SRI International

A Storage System for Scalable Knowledge Representation

Technical Note No. 547

August 23, 1994

By: Peter D. Karp, Sr. Computer Scientist
Suzanne M. Paley, Computer Scientist
Artificial Intelligence Center
Ira Greenberg, Computer Scientist
Computer Science Laboratory
Computing and Engineering Sciences Division

To appear in CIKM-94 (Conference on Information and Knowledge Management), Gaithersburg, Maryland, 1994

This work was supported by ARPA Contract No. F30602-92-C-0115, and by Grant No. R29-LM-05413-01A1 from the National Institutes of Health.

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

A Storage System for Scalable Knowledge Representation

Peter D. Karp, Suzanne M. Paley, Ira Greenberg*
Artificial Intelligence Center and *Computer Science Laboratory
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
voice: 415-859-6375
fax: 415-859-3735
pkarp@ai.sri.com

Abstract

Twenty years of AI research in knowledge representation has produced frame knowledge representation systems (FRSs) that incorporate a number of important advances. However, FRSs lack two important capabilities that prevent them from scaling up to realistic applications: they cannot provide high-speed access to large knowledge bases (KBs), and they do not support shared, concurrent KB access by multiple users. Our research investigates the hypothesis that one can employ an existing database management system (DBMS) as a storage subsystem for an FRS, to provide high-speed access to large, shared KBs. We describe the design and implementation of a general storage system that incrementally loads referenced frames from a DBMS, and saves modified frames back to the DBMS, for two different FRSs: LOOM and THEO. We also present experimental results showing that the performance of our prototype storage subsystem exceeds that of flat files for simulated applications that reference or update up to one third of the frames from a large LOOM KB.

To appear in CIKM-94 (Conference on Information and Knowledge Management), Gaithersburg MD, 1994.

1 Introduction

Twenty years of AI research in knowledge representation has produced frame knowledge representation systems (FRSs) that incorporate a number of important advances [6, 4]. FRSs provide inference capabilities such as production rules and classification to derive the deductive consequences of explicit information. Defeasible inheritance allows regularities (defaults) to be encoded and overridden for many objects with minimal effort. And run-time schema alteration capabilities support the evolution of complex knowledge bases (KBs).

However, FRSs lack two important capabilities that prevent them from scaling up to realistic applications: they cannot provide high-speed access to large KBs, and they do not support shared, concurrent KB access by multiple users. All existing FRSs process their KBs in data structures that exist entirely in virtual memory, forcing users to read the whole KB into memory from disk before its use. To provide persistence, KBs are written to disk files in their entirety. Saving or loading a KB can therefore become an expensive operation, taking time proportional to the size of the KB. An effective cap is placed on the size of a KB by the amount of time that users are willing to wait for save and load operations, with an absolute cap based on the size of virtual memory.

A more favorable arrangement would be one in which load time and memory usage are proportional to the number of frames *referenced*, and save time is proportional to the number of *frames updated*. This is the behavior supported by conventional database systems, which also offer other important storage management facilities, including transactions, error recovery, and concurrent access. We combine the information management capabilities of frame representation systems with the storage management capabilities of conventional database systems to form a single intelligent, persistent, and scalable information management system.

Our research investigates the hypothesis that we can employ an existing database management system (DBMS) as a storage subsystem for an FRS, to provide high-speed access to large, shared KBs. This paper discusses the design requirements that we identified for this storage subsystem, presents alternative storage-subsystem architectures that satisfy those requirements, and gives performance measurements from our prototype implementation.

Our prototype storage subsystem utilizes a commercial relational DBMS (RDBMS). To gain a fuller understanding of the issues involved in developing a storage system for an

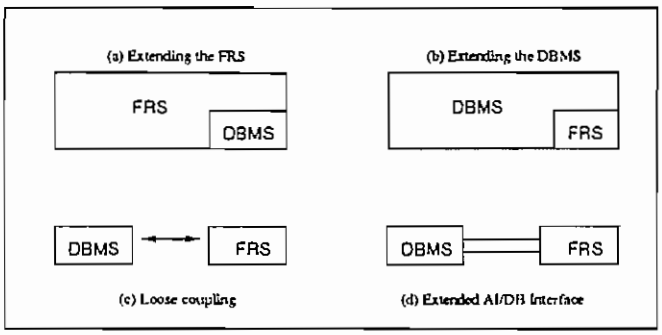

Figure 1: A characterization by McKay et al. of alternative strategies for coupling an AI system such as an FRS with a database system.

FRS, we have integrated this storage subsystem with two FRSs: THEO [11] and LOOM [5, 8]. LOOM is in the KL-ONE family of FRSs, whereas THEO is in the RLL family; differences in the philosophies and implementations of THEO and LOOM affect their exact requirements for a storage subsystem.

