A DATABASE IMPLEMENTATION FOR LARGE FRAME-BASED SYSTEMS

Dimitris Metaxas
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Timos Sellis
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract: A major requirement of database systems in engineering design and manufacturing applications is support for storage and maintenance of complex objects. Frame-based systems are capable of modeling complex objects. However, many of these systems are implemented in main memory. As the number of objects to be stored far exceeds the capacity of the main memory of a computer, such an implementation is often unusable. In this paper we present an implementation of a frame-based system on top of the POSTGRES extensible database system. Such an implementation combines the advantages of database management and frame-based systems and allows for the development of large engineering applications with minimal effort.

1 INTRODUCTION

One of the major requirements of database systems in engineering design and manufacturing applications is support for storing and maintaining complex objects. The object-oriented paradigm is one of the most common ways in Artificial Intelligence (AI) systems to model complex objects. Such a system is composed of a collection of objects that contain both data and procedures that are applied to the data. These objects are organized into an object hierarchy. The result is a so-called AI frame system. Due to the nature of object-oriented languages, these objects are stored in main memory. But in many manufacturing applications, the number of objects to be stored far exceeds the capabilities of the main memory of a computer, making such an implementation impossible. There are three ways to solve the above problem:

(1) The first is to link a main memory frame-based implementation to a Data Base Management System (DBMS). This can be done by embedding a DBMS query language in the retrieval language of the frame system, e.g., LISP. Queries are then generated whenever the frame system needs to access data from the DBMS. The major drawback of this alternative is that operations on the DBMS are performed differently depending on whether they are application dependent or provided by the DBMS's data model. In the latter case they are application independent (generic), and are usually interpreted, while in the former they are compiled. Such dual execution control is not in accordance with an AI system philosophy of treating all data operations in a uniform manner. Also the above need for interpretation or compilation brings up complexity and efficiency issues [Lafu83].

(2) A second approach is to implement the AI frame system on top of a DBMS. In particular, the relational model is a strong candidate due to its popularity and advanced functionality among the data models of widely available DBMSs. The advantage of such an approach is that one does not need to write a new DBMS and thus can avoid the effort already put in the development of DBMSs. A few years ago the disadvantage of such an approach could be the poor performance of the existing DBMSs. However, quite a few years of research and development have been put into building systems that exploit high performance and functionality.

(3) Clearly one last approach could be to build an AI system with data management capabilities. Given the advanced capabilities of modern relational systems, we consider this to be an inefficient way for building such AI systems.

Based on the above, the second approach seems to be the most appropriate for building a large frame-based system needed in many manufacturing applications. The selection of the DBMS is very important regarding the generality and ease of implementation of the AI frame-based system, as well as its performance. POSTGRES, a new database system currently under development at the University of California at Berkeley [Ston86], seems a very good candidate in this case. Although the underlying data model is still relational, POSTGRES offers several new features such as alerters, triggers, storing and processing rules, support for storing and executing general procedures, and, finally, a rich query language (POSTQUEL—POSTgres QUEry Language) [Rowe87]. All these features can be used in modeling the complex objects in a frame
system.

In this paper we follow the second approach and present a complete design of an AI frame system on top of POSTGRES. In section 2 we define the AI frame system, for which section 3 gives a POSTGRES based implementation. The fourth section surveys previous attempts and compares them to our design, while section 5 has the conclusions.

2 DEFINITION OF AN AI FRAME SYSTEM

In this section we define the structure and functionality of a frame system. Although this is just an example, it captures the functionality of most commonly used frame systems.

Definitions

(1) Each frame has one or more slots describing various components in the frame. For example, Toyota may have the following slots: Engine, Tires, Color.

(2) Each slot has one or more facets describing various properties of the slot. For example, the value of a slot is one of the slot's properties and is stored in a special Value facet. A facet can store a constant or a procedure which evaluates to a constant. The user has the freedom to create these slots and facets dynamically.

(3) Frames are organized in a hierarchy using two distinguished slots, Parents and Children. For these two slots, the Value facet contains a list of parent and child frames respectively. If for two frames p and g, p is in the Value facet of the Parents slot of g, we will call p a parent of g. Similarly, g will be the child of p, and g will be in the Value facet of the Children slot of p.

(4) Slots and facets can be inherited from parent to child. For example, suppose frame p has a slot called s and that s has a facet called fct. If g is a child of p, then g will also have a slot s which has a facet fct. Values stored in facets are also inherited. Suppose that in p, facet fct of s has the value Vp. Then in g, unless a different value has been explicitly stored in facet fct of s, the value Vp will be inherited from p.

