequence of detachments, restrictions, and simple joins.

The first set of restrictions had the effect of replacing v' and w' with copies of u. Since no detachments were performed on any conceptual relations of w, the resulting graph before the joins must have been the union of u with $w - w'$. Since the original graph v was connected, every concept and conceptual relation of $v - v'$ must have been connected to some concept of v' by conceptual relations not in v'. Therefore, the final series of joins would result in combining the concepts and conceptual relations of $v - v'$ with those of the union of u with $w - w'$.

When two graphs are drawn on transparent sheets, a join on a common projection could be illustrated by covering part of one graph with part of the other graph. Overlapping conceptual relations have to match exactly, but overlapping concepts are restricted to common subsorts.

Definition If v and w are joined on a common projection u, then all concepts and conceptual relations in the projective origin of u in v and the projective origin of u in w are said to be *covered by the join.* In particular, if the projective origin of u in v includes all of v, then the entire graph v is said to be covered by the join.

The notion of covering is important for answering a data base query. The user's original question is translated into a query graph, and the system generates an answer graph whose join with the query graph covers it completely. The next three definitions introduce maximal joins, which are used in deriving the answer graph.

Definition If u is a common projection of v and w, then a *kernel* of the common projection consists of three concepts: any concept a in u, the concept b that corresponds to a in a projective origin of u in v, and the concept c that corresponds to a in a projective origin of u in w.

A kernel of a common projection is important because a basic algorithm for computing common projections is to start with a kernel and then build it up into larger graphs by adding other concepts and conceptual relations.

Definition Let u be a common projection of v and w with a kernel $k = (a, b, c)$. Then u is called a *maximal common projection* with respect to the kernel k if there is no graph t with the following properties: t is a common projection of v and w with the same kernel k, u is a projection of t, and u is not identical to t.

Definition Let u be a maximal common projection of v and w with respect to the kernel $k = (a, b, c)$. Then a *maximal join* of v and w with respect to k is a graph obtained by joining v and w on the common projection u under the condition that the concept b in v is joined to the concept c in w.

Figure 12 is a maximal common projection of the graphs in Fig. 5 with respect to a kernel consisting of the three concepts labeled HIRE: one in each of the graphs of Fig. 5 and the one in Fig. 12. By a maximal join on this graph, Fig. 9 could be derived from Fig. 5 in one step.

A single concept MANAGER would also form a maximal common projection of the same two graphs, with the kernel containing the concept labeled MANAGER in the first graph of Fig. 5 and EMPLOYEE in the second graph. Figure 13 shows the corresponding maximal join; this join cannot be extended as far as the two concepts HIRE because the conceptual relations AGNT and PTNT are different. The derived graph may be read as "A manager who hired a person was hired at a certain date." As this example shows, two graphs may have several different maximal joins.

Values and quantifiers

The concepts described so far are generic concepts that may represent anything of a given sort. To describe actual information about particular entities or events in the data base, concepts must be associated with particular instances. A concept behaves like a variable in the predicate calculus: the sort label is analogous to a subscript in sorted logic or a data type in a programming language; it determines the kind of entities, events, or properties that the concept may represent. The concept NUMBER, for example, may represent any number; to specify a particular number, it must be assigned a value. Figure 14 shows concepts with values specified by placing either a literal or a proper name after the sort label.

A concept may be indefinite, constant, or quantified. An indefinite concept has just a sort label inside the box, a constant concept has a specified value, and a quantified concept has a logical quantifier. Since values and quantifiers are mutually exclusive, they are both written in the same position in the box. The symbol ∀ represents a universal quantifier and ∃ an existential quantifier.

Besides representing quantifiers, conceptual graphs must indicate their scope. For each existential quantifier that depends on one or more universal quantifiers, dotted lines may be drawn from the universal quantifiers to the existential. Figure 15, for example, represents the proposition

$$(\forall x)(\forall y)(\exists z)(z = \textit{difference}\ (x, y)).$$

Note that the variable names x, y, and z have disappeared from Fig. 15. The purpose of named variables in logic is to indicate repeated uses of the same variable by repeated occurrences of its name. In conceptual graphs, however, a variable appears as a box, and all uses of that variable are linked to the same box. By eliminating named variables, the graphs eliminate accidental variations caused by different choices of names and avoid the need to rename variables in substitutions.

Dotted lines showing the scope of quantifiers can express finer distinctions than the standard predicate calculus. For example, consider a predicate $P(x, y, z, w)$ with x and y universally quantified, z existentially quantified depending only on x, and w existentially quantified depending only on y. Both of the following formulas in standard logic introduce irrelevant dependencies of w on x or z on y:

$$(\forall x)(\exists z)(\forall y)(\exists w)\quad P(x,y,z,w),$$
$$(\forall y)(\exists w)(\forall x)(\exists z)\quad P(x,y,z,w).$$

The dotted lines overlaid on a conceptual graph represent only those dependencies that are logically necessary [15].

The ordinary existential quantifier ∃ states that there exists one or more entities that meet the given conditions. For the graph in Fig. 15, the result of the function is unique; therefore, the unique existential quantifier E^1 may be used to state that there exists exactly one value of z for each pair of x and y. In general, a function of n arguments is determined whenever the unique existential E^1 depends on n universal quantifiers. Since the dotted lines then define a function, they are called functional dependencies [16]. The basic conceptual graph represents the domain roles, and the functional dependencies are a separate graph structure overlaid on top of the original graph; the two structures represent complementary information, and each is necessary for a full description of the data base relations.

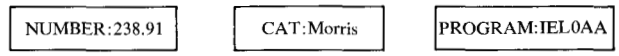

Figure 14 Concepts with specified values.

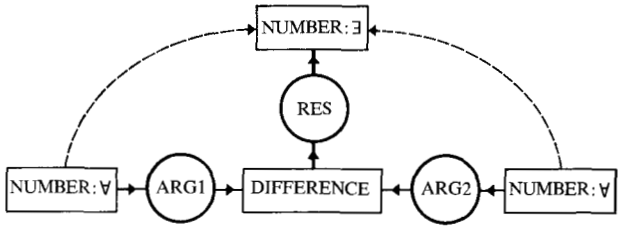

Figure 15 Quantified concepts with lines indicating scope.

Definition A *functional dependency* in a conceptual graph is a set of *function links* from one or more concepts called *sources* to a concept called the *target* of the functional dependency. Associated with each functional dependency is an *access procedure*. Whenever all the sources of a functional dependency have values, the access procedure can compute a value for the target.

In the ordinary predicate calculus, functional dependencies are seldom stated explicitly. In a system for accessing data bases and other computational facilities, a statement that one variable depends on another is not as useful as an access procedure that can compute the dependency upon request. The technique of combining a logical notation with procedures that evaluate functions has great generality: Woods [17] developed such a technique for his question answering system, Winograd [18] suggested a method that he called procedural attachment in his discussion of frame systems, and Weyhrauch and Thomas [19] used a similar method of semantic attachment in their proof checking system.

Plural nouns normally have sets of entities as their values. For the question "What employees were hired by Jones?" the expected answer would be a set. To answer such questions, concepts may have sets as values, and a new symbol, E-set, is introduced to represent "There exists a set of sort. . . ." This quantifier is weaker than the ordinary existential because it allows empty sets as values of the concept. Since E^1 defines an ordinary function, it can support functional compositions; E-set, however, cannot support the same compositions because it would create fallacies such as the connection trap [20].

