# Parallel Knowledge-Based Information Retrieval on the NON-VON Machine

David Elliot Shaw

Computer Science Department
Columbia University
New York, NY 10027

August 1981

**Abstract**

NON-VON [Shaw, 1979; Shaw, et al., 1981] is a highly parallel machine adapted to the extremely efficient support of certain operations that appear central to a wide range of large-scale, knowledge-based information processing tasks. In order to demonstrate the utility of the NON-VON architecture for large-scale Artificial Intelligence applications, we have implemented a simple *knowledge-based retrieval system*, with the primitive NON-VON machine instructions emulated in software. The system compares KRL-like descriptions [Bobrow and Winograd, 1977] with the contents of what would in practice be a very large database, retrieving all "matching" descriptions; the criteria for description-matching require deductive inference over a domain-specific knowledge base. This paper describes the essential mechanisms employed in our experimental knowledge-based retrieval system.

*This research was supported in part by the Defense Advanced Research Projects Agency under contracts MDA903-77-C-0322 and N00039-80-G-0132.*

## 1. Introduction

The central focus of our recent research has been the investigation of highly parallel non-von Neumann machine architectures adapted to the kinds of operations that appear central to the operation of a broad class of large-scale knowledge-based systems. Our approach is based on an architecture [Shaw, 1979] that supports the highly efficient evaluation of the most "difficult" set theoretic and relational algebraic operators. The machine comprises a tree-structured Primary Processing Subsystem (PPS), which we are implementing using custom nMOS VLSI chips, and a Secondary Processing Subsystem (SPS) incorporating modified, highly intelligent disk drives. Our initial efforts, reported in a recent Stanford doctoral dissertation [Shaw, 1980a], have yielded:

> 1. The "top-level" architectural specification of a highly parallel machine which we have since come to call NON-VON.
>
> 2. An analysis of the time complexity of the essential parallel hardware algorithms to be executed on the NON-VON machine in the course of large-scale, meaning-based data manipulation.
>
> 3. The implementation of an operational *knowledge-based information retrieval system*, demonstrating the use of NON-VON (emulated in software) in support of a very high level descriptive formalism based on the language KRL [Bobrow and Winograd, 1977].

Portions of the NON-VON machine are now in the earliest stages of physical implementation as part of a cooperative effort involving the new Computer Science Department at Columbia and the Knowledge Base Management Systems Project at Stanford. Details of the NON-VON hardware are presented elsewhere [Shaw, et al., 1981], and will not be described here. The central focus of the present paper is the structure and function of the demonstration system which we have implemented as a vehicle for illustrating the use of NON-VON in a simple, but characteristic large-scale AI application. This system demonstrates the manner in which NON-VON might be utilized in support of the highly efficient retrieval of records from very large databases in applications where the criteria for description-matching require deductive inference over a domain-specific "knowledge base". Our demonstration system, which was implemented in MACLISP on the DEC PDP-10 at the Stanford Artificial Intelligence Laboratory, emulates the primitive operations of the NON-VON machine in software.

Our demonstration system uses a restricted first-order predicate calculus as a sort of "intermediate form", bridging the gap between the semantics of our KRL-like description language and certain operators of a *relational algebra* having particular importance in the computational task of *logical satisfaction*. These relational algebraic operators are evaluated in parallel by the NON-VON hardware, yielding a significant improvement over the best methods known for performing equivalent operations on a conventional computer system. In this paper, we will trace the operation of our knowledge-based retrieval system from the level of KRL-like descriptions, through that of the predicate logic-based intermediate form, and down to the level of the primitive relational algebraic operators, which are evaluated in parallel on the NON-VON hardware. We begin with a discussion of a general and important class of problems which may be thought of as *conceptual matching* tasks, of which the knowledge-based information retrieval application is a particular instance.

## 2. The General Conceptual Matching Problem

Many information processing tasks performed by men and machines alike involve the process of *matching*, by which a correspondence is assigned between members of two sets of entities. The criteria for certain sorts of matches are quite simple to describe. Letters are routinely matched with mailboxes, for example, and Congressmen with constituents, according to straightforward algorithms based on simple, single properties of the entities in question. By contrast, our demonstration system may be thought of as concerned with a more interesting class of tasks which might be termed *conceptual* matching problems. This more demanding sort of task, which is nonetheless a common part of our cognitive experience, involves the assignment of a correspondence between entities which humans perceive as being highly and systematically structured, based on selected significant characteristics of those structures.

A surprisingly large share of the kinds of information processing activities with which both human and automated data processors are charged may be viewed as involving various kinds of conceptual matching problems. Although meaning-based matching may, in a COBOL-based inventory control system for airplane parts, be deeply embedded within program loops and sort routines, and hence difficult to recognize as such, the fact remains that a large proportion of the CPU cycles which will be expended in any particular run might be thought of as identifying appropriate records for manipulation on the basis of domain-specific criteria—in our example, criteria involving, say, airplanes, motors and aircraft-specific part-whole relationships. In the general spirit of much of the recent work on knowledge-based systems, we might ask for the ability to describe these matching criteria, along with the domain-specific entities and relationships on which they are based, in a very direct, modular, easily understandable manner, mapping salient facts, rules and relationships onto independently expressable assertions within the programming system.

Previous work in the field of artificial intelligence offers a rich set of knowledge representation and matching techniques which might be employed in the pursuit of this general approach to the problem of conceptual matching. The kinds of applications with which we are most concerned, though, are those in which the quantity of data to which conceptual matching techniques must be applied may be quite large. More specifically, our doctoral research attacks the problem of matching a given *pattern* description against the members of what may be a *very large* set of candidate *target* descriptions according to meaning-based criteria. The potential size of the collection of target descriptions imposes special constraints on the sorts of conceptual matching techniques that might be successfully applied in practice.

## 3. Knowledge-Based Information Retrieval

The particular instance of the conceptual matching task that we have chosen for our demonstration system is borrowed from the general paradigm of *information retrieval*, and is itself most easily exemplified by the *document retrieval* (more accurately, *reference* retrieval) application. In an ordinary document retrieval system, a collection of *target documents*—all the books in a computer science library, for example—is first *indexed* by associating a target description with each document in the collection. The end user of the system, who we will call the *searcher*, then prepares a *pattern description* which embodies some of the

salient characteristics of the sorts of documents in which he is interested. The system then compares the pattern description with the candidate target descriptions in the collection, returning all targets that "match" according to certain prespecified criteria.

It is the nature of these criteria that distinguishes the behavior of a *knowledge-based* information retrieval system, and indeed, of a system for conceptual matching in general. In such applications, it is not possible in general to decide whether a match should succeed in a strictly mechanical, "syntactic" manner; instead, the acceptability of a match may depend on domain-specific entities and relationships, and on deductive inferences over these entities and relationships. In the case of the computer science library, for example, the system might be required to "know about" such entities as *computers*, *algorithms*, *programmers*, and *storage devices*.

Certain *characteristic attributes* of these entities (the *storage medium* attribute, whose values differ for different kinds of *storage devices*, for example) might also be included in this domain-specific knowledge. Among the typical kinds of relationships that might be embodied in the *knowledge base* of such a system is the fact that a *tape drive* is a particular kind of *storage device* whose *storage medium* is always *magnetic tape*; one simple deductive inference involving this relationship might establish the fact that a pattern description in which the subject of a document is described as involving a *storage device* with *magnetic tape* as its medium would be satisfied by a target description in which a *tape drive* appeared in the corresponding position.

Let us now briefly examine the manner in which such a knowledge-based information retrieval system might be used in practice. In contrast with an ordinary information retrieval system, three distinct classes of users would be involved in the operation of a knowledge-based retrieval system. In addition to the searchers and indexers, a third class of users having expertise in the subject areas of the documents to be indexed would be required to formulate and encode the sorts of domain-specific knowledge described above for use in indexing and retrieval. Members of this third class of users, which has no analogue within the context of the conventional information retrieval system, might be called *knowledge engineers*. Our primary concern in this paper, however, will be with the process of retrieval by searching end-users, under the assumption that the knowledge base has previously been constructed and all documents in the collection indexed.

While space does not permit a detailed discussion of the weaknesses of existing information retrieval systems, and of the manner in which our knowledge-based retrieval system addresses these limitations, it is worth mentioning that our approach would offer the greatest advantages in the case where highly specific targets are to be retrieved from among a large class of "conceptually heterogeneous" documents. Phrased differently, knowledge-based retrieval methods should prove most critical in the context of tasks in which the semantic criteria for satisfaction of a user's request are meaningful for only a comparatively small subset of the target collection. A very ambitious example of such a task might be the selection of a specialized journal article whose relevance might only be apparent to, say, a graduate student working in the field who had read the paper, from among the set of all documents in a large university collection.

## 4. A System for Knowledge-Based Retrieval

In order to demonstrate the way in which a NON-VON-like machine might be used in an actual AI application,

we have implemented a simple knowledge-based information retrieval system having the basic structure outlined above. In the interest of applicability to problems other than our sample document retrieval application, however, the system we have implemented is in fact somewhat more general in one respect than suggested by the above discussion. Specifically, the rules defining the semantics of matching within the document description language have not been embedded inextricably within the code of the retrieval system, but have instead been explicitly formulated as an independent, separable set of axioms expressed in restricted first-order predicate calculus. Specification of the match semantics in the form of a separate set of declaratively specified rules also contributes to the flexibility of our demonstration system, in that the matching axioms could be easily modified to reflect changes in the description language or in the rules for description matching without affecting the behavior of the system as a whole through modifications of the code itself. More accurately, then, the knowledge-based retrieval system which we have implemented may be thought of as making reference to three conceptually distinct "databases": a target collection, a domain-specific knowledge base, and a *match specification* defining the matching semantics of the knowledge-based description language.

As will be seen shortly, the flexible approach to the the specification of match semantics that we have outlined, together with the capacity for the use of domain-specific knowledge in evaluating the success of potential matches, supports a very powerful and highly general set of capabilities not available in a conventional information retrieval system. It is not difficult to construct a scenario in which this sort of generalized knowledge-based retrieval system might offer a number of important practical advantages by comparison with a conventional information retrieval system. Unfortunately, there is a fundamental respect in which our presently operational demonstration system, which runs on conventional computer hardware, would not be practical for use in an actual application involving a large target collection.

Specifically, the demonstration system relies very heavily on the execution of several operations which, on a von Neumann machine, are quite expensive when the operands comprise a large amount of data. As it happens, it is precisely in the case of a very large target collection that our knowledge-based approach is of the greatest potential utility. If this approach is to be regarded as a candidate for practical application, consideration must thus be given to any alternatives to the von Neumann machine architecture that might support the more efficient execution of these operations.

