SOFTWARE COMPONENTS FOR OBJECT-ORIENTED DATABASE SYSTEMS

TR-92-27

Don Batory and Devang Vasavada

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

May 1992 (Revised October 1992)

Software Components for Object-Oriented Database Systems**

Don Batory and Devang Vasavada
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

Genesis is a software system generator for database management systems that relies exclusively on as-is large scale component reuse. We review the general model of software components on which Genesis is based and discuss component libraries for relational database systems that we have implemented. We then explain how we evolved Genesis and its libraries to be able to synthesize important classes of object-oriented database systems. We study a subproblem of creating "self-tuning" software systems by examining the performance of selected components for object-oriented database systems.

Keywords: inheritance, large scale reuse, GenVoca, Genesis, relational database systems, object-oriented database systems.

** This research was supported in part by a grant from Texas Instruments and Digital Equipment Corporation.

1. Introduction

Contemporary software design techniques encourage the creation of one-of-a-kind systems. As a consequence, most software is hand-crafted from scratch. *As-is* software reuse - i.e., the reuse of software without modification - is largely constrained to the reuse of platforms (operating systems, graphics packages, etc.) on which new software systems are constructed. While platform reuse is indeed important, it does not represent a complete picture of what is possible.

There are three granularities of as-is software reuse. *Small scale reuse* (SSR) is the reuse of algorithms or functions. The Unix function libraries and mathematical libraries are examples of repositories whose functions are reused as-is. We believe SSR is well-understood.

Medium scale reuse (MSR) is the as-is reuse of abstract data types (ADTs) or object oriented classes; i.e., the unit of reuse is a suite of tightly interrelated functions. Object-oriented programming environments, such as Smalltalk and C++, offer different mechanisms for reusing classes. Inheritance is one such mechanism. We believe MSR is also well-understood.

Large scale reuse (LSR) is as-is subsystem reuse; i.e., the unit of reuse is a suite of tightly interrelated classes or ADTs. As mentioned above, taking a subsystem from some software system A and a second subsystem from another system B to form a third system, C, is indeed rare. There are no accepted methodologies or techniques for achieving LSR; LSR is clearly not a well-understood problem.

Our research has focused on domain-specific LSR and has lead to two contributions. First, we have shown how as-is large scale reuse can be achieved in the domain of database management systems (DBMSs). We built Genesis, a system that enables customized DBMSs to be assembled quickly from prewritten, standardized, and plug-compatible components [Bat88a-b]. Second, we have extracted a domain-independent model of as-is LSR from Genesis and its network software counterpart, Avoca/*x*-kernel [OMa90, Pet90, Hut91]. This model is a blue-print for achieving as-is LSR in mature software domains [Bat92b].

From our experience, large scale software component technologies are indeed possible. We present in this paper some of our results and experiences and identify open problems in this important area of software engineering. Although not all readers will be interested in database-related issues, the basic themes that we address - namely the design of standardized and plug-compatible components, system synthesis, specification, customization, and optimization - are fundamental to software component technologies.

We begin with a brief overview of our general LSR model, called the *GenVoca* model. The basic idea of GenVoca is to standardize a domain of similar software systems in terms of a collection of plug-compatible components. We show that software systems of considerable complexity can be modeled by type expressions, which are compositions of components. We illustrate these ideas by discussing components that underly relational database systems.

Domains are not static, but constantly evolve through the influx of new technologies. Object-oriented database systems (OODBMSs) can use some, but not all, of the components of relational systems. We explain how we have evolved our component library to include building blocks for class inheritance, so that OODBMSs can be synthesized just like relational systems. Finally, we examine the performance of inheritance building blocks from the perspective of self-tuning software.

2. The GenVoca Model

The structure of large scale software systems can be captured by an elementary model that reflects the fact that systems are designed as assemblies of components and that components fit together in very specific ways. The GenVoca model postulates that components are instances of types and components themselves may be parameterized. The ways in which components fit together to form systems is captured by typed parameters and typed expressions.

Realms. Let R be the interface to one or more classes (say, in C++ [Str91]). Suppose interface R consists of three classes: I, J, and K. An implementation of R is an implementation of each of its classes. We call such an implementation a *component* of R.**

In principle, implementations (i.e., components) of R are not unique. We define the *realm* of R to be the set of components each of which is an implementation of R. Suppose the membership of `Realm_R` and a second realm, `Realm_S`, are given below:

```
Realm_R  =  {  a, b, c  }
Realm_S  =  {  d[ x:Realm_R ], e[ x:Realm_R ], f[ x:Realm_R ]  }
```

Note that interface R has three distinct implementations, namely the components `a`, `b`, and `c`. Typically each of these components has a distinct implementation for each of the three classes (I, J, and K) that define R. Similarly, interface S also has three implementations: components `d`, `e`, and `f`.

Parameters. Components may have parameters. All components of `Realm_S`, for example, have a single parameter `x` of type `Realm_R`. Each component of `Realm_S`, say `d[ x:Realm_R ]`, exports interface S and imports interface R. That is, component `d[ ]` translates objects and operations of S to objects and operations of R. The key idea is that the translation itself *does not depend* on how R is implemented; *any* implementation of R can be 'plugged' into component `d[ ]` to make `d[ ]` work. This underscores an important property of realms: realms are sets of components that are *plug-compatible* and *interchangeable*. In the above example where component `a` was used, components `b` and `c` could have been used instead.

Type Expressions. An important consequence of the above is that software systems can be modeled as *type expressions*. Consider the two systems shown below:

```
System_1  =  d[ b ]
System_2  =  f[ b ]
```

** A component is the combination of an interface R plus an implementation of each of R's classes. So technically the term "component" of R is not equivalent to the term "implementation" of R. However, the difference is so slight that it is not ambiguous to use both terms interchangeably.

`System_1` is a composition of component `d` with `b`; `System_2` is a composition of component `f` with `b`. Since both of these systems present the same interface (i.e., both present S), `System_1` and `System_2` are also interchangeable implementations of S.

Layers of Hierarchical Systems. The stacking of components in software systems can be interpreted as the composition of layers in hierarchical software systems. We will use the terms component and layers interchangeably, although we note that our use of the term 'layer' is probably different than its typical ad hoc usage.

Symmetry. A fundamental concept in the GenVoca model is the existence of symmetric components. Symmetric components have the unusual property that they can be composed in arbitrary ways. More specifically, a component is *symmetric* if it exports the same interface that it imports (i.e., a symmetric component of realm `W` has at least one parameter of type `W`). In the realm shown below, components `n[ ]` and `m[ ]` are symmetric whereas `p[ ]` is not.

```
Realm_T  =  {  n[ x:Realm_T ], m[ x:Realm_T ], p[ x:Realm_R ], ...  }
```

Because `n[ ]` and `m[ ]` are symmetric, compositions `n[m[ ]]` and `m[n[ ]]` are possible. Unix file filters are classical examples of symmetric components; file filters can be composed in virtually arbitrary orders. In general, the order in which components are composed affects both the semantics and performance of the resulting (sub)system.

Reuse. Software reuse is often difficult to quantify [Big89a-b]. A useful byproduct of the GenVoca model is that *as-is reuse* is easy to recognize. Consider two systems and their type expressions. If both expressions reference the same component, then that component is being reused. `System_1` and `System_2`, for example, reuse component `b`. More generally, if two systems have a common subexpression, then they share a common subsystem.

Domain Modeling. As mentioned in Section 1, large-scale as-is component reuse is unusual in today's software systems. This is a consequence of the fact that contemporary software systems are designed to be one-of-a-kind. A different approach to software design is needed to achieve LSR. *The value of the GenVoca model becomes evident when interfaces and their components are standardized and capture fundamental programming abstractions of a domain.* By having different implementation teams agree to use the same interfaces and abstractions, component reuse becomes possible. Designing generic interfaces and identifying basic abstractions through an in depth study of existing systems is called *domain analysis*, a topic that is currently under investigation [Pri91].

The GenVoca model has been successfully applied to the database and network/communication software domains [Bat92b]. In this paper, we show how it has been applied to the database domain. Other applications of the model are discussed elsewhere [Bat92c, Bat93].

3. Components of Relational DBMSs

Relational and object-oriented DBMSs can share many, but not all, components. Inheritance relationships among classes introduce subtle but important distinctions between classes and relations that preclude the wholesale interchange of relational DBMS and OODBMS components.

In order to expose these differences and relationships, we review components of relational DBMSs that we have built. In Section 4, we build upon these examples and ideas by explaining the generalizations and additional components that are needed to support inheritance.