Another refinement that increases the expressive power of the notation is the possibility of introducing compound sort labels, which are themselves conceptual graphs. For the nonrestrictive phrase "All elephants, which have trunks," the quantifier "all" applies only to

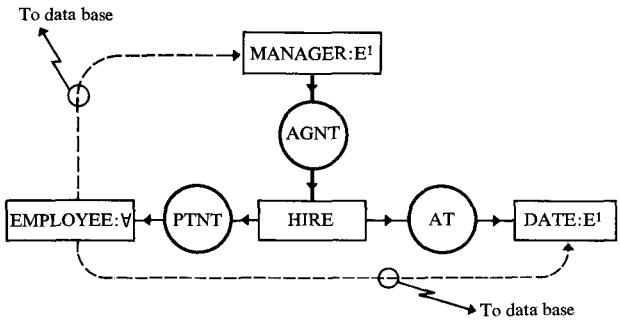

Figure 16 Conceptual schema for the HIRE relation.

the sort ELEPHANT; but in the restrictive phrase "All elephants that perform in circuses" the quantifier applies to the sort ELEPHANT-THAT-PERFORM-IN-CIRCUS. Although a sorted logic could use roundabout paraphrases to avoid adding new sort lables, Altham and Tennant [21] developed a notation for sort expressions that reflects the structure of the original English sentence. For this example, the compound sort expression would itself include an existential quantifier on CIRCUS.

Altham and Tennant's approach could be adapted to conceptual graphs by making the set of sort labels open-ended and accepting graphs that meet certain conditions as new labels. Formalizing that approach, however, would require considerable discussion that is beyond the scope of this paper. Values and quantifiers on concepts are defined formally by introducing two selector functions, *value* and *quant*, which apply to a concept and return whatever value or quantifier is written inside the box.

Assumption The two functions *value* and *quant* may be applied to any concept c. If both $value(c)$ and $quant(c)$ are undefined, then c is *indefinite*. If $value(c)$ is defined, then c is *constant*. If $quant(c)$ is defined, then c is *quantified*. These two functions obey the following rules:

- No concept is both constant and quantified: for all c, either $value(c)$ or $quant(c)$ is undefined.
- Whenever $quant\ (c)$ is defined, its value is one of the four symbols $\{\forall, \exists, E^1, \text{E-set}\}$.
- There is a function *permissible*, which defines a set of permissible values for a concept with a given sort label: if c is any constant concept, then $value(c)$ must be in the set $permissible(sort(c))$.
- If s and t are sort labels for which $s \leq t$, then $permissible(s)$ is a subset of $permissible(t)$.

Note that the formal definitions are independent of the box and circle notation. By using the selector functions *sort*, *value*, and *quant*, definitions and theorems can refer to the components of a concept without mentioning the way they happen to be drawn on paper or represented in computer storage. Although diagrams are important for helping people to visualize conceptual graphs, the formalism should not depend on some artist's drafting techniques.

Conceptual schema

A conceptual schema is a conceptual graph that has certain combinations of quantifiers and functional dependencies. It is a mediator between the conceptual graphs and the other facilities in the computer system: its underlying structure is a conceptual graph, but its functional dependencies form a superimposed structure that provides a direct mapping to the data base. To answer a user's question, the system would find or construct a schema having functional dependencies that would compute the desired answer. The computed values may be simple scalars if the target of a functional dependency is quantified with E^1, or they may be sets if it is quantified with E-set. The ordinary existential quantifier $\exists$ is not used in a schema because it does not define a unique function.

Definition A *conceptual schema* is a well-formed conceptual graph having one or more functional dependencies. If the concept c is a source of one or more functional dependencies, but not a target of any dependency, then $quant(c) = \forall$. If c is a target of one functional dependency and a source of one or more functional dependencies, then $quant(c) = E^1$. If c is a target but not a source, then $quant(c)$ is either E^1 or E-set. No concept may be the target of two or more functional dependencies. All other concepts in the schema are indefinite and are called *selectors*.

For a relational data base, the relations are described by conceptual schemata. If a relation is in Codd's third normal form [22], it has a *key* consisting of one or more domains, all other domains are functionally dependent upon the key, and there are no transitive dependencies. For such a relation, the conceptual schema would have a quantified concept for each domain as well as selector concepts and conceptual relations that describe the domain roles. In the relation HIRE, for example, the key domain EMPLOYEE is universally quantified, the concepts for the nonkey domains MANAGER and DATE have the quantifier E^1, and the selector concept HIRE is indefinite (Fig. 16). Attached to the two functional dependencies are the access paths for some data base relation.

Many relations and functions have more than one set of key domains. For such relations, the system needs a separate schema for every possible set of keys. *Differ-*

ence is a function with three possible keys: in normal use, the two arguments are keys that determine the result, but with either argument and the result, the other argument is determined. In order to compute any of the three values, given the other two, the system would need three schemata: besides Fig. 15, it would need a schema that showed argument 1 functionally dependent on result and argument 2, and another schema that showed argument 2 functionally dependent on result and argument 1. The system would select the appropriate schema for a given problem, depending on which values were specified and which were to be determined. Since the algorithms for a function and its inverse are usually quite different, each of the three schemata would have to specify a different procedure.

The conceptual schema for *difference* illustrates the use of schemata for representing functions that access a procedure instead of a stored file. The greater-than relation, for example, has infinitely many entries, but it could be evaluated by a simple procedure; its conceptual schema would look like a data base schema, but its function links would be attached to a procedure [23]. As this example illustrates, a conceptual schema can present the same interface to the user for either a computed or a stored relation.

For some data base relations, all domains are part of the key. The STOCK relation, for example, is a many-to-many relation between parts and the supplier who stock them; each supplier may stock multiple parts, and each part may be in the stock of multiple suppliers. This relation has two domains, both domains are part of the key, and there are no other domains functionally dependent on the key. To place universal quantifiers on both domains in the schema would be inaccurate because it would imply that all suppliers stock all parts. Instead, two schemata, as in Fig. 17, are needed: one says that for each supplier there exists a set of parts, and the other says that for each part there exists a set of suppliers.

The quantifier E-set includes the possibility that the set may be empty. Some suppliers, for example, may not stock any parts at the present time. Whenever a domain Y is functionally dependent on domain X, e.g, f: $X \rightarrow Y$, the inverse function f^{-1} determines a set of values in X (possibly empty) for each value in Y. For the HIRE relation, DATE is functionally dependent on EMPLOYEE; therefore, for all dates, there exists a possibly empty set of employees hired on that date. Since this fact is implied by the schema in Fig. 16, the system could either have a separate schema for the inverse, or it could have a mechanism for deriving an inverse schema when necessary.

Any relation over n domains can be characterized by a conceptual schema having $n - 1$ source concepts and a target concept quantified with E-set. This is the weakest

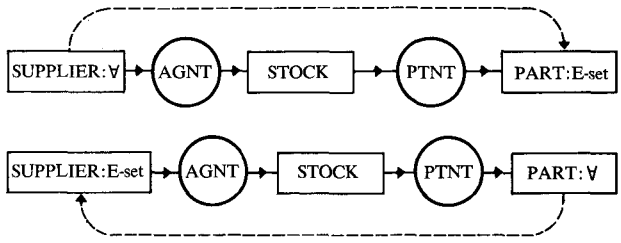

Figure 17 Conceptual schemata for the STOCK relation.

possible assumption and also the least interesting. In most practical applications, a relation has a key of less than n domains and the nonkey domains have unique values for each combination of values for the key domains. When a target concept has the unique existential E^1, targets and sources may be joined to form intricate paths of functional dependencies to answer complex questions. The selector concepts in a schema are like the knobs and indentations on the pieces of a jigsaw puzzle: they determine the ways in which a schema may be joined to other schemata. In the extreme case, the analogy with a jigsaw puzzle is complete; the schemata fit together in only one way to answer a fixed number of possible questions. In the general case, there is an endless variety of combinations; the selectors match concepts in the user's question to determine the selection of schemata required to answer it.

In the SPARC/DBMS approach [24], the term conceptual schema refers to a complete description of all the logical relations in the entire data base; it is a description of the user's view at a conceptual level rather than a description of the system's view of the data as stored. Since a conceptual schema as defined in this paper describes only a part of the data base, it would correspond to a conceptual subschema in the SPARC/DBMS terms. Except for this difference in terminology, the SPARC/DBMS approach is compatible with the theory developed in this paper. In fact, the formalism for conceptual graphs and schemata may be considered as a proposed description language for SPARC/DBMS, which as yet has not developed a formal notation.

