Distinguishing Knowledge Bases and Data Bases: Who's on First and What's on Second

Technical Note No. 546

August 23, 1994

By: Peter D. Karp, Sr. Computer Scientist
Artificial Intelligence Center
Computing and Engineering Sciences Division

This work was supported by Grant No. R29-LM-05413-01A1 from the National Library of Medicine.

SRI International

333 Ravenswood Ave. • Menlo Park, CA 94025 • (415) 326-6200 • TWX: 910-373-2046 • Telex: 334486

Abstract

This paper compares and contrasts relational data base management systems (RDBMSs) and frame knowledge representation systems (FRSs). We compare the capabilities that RDBMSs and FRSs provide to designers and users of computer information systems. We consider capabilities at three different levels: symbol-level capabilities such as storage capacity, persistence, and shared use; system-engineering level capabilities such as run-time schema alteration, design guidelines for schema organization, inheritance, and classification; and knowledge-level capabilities such as expressive power and inference. No previous publication considers the full range of differences between these systems that are presented here. We also consider the factors that have shaped the current differences between FRSs and RDBMSs, and present trends in the continuing evolution of these systems. FRSs are in danger of extinction should their architects not provide them with certain symbol-level DBMS capabilities.

1 Introduction

To many observers, data bases and knowledge bases must appear as very similar technologies. Both types of systems resulted from research whose goal was to identify general information management functions that are common to many application programs. Once identified, these functions are extracted from the applications and consolidated in a general, application-independent information management system that can be reused by a diverse set of applications. Although they were motivated by similar goals, existing data bases and knowledge bases differ in a large number of respects. That is, researchers extracted and generalized *different* information-management capabilities from the application programs they considered. The central focus of this paper is to improve our understanding of these differences. This understanding is important because these systems have complementary strengths and weaknesses, and each can be improved by adopting some strengths of the other.

This paper can also be viewed as a call to arms: given that both types of systems have advantages with respect to the other, why do data base systems have the overwhelming popularity in the commercial world? Although I believe that knowledge representation systems (KRSs) have been shaped by a wealth of excellent ideas, we will soon see the demise of these systems if their shortcomings are not remedied. Since data base researchers have been working overtime to include AI capabilities in data base systems, the consumers of information-management systems will soon have little reason to employ KRSs. But I propose that the advantages of KRSs are overshadowed by only a few shortcomings, which if rectified will lead to a quantum leap in their utility and popularity. Sections 2 and 7summarize my recommendations to the KR community as to how to improve the utility of KRSs. Unless these recommendations are followed, I believe that the great potential of KRSs will be lost.

Let us use the terms "data base" and "knowledge base" to refer to the contents of an information management system; we use the terms "data base management system" (DBMS) and "knowledge representation system" to refer to the information management system itself. But even these terms refer to large classes of systems such as network DBMSs, relational DBMSs, and object-oriented DBMSs, plus frame KRSs, rule-based systems, logic programming systems, and probabilistic reasoning systems. Most previous authors have discussed DBMSs and KRSs in the abstract, rather than focusing on subtypes of these systems such as RDBMSs and FRSs. But a meaningful abstract comparison is virtually impossible because of the difficulty in formulating reasonably precise (and true) statements about such a large class of systems.[1] Comparing every type of DBMS with every type of KRS would be a fascinating but daunting task that is beyond the energies of this author. Therefore, this paper focuses on two subclasses: we compare relational DBMSs (RDBMSs) with frame knowledge representation systems (FRSs) (also known as *semantic networks*) [1, 2, 3]. FRSs are the type of AI system most similar to a DBMS. FRSs and RDBMSs both represent mature, commercialized technology that has existed for approximately 20 years.[2] We also do not consider research on logic data bases, such as research on combining PROLOG with RDBMSs.

This paper is aimed at researchers and students in both the data base (DB) and AI communities. I ask for the reader's patience when reading elementary definitions; researchers in one field may not be familiar with terminology from the other, and it is useful to have a single publication that is accessible to workers in both fields.

[1] In contrast, there are some authors who interpret "KB" to mean virtually any type of AI program (many of which have no resemblance to DBs), and still others who interpret "KB" to mean a very narrow class of rule-based systems such as MYCIN.

[2] Although in some respects object-oriented DBMSs are more similar to FRSs than are RDBMSs, there has been more effort to incorporate KRS characteristics into RDBMSs than into object-oriented DBMSs, and therefore it will be more interesting to examine what differences this research focuses on, and the degree to which it has succeeded.

This paper compares the *capabilities* that RDBMSs and FRSs provide to users and designers of computer information systems. Previous authors have discussed only some of the capabilities that are considered here [4, 5, 6, 7, 8]. We begin by discussing the historical factors that influenced data base and AI researchers to design RDBMSs and FRSs with their present capabilities. We then discuss each capability along which these systems differ, noting how each capability is evolving due to current research. I have grouped the capabilities into three different levels — levels that were defined by Brachman and Levesque [9] and derived from work by Newell [10]. The *symbol level* is concerned with implementation-level capabilities of an information system, such as its speed, its storage capacity, and its ability to maintain data integrity in the face of hardware and software failures. *System-engineering level* capabilities facilitate the understandability and maintainability of an information system from the perspective of the information-system designer. Examples of system-engineering level capabilities include mechanisms for maintaining high levels of data integrity, for protecting information from unauthorized access, and for changing the schema of an information system without recompiling it. Finally, at the *knowledge-level* we describe the functional operations that the information system provides to a user, and provide an epistemological account of what types of information the system is able to represent. For example, is the system able to represent general statements (e.g., universal quantification), or only particulars (ground axioms)? Table 1 summarizes the findings of this paper.

2 Recommendations for KRS Research

This section summarizes the capabilities that must be added to FRSs in order to make them viable competitors to data base systems (the meaning of each capability is explained later in the paper). FRSs excel at knowledge-level operations, but are lacking at the symbol level and at the systems-engineering level.

At the symbol level, FRSs require reliable, high-speed read and write access to large persistent knowledge bases. FRSs also greatly need the symbol-level capability of shared access by multiple users. At the systems engineering level, I recommend including runtime flexibility in those FRSs that lack it, and adding security features modeled after those of data bases. I also suggest research into constraints on FRS schema organization that are analogous to the data base normal forms. FRSs will also benefit from the declarative query languages that are under development within the knowledge sharing effort. Finally, FRSs must be accessible to programming languages other than LISP.

My research group at SRI is pursuing some of these recommendations. We are working with the FRSs LOOM [11] and THEO [12] to endow them with high speed access to large persistent knowledge bases, and with shared multi-user access. Our approach is to utilize an existing DBMS as a storage subsystem for a FRS. The DBMS is submerged within the FRS, and is invisible to FRS users. The FRS requests frame data from the DBMS on demand as the application accesses different frames, and only modified frames need be saved back to the DBMS. We have prototype implementations working for both LOOM and THEO, and we are beginning performance studies to determine which of several organizations for frame data within the DBMS is optimal for different applications. We are also experimenting with several DBMSs (relational, object-oriented, and extensible) to determine which is optimal as a frame storage subsystem.

Capability	Presence in FRSs	Presence in RDBMSs
Knowledge Level		
Expressiveness	generalizations, particulars, defaults,definitions	particulars, defaults
Inference	inheritance, classification, production rules	production rules (in newer systems)
System-Engineering Level		
Data primitive	frame	relation
Inheritance	yes	no
Runtime flexibility	more	less
Schema organization constraints	no	yes
Declarative query language	sometimes, nonstandard	yes, standard
Programming language	lisp	c
Derived data	yes	yes
Security	no	yes
Use of value constraints	restriction, classification	restriction
Symbol Level		
Persistence and reliability	experimental	yes
High speed access to large information volumes	experimental	yes
concurrency and sharability	experimental	yes

Table 1: Summary of RDBMS and FRS capabilities.

