A Persistent Store for Large Shared Knowledge Bases

Eric Mays and Sitaram Lanka* and Bob Dionne and Robert Weida†

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

The issues addressed in this paper are: (i) to achieve persistence for knowledge bases (KBs), (ii) to achieve sharing of KBs, and (iii) to support the development and maintenance of large KBs. Persistence refers to storing a KB on a stable storage medium such as magnetic disk. We propose a Knowledge Base Management System (KBMS) in which a large KB is concurrently developed by a team of collaborating knowledge engineers. At the heart of the KBMS is a version store which is a persistent storage structure for a KB. Furthermore, to support the concurrent collaborative work, the version store maintains multiple versions of a KB such that a knowledge engineer can access and modify any version. Retrieve and Update operations have been defined on the version store to efficiently access and modify any version. Objects in a version store are clustered to support efficient access of an entire version of the KB or sub-parts of it. The retrieval algorithm has been validated through simulation. A prototype of the version store has been implemented, and is being integrated into the user interface.

1 Introduction

In this paper we will motivate the need for new techniques for managing problems associated with the scalability on large knowledge based systems. This investigation is based on our experience in building a large knowledge based system, FAME [9], and our perceptions regarding future technological requirements to support its ongoing development.

It is our perception that an unaddressed need exists regarding the technology required for the wide scale deployment of knowledge based systems. In particular, little attention has been given to knowledge systems languages and techniques as related to knowledge base development by a large and diverse group of knowledge engineers. For traditional software systems are any guide, large systems using large numbers of people will grow as the power of the tools employed increases.

In current practice, persistence of knowledge bases is obtained primarily through the storage of source level definitions in text files of the host computer's standard file system. That is, a knowledge base is stored the way a standard source file for a program is stored. This causes a waste in the storage required to maintain the various versions of the knowledge base, and also limits the concurrency achievable by the knowledge engineers.

A useful analogy can be drawn between the practice of knowledge based systems in the 80's and data based systems in the 60's. Early data processing was carried out with each application having its own file format. When applications needed to share data, programs were used to do file conversion and extraction. There was much duplication of data, and along with that came a potential loss in data integrity - applications had an inconsistent view of the same data. Data base management systems were developed to provide a unified, shared view of an enterprise's data. This enabled new applications to access data acquired from several diverse applications. Currently, knowledge based systems are characterized similarly to early data based systems. Each knowledge based system has its own knowledge base. When new knowledge based applications are developed, knowledge reuse and knowledge sharing is difficult to achieve. Moreover, as these knowledge bases evolve, they also diverge.

We see the need for a knowledge base management system (KBMS) which would support shared access to a large persistent knowledge base by multiple applications. Such a KBMS would support knowledge engineers in the development and maintenance of the knowledge base.

During the knowledge base development cycle knowledge engineers will load large portions of a shared knowledge base into a workspace. A knowledge engineer will make modifications to the knowledge base, test out the modifications, and perhaps commit those changes to persistent storage. However, those modifications may be of a tentative nature, in which case they will not need to be accessed by other knowledge engineers. At some point, the modifications made by several knowledge engineers will be merged to form a unified shared version. Obviously many variations on this central theme are possible. We choose this as our canonical problem, since most of the major issues involved in the variations will be solved by working on the central problem. In addition, the consistency of the knowledge base must be maintained throughout the knowledge base development cycle. The KBMS should ensure the consistency of the knowledge base as updates are processed so as to maintain the usability and coherence of the knowledge base.

Our goals for a KBMS consist of the following: (i) to allow a knowledge engineer to update a knowledge base and have these updates persist on secondary storage, (ii) to allow multiple knowledge engineers to have shared access to a knowledge base and modify the knowledge base concurrently, and (iii) to maintain consistency of the shared knowledge base as it evolves.

Our approach to this problem is to adopt a version oriented concurrency protocol [8] [1] [2] in which each knowledge engineer makes modifications to the shared knowledge base, thus deriving multiple versions. The version oriented protocol handles the problems relating to long transaction times and large lock grain sizes. Additionally, it places no temporal dependencies on the

---

*Visiting from The Pennsylvania State University.
†Also with Columbia University.
updates arriving from multiple knowledge engineers. From the multiple versions of the knowledge base it is necessary that a single unified knowledge base emerge. This requirement is achieved via the merge operation. Additionally, conflicting updates will be resolved through arbitration among the knowledge engineers. Since it would be unsatisfactory for knowledge engineers to work completely independently, we introduce the notion of groups to support the collaborative work process. Groups allow for consistency to be incrementally maintained among knowledge engineers who intend to merge their versions. While these techniques will be generally applicable, we are primarily concerned with an object-centered representation language, K-Rep [10], which we have developed and used extensively.