As in the SPARC/DBMS approach, the first step in defining a data base is to develop a formalized model of an enterprise, with a list of all the entities and relationships to be represented. For each relation to be stored in the data base, the data base designer would define one or more conceptual schemata to represent the roles of the entities and the functional dependencies between them. He would then map the access procedures to internal paths in the data base, and he would specify a domain in some relation for each quantified concept. If a schema

had n quantified concepts, the mapping would specify n different domains in the data base, possibly in different relations. The formation rules for conceptual graphs could then be used to form the schemata of derived relations: an equal-join of two data base relations would correspond to a join of their schemata, and a projection of a relation would correspond to a projection of its schema.

Boolean connectives

The world, according to Wittgenstein, is all that is the case [25]. For a particular aspect of the world, a data base is all that is known to be the case. Wittgenstein's view of the world as a "totality of facts," an enormous conjunction of elementary propositions, is a position that he modified in his later philosophy, but his early position is an apt characterization of a data base: a data base is a large conjunction of propositions, all asserted to be true.

The main advantage of relational data bases is that the propositions are stored in a simple logical form. The data base is organized as a conjunction of relations, each represented by a list of n-tuples for which it is known to be true; the structure is further simplified by storing the relations in third normal form. Furthermore, the propositions in a data base are positive; data bases seldom if ever contain negated predicates and relations. Other Boolean connectives, such as disjunctions, are also absent in the data as stored, although they may be used in a data base query. Conditionals are never present in stored data and are rarely used in queries, but they are common in stating constraints.

These observations imply that the Boolean connectives have clearly distinct uses in a data base system. Reaction-time experiments show that people also find some connectives more difficult to process than others: disjunctions normally take more time than conjunctions; negative statements take longer to interpret than positive statements if they are presented in isolation, but they are just as easy to process as positive statements if they are negating a presupposition that underlies the current discussion [26]. The apparent symmetry in the standard notation for symbolic logic is misleading because it suggests that conceptual graphs should represent them all in a parallel form. In fact, conjunction is the only one that is easy to represent—simply by joining graphs. To represent all Boolean connectives in a conceptual graph, there are two basic approaches, which may be called abstract and direct.

The abstract approach treats Boolean connectives as functions of truth values, as in symbolic logic. It introduces two new sort labels, SITUATION and TRUTH-VALUE, and a conceptual relation MODE. A concept labeled SITUATION may have conceptual graphs as values, and one labeled TRUTH-VALUE may have values *true, false, possible, unlikely*, etc. MODE is a dyadic relation that links SITUATION, whose value is a conceptual graph, to TRUTH-VALUE, whose value states whether the graph is true or false. Then all Boolean connectives are represented as concepts of functions that take truth values as arguments and produce truth values as results; for any formula in the propositional calculus, each Boolean connective would correspond to a concept of a Boolean function, and each propositional symbol would correspond to a concept with sort label SITUATION. With this construction, formulas in symbolic logic can be mapped directly into conceptual graphs. This approach shows that conceptual graphs are at least as general as standard logic, but it does not take advantage of the special properties of the graphs.

The direct approach is based on the hypothesis that conceptual graphs are isomorphic to the mental structures underlying human thinking; it uses psychological and linguistic evidence to formulate the rules and computer simulations to test their efficiency. This approach, which is discussed in a forthcoming book [27], assumes that the Boolean connectives may be represented statically, as in the abstract approach, but that their primary role is to indicate operations for combining conceptual graphs; the system has a current working graph, and the Boolean connectives specify operations on that graph:

- *Conjunction* Join a new graph to the working graph.
- *Negation* Detach the negated subgraph from the working graph; an isolated negation would have nothing to detach and would require the creation of an artificial working graph.
- *Disjunction* Create an extra copy of the current working graph, and join each alternative to one of the copies.
- *Implication* If some subgraph of the working graph is a projection of the antecedent, then join the consequent to the working graph.

Since the topic of this paper is not psychology, but data bases, further elaboration and justification for these rules is left to the forthcoming book. Two observations, however, are worth making: first, the procedural aspect of these rules may make them useful in a computer simulation; and second, they are compatible with the abstract approach, since the working graph could be made up of concepts of situations, truth values, and Boolean connectives. People could therefore learn to do symbolic logic, but they would have to perform multiple "direct" steps for each "abstract" operation.

For answering data base queries, the system could use both direct operations for combining graphs and abstract

representations for Boolean connectives. If the user's query contained Boolean connectives, they would initially be translated into the abstract style. But to determine what data base relations should be accessed, the system would rely upon operations for directly combining conceptual schemata. Since the abstract approach does not illustrate the novel features of conceptual graphs, the remainder of this paper concentrates on techniques using direct operations.

Answering a query

Since the logical structure of a data base has such a simple form, a special algorithm for answering data base queries can be more efficient than a general procedure for proving theorems. Whereas a theorem prover deduces a general theorem, a data base system starts with facts about particular entities and determines which entities satisfy a given relation. A typical query may have the logical form,

Find all pairs (x, y), for which $R(x, y, a, b, c)$ is true.

In this example, R is a relation, x and y are variables whose values are to be determined, and a, b, and c are constants specified in the query. If R happens to be one of the basic relations around which the data base is organized, the logical problem of answering this query is trivial, although the programming effort may be significant for nonrelational data bases. A major difficulty for data base query systems occurs when R is not one of the basic relations but must be determined by some combination of relations.

Most query languages avoid the difficulty by requiring the user to learn the files or relations in the data base and then to state his query in terms of them. Even the sophisticated systems based on relational data bases, such as SEQUEL, SQUARE, or Query-by-Example, require the user to name each relation explicitly. If the query requires data from two or more relations, the sequence of operations for combining them can quickly exceed the abilities of a nonprogrammer. Menus and on-line help facilities can remind the user of the available relations and give him a refresher course on how to combine them. Help facilities, however, do not make the learning problem go away; they just provide a piecemeal tutorial instead of a complete user's guide.

For systems with a natural language interface, the inference problem cannot be avoided. The following statement, for example, is not natural English:

In relation HIRE, find EMPLOYEE where MANAGER is Jones.

People never talk like that to other people. They would say, "Who did Jones hire?" and expect the listener to infer what relations and domains are involved. Many natural language systems can answer this type of question because the required domains are in a single data base relation; the REQUEST system described by Plath [28] and Petrick [29] supports a natural syntax for such questions. Queries that apply a function to the data are also easy to handle if the user's question specifies the function by an appropriate keyword, such as "average" or "total." Queries that combine data from two or more relations can be handled if the system designer anticipates the form of the question by providing a macro or procedure for answering it. But conceptual graphs are designed for the more general problem of having the system determine for itself what relations and domains are necessary to answer a given question.

If the user's question is incomplete or ambiguous, the system may prompt him for further information; and it may refuse to accept words or constructions that it doesn't understand. But in no case should the system require the user to specify the stored relations and access paths in the data base. Determining the required relations is not a problem of syntax, but semantics. The REQUEST system, for example, has a sophisticated syntax that can translate a usable subset of English into a formal notation, such as a conceptual graph or a relational query language. Heidorn's natural language processor [30] would also be adequate; his internal problem descriptions are similar in structure to conceptual graphs, and his prompting technique could support a dialogue for handling complex queries. The inference problem arises *after* the natural language statement has been translated into a formal notation: merely converting the syntax cannot add information; if the user did not specify the relations, the translated form will not specify them either.

For the TORUS system [2], the designer must solve the inference problem in advance by creating a predefined network of concepts with all possible combinations that anyone might ever ask about. When the user types in a question, the system translates it into a query graph and attempts to match it to some part of the predefined network. Associated with various parts of the network are links to appropriate data base relations. Which relations are required to answer a given question is determined by the parts of the network that match the query graph.