On the NON-VON machine, the most expensive low-level operations involved in our approach to knowledge-based retrieval—in particular, the most computationally expensive primitive operators of a *relational algebra*—may be evaluated in a highly efficient manner. Based on a hierarchy of inteiligent storage devices, this architecture in fact permits an $O(\log n)$ improvement (with very favorable constant factors) in time complexity over the evaluation methods used for these operators on a conventional computer system, without the use of redundant storage, and using currently available and potentially competitive technology. Although it was not our original goal in pursuing this research, these results have recently begun to attract attention within the database management community by virtue of the important role played by these "difficult" relational algebraic primitives within database management systems based on the relational model of data [Codd, 1970].

Because of this connection with the concerns of relational database systems, together with the close relationship between our architectural work and earlier research on specialized hardware for database management, NON-VON is sometimes referred to as a (relational) *database machine*. On close examination, this more immediate byproduct of our research is not quite the coincidence it might seem at first glance; indeed, the reasons these operations have proven to be so central to our own work is closely related to at least one of the reasons for their frequent appearance in the relational database literature, as will be seen shortly.

## 5. Organization of the Retrieval System

The nature of the problem we have chosen to attack, and of the approach to its solution that we have adopted, imposes what might be described as a *vertically integrated* organization on this paper. Although we are concerned with only a single task, which is itself reasonably well defined and manageable in scope, the methodical reader will be following our approach to its solution along a long route extending from the level of very high level descriptions, through the arena of logical formula manipulation, and down to the domain of actual parallel operations in hardware. It has been our experience in describing this research before a number of audiences that the "vertical distance" which must be covered in its exposition may make it difficult to retain the overall structure of our work while attending to the details of each of these "layers".

In an attempt to mitigate this difficulty, we have attempted to illustrate in Figure 5.1 the vertical structure of our system, which is closely mirrored by the organization of this paper; the reader may wish to refer back to this figure periodically for purposes of reorientation (or at very least, reassurance). At the top level, documents and domain-specific knowledge are represented using a *knowledge-based description language*. Examples of this language appear immediately to the right of the phrase "knowledge-based description language" in Figure 5.1; they need not be studied carefully at this point, but are presented to give some feeling for the kind of information they embody. At the bottom of our vertical chain, the primitive operations of a relational algebra (again illustrated immediately to the right of the corresponding title) are interpreted by the NON-VON machine.

Most of the activity of our actual demonstration system, however, takes place in the middle portion of our illustration—the part dealing with formulas expressed in a restricted first order logic. As we shall see, the strength of predicate calculus as an "intermediate form" for our system derives from its flexibility as a descriptive tool, on the one hand, and from its interesting and useful relationship to the relational algebraic primitives, on the other. In our demonstration system, all descriptions, both target and pattern, are first converted into a special *normalized* form, and the resulting data structures manipulated according to the rules embodied in the match specification (which, we recall, are also expressed in predicate logic) to locate all matching target documents. The details of these manipulations are embodied in an algorithm called *LSEC* (for *Logical Satisfaction by Extensional Constraint*), whose primitive (relational algebraic) operations are implemented directly as highly parallel macroinstructions of the NON-VON machine.

The remainder of this paper describes the structure and function of the demonstration system, from the level of knowledge-based descriptions through the invocation (but not the evaluation) of the relational

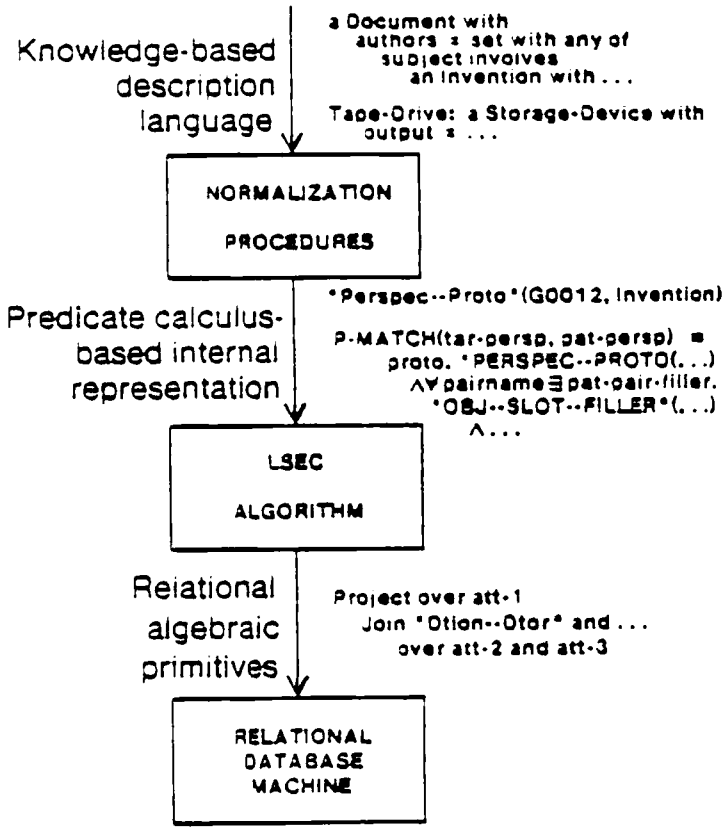



Figure 5.1 'Vertical Organization' of the System

algebraic primitives which form the macroinstructions of the NON-VON machine. Section 6 introduces and exemplifies the knowledge-based description language. In Section 7, such aspects of the relational model of data, its relationship to predicate calculus, and its use as a database query language as are essential to our own work are presented and illustrated; the application of these techniques in our own work is then introduced. Finally, Section 8 describes the LSEC algorithm, whose execution is central to the operation of the working demonstration system.

## 6. The Knowledge-Based Description Language

The knowledge-based description language that we have adopted for use in our demonstration system is based closely on a simple subset of the knowledge representation language *KRL* [Bobrow and Winograd, 1977]. Rather than begin with a formal treatment this description language, we have chosen to first exemplify its descriptions with a simple (albeit rather contrived) example of a hypothetical document description:

```
a Document
  authors
      set-with-all-of
        Thompson
        Walters
      set-of
        an Engineer
  countries-of-publication
      set-with-any-of
        USA
        Great-Britain
  subject
      involves
        an Invention with
          purpose
              or
                Power-generation
                Power-transmission
  printing-dates
      set-with-exactly
        1959
        1962
a Textbook
```

(In the actual description language, parentheses are used extensively for purposes of grouping; we have taken the liberty of omitting the parentheses in this and most other examples, using indentation to convey the same structural information.)

The above *description* is made up of two *descriptors*—one of which characterizes its referent as a *Document*, and the other more specifically as a *Textbook*. The use of descriptions composed of multiple descriptors, each reflecting a different way of viewing the same real-world entity, is an essential part of our knowledge-based description language.

In all, there are eight types of descriptors that may be included in the descriptions of our language, each of which is illustrated in the above example. The two that we have just mentioned are called *perspectives*.

Each perspective includes a single *prototype* (*Document* and *Textbook*, in the examples under consideration) with which is associated a particular set of characteristic *slots*. One or more of these slots may be filled with another description in any particular instance of a perspective. In our example, the *authors* and *countries-of-publication* slots (among others) are both filled. Alternately, a perspective may appear without any of its slots filled, as in the case of the *Textbook* descriptor. A target perspective is deemed to "syntactically" match (see Subsection 7.7) a pattern perspective exactly in the (recursively defined) case where the prototypes are identical, and where every slot that is filled in the pattern has a matching description in the corresponding position in the target.

Several other types of descriptors are embedded within the slots of our top-level example description. The simplest of these are the *individual* descriptors *Thompson*, *Walters*, *USA*, *Great Britain*, *Power-generation*, *Power-transmission*, and the two dates. Individuals are the only "atomic" (i.e., non-decomposable) descriptor type, and may, to a first approximation, be thought of as corresponding to single, specific entities in the "real world". Individual descriptors in the pattern description are matched only by identical individual descriptors in the corresponding target descriptions.

The semantics of the four "set types"—**set of**, **set with exactly**, **set with all of** and **set with any of**—conform reasonably well to the meanings suggested by their names. In our example, the authors are described as a set whose members are all engineers, and which includes both Thompson and Walters in particular (possibly along with others). The set of countries in which the hypothetical document is published is required to include either the USA, Great Britain or both. Finally, the document is to bear exactly two printing dates: 1959 and 1962. For the most part, the rules for matching descriptors of the various set types are fairly intuitive; for details, the reader is referred to the matching axioms themselves, which are included as an appendix to our doctoral dissertation [Shaw, 1980a].

*Involves* descriptors support description-matching on the basis of *structural embedding*. Specifically, an **involves** descriptor in the pattern which has a description $D$ as its argument will successfully match against any description in the corresponding position within the target which either itself matches $D$ or has some *subdescription* which matches $D$. In intuitive terms, description $D_1$ is called a subdescription of description $D_2$ if $D_1$ is lexically nested at any depth within the body of $D_2$, either as a slot filler (in the case of a perspective) or an argument (in the case of a set type or disjunctive descriptor). For a more precise definition, the reader is again referred to the matching axioms.

Finally, the *disjunctive* descriptor is defined to match against a given target descriptor if *either* of its two arguments (the individuals *Power generation* and *Power transmission*, in our example) would match against that target descriptor.

Figure 8.1 defines the precise syntax of our knowledge-based description language. Consistent with the usual conventions, the names of nonterminals are enclosed in angle brackets, with alternative choices separated by vertical bars. Braces are used as meta-symbols for purposes of grouping; parentheses, on the other hand, are symbols of the description language itself. A "plus" superscript is used to indicate that a syntactic element may occur one or more times, while a superscripted asterisk marks an element which may occur zero or more

⟨ *description* ⟩ ::= (⟨ *descriptor* ⟩ $^{*}$ )

⟨ *descriptor* ⟩ ::=
⟨ *individual* ⟩
⟨ *perspective* ⟩ |
⟨ *set-of descriptor* ⟩ |
⟨ *set-with-exactly descriptor* ⟩ |
⟨ *set-with-all-of descriptor* ⟩ |
⟨ *set-with-any-of descriptor* ⟩ |
⟨ *involves descriptor* ⟩ |
⟨ *disjunctive descriptor* ⟩

⟨ *individual* ⟩ ::= ⟨ *LISP identifier* ⟩

⟨ *perspective* ⟩ ::= ⟨ { **a** | **an** } ⟨ *prototype* ⟩ ⟨ *filler pair* ⟩ $^{*}$ )

⟨ *prototype* ⟩ ::= ⟨ *LISP identifier* ⟩

⟨ *filler pair* ⟩ ::= (⟨ *slot name* ⟩ ⟨ *description* ⟩)$^{*}$

⟨ *set-of descriptor* ⟩ ::= ( **set-of** ⟨ *description* ⟩)

⟨ *set-with-exactly descriptor* ⟩ ::= ( **set-with-exactly** ⟨ *description* ⟩ $^{+}$ )

⟨ *set-with-all-of descriptor* ⟩ ::= ( **set-with-all-of** ⟨ *description* ⟩ $^{+}$ )

⟨ *set-with-any-of descriptor* ⟩ ::= ( **set-with-any-of** ⟨ *description* ⟩ $^{+}$ )

⟨ *involves descriptor* ⟩ ::= ( **involves** ⟨ *description* ⟩)

⟨ *disjunctive descriptor* ⟩ ::= ( **or** ⟨ *description* ⟩ $^{+}$ )

Figure 8.1 Syntax of the Knowledge-Based Description Language

times.

