Indexing in an Object-Oriented DBMS

David Maier
Jacob Stein

Technical Report CS/E-86-006
7 May 1986

Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

presented at the Workshop on Object-Oriented Databases, September 1986.

Indexing in an Object-Oriented DBMS

David Maier
Servio Logic Development Corp.
and Oregon Graduate Center

Jacob Stein
Servio Logic Development Corp.
15025 S.W. Koll Parkway, 1a
Beaverton, Oregon 97006
(503) 644-4242
CSNET: stein@oregon-grad

Abstract

We describe indexing in the GemStone object-oriented database server, which supports a model of objects similar to that of Smalltalk-80. We begin with a brief description of the system's architecture and the role of indexing in GemStone. We then discuss the properties of object-oriented systems, and GemStone in particular, that make indexing considerably different than in a more conventional data model. Various approaches to indexing in an object-oriented model are presented. We describe both the design and implementation of indexing in GemStone. Lastly, we describe related work and note performance results from the initial instrumentation of the system.

1. Introduction

The GemStone database system is the result of a development project started three years ago as Servio. Out goal was to merge object-oriented language concepts with those of database systems. GemStone provides an object-oriented database language called OPAL, which is used for data definition, data manipulation and general computation.

Conventional record-oriented database systems, such as commercial relational systems, often reduce application development time and improve data sharing among applications. However, these DBMSs are subject to the limitations of a finite set of data types and the need to normalize data [Ea, Si]. In contrast, object-oriented languages offer flexible abstract data-typing facilities, and the ability to encapsulate data and operations via the message metaphor.

Our premise is that a combination of object-oriented language capabilities with the storage management functions of a traditional data management system will result in a system that offers further reductions in application development efforts. The extensible data-typing facility of the system facilitates storing information not suited to normalized relations. In addition, we believe that an object-oriented language can be complete enough to handle database design, database access, and application coding. The GemStone data model and language take their syntax and semantics from the Smalltalk-80 system [GR, Kr]. Those readers not familiar with the Smalltalk language are directed to Goldberg and Robson [GR].

While the choice of Smalltalk as a starting point met some of the GemStone design goals, such as providing an extensible data model and a unified language for design, access and application writing, Smalltalk is by no means a database system. Smalltalk is oriented towards a single user on a dedicated processor, with data objects resident in main memory. We report elsewhere [CM, MOP, MSOP] on some of the requirements for making GemStone a multiuser disk-based system, such as concurrency, recovery, authorization and storage mangement. This paper concentrates on the particular challenges that the Smalltalk model and language present in providing associative access to objects. The remainder of this section describes the use of indexing in relational DBMSs, their analogs in the GemStone model, and the architecture of the GemStone system.

1.1. Indexing in Relational Database systems

Many relational DBMSs provide expressive power through a query language based on relational calculus [Ma]. The calculus allows the user to define the result of a query rather than tell the system how to produce the result. These relational systems can then select from a family of execution plans an efficient way to compute the desired result. The choice of plan is often influenced by the presence of auxiliary search structures such as indexes and hash tables. Indexes are especially useful when the user wishes to select a small subset of a relation's tuples based on the value of a specific attribute. In this case, a good execution plan will look up the desired attribute value in the index, then directly retrieve only the desired tuples. The presence of these search structures influences only the efficiency of producing the result, not the result itself.

1.2. Indexing in GemStone

In GemStone, the basic problem is to efficiently select from a collection those members meeting a selection criteria. We want to find all objects that either contain a given object, or an object equal to a given object, as the value of a particular instance variable. GemStone does not support direct navigation from an object O to objects for which O is the value of an instance variable. References from one object to another are uni-directional. Providing two-way links is problematical, as an object may be the value of an instance variable in several objects. For example, the same `Department` instance can fill the `worksIn` variable for many `Employee` objects. All GemStone objects are *independent*: no object's existence is constrained to depend on the existence of another object, nor can any object assume that it makes a unique reference to any of its instance variables' values.

One difference between GemStone objects and relational tuples is that objects are not flat. One should be able to index on instance variables that are nested several levels deep in an object to be indexed, such as the `manager` variable of the `Department` object that fills an `Employee`'s `worksIn` variable. An important feature of our model is that an object's identity remains the same regardless of changes in its internal state, and objects reference their components by identity, not value. Thus, the `manager` of a `Department` object can change with no change being apparent in an `Employee` object that references that `Department` object. As we shall see in Section 4, this localization of change influences the complexity of index maintenance.

While objects may be viewed as "fancy tuples" that permit attributes to have other tuples as values, it is misleading to equate relations with GemStone classes. A relation serves both to provide the scheme for its component tuples, and to collect all those tuples. In GemStone, a class defines the structure of its

To appear: International Workshop on Object-Oriented Database Systems

instances, but rarely keeps track of all those instances. Instead, collection objects — `Arrays`, `Bags`, `Sets` — serve to group those instances. An object may belong to more than one collection, unlike relational, hierarchical or network models, where a record belongs to a single relation, parent or set. Such multiple membership is allowed in the hybrid relational-network model of Haynie [Ha].