2 Storage Subsystem Architecture

McKay et al. describe four broad alternative architectures for coupling AI systems (FRSs) with database systems (see Figure 1) [10]. Our primary goal is to provide a storage system for an FRS with minimal disruption to the end user of the FRS. We therefore chose strategy (a) in Figure 1, namely to submerge a DBMS within an FRS such that the presence of the DBMS is invisible to the end user. For example, the user need not have any knowledge of the DBMS schema, nor must he establish a mapping between the schemas in the DBMS and the FRS. The Intelligent Database Interface (IDI) system developed by McKay et al. uses strategy (d) because their main objective is to import information from an existing DBMS into a knowledge representation system.

Once a general architecture is selected, several more choices must be made, such as, what type of DBMS is best suited to the role of a frame storage system? Because the answer to this question is not apparent, we are experimenting with a commercial relational DBMS, and an extensible storage management system called EXODUS from the University of Wisconsin [3]. This paper presents performance results for the relational DBMS.

Another decision concerns the manner in which FRS information is organized in the DBMS. One of our goals is that the user should not have to design a DBMS schema for every new KB. Instead, we as designers of the storage system must create a generic DBMS schema that accommodates all potential FRS information. In fact, more than one such generic schema exists, and we plan to evaluate the performance of several schemas empirically.

Another series of choices concerns the granularity at which information is transferred between the DBMS and the FRS. Our goals are for KB loading to take time proportional to the amount of information the application actually references; KB saving should take time proportional to the number of frames updated in the KB. The simplest mechanism that satisfies these constraints is to transfer a single frame from the DBMS to the FRS when the user application references a frame that is not currently in virtual memory. This demand-loading approach is analogous to the use of page faulting in operating systems. Our current implementation uses this approach, but we are also studying alternatives such as transferring only a piece of a referenced frame (e.g., a single slot), or transferring a cluster of related frames (e.g., a referenced frame plus all frames that it references).

In our current implementation, all modified frames are transferred from the FRS to the DBMS when the user performs a KB-save operation.

3 Storage Subsystem Implementation

The storage subsystem transmits ASCII encodings of LOOM and THEO frames between the DBMS, and the FRS. We first provide an overview of the frame structures that LOOM and THEO employ. We then discuss the architecture of the storage subsystem, and modifications we made to LOOM and THEO to interface them to the storage subsystem.

3.1 LOOM Structures and Operation

A LOOM KB contains three types of frames: concepts, instances, and relations (we have simplified the description of LOOM for expository purposes). A concept consists of a name and a definition. The concept definition is a set of necessary and sufficient conditions that an instance must meet in order to be an instance of the concept. The definition is a list of zero or more super-concepts, constraints on slot values, predicates, and other types of constraints and characteristics. Given this information, the LOOM classifier arranges all concepts into a subsumption (generalization) hierarchy.

A LOOM relation (not to be confused with the usual database definition of a relation as a table) is a KB-wide definition of the properties of a slot. We can define a domain and a range for a relation. The domain indicates all concepts whose instances can have values for the relation, and the range limits the types of objects that can serve as values.

Instances have one or more parent concepts and some set of slot (attribute) values. Based on these characteristics, the LOOM classifier can infer the concepts to which the instance belongs. For example, if Person is a primitive concept that is in the domain of the relations Sex and Age, and if a Female-person is a Person with Sex=Female, and a Girl is a Person with Sex=Female and Age<18, then LOOM will correctly infer that Girl belongs below Female-person in the concept hierarchy. If Sally is an instance of Person with Sex=Female and Age=17, then LOOM will correctly infer that Sally is an instance of Girl. If Sally has a birthday and her age changes to 18, then LOOM will revise that classification automatically.

Most commonly, LOOM performs two types of inferences: in backward-chaining mode values are computed only when requested, and in forward-chaining mode the consequences of an assertion are computed as soon as the assertion is made. All our tests and experiments have used LOOM's backward-chaining mode. We believe our system would also work with the forward-chaining mode. However, in order to make the required inferences, creation or modification of a single frame could trigger a large number of frame faults by LOOM's classifier, which could hurt performance. We have not attempted to support LOOM's production-rule inference.

3.2 THEO Structures and Operation

Because THEO is also an FRS, it shares many characteristics with LOOM. THEO frames are also arranged in a generalization hierarchy, and THEO frames consist of slots that contain values. However, THEO classes do not have associated definitions, and THEO does not compute the classification operation. Given the basic structural similarity of LOOM and THEO, it is natural to develop a storage system that can serve both systems.

For simplicity the remainder of the paper usually mentions LOOM only. All statements we make about the interaction of LOOM with our storage system also apply to THEO except where we state otherwise.

3.3 Relational Schema

The relational DBMS schema we employ to store LOOM KBs consists of five relational tables. An example is shown in Figure 2.

The *Frames* table contains frame definitions. A frame definition is a string of text that provides LOOM with all the information necessary to create the frame. We place concept and instance definitions together in the same table, because there are occasions when a frame is referenced without its type being known — the type field then identifies the frame as a concept or instance. Most definitions will be relatively short, but some may be quite long. For this reason, a sequence number is included, in case a definition exceeds the DBMS maximum column size and has to be split into multiple tuples. We record the number of parents of each frame to enable the storage subsystem to perform certain optimizations. A KB identifier is included in each table, to enable multiple KBs to be stored in one DBMS. The *KB Mapping* table associates a KB name with its unique identifier.