(5) Inheritance can be single or multiple, depending on the number of parents that a frame has. In case of multiple inheritance it can be depth-first or breadth-first. That is, one looks for parent frame slots that can provide a missing facet value by moving upwards the hierarchy on a depth-first or breadth-first manner, respectively.

(6) Triggers are procedures activated in response to an update, removal or access of a facet's value; they are discussed in more detail later in this section.

An example LISP implementation of an AI frame system having the above characteristics, uses the following functions.

Basic functions

The following basic functions show how to create a new frame, insert, delete and update a value stored in a facet.

(1) (newframe name parentlist)

newframe creates a new frame called name. It creates a Parents slot for name and puts parentlist into this slot's Value facet. It creates a Children slot for name and puts NIL into this slot's Value facet. For each frame f in parentlist, newframe appends name to the list stored in the Value facet of the Children slot of f. It then returns name.

(2) (putfacet f s fct V)

putfacet puts value V into facet fct of slot s in frame f. If there is not already a slot s or facet fct, it creates them. It then returns V.

(3) (getfacet f s fct)

If the fct facet for slot s in frame f has a value V, then getfacet returns a list whose only element is V. Otherwise it returns NIL.

(4) (rmfacet f s fct)

rmfacet removes the explicitly stored value V (if there is one) from slot s's facet fct. Thus, future calls to (getfacet f s fct) will return NIL. If a value V was found and removed, then rmfacet returns a list whose only element is V. Otherwise, it returns NIL.

Complex functions

The following functions show how triggers and inheritance are implemented in an AI frame system.

(1) (getval f s)

getval computes a value V as follows. If the Value facet of slot s in frame f has a value val (either explicitly stored or derived through inheritance), then V is a list whose only member is val. Otherwise, the value for slot s is computed by using a special facet Ifneeded. If the Ifneeded facet of slot s in frame f has a value func (explicitly stored or by inheritance), then V is a list whose only member is the result of evaluating this function. For example, in LISP this is done using (funcall func f s), that is, by calling function func with f and s as arguments. Otherwise V=NIL. getval returns V.

(2) (putval f s V)

putval puts V into the Value facet of slot s in frame f. Furthermore, if s has (explicitly or by inheritance) an Ifput facet, putval will then execute (in LISP) (funcall func f s V), where func is the function name stored in the Ifput facet. putval returns V.
(3) \textit{rmval \ f s}

\textit{rmval} removes the explicitly stored value (if there is one) from the Value facet of slot \textit{s} in frame \textit{f}. Thus, future calls to \textit{(getfacet \ f s Value)} will return NIL. Furthermore, if \textit{s} has (explicitly or by inheritance) an Ifrm facet, then \textit{rmval} will execute (in LISP) \textit{(funcall func \ f s)}, where \textit{func} is the function name stored in the Ifrm facet. If an explicitly stored value \textit{V} was removed, then \textit{rmval} returns a list whose only element is \textit{V}. Otherwise it returns NIL.

Notice that the Ifneeded, Ifput, and Ifrm facets are special facet names used to distinguish facets containing trigger actions from other, user-defined, facets.

3 IMPLEMENTING FRAME SYSTEMS USING POSTGRES

The frame system presented in the previous section includes structures (i.e. frames, slots, facets) and functions on them. In the two subsections that follow we describe the implementation of these structures and functions using POSTGRES.

3.1 Mapping frames to POSTGRES relations

The structure of frames is very similar to relations, therefore the mapping is straightforward. In particular,

1. Sets of frames in the frame system correspond to relations. The attributes of such a relation, say FRAMES, are as follows:
   \begin{itemize}
   \item \textit{Frame-id}: Unique identifier (name) for frames
   \item \textit{Slot-id}: Unique, within a frame, identifier (name) for slots of a frame
   \item \textit{Facet-id}: Unique, within a slot, identifier (name) for facets of a slot
   \item \textit{ValueOfFacet}: Value of a specific facet.
   \end{itemize}

2. Every frame corresponds to at least one tuple in the FRAMES relation. If the frame has multiple slots, or multiple facets within a slot, then several tuples will have to be stored in FRAMES for that frame.