In this paper we focus on the storage structure within the KBMS. In the next section we briefly discuss related work. Section three characterizes the representation language. In section four we discuss persistent storage of an unversioned knowledge base and related retrieval and update operations. Full persistence of the knowledge base is addressed in section five. The implementation and simulation results are in section six. Finally, we conclude with a summary and future work.

2 Comparison with Existing Work

Similar to the work in design databases [8, 4, 3, 5, 7] we adopt a version oriented approach to handle concurrent updates to the shared KB. However, we differ from them on two accounts: (i) the level of granularity at which the versions are maintained, (ii) bringing the notion of consistency into the realm of the KBMS. In design databases, versions are maintained at the level of a file, while we maintain versions at the level of an object. The merit of our approach is that it requires a lot less space to handle changes to the KB. In the design database realm, checking for the consistency of a particular design is performed outside the database system. That is, the onus of checking the consistency of a design is on a source external to the database. In contrast, in our case the KBMS takes on the responsibility of maintaining consistency.

In [6] the issue of maintaining and updating various versions of a linked data structure such as a binary search tree is addressed. The manner in which we maintain various versions of a KB is somewhat similar to their fat node method. However, many of our concerns are quite different from theirs because we focus on minimizing the number of secondary storage block accesses.

3 The Representation Language

In this section we make precise what we mean by a knowledge base (KB). We will describe a KB in terms of the knowledge representation language K-Rep [10] which is in the KL-One [3] family of knowledge representation languages. A basic object for representation in K-Rep is a concept. Attributive information about concepts is captured via binary relations called roles. Concepts are organized in a generalization hierarchy with inheritance. Concept A subsumes concept B when every role, R, of A has a corresponding role, Rb, in B, such that the value restriction of Rb subsumes the value restriction of R. This representation language has the enforced semantics that if concept A is defined to be more general than concept B, and both A and B have a role R, then the value restriction of R on A must be more general than the value restriction of R on B. For expository purposes we will ignore other elements of the K-Rep language. A KB is a collection of sub-KBs, concepts, and roles. The name space of concepts belonging to a KB is partitioned into various sub-KBs. In other words a sub-KB contains pointers to the concepts contained in it.

A KB can be thought of as a collection of nodes corresponding to the sub-KBs, concepts, and roles to form a directed acyclic graph (DAG). The DAG is rooted at a special node, KB”. There is a directed edge from the root to each of the sub-KB nodes, a directed edge from a sub-KB node to each of the concept nodes contained in it, and a directed edge from a concept node to each of its role nodes. These edges are labeled "red" and will be referred to as red edges. A concept belonging to a sub-KB can refer to concepts in other sub-KBs. This dependence of a sub-KB on other sub-KBs is captured by a directed edge from sub-KB node n1 to sub-KB node n2 whenever at least one concept definition in n1 refers to definitions of concepts in n2. Note that the dependencies among sub-KBs form a DAG in their own right. The subsumption relation defined on pairs of concepts is captured as follows. If concept node n1 subsumes concept node n2 then by definition there is a directed edge from node n2 to n1. If the value restriction of a role is a concept, an edge from the role node to the concept node exists. In addition we assume that no edge introduces a cycle. Figure 1 illustrates a DAG corresponding to an entire KB.

The KB graph can be partitioned into four levels (the root is at level zero), and each partition is referred to as a horizontal partition. The 0th level horizontal partition contains a single node, KB", the 1st horizontal partition contains the sub-KBs, the 2nd horizontal partition contains the concepts, and the 3rd horizontal partition contains the roles. The red edges that link two adjacent horizontal partitions impose an onto mapping. That is, every node at level i can be reached from a node at level (i-1) via the red edges. Furthermore, each horizontal partition is further partitioned and each such partition is referred to as a vertical partition. All nodes reachable from a node via the red edges are placed in a vertical partition. Thus, the 1st level horizontal partition has one vertical partition containing all the sub-KB nodes, the 2nd level horizontal partition contains each sub-KB form a vertical partition, and at the 3rd horizontal partition roles of each concept form a vertical partition. Henceforth, it is useful to think of a KB as a DAG with the above characteristics and the operations on a KB are described in terms of a graph.