For a data base with a small number of domains that can only be related in a fixed number of combinations, the predefined network is adequate. But in the general case, the number of combinations may be infinite; someone might ask the question "Who was the person who hired the person who hired Jones?" where a single relation may be iterated arbitrarily many times. If the system can join conceptual schemata as needed, it can gen-

Figure 18 A sample query graph.

erate only those combinations required for the question at hand; otherwise, it would have to store an enormous number of combinations, most of which would never be used. Another weakness of the giant network is the lack of modularity: if the data base designer wanted to add, delete, or redefine a data base relation, he would have to change every part of the network from which that relation could be accessed; in an incremental approach, however, he might only have to change one schema that was mapped to that relation. A third weakness of the predefined network is that it places the burden on the data base designer to foresee all possible combinations at the time he is defining the network; an incremental approach would allow him to enter simple schemata and let the system form the combinations. Questions of relative efficiency depend on the implementation: whether it is faster to search through a large graph or to copy and join small graphs; whether a single large graph requires more I/O transfers than several small graphs; and whether search techniques can more easily find a path through a large graph or find multiple small graphs.

With conceptual graphs, either the single network or the collection of schemata could be used. The recommended approach, however, is to join schemata as needed to answer a given query. For efficiency, some combinations that are frequently used together could be included in a single schema. The value of the concept AGE, for example, may be computed by finding a date of birth in the data base and calling a procedure that subtracts two dates; therefore, the schema that defines AGE could include access links for both the data base relation and the procedure. Efficiency, of course, is meaningless unless the system can answer the original question and guarantee that the answer is correct; it must start with the query graph and determine which schemata to join and which data base accesses to make in order to compute the answer.

When the user types in a question, the input analyzer should translate it into a well-formed conceptual graph q (the systems described by Heidorn or Petrick could be adapted to do the translation). Every concept in the graph q whose value is to be determined would be flagged with a question mark. To determine values for the flagged concepts, the system should generate an answer graph w that meets the following criteria:

1. w is a well-formed conceptual graph.
2. w is true if the data base is correct.
3. The entire query graph q is covered by a join with the answer graph w.
4. For every concept in q that has a value, the corresponding concept in w has the same value.
5. For every concept in q that had a question mark, the corresponding concept in w has a value.

Point 1 would be satisfied if the system generates w by using the basic formation rules or the derived rules such as projection and maximal join. Point 2 guarantees that the system is sound; i.e., it will not generate incorrect answers. Point 3 implies that w includes all of the domain roles and relationships of the query graph, although some of the concepts may be further restricted; for example, PERSON in the query graph may be restricted to EMPLOYEE or MANAGER as a result of joins with various conceptual schemata. Point 4 insures that the answer is talking about the same entities that the user asked about. And point 5 states that w must include an answer to the question. If the original question was incomplete or ambiguous, then it would not have a unique answer. In that case, the system should prompt the user for further information; it should not require him to restate the entire question, but only to add values or conditions that are necessary to complete it.

Algorithms for generating an answer graph

Before considering a general algorithm, we should look at a special case where the answer graph is easy to find. Suppose someone asked the question "Who hired Lee?". The query graph for this example is Fig. 18.

The conceptual schema in Fig. 16 almost meets the criteria for an answer graph: a maximal join with the schema for HIRE would cover the entire query graph. The schema does not satisfy criterion 4 or 5, however, because it does not have values for the concepts in the query graph having a value or a "?". To determine values, Fig. 18 may be joined with Fig. 16 to produce Fig. 19. The question mark from the query graph is carried over, and the value "Lee" replaces the universal quantifier. (In the standard predicate calculus, a universally quantified variable may always be replaced by a constant; but in a sorted logic, the replacement is permissible only if the sorts match. The corresponding rule for conceptual graphs implies that the system must check whether Lee is an employee before restricting PERSON:Lee to EMPLOYEE:Lee in order to perform the join.)

When the target of a function is flagged with a question mark and all of its sources have values, then a value for the target can be computed by the access procedure. Since the concept MANAGER:E[1]? is functionally dependent on EMPLOYEE:Lee, the value for MANAGER can be obtained from the data base. Then the

manager's name would be substituted for the quantifier, and the resulting graph would satisfy all the criteria for an answer. In a more complex case, the target of a functional dependency may be flagged with a question mark, but one or more sources may not yet have values; in that case, question marks could be propagated backwards along the function links to flag the source concepts whose values are requested. If the question marks eventually stop on concepts with values, then the access procedures can be called and the results returned to the original question mark, which came from the user's query.

The critical problem arises when a question mark stops on a concept that has neither a value nor a function link leading to it; then there is no procedure to execute or place to propagate another question mark. Such a state is similar to an original query graph. Figure 18, for example, had a question mark on a concept but no function links. For Fig. 18, the answer was obtained by joining the query graph to a conceptual schema so that the question mark was joined to a target concept. This technique could be generalized to form algorithm A:

> Start with the concepts on the query graph that are flagged with question marks; join conceptual schemata to the graph so that the flagged concepts are covered by target concepts; propagate the question marks backwards along the function links; evaluate any functional dependencies whose sources all have values; and repeat until the original question is answered.

This algorithm sounds plausible, but will it always terminate with a result, and will the result be the correct answer to the original question?

Unfortunately, algorithm A may not always terminate, and it can generate incorrect results. If the original question was incomplete, the algorithm makes no provision for generating prompts: instead, it keeps joining schemata and propagating question marks without ever having enough values to answer them. Although every function may generate correct results, there could be multiple paths of function links in the data base, and the algorithm might stumble upon a path that answered a question different from the one the user asked; the user may have asked for the quantity of widgets on hand, and the algorithm could generate the quantity ordered.

To keep the system from looping endlessly on unsolvable problems, algorithm B imposes another condition on algorithm A: every join of a new schema to the developing answer graph must cover at least one concept of the original query graph. This restriction keeps the graphs from growing too large, with branches far remote from the concepts of the original query. By avoiding remote joins, algorithm B may be unable to answer some complex queries automatically, but it could still answer

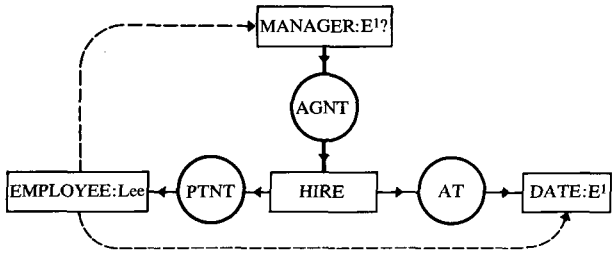

Figure 19 First step towards the answer graph.

a potentially infinite number of questions. For example, "Who was the person who hired the person who hired the person who hired Jones?" could be answered because every schema joined would cover part of the original query graph. If the system runs out of schemata to join to the graph and it still has unanswered question marks, it could use the concepts left with question marks as the starting points for prompting the user for further information. By asking for help when it runs into a dead end, the system could extend the query graph and eventually generate any answer derivable from the data base.

The additional restriction for algorithm B keeps it from looping, but it does not guarantee correct answers. The next three definitions characterize the permissible conditions for joining schemata, the schematic universe that includes all possible schemata that may be derived by repeated joins, and the set of all correct answer graphs. Every answer graph may be generated by determining values for the quantified concepts of some schema in the schematic universe. For the TORUS system, the analog of the schematic universe is its single large conceptual network. For this theory, however, the schematic universe would never be generated in its entirety; instead, schemata would be joined as needed to answer a given query.

Definition A *schematic join* is a join either of two conceptual graphs or of one conceptual graph with itself under the following conditions:

- The join must be maximal.
- The result inherits all function links; the sources and target of each functional dependency in the resulting graph are the concepts covered by the source and target concepts in the original graph.
- When an indefinite concept is joined to a quantified concept, the resulting concept has the same quantifier.
- When the quantifier E^1 is joined to the quantifier $\forall$, the result has quantifier E^1.
- When the quantifier $\forall$ is joined to the quantifier $\forall$, the result has quantifier $\forall$.

- All other joins of quantifiers (E^1 to E^1, E^1 to E-set, E-set to E-set, and E-set to $\forall$) are prohibited; if this restriction prevents a join from being maximal, then the prospective schematic join is rejected.

Theorem The graph that results from a schematic join of one or two conceptual schemata is also a conceptual schema.

Proof This result follows from the observation that the properties defined for a conceptual schema are preserved by the conditions for a schematic join.