1.3. GemStone Architecture

Figure 1 shows the major pieces of the GemStone system. Stone and Gem correspond roughly to the object memory and the virtual machine of the standard Smalltalk implementation [GR]. Stone provides secondary storage management, concurrency control, authorization, transactions and recovery. Stone also manages workspaces for active sessions. Stone uses unique surrogates called **object-oriented pointers** (OOPs) to refer to objects, and an **object table** to map an OOP to a physical location. This layer of indirection means that objects can easily be moved in secondary memory. While the object table can potentially have 2^{31} entries, we expect that the portion for objects currently in use by various sessions is small enough to fit in main memory. Stone is built upon the underlying VMS file system. The data model that Stone provides is somewhat simpler than the full GemStone model, and only provides operators for structural update and access. An object may be stored separately from the objects it references, but the OOPs for the values of an object's instance variables are grouped together. Others have considered decomposed representations of objects [CFLR, Ch+, CK].

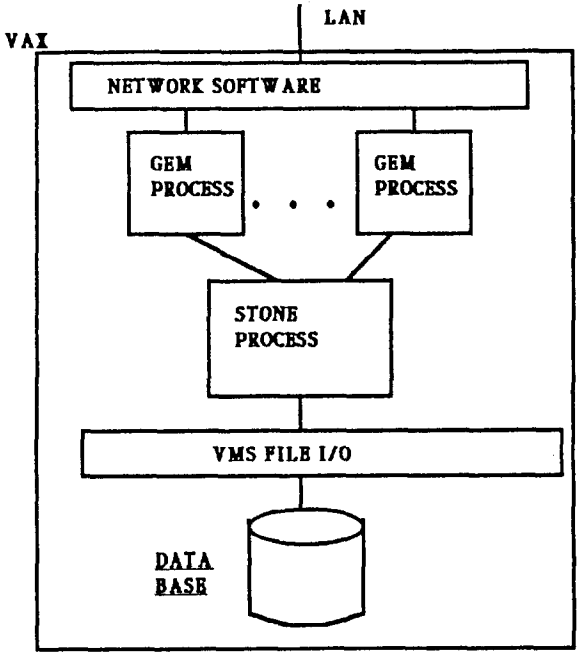


Figure 1

Stone supports five basic storage formats for objects, **self identifying** (e.g. `SmallInt`, `Character`, `Boolean`), **byte** (e.g. `String`, `DateTime`, `Float`), **named, indexed** and **non-sequenceable collections**. The byte format is used for classes whose instances may be considered atomic. The named format supports access to the components of an object by unique identifiers, instance variable names. The indexed format supports access to the components of an object by number, as in instances of class `Array`. This format supports insertions of components into the middle of an object, and can grow to accomodate more components. The non-sequenceable collection (NSC) format is used for collection classes, such as `Bag` and `Set`, in which instance variables are **anonymous**: members of such collections are not identified by name or index, but a collection can be queried for membership, and have members added, removed or enumerated. Both the indexed and NSC format support dynamic growth of objects, and are bounded in size only by the total number of objects in the system and the physical limits of secondary storage. When objects in these formats grow large, their representation switches from a contiguous one to a B-tree which maintains the members by OOP for NSC's, and by offset for indexed object. The byte format also supports dynamic growth in a manner similar to that for the indexed format. Stone groups objects into logical **segments**, which are the unit of conflict in concurrency control, and the unit of ownership for authorization.

Gem sits atop Stone, and elaborates Stone's storage model into the full GemStone model. Gem also adds the capabilities of compiling OPAL methods into bytecodes and executing that code, user authentication, and session control. (OPAL bytecodes are similar to the bytecodes used in Smalltalk.) The Gem layer contains the **virtual image**: the collection of OPAL classes, methods and objects that are supplied with every GemStone system. OPAL, being a computationally complete language, can express various associative searches on a collection, such as

```
aCollection select: aBlock
```

which returns a new collection consisting of all elements of `aCollection` for which `aBlock` returns `true`. In Smalltalk such a search is done by iterating through `aCollection` and evaluating `aBlock` for each member.

2. The Problems and Solutions

In this section we elaborate on the problems the GemStone model presents with respect to associative access and indexing. We also present possible solutions to these problems, but leave the details of the solutions actually selected to Sections 3 and 4.

2.1. Language Issues

The two issues here are when to invoke auxiliary access paths for associative searching, and whether indexes should be keyed on an object's structure or its protocol.