The tables *Supers* and *Instance Classes* enable reconstruction of the concept and instance hierarchy outside of LOOM. The former lists the super–sub relationships between concepts; the latter documents the relationship between instances and their parent concepts. Separate indices are built to retrieve the subconcepts of a concept, the superconcepts of a concept, the instances of a concept, and the parent concepts of an instance. This information is necessary for two reasons. First, in order for a concept or instance to be defined in LOOM, all parent concepts must already be loaded, or LOOM will not be able to classify the new frame. Thus, we must be able to determine the concepts from which a given concept or instance inherits. Second, the definition does not contain information about subconcepts or instances of a concept, so we must provide that information to LOOM directly, outside the normal channels.

3.4 Frame Faulting

A *frame fault* occurs when an application (or LOOM itself) references a frame F that is not in virtual memory. Examples of frame references include retrieving or altering slot values of F, and requesting a list of the parents of F. When faulting a frame into memory, we retrieve its definition from the DBMS by issuing one or more SQL queries.[1] This approach allows multiple users to access and to update the same KB from the RDBMS server in a distributed (but uncoordinated) fashion. Our future work will investigate methods of controlling multiple updates to a shared KB.

We then call standard LOOM functions to add the frame to its LOOM KB. This process is complicated by the fact that most frames are related (connected) to other frames in the KB. For example, a concept is related to its superconcepts, subconcepts, and instances. An instance will contain references to its parent concepts. In addition, an instance may contain references to other instances serving as fillers of the instance's slots. LOOM normally expects all of these other frames to be present in memory. When faulting a frame into memory, we also give LOOM just enough of the context required to process the faulted frame.

The process of faulting F into memory involves three steps: processing the parents of F, informing LOOM of the definition of F, and processing connections from F to frames other than its parents.

Connections to parent frames We treat connections to parent frames differently than connections to subconcepts, instances, and slot-value references. The RDBMS tables *Supers* and *Instance Classes* record the direct parents of every frame (as inferred by LOOM by classification before the KB was last saved). When processing a fault to frame F, we first generate faults to every direct parent of F that is not currently in virtual memory (faults to the parents of these parents may then be generated recursively). Therefore, all parents of F are loaded before F is defined.

Connections to other frames LOOM implements all frame references as LISP pointers to the actual LOOM data structures for the frames in question. In the RDBMS definition of the frame, these references are symbolic frame names. Normally, when LOOM processes a frame definition it internally converts the names to pointers to the actual objects. However, consider the situation where a slot of F references a frame G, and G has not yet been loaded from the RDBMS. There would be no LOOM data structure to which F could point. Although we could now fault in G, we wish to avoid loading a frame just because we need to point to it, because for some KBs this strategy could recursively fault in the entire KB. Instead, we create a stub for G — a dummy object with a name but containing no information — to serve as a place-holder for G. LOOM can store and pass around pointers to a stub just as it would a pointer to any frame. A pointer to G is then manually inserted into the appropriate slot of F. A future attempt to retrieve information (other than the name) from, or to write information to a stub, causes a trap to the storage subsystem, and the actual frame is then faulted in. LOOM inferencing is not affected by the presence of stubs, as any attempt to reason using a stub will result in a frame fault. This mechanism is analogous to the swizzling operation performed in object-oriented database management systems (OODBMSs), which convert an object ID into a pointer [7]. Object IDs are typically numbers; symbolic frame names in LOOM are analogous to object IDs.

This topic is one of the more significant differences between LOOM and THEO. In THEO, connections from one frame to another are implemented as frame names that are LISP symbols, rather than as pointers to a frame data structure (a CLOS object) as in LOOM. LISP symbols are of course implemented as pointers, but these pointers point to the LISP symbol table where all symbols are interned. The THEO frame definition is stored on the property list of the symbol. We can consider LISP symbol interning to be a stub mechanism of sorts, because entries in this symbol ta-

[1] We call on the IDI from Paramax to communicate with the RDBMS server from LISP using SQL queries that can be transported over a network. We are not employing the full power of the IDI; we utilize only the module of the IDI that formulates and unpacks SQL queries.

Frames

KB ID	Frame Name	Loom Defn	Kind	Seq#	# of Parents
1	Army	(•••)	class	0	2
1	Armed-Forces	(•••)	class	0	0
1	Ground-Unit	(•••)	class	0	0
1	5th-Brigade	(•••)	instance	0	1

Relations

KB ID	Rel Name	Rel Defn
1		(•••)

KB Mapping

KB Name	KB ID
Forces	1
Supplies	2

Supers

KB ID	Class Name	Super
1	Army	Armed-Forces
1	Army	Ground-Unit

Instance Classes

KB ID	Instance Name	Class Name
1	5th-Brigade	Army

Figure 2: The relational schema used to store LOOM KBs in an RDBMS, with sample data.

ble provide a place to which symbolic frame references can point. Therefore, no stub frames need to be created for the THEO storage subsystem.