We presume a familiarity with database concepts in this and the following two sections. Software components, in general, encapsulate complex mappings and it helps considerably to have some familiarity with basic database concepts. This is true for all applications of the GenVoca model that we have encountered.

As a general rule, the basic paradigm of encapsulation, as stated earlier, is the abstract to concrete mappings of data and operations. Although simple, this leads to layer/component boundaries that are quite different than those encountered in ad hoc "layered" systems built today. We attempt to bridge the unintuitive aspects of our software components by providing overviews of the ideas being discussed. Ultimately, the correctness of components and their compositions is borne out through implementation, which we discuss in Section 5.

3.1 Overview

A relational DBMS uses components from many realms. Genesis 2.2 alone supports thirteen distinct realms; more than twenty have been identified. In order to understand how structural inheritance impacts relational DBMSs, only three realms need to be considered. They are FMAP, LINK, and LANG:

```
FMAP  =  { index[ d,i:FMAP ], btree, heap, unordered, rle[ x:FMAP ], ... }

LINK  =  { pointer_array[ d:LINK ], ring_list[ d:LINK ], mjoin[ d:LINK ],
           nloops[ d:LINK ], link_term[ f:FMAP ], ... }

LANG  =  { sql[ x:LINK ], quel[ x:LINK ], ... }
```

As a brief description: FMAP components provide different ways to store relations, LINK components are different ways of implementing relational joins, and LANG components are different relational data languages. Each realm is discussed in detail in the following sections.

3.2 The FMAP Realm

FMAP is the realm of components that presents a procedural interface to files or relations. (We will use the terms 'file' and 'relation' interchangeably; we also make no distinction between the terms 'record' and 'tuple'). Among the exported operations of FMAP are the retrieval, insertion, deletion, and modification of records.

Each FMAP component maps the records and operations of an abstract file to records and operations on one or more concrete files. Note that a 'concrete' file to one component

may be an 'abstract' file to another. Hence, many FMAP components are symmetric; rle[x:FMAP] and index[d,i:FMAP] are two examples that we discuss below.

The rle[x:FMAP] component maps an uncompressed file to a compressed file using run-length encoding. rle translates operations on uncompressed records into operations on compressed records. For example, an insertion of a record r is mapped to an insertion of a run-length-compressed record r'. The parameter x indicates that the mapping performed by rle *does not* depend on how compressed records are stored. Thus, the rle component defines a generic mapping which can be reused in many contexts.

The index[d,i:FMAP] component maps an abstract file to an inverted file that has exactly one concrete data file and zero or more concrete index files. Operations on abstract records are translated by index into operations on data records and index records. Parameter d is the implementation of the data file, and parameter i is the implementation of the index files. This parameterization means that the mappings of index do not depend on how data records and index records are stored.

Terminal FMAP components (i.e., components without parameters) are file structures, such as btree, heap, and unordered.

Example 3.1. Suppose a relation is mapped to an inverted file, where tuples are stored in a heap structure and index files are stored in B+ trees. The composition of FMAP components that defines this mapping is F1:

```
F1  =  index[ heap, bplus ]
```

To appreciate the large number of details this compact expression hides, recall that the general paradigm for operation mapping is: a layer receives an operation, it performs local processing (e.g., updates) that is triggered by the received operation, and then the layer transmits the operation to the next lower layer(s). This process repeats until the lowest layer (i.e., a terminal or nonparameterized component) is reached. Consider the following.

Suppose tuples are to be retrieved from a relation that is stored according to F1. The retrieval operation of the top-most layer (i.e, index) is called. This operation is mapped by index into a retrieval of data records and index records; data record retrievals are mapped by heap into heap retrievals and index record retrievals are mapped by bplus into B+ tree retrievals.

Now consider what happens when a tuple is to be updated. The update operation of the index layer is called. index updates the index records that are affected by this change, and then passes the update to the next lower layer (which is heap). heap, in turn, makes the changes to the record in heap storage. The updates of index records are processed by bplus.

Example 3.2. Suppose the relation in Example 3.1 is to be compressed via run-length encoding after indexing but before heap storage. The composition of FMAP components that defines this mapping is F2:

```
F2  =  index[ rle[ heap ], bplus ]
```

3.3 The LINK Realm

A link is a logical relationship between two files; link traversals are relational join operations. `LINK` is the realm of components that presents a procedural interface to *both* files and links. This means that `LINK` components map both file and link operations.

Although links are not part of the relational model, they are central features of semantic data models, and to a lesser extent, of object-based data models [BCN92, ACM91]. In the following discussions, we depart slightly from the relational model by admitting explicit links.

Every link in a database schema is tagged with an implementation. A `LINK` component encapsulates a mapping of a database of conceptual files and tagged conceptual links to a less-abstract database where links of a given tag have been removed. In essence, a `LINK` component rewrites operations on links (with a given tag) into file operations. Classical `LINK` components include nonpointer-based methods (i.e., join algorithms of merge-sort `mjoin` and nested loops `nloops`) and pointer-based methods (e.g, `pointer_array` and `ring_list`). Consider the following examples.

Example 3.3. Suppose relations of a database are stored in B+ trees. Links between relations are implemented as either ring lists or pointer arrays. The compositions of `LINK` and `FMAP` components that define this mapping are `X1` and `X2`:

```
X1 = ring_list[ pointer_array[ link_term[ bplus ] ] ]
X2 = pointer_array[ ring_list[ link_term[ bplus ] ] ]
```

To understand what these expressions mean and how they differ, consider how join and file operations are mapped.

When a query requires a join, a *join cursor* (also called a *link cursor*) is created and tagged with the name of the layer that is to perform the link traversal.** In `X1` and `X2`, the only legal tags are `ring_list` and `pointer_array`. (Although query optimization has not yet been discussed, it is the query optimizer that creates the cursor and tags it). Thus, if a join cursor is tagged with '`pointer_array`', the `pointer_array` layer will perform the join; if the cursor was tagged with '`ring_list`', the `ring_list` layer performs the join.

When a join cursor is initialized, the join initialization operation of the top-most layer is called. If that layer recognizes its tag on the join cursor, it performs the cursor initialization. Otherwise, it does nothing and passes the cursor to the next lower layer. Eventually, a layer is encountered that recognizes the tag and performs the initialization.

What happens if a cursor is incorrectly tagged? The `link_term` component is responsible for resolving this question. All `link_term` does is to trap link operations and report them as errors; no layer above `link_term` recognized the tag of the join cursor and thus no layer above `link_term` processed the operation. Such errors are 'fatal' because

---

** A *cursor* or *iterator* is a run-time object that is used to reference records within files [Kor91, ACM91].

they should never occur in a correctly functioning DBMS or query optimizer. In effect, link_term provides a run-time safety net.**

When a join operation is performed, the layer L that is responsible for the join issues file retrieval operation calls. These calls are transmitted through LINK layers beneath L without modification. (The reason is that file retrieval operations, in general, are identity mapped by all LINK layers). The processing of file retrieval operations begins at the first FMAP layer that is encountered.

The difference between X1 and X2 is the order in which update operations are processed by pointer_array and ring_list. In X1, update operations are first processed by the ring_list layer and only after all ring list interlinkages are made will the update operation be propagated to the pointer_array layer; the opposite is true of X2. For practical purposes, this is an insignificant difference. (Generally the ordering of layers is important, but in this particular example it is not significant).

Example 3.4. IBM's DB2 stores relations in a manner similar to that of F1. DB2 also uses merge-join and nested loop algorithms to implement links. The expressions that define this mapping are X3 and X4:

```
X3  =  nloops[ mjoin[ link_term[ F1 ] ] ]
```

As in the case of X1 and X2, there is no practical difference between the ordering of the nloops and mjoin layers in X3 and X4; both expressions denote semantically equivalent systems.

3.4 The LANG Realm

LANG is the realm of components that presents a nonprocedural data language interface to a database of files/relations and links. Each LANG component embodies a query optimizer that translates nonprocedural queries into efficient expressions that reference operations on conceptual files and conceptual links. Each LANG component is parameterized by the method in which links between conceptual files are implemented. This parameterization means that the mappings of LANG components do not depend on how conceptual links or conceptual files are implemented. Classical LANG components include SQL- and QUEL-based languages.

Expressions of type LANG correspond to relational database systems. Consider the following two examples.

Example 3.5. IBM's DB2 presents an SQL interface on top of a file-and-link database that is stored according to X3. The expression that defines DB2 is:

```
D1  =  sql[ X3 ]
```

** link_term[x:FMAP] also serves the useful purpose in type expressions as the place to insert FMAP components; LINK components generally do not have FMAP parameters.