How should we indicate or permit the use of indexes in the OPAL language? One solution is to do nothing with the language, and simply provide system classes in the virtual image for building and using indexes. One drawback with this approach is having to ensure that every application that modifies objects also performs appropriate index maintenance. A second drawback is that having to explicitly deal with an index to support an associative search means application code no longer has physical data independence.

At the other extreme, we could go without modifications to the language, and treat every OPAL expression as a candidate for use of indexing structures in evaluation. OPAL is computationally complete; it is not just a query language. Much of the code operates on single objects, where an index either can't be applied, or has no benefit. Indexes are helpful only for operations iterated over members of a large collection. Identifying appropriate iterations for indexing, and determining the intent of the iteration at a high enough level to permit code transformations is a challenging problem in data flow analysis.

Some more moderate positions are to designate certain messages as the only ones for which index use will be attempted, and scan methods for occurrences of those messages, or to add a data sublanguage to OPAL for expressing associative searches, and tailoring the sublanguage to make use of indexes. Adding a sublanguage complicates the language and its compiler. It also introduces the danger of an "impedance mismatch": the sub-

language will be incompatible with the main language as regards data structures or the processing paradigm [CFLR, MO, MP, RS]. If the sublanguage route is chosen, the question is, what kind of sublanguage? It could be a declarative language — an analog to relational calculus. We note that GemStone already has a declarative flavor to it already. Messages to objects say *what* to do; it is left up to the object which method to use — the determination of *how* the message should be performed.

A calculus-like language could support associative searching, extraction of subparts of objects and creation of new objects as the answer to a query. One problem here is, of what class are the resulting objects? Allowing a query to assemble new objects arbitrarily would mean creating classes on the fly. Another problem is whether variables in calculus queries range over classes or collections. Allowing a variable to range over all instances of a class means that a collection of those instances must be maintained. The collection of all instances of a class is not necessarily a sematically useful entity for querying. Usually it is collections of a subset of all instances that are of interest. Also, how do we deal with the situation where a user is only authorized to access some of those instances? Having variables range over collections introduces a problem with binding the query to the specific collections. If the actual collections are not determined until run time, little preprocessing of the query can take place.

A disadvantage to a full calculus sublanguage is that a query must be translated before it can be processed. Being able to interpret a query as regular OPAL code and evaluate it by brute force is useful for avoiding the translation overhead on queries involving small numbers of objects, and to have a "semantic benchmark" for validating associative access routines. A more restricted declarative language might support only selection, avoiding the need to create new classes for the results of queries. The selection conditions could be arbitrary blocks of OPAL code, or have some restrictions. A problem with arbitrary blocks of code is that such code can have side effects on the objects being examined. Side effects make it hard to ensure that evaluating a query with and without an index gives the same answer.

A sublanguage could be more procedural, but still encapsulate iteration — an algebra for collections of objects. Some of the required operations are already present in GemStone, mainly Boolean operations on sets. We have the same problem with join operations as we had with a full calculus language: having a class for the result. While algebras have been proposed for non-1NF relations, we have yet to see a workable algebra for complex objects with identity. In particular, there are semantic difficulties with shared instance variable values, cycles of objects and value-based versus identity-based comparisons.

The other major issue regarding languages is whether indexes are based on the structure — the instance variables — of objects, or the protocol — the responses to messages. For example, if `anEmp` is an `Employee` object, we could access that employee's last name with some kind of structural notation, such as

```
anEmp.name.last,
```

where `name` and `last` are instance variables, or we can use a message notation

```
anEmp name last,
```

where `name` and `last` are unary messages. If indexes are based on message notation, we must know which structural changes in an object can influence the result of a message, so that we know when to update the appropriate indexes. Also, a method can change the state of an object, and we need assurances that a message will yield the same answer twice in a row if the structure of the object has not been explicitly changed. The method for a mesage can be overridden in a subclass, which presents problems in allowing kinds of[1] a class into a message-based index along with instances of the class. One problem of other models for message-based indexing not present in GemStone is attributes inherited through the "component-of" hierarchy [Br+]. For example, the absolute position of a machine part could be computed from its relative position to the assembly that contains it, which makes an index on absolute position prohibitive to maintain. OPAL methods may access an object's instance variables, but may not directly access objects that contain it as the value of an instance variable.

Indexing based on structure has the advantage that it can be supported at the Stone level, while message-based indexing requires access to the execution model at the Gem level. Indexing on structure violates the privacy of objects, as it bypasses an object's protocol. One could view an index on an object as being part of the implementation of the object's class, and hence being privy to the internal structure of the object. In our experience, most user-defined classes include methods for accessing each named instance variable anyway.

