Knowledge Representation in the Large

Peter D. Karp   Suzanne M. Paley
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Phone: 415-859-6375   Fax: 415-859-3735
pkarp@ai.sri.com paley@ai.sri.com

Abstract

Frame knowledge representation systems lack two important capabilities that prevent them from scaling up to large applications: they do not support fast access to large knowledge bases (KBs), nor do they provide concurrent multiuser access to shared KBs. We describe the design and implementation of a storage subsystem that submerges a database management system (DBMS) within a knowledge representation system. The storage subsystem incrementally loads referenced frames from the DBMS, and can save to the DBMS only those frames that have been updated in a given session. We present experimental results that show our approach to be an improvement over the use of flat files, and that evaluate several variations of our approach.

1 Introduction

The negligible impact of frame knowledge representation systems (FRSs) on the general practice of computing is an embarrassment to the field of artificial intelligence (AI). Knowledge representation (KR) researchers have investigated this style of information management for roughly 20 years. Although there is a rich history of theory and practice, a plethora of good ideas, and a large number of implemented systems (more than 50 by one count [4]), KR systems have not seen widespread use — either within the AI community, or within the broader computing world (such as the commercial sector).

This situation is a pity because FRSs are superior to conventional database-management systems (DBMSs) in a number of respects. They include inference capabilities based on production rules (which have been adopted in DBMSs of late), classification, and inheritance. Yet superior inference capabilities are not their only advantage, despite the focus of the KR community on inference. Some FRSs support run-time schema alteration, which facilitates the evolution of complex knowledge base (KB) schemas. Furthermore, whereas the database community discovered the conceptual benefits of the object-oriented data model only within the past 5 years, FRSs have used an object-oriented data model since their inception, and the FRS object model is richer than that used in object-oriented DBs (consider facets and concept-definition languages).

Our hypothesis is that inattention to engineering is one of the principal reasons for the failure of FRSs from a practical perspective. The architecture of FRSs does not scale up to support rapid storage and retrieval of large KBs, nor does it provide concurrent development of shared KBs by multiple users. Nor do FRSs support the distributed, networked architecture that is the foundation of modern computing. Without these capabilities, FRSs are doomed to remain applicable to building only small systems of at most a few thousand objects.

Our group is reengineering two FRSs — LOOM and THEO — to contain an underlying storage subsystem that provides fast, distributed, multiuser access to large KBs. The storage subsystem is implemented using a DBMS, but is invisible to the user. We therefore combine the advantageous knowledge-level capabilities of FRSs with the powerful symbol-level capabilities of DBMSs to obtain the best of both worlds.

Successful engineering relies on both design and experimentation. This paper presents our design of an FRS storage subsystem and the results of experiments with a number of variations of that design to assess their relative merits.[1] Our results show that our architecture is an advance over current systems in several respects. We also present a prefetching strategy that can further improve storage-subsystem performance.

2 Design of the Storage Subsystem

All existing FRSs process their KBs in data structures that exist entirely in 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 be 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

[1] A lack of experimentation within the KR community is another factor behind the limited success of these systems. Few publications provide us with any empirical understanding of how FRSs perform in practice, or of the practical tradeoffs among different FRS features.

size of virtual memory. In a more scalable arrangement, load time and memory usage would be proportional to the number of frames *referenced;* save time would be proportional to the number of frames *updated.*

Our storage subsystem submerges a DBMS within an FRS. The FRS retrieves frames incrementally, on demand, from the DBMS. The FRS tracks which frames have been modified and transmits those frames back to the DBMS during a KB-save operation. Given this basic architecture, other choices must be made: How should FRS information be organized in the DBMS? One of our goals is that the DBMS should be invisible to the end user. The user should not be obliged to understand the DBMS schema, nor to design a new DBMS schema for every new KB. Instead, we, as designers of the storage system, must create a generic schema that accommodates all potential FRS information. In fact, we have designed and evaluated several such schemas empirically (results omitted here because of space limitations).

Another choice 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. 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 memory (analogous to page faulting in operating systems). We have also explored the transferring of several frames together.