3 Influences on System Capabilities

Just as the "data base concept" involves extracting common data management modules from application systems to create application-independent data base management systems, so the "knowledge base concept" involves creating application-independent knowledge management systems with commonly needed functionality such as production-rules and inheritance. Why did AI and data base researchers extract different information management functions from the applications they studied?

Many of these differences can be traced to differences among the applications that the two communities studied. Diverse applications yield diverse system requirements. But because the applications did sometimes yield the same requirements, this explanation is not sufficient. A second major factor is the differing research cultures of the data base and AI communities — their general goals and their training. Dissimilar cultures can cause different researchers to see different requirements in the same application, to focus on different research goals, and to satisfy similar requirements with different implementation techniques and formalisms.

Figure 1 illustrates how the applications and the research culture of the data base community influenced

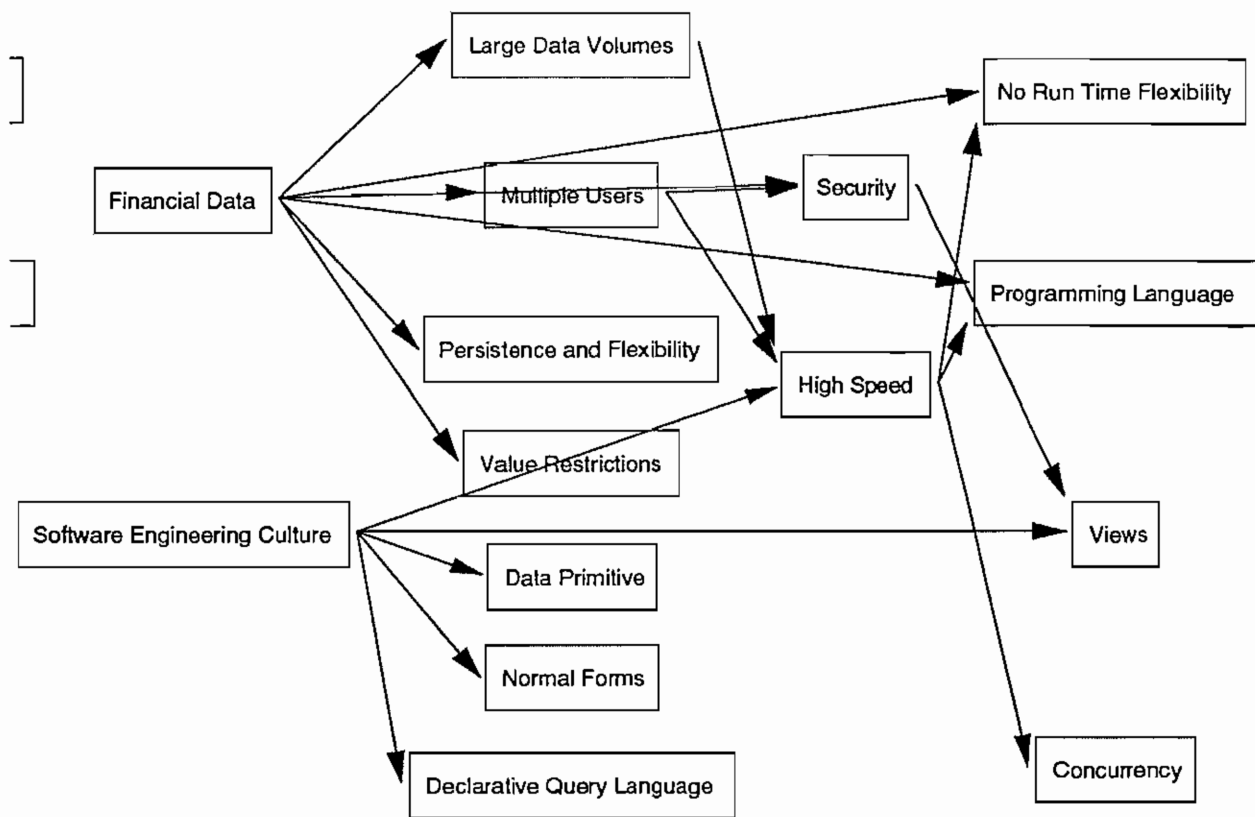

Figure 1: Influences on the development of RDBMSs.

the characteristics of RDBMSs. A representative class of driving applications involved management of large volumes of financial data; these applications required that multiple users be able to access large amounts of data; the economic value of the data required that it be stored persistently and reliably, and that constraints on its correctness be defined. In turn, these characteristics had other effects: high performance became crucial given the volume of data and the number of users. Concurrency increases performance, as does a lack of run-time flexibility — which is acceptable because of the relatively simple semantics of the domain. A number of the software-engineering level characteristics of RDBMSs are not strictly required by the driving applications, but are derived from the data base culture, which includes the use of good software engineering practices (views, declarative query languages), and ideas from theoretical computer science (a data primitive based on set theory, formal constraints on schema organization).

Figure 2 illustrates influences on the characteristics of FRSs. Early FRSs grew out of research in natural-language understanding and expert systems. Natural-language researchers were particularly interested in encoding *definitions* of natural-language concepts — in specifying necessary and sufficient conditions for what it means to be, for example, a human, a male, a man, or a boy. Thus, the frame data primitive was associated with a concept early on, and it was natural to link these concepts in an inheritance hierarchy organized by the subsumption relation. Run-time flexibility was essential because a given KB might encode hundreds or thousands of different concept definitions that were constantly evolving because of their complexity. A change to any concept definition corresponds to a data base schema change. Expert-system applications such as diagnosis problems reinforced the need for inference, and benefitted from reason-maintenance capabilities that support reasoning under different assumptions.

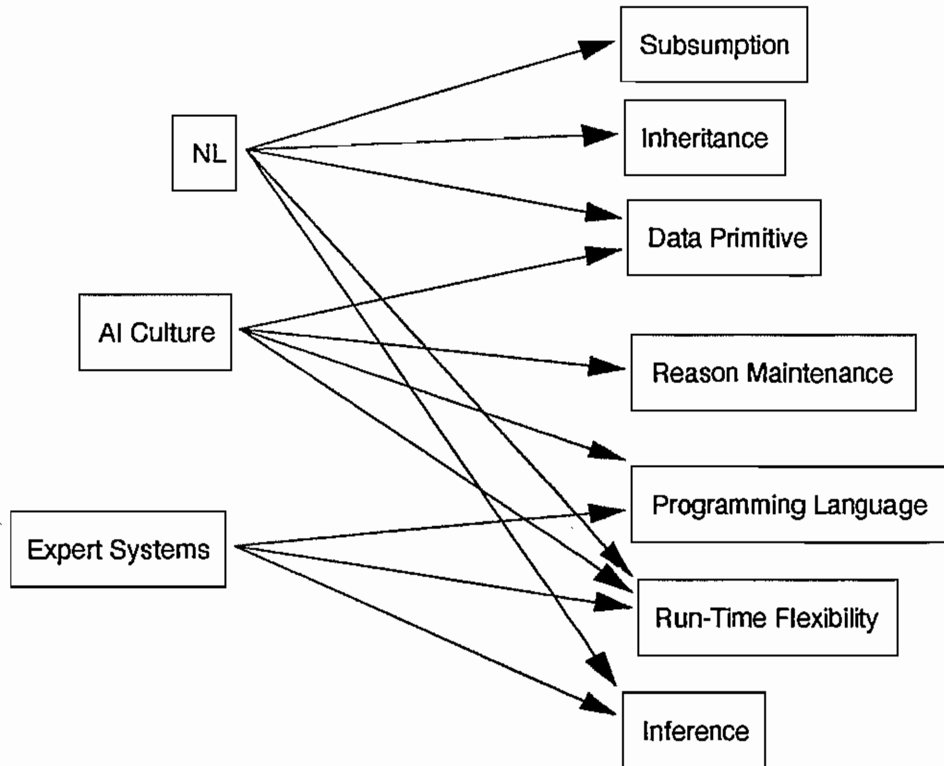

Figure 2: Influences on the development of FRSs.