One reason for requiring the joins to be maximal is to force the paths of selector concepts to coalesce whenever possible; otherwise, redundant or spurious paths could be generated that the data base designer had not intended. Since universal quantifiers are the sources of function links, the effect of joining two universals is to specify the same argument for two different functions or for two arguments of the same function. Joining E^1 to $\forall$ specifies the result of one function as an input argument of another. Joining two existential quantifiers is prohibited because two different functional dependencies would then have the same target concept, which might be assigned two inconsistent values. The requirement for maximal joins together with the prohibition against joining two existential quantifiers prevents the same schema from being joined more than once in exactly the same position; this restriction prevents the system from getting into a loop when it is generating an answer graph. The rule against joining E-set with a universal quantifier prohibits sets as inputs to functions; further extensions to the theory could allow sets as inputs, but these will not be considered in this paper.

Definition The *schematic universe* determined by a set S of conceptual schemata is the set of all schemata obtained by the following operations:

- All schemata in S are in the schematic universe.
- For each schema s in the schematic universe, if a schematic join of s with itself is possible, then the schema that results from that join is in the schematic universe.
- For each pair of schemata (s, t) in the schematic universe, if a schematic join of s and t is possible, then the schema that results from that join is in the schematic universe.

Since the schemata in S define functions, the schematic universe represents all possible functions that may be derived by composition of the functions in S. As examples, let $f(x)$ and $g(x, y)$ be functions defined by schemata (cf. Fig. 15 for a diagram of such a schema). Then a schematic join of the target concept of the schema for f with the source concept of another copy of the same schema would produce the schema for $f(f(x))$. The join of the target of f with the source concept of the same copy would produce the schema for $x = f(x)$. A join of the two source concepts for g would produce the schema for $g(x, x)$. And a join of a schema for f with a schema for g in all possible combinations would produce schemata for $g(f(x), y)$, $g(x, f(y))$, and $f(g(x, y))$ when the target of one functional dependency is joined to a source of another; but it would also produce the combinations $\{f(x), g(x, y)\}$ and $\{g(x, y), f(y)\}$ when the sources are joined. Note that joining two copies of the schema for f could not produce $\{f(x), f(x)\}$ because a maximal join would cause the two identical function concepts as well as the two target concepts to be overlaid on top of each other; the prohibition against joining two existentials would then cause the join to be rejected.

When all universally quantified concepts in a schema are assigned values from the data base, then the access procedures can compute values for the other quantified concepts. By systematically generating values for all schemata in the schematic universe, the set of all possible answer graphs may be enumerated. If a closed cycle of function links has been created by some schematic join, then the resulting schema can never obtain values for targets in the cycle and cannot lead to an answer graph. A conflict may arise when a target concept has been restricted for some join; then a value computed for the target may not belong to the subsort to which the concept has been restricted; any graph with such a conflict is rejected [31].

Definition An *answer graph* is a conceptual graph obtained by assigning values to the quantified concepts of some schema s according to the following rules:

- For each concept c in s where *quant* $(c) = \forall$, assign a value in the set *permissible* (*sort* (c)).
- When all source concepts of a functional dependency have values and the target does not have a value, then use the access procedure to compute a value for the target. Repeat as long as there is a dependency whose sources have values and target does not.
- If some target concept remains without a value, then reject the graph.
- If some target concept c has been assigned a value that is not in the set *permissible* (*sort* (c)), then reject the graph.
- Otherwise, accept the graph as an answer graph.

The answer graphs are all well-formed conceptual graphs because they are simply schemata with values assigned. The question of whether the answer graphs are

all true depends on the adequacy of the original set of conceptual schemata. Each answer graph is a statement of some relationships between entities recorded in the data base; if the stored data were correct and if each of the original schemata correctly stated functional dependencies, then their joins would also state a correct functional dependency. Possible errors could arise because some combination of schemata might cause the selector concepts (the indefinite concepts such as HIRE in Fig. 16) to form unexpected paths that are not true. The selector concepts are necessary to distinguish different domain roles and to select the correct schemata for answering a given query. In order to avoid undesired combinations, however, the number of selectors should be kept to the minimum necessary to distinguish the domain roles. An important topic for further study is a set of guidelines to help data base designers define schemata that avoid such combinations.

If the data base has a set of conceptual schemata that rule out incorrect combinations, then the system can answer a user's question simply by picking the correct answer graph. Unfortunately, there are too many possible answer graphs to let the system generate them one at a time and check them against the query graph. Therefore, the system must be more selective and join only those schemata that have a good chance of leading to a satisfactory answer graph. The next four definitions describe an algorithm that avoids incorrect combinations and uses a set of preference rules as a heuristic guide for speeding up the search.

Definition A *query graph* is a well-formed conceptual graph with the following properties: it contains no quantifier or function link, one or more of its concepts have values, one or more concepts have a question mark, and no concept has both a value and a question mark.

The query graph is generated by the input analyzer from the user's original question. The input analyzer must have a starting set of well-formed conceptual graphs that are compatible with the data base schemata: every starting graph used by the input analyzer must be coverable by a join with some schema in the schematic universe. This is a necessary condition for deriving answerable query graphs. It is not a sufficient condition because the user can always ask a question with incomplete information; but, in that case, the system should prompt him for the missing information.

Definition For a set of conceptual schemata S and a query graph q, a *working graph* for q is any conceptual graph that may be obtained by the following operations:

- The query graph q is a working graph for q.
- If w is a working graph for q, and s is a schema in S, then the result of a schematic join either of w with itself or of w with s is also a working graph for q, provided that no concept that has a value is joined to a concept with a quantifier E^1 or E-set, no concept that has a value is restricted to a subsort for which the value is not permitted, and all values and question marks in w are copied over to the corresponding concepts of the resulting graph (some universal quantifiers may therefore be replaced with values).
- If w is a working graph for q and the target of some functional dependency f in w has a question mark, then the graph obtained by adding question marks to every source of f that does not have a value is also a working graph for q.
- If w is a working graph for q, all sources of some functional dependency f in w have values, and the target of f does not have a value, then the graph obtained by evaluating the access procedure for f, replacing the quantifier on the target concept with the value, and erasing the question mark on the target (if present) is also a working graph for q.

The working graphs are steps along the way towards answering a query. The following theorem shows that when all the quantified concepts of a working graph have been given values, the resulting values are the same as those obtained from some answer graph. The theorem is general enough to include algorithms that permit joins arbitrarily remote from the original query graph as well as algorithms that require each join to cover at least one concept of the query graph.

Theorem Let S be a set of conceptual schemata, q be a query graph, and w be a working graph for q. If no concepts in w have quantifiers or question marks and if every concept and conceptual relation of q has been covered by a join with some schema from S, then the values in w for the concepts of q having question marks are the same as those obtained by joining q to some answer graph that covers it completely.

Proof Let $s^1, s^2, \cdots$, by the sequence of schemata that were joined to q in deriving w. Observe that the criteria for joining a schema to a working graph are stronger than the criteria for a schematic join; therefore, the schemata $s^1, s^2, \cdots$ may be joined by schematic joins to each other in the same order and position that they were joined in forming w. The result of these joins is a schema s that is isomorphic to w and contains the same composition of function links used to derive w. Since w has no quantifiers left, the query graph q must have had values to assign to each universal quantifier of s, and the results of evaluating the access procedures must have generated

values for every existentially quantified concept. Therefore, assign the values from q to the universal quantifiers of s, and evaluate all access procedures to determine values for the existentials; the result is an answer graph w', which is isomorphic to w, has the same values for corresponding concepts, but may have different sort labels because of the different order of performing restrictions. Since the values generated for w satisfied all the question marks of q, the same values of w' must also satisfy them. Therefore, a join of q to the answer graph w' would cover q and assign the same values as w to the question marks of q.

This theorem means that any algorithm obeying the conditions for deriving a working graph will generate a correct answer to a query *provided that* the set of schemata do not permit incorrect answer graphs. The next definition states preference rules for choosing between various possible schematic joins. The preference rules have no effect upon the correctness or incorrectness of the answers generated; they are heuristic rules for encouraging joins that have a good chance of answering the question while avoiding paths that are remote from the original question. The preference rules lead to graphs with high "semantic density" as in the technique of preference semantics that Wilks [32] developed for analyzing natural language.