2.2. Index Structure

If we want to index objects on their internal structure, one question is, how deep to index? Do we index only the immediate instance variables of an object, or do we allow indexes on instance variables of instance variables? With a one-level index, we always have the object in hand when a change that can affect its position in an index occurs. With a multilevel index, as `anEmp.worksIn.manager`, we have the problem that an object's position in an index can be invalidated by a change in a subobject that is not manifested in the object itself. (A `Department` gets a new `manager`.)

If we do index on paths with multiple links (multiple instance variables), we have the choice of a single index for the whole path, or several indexes, one for each link. For `anEmp.worksIn.manager`, we could have one index on `worksIn.manager` mapping managers directly to the employees they manage, or we can have one index on `worksIn` mapping departments to employees, and another on `manager`, mapping managers to departments. With a single index on the entire path, there are fewer indexes to maintain, and fewer consultations needed for an associative lookup. However, not all the indexes for link indexing will have as many entries as the index for the entire path. One thousand `Employees` may reference only twenty different `Departments` in their `worksIn` field.

Indexing by links means prefixes of a path are indexed as well: indexing ability on `anEmp.worksIn.manager` implies ability on `anEmp.worksIn`. Supporting a path index as multiple link indexes also allows sharing between path indexes with a common prefix, such as `anEmp.worksIn.manager` and `anEmp.worksIn.division`.

Since GemStone associates types with objects rather than identifiers, we can not tell a priori that an object supports a certain path, or the class of the object at the end of the path. In a collection of `Employee` objects, if we want to create an index on `anEmp.name.last` we need to know that every object in the collection has such a path. That is, that the value of the `name` instance variable is an object that contains a `last` variable. Further, if the index is to be ordered, we need to know that the `last` variables of elements hold comparable values, such as `Strings`. We need typing on collection elements and instance variables to support indexes, unless we want to deal with the complication that not all collection elements will be indexed.

Additionally, we need to consider the following questions on indexing strategy:

[1] A object *O* is a kind of its class and its class's superclasses.

1. What do we use as types: classes, kinds or some sort of structural or operational template? Classes as types are easiest to check at the Stone level, since an object carries a reference to its class. A kind (a class plus all of its subclasses) requires access to the class hierarchy for checking, but seems more natural for most applications.

2. Is `nil` a member of every type? What about `nil` values along a path? Should they be disallowed? Should selection conditions be given the semantics that the existence of the path is presumed? That is, should both

```
anEmp.name.last = 'Ross'
```

and

```
anEmp.name.last ~= 'Ross'    (not equal)
```

be false if `anEmp.name` is `nil`? We could interpret the value of any path with `nil` along it as `nil` for the whole path (in which case the second comparison above will succeed).

3. Should an index be based on identity of key objects or their values? An identity index is immune to changes in the key object's state. However, an identity index on `Strings` will not support range queries. On the other hand, if we build an index on `String` objects sorted on their contents, we must detect the case where some method changes the characters of one of those `Strings`. An alternative to detecting changes is to disallow them by constructing immutable subclasses of objects in which state change is not allowed after creation.

4. If range indexes are allowed, what comparison operators are allowed for the sort order? If the programmer supplies a method for the comparison, how do we know it is transitive? Suppose we type by kinds and the comparison method is overridden in a subclass?

2.3. Indexing on Classes versus Collections

Another decision in designing associative access is what to index, classes or collections? Several applications can use instances of the same class, and store them in different collections (like having several relations on the same scheme [Ha]). Indexing on the class requires applications that do not use the index to still bear index-related overhead for indexed instances that they use. Further, a classwide index presents authorization problems. No one user may have read access to the set of all instances of the class, so no one is able to request that the index be created. Also, indexing a collection allows the possibility that instances of subclasses be included in a collection that is indexed. Indexing on a class basis makes it easier to trace changes to the state of an object that could cause the object to be positioned differently within an index. We can flag a class-defining object to indicate which instance variables are indexed, and trap to index maintenance routines in Stone.

As an object may participate in many collections, if we index on a class basis, but pose queries against collections, there will be a test for collection membership needed in addition to the the index access. On the other hand, if we index by collections, and use references from objects to indexes to support update, each object must be able to reference a number of indexes, not just one. Of course, even if indexing is on collections, it is possible to have a particular class collect all of its instances. If we do index on classes, there is a question of whether a class indexes all the instances of its subclasses as well, or if each subclass must maintain its own index. The latter course probably has unacceptable overhead for a class with more than just a few subclasses.

There is middle ground. We can maintain a single index (per instance variable) per class, but only add members of selected collections to that index [Pu]. With this hybrid scheme, a collection knows if it is indexed, and informs the appropriate class when it adds or deletes elements. However, every instance of the class still pays a penalty on update, as it must be checked for membership in one of the indexed collections.

3. Path Expressions and Typing in OPAL

In this section and the following one, we outline the actual choices made on language design and indexing strategy in GemStone. In order to facilitate associative access, both paths and instance variable typing have been introduced into OPAL.