In the course of examining a number of actual documents chosen from the domain of computer science, we were able to identify several kinds of deductive inference mechanisms that might well prove useful in a working knowledge-based retrieval system. In our actual demonstration system, however, only a single, relatively simple form of inference—based on *antecedent-consequent* (or simply *AC*) *pairs*—was chosen for purposes of demonstrating our approach to meaning-based matching. Each antecedent-consequent rule expresses a relationship between two descriptions—the *antecedent* and the *consequent*—which may be thought of either in terms of *implication* or *specialization*. Under the first interpretation, the fact that a given "real-world" entity may be appropriately described by the antecedent description is taken to *logically imply* that the entity in question may also be described by the consequent. From the (formally equivalent) alternative viewpoint, the antecedent is considered to be a *special* case of the consequent.

Our system places no restrictions on the form of the antecedent and consequent descriptions. In particular, it is possible to formulate AC pairs which express *simple generalization* relationships (the fact that a *Tape drive* is a special kind of *Storage device*, for example), and *elaborated generalizations* (e.g., that a *Tape drive* is a *Storage device* whose *medium* is *magnetic tape*), along with a number of more general relationships. It should be noted that the relationship expressed in AC pairs is *transitive*. As it happens, the transitivity of implication is one of the characteristics which impose certain special requirements on the matching procedures. In section 7, we will see how this sort of requirement is accomodated by the LSEC algorithm.

## 7. Logic, Relations and Retrieval

As noted earlier, predicate logic and the relational algebra together provide a critical link between our knowledge-based retrieval language and the underlying NON-VON machine. In this section, the connections between these two mathematical tools, and their use in both conventional database management and knowledge-based retrieval tasks, will be introduced. Subsection 7.1 reviews those aspects of the relational model of data that are essential to our research interests. In Subsection 7.2, we illustrate the simple procedure by which pattern and target descriptions, along with all antecedent-consequent pairs to be included in the knowledge base, are *normalized*—that is, converted into relational form for manipulation using the logical and relational algebraic tools. The relationship betwen predicate logic and the relational algebra is introduced in Subsection 7.3, using a simple example involving the evaluation of an ordinary (as opposed to knowledge-based) database query for purposes of illustration. The full set of relational algebraic primitives of concern to our research are described more systematically in Subsection 7.4. The remaining three subsections describe the tools used to define the rules for matching knowledge-based descriptions, ending with a discussion of the 28 matching axioms. These axioms comprise the match specification actually used in our working demonstration system, and appeared as an appendix to our Ph.D. thesis [Shaw, 1980a].

### *7.1 The relational model of data*

The relational model of data, as typically formulated by researchers in database management systems, has its

roots in two seminal papers by Codd [1970, 1972]. In this context, the term *relation* is used to denote a set of structured entities called *tuples* which, within a single relation, share a common *attribute structure*. We may more rigorously define a *normalized relation* of degree $n$ as a set of *tuples*, where each tuple is an element of the cartesian product of $n$ (not necessarily distinct) sets—called the *underlying domains* of the relation—of non-decomposable entities. Since relations are sets, we may refer to the number of elements—in this case, tuples—in a relation as the *cardinality* of that relation. Intuitively, relations may be thought of as "tables", in which each "row" represents one tuple and each column represents one of the $n$ *(simple) attributes* of that relation.

It is conventional to either name or number the attributes of a relation for convenience in referring either to a whole "column" of the relation, or to the *value* of the attribute in question within a particular tuple. In some discussions (and in particular, in much of this paper), it is also useful to group several attributes (some possibly repeated) together, referring to them jointly as a *compound attribute*.

The term *normalized* reflects the "type distinction" between underlying domain elements, which may serve as the values of attributes, and tuples and relations, which may not. A single tuple thus can not be used to directly represent a hierarchically nested data structure. (Our use of the term *normalized* corresponds to what is now commonly referred to as *first normal form* in the database management community.) The example binary (order two) relation shown below expresses a part-whole relationship between airplanes and their (hypothetical) constituent parts:

| *PRODUCT* | *PART* |
|---|---|
| *DC-10* | *wheel* |
| *DC-10* | *engine-mount* |
| *DC-3* | *oxygen-mask* |
| *DC-10* | *oxygen-mask* |
| *DC-10* | *radio* |

This example relation, which will be used several times in the remainder of this section, should be interpreted as asserting that a DC-10 includes as parts a wheel, an engine-mount, an oxygen mask and a radio, while a DC-3 includes an oxygen mask. Note that each of the entries in the table are "primitive domain elements"; rather than represent the fact that the DC-10 contains each of these four parts by associating an explicit list of parts with a single entry for DC-10, the airplane's identity must be repeated for each part in order to satisfy the normalization requirements. As we shall see in the next subsection, however, data structures expressed as lists, trees, etc. may be easily "normalized" in a simple and mechanical fashion.

### *7.2 Normalization of knowledge-based descriptions*

Upon casual inspection, our knowledge-based descriptions already appear to have the basic structure of relations. To be sure, the attribute/value structure observed in the tuples of a relation are close analogues, both in form and function, of the slot/filler constructs of our knowledge-based description language. Note, however, that while the value of a simple attribute within a given tuple must be a non-decomposable element

of a particular primitive domain, the filler of a slot may itself be a general description, possibly containing its own embedded subdescriptions, and so on. Furthermore, the inclusion within a slot of descriptors of any type but individual, along with the capability for multiple abstraction through the use of multiple-descriptor descriptions, also violates the constraints of relational normalization.

The capacity for multiple viewpoints, and for an arbitrary degree of structural embedding, is essential to the kind of knowledge-based retrieval with which we are concerned. This difference between the "schematic" structures embodied in our knowledge-based descriptions, on the one hand, and the tuple of a normalized relation, on the other, is thus quite fundamental to our approach to knowledge-based retrieval. Consequently, although it is certainly tempting to apply the tools of logic and relational algebra more directly to the task at hand, all knowledge-based descriptions must be converted to normalized relational form before this machinery can be applied. We will thus now examine the procedure by which our demonstration system normalizes the knowledge-based descriptions prior to the application of logical and relational algebraic mechanisms.

Normalization of the *external* (unnormalized) form of a description involves the addition of new tuples to various relations in the single *extensional database* which embodies all the descriptive information manipulated by the system. (The term "extensional" will be motivated in Subsection 7.5.) Consider, for example, the following description, which is composed of a single perspective descriptor having *Storage-device* as its prototype and *Mag-tape* as the filler of its *medium* slot:

a *Storage-device*
  *medium* = *Mag-tape*

The *internal* (normalized) form of this description comprises the following set of six tuples, which would be added to five distinct relations in the extensional database (since the relation •DTOR--DTYPE• acquires two new tuples):

•*DTION--DTOR*• *(G001, G002)*
•*DTOR--DTYPE*• *(G002, Perspective)*
•*PERSPECTIVE--PROTOTYPE*• *(G002, Storage-device)*
•*OBJECT--SLOT--FILLER*• *(G002, Medium, Storage-device)*
•*DTOR--DTYPE*• *(G003, Individual)*
•*INDIVIDUAL-DTOR--INDIVIDUAL*• *(G003, Mag-tape)*

(In the interest of brevity, the word *DESCRIPTION* has been abbreviated as *DTION*, *DESCRIPTOR* as *DTOR* and *IMPLIES* as *IMP*, as in our actual system. We will also be using the abbreviations *PAT*, for *PATTERN*, and *TAR*, for *TARGET*.)

Note that, in order to explicitly indicate to which relation the tuples in question will be added, we have used a somewhat different representation scheme, called *intensional* notation, for representing the normalised form of our example description. In intensional notation, the name of the relation appears first, followed by a parenthesized list which specifies (in a predetermined order associated with the relation in question) the value of each simple attribute in the particular tuple being represented. Particular tuples in intensional notation thus have the form of logical *predicates* (with constant arguments), which, as we shall see in the following subsection, will permit their incorporation in well-formed formulæ of a restricted first order predicate calculus. The significance of the asterisks surrounding the relation name in intensional notation will be discussed in Subsection 7.5.

While it will not be necessary to consider the detailed semantics of each of the sixteen relations which are employed in our demonstration system, and their relationship to the external descriptions from which they are generated, it is worth noting a few characteristic differences between the external and internal forms of a description. Perhaps most obvious is the appearance within the internal form of a number of primitive elements (those of the form *G00n* in our example) which may be said to "carry meaning" solely on the basis of their relationship with other, explicitly named, primitive elements. (This technique, along with the naming convention for such elements, should be familiar to LISP programmers as an analogue of the "GENSYMed" atom.)

More interesting, however, is the appearance within the internal form of certain "naturally" named primitive elements which nonetheless do not appear explicitly within the original external description. In particular, note that while some of the named elements correspond to *semantically* meaningful tokens found in the external description, others (in our example, *Perspective* and *Individual*) serve a purely *syntactic* function, explicitly representing in normalized relational form the structural information that was implicit in the syntax of our knowledge-based description language. Loosely speaking, the process of normalization involves a *flattening* of the original tree structure of the external description, together with an *expansion* of the original description to explicitly represent syntactic information.

Since descriptions serve roles as patterns, targets, antecedents or consequents within our system, tuples must be added to the extensional database to reflect these roles. If our example were in fact a target description (which of course seems unnatural in this case), this connection would be drawn by the addition of the single tuple

*•TARGET--COLLECTION• (G001, ( collection name ))*

to the *•TARGET--COLLECTION•* relation. If, on the other hand, the example description were the consequent of an AC-pair whose antecedent description was a *Tape-drive*, the following four tuples would instead be added to the extensional database:

•ANTECEDENT--CONSEQUENT• (G004, G001)

•DTION--DTOR• (G004, G005)

•DTOR--DTYPE• (G005, Perspective)

•PERSPECTIVE--PROTOTYPE• (G005, Tape-drive)

Note that the knowledge base, which was characterized in Section 4 as a separate database for pedagogical reasons, is actually implemented as part of the single extensional database, together with all candidate target descriptions in the collection and the internal form of the pattern description used in the course of a particular retrieval session.