Example 1:

Suppose two frames Toyota and Car are defined as follows:

\begin{itemize}
\item \textbf{Frame} Toyota
\item \textbf{Slot} Engine\textunderscore Type
  \textbf{Facet} Value: A-33
\item \textbf{Slot} Tires
\item \textbf{Facet} Type: Michelin
\item \textbf{Slot Parents}
  \textbf{Facet} Value: Car
\item \textbf{Frame} Car
\item \textbf{Slot} Type
  \textbf{Facet} Number\textunderscore of\textunderscore Wheels: 4
  \textbf{Facet} Country: Japan
\item \textbf{Slot Children}
  \textbf{Facet} Value: Toyota
\end{itemize}

The relation FRAMES will then be as in Table 1. For efficiency reasons, frames may be organized in classes. This way a large number of tuples in FRAMES can be clustered according to some of their properties (e.g. all frames storing information about cars can be grouped in one relation). This was not the case in the example frame system of Section 2. Classes will then be simulated by separate FRAMES relations, one for each class. The discussion that follows applies to this case as well.

3.2 Implementation of basic functions

Functions of this group are easily implemented through POSTGRES. We will give POSTGRES commands [Ston86] for each of the functions of Section 2.

Frames are stored in a POSTGRES relation FRAMES. The scheme of this relation is as follows:

\textit{FRAMES} (Frame\textunderscore id=text, Slot\textunderscore id=text, Facet\textunderscore id=text, ValueOfFacet=text)

We use "text" for the type of the last attribute just for convenience. Clearly this attribute can have values of any type, i.e. integer, real or string, and for simplicity we allow all of them to be stored as ASCII strings.

Given this relation, the basic functions are then implemented as follows.

1. \textit{(newframe name parentlist)}
   \textit{newframe} is implemented using the \textit{putfacet} command. For each \textit{f} in \textit{parentlist} we perform \textit{(putfacet f Children Value name)} to show that \textit{name} is a child of \textit{f}. Then, using \textit{(putfacet name Parent Value parentlist)} we store in \textit{name} the information about its parents. In this way the appropriate links are established between the frame \textit{name} and all frames in \textit{parentlist}. The implementation of the \textit{putfacet} command is shown below.

2. \textit{(putfacet f a fct V)}
   \textit{putfacet} is implemented using the append command in the following way.
append FRAMES (Frame-id = f, Slot-id = s, Facet-id = fct, ValueOfFacet = V)

Thus the FRAMES relation will be updated to include information about the new value.

Example 2:
The function call (putval Toyota Engine-type A-33) will be implemented as follows:

execute (FRAMES.put) with ("A-33")

where FRAMES.Frame-id = "Toyota" and FRAMES.Slot-id = "Engine-type" and FRAMES.Facet-id = "Value".

The "with" clause is used in POSTGRES to pass values to the commands stored in put [Ston86]. The two POSTQUEL commands stored in the put field will be:

1. An append command to put A-33 as the value of the Value facet. It has the form described in the basic functions implementation, i.e.

append FRAMES (Frame-id = f, Slot-id = s, Facet-id = fct, ValueOfFacet = V)

2. A POSTQUEL command corresponding to the trigger stored in the Ifput facet of the frame (for example, a command that updates fields of other, related, frame slots).

A similar implementation is used for functions getval and rmval.

Sharing common procedures

In case the above triggers are common to specific classes of frames and only the arguments change, a new relation, Trigger, can be created to store the various triggering procedures, i.e. the put, get, and rm fields mentioned above.

A field Trigger-id is used to differentiate among the triggers used in various classes, as shown in Table 2.

Based on this design, putval, getval, and rmval can be implemented using the execute command. For example, suppose that the trigger which was used above when Toyota’s engine type was put in the system, has a Trigger-id T1. Then, putval will be implemented as:

execute (Trigger.put) with ("A-33")

where Trigger-id = "T1"

where the put attribute, which is the first attribute in relation Trigger, will contain the two POSTQUEL commands

<table>
<thead>
<tr>
<th>Frame-id</th>
<th>Slot-id</th>
<th>Facet-id</th>
<th>ValueOfFacet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota</td>
<td>Engine-type</td>
<td>Value</td>
<td>A-33</td>
</tr>
<tr>
<td>Toyota</td>
<td>Tires</td>
<td>Type</td>
<td>Michelin</td>
</tr>
<tr>
<td>Toyota</td>
<td>Parents</td>
<td>Value</td>
<td>Car</td>
</tr>
<tr>
<td>Car</td>
<td>Type</td>
<td>Number-of-Wheels</td>
<td>4</td>
</tr>
<tr>
<td>Car</td>
<td>Type</td>
<td>Country</td>
<td>Japan</td>
</tr>
<tr>
<td>Car</td>
<td>Children</td>
<td>Value</td>
<td>Toyota</td>
</tr>
</tbody>
</table>

Table 1: Relation FRAMES
discussed earlier.