3.1. Path Expressions

A **path expression** (or simply a **path**) is a variable name followed by a sequence of zero or more instance variable names called **links**. The variable name appearing in a path is called the **path prefix**; the sequence of links, the **path suffix**. The value of a path expression $A.L_1.L_2.\cdots.L_n$ is defined as follows:

1. If $n=0$, then the value of the path expression is the value of A.

2. If $n>0$, then the value of the path expression is the value of instance variable L_n within the value of $A.L_1.L_2.\cdots.L_{n-1}$ if $A.L_1.L_2.\cdots.L_{n-1}$ is defined and L_n is an instance variable in the value of $A.L_1.L_2.\cdots.L_{n-1}$. Otherwise, the value of the path expression is undefined.

A path suffix S is **defined with respect to** a path prefix P if the value of $P.S$ is defined.

Consider a variable `anEmp` whose value is an instance of `Employee`. The value of the path `anEmp.name` is defined if `name` is an instance variable defined in `Employee`. Its value would be the value of `anEmp`'s `name` instance variable. The value of `anEmp.name.first` is defined if the value of `anEmp.name` is defined and `first` is an instance variable in the value of `anEmp.name`. Its value would be the value of instance variable `first` in the value of `anEmp.name`.

Path expressions may be used anywhere in OPAL that an expression is allowed. The evaluation of a path expression is relatively straightforward, and follows directly from the definition above. It should be noted that determining whether an instance variable is defined within an object requires accessing the object's class, and a list of instance variable names on which string comparisons must be made. Unary messages that return the value of an instance variable are optimized in GemStone. Thus, while path expressions can be used apart from associative access, such use is less efficient than the equivalent sequence of unary messages.

In general, given two objects of the same class, A and A', we can not infer that a path suffix is defined with respect to A' from the fact that the suffix is defined with respect to A. However, consider instances of `Employee` objects. If we knew that the class of the value of instance variable `name` were the same in all objects of class `Employee`, and that the path suffix `name.first` is defined for any object of class `Employee`, then we would know that the suffix is defined for all `Employee` objects.

3.2. Typing

In OPAL, constraints on the values of named instance variables may be specified when creating classes. For each named instance variable defined in a class C, a class that constrains the allowable values for the instance variable in instances of C may be specified. The specified constraining class is known as the instance variable's **class-kind**. In an object of class C, each named instance variable, for which a class-kind is specified in C,

may only have a value that is either `nil` or a kind of the class-kind specified for the instance variable in *C*. One may think of a class-kind constraint being specified for every named instance variable in every class, where the default class-kind constraint is class `Object`.

Consider class `Employee` discussed above. If instance variable `name`'s class-kind is `PersonName`, and instance variable `first` is defined in `PersonName`, then in any `Employee` object *anEmp* where *anEmp*.`name` is not `nil`, the path suffix `name.first` is defined.

Class-kind constraints are inherited through the class hierarchy. While class-kind constraints may not be removed in a class's subclasses, they can be made more restrictive. For example, in `ClassifiedEmployee`, a subclass of `Employee`, instance variable `name`'s class-kind could be `ClassifiedPersonName` if `ClassifiedPersonName` were a subclass of `PersonName`. Within `ClassifiedPersonName`, instance variable `first`'s class-kind could be `InvariantString` since `InvariantString` is a subclass of `Object`.

A path expression is **constrained** when the class-kinds of all the links in the suffix can be inferred. The **class-Kind** of a constrained path is the class kind of the last link in the path's suffix. More formally, a path expression with no suffix is always constrained. The class-kind of a path *A* is the class of object *A*. A path $A.L_1.L_2.\cdots.L_n$ is constrained if $A.L_1.L_2.\cdots.L_{n-1}$ is constrained, and within the class-kind of $A.L_1.L_2.\cdots.L_{n-1}$ a constrained instance variable L_n is defined. The class-kind of $A.L_1.L_2.\cdots.L_n$ is the class-kind of L_n in the class-kind of $A.L_1.L_2.\cdots.L_{n-1}$. A path expression $A.L_1.L_2.\cdots.L_n$ is **partially constrained** if $A.L_1.L_2.\cdots.L_{n-1}$ is constrained. (There is no class-kind for a partially constrained path that is not also constrained.) A path suffix *S* is **(partially) constrained with respect to** a path prefix *P* if *P.S* is a (partially) constrained path.

Note that if *S* is (partially) constrained with respect to *P* and *P'* is an object of the same class as *P*, then *S* is (partially) constrained with respect to *P'*. This observation allows us to say that a path suffix is **(partially) constrained with respect to** a class *C* if it is (partially) constrained with respect to any, and therefore all, objects of class *C*. Furthermore, given the inheritance of instance variable constraints, if a path is (partially) constrained with respect to a class, then the path is (partially) constrained with respect to *all of the class's subclasses*.