*7.3 Logic as an effective query language*

In this subsection, we will examine some fundamental connections between the descriptive capabilities of the first order predicate calculus and the use of relational algebraic operators to make such logical descriptions computationally effective. While we might have chosen to exemplify this relationship using a realistic example of the behavior of our demonstration system, the relative complexity of the knowledge-based matching operation would have made it unnecessarily difficult to identify the essential import of such an example. We have thus chosen to illustrate the central aspects of our use of mathematical logic and relational algebra through the use of a simple example typical of conventional (as opposed to knowledge-based) retrieval in a relational database management context. Of interest in this regard is the observation that a wide range of typical database queries can be expressed in the form of

1. a well-formed formula in the first-order predicate calculus (without function symbols), together with
2. a list of free variables of that well-formed formula, all possible joint instantiations of which are to be returned as the value of the query.

This observation is best illustrated by considering a simple example. To this end, we have supplemented the hypothetical •*PRODUCT--PART*• relation introduced in Subsection 7.1 with a new •*CUSTOMER--PRODUCT*• relation, which is to be interpreted as asserting that American Airlines has purchased one or more DC-3's and one or more DC-10's, while Western Airlines owns one or more DC-10's only:

| *CUSTOMER* | *PRODUCT* |
|---|---|
| *American* | *DC-3* |
| *Western* | *DC-10* |
| *American* | *DC-10* |

| *PRODUCT* | *PART* |
|---|---|
| *DC-10* | *wheel* |
| *DC-10* | *engine-mount* |
| *DC-3* | *oxygen-mask* |
| *DC-10* | *oxygen-mask* |
| *DC-10* | *radio* |

Suppose we wished to produce a list pairing each airline with each part which could be found in the inventory of that airline, independent of the identity of the model (or models) of airplane which accounted for the presence of that part within the inventory of that airline. In relational terms, we should like the result of our query to be a new binary relation having two attributes—one ranging over the same primitive domain as the *CUSTOMER* attribute of the •*CUSTOMER--PRODUCT*• relation, and one over that of the *PART* attribute of the •*PRODUCT--PART*• relation—each of whose tuples satisfies the relationship in question.

Such a query might be expressed in the following way using the language of first-order logic:

$$(x, z) : \exists y.$$
$$(\bullet CUSTOMER\text{--}PRODUCT\bullet\ (x, y) \wedge$$
$$\bullet PRODUCT\text{--}PART\bullet\ (y, z))$$

where the *result variables* are specified in a parenthesized list which is prefixed to the well-formed formula and followed by a colon. Here, we are assigning a correspondence between the *predicate* •*CUSTOMER--PRODUCT*• and the *relation* having *CUSTOMER* and *PRODUCT* as its attributes, and similarly, between the •*PRODUCT--PART*• predicate and the other relation. The *result* of this query is defined to be a new relation, having two attributes (corresponding to the two free variables $x$ and $z$ specified in the result variable list), whose tuples enumerate all of the combinations of one instantiation of $x$ and one instantiation of $z$ for which the well-formed formula is true for *some* instantiation of the existentially quantified variable $y$.

Let us now consider how the result of such a query might be computed. All possible combinations of tuples, one of which is chosen from the •*CUSTOMER--PRODUCT*• relation, the other from •*PRODUCT--PART*•, whose *product* attributes share a common value are identified, as illustrated below by the lines connecting tuples of the two argument relations:

| *CUSTOMER* | *PRODUCT* |
|---|---|
| *American* | *DC-3* |
| *Western* | *DC-10* |
| *American* | *DC-10* |

| *PRODUCT* | *PART* |
|---|---|
| *DC-10* | *wheel* |
| *DC-10* | *engine-mount* |
| *DC-3* | *oxygen-mask* |
| *DC-10* | *oxygen-mask* |
| *DC-10* | *radio* |

For each such matching pair of tuples, a new tuple is created by concatenating the two and eliminating one copy of the common *PRODUCT* attribute, thus yielding the following ternary relation:

| CUSTOMER | PRODUCT | PART |
|---|---|---|
| American | DC-3 | oxygen-mask |
| Western | DC-10 | wheel |
| Western | DC-10 | engine-mount |
| Western | DC-10 | oxygen-mask |
| Western | DC-10 | radio |
| American | DC-10 | wheel |
| American | DC-10 | engine-mount |
| American | DC-10 | oxygen-mask |
| American | DC-10 | radio |

The operation that we have just described provides our first example of a relational algebraic operation, which is called the *join* (more precisely, the *natural join*) of the two argument relations. The *PRODUCT* attributes of each of the two argument relations are together referred to as the *join attributes*.

Recall, however, that our formulation of the query made no reference to the product by which the customer and part are related. To produce the desired result relation, we must therefore remove the *PRODUCT* attribute from our intermediate result. Notice, however, that the first and eighth tuples in the intermediate result are distinguished only by the value of their respective *product* attributes. Upon removal of this attribute, these two tuples would no longer differ, introducing a redundancy in the result relation which is prohibited by the fact that relations are sets. (As we shall see in the following subsection, our injunction against relations with redundant tuples does not reflect a superstitious adherence to our formal definition of relations, but is in fact motivated by important practical considerations.)

The final step of our example thus involves not only removal of the *PRODUCT* attribute, but also elimination of the redundant tuples that would otherwise result from the removal of formerly distinguishing attribute values:

| CUSTOMER | PART |
|---|---|
| American | oxygen-mask |
| Western | wheel |
| Western | engine-mount |
| Western | oxygen-mask |
| Western | radio |
| American | wheel |
| American | engine-mount |
| American | radio |

This combined operation is an example of another relational algebraic operation, called *projection*. We refer to the *CUSTOMER* and *PART* attributes, which are carried through to the result relation in our example, as *projected attributes*, while the *PRODUCT* attribute is described as *projected out* of the argument relation.

The very simple example that we have just considered has required only the informal introduction of the join and project operators. In the following subsection, we will define a larger set of relational algebraic

primitives, providing a more rigorous definition for each one. The importance of the relational algebra to our task (and indeed, to a great many knowledge-based tasks) derives from the fact that this more complete set of relational algebraic primitives has all the "descriptive power" of the logic-based query language introduced above.

Perhaps the best known analogue of this observation in the relational database literature is the so-called *completeness* result due to Codd [1972]. In essence, Codd gave a constructive proof that any query formulated using the *calculus of α-expressions* (often called the *relational calculus*)—a descriptive language quite similar to our own first-order predicate calculus-based query language, but where, among other distinctions, quantification is over tuples and not elements of the primitive domains—could be computed by application of a suitable sequence of relational algebraic operations. Ignoring for the moment the differences between strict first-order logic and relational calculus, Codd's result thus provides a systematic (though generally inefficient) way of computing the result of an arbitrary first-order query, as defined above, using only the relational algebraic operations that will be defined in the next subsection.

One way of viewing the roles of logic and relational algebra in this sort of retrieval task that we have found particularly useful in our work is based on the notion of a *formal theory*. Within this theoretical framework, we view the query as part of a *first-order theory*, and the relations in the (extensional) database as a particular *finite interpretation* of that theory. In particular, each primitive predicate in the query is associated with one relation in the extensional database, which is treated as its *interpretation*. Within this framework, we can view the problem of finding the result of the query as that of finding all possible values of the (free) result variables such that the query well-formed formula is *logically satisfied* under that interpretation. For this reason, we sometimes call refer to the task of computing the result of a logical query as one of *satisfaction* over a finite (albeit generally large) domain.

### *7.4 The relational algebraic primitives*

The relational algebra we have used in our research is based on a small set of algebraic operators enumerated by Codd [1972] which take one or more relations (along with certain "control" information) as arguments, returning a single new relation as their value. This set of primitives includes the ordinary set operations—which, with one restriction, are defined for relations in much the same way as for other sets—along with several *structured* operators, which make reference to the internal attribute structure of the constituent tuples. The two most important structured operators, *project* and *join*, have already been informally described in the previous subsection. Several other structured operations will also be introduced in this subsection which may in fact be derived from project, join and the unstructured set operations, but which serve certain particularly important functions in many practical applications of relational algebraic systems.

Specifically, we will be concerned in this paper with the following relational algebraic primitives:

1. Union
2. Intersection
3. Set difference

4. Projection

5. Join

7. Selection

8. Restriction

The three binary set operators *union*, *intersection* and *set difference* are defined in a relational algebraic system in the same way as for sets in general, with one exception: the relational version of each is defined only when the two relations that serve as its operands are *union-compatible*. Two relations are said to be union-compatible if and only if they are of the same degree $n$ and the underlying domains of the $i$-th simple attributes of the two relations are the same for all $i$, $(1 \leq i \leq n)$.

We thus define the union of two union-compatible relations $R_1$ and $R_2$, denoted $(R_1 \cup R_2)$, as a relation consisting of exactly those tuples that are an element of $R_1$, of $R_2$, or both. The intersection $(R_1 \cap R_2)$ is defined as that relation containing all tuples found in *both* $R_1$ and $R_2$. Finally, the set difference $(R_1 - R_2)$ is defined to consist of exactly those tuples of $R_1$ that are *not* present in $R_2$.

In preparation for our formal definition of the projection operator, we will need to introduce some additional notation. First, we adopt the convention that a list of primitive domain elements enclosed by angle brackets ("⟨" and "⟩") will designate a new tuple containing the specified elements as the values of its simple attributes, in the order listed. Futhermore, if $r$ is a tuple of some $n$-ary relation $R$, we will define $r[j]$ to be the value of the $j$-th attribute of $r$, $(1 \leq j \leq n)$. It will be convenient to extend this notation to allow expressions such as $r[A]$, where $A$ is a *compound* attribute of $R$ consisting of the $m$ (not necessarily distinct) simple attributes numbered $j_1, j_2, \ldots, j_m$, defined such that $(r[A])$ represents the new tuple $\langle r[j_1], r[j_2], \ldots, r[j_m] \rangle$.

We may now define the projection of a relation $R$ over the compound attribute $A$ as the set

$$\{\langle r[A] \rangle : r \in R\} \quad .$$

Note that we have defined the projection operator in such a way that simple attributes within the compound attribute $A$ may be *replicated* in the course of projection. Depending on certain details in the definition of the join operation, this convention may have important theoretical consequences affecting the expressive power of the resulting algebra.

The projection operator may be thought of as a sort of "vertical subsetting" operation, in which

1. the "non-projected" attributes of each tuple in the argument relation are eliminated,
2. the remaining attributes may be permuted and/or replicated, and
3. any duplicate tuples that result from the elimination of values that formerly distinguished different tuples are then removed.

In most implementations on a von Neumann machine—that is, a "conventional" computer system having a single central processing unit acting on a single bank of random access memory—the attribute elimination and permutation/replication functions can both be implemented using a simple and computationally inexpensive procedure whose complexity is linear in the cardinality of the argument relation. The elimination of

redundant tuples, on the other hand, may be surprisingly time-consuming, particularly when the argument relation is large. In fact, one common convention in some von Neumann implementations is to relax the requirement that relations be true sets, allowing the introduction of duplication during some or all projections. This approach introduces the following problems, however:

1. the maintenance of duplicate tuples may lead to combinatorially explosive growth in the cardinality of the intermediate results of a complex query, and
2. functions sensitive to the repetition of identical tuples—the calculation of numerical counts and statistical measures, for example—will not yield accurate results if redundant tuples are not first eliminated.