4 Persistent Storage of Unversioned KB

4.1 The Model

A KB is stored in secondary storage and is shared by multiple knowledge engineers (KEs). Each KE is allocated a separate workspace. To perform any operation on the KB (query it, execute problem solving services on it, or modify it), the shared KB must be first downloaded into an individual workspace. In the course of problem solving activity, the KB in a workspace may be modified. These modifications are referred to as the private component of a KB. Private components reside in a particular workspace, and are not shared by other users. A KE may make a number of changes to the KB, however, only the changes which the KE explicitly commits to the shared KB are written to the shared KB. With this model in mind, two kinds of operations are allowed on a KB: retrieval and update. Retrieval is defined as reading an entire KB or subparts of it (its sub-KBs) from secondary storage to virtual memory (workspace). While retrieving a sub-KB, other sub-KBs which are reachable from it in the DAG formed by the sub-KBs are also retrieved. An update operation is defined as making changes to the KB. The KB resides on secondary storage.
and accessing a secondary storage block will be the dominant cost while performing the operations on a KB. Minimizing the number of secondary storage block accesses will be the primary concern while proposing algorithms for the operations on a KB.

4.2 Retrieval Operation

The problem of retrieving a KB is equivalent to copying a DAG from secondary storage to virtual memory. The copy operation requires two passes because of the potential need to resolve forward references. In the first pass, the DAG in secondary storage is traversed such that all the nodes are visited. As each node is visited, a corresponding node in virtual memory is created. Notice that some of the descendants of a node may not have been created yet. So, at the time of creating a node, it may not be possible to store the pointers to its descendants. To facilitate this, two data structures are maintained. The first data structure carries information about the unresolved descendants. The second maintains a correspondence between a node and its virtual memory address. Now, whenever a node is created in virtual memory, the pointer to its corresponding secondary storage block is recorded.

In the second pass, with the aid of the two data structures the forward references are resolved, i.e., the information regarding the descendants at each node is completed.

In the copy operation, traversing the DAG is the most expensive operation as it involves secondary storage block accesses. Hence, the crux of designing an efficient algorithm to perform the copy operation is to traverse the DAG or a sub-DAG optimally, and we wish to focus on this problem. Three factors crucially affect the optimality of traversal of a DAG or a sub-DAG: (i) the connectivity of the DAG, (ii) the manner in which the DAG is stored, and (iii) the amount of information extracted from a block once it is in a virtual memory buffer.

Computing the transitive closure of all the nodes reachable via the red edges from the root of the DAG ensures that every node in the DAG will be visited. This claim is true because of the onto mapping imposed by the red edges between any two horizontal partitions. The DAG is stored as follows: Each vertical partition is assigned a separate area which consists of one or more blocks. And when a new node is defined, it is stored in its appropriate vertical partition. This implies that all the concepts of a horizontal partition are clustered together and so on. A merit of storing a DAG in this manner is that it is possible to traverse either the entire DAG or any of its sub-DAGs optimally. The key idea is that the nodes of a sub-DAG are not spread across all the blocks that hold the entire DAG. Due to limited buffer space in the virtual memory, it is crucial to extract maximum relevant information from a block once it is in a virtual memory buffer. In the absence of extracting maximum relevant information, performance degrades due to "thrashing" as illustrated by the following example. Consider visiting n nodes which fit on two blocks. Assume that the buffer space can accommodate only a single block. In the worst case, a block access is required (nodes are visited such that they are from alternate blocks). In contrast, if the nodes are visited in order of their block address, only two block accesses are required. This suggests that an efficient way to visit a set of nodes is to sort them by their block addresses and bring in the relevant blocks one at a time. Once a block is in a virtual memory buffer, read all the relevant information from the block. In this case, irrespective of the buffer replacement strategy, a block is brought in at most once.

We propose the following sketch of an algorithm to traverse a DAG which takes into account the above observations. The function "associate-node" records the association of a virtual memory address with a secondary storage address for a node.

In this algorithm the descendants reachable via the red edges of all the nodes at a horizontal partition are sorted by their block addresses, hence, no block is accessed twice. The time taken to retrieve an entire KB is $O(B)$, where B is the total number of blocks required to store the DAG. The number of nodes in a horizontal partition can be potentially very large, and in such cases, the time taken to sort their red edges may be substantial, and it must be taken into account.