**Implementation of Inheritance**

We will discuss the implementation of single and multiple inheritance separately.

**Single Inheritance**

In this case each frame can have only one parent. Single inheritance can then be implemented as follows. In the FRAMES tuple corresponding to the child frame, the attribute `ValueOfFacet` is defined to be of type POSTQUEL, instead of `test` that was used in the previous subsection. The reason behind such a decision is that constants can be a special case of POSTQUEL. In particular, if we want to store explicitly a value `const`, we can use the POSTQUEL command:

```
retrieve (value = const)
```

For example, in the Toyota frame, we store in the “engine type” tuple the POSTQUEL command:

```
retrieve (value = "A-33")
```

instead of the actual "A-33" value. Then, Toyota’s engine type can be retrieved as follows:

```
retrieve (FRAMES.ValueOfFacet.value)
where FRAMES.Frame-id = "Toyota"
and FRAMES.Slot-id = "Engine-type"
and FRAMES.Facet-id = "Value"
```

Notice the multi-dot notation in the target list of the retrieve command. POSTGRES will first produce the result of processing the POSTQUEL command stored in the `ValueOfFacet` field of the qualifying Toyota tuple (engine type slot), and then will project on the value field of that result. A similar notation was proposed by Zaniolo for GEM [Zani83].

In case we want this value to be inherited from Toyota’s parent frame, in which our example is the frame of Car, we store in the `ValueOfFacet` field of Toyota’s frame a command that retrieves the necessary information from the parent frame. This operation basically tries to find a FRAMES tuple whose `Frame-id` matches the value stored in the Parents slot of Toyota’s frame, i.e. a join. In particular, the operation has the following form (FRAMES1 and FRAMES2 range over FRAMES and correspond to the Toyota and its parent frame respectively):

```
retrieve (value = FRAMES2.ValueOfFacet.value)
where
FRAMES2.Frame-id = FRAMES1.ValueOfFacet
and FRAMES1.slot-id = "Parents"
and FRAMES1.Facet-id = "Value"
and FRAMES1.Frame-id = "Toyota"
and FRAMES2.Slot-id = "Engine.type"
and FRAMES2.Facet-id = "Value"
```

and we use the same retrieve command as before:

```
retrieve (FRAMES.ValueOfFacet.value)
where FRAMES.Frame-id = "Toyota"
and FRAMES.Slot-id = "Engine.type"
and FRAMES.Facet-id = "Value"
```

**Multiple Inheritance**

In this case a frame can have a list of parents, instead of just one. For example, Toyota can have as parents Car and Japanese Export stored in the form (Japanese.Export, Car) in the attribute `ValueOfFacet`, corresponding to attribute’s `Facet-id` value of `Slot-id` Parents. In this case the attribute `ValueOfFacet` will not be defined to be of type POSTQUEL but of type `cproc` (C procedure) [Ston86]. The reason is that POSTQUEL is not powerful enough to model dept-first or breadth-first searches. To implement multiple inheritance, we add a new attribute to FRAMES called `Explicit-ValueOfFacet`, of type `text`. In this attribute, we explicitly store the value of `ValueOfFacet` if it exists, otherwise we store NIL.

The procedure stored in the `ValueOfFacet` field is then of the following form:

```
If retrieval of the corresponding Explicit-ValueOfFacet returns a value not equal to NIL, return this value,
otherwise, recursively use each entry, name, in the list stored in attribute Explicit-ValueOfFacet of the tuple with Facet-id = Value and Slot-id = Parents,
to retrieve the missing value from name’s corresponding Explicit-ValueOfFacet entry.
```

Notice that the form of the recursive execution of the second step, depends on whether inheritance is breadth- or depth-first. The fact that the decision of the type of inheritance is done through storing a suitable procedure in an attribute, makes our design flexible and suitable for many applications. Also since that procedure is a general C procedure, we can check for integrity constraints on the

<table>
<thead>
<tr>
<th>Trigger-id</th>
<th>put</th>
<th>get</th>
<th>rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>trig1</td>
<td>trig2</td>
<td>trig3</td>
</tr>
<tr>
<td>T2</td>
<td>trig11</td>
<td>trig22</td>
<td>trig33</td>
</tr>
</tbody>
</table>

Table 2: Relation Trigger
inherited values.