But an important question remains. Natural-language applications could require encoding tens of thousands of concepts. Applications such as VLSI, computer configuration, and CAD/CAM — all of which have been addressed by AI researchers — can involve tens or hundreds of thousands of components that in the general case must be accessed simultaneously by multiple users. Why has it taken 20 years for the FRS community to begin to address requirements such as multi-user access, and fast performance for large data volumes?

I propose three answers. First, all of these traditional FRS applications differ from typical data base applications such as banking and airline reservation systems by several orders of magnitude in the typical rate of *write* transactions, thus allowing multiple KB updates to be performed sequentially; a KB can then be replicated for read or non-permanent write access by multiple users. Second, applications such as natural-language understanding and expert systems involve a vast array of challenging research problems (e.g., problems of computational linguistics or of how to perform diagnostic problem solving) that can sap all of a researcher's attentions for years with a KB size of hundreds of frames. One could argue that it has taken 20 years of AI research to advance problem-solving methods to the point where very large KBs can be properly employed. Third, a member of the AI research culture is almost by definition more likely to be attracted to research goals involving knowledge-level support for problem-solving tasks than with the symbol-level capabilities that are lacking in most FRSs. The community views knowledge-level tasks as more central to its primary mission of understanding intelligence. Although this proposition is probably true, I would also claim that this inattention to engineering is a major reason for the potential demise of FRSs.

4 The Knowledge Level

A knowledge-level account of an information system is concerned with two types of issues. First, what can the information system know? That is, what types of knowledge can it represent, and what can it infer from the knowledge it encodes explicitly? Second, what functional operations does the system provide to the user for accessing and modifying the contents of the information base?

4.1 Expressiveness

One way to characterize the expressive power of an information system is to use a formalism such as first-order logic to model the representational structures of the system. We can then compare the expressiveness of two different information systems by comparing the models of these systems in that common formalism. Several authors have produced logical models of RDBMSs and of FRSs [13, 14, 15, 16] that illustrate differences in the expressive power of these systems.

One difference is that RDBMSs are geared towards encoding knowledge of *particulars,* whereas FRSs can represent knowledge of particulars and *generalizations.* RDBMSs store information about particular objects in the world, such as the manufacturer of a particular computer or the bank balance of a particular person. Although an instance frame can easily encode the preceding information, class frames provide the additional power to represent knowledge about entire classes of entities (universals), such as the fact that all workstations are computers, and the generalization that by default the `Word_Size` in bits of all members of the VAX family of computers is 32. There are two exceptions to the rule that RDBMSs cannot represent generalizations. The first is integrity constraints, which allow RDBMSs to make general assertions about what values are and are not allowed in a particular attribute. Second, some RDBMSs can associate default values with a given column of a relation. These defaults are overridden if an application asserts a value into that column. These defaults are not as powerful as those found in FRSs because FRS defaults can flow through a number of links in a taxonomic hierarchy, and be modified more than once in the specializations of a class. The defaults in different RDBMS relations are completely independent.

RDBMSs typically make the *closed world assumption:* they assume that if the DB does not explicitly contain some fact then the fact must be false. For example, if our DB listed three companies that can repair the VAX-11/780, we probably do not want to assume that these are the only three such companies in the world. Some FRSs make this assumption and some do not, and it varies among applications whether this assumption is warranted.

It is important to note that increased expressiveness does not come for free: Levesque and Brachman [14] show that there is a tradeoff between the expressive power of an information system and the computational cost of performing inference (classification) within that system. (Of course, most data base systems do not perform classification).

Wiederhold distinguishes DBs from KBs by distinguishing data from knowledge [17]. He considers knowledge to consist of generalizations and abstractions that are used for decision making, and believes that knowledge can be obtained only from experts. In contrast, data "reflects the current state of the world at the level of instances" and is much more detailed than is knowledge — "if we can trust an automatic process or clerk to collect the material then we are talking about *data*." Because data reflects the state of the world, we expect data to be voluminous and to change quickly.

This distinction does capture some of the typical differences between the information typically stored in DBs and KBs. However, we just noted that FRSs can represent both generalizations and instances,

therefore KB technology can represent both knowledge and data, whereas many authors seem to equate KB technology with rules (a particular type of generalization). Similarly, a RDBMS is able to store information that is not voluminous, or that does not change quickly. Therefore, it would be inaccurate to say that the central difference between KB technology and DB technology concerns their abilities to represent data and knowledge of the sort that Wiederhold discusses.

4.2 Inference

Most FRSs provide one or more inference capabilities. These reasoning facilities allow a FRS to use the information explicitly stored in a KB to derive new information that is logically implied by the current contents of the KB.

One common type of inference is the mechanism of inheritance that will be discussed in Section 5.2. That section discusses how a child frame inherits the slot present in a parent frame, plus integrity constraints associated with that slot. In addition, inheritance will cause *values* stored in the parent slot to be propagated to the child slot — unless the user stores new values in the child slot to override the parent values. This type of inference is called default reasoning because we assume by default that the information stored at the parent applies to the child unless that assumption is explicitly overridden by the KB. (There is considerable variation among FRSs in exactly how inheritance of default information is computed.) RDBMSs can supply a default value for a column in a relation, but that default cannot apply several related relations.

A second type of inference used in FRSs is based on production rules of the form `If X Then Y` that are used to derive a set of conclusions `Y` when a set of conditions `X` are satisfied. Traditional production systems referenced a working memory — a list of assertions that represent the current state of knowledge. A production rule interpreter executes a set of rules by repeatedly scanning through the rules, evaluating their conditions with respect to working memory, and adding to working memory the conclusions of rules whose conditions are satisfied. When production rules are coupled to a FRS, the KB becomes the working memory: rule conditions query slot values and rule conclusions modify slot values. This architecture allows us to build expert systems by encoding problem-solving expertise in production rules, and by encoding background knowledge, the initial conditions of the problem, and the evolving problem solution in the frame KB. (In fact, most systems also encode production rules themselves as frames.)

The classification operation discussed in Section 5.9 is a third type of inference. An FRS that computes the subsumption relation compares the definitions of two frames to infer that the definition of one frame A subsumes the definition of another frame B. Classification is used to perform problem solving in three different ways. First, when the FRS concludes that A subsumes B, it links B under A in the inheritance hierarchy; inheritance may then propagate certain knowledge about A to B. Second, the conclusion that A subsumes B may in itself be the solution to a problem, if for example A is the definition of a disease and B is a description of a person suffering from that disease. Third, classification can be used for query answering: if a user issued a query to retrieve all frames representing 32-bit computers manufactured by American companies, the query processor would create a class frame that had these properties. Once classified, the set of instance-frame descendants of that class frame would comprise the answer to the query.