Defining the frame The storage system retrieves the definition of F from the *Frames* table of the RDBMS, and invokes LOOM procedures to define F based on the retrieved definition string. If a stub definition already existed for F because of a connection from a previously faulted frame to F, the stub object is directly converted to a LOOM object (LOOM is written using CLOS, which allows this class-conversion operation). This approach maintains the validity of all previously existing pointers to the stub.

Modifications to LOOM We made several modifications to LOOM to implement demand loading of frames. (1) Normally, when LOOM converts an identifier to an object pointer, it checks a collection of hash tables to find the object. We changed LOOM so that if the hash table lookup fails, a frame fault is triggered in most cases. (2) No frame fault is triggered, however, if a flag is set to indicate that we are already in the process of fanlting in a frame. In that case, the implementation finds or creates a stub for the object, and returns a pointer to the stub. The frame-creation routines of LOOM have been altered to check for a stub for an object before creating a new object: if the stub already exists, it is directly converted to the new object. (3) Before processing any instance or query, LOOM normally performs a series of operations on all concepts that have been defined or modified since the last instance or query was processed, to ensure that the concept hierarchy is properly formed. This *sealing* process involves following all superconcept and subconcept links. However, in the context of the storage subsystem, following subconcept links would cause many additional concepts to be faulted into memory, even though they have not been referenced. We have altered LOOM to follow links only to subconcepts already in memory.

Modifications to THEO THEO was simpler to modify because stubs are not needed for THEO, and because THEO performs no sealing of class (concept) frames. The only change necessary was to the THEO frame lookup procedure: if THEO does not find that a referenced frame is defined on the expected property-list entry, a frame fault is triggered.

4 Experimental Methods

To evaluate the storage subsystem, we want to test how it performs on a variety of KBs of different types. Ideally, we would have a series of KBs, each differing from the others in a single aspect, so as to be able to pinpoint how different factors affect performance. Unfortunately, finding a set of real KBs that show such systematic differences is virtually impossible. For this reason, we have developed a parameterized random KB generator (RKBG) and concomitant tools, to generate KBs according to input specifications (but with meaningless data). By altering one parameter at a time, we can conduct controlled experiments with interpretable results.

Our random KB suite consists of three tools. The KB generator creates LOOM and THEO KBs with storage characteristics that the user defines. A simulated KB application generates accesses and updates to the frames of randomly generated KBs. Finally, the KB measurer examines real KBs to determine their particular storage characteristics, so that we have a realistic set of parameters to feed into the RKBG. All three tools have been implemented and tested. The KBs used in our timing experiments were all generated by the RKBG.

4.1 The Random KB Generator

Many attributes of a KB can affect its storage characteristics. Input parameters to the RKBG allow the user to determine many of these attributes. We wanted to keep the generator simple, yet flexible enough to generate nonhomogeneous, realistic-looking KBs. For example, in our implementation, different slots can take different datatypes for their fillers; the constraints generated for a slot will depend on its datatype, just as would be observed in real KBs. We wanted to provide enough input parameters to enable users to create a rich and varied assortment of KBs, while concentrating primarily on parameters we believed to be most relevant to the storage subsystem and its interactions with LOOM. For example, the average string length affects the size of a frame, and therefore the amount of data that needs to be retrieved during a frame fault, so we believed it was a valuable parameter to include, whereas the range of integer values used as slot fillers has little effect on storage

processing, so we fixed it arbitrarily.

The parameters to the RKBG include such properties as number of concepts and instances, average number of slots per frame and fillers per slot, proportions of various datatypes for slot-fillers, maximum depth of the concept hierarchy, etc. In these respects, we can make our random KBs very similar to real KBs, and the parameters for our base random KB are in fact based on the measurements of a real LOOM KB. The shape of the concept hierarchy and the distribution of instances among concepts is entirely random, however, so these aspects of our generated KBs do not necessarily resemble actual KBs.

4.2 The Experiments

The goal of the experiments discussed herein was to measure storage system performance as a function of knowledge base size. We therefore chose to keep all RKBG parameters constant except for the number of iustances in the KB. Each KB had 100 concepts, all primitive, with just one super each. Instances averaged 5 slots apiece, with an average of 2 fillers per slot. Half the slots were filled by integers, with the other half filled by symbols. These parameters were chosen because they approximate the characteristics of the transportation-planning KB that is driving our work with LOOM[12], as measured by our KB measurer. The same random seed was used to create every KB, so the concept hierarchy remained the same, regardless of the number of instances. Knowledge bases were generated with 500, 1000, 2000, 4000 and 5000 instances. For comparison, the same set of KBs were generated and saved to native LOOM flat files, to native THEO flat files, and to the RDBMS (both LOOM and THEO versions). These four variations of four KBs form the basis for our experiments.