Which type of DBMS is best suited to the role of a frame storage system? Our storage-subsystem design does not place many requirements on the underlying DBMS. The requirements are that frame definitions must be stored as uninterpreted collections of bytes, that frame definitions be saved and fetched in their entirety, and that only a few interframe relationships are stored explicitly in the DBMS, such as the class-instance relationship. Given the maturity of relational technology, it was logical to consider using a commercial relational DBMS (RDBMS) for persistent storage. However, the simple nature of our storage requirements led us to also consider using a low-level object manager. Our reasoning is that a low-level storage manager might be more efficient than a sophisticated commercial RDBMS. By using a small group of pertinent features, we might be able to avoid the overhead of a general-purpose system. We have experimented with both a commercial RDBMS, and an extensible storage management system, EXODUS, from the University of Wisconsin [3]. Here, we compare the performance results for EXODUS with previous results obtained using the RDBMS [5].

The EXODUS storage manager is a flexible, low-level system intended for use as the foundation for domain-specific information management systems. Unlike a full database system, it does not provide advanced features such as high-level schema creation and manipulation operations, or query specification and optimization. It is accessed directly through a library of client interface routines, rather than through a declarative query language such as SQL. We thought that its simplicity, and consequent efficiency, might be well-suited for our project, where there is only one application (the FRS), and the objects to be stored are relatively straightforward.

3 Storage Subsystem Implementation

This section first provides an overview of the frame structures that LOOM and THEO employ, and then discusses the architecture of the storage subsystem, and its interactions with LOOM and THEO.

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 (or class) consists of a name and a definition. The concept definition is a set of necessary and sufficient conditions that an instance must meet to be an instance of the concept. Given this information, the LOOM classifier arranges all concepts into a subsumption (generalization) hierarchy. A LOOM relation (not to be confused with the database definition of a relation) is a KB-wide specification of the properties of a slot, such as its domain and range.

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. LOOM can perform both forward- and backward-chaining classification-based inference. 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, 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 do not currently support LOOM's production-rule inference.

3.2 THEO Structures and Operation

THEO shares many characteristics with LOOM. THEO frames are 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 perform classification. Given the basic structural similarity of LOOM and THEO, it is natural to develop a storage system that can serve both FRSs.

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

3.3 EXODUS Schema

We have designed an organization of EXODUS storage structures that can simultaneously store multiple frame KBs. The organization for a sample KB is shown in Figure 1. Each KB is represented by two EXODUS files and six EXODUS indexes. An EXODUS file is a collection of objects; any object can be retrieved quickly given its object id (OID) — an integer. An EXODUS index allows fast retrieval of a datum given a key. For example, the

Frames index in Figure 1 allows us to map a symbolic frame name to its OID. Each frame is stored as a single object in the frames file. A frame object contains the frame's *body* (an ASCII string that encodes all information required to recreate the frame) and type. Two types of frame are supported: classes and instances. LOOM relations are stored in a separate relations file.

The frames and relations indexes relate frame names to their corresponding OIDs. Relationships among classes and instances are maintained in the other four indexes. The supers and subs indexes relate class names to their superclasses and subclasses, respectively. The instances and classes indexes relate classes to their instances and instances to their parent classes. Our approach therefore stores the taxonomic hierarchy for a KB persistently. When the hierarchy changes, our storage system will store these changes persistently. This storage organization allows fast retrieval of individual frames by name, as well as retrieval of portions of the taxonomic hierarchy.

Each EXODUS volume has a special *root entry area,* in which meta-information about a KB is stored. This information consists of handles for the two EXODUS files and six indexes, and meta-information about the KB that is used by the application. An application can have several KBs open simultaneously. The EXODUS client interface (ECI) maintains a table of open KBs.

3.4 Frame Faulting

Users employ both function-call interfaces and declarative languages to manipulate LOOM KBs. These operations (such as retrieving or altering the value of a slot within a particular frame) generate frame *references.* LOOM resolves these references by searching internal tables that associate frame names with the data structures that implement frames. A *frame fault* occurs when an application (or LOOM itself) references a frame *F* that is not in memory. We have modified LOOM to call our storage system when a frame fault occurs. The storage subsystem faults a frame into memory by retrieving its body from the DBMS server. Because the ECI is networked, multiple users can access and update the same KB in a distributed (but uncoordinated) fashion. (Our future work will investigate methods of controlling multiple updates to a shared KB.)

After retrieving the body of *F* from the DBMS, the storage subsystem calls standard LOOM functions to create *F* within the 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, and possibly to other instances serving as fillers of its slots. LOOM normally expects all of these other frames (called the context of *F)* to be present in memory. Because it can be expensive to fault in the entire context of F, we load in as small a portion of the context as possible.