The *reason maintenance systems* that exist in a number of FRSs (see, for example, PROTEUS and KEE [18, 19, 20]) provide additional support for inference. These systems record the dependencies that underly derived beliefs. For example, a conclusion asserted by a production rule is dependent on the facts that satisfied the conditions of the rule, and on the rule itself. This capability is useful

NAME	MANUFACTURER	WORD_SIZE
PDP-11/60	DIGITAL Equipment	16
VAX-11/780	DIGITAL Equipment	32
SUN-3	SUN Microsystems	32
SPARCstation 1	SUN Microsystems	32

Table 2: The PROCESSOR relation. Each row of the table is one tuple, and each column of the table is one attribute. In the tuple describing the VAX-11/780, the value of the NAME attribute is the string VAX-11/780, and the value of the MANUFACTURER attribute is the string DIGITAL. The NAME attribute is a *key* for this relation because this attribute has a unique value in every tuple.

because it allows us: to simultaneously maintain multiple alternative evolving solutions to a problem, to represent the information that is common to these alternative problem solutions more compactly, to detect when an potential solution becomes internally inconsistent, and to generate explanations of problem-solving behavior.

None of these inference capabilities exist in traditional RDBMSs. But recently data base researchers have integrated into RDBMSs rule systems (also called triggers) that were modeled after the production-rule systems of FRSs and other knowledge-based systems. Within RDBMSs, rules are used for a variety of purposes including enforcing data integrity, providing security, and providing derived data capabilities. Surprisingly, although "AI applications" are often cited as a reason for including rule capabilities in RDBMSs, none of the publications that this author encountered actually used RDBMS rules to implement the type of inference found in typical expert systems, such as diagnosis or classification problem solving. Therefore, this paper discusses RDBMS rules in the context of derived data (Section 5.7), not in the context of inference.

5 The System-Engineering Level

The system-engineering level of an information system defines operations that facilitate the design, maintenance, and understandability of information bases. These capabilities are typically used by the designers and managers of an information base rather than by its users.

5.1 Fundamental Data Primitive

RDBMSs and FRSs have as their basis two different primitive data abstractions. Because the designer of every information base seeks to transform her own mental conceptualization of the application domain into a conceptualization based on the data primitive of the information system in use, that data primitive occupies a central role in information-base design.

The data primitive of a RDBMS is the *relation.* A relation is usually compared to a table that consists of an unordered set of rows, each of which contains the same ordered sequence of columns. Each row of the table corresponds to a *tuple* of the relation, and each column of the table corresponds to an an *attribute* of the relation. In RDBMSs every attribute must be an atomic datatype such as an integer or a string. The relational model has a mathematical foundation that provides a set of operations for manipulating relations, such as the join, projection, and selection operations. Tables 2 and 3 show two sample relations.

NAME	OPERATING_SYSTEM
PDP-11/60	RSX-11
PDP-11/60	UNIX
VAX-11/780	VAX/VMS
VAX-11/780	UNIX
SUN-3	UNIX
SPARCstation 1	UNIX

Table 3: The OPERATING_SYSTEM relation.

The data primitive of a FRS is the *frame*. Every frame has a unique name, and every frame contains a set of *slots*. In turn, every slot consists of a set of *facets*, one of which is the *value* facet. (There is less consensus as to exactly what a frame is than there is as to what a relation is; put another way, there is considerable variation in exactly how different FRSs define a frame.) Figure 3 shows a sample frame. The value facet of a slot holds the principal value of that slot; generally slot values can be any LISP S-expression, such as a number, a string, a list of numbers or strings, or a more complex list structure. The other facets associated with a slot hold other information associated with the slot such as a comment, constraints on the value of the slot, justifications that describe how the value of the slot was computed, and the names of procedures that can be used to compute the value of the slot. Predicate calculus forms a partial underlying theory for FRSs.

The tuple is analogous to the frame because a DB consists of a collection of tuples (usually from a number of different relations), and a frame KB consists of a collection of frames. More specifically, the tuple is analogous to the instance frame; class frames are analogous to relations (or to the definition of a relation in the schema). The slots of an instance frame are analogous to the attributes of a tuple, but they differ in that the facets of a slot associate more information with the slot than does the single value of an attribute. They also differ because slot values need not be atomic — they can be lists or sets. Complex datatypes are explicitly forbidden by the normalization rules of the relational model, which aim to limit data complexity.

Another difference between the two types of systems is that RDBMSs contain a special structure called the *catalog* (or *data dictionary*) that describes what relations exist in a given DB, and what the allowable attributes of each relation are (the relational *schema*). The catalog would be used to describe the relation PROCESSOR in Table 2 as having attributes NAME, MANUFACTURER, etc. The catalog is stored in a special predefined set of relations in the RDBMS. FRSs do not contain a separate catalog. Instead, users create *class* frames that define templates for the creation of *instance* frames. The set of class frames in a given KB constitutes its schema. Every instance frame is defined to be the child of some class frame; this definition means that the instance inherits all of the slots contained in the class frame. It is also worth noting that although RDBMS catalogs can be managed by the native query and update facilities, they are often managed using specially developed software that is external to the RDBMS itself (such as graphical tools for managing entity-relationship diagrams). In contrast, FRS schemas are managed using the query, browsing, and update facilities of the FRS itself. This approach obviates the need for developing additional software, and for importing schema definitions from software external to the information system.

The preceding differences between relations and frames are important in several respects. Using relations, it is awkward (although possible) to represent complex datatypes, or to represent information *about* a value. FRSs utilize facets extensively to represent information such as the slot datatype, constraints on the slot value, what mode of inheritance is used for the slot, a certainty factor for the

```
frame: VAX-11/780 in knowledge base KBSDBS
instance of: PROCESSORS

slot: MANUFACTURER from VAX-11/780
  inheritance: OVERRIDE.VALUES
  valueclass: CORPORATIONS
  cardinality.min: 1
  comment: "Names the manufacturer of this computer."
  values: "DIGITAL Equipment"

slot: WORD_SIZE from VAX-11/780
  inheritance: OVERRIDE.VALUES
  valueclass: INTEGER
  cardinality.min: 1
  cardinality.max: 1
  comment: "Specifies the word size of this computer
           in bits."
  values: 32

slot: OPERATING_SYSTEM from VAX-11/780
  inheritance: OVERRIDE.VALUES
  cardinality.min: 1
  comment: "Names the operating system(s) that may be
           used by this computer."
  values: VAX/VMS, UNIX
```

Figure 3: A sample frame called VAX-11/780. This frame is a child of a class frame called PROCESSORS. The MANUFACTURER slot has the value "DIGITAL Equipment", and its value is constrained to name a frame that is an instance of the class CORPORATIONS. The slot is allowed to have only one value, and inheritance for this slot is computed according to a procedure called OVERRIDE.VALUES, meaning that if values are defined in a child frame, these values override values defined in ancestor frames. This example was created using IntelliCorp's KEE system.

slot, or a justification for the value of the slot (in the truth-maintenance sense). Such facets are not considered only applications data, but are used internally by the FRS itself in various ways, acting as parameters that control the computation of subsumption and inheritance. Furthermore, facets are inherited in the class hierarchy, and their values are often modified lower in the hierarchy, such as by adding additional constraints on allowable slot values. One could argue that, for example, the datatype information stored in the RDBMS catalog is highly analogous to a datatype facet. The differences are that FRSs generally utilize more kinds of such meta information, that the inheritance of facets allows this information to easily be saved and modified in a large number of classes, and that it is easier for people to comprehend an information base when all relevant information about a complex entity (object) is gathered together in one place. That is, a relational encoding of facets and complex values would spread what could be encoded as a single slot in a single frame across a potentially large number of different RDBMS relations. For example, a slot with a list of values would be normalized to an additional relation, and each slot facet would yield a new relation, thus complicating the RDBMS schema significantly.