**Algorithm Traverse-DAG-Union (node-list)**

(setf node-list (sort node-list))

(let ((successors (loop for n in node-list
                       do (associate-node n (copy-node-from-disk n))
                       collect (node-red-successors n))))
  (when successors
    (Traverse-DAG-Union (apply 'append successors)))

(loop for n in node-list
      for s in successors
      do (set-successors n s)))

4.3 Update Operation

The notion of a transaction is used to define an update operation. A transaction consists of a start operation, followed by a set of update steps, followed by a commit or abort operation. The start operation signifies the beginning of a new transaction. Commit or abort operations indicate the termination of the transaction. Commit indicates that the transaction has terminated normally and that its effects must be stored permanently in the shared KB. Abort indicates that the transaction has terminated abnormally and none of its effects must be stored in the shared KB.

The changes that are allowed on a KB are: adding a new concept definition, deleting an existing concept definition, modifying an existing concept definition by either (a) modifying the list of concepts that subsume the concept, or (b) modifying the roles of the concept (adding a role, deleting a role, modifying a role value restriction), creating a new sub-KB, deleting a sub-KB, and modifying an existing sub-KB. These changes are expressed via the update steps. An update step expresses changes at the level of the objects. Therefore, except for defining a new concept, all other changes to the KB can be expressed in one update step.

Defining a new concept requires $2 + n$ update steps — one update step to state the sub-KB it belongs to, another to define the body of the concept and $n$ update steps to capture its $n$ roles. At the transaction level, each update step is considered to be atomic.

The operational semantics of the execution of a transaction is as follows. If the transaction preserves the consistency of the shared KB then, it is committed, otherwise, the transaction is aborted. The effect of each update step of a committed transaction is stored in the shared KB. The time taken to store the effects of an update step is $O(1)$ (see section 5.2.2).

A transaction preserves consistency of the shared KB if collectively the effect of all the update steps in it yields a consistent KB. The effect of an update step is defined to be consistent if (i) it does not violate inheritance, i.e., the subsumption relation defined between concepts is preserved, (ii) it does not introduce a loop. We define a loop to exist if a node is reachable from itself. A node $n_2$ is reachable from a node $n_1$ if there is a directed edge from $n_1$ to $n_2$, and by transitivity, $n_3$ is reachable from $n_1$ if $n_3$ is reachable from $n_2$, and $n_3$ is reachable from $n_2$. Potentially, a loop is introduced in a KB whenever a subsumption relation is added, or a new value restriction for a role is defined. Thus, adding either $n_2$ subsumes $n_1$, or adding the value restriction for a role of concept $n_1$ to be concept $n_2$ can be construed as adding a directed edge from $n_1$ to $n_2$. Now, to check if a loop is introduced, test whether $n_2$ is contained in the transitive closure of the nodes reachable from $n_2$. A positive answer indicates that a loop exists, otherwise, it is loop free. The time taken to compute the transitive closure at a node is $O(n)$ in the worst case, where $n$ is the number of nodes in the sub-DAG rooted at that node. That is, it requires $n$ block accesses. This is because we have no way of telling ahead of time which nodes will be accessed.

Whenever a concept is modified, it is necessary to check if all the subsumption relations defined on it hold. To check the subsumption relation between two nodes it may be necessary to traverse all the nodes in the sub-DAG rooted at one of the nodes and check if its subsumption holds true with the other node.
to visit ancestor nodes in the subsumption hierarchy. Assume on an average m ancestor nodes are visited to check whether a subsumption relation holds. Let \( b \) be the average number of subsumption relations a node participates in. The time taken to check for violation of inheritance is \( O(m.b) \).