One of the capabilities of the NON-VON machine is the performance of true projection without the high cost of redundant tuple elimination.

Definition of the join operation requires the definition of one additional construct: the *concatenation* of two tuples. If $r_1$ is a tuple of a relation $R_1$, having degree $n_1$, and $r_2$ is a tuple of relation $R_2$, having degree $n_2$, the *concatenation* $(r_1|r_2)$ of $r_1$ and $r_2$ is defined to be the new $(n_1 + n_2)$-tuple

$$(r_1[1], r_1[2], \ldots, r_1[n_1], r_2[1], r_2[2], \ldots, r_2[n_2]) \quad .$$

Several variations of the join operator are commonly discussed in the literature; we will begin by defining a particularly important variant known as the *equi-join*. The equi-join of two relations $R_1$ and $R_2$ over the compound attributes $A_1$ and $A_2$, respectively (each assumed to be composed of the same number of simple attributes, with corresponding simple attributes having underlying domains that are comparable under the equality predicate) is defined as

$$\{(r_1|r_2) : r_1 \epsilon R_1 \wedge r_2 \epsilon R_2 \wedge r_1[A_1] = r_2[A_2]\} \quad .$$

$A_1$ and $A_2$ are referred to as the (compound) *join attributes*, which will have special significance in the algorithms introduced in this paper. In the case where $A_1$ and $A_2$ are the degenerate compound attributes containing no simple attributes, equi-join reduces to the *extended cartesian product* of the tuples of $R_1$ and $R_2$—that is, to the set of all possible concatenations of one tuple from $R_1$ with one from $R_2$. The more general join operation may be intuitively thought of as a process of filtering the extended cartesian product of $R_1$ and $R_2$ by removing from the result all conjoined tuples whose respective join attributes have different values. (The computational method suggested by this interpretation, of course, would in general be impractically inefficient.)

The join operation is in general quite expensive on a conventional von Neumann machine, since the tuples of $R_1$ and $R_2$ must be paired for equality with respect to the join attributes before the extended cartesian product of each group of "matching" tuples can be computed. In the absence of physical clustering with respect to the join attributes (whose identity may vary in different joins over the same pair of relations), or the use of various techniques requiring a large amount of redundant storage, joining is typically accomplished

most efficiently on a von Neumann machine by pre-sorting the two argument relations with respect to the join attributes. The *order* of the tuples following the sort is actually gratuitous information from the viewpoint of the join operation. From a strictly formal perspective, the requirements of a join—that the tuples be paired in such a way that the values of the join attribute *match*—are significantly weaker than those of a sort, which require that the resulting set be *sequenced* according to those values. The distinction is moot in the case of a von Neumann machine, where no asymptotically superior general solution to this pairing problem than sorting is presently known. One of the design goals of the NON-VON machine, however, is to make use of the weaker constraints involved in the definition of this kind of operation to obviate the need for either pre-sorting or the extravagant use of redundant storage.

One common variant of the equi-join operator is the *natural join*, introduced in the example of the previous section, in which one of the two join attributes, which are redundantly represented in the result relation in the case of equi-join, is eliminated (as if by projection). Our architecture supports both the natural and equi-join in a highly efficient manner. A more general form of join often discussed in the literature is the $\vartheta$-join, whose definition is similar to that of the equi-join, but with the equality predicate replaced by a more general binary predicate $\vartheta$. (In Codd's definition, $\vartheta$ is defined to be one of the arithmetic operations $=$, $\neq$, $<$, $\leq$, $>$ or $\geq$.) Considerations for the efficient evaluation of the general $\vartheta$-join operator differ in several respects from those involved in evaluating the equi-join. We will not discuss this more general case in the present paper.

Each of the other relational algebraic operators to be described in this subsection can in fact be derived from the structured operators project and join and the three unstructured set operators, and are defined here for one or both of the following reasons:

1. The operator embodies a special case of one or more of the previously defined primitives which might admit the possibility of either a less complex, or a more efficient, hardware implementation
2. The operator represents an important and frequently encountered use of some composition of the primitives defined earlier

One derived operation that occurs frequently in both practical and theoretical discussions, and which plays a particularly important role in our approach, is called *selection*. Most algorithms and architectures for "associative retrieval" implement what is essentially a process of relational selection. In the NON-VON machine, selection requires only a small, fixed amount of time, *independent* of the size of the database; unlike most associative processor designs, however, our architecture explicitly addresses the problems of efficiently implementing other relational operators as well. The select operator returns a subset of its single argument relation consisting of all tuples that satisfy a list of attribute/value pairs. The select operator may thus be regarded as a natural join of the argument relation with a *singleton* relation (a relation consisting of exactly one explicitly specified tuple) over all attributes of the singleton. More precisely, the result of a selection from relation $R$ with compound attribute $A$ and value tuple $V$ is

$$\{r : r \in R \wedge r[A] = V\} \quad ,$$

where the corresponding $A$ and $V$ domains are again assumed to be compatible with respect to equality.

Another important derived operation is known as *restriction*. While restriction, like the join operator, is sometimes defined in terms of a general $\theta$, we will again be concerned only with the case where $\theta$ is the binary equality predicate. The restriction of a relation $R$ over the compound attributes $A_1$ and $A_2$ (both composed of simple attributes of $R$) is defined as

$$\{r : r \in R \wedge r[A_1] = r[A_2]\} \quad .$$

In its most common form, where the compound attributes $A_1$ and $A_2$ are each composed of exactly one simple attribute, the restriction operator returns all tuples of its argument relation in which the values of the two specified attributes are equal. Although restriction can be defined solely in terms of the join and project operators, an implementation based in a straightforward way on this derivation would be considerably more complex and inefficient than one specifically tailored to support the restrict operator. Restriction is an important enough operation in practice that we have treated the capacity for direct (and efficient) evaluation of restrictive expressions as a significant design objective.

Finally, we must acknowledge a derived operation that has considerable theoretical and practical importance in many applications, but to which we have devoted little special attention in our evaluation of alternative architectures. This operation, called *division*, is used to achieve the effects of universal quantification within the queries of a language based on the relational calculus (Codd [1972]) and may well be worthy of special attention in course of designing a generally-applicable relational database machine. Since it was not necessary that this kind of operation be implemented efficiently in its full generality for purposes of our AI application, however, the relational division operator will not be given the same sort of special consideration in this paper as the other two derived operators described above.

### *7.5 Defined predicates*

The example query formulated in Subsection 7.3 made use of two logical predicates, each associated with an explicitly defined relation that might be stored in an "extensional database" of the sort used in our demonstration system. We refer to a predicate of this kind, whose meaning derives from its association with a relation whose constituent tuples are explicitly enumerated, as a *primitive predicate*. As we shall see in Subsection 7.7, though, our knowledge-based retrieval task requires the use of another kind of predicate, which we will call a *defined predicate*.

A defined predicate corresponds to no fixed, explicitly defined relation, as does a primitive predicate, but instead derives its meaning from a *proper* (non-logical) *axiom* expressing its equivalence to a well-formed formula involving other (defined or primitive) predicates. (Our notion of a defined predicate is closely related to that of a *view*, as defined in the relational database literature, and to other constructs that have been introduced periodically by researchers in other areas of computer science; it is our use of defined predicates that should be of interest here.) Following a convention employed in certain recent work on logic and databases, we will sometimes refer to the set of defined predicates as the *intensional database* of our system (by contrast

with the extensional database, which is composed of relations specified explicitly by enumerating all their tuples, each of which corresponds to a primitive predicate in the logical query language). In the interest of perspicacity, we adopt the convention of surrounding the name of each primitive predicate by asterisks, as in the examples we have already seen, while the names of defined predicates will not be so enclosed.

By way of illustration, let us consider a very simple example of a defined predicate, called *CUSTOMER--PART*, whose body is precisely the well-formed formula from the sample query introduced in Subsection 7.3:

$$
\begin{aligned}
&\mathit{CUSTOMER\text{--}PART}(x, z) \equiv \\
&\quad \exists y. \\
&\qquad (\ast \mathit{CUSTOMER\text{--}PRODUCT} \ast (x, y) \wedge \\
&\qquad\ \ast \mathit{PRODUCT\text{--}PART} \ast (y, z))
\end{aligned}
$$

In this example, the defined predicate *CUSTOMER--PART* is defined in terms of the two primitive predicates $\ast CUSTOMER\text{--}PRODUCT\ast$ and $\ast PRODUCT\text{--}PART\ast$, allowing its *reduction* to predicates whose interpretations are explicitly available in the extensional database. The newly defined predicate could thus be used as a sort of "shorthand" for the sample query, which might now be expressed as

$$
\begin{aligned}
&(x, z): \\
&\quad \mathit{CUSTOMER\text{-}PART}\,(x, z) \quad .
\end{aligned}
$$

Although intensionally defined predicates (more precisely, their analogues within the calculus of $\alpha$-expressions) were not included as part of Codd's original formalism for the relational model of data, more recent work by several researchers suggests one possible manner in which defined predicates might be employed within a first-order query. At the risk of oversimplification, this approach (exemplified by Reiter [1977] and by Chang and Lee [1973]) involves a two-step procedure for the evaluation of queries. During the first step, the query (which may involve both primitive and defined predicates), along with the predicate definition axioms found in the intensional database, is manipulated using automatic theorem-proving techniques to remove all occurrences of the defined predicates, yielding a transformed query containing only primitive predicates. The resulting transformed query is then evaluated using ordinary database retrieval techniques. As we shall see in the next subsection, however, the lack of extensional feedback in the course of intensional manipulation introduces certain problems which preclude the possibility of a straightforward application of this approach to our knowledge-based retrieval task.

*7.6 Recursive predicate definition*

From the perspective of our own application, one of the most serious limitations of the approach to retrieval that we informally outlined at the end of the last subsection relates to the need to support *recursively*-defined predicates. Before considering the role of recursive predicate definition within our system for knowledge-based retrieval, let us consider a very simple example of a recursively-defined predicate. Imagine for the moment that the extensional database contained a binary relation, called *•CHILD--PARENT•*, asserting that *Suzanne* has *Charles-Jr* and *Marilyn* as parents, while *Charles-Jr* is in turn the son of *Charles-Sr* and *Estelle*, etc., as illustrated below:

| *CHILD* | *PARENT* |
|---|---|
| *Suzanne* | *Charles-Jr* |
| *Suzanne* | *Marilyn* |
| *Charles-Jr* | *Charles-Sr* |
| *Charles-Jr* | *Estelle* |
| *Marilyn* | *Benjamin* |
| *Marilyn* | *Esther* |

Suppose now that we wished to construct a defined predicate *DESCENDANT--ANCESTOR* having the value *true* whenever its first argument is either the child, grandchild, great-grandchild, etc. of its second argument.