More recent RDBMSs provide support for user-defined datatypes and access methods for those datatypes. For example, Stonebraker extended the INGRES RDBMS to allow users to define new abstract datatypes subject to the restriction that they be of fixed storage length [21]. A sample

datatype is a two-dimensional box for a graphics application; each box is described by the coordinates of two of its opposite corner points. A user-defined operator for this type is "overlaps," which returns TRUE if its two argument boxes overlap one another. This new operator can be employed in data base query operations. [21] describes how to provide both fast data base access methods, and query optimizations, for these user-defined datatypes. These latter capabilities go beyond the abstract datatype capabilities of FRSs, which provide neither fast access methods nor query optimization for the structured data that might be stored in a slot.

5.2 Inheritance

An information base that describes a very complex application domain is likely to contain many different types of entities: a RDBMS would contain many relations, and a FRS would contain many class frames. Put another way, the schema of such an information base will be very complex. Furthermore, the different types of entities defined in the schema may be very similar to one another — types may differ only by a few additional properties or integrity constraints. The FRS capability of inheritance allows a KB designer to define a new class of frames that is a child of an existing class, and that therefore acquires the slot definitions, slot integrity constraints, and slot values defined in the parent. This information can be supplemented, modified, or replaced in the child frame. Inheritance is therefore an extremely useful device for managing complex KB schemas because these schemas tend to contain many similar classes, and because inheritance allows subclasses to be defined as modifications of superclasses.

Inheritance is also a powerful tool for both conceptual modeling and modular programming. It promotes compact representations. It also simplifies knowledge-base extensions because special cases can be defined by modifying existing generalizations, and because KB modifications are systematically propagated from a modified class to all descendants of that class in the knowledge base. Finally, it allows the knowledge engineer to establish default values for slots in instance frames that can be overridden in the presence of additional information.

RDBMSs do not provide inheritance — all relation definitions are completely independent, requiring the designer to enter common aspects of similar relations repeatedly. Although RDBMS views (see Section 5.7) can provide a rough approximation to inheritance by constructing new relations from attributes of existing relations, this mechanism is awkward and limited. Views may combine existing attributes only, as opposed to combining existing attributes with newly introduced attributes. In addition, views cannot modify the existing attributes, for example by restricting their range of values. Nor can views provide default values.

5.3 Run-Time Flexibility

Another property of complex application domains is that they often force the information-base designer to repeatedly alter the schema of the information base as her conceptualization of the domain evolves. A complex domain such as natural-language understanding may involve the definition of hundreds or thousands of concepts, each represented by a single class. The complexity of these class definitions ensures that knowledge engineers will not enter each definition correctly the first time, and that class definitions will therefore be in flux for a long period of the KB life cycle. Each change to a class definition corresponds to a schema alteration. Dynamic schema-alteration capabilities are also useful during the prototyping stages of less complex KBs, when schemas change very frequently. Brachman has argued that in general KB schemas tend to change more often than do DB schemas [7].

Many FRSs support schema evolution extremely well by providing operations to dynamically add or remove or rename frame classes in a knowledge base, or to add, remove, or modify the slots of a given frame. These schema alterations are performed in seconds. (Note that only some FRSs provide this type of dynamic schema alteration, such as KEE and LOOM [22, 11].) IT is true that some schema alterations could require modification of every instance of a given class frame, which could be an expensive process for a large KB. The traditional limits on FRS KB size have allowed FRS designers to largely ignore the efficiency of such operations.

Conversely, changing the schema of a RDBMS can be a tedious and time-consuming process. Although it is relatively easy to define new relations and to add columns to existing relations, operations such as deleting and renaming columns, or modifying a column datatype, can force a DB manager to dump out the data in the initial set of relations, reconfigure the DB, and then reload the old data into the new relations — a very time consuming process.

Run time flexibility is likely to impose a significant performance penalty. By limiting the ways in which a schema can be altered at run time, we would expect that the architects of an information system could increase its performance. If the set of columns in a relation cannot be changed, and are of fixed size, an RDBMS can store each column at a fixed offset within a block of memory, rather than employing a slower and more complicated storage organization.

5.4 Constraints on Schema Organization

DB researchers have articulated a series of well-defined constraints on the organization of the schema for a RDBMS: the *normal forms*. These constraints are intended to prevent DB designers from designing bad schemas. A schema might be bad because it leads to redundancy in a DB, or because it increases the probability that inconsistencies will arise in the DB as a result of DB updates. A relation is in first normal form if all of its attributes contain atomic values that have no internal structure. A relation is in second normal form if it is in first normal form and if all of its nonkey attributes are fully dependent on every key of the relation. A relation is in third normal form if it is in second normal form, and if all of its nonkey attributes are not transitively dependent on any key of the relation. See [23] (p96) for more detailed definitions.

In contrast, AI researchers have not developed comparable design guidelines for FRSs.

5.5 Declarative Query Languages

Most RDBMSs support a language called SQL that is an implementation of relational algebra — operations on relations that produce relations as results. Users employ SQL (and other similar languages) to specify DB queries and updates, perhaps across a network. Although declarative query languages have been developed for several FRSs (see for example the languages of KEE, PROTEUS, KRYPTON, and LOOM [22, 24, 18, 19, 25, 26]), it is significant that no language has gained anywhere near the degree of standardization or widespread acceptance of SQL. Nor have any FRSs provided interfaces that allow queries or updates across a computer network.

Work is under way to improve this situation by developing a standardized query and assertion language for knowledge representation systems, called KQML [27]. This work is part of a larger standardization effort within the knowledge representation community whose goal is to facilitate the reuse and sharing of KBs, as well as the interoperability of knowledge representation systems with other computer systems, such as data base systems [27]. Other activities within this effort including the development

of a standard language, or "interlingua," for knowledge interchange; development of standards for the operation of individual families of knowledge-representation systems, such as the KL-ONE family of FRSs; and development of methodologies for knowledge sharing, such as for collaborative knowledge base design.

5.6 Application Programming Language

Although many RDBMSs and FRSs provide a high-level query language, it is also possible to use a traditional programming language to write application programs that manipulate a DB or a KB. Programmers typically use the languages C, COBOL, or PL/I to manipulate RDBMSs, whereas for FRSs the language COMMON LISP is typically used (exceptions include IntelliCorp's ProKappa and C-CLASSIC [28]).

5.7 Derived Data

Traditional RDBMSs provide a capability called *views* that allows a DB designer to define new "virtual relations" whose values are derived on demand from relations that are physically stored in the DB. Views are usually defined using SQL operations such as selection to construct a new relation by combining columns of existing relations. Views have several uses: they can make a DB look different to different classes of users (simplifying views might omit data for less sophisticated users), and they provide logical data independence by insulating the user from changes to the DB schema (such as addition of attributes to a relation or a rearrangement of attributes among relations).

Some FRSs provide an analogous mechanism called access-oriented programming [29]. An AOP facility allows the programmer to associate programmatic annotations with data, such that the annotations are executed when different classes of operations are performed on the data (in a FRS, these data are slot values). The annotations can be dynamically attached to and removed from the data, and the existence of the annotations is transparent to programs that are not explicitly attempting to manipulate the annotations (i.e., to programs that are only attempting to manipulate the data). Usually the annotations are LISP procedures, but in THEO, for example, annotations can be PROLOG rules [12]. AOP is somewhat more general than views since SQL is less expressive than a general-purpose language such as LISP.