Caveat: Individually the effects of an update step may be inconsistent but it may be consistent in conjunction with other update steps in the transaction. This is illustrated by the following example. Consider a fragment of a KB containing three concepts, \( c_1 \), \( c_2 \), and \( c_3 \), where \( c_1 \) subsumes \( c_2 \) and \( c_2 \) subsumes \( c_3 \). Consider a transaction with two update steps: (i) add a directed edge from \( c_1 \) to \( c_3 \) (\( c_3 \) subsumes \( c_1 \)), and (ii) delete a directed edge from \( c_2 \) to \( c_3 \) (\( c_3 \) does not subsume \( c_2 \)). The effect of the first update step introduces an inconsistency, i.e., a loop, while the effect of the second update step resolves this inconsistency. Therefore, in the process of checking for consistency, a transaction is not aborted as soon as an update step is detected to introduce a potential inconsistency. Instead, the inconsistency introduced by it is temporarily admitted. In addition, a tally of the number of inconsistencies introduced is maintained. As and when an update step resolves an inconsistency the tally is appropriately adjusted. After processing all the update steps if the tally indicates no inconsistencies exist then the transaction is committed, otherwise it is aborted.

We make the following important observation. As described above, to make changes to a KB it must be first downloaded into a workspace. Since the relevant portion of the KB is already in the workspace, one could check the consistency of an update step without the workspace. However, we prefer not to do this because the copy of the KB in a workspace and the copy of the KB in stable storage need not be isomorphic. This is because the private updates made on a KB either by a problem solver or a KE are not shared by the copy in stable storage. Hence, consistency is checked with respect to the shared KB in secondary storage.

### 4.4 Data Structures to Store a KB

We will now describe the data structures used to physically store a KB. We associate a record type with each node of the DAG corresponding to a KB. That is, each node of a KB is associated with a record type. A KB record type consists of a sub-KB record type with each of its sub-KB nodes, a concept record type with each concept node, and a role record type with each role node. The root node of the DAG is associated with a KB record type.

#### 5 Full Persistence of the KB

In the last section we described a data structure used to store a KB. That data structure is ephemeral in that an update to the KB destroys the old version and only the new version is retained. Clearly, this is not desirable while developing a KB, especially when multiple knowledge engineers are simultaneously working on the task. What is called for is a fully persistent data structure that stores the KB. A data structure is fully persistent if multiple versions of it are maintained and every version can be accessed and modified. In this section, we will address the following problem: given an ephemeral data structure for a KB, how to make it fully persistent. In a fully persistent structure all versions are simultaneously maintained, and operations such as retrieval and update can be applied on any version.

#### 5.1 Version Store

A version store is a fully persistent data structure that stores a KB. Persistence can be achieved as follows: any change made to a node by an update step is recorded in the node itself without ever throwing out the old values. Information at a node grows as updates come in and hence they are referred to as fat nodes. There is a one-to-one correspondence between a node of an ephemeral KB and a fat node of the version store. To physically store a fat node we associate two data structures with it: a version block and a record type. A version block is a list of ordered pairs—version identifier, and pointer to a record. A version identifier uniquely identifies a version. Whenever a transaction is committed, the effects of it are stored as a new version of the KB. A fat node captures all the changes made at a node; each entry of the version block along with the associated record corresponds to a version of the node.

Different versions of the definition of the generic concept \( C_3 \) are captured in Figure 3. In version 1, \( C_3 \) is defined with \( C_2 \) as its super-concept, having two roles \( R_1 \) and \( R_2 \) which take reals and \( C_3 \) as their values, respectively. In version 2, definition of \( C_3 \) is updated such that the value restriction of role \( R_2 \) is changed to integers. Note that this update is incrementally recorded by creating a new role record for \( R_2 \). Since no changes were made in the concept record, there is no need to have another entry in the other version blocks corresponding to version 2, and version 3 reflects changes to role \( R_2 \).

##### 5.1.1 Version Graph

The various versions of a fully persistent KB form a partial order. This partial order is defined by a rooted DAG called a version graph, whose nodes correspond to version identifiers and a directed edge from node \( i \) to node \( j \) exists if node \( j \) is obtained by updating version \( i \). A version graph is maintained in conjunction with a version store as the partial order information is not stored within the version store.

#### 5.2 Operations on Version Store

As in the case of an ephemeral KB we allow two kinds of operations on a version store—retrieval and update. Retrieval is defined as reading any given version of a KB or subparts of it.
from secondary storage to virtual memory. Update is defined as making changes to any given version of a KB.

5.2.1 Retrieval Operation

Intuitively, the retrieval operation on a version store is similar to the retrieval operation on an ephemeral KB. In order to retrieve version $i$ of a KB, compute the transitive closure of the fat nodes reachable from the root via the red edges. We will now describe navigation within a fat node to choose the version corresponding to $i$. First we need to introduce the notion of an ancestor set. The ancestor set of a version is defined as the set of all version identifiers in the path from that version to the root, in the version graph. At each version block choose an entry whose version identifier is maximal with respect to the ancestor set of the version block such that $z$ and $x$ in the version block such that $z > i$ and $x < i$.