In summary, a constrained path always leads to an object that is either `nil` or of a certain kind; a partially constrained path always leads to an object, but we do not know what kind of object it leads to.

Allowing `nil` to be the value of a constrained instance variable slightly complicates the notion of a path suffix being defined. To overcome this complication, we introduce an object `undefined` and redefine the value of a path expression $A.L_1.L_2.\cdots.L_n$ whose suffix is partially constrained with respect to its prefix as follows:

1. If $n=0$, then the value of the path expression is *A*.
2. If $n>0$, then if the value of $A.L_1.L_2.\cdots.L_{n-1}$ is `nil` or `undefined`, the value of the path expression is `undefined`. Otherwise, the path expression's value is that of instance variable L_n in the value of $A.L_1.L_2.\cdots.L_{n-1}$.

That the above definition is well formed follows directly from the fact that for any partially constrained path $A.L_1.L_2.\cdots.L_n$, if the value of $A.L_1.L_2.\cdots.L_{n-1}$ is neither `nil` nor `undefined`, then L_n is an instance variable in the value of $A.L_1.L_2.\cdots.L_{n-1}$. What distinguishes a path whose value is `undefined` from one whose value is `nil` is that in the former case the path can not be fully traversed, while in the latter, the path can be traversed, and leads to the value `nil`.

Among `Collection`'s subclasses, class `Bag` and its subclasses are unordered, containing only anonymous instance variables. These are the non-sequenceable collection classes (NSCs) introduced in Section 2, which provide relatively fast identity-based membership, union, intersection and difference operations.

A class-kind constraint may be specified for an NSC class. An NSC may only contain `nil` and objects that are a kind of the class-kind specified in the NSC's class. One may view the class-kind constraint specified for an NSC class as a constraint on the anonymous instance variable. The class-kind of an NSC object is the class-kind specified in its class. If a path suffix is (partially) constrained with respect to the class-kind of an NSC, then the suffix is **(partially) constrained with respect to** both the NSC and its class.

Consider class `Employee` discussed above. By creating a subclass of `Bag` or `Set` whose class-kind is `Employee`, NSC's can be created that contain only `nil` and objects whose class is a kind of `Employee`.

Class-kind constraints for NSCs are inherited throughout the class hierarchy. In the same manner as for named instance variables, class-kind constraints can be made more restrictive in subclasses of an NSC class. For example, given a class `SetOfEmployee` whose class-kind is `Employee`, a subclass of `SetOfEmployee`, say `SetOfClassifiedEmployee`, can be created whose class-kind is `ClassifiedEmployee`, since `ClassifiedEmployee` is a subclass of `Employee`. Note that if a path suffix is (partially) constrained with respect to an NSC class *C*, then the path suffix is also (partially) constrained with respect to all subclasses of *C*.

4. Indexing in OPAL

4.1. Design Considerations

In OPAL, indexes index into NSC's, and are only allowed on constrained and partially constrained paths. By so restricting indexing, the access path that an index represents can be determined at the time of index creation, using only class objects; there is no need to recompute the access path represented by a constrained or partially constrained path expression for each element of an NSC.

OPAL supports two kinds of indexes: identity and equality indexes. Since the identity of an object is independent of its class, identity indexes may be created on partially constrained paths and support the operators == (identical to) and ~~ (not identical to). Equality indexes support the operators =, ~=, <, <=, > and >=. In order to support these operators, the structure (class-kind) of indexed path values needs to be known. For this reason, equality indexes may be created only on constrained paths. Furthermore, in order to avoid the interpretive execution of the operators supported by equality-indexes, the class-kind of equality indexed paths is restricted to `Boolean`, `Character`, `DateTime`, `Float`, `Integer`, `String`, `SmallInteger`, and subclasses thereof. Note that `undefined` is equal and identical to only `undefined`, and is not less than or greater than any object.

The user needs to be aware that if he changes the meaning of one of the supported operators for a subclass of one of the above classes, equality indexes will not support the modified meaning of the operator. However, we consider the likelihood of such modifications low, and in any event, believe that the vast

majority of applications will use the default meanings of these operators with respect to the class-kinds upon which equality indexes may be built. Furthermore, the system administrator can, if desired, protect the methods that implement these operators from being overwritten.

Consider class `Employee` discussed above. In addition to instance variable `name`, let `address` be an instance variable defined in `Employee` that is constrained to `Address`. Further, in `Address` let `state` be an instance variable constrained to `String` and let `zip` be an instance variable constrained to `SmallInteger`. In `SetOfEmployee` objects either identity or equality indexes can be created on the suffixes `name.first`, `address.state` and `address.zip`. Identity indexes can be created on `address`, `name`, the empty path and any other path that is partially constrained with respect to `SetOfEmployee`.