Part of the context is the ancestor frames of F. When processing a fault to frame F, we first fault in every direct parent of F that is not currently in memory (references to the parents of these parents are generated recursively). Therefore, all parents of F are loaded before F is defined.

The second part of the context is those frames referred to by F. The LOOM frame data structures implement such interframe references as LISP pointers. Imagine that F refers to a frame *G* that has not yet been faulted into memory; therefore no pointer to *G* can be defined. One solution to this problem is to fault *G* into memory — a solution we reject because when applied recursively it could conceivably cause the entire KB to be faulted in. The solution we chose is to create a stub object (a placeholder for G), to which a pointer can be created. If *G* is later faulted in, the storage subsystem replaces the stub in a manner that retains the validity of existing pointers. See [5] for more details on frame faulting and stub management.

4 Performance Experiments

The goal of the experiments discussed herein was to measure storage system performance as a function of knowledge base size. We therefore generated a series of random KBs, identical in every respect except number of instances. Each KB had 100 concepts, all primitive, with just one super each. Instances averaged five slots apiece, with an average of two fillers per slot. Half the slots were filled by integers, and the other half were filled by symbols. These parameters were chosen because they approximate the characteristics of SOCAP, the transportation-planning KB that is driving our work with LOOM [8]. Knowledge bases were generated with 500, 1000, 2000, 4000, and 5000 instances. For comparison, the same set of KBs was generated and saved to native LOOM flat files, to native THEO flat files, to the DBMS (both LOOM and THEO versions), and to ExoDUS (both LOOM and THEO versions). These variations of five 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 DBMS server were running on the same workstation, a SPARCstation[2] 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 (repetitions typically varied by less than 10%). Overall elapsed times were measured using the LISP time function. We measured the time spent in LOOM, THEO, the ECI, and the storage subsystem by monitoring key procedures using the CMU monitoring package. The CPU time spent in the DBMS server process was measured using the UNIX ps utility to observe total CPU time before and after each experiment.

Figure 2 shows the time required to reference some number of randomly chosen instances from KBs of different sizes for both LOOM and THEO. Each reference faults in at least one frame from the DBMS (when the parent classes of an instance are not memory resident, they are also faulted in). Each dashed line in these

[2] All product names mentioned in this paper are the trademarks of their respective holders.

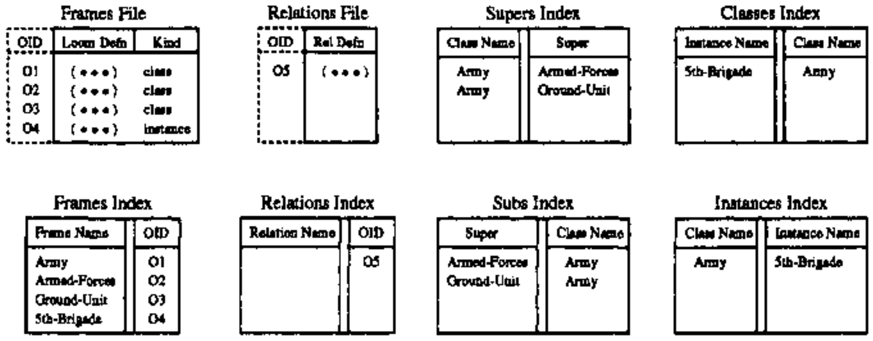


Figure 1: The organization of KB data in the EXODUS storage manager.

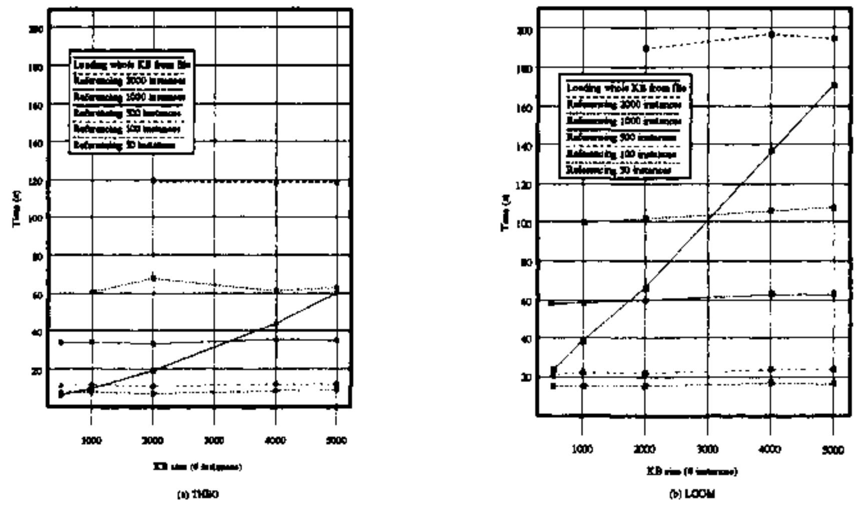


Figure 2: Comparison of KB loading times for THEO and for LOOM. The solid line shows the time required to load entire KBs of various sizes from flat files. The dashed lines show times required to fault frames from EXODUS in response to references to instances in KBs of various sizes.