Example 3.6. A DBMS that presents an SQL interface, implements links by nested loops and pointer arrays, and stores relations in an inverted file, where both data files and index files are stored in isam structures is:

```
D2  =  sql[ nloops[ ptr_array[ link_term[ index[ isam, isam ] ] ] ] ]
```

It is worth noting that `D1` and `D2` are among the many relational DBMSs that we have built with the components that have been listed. Each system consists of approximately 70K lines of C.

3.5 Operational Aspects

Database schemas are defined in the Genesis Data Definition Language (DDL), which is an amalgam of concepts taken from different relational and Entity-Relationship (E-R) data definition languages. A *schema* is a collection of file or relation declarations followed by a list of zero or more links declarations. A link is defined by its name, followed by a join predicate that specifies the relationship between two files. Figure 3.1 shows the schema of an `atlas` database, which consists of three relations: `nation` - which describes a nation, `border` - which records what nations border on each other, and `inventor` - a table of inventions and the nation in which each invention was created, and two links `invented_in` - which connects `nation` tuples to `inventor` tuples, and `borders_on` - which connects a `nation` tuple to its related `border` tuples.

Implementation hints are indicated by tags, which are exported by the components that define a DBMS. The tags present in Figure 3.1 are consistent with the DBMS (mapping) of `D2` of Example 3.6: `primary_key` is exported by `isam`, `indexed` is exported by `index`, and `ptr_array` is exported by `ptr_array`.

The semantics of each tag is straightforward. The `indexed` tag is used to adorn attributes for which secondary indices are to be created (e.g., `nation.name` and `inventor.category`). The `primary_key` tag adorns attributes that define the primary key of a relation. The primary key of `nation` is `nation_code`; the primary key of `border` is compound (`nation1, nation2`); and the primary key of `inventor` is compound (`invention, name`). The `ptr_array` tag adorns links to indicate a pointer-array implementation (e.g., `invented_in`). If a link has no tag (e.g., `borders_on`), then a join algorithm is assigned by the query optimizer. (Note that in `D2`, the algorithm would be nested-loops).

```
DATABASE atlas {

RELATIONS

   nation {
      nation_code    INT              primary_key  /* nation code           */ ;
      name           CSTRING ( 30 )   indexed      /* name of nation        */ ;
      capital        CSTRING ( 30 )                /* name of capital city  */ ;
      sqmile         INT                           /* size in sq miles      */ ;
      population     INT                           /* population of country */ ;
   };

   border {
      nation1        INT              primary_key ;
      nation2        INT              primary_key ;
      north          BOOLEAN                      ;
      south          BOOLEAN                      ;
      east           BOOLEAN                      ;
      west           BOOLEAN                      ;
   }

   inventor {
      category       CSTRING ( 20 )   indexed      /* general category    */ ;
      invention      CSTRING ( 30 )   primary_key  /* specific invention  */ ;
      name           CSTRING ( 30 )   primary_key  /* name of inventor(s) */ ;
      year           INT                           /* year of invention   */ ;
      nation_code    INT                           /* nation code         */ ;
   };

LINKS
   /* 1:n links */

   invented_in : nation.nation_code = inventor.nation_code    ptr_array     ;
   borders_on  : nation.nation_code = border.nation1                        ;

}.
```

Figure 3.1 An Example Schema

The query language of D2 (provided by Genesis) is standard SQL with one embellishment: link names can be substituted in place of their join predicates. For example, to retrieve the name of the inventions created in Mexico can be written in two equivalent ways:

```
select invention
from nation, inventor
where nation.name = "Mexico" and
      nation.nation_code = inventor.nation_code
```

```
select invention
from nation, inventor
where nation.name = "Mexico"
      and invented_in
```

3.6 DaTE - Putting It All Together

As type expressions are not the easiest notation to read, Genesis has a layout editor, called DaTE, which enables components of different realms to be composed graphically

[Bat92a]. DaTE (Database Type Editor) is, in essense, a visual module interconnection language which allows university-quality relational DBMSs to be easily specified in about ten minutes. An output of DaTE is a set of configuration files, which when compiled with the Genesis library, produces the target DBMS. The DBMS that is generated is *untuned* because tuning constants - that are part of every component - are assigned default values. By performing benchmarks, it is possible to tune the generated software by selectively altering these constants and recompiling. We will return to this issue in Section 5.

Another purpose of DaTE is to enforce design rules. All the components that we have presented sofar can be composed without restrictions. In general, composition restrictions - beyond matching type signatures of components - do exist. For example, some components (in other database realms) cannot work in the presense of other components; similarly, some components can only work when other specific components are present. DaTE enforces these restrictions (called *design rules*) to ensure that all DaTE-specified DBMSs are correct. Further discussions on DaTE and design rule checking are presented in [Bat92a-b].

3.7 Recap

Relational DBMSs can be assembled into seamless compositions of components. We have reviewed three realms of components whose compositions form the backbone of relational DBMS implementations. The (sub)systems we have described are among the many that have been built with Genesis.

The sizes of the components that we have discussed are tabulated in Figure 3.2. Some of these values are estimates, as the source code for several components (e.g., `quel[ ]` and `sql[ ]`) overlap considerably, making it difficult to clearly associate specific lines of code with individual components. Although 18K lines of source are specifically attributed to the components in Figure 3.2, there is approximately 37K lines of utilities (for query evaluation, buffering, recovery, etc.) and 9K lines of headers that are shared by most components. Thus, the size of DBMSs that are generated by Genesis can be quite large. Genesis source itself totals over 130K lines of C.

realm	component	lines of C
FMAP	index	1400
	rle	1000
	bplus	*2000
	heap	*1000
LINK	ptr_array	2400
	ring_list	3100
	nloops	1000
	mjoin	1050
LANG	sql	*2600
	quel	*2600

* means estimated

Figure 3.2 Size of Components

We remind readers that we have deliberately simplified the type expressions that have been presented. Issues of page buffer management, recovery, method of blocking records, primitive data types, etc. are encapsulated in their own components, whose presense are implicit in these expressions. Again, this is to help restrict our discussions only to the relevant issues of comparing components of relational and object-oriented DBMSs in Genesis.

In the next section, we examine components for object-oriented DBMSs and how some of these blocks have evolved from their relational counterparts.

4. Components of Object-Oriented DBMSs

A basic feature that distinguishes relational DBMSs from OODBMSs is the support of inheritance. Certainly there are other features, such as user-defined data types, user-defined operators, a close coupling with programming languages (e.g., C++ or CLOS), and possibly different concurrency control mechanisms, but these features have been present in relational systems in the past [Sch77, Sto85, Alb85]; it is the support of inheritance that we believe is the fundamental distinction between relational and OODBMSs. In this section, we will focus exclusively on the important problem of encapsulating implementations of inheritance as GenVoca components.

We differentiate the terms 'classes' and 'relations' in the following way. One can indeed visualize objects of a class and their attributes in a tabular form, just like tuples of a relation. However, classes are related via inheritance whereas relations are not. Thus, an object in one class is also an object in all of its super classes; there are no corresponding concepts such as 'super relations' or 'sub relations' in the relational model. This difference leads to the distinctions between relational and object-oriented components in Genesis.

OODBMSs consist of as compositions of components that belong to three realms. They are `CMAP`, `CLINK`, and `CLANG`:

```
CMAP    =   { class_index[ d:CMAP, i:FMAP ], store_all[ d:FMAP, o:FMAP ],
              store_few[ d:FMAP, o:FMAP ],  ... }

CLINK   =   { oo_pointer_array[ f:CMAP ], oo_ring_list[ f:CMAP ],
              oo_mjoin[ f:CMAP ], oo_nloops[ f:CMAP ], ... }

CLANG   =   { oosql[ x:CLINK ], ooquel[ x:CLINK ], postquel[ x:CLINK ] ... }
```

As a brief description: `CMAP` components provide different ways of storing classes, `CLINK` components offer different ways of joining objects in one class with objects in another, and `CLANG` are different object-oriented query languages. Each realm is described in the following sections.

4.1 The CMAP Realm

`CMAP` is the class counterpart to `FMAP`: `CMAP` is the realm of components that present a procedural interface to classes. `CMAP` components implement retrieval, insertion, deletion, and modification operations on objects.

Each `CMAP` component transforms an abstract object class to one or more concrete object classes. Like their `FMAP` counterparts, `CMAP` components can be symmetric. An example is `class_index[d:CMAP,i:FMAP]`. The primary `CMAP` components are those that implement inheritance; they are `store_all[ ]`, `store_few[ ]`, and `store_trunc[ ]`. The mappings of each are outlined in the following paragraphs.