If it were not for the problem of recursion within a predicate definition, the *DESCENDANT--ANCESTOR* predicate could be defined as

$$
\begin{aligned}
&\textit{DESCENDANT--ANCESTOR}(x,\ z) \equiv \\
&\quad \bullet\textit{CHILD--PARENT}\bullet\ (x,\ z)\ \vee \\
&\quad \exists y. \\
&\qquad (\bullet\textit{CHILD--PARENT}\bullet\ (x,\ y)\ \wedge \\
&\qquad\ \bullet\textit{DESCENDANT--ANCESTOR}\bullet\ (y,\ z))
\end{aligned}
$$

Note, however, that the first step (query transformation) of the hypothetical two-step process outlined in the previous subsection could no longer simply replace each occurrence of the defined predicate *DESCENDANT--ANCESTOR* with its body, since that body itself contains another invocation of *DESCENDANT--ANCESTOR*; the theorem-proving technique would thus either fail to terminate, or fail to find all possible results of the query.

The approach that we adopted in the LSEC algorithm to the problem of recursive predicate definition uses intermediate results of the query evaluation in order to terminate computation after all potentially relevant results have been obtained, avoiding computational loops based on the endless expansion of cycles of mutually-defined predicates. While we will not examine the details of the LSEC algorithm at this point, the basic mechanism by which LSEC handles recursion is quite simple. In essence, our approach involves the

treatment of the conjunctive operator in a query well-formed formula as a "computational" (as opposed to a strictly "logical") *AND*. As in the case of the LISP *AND*, the operands of such a conjunction are evaluated in left-to-right order until the first "failure"—specifically, until the first case in which application of the LSEC procedure to some operand yields a *false-extension* (defined rigorously in Section 8), which may regarded as a generalization of the boolean constant *false*. In the above example, •DESCENDANT--ANCESTOR• $(y, z)$ will not be recursively evaluated after the primitive predicate •CHILD--PARENT• $(x, y)$ is found to have no possible instantiations consistent with the current binding of $z$—that is, after reaching the level of the "grandparents".

In passing, it should be acknowledged that the importance of recursive predicate definition within our own application may well reflect characteristics specific to our thesis task. We are thus unable to make any claims regarding the suitability of our approach for use in conventional relational database applications. Indeed, Gallaire, et al. [1978], after reviewing the problems of recursive predicate definition, suggest that "it is not clear that one should permit recursive axioms (in the intensional database) for realistic problems". The suspicion that the additional theoretical power provided by recursive predicate definition may have minimal significance in the context of contemporary practical database requirements has been echoed by other authors. While our own areas of expertise do not permit a qualified opinion in this regard, it should be emphasized that in our application, in which defined predicates are not directly incorporated in a user-level description language, but instead are used internally to express and implement the semantics of a higher-level description language, *recursive predicate definition is essential.*

In the following subsection, we will examine the way in which recursively defined predicates in the intensional database are used to define the semantics of our knowledge-based description language.

### *7.7 Axioms defining the match semantics*

Having now introduced the essential logical and relational tools, let us now consider the manner in which a user's knowledge-based pattern description is matched against the collection of candidate target descriptions according to the axioms defined in the match specification. In each such query task, regardless of the particular pattern description, the "top-level" logical query is has the form

$$(x) : \mathit{DTION\text{-}IMP\text{-}DTION}(x, \mathit{pat\text{-}dtion}) \quad .$$

The intended result of this query is a unary relation, each of whose tuples has as the value of its single attribute a particular target description (more precisely, the primitive domain element that anchors the internal form of that target description) which matches the given pattern description according to the rules defined in the match specification. Specifically, the match specification comprises a definition of the defined predicate *DTION-IMP-DTION* in terms of other predicates, some of which are themselves defined in other axioms, and so on.

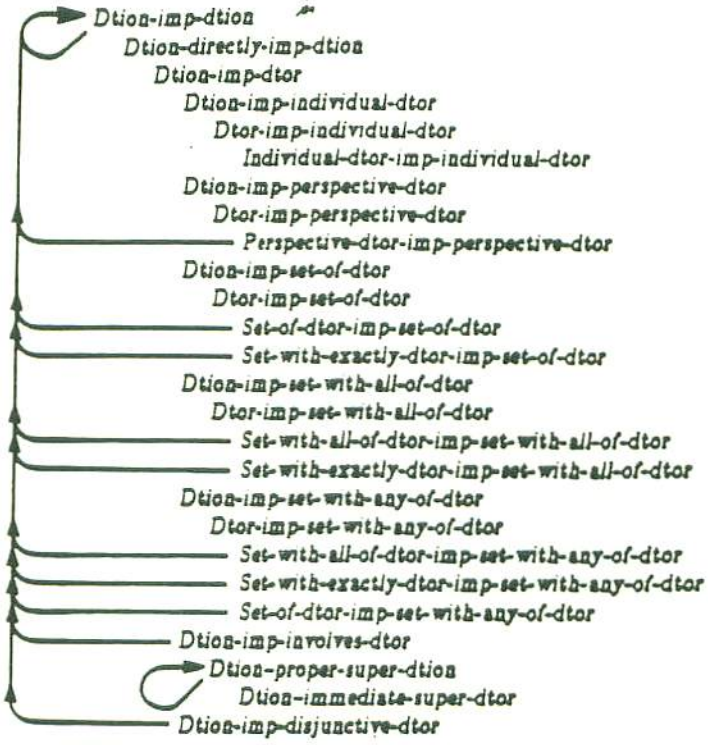



**Figure 7.2** Axioms Defining the Match Semantics

The names of each of the 26 axioms defining the match semantics of our knowledge-based description language are listed in Figure 7.2. We have indented this list to indicate which predicates are defined in terms of which other predicates, with the name of a defined predicate shown below, and indented by one unit with respect to, the name of the defined predicate in whose body it is first used. (The "top-level" defined predicate *DTION-IMP-DTION*, for example, is defined in terms of the defined predicate *DTION-DIRECTLY-IMP-DTION*.) Exceptional definitional directions are indicated explicitly using arrows. (*PERSPECTIVE-DTOR-IMP-PERSPECTIVE-DTOR*, for example, is defined in terms of the top-level defined predicate *DTION-IMP-DTION*.) Each of the upward-pointing arrows thus identifies a recursive predicate definition loop, underscoring the central role such definitions occupy in our thesis system.

Although it will not be necessary at this point to consider the details of each defined predicate in the match specification, it may be instructive to consider one typical such predicate in an attempt to convey some feeling for the kind of information embodied in these matching rules. To this end, we consider the defined predicate *PERSPECTIVE-DTOR-IMP-PERSPECTIVE-DTOR*, which implements what we may call *syntactic perspective matching*:

*Perspective-dtor-imp-perspective-dtor (tar-per, pat-per)* ≡

∃ *proto* .

(*Per-proto (pat-per, proto)* ∧

*Per-proto (tar-per, proto))* ∧

∀ *slot* .

((∃ *pat-fill* .

*Obj-slot-fill (pat-per, slot, pat-fill))*

⊃ ∃ *pat-fill, tar-fill* .

(*Obj-slot-fill (pat-per, slot, pat-fill)* ∧

*Obj-slot-fill (tar-per, slot, tar-fill)* ∧

*Dtion-imp-dtion (tar-fill, pat-fill)))*

In the course of syntactic perspective matching, all target perspectives that match a given pattern perspective and have the same prototype as that pattern perspective are identified. By contrast with *semantic perspective matching*, this procedure does not identify those target perspectives that have *different* prototypes than that of the pattern, but that would in fact satisfy the matching criteria on the basis of domain-specific specialization relationships derived from the knowledge base. The meaning of this defined predicate is reasonably straightforward: in order for a "target-perspective-dtor" to match a "pattern-perspective-dtor" on the basis of this defined predicate, the prototypes of each must be the same, and for each slot which is filled in the pattern, the corresponding slot must be filled in a compatible way (as specified by the recursive call to *DTION-IMP-DTION*) within the target.

The full set of 26 axioms that together constitute the match specification for our knowledge-based description language are presented as an appendix to our doctoral dissertation [Shaw, 1980a].

Having now examined an example of the use of well-formed formulæ within our demonstration system, it seems appropriate to explicitly list the respects in which our logic-based query and predicate definition language is in fact restricted by comparison with the full language of first-order predicate calculus. First, function symbols (which in fact add no formal expressive power to the predicate calculus) have been eliminated from consideration. Second, our present application has not required the use of explicit *negation*; the *NOT* operator has thus been omitted from our language as well. Note that by contrast with the case of functions, the addition of negation would significantly expand the expressive capabilities of our language; we consider such an extension to represent a potentially important avenue along which our own work might be extended. Because negation introduces a number of surprisingly difficult problems when used in a computationally effective description language, however, we have chosen to omit this construct from consideration in our work to date.

Finally, we have intentionally permitted only a restricted form of universal quantification. Rather than permit universal quantification of the general form

$$\forall x.P(x) \quad ,$$

we restrict $P(x)$ to be of the form

$$Q(x) \supset B(x) \quad ,$$

where $Q(x)$, called the *qualification clause*, is further restricted to contain no disjunction or universal quantification, while the *body* $B(x)$ is an unrestricted well-formed formula. While detailed consideration of universal quantification will be deferred to Section 7, we note for now that the qualification clause serves the important practical function of restricting the plausible range of universally quantified variables, thus limiting the set of "$x$ values" which must be considered to those values for which $Q(x)$ is satisfied.

## 8. The LSEC Algorithm for Logical Satisfaction

In this section, we will describe the function of the LSEC algorithm, through which the connection between logical descriptive mechanisms and actual relational algebraic operations is established in our demonstration system. The LSEC algorithm has been fully implemented in our demonstration system and tested on carefully chosen examples designed to exercise each portion of the algorithm.

We begin in Subsection 8.1 with an example of the use of LSEC in evaluating the results of a simple conventional database query, avoiding many of the spurious complications arising in our knowledge-based retrieval application that are peripheral to the essential operation of the LSEC algorithm. In the remainder of the section, we describe the behavior of the algorithm upon encountering each of the six types of logical formula used in constructing the match specification, ending with the procedure by which the result relation is constructed.

### *8.1 A simple example*

The example we have chosen to illustrate the process of *extensional constraint* has the virtue of illustrating the essential behavior of the LSEC algorithm, but may well seem a bit contrived to the reader. To be sure that our somewhat unnatural example does not obscure the features we will be attempting to illustrate, we thus begin with a brief discussion of its "real-world" setting.