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 2(a) shows that for THEO, the time required to reference 1000 instances from a KB containing 5000 total instances is about the same as the time required to load that KB in its entirety from a flat file.

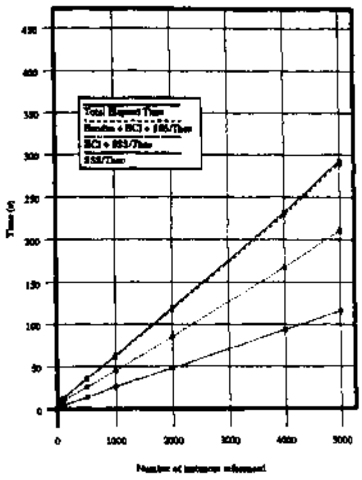
Figure 3 breaks down the total time spent processing frame faults into several components: the time spent in the EXODUS server, the ECI, our storage system, the FRS (LOOM or THEO), and other processing (presumably I/O), as a function of the number of instances referenced for a fixed KB of 5000 instances.

The next 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. Selected results for LOOM are shown in Figure 4. For comparison, we include 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 (not shown) are similar.

4.1 Discussion

Our experiments answer several questions: Does the performance of our DBMS-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 DBMS server contribute to its overall performance?

Figure 3 demonstrates that our architecture achieves the linearity goal: the time spent loading frames is a linear function of the number of frames referenced. Figure 2



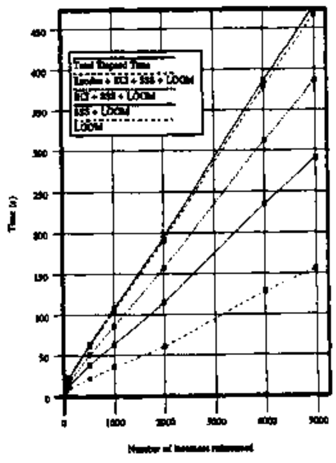


Figure 3: The total elapsed time for referencing and faulting N instances into memory from an EXODUS KB of 5000 instances is broken down into several components. The vertical distances between lines represent (starting at the bottom) time spent in the FRS, in our storage subsystem (SSS), in the ECI, and in the EXODUS server.

shows that frame loading time also depends to a small extent on KB size when a fixed number of instances are referenced. This dependency most likely occurs because (a) the parents of any referenced instance are faulted in along with the instance (if not already in memory), and (b) when a class is faulted in, the names of all its instances are also retrieved from the database. Since our experimental KBs contained a fixed 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.

Figure 2 lets us evaluate the relative merits of loading frames from the DBMS versus loading from fiat files. For THEO, loading N instances from the DBMS is 5 to 8 times slower than loading an entire KB of N instances from a flat file.[3] For LOOM, which must perform expensive classification operations on newly loaded frames, loading N instances from the DBMS is 3-4 times slower than loading that KB from a flat file. Therefore, the performance of the DBMS storage subsystem is on a par with a flat file when a user references up to 15% of the frames in a THEO KB or 30% of the frames in a LOOM KB in a given session. We believe that the performance of the storage subsystem is acceptable in practice, given our assumption that as KB size grows, users will reference only a fraction of its frames in a given session.

Our experiments (data not included) show that our EXODUS-based architecture achieves the goal of saving KB changes in time linear in the number of updates. Saving *N* updated frames to the EXODUS storage manager is roughly 7 times slower than saving an entire KB of *N* frames to a flat file. Therefore, our storage subsystem for EXODUS is faster than the flat file for saving information when less than 15% of the KB has been changed.