Class Indexing. The `class_index[d:CMAP,i:FMAP]` component is the class counterpart to `index[d:FMAP,i:FMAP]`. It encapsulates the mapping of an abstract class to an inverted class [Mai86, Kim87]. This mapping differs from `index[ ]` in the following ways: if an attribute A of class C is indexed, then attribute A of all subclasses of C are also indexed. Thus, when an object of a subclass C' of C is inserted, it is indexed on

attribute A.

An index file for attribute A is a set of <*value, class, oid*> tuples. *oid* is the identifier of the object that has *value* as its A attribute; *class* is the identifier of the class in which the referenced object was initially inserted. The *class* attribute is not present in `FMAP` index records, which are simply <*value, rid*> tuples where *rid* is a record id. The *class* attribute of oid tuples differentiates objects in subclasses of C. Suppose all objects from class C' are to be retrieved that satisfy the predicate (A=v) for some value v. C' is a subclass of C. By examining the *class* value of an index record, it is easy to determine if the referenced object belongs to C' or any of its subclasses. If so, this object is returned. If the object is not a member of C' (and hence, is a member of a superclass of C'), it is not returned.

These differences require subtle but not significant changes to the relational component `index[ ]`. In fact, only eight percent (130 lines out of 1500 lines of C code) of `index[ ]` was modified to produce `class_index[ ]`. In our experience, converting `FMAP` components to their `CMAP` counterparts has not been difficult. Later we will see that some components do not change at all.

There are many ways of implementing inheritance relationships between classes. Every method known to us defines a 1:1 correspondence between each class of a database and an implementing relation. Methods differ in the ways objects are mapped to tuples. Below we discuss three methods that we have implemented. As stated earlier, each can be understood as a mapping of a database with inheritance relationships to a database without inheritance relationships.

Store All. The `store_all[d:FMAP,i:FMAP]` component relies on tuple replication as a method to implement class inheritance. Every class C has a corresponding relation RC where each attribute of C is also an attribute of RC. The object's identifier is technically not an explicit attribute of the object (although one can think of oids in this way). However, oids are stored explicitly as a field in RC tuples. Each object c in C is represented by a tuple rc in RC, where for each attribute A, c.A = rc.A (i.e., attribute values of each object equal the attribute values of the corresponding tuples). Figure 4.1 shows three classes, I, J, and K, along with their corresponding relations, RI, RJ, and RK. Note the identity of the tabular representations of each class and its relation.

To maintain the identity of tabular representations, inheritance requires each object c of class C to be an instance of each superclass C" of C. This means that there is a corresponding tuple rc" for object c in each relation RC", the relation underlying C". The `store_all` mapping clearly makes object insertions expensive. However, the benefit is that retrievals are inexpensive: retrieving objects from class C is equivalent to retrieving tuples from relation RC.

An integral part of the `store_all` mapping is the creation of an oid index. An oid index is a file of <*oid, rid*> tuples that define the correspondence between object identifiers and record ids. Whenever a tuple is stored by an `FMAP` component, it is assigned a record id. Depending on the storage structure used, a record id could be physical address or it could be a symbolic address. (Heap storage, for example, would assign physical addresses to records; B+ trees, in contrast, assign symbolic keys because records do not have permanent physical addresses in B+ tree structures). At the class interface, record ids are no longer visible and are replaced by object ids (which are 4-byte integers). To fetch an object from its oid

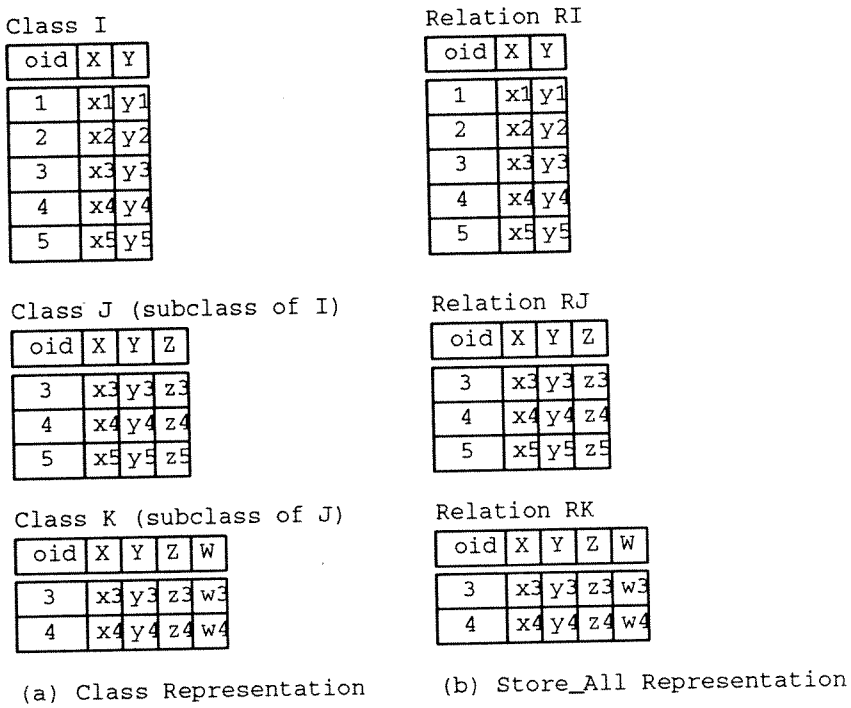

Figure 4.1 The Store All Mapping

requires a translation of the oid into an rid; this translation is the purpose of the oid index.

`store_all` has two parameters `d:FMAP` and `i:FMAP`. `d` specifies the implementation of the relations that are created, while `i` specifies the implementation of the oid file. `store_all` does not depend on how its relations or oid index are stored.

Store Few. The `store_few` mapping, in contrast, does not rely on tuple replication [Sto86]. As before, every class C has a corresponding relation RC. However, each object c in C is represented by a single tuple rc in RC iff c does not belong to any subclass of C. If c belongs to a subclass of C, then it is not explicitly represented in RC. Figure 4.2 shows the same three classes as in Figure 4.1 along side the relations that are created by `store_few`.

The store few method makes object insertions inexpensive; only a single tuple insertion per object insertion is made. However, retrievals tend to be more expensive than `store_all` since retrieving objects from C requires the retrieval of tuples from relation RC and relations that correspond to all subclasses of C.

Store Trunc. The `store_trunc` mapping is a variation on store all; it eliminates the redundant storage of attribute values [Fis87]. Once again, every class C has a corresponding relation RC where the attributes of RC include the oid and the attributes of C that *are not inherited.* Every object c in class C has a corresponding tuple rc in relation RC. The difference with `store_all` is that inherited attribute values are not replicated in `store_trunc`. The tables in Figure 4.3 illustrate this mapping.

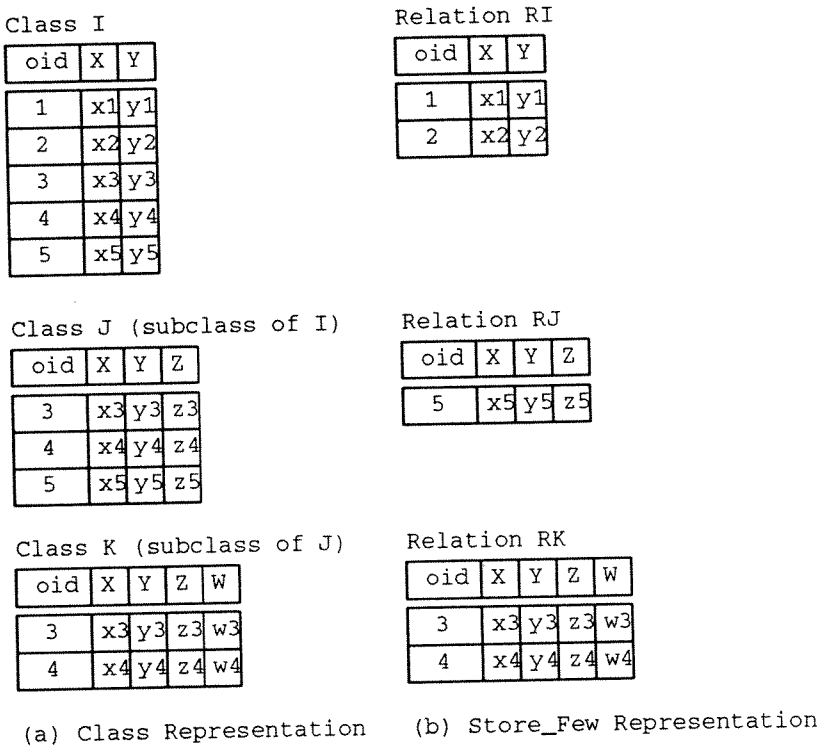

Figure 4.2 The Store Few Mapping

`store_trunc` makes updates more efficient than `store_all`, simply because redundant attribute values have been eliminated. The tradeoff is a higher cost for retrieval; to retrieve all attributes of an object in class C requires a join over the oid field of the relations that correspond to C and its superclasses.