Even in the absence of indexes, OPAL takes advantage of constrained and partially constrained paths in evaluating queries against NSCs. By being able to apply the same access strategy to each element of an NSC for a given path and being able to evaluate the comparison operator without the use of message sends, terms that use a constrained or partially constrained path and an operator that the path supports can be evaluated efficiently.

For `Boolean`, `Character` and `SmallInteger` class-kinds there is no distinction between equality and identity indexes. The operators supported by equality indexes may be applied to objects of these classes by applying the operator to the OOPs of the objects directly. Therefore, when either an equality or an identity index on a path whose class-kind is a kind of one of these classes is created, an identity index is created on the path and is used to support the equality operators. An identity index on a path whose class-kind is `SmallInteger` also supports equality operators, whereas an identity index on a path whose class-kind is `Integer` does not support equality operators. Additionally, an equality index on a path whose class-kind is `SmallInteger` has a more efficient implementation that an equality index on a path whose class-kind is `Integer`.

4.2. Implementation

Indexes on paths are implemented by a sequence of index components, one for each link in the path suffix. For an index into a `SetOfEmployees` object on `name.first`, there would be an index component from `name` values of elements of the `SetOfEmployee` object to elements of the `SetOfEmployee` objects, and a component from `first` values of `name` objects to `name` objects that are values of elements of the `SetOfEmployee` object. By our method of implementing indexes, creating either an identity or equality index on a path suffix $L_1.L_2.\cdots.L_n$ implicitly creates $n-1$ identity indexes on $L_1.L_2.\cdots.L_i$, for $1 \le i < n$.

In describing the implementation of indexes in OPAL, we shall make use of the classes described in Figure 2. In this figure, a Pascal-like notation is used for class definitions. In `Address`, `state` is an instance variable whose class-kind is `String`. Within objects, named instance variables are accessed by their offsets within the objects. In the declarations of Figure 2, the offset of a named instance variable corresponds to the order in which it is declared. The offset of instance variable `state` in `Address` objects is one; the offset of instance variable `zip`, two. The offsets of instance variables inherited from a superclass are the same as in the superclass.

All data structures used in implementing indexes are stored in object space, and so are managed by Stone. However, they are objects that are not directly accessible to the user. In this manner, OPAL's concurrency control mechanism handles concurrency conflicts on index structures.

```
Class

Name:
  first, last: String;
        ⋮

END;

Address:
  state: string;
  zip: smallInteger;
        ⋮

END;

Employee:
  name: Name;
  address: Address;
        ⋮

END;

EmployeeBag: BAG OF EmployeeClass;
```

Figure 2

Every NSC object has a named instance variable, `NSCDict`, that is not accessible to the user. If there are no indexes into an NSC, then the value of `NSCDict` is `nil`; otherwise, the value of `NSCDict` is the OOP of an **index dictionary**. An index dictionary contains the OOPs of one or more **dictionary entries**.

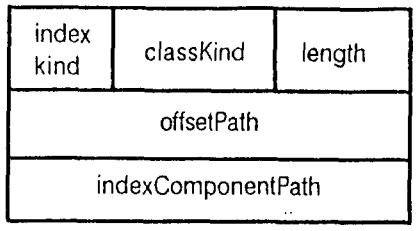
The structure of a dictionary entry is given in Figure 3. The `indexType` field represents the kind of the index, either identity or equality. The `classKind` field is significant only for equality indexes, and stores the class-kind of the indexed path. The `length` field stores the length of the path suffix. Currently, indexes may be created on paths whose suffix contains at most sixteen links. The fields `offsetPath` and `indexComponentPath` are two parallel arrays. The `offsetPath` field contains an offset representation of the path suffix. For example the offset representation of the suffix `address.state` with respect to `EmployeeBag` is (2,1). The `indexComponent` field contains the OOP of the **index component** for each instance variable in the path suffix.

The structure of an index component is given in Figure 3. Currently, all index components are implemented using B^+-trees. The **treeRoot** field contains the OOP of the root of the B^+-tree of the component. Given the operators supported by identity indexes, it would be preferable to use linear hashing for the components of identity indexes. Unfortunately, this would disallow the sharing of components between identity and equality indexes, and prevent identity indexes on `Boolean`, `Character` and `SmallIntegers` from supporting the operators of equality indexes.

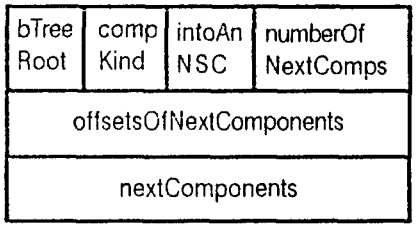
The `compKind` field defines the ordering of keys in the component's B-tree. For all components but the last, the ordering is defined on the OOPs of key values. For the last component of an identity index, the ordering is also on the OOPs of key values. For the last component of an equality index, the ordering of key values is determined by the class-kind of the indexed path, and may be a `DateTime`, `Float`, `Integer` or `String` ordering.