In our example, a hypothetical capitalist wishes to affect the behavior of major U.S. corporations by exerting an indirect influence on key individuals within those corporations—specifically, on the officers and directors of those firms. This indirect influence is in turn to be mediated by the attorneys and accountants of these key individuals, on whom these individuals are presumed to rely for information and advice. The capitalist might succeed, for example, in influencing the behavior of the corporation by bribing its president's attorney, or the accountant of one of its directors. To this end, our capitalist wishes to review a list of professionals (attorneys and accountants), paired with corporations on whom these professionals could ultimately exert an influence through some third party whose identity is of no concern to the capitalist.

Four two-attribute relations will be used in our example:

| ATTY | CLNT |
|---|---|
| Atty1 | Jones |
| Atty1 | Stone |
| Atty2 | Jones |

| ACCT | CLNT |
|---|---|
| Acct1 | Smith |
| Acct1 | Jones |

| OFCR | CORP |
|---|---|
| Smith | XCorp |

| DRTR | CORP |
|---|---|
| Jones | YCorp |
| Jones | XCorp |
| Smith | XCorp |

The first associates attorneys with their clients; the second, accountants with their clients; the third, officers with the corporations by which they are employed; the fourth, directors with the corporations on whose boards they serve. The binary relation which our hypothetical capitalist wishes to review may be described using the logical query

(*professional*, *corporation*): ∃ *client*.
((•*ATTY*--*CLNT*• (*professional*, *client*) ∨
•*ACCT*--*CLNT*• (*professional*, *client*)) ∧
(•*OFCR*--*CORP*• (*client*, *corporation*) ∨
•*DRTR*--*CORP*• (*client*, *corporation*)))

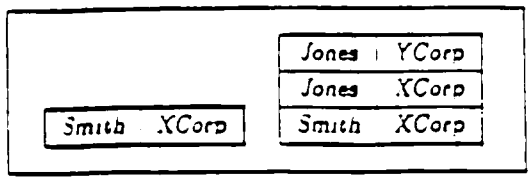
Without concentrating on the details of the algorithm, we will now sketch the manner in which LSEC finds the results of this query.

The most important data structure maintained and manipulated by the LSEC algorithm is called the *Accumulated Disjoint Extension* (sometimes referred to as the *ADE*, or somewhat less precisely, as simply the *extension*). Loosely speaking, the ADE may be thought of as a set of relations, each of which describes one way in which that part of the logical formula thus far encountered may be satisfied. (In one sense, the ADE may thus be thought of as a sort of dynamic, extensional analogue to the intensional notion of a *disjunctive normal form*.) The initial ADE in any LSEC session is always the distinguished ADE *true-extension*, which is a set consisting of the single degenerate relation containing no attributes and no tuples (the *true-relation*). In the course of processing the top-level query (and the defined predicate bodies introduced in the course of expanding that query), this ADE is constrained by the various logical subformulæ; after the whole query has been processed, the result relation may be extracted (in a manner detailed in Subsection 3.9) from the final ADE.

The tree-traversal algorithm embodied in the LSEC algorithm causes the initial ADE (*false-extension*) to be constrained first by the disjunction

(•*OFCR*--*CORP*• (*client*, *corporation*) ∨
•*DRTR*--*CORP*• (*client*, *corporation*))

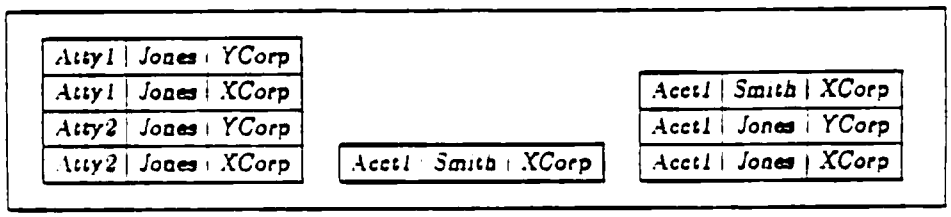
The resulting ADE is a set of two relations, the first corresponding to influences exerted through officers, and the second through directors:



| Smith | XCorp |
|---|---|

| Jones | YCorp |
|---|---|
| Jones | XCorp |
| Smith | XCorp |

Constraining this ADE further to reflect the content of the first conjoined subformula of the body of our query (to reflect the fact that access to these key individuals may be through either their attorneys or accountants), LSEC "multiplies through" the original •ATTY--CLNT• and •ACCT--CLNT• relations by *joining* each with each of the two *terms* (constituent relations) in the ADE over the common existentially quantified variable *client*. Because none of the attorneys in our example has as client the lone individual (*Smith*) who serves as a corporate officer, one of the four resulting terms will be a *false-relation*—a relation having any attribute structure, but no tuples—which will be eliminated from the ADE, yielding the three-relation ADE

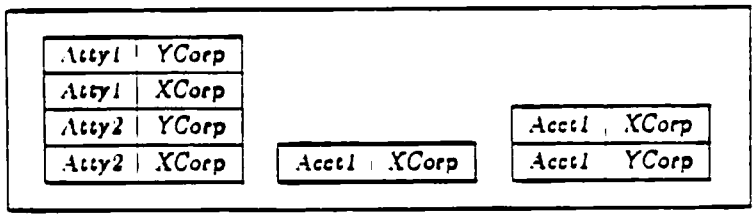


| Atty1 | Jones | YCorp |
|---|---|---|
| Atty1 | Jones | XCorp |
| Atty2 | Jones | YCorp |
| Atty2 | Jones | XCorp |

| Acct1 | Smith | XCorp |
|---|---|---|

| Acct1 | Smith | XCorp |
|---|---|---|
| Acct1 | Jones | YCorp |
| Acct1 | Jones | XCorp |

Notice that by existentially quantifying *client*, instead of leaving it free in the query well-formed formula, we have indicated that the identity of the client is of no concern to the end-user; it is used only to establish that some link between professional and corporation exists. Upon exiting from the scope of the existentially quantified formula, we thus *project* out the *CLNT* attribute from all terms in the current ADE. The result (after eliminating one redundant tuple in the third relation) is



| Atty1 | YCorp |
|---|---|
| Atty1 | XCorp |
| Atty2 | YCorp |
| Atty2 | XCorp |

| Acct1 | XCorp |
|---|---|

| Acct1 | XCorp |
|---|---|
| Acct1 | YCorp |

Finally, the *union* of all relations in the ADE is taken, thus combining all its terms into a single result relation:

| | |
|---|---|
| *Atty1* | *YCorp* |
| *Atty1* | *XCorp* |
| *Atty2* | *YCorp* |
| *Atty2* | *XCorp* |
| *Acct1* | *XCorp* |
| *Acct1* | *YCorp* |

Note that during the performance of this final union operation, one more tuple becomes redundant, and must be eliminated.

By way of summary, then, the LSEC algorithm starts with an unconstrained extension (the ADE *true-extension*), which is then constrained by the query formula. In our demonstration system, the initial query is always the defined predicate

$$(z) : DTION\text{-}IMP\text{-}DTION(z, pat\text{-}dtion) \quad ,$$

whose body is expanded in the course of the algorithm, with different types of logical formula "surfacing" at different depths within this expansion. In the remainder of this section, we will review the behavior of the LSEC algorithm upon encountering each of these logical formula types.

*8.2 Existential quantification*

The processing of existentially quantified formulas within the top-level query, or within some defined predicate body encountered in the course of evaluating this query, is quite simple, but provides a good introduction to the use of one of the essential data structures manipulated by the algorithm: the *logical variable stack*. (Although it will be referred to informally as simply "the stack", the logical variable stack should not be confused with the stack maintained by the underlying LISP system, which maintains the bindings of the $\lambda$-variables involved in execution of our demonstration system.)

A new stack frame is created each time an existentially or universally quantified formula, or a defined predicate, is encountered, to hold certain information about each quantified variable or formal parameter (in the case of the defined predicate) introduced within the current formula. There are two components to this information: the first specifies the variable's *type*, which may be either *constant-valued* or *attribute-id-valued*, while the second is the value itself. Quantified variables, though, are always of the latter type: upon encountering the existentially quantified formula, a unique name called an *attribute identifier* (*attribute-id*) is generated for each existentially quantified variable to designate a particular attribute that will ultimately appear in one or more relations in the ADE. Although the same attribute may be referenced using different names within the bodies of different defined predicates, each such name will be "bound" to the same attribute-id at different locations within the logical variable stack. Conversely, the same variable name may appear in various frames within the current stack in the case where the same variable name is re-used (possibly from

the same place within two nested invocations of the same defined predicate) within a scope in which it has already been defined. In each of its appearances, such a name may be bound to a different attribute-id; in such cases, the most recently added stack frame (the frame on the "top" of the stack) is treated as "current".

The processing of existentially quantified formulæ is now quite simple to describe. Upon entering such a formula, LSEC first creates a new stack frame containing information about each of the existentially quantified variables introduced in the current formula. Next, the current ADE is (recursively) constrained by its body, which is implicitly treated as a conjunction of one or more subformulæ (see Subsection 3.1). Upon exit from the existentially quantified formula, each of these existentially quantified variables is *projected out* of each term in the resulting ADE for reasons of efficiency. Readers familiar with Codd's constructive completeness proof [Codd, 1972] may notice that our maintenance of a logical variable stack serves what may be regarded as a dynamic analogue of the process of conversion to prenex normal form, but adapted to the case where the required renaming operations cannot be determined statically on a purely lexical basis.

*3.3 Universal quantification*

As in the case of the existential formula, processing of universally quantified formulæ begins with the creation of new attribute-ids to identify each of the newly-introduced universally quantified variables, and a new stack frame is created to record the type ("attribute-id-valued") and value (the newly generated unique attribute-id name) of each such variable. The substantive portion of the processing of universally quantified formulæ within LSEC, however, is considerably more complex than the corresponding part of the procedure for processing existentially quantified formulæ. As noted in Subsection 7.7, all such formulæ are of the form

$$\forall x.Q(x) \supset B(x) \quad ,$$

where the *qualification clause* $Q(x)$ is restricted to contain no disjunction or universal quantification. This qualification clause is used to restrict the range of the universally quantified variable $x$ in each of the possible *contexts* defined by alternative joint instantiations of the various quantified variables that are "visible" within the current scope. We now consider in some detail the manner in which LSEC constrains the current ADE by a universally quantified formula.

Recall first that the ADE is in general a set of *several* relations, called the *terms* of that ADE. LSEC constrains each term by the universal formula, then takes the union of the results to form a new ADE. In order to constrain a given extension term by a universal formula, LSEC must first identify a crucial set of variables called the *context variables* of that formula. The context variables are precisely those attribute-id-valued variables that appear within the qualification clause, but whose innermost scope is not local to the qualification clause. (Our somewhat unwieldy definition is required to insure that a variable appearing within the qualification clause at a point where its name is bound both locally—that is, within the qualification clause—and globally is *not* treated as a context variable.)