4.3 The CLINK and CLANG Realms

`CLINK` is the class counterpart of `LINK`: it is the realm of components that presents a procedural interface to both classes and links between classes. Thus, the `CLINK` interface provides a superset of the operations exported by `CMAP`.

Join algorithm components, such as nested loops and merge joins (`oo_nloops[ ]`, `oo_mjoin[ ]`), are identical to their `LINK` counterparts. Components that connect objects via pointers (oids), such as pointer arrays and ring_lists (`oo_ptr_array[ ]`, `oo_ring_list[ ]`), are slightly different than their `LINK` counterparts. (Recall that `LINK` components augment fields to both the parent and child relations of a link. A `CLINK` component goes a step further by propagating augmented fields to all subclasses of the parent and child classes. As in the case of `class_index[ ]` these differences are marginal).

`CLANG` is the class counterpart of `LANG`: it is the realm of components that present nonprocedural data language interfaces to a database of classes, links, and inheritance relationships. Standard relational interfaces, such as SQL and QUEL, can be used to query and update objects in a OODBMS. (In fact, our `oo_sql[ ]` and `oo_quel[ ]` components

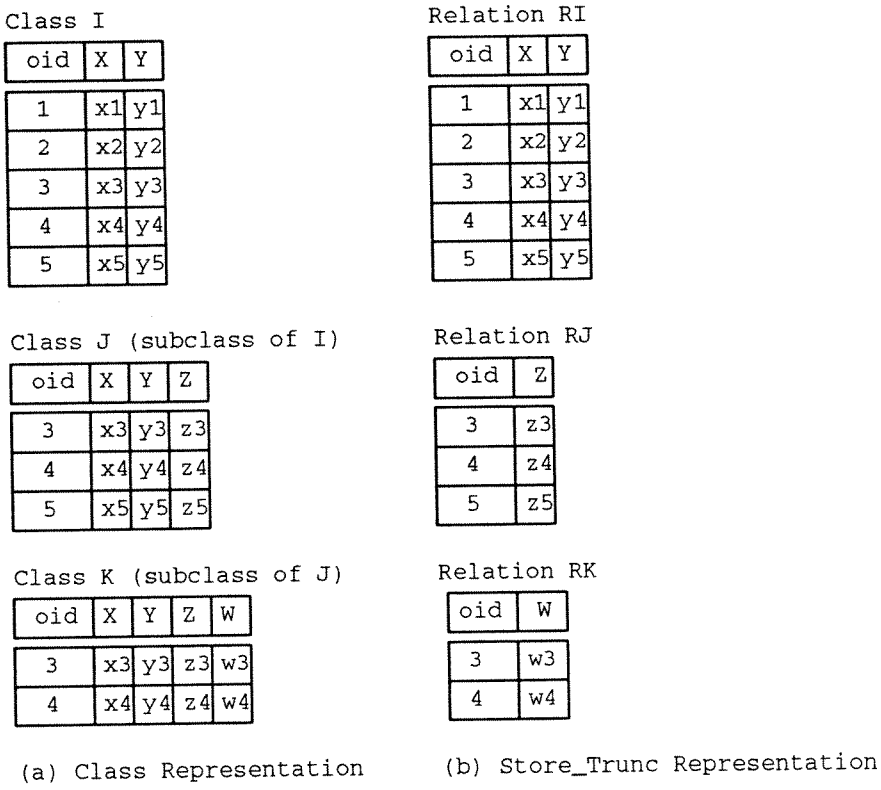

Figure 4.3 The Store Trunc Mapping

are identical to their LANG counterparts). This is not to say that object-oriented query languages are the same as relational languages; it is more of the case that relational languages provide a subset of the capabilities of OO languages. We may eventually supplement the CLANG realm with the `postquel[ ]` and `oql[ ]` components.

4.4 Object-Oriented Database Systems

Expressions of type CLANG denote object-oriented database systems. Consider the following examples.

Example 4.1. System B1 presents a Quel interface. It implements links by merge join and nested loop algorithms. Classes are indexed, where index files are stored in B+ trees. Classes are mapped by store all, where the oid index is stored in a B+ tree and the underlying relations are stored in heaps. The expression that defines B1 is:

```
B1  =  oo_quel[ oo_mjoin[ oo_nloops[ oo_term_link[ B1' ] ] ] ]

B1' =  class_index[ store_all[ heap, bplus ], bplus ]
```

Example 4.2. System B2 differs from B1 in that it presents an SQL interface, uses store few to implement class inheritance, and compresses its tuple representation of objects prior to storage in heaps. The oid and attribute indices are stored in isam structures. The expression for B2 is:

```
B2  =  oo_sql[ oo_mjoin[ oo_nloops[ oo_term_link[ B2' ] ] ] ]

B2' =  class_index[ store_few[ rle[ heap ], isam ], isam ]
```

B1 and B2 are among the many OODBMSs that we have been able to synthesize from components in the CLANG, CLINK, CMAP, and FMAP realms. The approximate size of each system is 75K lines of C.

5. Performance Results

The availability of components makes it very easy to reconfigure a software system. The number of distinct relational DBMSs that can be formed from the components listed in Section 3.1 is over 350 and the number of distinct OODBMSs that can be formed from the components listed in Sections 3 and 4 is over 2500.** As the number of components increases, the number of systems that can be assembled grows exponentially.

A unique and important feature of software component technologies (as we have described them) is the possibility of enumerating a spectrum of different software systems. An important question arises: what combination of components yields the system (in our case a DBMS) with the best performance for a given workload? An interesting variation on this problem is "self-tuning" software. A software system monitors its workload and periodically optimizes and reconfigures itself as its workload changes. These are clearly fundamental and formidable optimization problems. As we have noted even for a small number of components, the number of possible systems is enormous. Heuristics must be used to limit the search space of possibilities.

Our first steps to address both problems are modest: we are attempting to understand the tradeoffs of using different components in various situations. We take an experimental approach in this paper to compare and contrast the performance of the `store_few`, `store_all`, and `store_trunc` components under different workloads. By understanding their tradeoffs better, we believe in the long run it is possible to formalize these tradeoffs as DBMS design heuristics.

The DBMSs that we will use in our experiments present an SQL interface, use nested loop and merge join algorithms, perform class indexing and store oid and attribute indices in B+ trees and data records in unordered files. The type expressions that define this system is:

```
ODBMS( y )  =  oo_sql[ oo_nloops[ oo_mjoin[ oo_linkterm[ SS(y) ]  ] ] ]
SS(y)       =  class_index[ y[ unordered, bplus], bplus ]
```

where parameter `y` is to be instantiated by the components `store_all`, `store_few`, and `store_trunc`.

In the following sections, we survey numerous conditions and assumptions associated with experiments of this type to give readers an idea of the difficulty of addressing these problems.

5.1 Synthetic Databases

The complexity of a database can be described by the number of classes, the inheritance relationship among classes, the number and type of attributes per class, the distribution of attribute values that are assigned to objects, and the number of objects per class. Varying each of these parameters can alter a database noticeably and hence can affect the overall

---

** 350 and 2500 estimate the number of legal type expressions that can be formed from the components in the realms of Sections 3 and 4.

performance.

Unfortunately, it is not clear what are typical values for each of these parameters. We propose to assign values that are outwardly reasonable, knowing full well that any set or range of values that we could have used would be inherently incomplete. Our goal is not to be exhaustive in quantifying tradeoffs, but rather to improve our insight about the strengths and weaknesses of different implementation methods.

We adapted an approach taken in the Wisconsin Benchmark to generate synthetic classes [Bit88]. We examined class hierarchies that are a linear chain of classes where the number of classes ranges from two to five. Every class is originally empty. At database load time, 2000 objects are inserted into each class; thus in a linear hierarchy of five classes, the root class would have a total 10000 distinct objects; in a hierarchy of two classes, the root would have 4000 objects.

To further minimize database complexity, we assume that all attributes are uniform. In particular, attributes are strings 10 bytes wide. "Rotating" character strings are assigned to each attribute as prescribed in [Bit88], where the selectivity of an attribute is 1/100.** The root class has six attributes; each subclass specializes its superclass by adding another two attributes. In a hierarchy of n classes, objects in the leaf class are 60 + n*20 bytes wide (n=5 $\rightarrow$ 160 bytes).