One common type of annotation function is invoked when a user requests the value of a slot; the function computes the value of the slot and returns the value to the user in a transparent fashion — as if the value were stored in that slot. Another common type of annotation function is invoked when a user modifies the value of a slot. The annotation function might update other slots, or perhaps external data bases, whose values depend on the modified slot. In FRAMEKIT for example [30], a user can annotate a slot by storing LISP functions in facets called `If-Needed`, `If-Accessed`, `If-Added`, and `If-Erased`. The `If-Needed` function will be invoked if a user attempts to get the value of the associated slot, and if that slot currently has no value; if the slot does have a value then the `If-Accessed` function (if any) will be invoked instead. The `If-Added` function will be invoked when the user adds a new value to a slot, and the `If-Erased` function will be called when the user erases the value of a slot. SRL has a more general mechanism [31]. An SRL annotation is itself described by a frame that specifies such things as: by what type of slot operation the annotation is to be invoked (e.g., a get or a put operation); whether the annotation should be invoked before or after that slot operation is performed; and what "effect" the annotation has — does it alter the value returned by

the slot operation, does it have a side effect that does not alter the returned value, or should the slot operation be completely blocked from occurring?

More recent RDBMSs provide derived (active) data using rules (triggers) [32, 33, 34, 35, 36]. We discuss two examples: Postgres rules and the ECA model.

Postgres rules are derived from a modification to the Postgres query language — rules are queries that "run indefinitely." For example, the meaning of the rule

```
replace always EMP (salary = E.salary) using E in EMP
where EMP.name = "Mike" and
E.name = "Bill"
```

is that the value of Mike's salary is always derived from the value of Bill's salary [35]. The actual implementation of this rule can take two forms, corresponding to forward and backward chaining, which Stonebraker et al call early and late evaluation. In early evaluation, the rule is executed whenever Bill's salary changes, thus propagating Bill's salary to Mike. In late evaluation, the rule is executed whenever Mike's salary is queried. The choice of which form of evaluation to use can greatly affect system performance (a well-known lesson from AI production systems); the Postgres optimizer automatically determines which form of evaluation to use based on the frequency of reads and writes to the relations referenced in a rule. Stonebraker et al discuss interdependencies between the rule system and other parts of the RDBMS, for example, it is not possible to index relational attributes that are referenced in rules with late evaluation.

In the ECA model, each rule consists of an event, a condition, and an action. When the data base event associated with a particular rule occurs, then the condition of the rule is evaluated. If that condition is true, then the rule action is executed. In a sense, events are simply a distinguished subclass of conditions. Events include timer alarms, hardware failures, and updates, retrieves, or inserts to a particular relation. ECA conditions consist of arbitrary queries over the data base. An action is an arbitrary program that can include operations over the data base, as well as other computations such as interactions with the user. Thus, we would model the preceding Postgres rule in its late-evaluation form in the ECA model with a rule whose event specified a retrieve of Mike's salary, whose condition is null, and whose action returned Bill's salary.

As mentioned earlier, RDBMS rules are used for a variety of purposes including enforcing data integrity, providing security, and providing derived data capabilities. In the ECA model, for example, data integrity could be enforced using a rule that is activated by an insertion event, whose condition included the integrity constraint, and whose action executed the insertion. Security could be implemented using a rule that is activated by a retrieve event, whose condition checked the identity of the retrieving user, and whose action executed the retrieve.

Three observations apply to the FRS and RDBMS derived-data capabilities. It is interesting that Postgres rules were derived from the Postgres query language, because many FRSs derived their query languages from their rule language. This duality suggests that in general rule-languages and query languages should be designed together. The use of RDBMS rules for security suggests that FRS AOP facilities could provide a simple way of implementing this feature in FRSs, which generally lack security capabilities. Finally, note the similarities between the SRL model of AOP and the ECA model of triggers; it would seem worthwhile to unify the two models.

5.8 Security

Another use of views in RDBMSs is to protect regions of a DB from access by unauthorized classes of users. The DB manager specifies for every user what operations that user can perform on every relation in the DB, such as whether he can read a relation, modify its tuples, insert new tuples, or generate an index for the relation. The manager might create a view R' of a relation R such that R' includes only a subset of the columns in R, and grant certain users access to R' but not to R.

FRSs generally provide no means of restricting user access to frames since FRSs have traditionally been single-user systems.

5.9 Value Restrictions and Classification

Both RDBMSs and FRSs provide users with the ability to define the datatypes of attributes and slots, as well as to define restrictions on the allowable values of attributes and slots. For example, we could define an integer attribute or slot whose value was constrained to be between 1 and 200. Both the RDBMS and the FRS would signal an error if the user attempted to assign a value of -1 to this attribute or slot. Similarly, in the processor relation in Table 2 we could specify that the value of the `MANUFACTURER` attribute must be the key of a tuple from the `MANUFACTURERS` relation; and in Figure 3 we could specify that the value of the `MANUFACTURER` slot must be the name of an instance of a `MANUFACTURERS` class.

However, some FRSs use value restrictions in a way that RDBMSs do not: to compute the *subsumption* relation between class frames. Class A subsumes class B if A is more general than B, meaning that every instance of B is by definition an instance of A. For example, imagine that in addition to the `PROCESSSORS` class we wished our KB to contain a class called `JAPANESE_PROCESSORS`. This class has the exact same slots as `PROCESSORS`, except that the `MANUFACTURER` slot of the former is constrained to name an instance of another new class called `JAPANESE_MANUFACTURERS`, which is known to be a subclass of `MANUFACTURERS`. Given these definitions of the class frames `PROCESSORS` and `JAPANESE_PROCESSORS`, some FRSs can automatically infer that `PROCESSORS` subsumes `JAPANESE_PROCESSORS`. These FRSs will place `JAPANESE_PROCESSORS` below `PROCESSORS` in the inheritance hierarchy, thereby *classifying* the concept. FRSs that compute subsumption can automatically place new concepts that users define into the correct position in the generalization hierarchy.

Section 4.2 discussed two ways of using classification to perform inference. The other important use of classification is in a variety of knowledge-base housekeeping operations such as detecting redundancies (different concepts that have equivalent definitions), inconsistencies (concepts whose definitions preclude them from having any instances), and vacuous concepts (concepts of which all instances would be members).

See [37] for an early introduction to classification, and [38] for a summary of experiences with an FRS that classifies. See [39] for a discussion of an algorithm for classification, and [14] for an analysis of the computational tractability of subsumption.

6 The Symbol Level

The symbol level of an information system includes algorithms and data structures that are generally invisible to the users of these systems. But the capabilities defined at the symbol level have a dramatic

effect on the speed of the system, the volume of information that the system can manage, the degree to which multiple users can concurrently share information, and whether or not information persists across system crashes.

6.1 Size and Speed

Many aspects of DB research have been driven by the need to provide users with high-speed access to very large (gigabytes to terabytes) bases of information. Modern RDBMSs provide this capability using a number of different techniques such as: special data structures for arranging data on disk, associated access algorithms, and query-optimization methods for efficiently evaluating user queries over huge amounts of information. In addition, RDBMSs allow the data base designer to specify certain attributes of the physical organization of a data base as distinct from its logical organization in order to enhance performance (such as the creation of indices). Put another way, data bases separate symbol-level design from system-engineering level design.

AI researchers have generally not constructed KBs larger than approximately several megabytes, and most KBs are considerably smaller — on the order of hundreds of frames. Therefore virtually all FRSs simply load the entire KB into virtual memory at the start of a session. This approach is not tractable for very large information bases. Further, FRSs do not provide KB designers with any influence on the physical organization of a KB — for example they have no notion of an index.

Early exceptions to this rule were the UNIT Package and RLL, both of which provided demand paging of frames [40, p69][41]. These techniques were developed to allow KB size to exceed virtual-memory size on machines of the 1970s that had limited address spaces. Frames were read into virtual memory when first accessed, and are saved back to disk on a least-recently used basis during LISP garbage collections. OZONE used similar techniques, but its slot spaces were used to distinguish slots that were read on demand from slots that were always memory resident (such as slots containing parent-link information) [42].