An earlier paper [5] describes the results of timing experiments using a relational DBMS in place of EXODUS. The outcome of the timing experiments with EXODUS reinforces the earlier results. In fact, not only are the shapes of the graphs similar, but so are the absolute values of the data points. Although there are minor differences, the bottom line is that we found the difference in performance between EXODUS and the RDBMS to be minimal. However, the RDBMS is much easier to work with from a practical point of view, because SQL provides a much higher level of interaction than does the extensive C++ programming necessary to interact with EXODUS. Therefore, we have chosen to use the RDBMS for our future work.[5]

Another advantage of the RDBMS declarative query language is its potential for evaluating complex KB queries within the DBMS. The RDBMS schema presented in [5] precludes such an approach because, like the schema presented in Figure 1, every frame is an uninterpreted blob within the DBMS. In subsequent work we have designed a more complex schema that makes individual slot values accessible to SQL. Using that schema we are able to index KB slots to support fast answers

[3]Retrieving N bytes from the DBMS incurs a much higher overhead than retrieving N bytes from a disk file due to factors such as query processing, network delays, buffer management, etc.

[4]In fact these classification operations are unnecessary since the DBMS already stores the results of previous classifications of these frames. We will consult with the LOOM developers about how to quickly insert into a KB frames with known subsumption relationships.

[5] Note added in proof: recent optimizations to the RDBMS storage subsystem have improved its performance substantially; the RDBMS is now faster than flat files when up to 70% (rather than 30%) of the KB is referenced in a session.

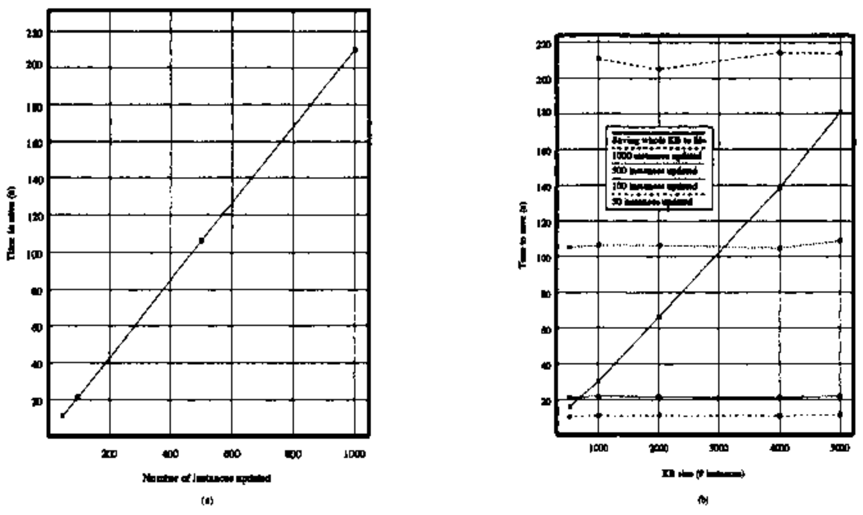


Figure 4: Time required to save instances from Loom to the EXODUS storage manager.

to declarative KB queries. The problem with this slot-based schema is that frame faulting is much slower because a number of DBMS queries are required to retrieve all the slots of each frame. By combining the frame-based schema with the slot-based schema we get the best of both worlds: we use the frame-based schema for faulting frames, and the slot-based schema for query processing. This approach does require redundant storage, and slows down the save operation. But if it is known in advance which slots will be queried, only those slots need be stored in the slot-based schema. Note that many FRSs support no indexing whatsoever of their virtual memory data structures, except that provided by the taxonomic hierarchy (LOOM does support such indexing).

5 Prefetching

Can the delays imposed by demand faulting of frames be decreased by prefetching frames that are likely to be referenced in the future? Our current system does not consider memory to be the limiting resource — we assume that all KBs can fit entirely in virtual memory. Our main concern is decreasing the time spent faulting frames. Since we never discard a frame once it is faulted into memory,[6] if a prefetched frame is ever referenced, then the prefetch did eliminate a demand fetch. Thus, prefetching has a much greater chance of success in our system than it does in, for example, page management by an operating system, in which a page must be discarded for every page prefetched.

Prefetching might improve performance in three different ways. If the application has idle time (e.g., waiting for user input or disk I/O), any useful work that the storage subsystem can do during these periods will eliminate demand fetches. Making effective use of idle time offers great potential for performance speedups, particularly in the situation in which a user is interactively browsing or editing a KB. However, such speedups will vary from application to application, making them difficult to evaluate experimentally.