We settled on this particular database for several reasons. First, the uniformity assumptions (if not the values) chosen are similar to those used by other researchers in experimental or analytic models [Kim86, Bit88, Sel79]. Second, these databases are among the simplest that could be generated. Clearly, more complex and varied classes and class hierarchies could have been considered, but we felt that the essence of the algorithms and their tradeoffs would be more clearly exposed by a simple database.

In the following, we report averaged results over a set of experiments that were run on a diskless SparcStation 1+.

5.2 Load Performance

Figure 5.1 shows the CPU time (system plus DBMS) that was needed to load the synthetic databases. Database loading is a sequence of object insertions into an initially empty database and thus reflects insertion (and more generally, update) performance.

As expected, `store_few` is the fastest, while `store_trunc` and `store_all` were progressively slower. Although the database grows linearly (in terms of objects inserted) as the number of classes increases, the performance of `store_all` and `store_trunc` is nonlinear. Each time an object is inserted into class C, a projected object is inserted into each of C's superclasses, leading to an $O(d^2)$ behavior, where d is the depth of the inheritance lattice. Thus, for deep lattices, there will be a substantial difference in the performance of `store_few` than `store_all` and `store_trunc` for insertion (and in general, for updates).

** The *selectivity* of an attribute is the average fraction of a relation that has a given value for that attribute. A selectivity of .01 means that an attribute has 100 unique values and that .01*n objects on average in a class of n objects will share a given value.

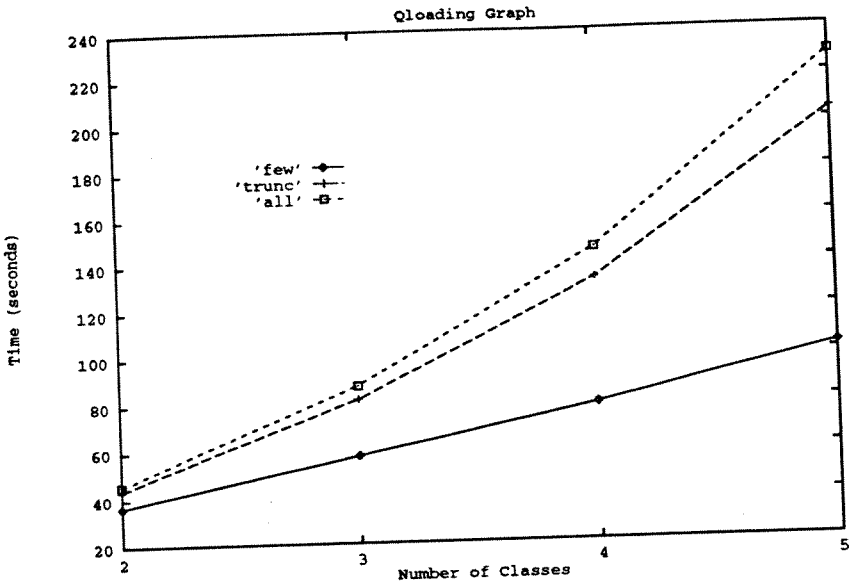

Figure 5.1 Insertion/Load Performance

5.3 Synthetic Retrieval Workloads

There are at least three degrees of freedom in categorizing retrieval workloads: predicate selectivity, attribute selection, and queried class. Each category has two possible subclassifications which are described below.

There are two basic types of selection predicates: those that can be processed using an index and those that can not. The latter type of query requires the examination of every object in the class (i.e., a scan) to determine the objects that satisfy the selection predicate. Indexable predicates were of the form (Root_Class.Attribute = value) while the nonindexable predicate that we used was 'true' (i.e., all objects qualified).

Two different types of projection lists can accompany a selection predicate: either all attribute values of an object are retrieved or just the value of a single attribute. To maximize the performance differences between `store_trunc` and the other methods, we made the retrieved attribute different than the attribute on which objects were qualified.

Another way to maximize performance differences between the various inheritance methods is to direct queries to both the root of a class hierarchy and to the leaves. In Figure 5.2, we list eight different query types that reflect the different possibilities of selection predicates, projection lists, and classes referenced in queries.

Query	Selection Predicate	Projection Type	Class Queried
Q0	nonindexable	one	leaf
Q1	nonindexable	all	leaf
Q2	nonindexable	one	root
Q3	nonindexable	all	root
Q4	indexable	one	leaf
Q5	indexable	all	leaf
Q6	indexable	one	root
Q7	indexable	all	root

Figure 5.2 A Classification of Query Types

In the following paragraphs, we examine the performance of each query class. We begin with queries Q0 and Q1 as they reveal performance information that is relevant to interpreting the performance graphs for all other queries. The graphs depicted in the subsequent figures are the average CPU time consumed by the system and DBMS over 70 different queries for each of the eight query types. CPU time is plotted with respect to the number of classes in the database.

Queries Q0 and Q1. Figure 5.3 shows the performance graphs for queries Q0 and Q1. Both of these queries scan the bottom-most class of a class hierarchy; they are different in that Q0 retrieves a single attribute for all objects whereas Q1 retrieves all attributes. Both queries return the same number of objects (2000).

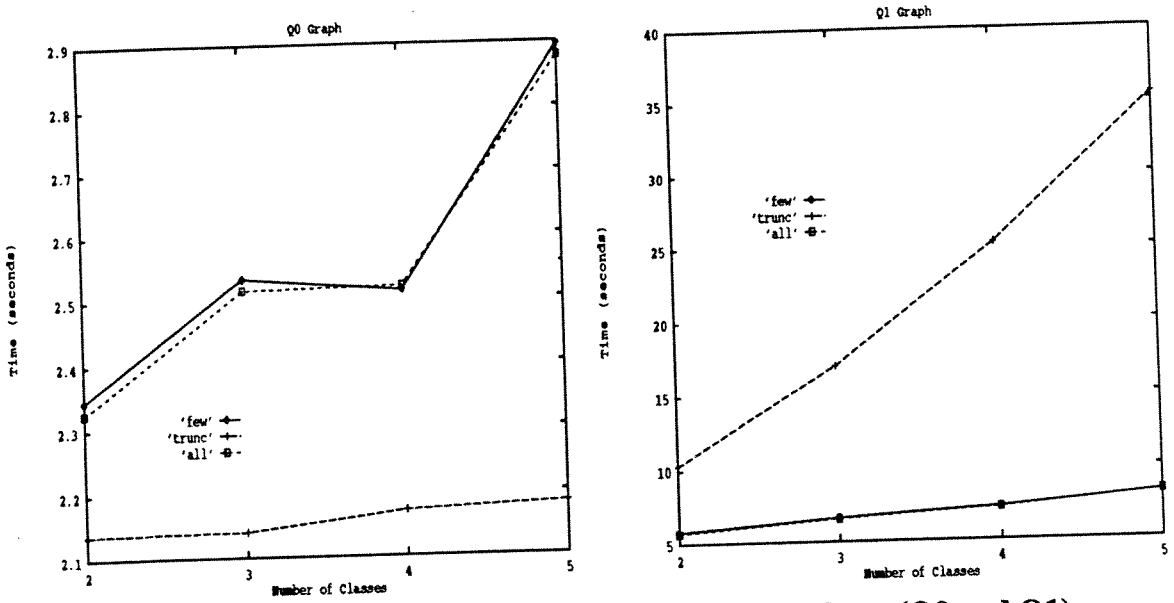

Figure 5.3 NonIndex-Processable Queries on the Leaf Class (Q0 and Q1)

Although outwardly it would seem that graphs for Q0 and Q1 should be similar, they are clearly very different. Consider the graph for Q0. `store_trunc` is by far the most efficient, as we would expect, since only one relation (containing rather short tuples 24 bytes wide) is scanned. For `store_all` and `store_few`, the size of the tuples that are being retrieved varies with the number of classes. Specifically, the length of a tuple is 64+20*n, where n is the number of classes.** The nonlinear behavior of the `store_few` and `store_all` graphs is explained by the number of blocks that must be accessed to scan the underlying relation. At n=2, four tuples can be stored inside a block of 512 bytes; at n=3 and n=4, three tuples can be stored per block; and at n=5, only two tuples can be stored per block. As the number of tuples remains constant, the stair-stepped graph merely reflects the number of blocks that are accessed in a relation scan (where the relations in cases n=3 and n=4 are packed into the same number of blocks).

** The width of an object is 60+20*n bytes. The width of its underlying tuple is 64+20*n bytes. The additional 4 bytes is the oid that is stored.