More recently, Ballou et al [43] have extended the storage capabilities of the PROTEUS FRS by coupling it to the ORION object-oriented data base. Every PROTEUS frame is stored on secondary storage as an ORION object, therefore the entire KB need not fit in virtual memory. However, they have not published a detailed account of the system architecture. In addition, Abarbanel et al [44] coupled the KEE FRS to an RDBMS. KEEconnection increases the storage capacity of KEE, and to allows RDBMS data to be imported into an AI environment. KEE instance frames only, and not class frames, can be stored in the RDBMS. The intelligent data base interface (IDI) developed by McKay et al. uses an approach similar to that of KEEconnection, but more sophisticated in its caching of retrieved relations [45]. Unfortunately, the impact of these three projects is quite limited because the authors either do not discuss the architecture of the system in sufficient detail to understand or replicate it, and because none of these publications discuss the performance of the resulting system in enough detail for the reader to determine what price is paid for using the system — that is, to determine whether high-speed access has actually been provided. For that matter, only McKay et al ask even more interesting questions about the benefits of storage system features such as caching.

6.2 Persistence and Reliability

Users generally require that their information persist indefinitely, despite events such as the termination of a program that accesses the information and the powering-down of the host computer system.

Because the DBs managed by a RDBMS reside on disk storage, and because DB updates are saved to disk at the end of every transaction, users of RDBMSs can take persistence for granted (that is, persistence across executions — persistence across corruptive system crashes is sometimes but not always obtainable). Not so for users of FRSs: because KBs exist in virtual memory, when the FRS process terminates or the host computer is powered down, the KB itself is destroyed. It is of course possible for the user to periodically save the entire KB to disk, but this operation can take on the order of several minutes for a very large KB.

As information bases grow in size they are more likely to become corrupted by hardware or software errors such as disk-head crashes and operating-system crashes. DB researchers have developed methods for detecting DB corruption, for lessening its effects by replicating data on multiple sites, and for automating recovery from crashes through the use of transaction logging and backups.

Conversely, most FRSs do not provide persistent or reliable KB storage services that have reasonable performance. Therefore, a KB that becomes corrupted in virtual memory is lost unless updates to it have been saved to a disk file; if this file is corrupted it cannot be recovered unless it has been backed-up to another medium in its entirety. For most FRSs, the time required to save updates to a KB is proportional to the size of the entire KB, since the only way to save updates is to write the entire KB to secondary storage. This approach becomes unacceptable for large KBs. Recent improvements on this model include the work of Mays et al [46], who have devised a persistent version store for the K-REP FRS in which updates are transmitted to the Statice OODBMS in time proportional to the number of updates. And although the authors do not state this explicitly, the same property presumably hold of the hybrid PROTEUS/ORION system [43].

6.3 Concurrency and Sharability

Large information bases often contain information that has been contributed by a large number of users, which information may be of value throughout a large community (a corporation or a scientific community). Therefore it is likely that multiple users will wish to access such information bases simultaneously. DB researchers have developed methods such as locking, concurrency protocols, atomic updates, data replication, and network interfaces to provide RDBMSs with these capabilities.

In contrast, most FRSs are single-user systems that allow only one person at a time to access a KB. Multiple users can of course make private copies of the KB for simultaneous access, but FRSs provide no mechanisms for automatically integrating modifications that users may make to the replicated copies of the KB.

Exceptions to this model include K-REP, PROTEUS/ORION, and CYCL [47]. In the CYCL system, multiple users make individual copies of a shared KB that is stored on a central server; updates to the KB are transmitted back to the central server, where the master copy of the KB is updated on a FIFO basis. Updates that violate KB consistency constraints at the server are undone, and notification messages are transmitted back to the originating user. Multiple users of K-REP can each load a copy of a shared KB into an individual workspace. Modifications to a workspace are integrated back into the shared KB by an explicit updated operation, at which time a number of consistency criteria are evaluated with respect to these two different copies of the KB. The updated to the shared KB are stored as a new version of the KB, and special data structures are provided to maintain past versions of the KB.

7 Summary

This paper has examined the differences between traditional frame knowledge representation systems and traditional relational data base management systems by comparing the capabilities of these systems at three different levels. At the symbol level, RDBMSs provide high-speed access to massive volumes of information. Multiple users can access this information concurrently, and the information persists across system crashes. Although a few FRS researchers have begun to explore these issues, these capabilities are not present in any of the most commonly used FRSs. Nor have investigations of these capabilities been particularly systematic: some researchers have not published descriptions of their work in these areas, and most researchers who have published have not assessed the costs and benefits of the many architectural variations that are possible for these symbol-level capabilities.

At the system-engineering level, we contrasted the fundamental data primitives of the RDBMS and the FRS: the relation and the frame. Frames are better suited to representing complex datatypes, and typically store a good deal of meta information for each slot. In addition, FRSs facilitate the design and evolution of complex information-base schemas through inheritance and run-time schema alteration. The two types of systems are typically accessed from different application programming languages, but they employ similar approaches to computed data: access-oriented programming and triggers. RDBMSs have superior support for information security and for standardized data base query languages. And although the systems have fairly similar abilities to express restrictions that help ensure information correctness, some FRSs treat these restrictions as definitions that can be used for concept classification and instance recognition. Explicit guidelines exist for the design of RDBMS schemas, whereas no comparable guidelines exist for FRSs.

At the knowledge level we saw that FRSs are more expressive than are RDBMSs — FRSs can represent generalizations as well as particulars, they can represent partial information, and they do not make a closed-world assumption. FRSs also have up to three inference mechanisms: inheritance, production rules, and classification, as well as reason-maintenance systems, none of which RDBMSs possess.

It should be clear that FRSs and RDBMSs both have advantages and disadvantages with respect to the other. FRSs excel at knowledge-level operations, whereas RDBMSs excel at the symbol-level. Neither system has a clear edge at the system-engineering level. Assuming that these technical differences are a significant factor behind the huge disparity in market share between the two types of systems, I recommend that KR researchers integrate into FRSs most of the RDBMS capabilities that they currently lack.

At the symbol level, FRSs require reliable, high-speed read and write access to large persistent knowledge bases. FRSs also greatly need the symbol-level capability of shared access by multiple users. In my view these are the most important capabilities that FRSs currently lack. Many modern applications require access to large volumes of information, which almost by definition must be developed by multiple users since their creation is likely to be beyond the capabilities of a single individual. In addition, the large cost of developing such large KBs encourages their sharing and reuse, which is greatly facilitated by multi-user access capabilities. Put another way, FRSs should be able to participate in modern, distributed computing environments.

At the systems engineering level, I recommend including runtime flexibility in those FRSs that lack it, and adding security features modeled after those of data bases. I also suggest research into constraints on FRS schema organization that are analogous to the data base normal forms. FRSs will also benefit from the declarative query languages that are under development within the knowledge sharing effort. That users of FRSs must program in Lisp is surely restricting the use of these systems. Among the

possible remedies are to construct C-language interfaces to FRSs (e.g., through Lisp foreign-function interfaces or through a network-based query language) or to implement an FRS solely in C, as has been done for Classic.

Acknowledgments

This paper benefited from comments from Thierry Barsalou, Tim Clark, Eric Mays, Marianne Winslett, and from an anonymous reviewer. This work was supported by the National Library of Medicine, where the author was a postdoctoral fellow, by SRI International, and by grant R29-LM05413-01A1 from the National Library of Medicine.

References

[1] R. Fikes and T. Kehler, "The role of frame-based representation in reasoning," *Communications of the Association for Computing Machinery*, vol. 28, no. 9, pp. 904–920, 1985.