Second, if we can bundle a request for several frames into a single DBMS query, we might decrease the overhead involved in query processing and data transmission, compared to that of fetching each frame individually. We performed experiments to determine the cost of fetching frames from the database as a function of the number of frames fetched at a time. We retrieved 639 SOCAP frames (161 concepts and 478 instances) and 1073 random KB frames (98 concepts and 975 instances), varying the fetch granularity from one frame at a time to 100 frames at a time. The average elapsed times per concept and per instance are shown as a function of the number of frames fetched at a time in Figure 5. Each value is an average of five trials.

For both KBs, and for both concepts and instances, as the number of frames fetched at a time increases, the time per frame drops sharply. At its minimum, the time per frame is one half that required to fetch frames individually. These results suggest that so long as 10 to 40 frames are fetched at a time, and more than half of the prefetched frames are actually referenced by the application, there will be a net gain in performance.

Finally, if the DBMS server is on a different machine than is LOOM, we might find an arrangement where both machines work in parallel. (Parallelism cannot be obtained with demand fetching, since we can't continue with processing until the query has completed.)

The prefetching scheme uses all three of the above strategies. The first two strategies are implemented for LOOM in conjunction with the RDBMS.

5.1 Implementation of Prefetching

One problem with prefetching is that fetching and loading a frame into LOOM requires a significant amount of

[6]We expect to remove this restriction in future work so that KB size is not limited by virtual memory.

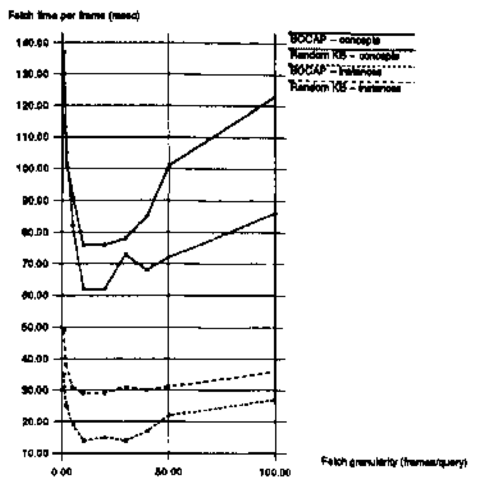


Figure 5: The time per frame to fetch frames from SOCAP and from our base random KB as a function of the number of frames fetched in one operation.

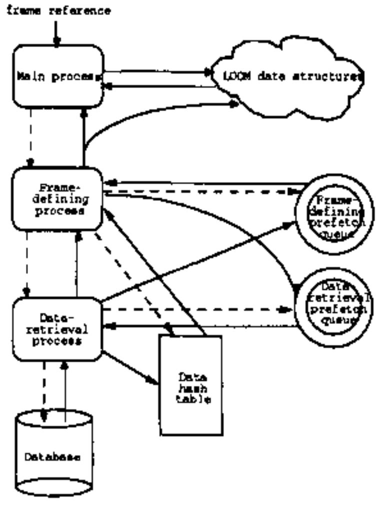


Figure 6: The processes and data structures involved in prefetching. Dashed lines represent requests only. Solid lines represent information flow.

computation on the local machine, for example, for classification. To allow this local processing and other user computation to occur in parallel with processing by the DBMS server, we divide frame fetching into two components: that part associated with retrieving data from the database and that part spent inserting the frame into the LOOM KB. The majority of the time involved in retrieving data from the database is spent on the DBMS server or in communication. Thus, we can perform data retrieval (DR) in parallel with local processing without significantly impacting local performance. The frame-defining (FD) task is performed locally, so we invoke it only when a frame is demanded or when the user process is idle. The DR component can obtain multiple frames with a single query, even though the FD component must define them one at a time. Thus, this scheme allows us to maximize parallelism at minimal cost to the main (user) process, to retrieve data for multiple frames in a single query, and to use extra local CPU cycles when the main process is idle.

Our implementation uses the Lucid Common Lisp multitasking facility to define two processes, a DR process and an FD process, in addition to the main process. The DR process runs with the same priority as the main process (i.e., they time-share). It chooses a frame or set of frames to retrieve (either a demanded frame or frames from a prefetch queue), initiates the appropriate DBMS queries, organizes the resulting bodies, and either returns them (if required as part of a demand fetch) or adds the body of each frame to a hash table for storage until needed (if a prefetch operation). The FD process runs at a lower priority, so it runs only when the other processes block, as in the case of a demand fetch, or are idle. It chooses a frame to define (either one that has been demanded, or one from the prefetch queue), gets the frame body either from the above hash table or by requesting that DR process fetch it, and invokes procedures to define the LOOM frame.