In contrast, when all attributes are to be retrieved, `store_trunc` performs the worst. The reason is that attribute data for individual objects is stored in different relations; simultaneously retrieving from different relations - in particular when there is a limited amount of buffer space - causes a significant increase in retrieval costs. On the other hand, `store_few` and `store_all` do not fragment attribute data of an individual object, thereby reducing the number of blocks that need to be in DBMS buffers at any one time. (The graph for `store_all` and `store_few` is still stair-stepped, but the effective slope is rather flat compared to that for `store_trunc`). Although the result that `store_few` can be significantly better or worse in performance than `store_all` and `store_few` is not surprising. However, we were surprised at the magnitude of the performance differences.

Queries Q2 and Q3. Figure 5.4 shows the performance graphs for queries Q2 and Q3. Both of these queries scan the topmost class of the class hierarchy; the only difference is that Q2 returns a single attribute per object whereas Q3 returns all attributes. The number of objects that are returned increases with the number of classes. (The reason is that the selectivity is fixed at .01, but the number of objects in the root is 2000*n, where n is the number of classes in the class hierarchy). It is for this latter reason that the graphs overall are linear in nature.

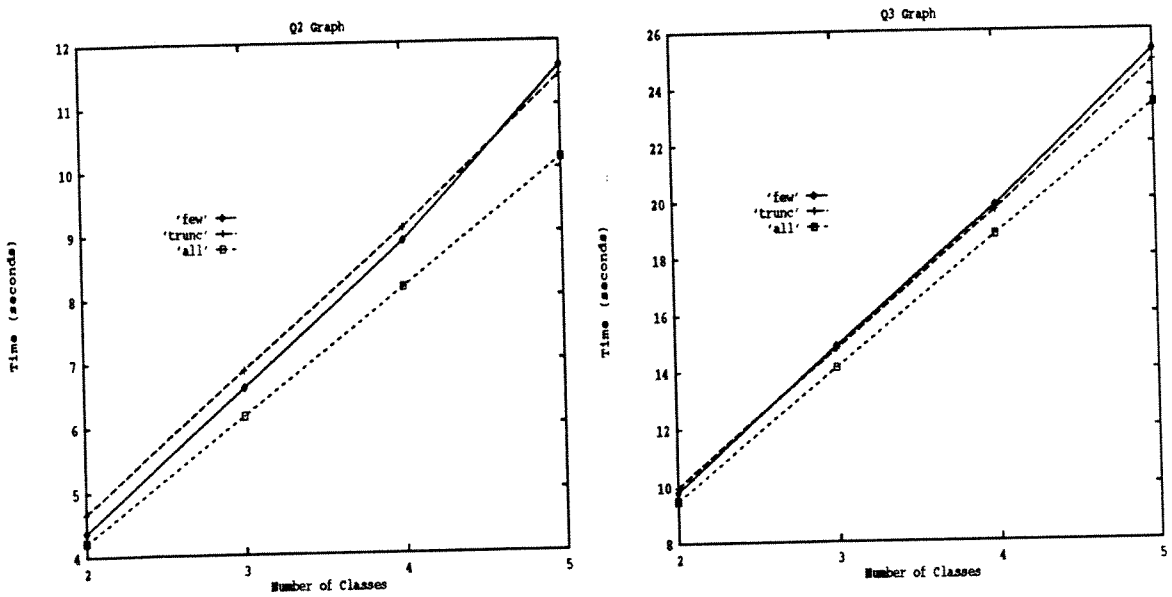

Figure 5.4 NonIndex-Processable Queries on the Root Class (Q2 and Q3)

The performance of `store_all` is slightly better than that of `store_few` and `store_trunc`. The reason is that there appears to be less overhead in initializing retrievals. As these overheads are constants, it is possible that a tuned version of these components would reduce their differences further to the point where the same basic performance would be noted. Such tuning is a subject of future work.

Queries Q4 and Q5. Figure 5.5 shows the performance graphs for queries Q4 and Q5. Both of these index-processable queries are directed at the leaf class of the inheritance lattice. The main difference between these queries is that Q4 returns a single attribute whereas Q5 returns them all. Similar to queries Q0 and Q1, the number of records returned is the

same, no matter how many classes exist in the hierarchy. (Once again, the query selectivity is constant (.01) and the number of objects in the leaf class remains fixed at 2000). Thus, we would expect performance graphs to be linear because of the random-access nature of processing queries using indices. In particular we would expect a horizontal graph (i.e., a line with no slope) at least for `store_few` and `store_all`.

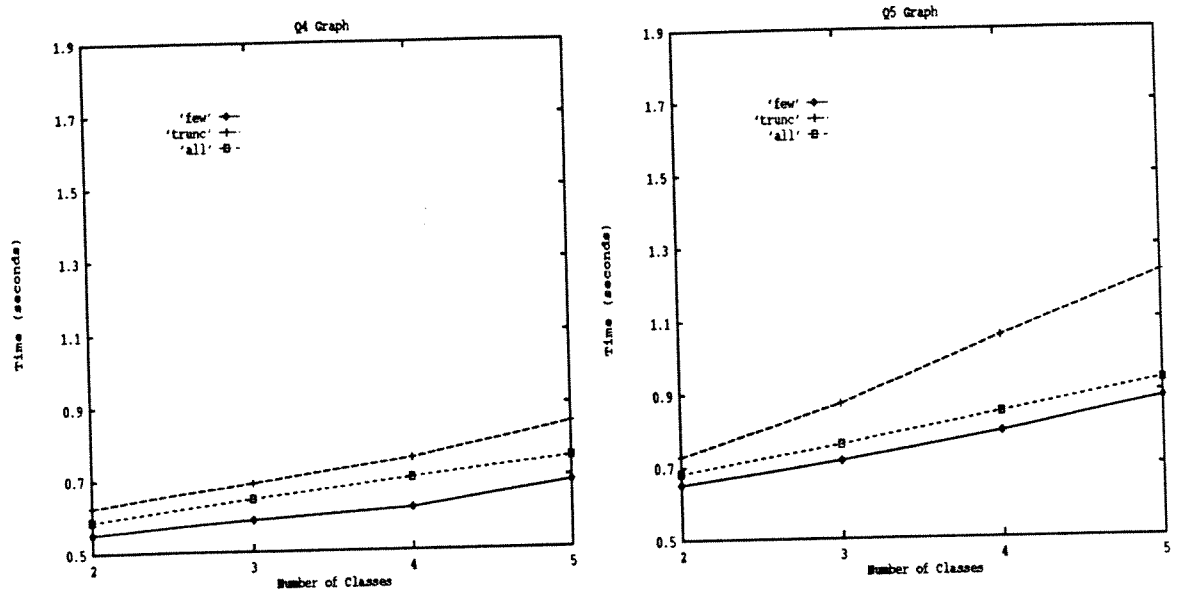

Figure 5.5 Index-Processable Queries on the Leaf Class (Q4 and Q5)

In reality, the plotted graphs have a slight slope that increases with the number of classes. This is because of three reasons: (1) the length of tuples that are stored increases linearly with the number of classes. As tuple length increases, more blocks are needed to store the relation. This, in turn, means that a block in the DBMS buffers is less likely to be needed for subsequent random accesses. The net effect is that slopes of performance graphs should increase (slightly) as the number of classes increases.

(2) Because we are dealing with class indexing, as the number of tuples increases there is a logarithmic increase in searching B+ tree index structures. And (3), the overheads for initializing a retrieval for the three different methods are different. In this particular case, initialization for `store_few` retrieval is slightly less than that for `store_all`. We expect a tuning of the `store_all` and `store_few` algorithms would minimize this disparity. `store_trunc` is the least efficient of the three. The reason is more initialization overhead and that in the case of retrieving all attributes (query Q5), multiple relations must be accessed.

Queries Q6 and Q7. Figure 5.6 shows the performance graphs for queries Q6 and Q7. Both of these index-processable queries are directed at the root class and as in the case of queries Q2 and Q3 the number of objects that are retrieved increases linearly as the number of classes increases. The difference between Q6 and Q7 is that Q7 retrieves all attributes of selected objects, whereas Q6 returns only one attribute.

The graphs for all methods are linear as expected. `store_all` is more efficient than `store_few` and `store_trunc`. The differences in performance we have seen earlier: less initialization overhead contributes a constant difference, larger tuple sizes (with

increasing numbers of classes) raises the number of blocks that are likely to be accessed, and logarithmic increases in the depth of B+ trees (with increasing numbers of classes) contribute to the non-zero slope of these graphs.