5.2.2 Update Operation

We allow a number of KEs to simultaneously update the shared KB. As in the ephemeral case, the notion of a transaction is used to define an update operation. The committed transaction of each KE is stored as a new version of the KB. The effect of a transaction on a version store is as follows. Consider a transaction which causes version $i$ to be created. If an update step in the transaction creates a new node, correspondingly we create a new fat node by creating a new version block with an entry, $< i, pt >$, where $i$ is the version identifier and $pt$ is a pointer to a new record of the appropriate record type which captures the information at the node. If an update step modifies a node, we make a new entry, $< i, pt >$, in the corresponding fat node's version block where $pt$ is a pointer to a new record which captures the modified information. Note that whenever a fat node corresponding to either a KB*, a sub-KB, or a concept is modified, the new record that captures the modified information must copy most of the information from the old version of the record.

For example, when a sub-KB node is modified (a concept is either added or deleted from it), a new sub-KB record must copy (most of) the concept list from the previous sub-KB record. That is, the update step which modifies a sub-KB object requires space in the size of the sub-KB record. Similarly, the update step that modifies a concept object requires space in the size of the concept record, and the update step that modifies the KB* node requires space proportional to the KB* record.

We propose to improve the above performance such that a version store uses only $O(1)$ space per update step. To achieve this we introduce a data structure called versioned set. This data structure allows us to capture incrementally only the new information required to generate the new version from the previous one. A versioned set is associated with a KB* node, and with each sub-KB and concept node and are referred to as sub-KB versioned set, concept versioned set and role versioned set. A versioned set data structure is a list of 3-tuples. The first projection of a tuple corresponds to the starting version identifier, the second projection corresponds to the ending version identifiers, and the third projection corresponds to a version block pointer. For example, in a concept versioned set each tuple corresponds to a concept in the sub-KB. The first projection of a tuple indicates the version in which the concept was included in the sub-KB, the second projection indicates the versions in which the concept was deleted from the sub-KB, and the third projection points to the concept's version block. Observe that adding a new concept to a sub-KB results in adding another tuple to the concept versioned set.

Deleting a concept from a sub-KB merely requires making an entry in the second projection of the role corresponding to the concept.

A similar description holds for the sub-KB versioned set and role versioned set. Thus, the version store uses $O(1)$ space per update step in the worst case. A versioned set is stored in the same area as the versioned set, as the various versions form a partial order.
as its associated version block. For instance, a sub-KB versioned set is stored in the same area as the sub-KB version blocks, a concept versioned set is stored along with its associated concept version blocks, and a role versioned set is stored along with its corresponding role version blocks. Again, facilitating efficient retrieval of a pre-specified version of the KB motivates clubbing the versioned set information with the version block information. The time complexity for the retrieval operation is approximately the same as before.

As in the ephemeral case a committed transaction must preserve the consistency of the version of the shared KB it is modifying. All the issues discussed regarding checking for consistency in the ephemeral case come to bear in this case.

5.2.3 Efficient Retrieval via an Index

The retrieval operation on a version store is a general mechanism to access any version of the KB. However, if we have a priori information that a particular version will be accessed often it would be beneficial to provide additional mechanisms such that the version can be accessed more efficiently than would be possible otherwise. We propose to build an index mechanism on top of a version store that would facilitate fast access of a pre-specified version of the KB. The idea here is to bypass the information related to other versions and directly access information that is pertinent to the pre-specified version. Thus, given an object corresponding to a fat node, the version block is bypassed and one directly gains access to the pointer to the pertinent version, and similarly given an object corresponding to a versioned set, one directly gains access to the associated fat node objects. This is achieved through an index. An index is a list of pairs whose first projection is a pointer to the pertinent record, and the second projection is either a pointer to a record or a set of object identifiers. This is shown in Figure 4. In the figure, the object identifiers are denoted as "#i" where i is an integer.

Now, retrieval involves determining the record pointer for a given object identifier through the index. Thus, no version blocks, or version sets are accessed. The time complexity of retrieving a version is \( O(B' + B) \) in the worst case, where \( B' \) is the total number of blocks required to store the record part of all the fat nodes and \( B \) is the number of blocks required to store the index.