Figure 6 shows the interaction of the three processes and associated data structures. On a frame fault, the main process issues a request to the FD process to create the frame. The FD process first looks for the body in the hash table, and, if unsuccessful, asks the DR process to query the database. As the frame is being created, any unloaded frames that it references are added to the DR prefetch queue. When the DR process runs, it checks the DR prefetch queue for frames to prefetch, fetches and adds them to the hash table, and moves the frame references to the FD prefetch queue. When both other processes are idle or blocked, the FD process checks the FD prefetch queue for frames to define, obtains their bodies from the hash table, and creates the LOOM frames.

5.2 Prefetching Strategy

An important decision concerns which frames to prefetch. In most cases, there is no way of knowing which frames will be referenced in the future. (There exist situations in which frame references are known in advance, in which case the application may inform the system of which frames to prefetch.) The principle of locality suggests that the frames most likely to be referenced in the future will be related to those referenced most recently. We consider three types of frame relationships: a frame can fill a slot in another frame, a frame can be a subconcept of another concept, and a frame can be an instance of a concept.

We decided against prefetching all instances of a referenced concept because in large KBs we expect many concepts to have large numbers of instances. In this case, the probability of prefetching the right instances is small. Our first choice is to prefetch subconcepts of recently retrieved concepts.[7] The reason for this choice goes beyond the principle of locality to our intuition that concept frames are more likely to be needed than instance frames, because any reference to an instance frame also requires that its parent concepts be in memory. The probability of referencing a concept frame is the sum of the probabilities of referencing each of its subconcepts and instances. Therefore, any concept frame in the current region of interest of the KB hierarchy is a good candidate for prefetching. When there are no more subconcept frames to prefetch, we prefetch slot-filler frames.

Our initial experiments indicate that prefetching does improve performance in some cases, but our evaluation of prefetching is not yet complete.

6 Related Work

KEEconnection couples the KEE FRS with a relational DBMS [l] and the Intelligent Database Interface (IDI) couples LOOM with a relational DBMS [7]. In both systems the DBMS and FRS are loosely coupled *peers*. The advantage of this architecture is that it allows existing information from a database to be imported into an AI environment. Its drawback is that the storage capabilities of LOOM are not enhanced transparently, as in our approach. Users of KEEconnection (and of the IDI) must define mappings between class frames and tables in the RDBMS; KEEconnection creates frame instances from analogously structured tuples stored in the RDBMS, and can store instance frames out to the DBMS. However, only slot values in instance frames can be transferred to the database — class frames cannot be 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 object-oriented DBMSs [6; 2]. The IBM effort differs from our approach in that a KB is read from the DBMS in its entirety when it is opened by a K-REP user, which we believe will be unacceptably slow for large KBs.

None of these researchers have published experimental investigations of alternative implementations, as we are doing. Without systematic experiments it is impossible to evaluate the relative merits of their architectures.

7 Summary

An FRS that performs demand loading of referenced frames, combined with incremental saving of updated frames, will scale to large KBs much more gracefully than the current generation of FRSs. 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. We have also presented a prefetching mechanism that will improve performance in certain situations.

Our future work will investigate means of controlling multiuser access to shared KBs.

[7] Note that although the principle of locality applies to recently referenced frames, we are using it only for recently fetched frames because the overhead of recording related frames is too high to invoke on every frame reference.

Acknowledgments

Ira Greenberg implemented the ExoDUS-based storage subsystem. 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. This work was supported by Rome Laboratory Contract No. F30602-92-C-0115, and by grant R29-LM-05413-01A1 from NIH. 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- stitutes 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] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Storage management for objects in EXODUS. In *Object-oriented concepts, databases, and applications,* pages 341-369. ACM Press, 1989.

[4] P.D. Karp. The design space of frame knowledge representation systems. Technical Report 520, SRI International AI Center, 1992. URL ftp://www.ai.sri.com/pub/papers/karp-freview.ps.Z.

[5] P.D. Karp, S.M. Paley, and I. Greenberg. A storage system for scalable knowledge representation. In N. Adam, editor, *Proceedings of the Third International Conference on Information and Knowledge Management,* New York, NY, 1994. Association for Computing Machinery, also available as SRI International AI Center technical report 547.

[6] 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.

[7] 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.

[8] 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.