Definition Let S be a set of schemata, q be a query graph, and w be a working graph for q. Then if j is any schematic join either of w with itself or of w with some schema in S, the *preference score* for the join j with w is the sum of the points determined by the following conditions:

- Add a point for each concept in w that is covered by j.
- Add an additional point for each concept in q that is covered by j. If this value is zero, then reject j.
- If a concept in w having a question mark is covered by a target of a function link, then add a point; if it is covered by a source of a function link, then subtract a point.
- If a concept in w that has a value is covered by a source of a function link, than add a point; if it is covered by a target of a function link, then reject j.
- For each concept and conceptual relation in q that has not yet been covered by any join, add a point if it is covered by j.

If every possible join has been rejected, then there is no *preferred join*. Otherwise, the one or more schematic joins with the highest preference score are *preferred joins*.

The preference rules are simply guidelines for choosing between alternative joins. If they require too much computation for a particular implementation, then the rules may be modified or replaced without fear of generating incorrect answers. One way of speeding up the search for preferred joins is to index the schemata according to the concepts they contain: one index for the selector concepts of a schema, another for the target concepts, and another for the source concepts. Then, instead of computing preference scores for all possible joins, the system could pick a question mark in the query graph, look in the index for a target concept that had a common subsort, and choose a join with that schema if its preference score was above a given threshold.

Definition Let S be a set of schemata and q be a query graph. Then the following procedure for generating working graphs for q is called algorithm C:

```
w: = q;
while (there is a preferred join j with w)
  do begin
    w: = result of performing j with w;
    while (there is a source concept a in w
          & a does not have a value
          & a does not have a question mark
          & the target of a has a question mark)
      do place a question mark on a;
    while (there is a target concept b in w
          & b has a question mark
          & all sources of b have values)
      do get a value for b from its access procedure;
    if (there are no question marks left in w
        and all of q has been covered by some join)
      then begin
            print answer;
            stop
            end
  end.
```

If there are any question marks left on w and no preferred joins to perform, then each universally quantified concept having a question mark is called a *prompting point*.

A prompting point is where the system begins when it asks a question to get further information. This method of prompting is similar to Heidorn's technique in his simulation system. Without prompting, algorithm C is not able to generate all possible answer graphs because each preferred join must cover at least one concept of the query graph; questions that require remote searches cannot be answered automatically. Furthermore, algorithm C does no backtracking; in cases where the set of schemata permit many possible combinations, the algorithm might try one combination that would preclude

others. If there are no question marks left on the working graph but not all of the query graph has been covered by some join, then either the original query contained irrelevant information or the system has found an alternative path through the data base that answers a question different from the one the user asked. In such cases, too, it would require help to get out of its predicament.

A system based on algorithm C would do what it was told explicitly and whatever was obviously implied by what it was told. Whenever it could not find an obvious solution to a problem, it would come back and ask for further instructions. A system that searched one level deep would relieve the user of the need to specify much tedious detail, and it could still answer arbitrarily complex questions with some prompting. Although one could relax the preference rules to let the system search deeper, further searching would increase the system overhead without substantially improving its usefulness. As the example in the next section shows, algorithm C can generate sophisticated inferences without a great deal of searching.

By computing an answer graph, the system can determine the state of some entities in the data base in answer to a specific question. If the user had asked a question containing Boolean connectives, the system would have to generate separate answers for each part of the question and then combine them according to the type of connective. If instead of asking about a specific entity such as PERSON:Lee, the question had been about all persons having a certain attribute, then the system should not compute an answer graph with specific values for the concepts; instead, it should generate a schema that could be repeatedly evaluated for every person. Any method, such as algorithm C, that can be used to generate specific answer graphs can also be used to determine a schema simply by erasing the values on the answer graph and saving the functional dependencies. A schema with its access procedures is a specialized program for answering query graphs of a particular shape; once the schema has been found, it can be evaluated for every element of a set. With extensions for Boolean connectives and repeated evaluations over sets, conceptual graphs could form the basis of a general data base query facility.

Example

Suppose a computer user typed in the question "What was Lee's age when hired?" If the system had a relation for all employees and their ages at time of hire, it could immediately find the answer. In most systems, however, that question would not be asked often enough to justify space for everybody's age when hired. To get an answer to that simple question, the user would have to find Lee's date of birth from one relation, find his date of hire

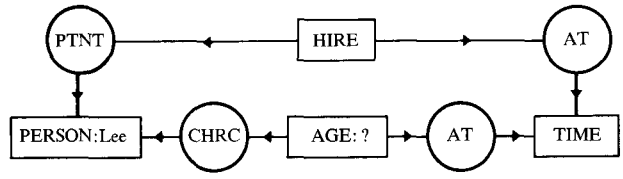

Figure 20 Query graph for "What was Lee's age when hired?"

from another relation, and then call a function to subtract the two dates. A system having conceptual graphs for its user interface, however, could accept the question as stated in English and determine for itself what relations and procedures to access.

Assume that the input analyzer can translate the user's question into the query graph shown in Fig. 20. The concept PERSON has the value Lee, and the value of AGE is to be determined. The conceptual relation CHRC has been borrowed from the TORUS system; it may be read "is a characteristic of." To determine which relation to insert between PERSON and AGE, the input analyzer would follow the rule that the preposition "of" or the possessive case marker "'s" indicates an unspecified relation between two nouns; the analyzer would search through its starting set of conceptual graphs and find CHRC as the default relation between AGE and PERSON. The conceptual relation AT is used for moments of time; the relation LOC is used for spatial locations. The input analyzer will translate phrases of the form "x when y" into a graph were x and y are shown to occur at the same time. Since "hired" is a passive participle, PERSON:Lee is linked as the patient of HIRE; for the question "What was Lee's age when hiring?" PERSON:Lee would be the agent of HIRE.

Since the question mark on AGE cannot be propagated anywhere, the system must find some schema to join to the query graph. It naturally starts with the concept AGE, which has the question mark, and searches for a schema in which AGE is functionally dependent on something that is computable. Figure 21 shows such a schema, which gives the definition of AGE. Associated with this schema are access links to a data base relation for a person's date of birth and an access procedure that computes the difference of two dates.

The definition of AGE is in second normal form, but not third normal form because there is a transitive dependency of AGE upon DATE and then upon PERSON. The BIRTH relation in the data base, however, may be stored in third normal form if convenient. This example illustrates the point that a schema may present a view of the data base different from the one that is actually stored. A complex schema can sometimes improve efficiency by reducing the number of steps in a data base inference.

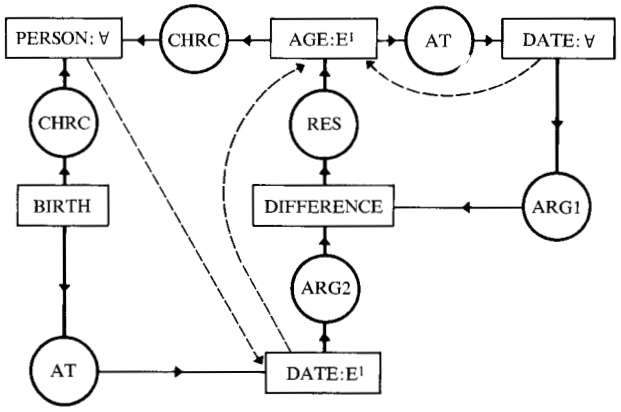

Figure 21 Schema for defining AGE.

Figure 22 Maximal common projection of Figs. 20 and 21.

Since DATE < TIME, Fig. 22 is a maximal common projection of the query graph with the schema in Fig. 21 having as a kernel the three concepts AGE. To compute the preference score for the join on this common projection, the system would add 3 points for the three concepts covered by the join, 3 more points because all three concepts are in the query graph, 1 point because a concept with a question mark is covered by a target concept, 1 point because a concept with a value is covered by a source, and 5 extra points because five concepts and conceptual relations of the query graph that had not previously been covered are covered by this join; the total preference score is 13. A join with the schema for HIRE (Fig. 16) would have a score of 12; it is almost as good, but the join with Fig. 21 is the preferred join.