Experiments were run using LOOM 2.1, and the February 1993 version of THEO, running on Lucid Common Lisp 4.1.1. Both the FRS and the RDBMS server were running on the same workstation, a SPARCstation 10 model 41 with 64 MB of physical memory. LISP was restarted before every trial, to avoid caching effects, and a garbage collection was executed immediately before timing. Each trial was repeated three times, and the results averaged. Overall elapsed times were measured using the LISP time function. Measuring the time spent in LOOM, THEO, IDI, and the storage subsystem was done by monitoring key procedures using the CMU monitoring package. The CPU time spent in the RDBMS server process was measured using the UNIX ps utility to observe total CPU time before and after each experiment.

A few experiments were also run with the FRS and the RDBMS running on different machines connected by a network. These results are not shown, but they were quite similar to the results from running both on the same machine. In general, there was more variance in the results when running on two machines, but the best times in that configuration were roughly the same as the times on a single machine. On a single machine, our elapsed times tend to be very close to the sum of the CPU times for each component, indicating that system overhead (e.g. from virtual memory swapping) was not a major factor.

The first set of experiments measured the time required to reference some number of randomly chosen instances from KBs of different sizes. Each reference faults in at least one frame from the RDBMS (when the parent classes of an instance are not memory resident, they are also faulted in).

Selected results for LOOM and THEO are shown in Figure 3. Each of the dashed lines in these graphs shows the time required to reference N instances in KBs of different sizes. For example, the highest line in each graph shows the time required to reference 2000 instances from KBs containing a total of 2000, 4000, and 5000 instances. Figure 3(a) shows that for THEO, the time required to reference 500 instances from a KB containing 4000 total instances is about the same as the time required to load that KB in its entirety from the flat files.

A second set of measurements breaks down the total time spent processing frame faults into several components: the time spent in the RDBMS server, the IDI, our storage system, and the FRS (LOOM and THEO). Figure 4 plots these component times as a function of the number of instances referenced for a fixed KB of 5000 instances. Figure 4(b) shows how the total time for referencing N instances breaks down into time spent in LOOM, our storage subsystem (SSS), IDI, the RDBMS, and other processing (presumably I/O). Figure 4(a) shows an analogous breakdown for THEO.

The third experiment measured the time required to save updates to some number of randomly chosen instances from KBs of various sizes. To be consistent with traditional LOOM behavior, updates are not written as they occur. Rather, we wait until the user issues a command to save updates, and then all are written at once in a single transaction. We varied the number of frames updated between 10 and 1000. Selected results are shown in Figure 5. For comparison, we have included the time to save KBs of varying sizes to LOOM flat files (the time is constant for a given KB regardless of the number of frames updated in that KB). KB save times for THEO are similar, and thus are not shown.

5 Discussion

Our primary goals in performing these experiments are to answer several questions. Does the performance of our RDBMS-based storage subsystem meet the goal of linear time as a function of number of frames referenced and number of updates stored? If so, is its speed fast enough to make the storage system usable in practice? And how do the different components of the storage subsystem such as the RDBMS server contribute to its overall performance?

Figure 4 demonstrates that our architecture achieves the linearity goal: the time spent loading frames is a linear function of the number of frames referenced.

Figure 3 lets us evaluate the relative merits of loading frames from the RDBMS versus from flat files. The relative merit differs for LOOM versus for THEO because LOOM takes significantly longer to load an entire KB of N frames than does THEO. The difference is that LOOM is performing computations (classification) on the KB that THEO is not. Because the same amount of data is transferred for each FRS during incremental loading of N concepts from the same KB, the database costs are about the same. Therefore the ratio of database costs to total costs is higher for THEO than for LOOM. For THEO, loading N instances from the DBMS is 8 times slower than loading an entire KB of N instances from a flat file. But for LOOM, loading N instances from the DBMS is only 3 times slower than loading a KB of that size from a flat file. Therefore the performance of the RDBMS storage subsystem is on par with a flat file when a user references up to 12% of the frames in a KB in a given session for THEO; for LOOM the user can reference up to 30% of the frames for equivalent performance. We take this result to mean that even for THEO the performance of the storage subsystem is acceptable in practice given our assumption that as KB