[2] T. Finin, "Understanding frame languages," *AI Expert*, pp. 44–50, November 1986.

[3] P. Karp, "The design space of frame knowledge representation systems," Tech. Rep. 520, SRI International Artificial Intelligence Center, 1992.

[4] Y. Freundlich, "Knowledge bases and databases: Converging technologies, diverging interests," *IEEE Computer*, vol. 23, pp. 51–57, November 1990.

[5] M. Brodie, "Future intelligent information systems: AI and database technologies working together," in *Readings in Artificial Intelligence and Databases* (J. Mylopoulos and M. Brodie, eds.), pp. 623–642, Morgan Kaufmann Publishers, 1988.

[6] M. Brodie and F. Manola, "Database management: A survey," in *Readings in Artificial Intelligence and Databases* (J. Mylopoulos and M. Brodie, eds.), pp. 10–34, Morgann Kaufman, 1988.

[7] R. Brachman, "The basics of knowledge representation and reasoning," *AT&T Technical Journal*, vol. 67, no. 1, pp. 7–24, 1988.

[8] M. Brodie and J. Mylopoulos, *On Knowledge Base Management Systems.* Springer-Verlag, 1986.

[9] R. Brachman and H. Levesque, "The knowledge level of a KBMS," in *On Knowledge Base Management Systems*, pp. 9–12, Springer-Verlag, 1986.

[10] A. Newell, "The knowledge level," *Artificial Intelligence*, vol. 18, no. 1, pp. 87–127, 1982.

[11] R. MacGregor and M. Burstein, "Using a description classifier to enhance knowledge representation," *IEEE Expert*, vol. 6, pp. 41–46, June 1991.

[12] T. Mitchell, J. Allen, P. Chalasani, J. Cheng, E. Etzioni, M. Ringuette, and J. Schlimmer, "Theo: A framework for self-improving systems," in *Architectures for Intelligence*, Erlbaum, 1989.

[13] R. Reiter, "Towards a logical reconstruction of relational database theory," in *Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pp. 191–233, Springer-Verlag, 1983.

[14] H. Levesque and R. Brachman, "Expressiveness and tractability in knowledge representation and reasoning," *Computational Intelligence*, vol. 3, no. 2, pp. 78–93, 1987.

[15] N. Nilsson, *Principles of Artificial Intelligence.* Los Altos, CA: Morgan Kaufmann, 1980.

[16] P. Hayes, "The logic of frames," in *Frame conceptions and text understanding* (D. Metzing, ed.), pp. 46–61, Berlin, West Germany: Walter de Gruyter and Co., 1979.

[17] G. Wiederhold, "Knowledge versus data," in *On Knowledge Base Management Systems*, pp. 77–82, Springer-Verlag, 1986.

[18] D. Russinoff, "Proteus: A frame-based nonmonotonic inference system," Tech. Rep. ACA-AI-302-87, MCC, 1987.

[19] C. J. Petrie, D. Russinoff, D. Steiner, and N. Ballou, "Proteus 2: System description," Tech. Rep. AI-136-87, MCC, May 1987.

[20] R. Filman, "Reasoning with worlds and truth maintenance in a knowledge-based system shell," *Communications of the Association for Computing Machinery*, vol. 31, April 1988.

[21] M. Stonebraker, "Inclusion of new types in relational data base systems," in *Proceedings of IEEE/Data Engineering*, pp. 262–269, IEEE, 1986.

[22] T. Kehler and G. Clemenson, "KEE the knowledge engineering environment for industry," *Systems And Software*, vol. 3, pp. 212–224, January 1984.

[23] D. Maier, *The Theory of Relational Databases*. Rockville, MD: Computer Science Press, 1983.

[24] IntelliCorp, *KEE User's Guide*, 1985.

[25] R. Brachman and H. Levesque, "Knowledge level interfaces to information systems," in *On Knowledge Base Management Systems*, pp. 13–34, Springer-Verlag, 1986.

[26] ISX Corporation, *LOOM Users Guide, version 1.4*, August 1991.

[27] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout, "Enabling technology for knowledge sharing," *AI Magazine*, vol. 12, no. 3, pp. 36–56, 1991.

[28] E. Weixelbaum, "C-classic reference manual," Tech. Rep. 59620-910731-017M, AT&T Bell Laboratories, 1991.

[29] M. Stefik, D. Bobrow, and K. Kahn, "Integrating access-oriented programming into a multiparadigm environment," *IEEE Software*, vol. 3, pp. 10–18, January 1986.

[30] E. Nyberg, "The FrameKit user's guide, version 2.0." Unpublished system manual, Carnegie Mellon University, 1988.

[31] M. Fox, J. Wright, and D. Adam, "Experiences with SRL: An analysis of a frame-based knowledge representation," in *Expert Database Systems*, Benjamin/Cummings, 1985.

[32] K. Eswaran, "Specifications, implementations, and interactions of a trigger subsystem in an integrated data base system," Tech. Rep. RJ1820, IBM, 1976.

[33] U. Dayal et al., "The HiPAC project: Combining active databases and timing constraints," *SIGMOD Record*, vol. 17, March 1988.

[34] U. Dayal, "Active database systems," in *Proc. Third International Conference on Data and Knowledge Bases*, June 1988.

[35] M. Stonebraker, "The design of the Postgres storage system," in *Readings in Database Systems* (M. Stonebraker, ed.), pp. 556–566, Morgan Kaufmann Publishers, 1988.

[36] A. Kotz, K. Dittrich, and J. Mulle, "Supporting semantic rules by a generalized event/trigger mechanism," in *International Conference on Extending Database Technology*, March 1988.

[37] R. Brachman, "What's in a concept: structural foundations for semantic networks," *Int. J. Man–Machine Studies*, vol. 9, pp. 127–152, 1977.

[38] J. Schmolze and W. Mark, "The NIKL experience," *Computational Intelligence*, 1991. In press; see also Tufts Unviersity Department of Computer Science technical report 90-6.

[39] J. Schmolze and T. Lipkis, "Classification in the KL-ONE knowledge representation system," in *Proceedings of the 1983 International Joint Conference on Artificial Intelligence*, (Los Altos, CA), Morgan Kaufmann Publishers, August 1983.

[40] R. Smith, P. Friedland, and M. Stefik, "Unit Package user's guide," Tech. Rep. HPP-80-28, Stanford University Knowledge Systems Laboratory, December 1980.

[41] D. Smith, "CORLL manual: A storage and file management system for knowledge bases," Tech. Rep. HPP-80-8, Stanford University Heuristic Programming Project, Stanford, CA, 1980.

[42] C. Lane, "The Ozone reference manual," Tech. Rep. KSL-86-40, Stanford University Knowledge Systems Laboratory, July 1986.

[43] N. Ballou, H. Chou, J. Garza, W. Kim, C. Petrie, D. Russinoff, D. Steiner, and D. Woelk, "Coupling an expert system shell with an object-oriented database system," *Journal of Object-Oriented Programming*, pp. 12–21, June/July 1988.

[44] R. Abarbanel and M. Williams, "A relational representation for knowlege bases," tech. rep., IntelliCorp, 1986.

[45] D. McKay, T. Finin, and A. O'Hare, "The intelligent database interface: Integrating AI and database systems," in *Proceedings of the 1990 National Conference on Artificial Intelligence*, pp. 677–684, Morgan Kaufmann Publishers, 1990.

[46] E. Mays, S. Lanka, B. Dionne, and R. Weida, "A persistent store for large shared knowledge bases," *IEEE Trans. on Knowledge and Data Eng.*, vol. 3, no. 1, pp. 33–41, 1991.

[47] D. Lenat and R. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project.* Addison-Wesley, 1990.