The schematic join of Fig. 21 with the query graph is the working graph in Fig. 23. In forming the join, the universal quantifier on PERSON is replaced with the value Lee, and the universally quantified concept DATE replaces the indefinite concept TIME. The question marks are propagated from targets to sources of functional dependencies, according to algorithm C.

When the question mark reaches PERSON:Lee, the system can use the access links to find Lee's date of birth from the data base. This value will satisfy one argument of DIFFERENCE. The other argument, however, has a question mark that cannot be propagated further. The system must find a schema in which the concept DATE is functionally dependent on some concept that has a known (or computable) value. The schema in Fig. 16 for the data base relation HIRE meets these criteria, and a join with Fig. 16 is now the preferred join; its preference score would be 11. The schema in Fig. 21 could not be joined again to the working graph in the same position as before, because a maximal join would cause two existential quantifiers to be joined, and such joins are prohibited by the rules for a schematic join. When the schematic join of Figs. 16 and 23 is performed, PERSON:Lee is restricted to EMPLOYEE:Lee. The system must therefore check the data base to determine whether Lee is an employee; if he is, the system can derive Fig. 24.

When the schema for HIRE is joined to the graph, the question mark on DATE is propagated back to EMPLOYEE:Lee. Since the source of the functional dependency has a value, the system can access the data base to find Lee's date of hire. Now both arguments of DIFFERENCE have values, and the access procedure can compute Lee's age. Note that the HIRE schema contains information about the manager, which is irrelevant to the current question; since it is not needed, it would not be evaluated. A good property of this technique is that schemata can be arbitrarily complex, and the system will simply ignore the unneeded information.

Once the answer has been generated, the functional dependencies in Fig. 24 are no longer needed. But if the user wanted to know the age when hired for Smith, Jones, and others, then the system should save the dependencies. The concepts and conceptual relations define the meaning of the domains and their interrelationships; the functional dependencies are a data flow graph for computing the actual values. If the same function is to be evaluated repeatedly, the system could erase the current set of values and compute new values using the same functional dependency graph. For optimized execution, the system could even compile the functional dependencies into COBOL or PL/I.

Towards a natural interface

As a computer interface, English has been much maligned for its supposed wordiness. Part of the blame for the bad reputation must be borne by "English-like" languages, such as COBOL, which often do little but pad a formal notation with English prepositions. One query language, for example, has the following notation:

```
SKILFILE JOBCODE EQ 'ENG'
SKILCODE EQ 'GERMAN'
LOC EQ 'NY'
LIST EMPLOYEE MANNBR DEPT SVCYRS.
```

The language has a macro facility that can provide an English-like interface. When the necessary macros have been defined, the system can translate the following English sentence into the above notation:

From the skills inventory get me the name, man number, department, and years in service of the engineers with knowledge of German located in the New York area.

For a practical system, such a half-hearted approach to English is useless. Whereas macros generally reduce the amount of typing, this macro is 74% longer than the formal notation. Furthermore, the phrase "From the skills inventory" may be easier to read than "SKILFILE," but it is no easier to remember. The macro language requires every phrase to be defined by a unique rule; "years in service" is translated to "SVCYRS" by one rule, but "years of service" would require a separate rule. A more natural interface should accept the following request:

For engineers in New York who know German, list name, man no., dept., service years.

This sentence is shorter than the formal notation and is easier to read than the English-like macro. It is also easier for the user to learn because it omits the file name "SKILFILE" and does not use the odd abbreviations "MANNBR" or "SVCYRS." Because it omits the file name, it cannot be translated to the formal notation by a change of syntax; instead, the system must use semantic mechanisms like the conceptual schema, which determine system dependencies as a result of processing the English sentence.

Although the above example required fewer keystrokes for the English syntax than for the formal notation, the primary advantage of natural language is not in syntax but in semantics. During a conversation, the most important semantic features appear in dialogue, inference, and metalanguage:

- *Dialogue* Natural languages are used in a dialogue where both parties contribute to the conversation and ask questions to clarify or expand an incomplete message.
- *Inference* Most sentences can be short and simple because the listener is expected to fill in the "obvious" gaps. A complete theorem proving system is not necessary to understand English, but a technique for inferring the obvious is essential.
- *Metalanguage* English is its own metalanguage. It can be used either to talk about a subject or to talk about what can be said about the subject; it can therefore support prompting and help facilities in the same language used for queries and programming.

These three features of natural language are the areas where conceptual graphs can make the biggest contribution. Inference was emphasized in this paper, but conceptual graphs can also help in dialogues and metalanguage. Prompting methods have already been mentioned, and many of Heidorn's dialogues can be adapted to conceptual graphs with little more than a change of notation. For help facilities, the same conceptual schemata used to access the data base could be translated into English sentences to answer a user's questions about command and data formats. By using the same schemata for accessing data and for generating messages, the system would guarantee that the diagnostic and help facilities would always be consistent with the implementation.

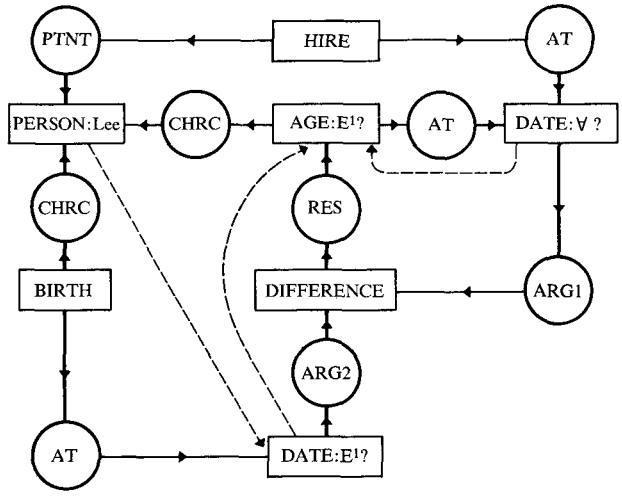

Figure 23 Working graph.

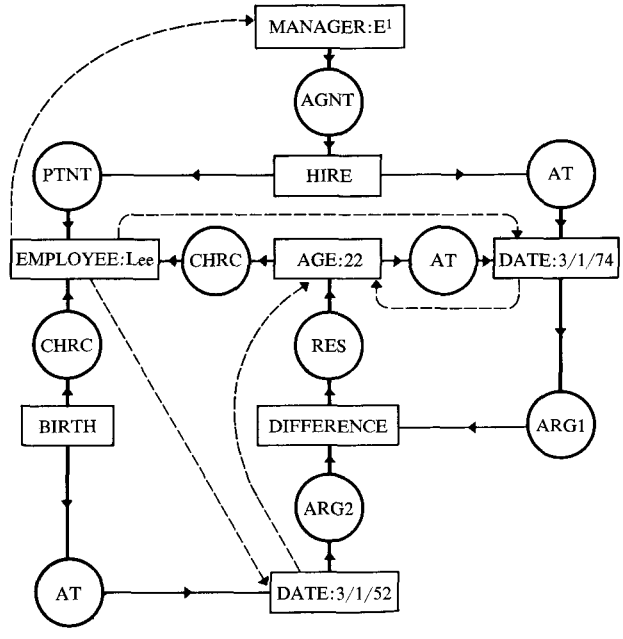

Figure 24 Final working graph.

Codd [33] emphasized the importance of a natural language interface for the casual user. Yet even the most experienced system programmers write comments in their native language because they find it more understandable and expressive than a programming language. The same properties that make English good for queries also make it good for programming: the abilities to support a dialogue for problem definition, to take care of machine dependent details without the user's assistance, and to answer questions about formats and conventions. A program to be executed in batch mode, where performance is critical, should not go through an interpretive natural language interface for every data base access; instead, the programmer could use that interface while *writing* the program and then have the system compile the accesses into a standard language for optimized execution.