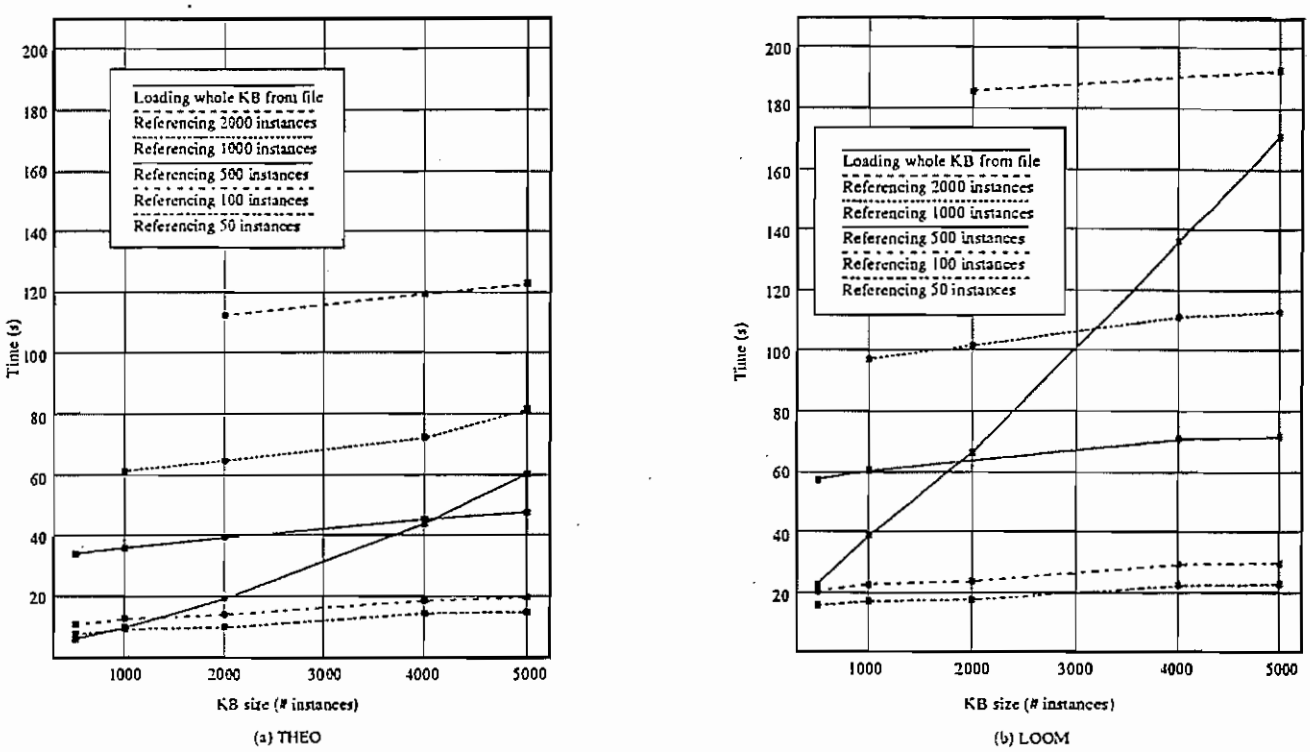

Figure 3: The solid line in each graph shows the time required to load entire KBs of varying sizes from flat files for THEO (a) and LOOM (b). The dashed lines show times required to fault in frames from the RDBMS due to references to instances by the application. Each dashed line shows the same number of instance references as a function of KB size. All times refer to total elapsed times. The vertical ordering of dashed lines in each graph and in its legend are the same.

size grows, users will reference only a fraction of its frames in a given session. Note that RDBMS loading also has a different response-time profile than does flat-file loading — flat-file loading requires a long wait at startup time, whereas demand loading hides loading waits across many operations.

Figure 3 shows that RDBMS frame loading time depends on KB size when a fixed number of instances are referenced. Because the parents of any referenced instances are faulted in along with the instances, a likely explanation is that the time to load classes depends on the size of the KB. When a class is faulted in, the names of all its instances must be retrieved from the database. Because all of our experimental KBs contain the same number of classes, the number of instances per class increases in proportion to KB size, requiring a greater amount of data to be retrieved per class for large KBs. We have no data yet on how the class:instance ratio depends on KB size for real KBs.

Figure 5(a) demonstrates that, as expected, our architecture achieves the goal of saving KB changes in time linear in the number of updates. Figure 5(b) shows that the time to save frames is not dependent on the size of the KB. Saving N updated frames to the RDBMS is roughly 5 times slower than saving an entire KB of N frames to a flat file. Therefore, our storage subsystem is faster than the flat file when less than 20% of the KB has been altered.

6 Related Work on FRS Storage Systems

KEEconnection couples the KEE FRS with a relational DBMS [1] and the IDI couples LOOM with a relational DBMS [10]. They are examples of architectures (c) and (d) in Figure 1, in which the DBMS and FRS are loosely coupled *peers*. The advantage of these architectures is to allow existing information from a database to be imported into an AI environment. The drawback is that this architecture does not transparently enhance the storage capabilities of LOOM as does our approach. Users of KEEconnection (and of the IDI) must define a mapping between a class frame and a table in the RDBMS; KEEconnection creates frame instances from analogously structured tuples stored in the RDBMS, and can store instance frames out to the DBMS. But note that only slot values in instance frames can be transferred to the database — class frames are not, so this information is not persistently stored using database techniques and cannot be accessed by multiple users. Our approach allows *all* information in a LOOM KB to be permanently stored in the DBMS.

Groups at IBM and at MCC have coupled FRSs to OODBMSs. Mays et al. coupled the K-REP system to the Statice OODBMS [9], and Ballou et al. coupled the PROTEUS FRS to the ORION OODBMS [2]. The IBM effort differs from our approach in that a KB is read from the OODBMS in its entirety when it is first referenced by a K-REP user, which we believe will be unacceptably slow for large KBs.