LSEC next computes what is called the *qualified extension term* by (recursively) constraining the extension term by the qualification clause ($Q(x)$) of the universal formula. The qualified extension term is then *projected*

over the attributes corresponding to each of the context variables (as determined by the current stack bindings for those context variables), yielding a set of *context tuples*. Each such tuple defines one *context* for evaluation of the body of the universally quantified formula. In intuitive terms, a context may be thought of as containing all relevant information about one possible way in which the qualification clause might be satisfied—that is, one "*such that* ..." condition in a universal formula which might be described by the English assertion "for all $x$ *such that* $Q(x)$ is true, $B(x)$ must also be true". Note that it is not sufficient in general to find all $x$ satisfying this "such that ..." clause, and to simply substitute all such $x$ into the body of the universal formula to obtain a new set of constraining formulas. In order to avoid excluding joint instantiations which which might well satisfy the top-level query, information regarding constraints on variables *global* to the universal formula must be propagated along with each such $x$ value.

Each of these context tuples will ultimately contribute to the result for the current extension term, the partial results due to each being appended together at the end to form a new extension. Let us now consider the manner in which a given context tuple is used to constrain the qualified extension term by the body of the universally quantified variable. First, the qualified extension term is *selected* on the attributes and values specified in the context variable list and the current context tuple, respectively, to obtain the *context-bound extension slice* for that context tuple. By projecting the context-bound extension slice over the universally quantified variable, it is now possible to identify the effective range of the universally quantified variable within the current context, resulting in a list of *context-bound universally quantified values*.

To obtain the partial result, the universally quantified variable is first projected out of the qualified extension term slice. The result (corresponding to the current context tuple) is then constrained by various instantiated versions of the body in succession—first, by a version of the body with the first context-bound universally quantified value substituted for the universally quantified variable, next with the second such value substituted for that variable, and so on for each of the possible values within the effective (context-bound) range of the universally quantified variable. The result corrresponding to this context tuple is now combined with those derived from each of the others to obtain a version of the original extension term constrained by the universally quantified formula. As noted above, the final result is obtained by taking the union of the results due to each extension slice in the original ADE.

As in the case of existential quantification, the universally quantified variables are projected out of the result upon leaving the scope of the universally quantified formula for reasons of efficiency.

### *8.4 Conjunction*

It is in the case of a conjunctive formula that the process of progressive constraint which underlies the operation of the LSEC algorithm is most evident. Upon encountering a set of conjoined subformulas

$$P(x) \wedge Q(x) \wedge R(x) \wedge \ldots \quad ,$$

the old ADE is first constrained by $P(x)$; the result is then further constrained by $Q(x)$, then by $R(x)$, and so on until either the extension has been constrained by the last conjoined subformula or a *false-extension* is returned by one such constraining step, in which case computation terminates with the value *false-extension*.

*3.5 Disjunction*

In the course of constraining the ADE by a disjunctive formula, the "width" of the ADE—more precisely, the number of relations which it comprises—is, in the general case, increased to reflect a larger number of alternative ways in which the query might be satisfied. To constrain the current ADE by a disjunction

$$P(x) \vee Q(x) \quad ,$$

for example, two copies of the ADE are made; one is constrained by $P(x)$, the other by $Q(x)$, and the results combined to form the new ADE.

*3.6 Defined predicates*

The processing of a defined predicate within the LSEC algorithm is analogous to the binding of λ-variables within LISP. A new stack frame is created to associate with the formal parameters all relevant information inherited from the actual parameters. At the time of binding, each formal parameter is designated as either a *constant-valued* variable, if the corresponding actual parameter is either a constant or has itself already been classified as a constant-valued variable, or an *attribute-id-valued* variable, in the case where the corresponding actual parameter is itself attribute-id-valued (by virtue of having been bound at some level to a quantified variable).

Following creation of the new stack frame, the current ADE is (recursively) constrained by the body of the defined predicate, with its current bindings. For syntactic simplicity, the body of a defined predicate may be a list of conjoined subformulas, and need not be an explicit conjunction. Typically, then, a defined predicate is processed by establishing new bindings and then evaluating the list of conjoined subformulas which make up its body.

*3.7 Primitive predicates*

It is in the processing of primitive predicates that most of the computational effort of the LSEC algorithm is expended. For a simple illustration of the computationally demanding aspects of this process, consider the processing of the simple query

$$(x, y) : \exists z.P(x, z) \wedge Q(z, y) \quad ,$$

where $P$ and $Q$ are both primitive predicates.

Since the body of the query is a conjunction, the initial ADE (*true-extension*) is first constrained by the primitive predicate $P(x, z)$. Constraint of the *true-extension* by a primitive predicate is treated as a special case by the LSEC algorithm. The resulting ADE contains a single relation, the *independent extension* of the primitive predicate in question. The independent extension of a primitive predicate is defined as the result of *selecting* the corresponding primitive relation in the extensional database on the values of any constant-valued arguments that may be specified, then *projecting* out the attributes corresponding to all such constant-valued

arguments. In our example, $P$ has no constant-valued arguments; the result is thus the degenerate case of an independent extension: a new ADE consisting of the single primitive relation corresponding to $P$.

This ADE is next constrained by the second conjoined factor, the primitive predicate $Q(z, y)$. In this more typical case, where the ADE is not equal to *true-extension*, the independent extension of $Q(z, y)$ is first computed by selection and projection of the primitive relation corresponding to $Q$, as described above, and then *joined* with the current ADE over all common attributes. In our example, the independent extension of $Q(z, y)$ would be joined with the old ADE (the independent extension of $P(z, z)$) over the common existentially quantified variable $z$. In our knowledge-based retrieval task, this join operation, which would in general be very expensive on an ordinary machine in the case where the relations involved are of large cardinality, occurs quite frequently in the course of executing the LSEC algorithm, and would probably account for most of the execution time in a realistic application. It is the need to perform such joins (or similar operations) in a highly efficient manner which thus provides what is probably the most important justification for the use of *parallel* hardware in the kinds of knowledge-based applications with which we are concerned.

### *3.8 The result formula*

Up to this point, we have considered only the treatment of existentially and universally quantified variables by the LSEC algorithm. It will be recalled, however, that the "top-level" query formula must always contain at least one, and possibly more, *free variables*, all possible satisfying combinations of which will ultimately be returned as the *result* of the query. During the bulk of the LSEC algorithm, free variables are treated in exactly the same manner as existentially quantified variables: distinct attribute-ids are created for each, and the associated information stored on the logical variable stack without any indication of their special status.

After the initial ADE (*true-extension*) has been constrained by the full well-formed formula, however, each of the relations in the resulting ADE is *projected* over the result variables, and the *union* of the (necessarily union-compatible—see Subsection 7.4) resulting relations is taken to yield the query result. In our demonstration system, for example, each relation in the final ADE is projected over the attribute-id corresponding to the top-level target document descriptions, and the union of the resulting unary relations—a new relation listing each of the matching targets—is displayed to the user.

### *3.9 On the complexity of LSEC*

As we have already noted, the NON-VON machine is designed to execute the primitive operations of the LSEC algorithm in a highly efficient manner. (The reader is referred to Shaw [1979] for the algorithms themselves, and to Shaw, et al. [1981] for details of the NON-VON architecture.) Since a number of these relational algebraic operations will in general be required in the course of an actual retrieval task, however, it is reasonable at this point to consider the complexity of the LSEC algorithm *in terms of* these relational algebraic primitives. First, it should be noted that the individual who constructs the set of defined predicates (which, in our demonstration system, implement the match semantics) may exercise a considerable degree of explicit control over the sequence of operations that will ultimately be performed in the course of executing the

LSEC algorithm. In practice, it has been our experience that predicate definition is an activity more nearly like ordinary (albeit very high-level) programming than, say, the analogous task confronting the architect of a resolution theorem proving system. In particular, it is possible to define two "weakly equivalent" sets of predicates—that is, two sets which are indistinguishable on the basis of their input/output behavior under the LSEC algorithm—such that one is considerably more efficient than the other.

It has been our experience that the number of relational algebraic operations which occur in the course of retrieving target descriptions matching pattern descriptions of realistic size and complexity is fairly modest (no more than a few dozen such evaluations for the most detailed of our test descriptions, for example). To be sure, the number of such operations could in theory grow quite rapidly as the size of the pattern description grew very large—the exact behavior depending both on intrinsic characteristics of the high-level description language and on factors under the control of the individual responsible for predicate definition. In practice, however, the issue of query size and complexity is much less important than that of database size, particularly in the case of the very large databases to which our research is directed. In this regard, it is the fact that the number of relational algebraic operations, while directly related to query complexity, is *independent* of the size of the database, which is of central concern. The critical determinant of system behavior in realistic large-scale database applications is thus the efficiency with which the individual relational algebraic primitives—particularly the join operator, by virtue of its complexity and frequency of invocation within LSEC—are performed on the underlying machine. It is here that the NON-VON architecture offers a potentially dramatic performance improvement (with a comparable investment in hardware) over conventional computer systems.

## References


Bobrow, Daniel G., and Winograd, Terry, "An Overview of KRL-0, a Knowledge Representation Language", *Cognitive Science*, 1 (1) (1977).

Chang, C. L. and Lee, R. C. T., *Symbolic Logic and Mechanical Theorem Proving*, Computer Science and Applied Mathematics Series, Academic Press, Inc., New York (1973).

Codd, E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, 13 (6) (June 1970).

Codd, E. F., "Relational completeness of data base sublanguages", in Rustin, Randall (ed.), *Courant Computer Science Symposium 6: Data Base Systems*, Englewood Cliffs, New Jersey, Prentice-Hall, Inc. (1972).

Gallaire, Herve, Minker, Jack, and Nicolas, J. M., "An overview and introduction to logic and data bases", in Gallaire, Herve and Minker, Jack, *Logic and Data Bases*, New York, Plenum Press (1978).

Reiter, R., "An approach to deductive question-answering", BBN Report No. 3649, Bolt, Beranek and Newman, Inc., Cambridge, Mass. (September 1977).

Shaw, David Elliot, "A Hierarchical Associative Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives", Stanford Computer Science Department Report STAN-CS-79-778 (October 1979).

Shaw, David Elliot, "A Relational Database Machine Architecture", *Proceedings of the 1980 Workshop on Computer Architecture for Non-Numeric Processing*, Asilomar, California, (March 1980).

Shaw, David Elliot, *Knowledge-Based Retrieval on a Relational Database Machine*, Stanford Ph.D. Dissertation and Stanford Computer Science Department Report STAN-CS-80-823, (August 1980a).

Shaw, David Elliot, Ibrahim, Hussein, Wiederhold, Gio and Andrews, J. A., 'A Highly Parallel VLSI-Based Subsystem of the NON-VON Database Machine', Columbia Computer Science Department Report, (July 1981).