6 Implementation

We have built the basic mechanisms of the version store using Symbolics' Statice object oriented DBMS. The choice of using an existing object oriented DBMS on which to layer the version store allows us to concentrate on the aspects that relate to KBMS design. While this frees us from having to implement basic storage allocation and page level concurrency algorithms it also constrains the amount of experimentation that may be done with storage allocation and page locking. KBMS transactions are composed out of several Statice transactions. Thus attention must be given to the atomicity of the KBMS transactions. We have successfully stored and retrieved knowledge bases consisting of several hundred concepts. Integration with the user interface is nearly complete.

We have also performed simulation runs comparing our Traverse-DAG-Union retrieval algorithm with the depth first (unsorted) retrieval algorithm. The results are given in Figure 5.

Two layout methods were used. The depth identity (di) method stores nodes on secondary storage using a depth first traversal of the tree. The union randomized (ur) method stores the nodes by randomly choosing a node within a horizontal partition. The randomized (ur) layout method approximates the worst case layout, while the depth first (di) is close to the optimal case. The two retrieval methods are a depth first (di) method and the union (ur) method. The x-axis is the branching factor of a synthesized balanced tree. The y-axis is the elapsed time. Note that the depth first access of a depth first layout (di di), the union access of the depth first layout (di ur), and the union access of the randomized layout (ur ur), all have similar performance characteristics. However, the depth first access of the randomized layout (ur di) is significantly worse. This indicates that the cost of the sort in the union method is much less than the additional disk accesses required for the depth first method. One can see from the graph that the union method is far superior to the depth first method.

7 Conclusion

We have identified the following problems regarding knowledge bases from our experience in building a large knowledge based system, FAME [9]. Present day environments that support KBs assume that the entire KB resides in main memory. In such an environment no special support is provided for the development of large KBs. Therefore, we recognize a need for an environment that supports (i) persistence of KBs (in the sense that a KB resides on secondary storage), (ii) sharing of KBs among knowledge engineers, and (iii) development and maintenance of large KBs. We have proposed a knowledge base management system (similar in spirit to a database management system) as such an environment. Our proposal with regard to the development of a large KB is to provide support for a team of knowledge engineers to concurrently work with the development of multiple versions of a KB. In addition, the assimilation of information from various versions is achieved by merging the versions. A merge operation involves collapsing the different versions into a single version and at the same time ensuring that the resulting version is consistent. During the merge operation, conflicts arising from the updates of multiple knowledge engineers are resolved through arbitration.

In this paper we have investigated the issues related to a central aspect of the KBMS, namely, the version store. A version store is a persistent storage structure for a KB. Furthermore, the version store maintains multiple versions of a KB such that one can access and modify any version of it. Retrieve and Update operations have been defined on the version store to efficiently access and modify any version. It takes \( O(1) \) space to store the effects of an update step in the version store. Objects in a version store are clustered so as to support efficient access of an entire version or some sub-parts of it. To access a version takes \( O(B') \) in the worst case, where \( B' \) is the number of secondary storage blocks on which the version store fits. In addition, to access a specified version via the index mechanism takes \( O(B' + B) \) in the worst case, where \( B \) is the number of blocks required to store all the records of fat nodes, and \( B' \) is the number of blocks required to store the index. Note that, \( B > B' + B \).

The retrieval algorithm has been validated through simulation. A prototype of the version store has been implemented, and is currently being integrated into the user interface.

The techniques presented in this paper are generally applicable to other representation languages. We have been primarily concerned with an object centered representation language, K-Rep [10], with which we have had experience developing and using. However, the requirements for consistency maintenance are highly dependent on the particular representation language of interest. Each language will have its own set of constraints which have to be met. These constraints can vary widely in their tractability.

We have begun to address issues relating to the maintenance of consistency of a KB. We have obtained complexity results for consistency checking of a KB maintained on secondary storage. In the context of multiple version of KBs, consistency also needs to be maintained across multiple versions which correspond to a group of cooperating knowledge engineers. This requires that some approximation of a merge be computed independent of dialog with the knowledge engineers. We have had some success in attempting to characterize an approximation to the performance relating to the interaction of multiple consistency constraints need to be further explored. We are investigating an incremental merge strategy in which the cost of the merge operation is
amortized over the time taken to perform all updates to the KB.

References