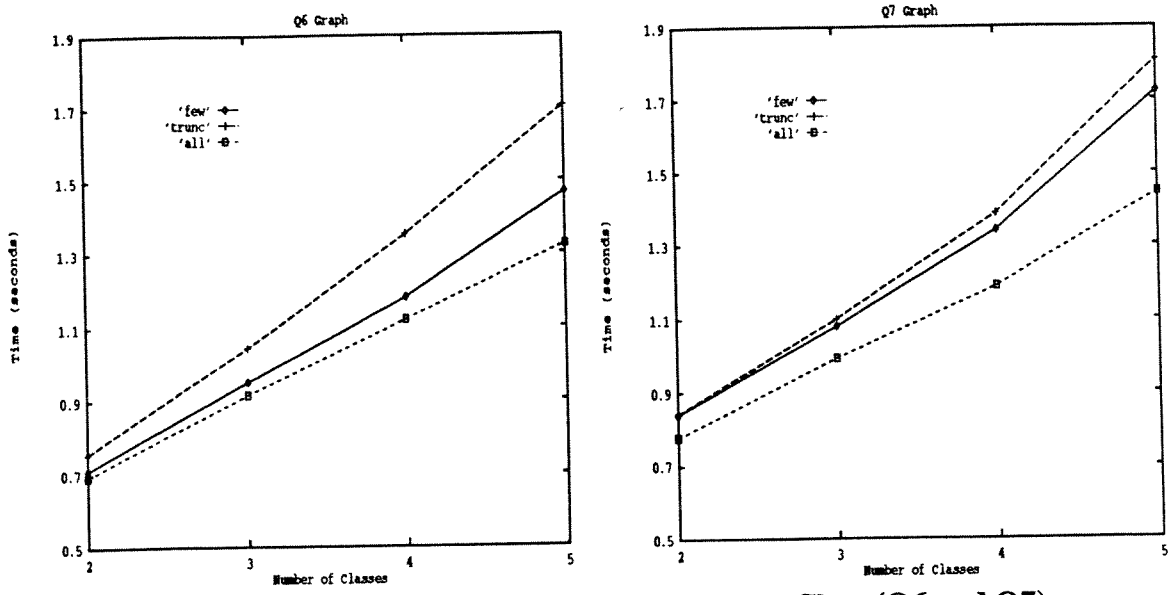

Figure 5.3 Index-Processable Queries on the Root Class (Q6 and Q7)

Summary. Software component technologies offer the possibility of easily customizing software systems to a particular application. Behind this potential lies very challenging and yet unsolved problems. On the one hand, how does one search the enormous space of possibilities to reduce the number of potential systems to examine? On the other hand, how does one actually evaluate a given assembly of components? From our experiments, while it appears that specific interactions between components are hard to anticipate, the general expectations of the behavior of specific components are reinforced.

The ultimate goal is to automate the customization of software systems given workload input. This opens up the possibility of self-tuning software systems which reconfigure themselves periodically, using actual workloads to drive the optimization. Much more work is needed to solve this interesting problem.

6. Conclusions

Large scale component reuse technologies are realizable. We created such a technology by exploiting the maturity of the domain of relational database management systems. We studied many such systems to discover a uniform and standard means by which all systems could be "conceptually" decomposed. The resulting components were primitive building blocks that could be used in the construction of many relational systems.

We presented a model of large scale component reuse that identified components with parameterized types and software systems with type expressions. We used this model to describe different libraries (called realms) of plug-compatible components that form the backbone of relational systems.

Software domains are not static, but are constantly evolving through the influx of new algorithms and ideas. Domain evolution means that new components and new realms must occasionally be introduced. This paper explained how we had to add new realms of components to be able to synthesize database systems that supported inheritance. It is well-known that the introduction of inheritance impacts many parts of a relational DBMS to the extent that not all relational components can be reused "as-is" to construct OODBMSs. Our research confirmed this finding. Moreover, the changes needed to convert relational components into their object-oriented counterparts was indeed slight (i.e., less than 10 percent of a relational component's source code was changed). More importantly, the basic software component model that we defined was invariant to these additions; the inclusion of new realms or new components did not invalidate existing components nor the methods by which software systems were constructed.

A fundamental property of software component technologies as we defined them is the ability to generate a spectrum of systems. This opens up the possibility of being able to optimize the design of a system by appropriately selecting and composing the components that are best suited for the application at hand. Another version of this problem is for a system to periodically record its workload and to automatically reconfigure and optimize itself as the workload changes. Thus, the possibility of creating "self-tuning" software systems may be a benefit of software component technologies.

We also examined the performance tradeoffs of different inheritance components under different workloads, viewing this particular study as a means to better understand the problems that will confront researchers in realizing "self-tuning" software. From our experiments, it is clear that specific interactions between components are hard to anticipate, but the basic behavior of individual components was easy to understand. Creating tools that can accurately predict the interaction of components is central to realizing "self-tuning" software systems. We believe that this will be a challenging and exciting problem for future study.

Acknowledgements. We thank Vivek Singhal, Nandit Soparkar, and Kelsey Shepherd for their helpful comments on an earlier draft of this paper. We also thank Dinesh Das for helping us create the graphs in Section 5.

7. References

[ACM91] ACM, 'Next Generation Database Systems', *Communications of the ACM*, October 1991.

[Alb85] A. Albano, L. Cardelli, and R. Orsini, 'Galileo: A Strongly-Typed, Interactive Conceptual Language', *ACM Trans. Database Systems*, June 1985.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe, *Conceptual Database Design: an Entity-Relationship Approach*, Benjamin-Cummings, 1992.

[Bat88a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, 'GENESIS: An Extensible Database Management System', *IEEE Trans. on Software Engineering*, November 1988.

[Bat88b] D.S. Batory, 'Concepts for a DBMS Synthesizer', in [Pri91] and *ACM PODS 1988*.

[Bat92a] D. Batory and J. Barnett, 'DaTE: The Genesis DBMS Software Layout Editor' in [Lou92].

[Bat92b] D. Batory and S. O'Malley, 'The Design and Implementation of Hierarchical Software Systems with Reusable Components', *ACM Trans. Software Engineering and Methodologies*, October 1992.

[Bat92c] D. Batory, V. Singhal, and M. Sirkin, 'Implementing a Domain Model for Data Structures', to appear in *Software Engineering and Knowledge Engineering*, 1992.

[Bat93] M. Sirkin, D. Batory, and V. Singhal, 'Software Components in a Data Structure Precompiler', submitted for publication. Also, TR-92-29 Department of Computer Sciences, University of Texas at Austin, May 1992.

[Big89a] T.J. Biggerstaff and A.J. Perlis, *Software Reusability I: Concepts and Models*, ACM Press, 1989.

[Big89b] T.J. Biggerstaff and A.J. Perlis, *Software Reusability II: Applications and Experience*, ACM Press, 1989.

[Bit88] D. Bitton and C. Turbyfill, 'A Retrospective on the Wisconsin Benchmark', in [Sto88].

[Fis87] D.H. Fishman, et al., 'IRIS: An Object-Oriented Database Management System', *ACM Trans. Office Information Systems*, January 1987.

[Hut91] N.C. Hutchinson and L.L. Peterson, 'The *x*-Kernel: An Architecture for Implementing Network Protocols', *IEEE Trans. on Software Engineering*, January 1991.

[Kim87] W. Kim, K-C. Kim, and A. Dale, 'Indexing Techniques for Object-Oriented Databases', TR-87-14 Department of Computer Sciences, University of Texas at Austin, 1987.

[Kor91] H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1991.

[Lou92] P. Loucopoulos and R. Zicari, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, Wiley, 1992.

[Mai86] D. Maier and J. Stein, 'Indexing in an Object-Oriented DBMS', TR CS/E-86-006, Oregon Graduate Center, 1986.

[OMa90] S.W. O'Malley and L. Peterson, 'A New Methodology for Designing Network Software', University of Arizona TR 90-29 (Sept. 1990). Submitted for publication.

[Pet90] L. Peterson, N. Hutchinson, and H. Rao, and S.W. O'Malley, 'The x-kernel: A Platform for Accessing Internet Resources'. *IEEE Computer (Special Issue on Operating Systems)*, May 1990.

[Pri91] R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Sch77] J.W. Schmidt, 'Some High Level Language Constructs for Data of Type Relation', *ACM Trans. Database Systems*, June 1977. Also in [Sch88].

[Sel79] P.G. Selinger, et al., 'Access Path Selection in a Relational Database Management System', *ACM SIGMOD 1979.* Also in [Sto88].

[Sto85] M. Stonebraker, 'Inclusion of New Types in Relational Data Base Systems', Report UCB/ERL M85/67, Electronics Research Laboratory, University of California, Berkeley, 1985. Also in [Sto88].

[Sto86] M. Stonebraker and L. Rowe, 'The Design of Postgres', *ACM SIGMOD 1986.*

[Sto88] M. Stonebraker ed., *Readings in Database Systems*, Morgan Kaufman, 1988.

[Str91] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.