Unfortunately, none of these researchers have published experimental investigations of alternative implementation choices, as we are doing. Without systematic experiments it is impossible to evaluate the relative merits of the many possible alternative architectures.

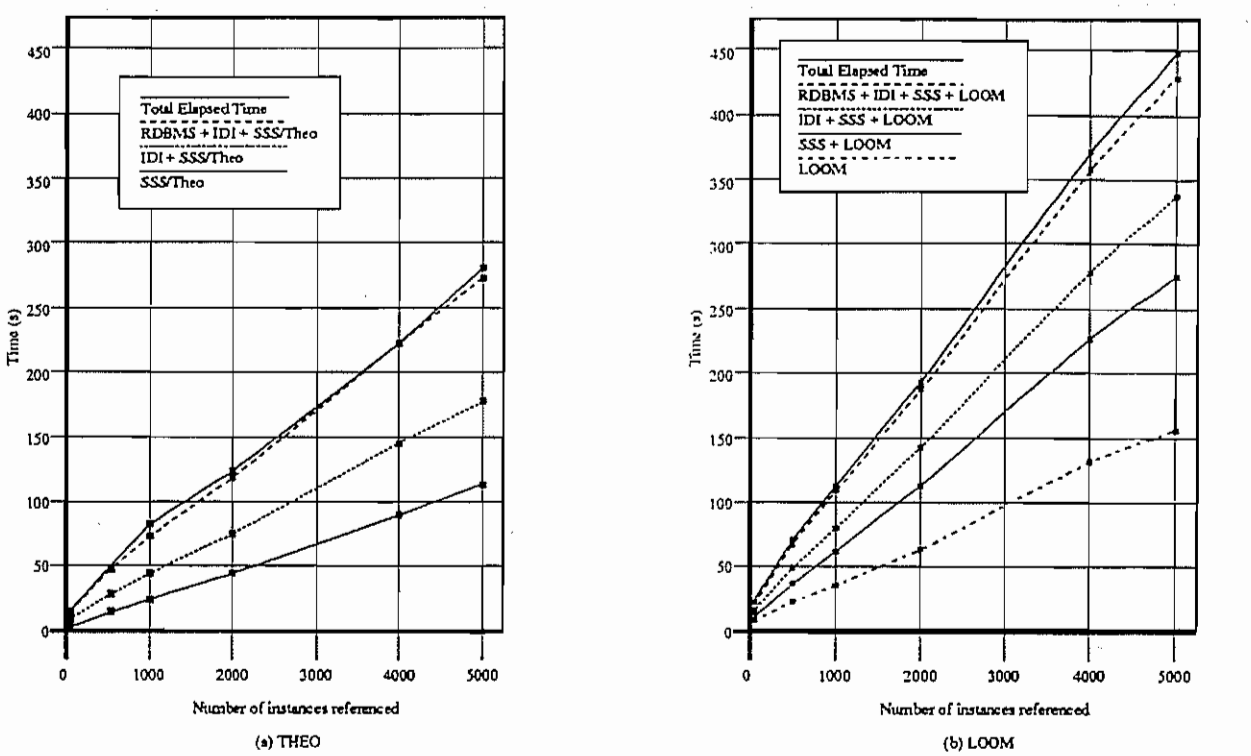

Figure 4: The total elapsed time for referencing and faulting N instances into memory from a KB of 5000 instances is separated into several components. (a) The vertical distances between lines represent time spent in THEO plus our storage subsystem (SSS), in the IDI, and in the RDBMS server. (b) The vertical distances between lines represent time spent in LOOM, in our storage subsystem (including stub creation), in the IDI, and in the RDBMS server. The solid line is an elapsed time, whereas all of the broken lines show CPU times.

7 Summary and Future Work

A FRS that performs demand loading of referenced frames, combined with incremental saving of updated frames, will scale to large KBs much more gracefully than an FRS that can only load or save frames in their entirety. We presented an architecture for an FRS storage subsystem that submerges a DBMS within the FRS in a manner that is transparent to the FRS user. Our experimental results with a prototype implementation show that this coupling performs well in practice, and that its performance is linear in the number of frames referenced or updated, as required.

Our future work will evaluate a number of variations on our current architecture, such as different RDBMS schemas, faulting multiple related frames into memory as a unit, and the use of other types of DBMSs, such as object-oriented DBMSs. Our experiments will involve several real KBs in addition to synthetic KBs. We are also investigating new paradigms of controlling multiuser access to shared KBs.

Acknowledgments

We are grateful to Bob MacGregor and other members of the LOOM group at ISI for discussions of LOOM internals, answers to questions, and prompt bug fixes. We are also grateful to Tom Mitchell for supplying THEO and for answering many questions. We thank Robin McEntire at Paramax for supplying and revising the IDI. This work was supported by ARPA Contract No. F30602-92-C-0115, and by grant R29-LM-05413-01A1 from the National Institutes of Health. The contents of this article are solely the responsibility of the authors and do not necessarily represent the official views of the Advanced Research Projects Agency or of the National Institutes of Health.