The above shows that a frame system can be completely supported using POSTGRES. The basic mechanisms needed, and not found in conventional relational database management systems, namely triggers and procedural attachment, make POSTGRES an ideal candidate for such an implementation.

4 RELATED WORK

In this section we compare our design with other similar efforts that have been reported in the past [Rowe87], [Mylo85], and [Smit83].

OBJFADS [Rowe87] supports the definition and use of object hierarchies and is implemented through the POSTGRES data model. However, OBJFADS does not capture all the features of an AI frame system. In particular, the following three common features of a frame system are not covered.

- In most frame systems, a frame has an arbitrary number of slots, each of them having an arbitrary number of facets. These facets (not discussed in [Rowe87]), have values (constants or procedures) describing the various properties of the slot they belong to.
- In OBJFADS, slots have values which are constants, while, in general, the value of a facet can also be a procedure.
- Finally, triggers are not captured by the model. Triggers are stored in facets in the form of procedures, and activated by operations on slots. For example, as discussed in section 2, removing, accessing or updating a slot may cause a trigger stored in a facet of this slot to fire.

TAXIS [Mylo85] also has a lot of similarities with AI frame systems. It is a specific implementation of a semantic data model which is less powerful than our design in the following ways.

- An instance of a class cannot have attributes which are not defined in that class.
- Procedures can only be attached to classes, while in our case they can be stored in any attribute (of a class or an instance of it) making the approach more flexible.
- TAXIS does not permit multiple valued attributes.
- Attributes are inherited and their values are statically determined at compile time, whereas in our case they are dynamically determined, through the execution of a procedure, at run-time. Again, this allows for more flexibility when writing applications.

Clearly, the advantages of our proposal over TAXIS do not come for free. Determining values at run-time and, especially, through retrievals from relations is expensive. However, POSTGRES has a powerful caching mechanism which improves performance significantly. POSTQUEL commands stored in relation fields are processed once, and their result is stored in a cache. Any subsequent references to the same command can be answered from the cache, without paying the penalty of re-execution. Mechanisms for checking the validity of cache entries have been studied [Sel88] and simulation results which have been used to measure the usefulness of caching are very encouraging [Jhin88].

Another effort less similar to our design is STROBE [Smit83]. STROBE is a low level system supporting structured objects which permits only breadth-first multiple inheritance. It also provides a less general framework for slots and facets since a facet in our design can represent any property of a slot. STROBE is limited to only two kinds of facets, value and datatype.

5 CONCLUSION

In this paper we have presented the main features of an AI frame system and have shown how they can be implemented using POSTGRES. We believe that this is a promising way to design large frame systems which, due to their size, cannot be stored in main memory.

One of the motivations is that if we try to follow the alternative of enhancing an AI main memory system with secondary storage management capabilities we have first to implement a DBMS. Thus, we will have to face problems such as writing a query optimizer, a transaction management system, protection and integrity systems to control access to frames and maintain data consistency. Following the second alternative of linking an AI frame system to a commercial DBMS will result in significant performance degradation due to the need of interpretation or compilation of queries produced by the query language embedded in the language (e.g. LISP) in which the AI system is implemented. Another very crucial issue in this implementation is the requirement of consistency of data stored in main and secondary memory due to operations in main or secondary memory such as updates and deletions. This requirement is very costly and is sometimes ignored or very poorly implemented in such an approach. For example, the KEEconnection4 [KEE87] loads data from the DBMS into the KEE system, but if that data changes in the database, the frames in KEE are not updated.

Our approach uses an advanced DBMS to avoid all the problems discussed above. In addition, POSTGRES provides facilities to improve the performance of query execution, using query compilation and "fast-path" [Stone86]. It also has built in all the tools needed to implement frame systems such as support for complex objects, alerters, triggers, fields of type POSTQUEL and procedure.

Our current efforts focus on the investigation of optimization issues that will allow for the efficient execution of the basic and complex functions presented in the previous sections. The current limited version of POSTGRES

---

4 KEEconnection is a trademark of IntelllCorp Inc.
that has been released does not have support for triggers and procedural data, but it is expected that the first complete version will be available soon and we will be able to implement and test our ideas on the prototype.

Acknowledgements: Our thanks to John Mylopoulos, Louiqa Raschid, Isabel Cruz and Theodore Norvell for their interesting comments on an earlier draft of this paper.

References