Suppose a programmer asked "Give me a COBOL procedure to compute a person's age when hired." For this case, the system would not derive an answer graph from the data base. Instead, it would derive a general schema for computing age when hired for any person. Every schema in the schematic universe is a general function. For a one-shot query, the system uses the schema only once; but for automatic programming, the system could compute a schema as though it were answering a query and then translate the schema into a program. Instead of immediately executing the calls upon access procedures, the system could compile them into COBOL statements, which the programmer could insert into a program for batch execution. The programmer could use the query facility as a programming assistant that would handle the machine dependent details of data base accesses.

For a system of distributed computers, conceptual graphs can support a clean separation between the message handling and the data base accesses. Instead of calling the access procedures for each value requested for some target concept, algorithm C could be modified to make all of the accesses at the end of the derivation. If the data base processor is in a different computer from the input analyzer, all of the prompting and interaction could be handled by a local computer, and only a list of specific accesses would need to be sent to the data base processor. This separation would be especially useful if the data base had a relatively small number of relations, but a very large number of entries in each relation: a computer that did message handling would only need a few schemata that could be stored on an ordinary disk, but the data base processor would require a mass storage facility.

Conceptual graphs are precise enough to support logical inferences and data base accesses, yet they are rich enough and flexible enough to serve as a semantic basis for natural language. As a formal notation, the graphs can be used directly by the data base designer for representing and analyzing relationships between various domains in the data base; displays and plotters could present the graphs for a two-dimensional view of the data base. The designer could see the graphs on a display, but the end user would not be aware of them. Instead, conceptual graphs could support an interface that would let the user talk about familiar data in a familiar terminology without the need for special query languages and computer-oriented conventions.

Acknowledgment

The ideas in this paper have evolved over a long period of time and have benefited from suggestions by numerous friends and colleagues. I especially thank J. M. Cadiou, A. K. Chandra, E. F. Codd, R. L. Griffith, G. E. Heidorn, H. D. Mills, C. P. Wang, and M. Wilson from IBM, L. G. Creary, professor of philosophy at Case Western Reserve University, and the unknown referees who forced me to make this a better paper.

This paper contains some material from a forthcoming book entitled *Conceptual Structures: Information Processing in Mind and Machine* to be published by the Addison-Wesley Publishing Co. as one volume in the *IBM Systems Programming Series*.

References and notes

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 377 (1970).
2. N. Roussopoulos and J. Mylopoulos, "Using Semantic Networks for Data Base Management," *Proceedings of the International Conference on Very Large Data Bases*, ACM, New York, 1975, p. 144.
3. G. E. Heidorn, "Automatic Programming through Natural Language Dialogue," *IBM J. Res. Develop.* **20,** 302 (1976, this issue).
4. *Computer Models of Thought and Language*, edited by R. C. Schank and K. M. Colby, W. H. Freeman and Company, San Francisco, 1973.
5. *Representation and Understanding: Studies in Cognitive Science*, edited by D. G. Bobrow and A. Collins, Academic Press, New York, 1975.
6. M. M. Astrahan and D. D. Chamberlin, "Implementation of a Structured English Query Language," *Commun. ACM* **18,** 580 (1975).
7. R. F. Boyce, D. D. Chamberlin, W. F. King, and M. M. Hammer, "Specifying Queries as Relational Expressions: The SQUARE Data Sublanguage," *Commun. ACM* **18,** 621 (1975).
8. M. M. Zloof, "Query by Example," *AFIPS Conf. Proc., Nat. Comput. Conf.*, 1975 p. 431.
9. Note that an employee number is not a common subsort of EMPLOYEE and NUMBER. Instead, it is a subsort of NUMBER that bears a particular relationship to EMPLOYEE.
10. C. J. Fillmore, "The Case for Case," *Universals in Linguistic Theory*, edited by E. Bach and R. T. Harms, Holt, Rinehart and Winston, New York, p. 1.
11. In the more complete theory, presented in a forthcoming book [27], there is only one primitive conceptual relation, named LINK. All others are defined by the process of relational abstraction, which is essentially the lambda calculus

generalized to graphs. This paper, however, presents a restricted version of the formalism, in which the conceptual relations are fixed in advance.
12. A. Schmidt, "Über deduktive Theorien mit mehreren Sorten von Grunddingen," *Math. Ann.* **115,** 485 (1938).
13. H. Wang, "The Logic of Many-Sorted Theories," *J. of Symbolic Logic* **17,** 105 (1952).
14. J. J. Katz and J. A. Fodor, "The Structure of a Semantic Theory," *Language* **39,** 170 (1963).
15. For a study of the increased expressive power introduced by such a notation, see W. J. Walker, "Finite Partially Ordered Quantification," *J. Symbolic Logic* **35,** 535 (1970).
16. A standard technique in methods of theorem proving is to replace existential quantifiers with Skolem functions, cf. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *J. Assoc. Comput. Mach.* **12,** 23 (1965). As a general method of representing quantifiers, however, this technique fails because the resulting formula is no longer equivalent to the original. When an existential quantifier specifies a unique entity, however, then the lines of scope determine a unique function, and the functional form is equivalent to the quantifier notation.
17. W. A. Woods, "Procedural Semantics for a Question-Answering Machine," *AFIPS Conf. Proc., Fall Jt. Comput. Conf.*, 1968 p. 457.
18. T. Winograd, "Frame Representations and the Declarative-Procedural Controversy," *Representation and Understanding: Studies in Cognitive Science*, edited by D. G. Bobrow and A. Collins, Academic Press, New York, 1975.
19. R. W. Weyhrauch and A. J. Thomas, "FOL: a Proof Checker for First-Order Logic," *Memo AIM-235*, Stanford Artificial Intelligence Laboratory, NTIS #AD/A-006 898, 1974.
20. See Codd [1], p. 385, for a discussion of the connection trap.
21. J. E. J. Altham and N. W. Tennant, "Sortal Quantification," *Formal Semantics of Natural Language*, edited by E. L. Keenan, Cambridge University Press, p. 46.
22. E. F. Codd, "Further Normalization of the Data Base Relational Model," *Data Base Systems*, edited by R. Rustin, Prentice-Hall, Englewood Cliffs, N.J., 1972, p. 33.
23. A greater-than join of two data base relations could be represented by joining their schemata with the schema for the greater-than relation.
24. *Interim Report, ANSI/X3/SPARC*, Study Group on Data Base Management Systems, reprinted in *FDT*, ACM-SIGMOD **7,** 1975.
25. L. Wittgenstein, *Tractatus Logico-Philosophicus*, Routledge and Kegan Paul, London.
26. P. C. Wason and P. N. Johnson-Laird, *Psychology of Reasoning*, B. T. Batsford, London, 1972.
27. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, Mass., to be published.
28. W. J. Plath, "REQUEST: A Natural-Language Question-Answering System," *IBM J. Res. Develop.* **20,** 326 (1976, this issue).
29. S. R. Petrick, "On Natural Language Based Computer Systems," *IBM J. Res. Develop.* **20,** 314 (1976, this issue).
30. G. E. Heidorn, "English as a Very High Level Language for Simulation Programming," *Proceedings of the Symposium on Very High Level Languages, SIGPLAN Notices* **9,** (April 1974), p. 91.
31. Readers who are familiar with the resolution principle should note the parallels between schematic joins and Robinson's unification algorithm [16]. Both techniques involve universally quantified functional compositions and the set of all possible answer graphs is the analog of the Herbrand universe. Algorithm C may be interpreted as a heuristic method for finding an element of the Herbrand universe that answers the user's question.
32. Y. Wilks, "An Intelligent Analyzer and Understander of English," *Commun. ACM* **18,** 264 (1975).
33. E. F. Codd, "Seven Steps to Rendezvous with the Casual User," *Data Base Management*, edited by J. W. Klimbie and K. L. Koffeman, North-Holland Publishing Co., Amsterdam, (1974) p. 179.

Received April 2, 1975; revised February 23, 1976

The author is located at the IBM Systems Research Institute, 219 E. 42nd Street, New York, NY 10017.