References

[1] R. Abarbanel and M. Williams. A relational representation for knowledge bases. Technical report, IntelliCorp, 1986.

[2] N. Ballou, H.T. Chou, J.F. 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*, pages 12–21, June/July 1988.

[3] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage Management for Objects in EXODUS. In Kim [7], pages 341–369.

[4] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the Association for Computing Machinery*, 28(9):904–920, 1985.

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

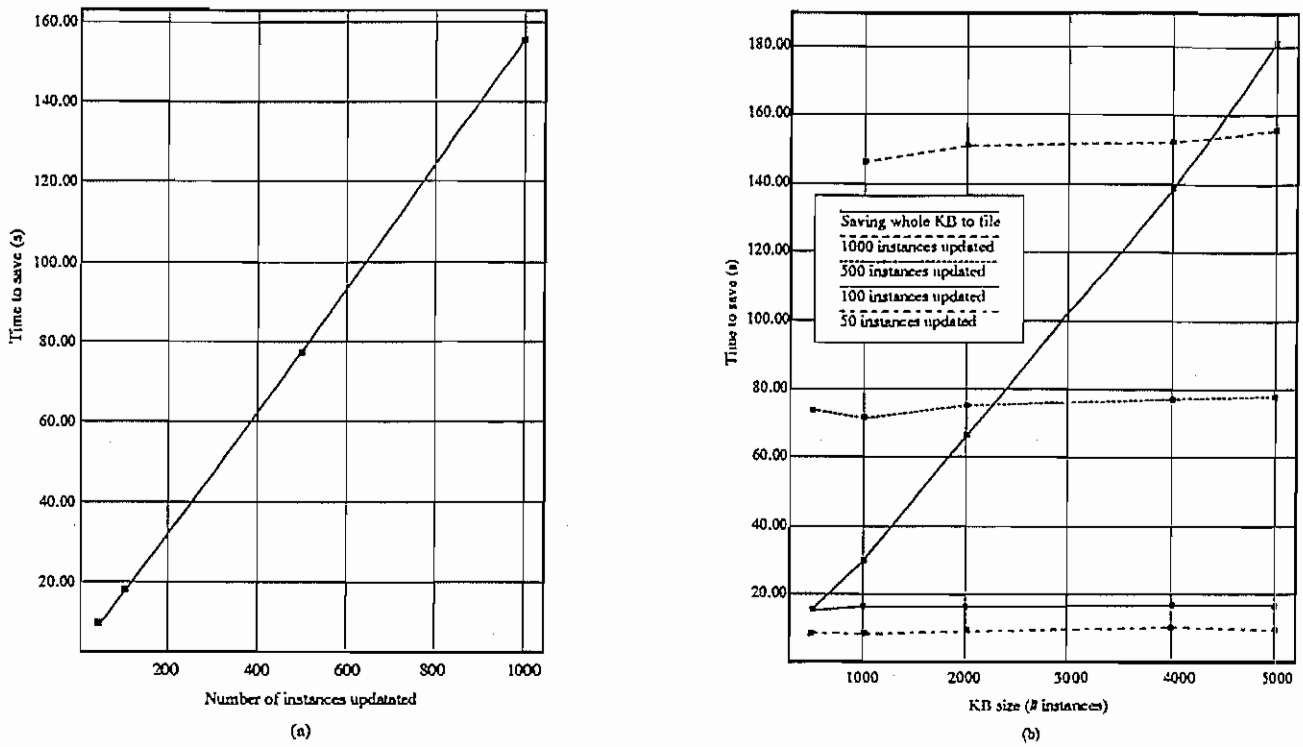

Figure 5: (a) Each data point represents the time to save a given number of instances for a fixed KB of 1000 instances. (b) The solid line shows the time required to save entire KBs of various sizes to LOOM flat files. The dashed lines show, for KBs of various sizes, the times required to store a given number of updated instances to the RDBMS. The time required to save 500 updated instances to the RDBMS is about the same as the time required to save an entire KB of 2300 frames to a flat file. All times are total elapsed times.

[6] P.D. Karp. The design space of frame knowledge representation systems. Technical Report 520, SRI International Artificial Intelligence Center, 1992.

[7] W. Kim and F.H. Lochovsky. *Object-oriented concepts, databases, and applications.* ACM Press, 1989.

[8] R. MacGregor and M.H. Burstein. Using a description classifier to enhance knowledge representation. *IEEE Expert*, 6(3):41–46, June 1991.

[9] E. Mays, S. Lanka, B. Dionne, and R. Weida. A persistent store for large shared knowledge bases. *IEEE Trans. on Knowledge and Data Eng.*, 3(1):33–41, 1991.

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

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

[12] D. E. Wilkins and R.V. Desimone. Applying an AI planner to military operations planning. In M. Fox and M. Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann Publishers, 1992.