The field `intoAnNSC` is `true` if the component is the first of an indexed path, and therefore, indexes directly into an NSC.

This field is used to determine if duplicate entries will be permitted in the B-tree, and is discussed further below:



dictionary entry



index component

Figure 3

If the path suffixes of two or more indexes into an NSC have a common prefix, then the indexes will share the index components on the common prefix. For example, if there were `address.state` and `address.zip` indexes into an `EmployeeBag` object, then both indexes would share the component from `Address` objects to elements of the NSC object. Note that this sharing would occur regardless of the kinds of the indexes. The `numberOfNextComps` field stores the number of indexed paths that share the component.

The parallel arrays `nextComponents` and `offsetsOfNextComponents` store the offset for, and OOP of, the index component for the next link in each indexed path that shares the component. We shall refer to the elements of the `nextComponents` field as **next-components.**

Objects in GemStone may be tagged with a **dependency list.** For every index component in which an object is a value in the component's B-tree, the object's dependency list will contain a pair of values consisting of the OOP of an index component and an offset. The pair indicates that if the value at the specified offset is updated then an update must be made to the corresponding index component. We say an index component is **dependent** on the value of the object at the given offset. Additionally, objects that appear as key values of the last component of an equality indexed path whose class-kind has a byte storage format (i.e. `Character`, `DateTime`, `Float`, `Integer`, `SmallInteger` `String`, and subclasses thereof) will have a dependency list consisting of the OOPs of index components that must be updated if the value of the string is modified.

4.3. Index Maintenance

Below, we consider the operations of index creation, index removal, NSC insertion, NSC deletion, and object modification. For each operation, we shall consider examples based upon indexing into an `EmployeeBag` object.

4.3.1. Index Creation

Figure 4 shows the dictionary structure for an `EmployeeBag` object with no extant indexes after an equality index on `name.last` has been created. The NSC's dictionary contains a single entry. The corresponding dictionary entry indicates that it is an equality index on a path suffix of length two and class-kind `String`. The first `offsetPath` entry of offset 1 indicates that key values for the first component of the indexed path may be found at offset 1 in objects belonging to the NSC. The second entry indicates that key values for the second component may be found at offset 2 within key values of the first component.

equality index on name.last

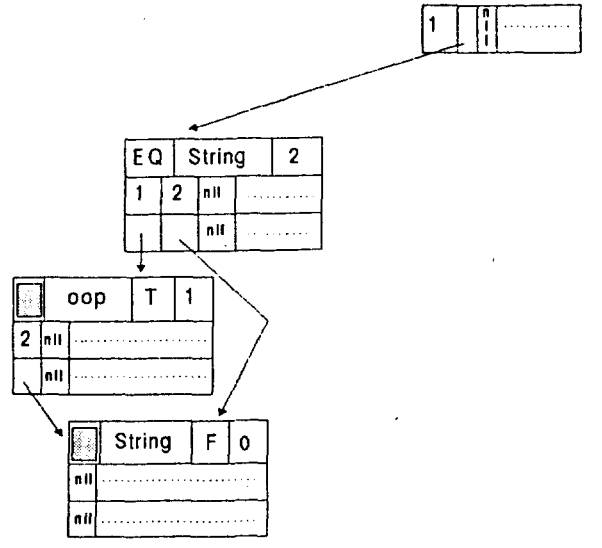


Figure 4

The first `indexComponentPath` entry is the OOP of an index component that uses `OOP` ordering in its B-tree, is into an NSC, and has one next-component. The first `offsetsOfNextComponents` entry indicates that key values for the first and only next-component are located at offset 2 within key values of the current component. The first `nextComponents` entry is the OOP of the first next-component. The second index component uses `String` ordering in its B-tree, is not into an NSC, and has no next-components.

The first index component's B-tree will have an entry for every element of the indexed NSC other than `nil`. (The effect of not having index entries for `nil` are discussed in Section 4.4.) Duplicate key, value-entries will be present for every `Employee` object that is present more than once in the NSC. This duplication allows us to use the index for a lookup without refering to the indexed NSC in order to determine the number of occurrences in the NSC of objects in the result of the lookup. The dependency lists for non-`nil` elements of the NSC will have entries indicating that the index component must be updated if the value at offset 1 is modified. `Nil` never has a dependency list as it has no instance variables and does not have a byte implementation.

The B-tree of the second index component will contain exactly one entry for each unique (by identity), non-`nil` `name` value of an element of the NSC. The dependency lists for each non-`nil` `name` value of an element of the NSC will have an entry indicating that the index component must be updated if the value at offset 2 is modified. Additionally, those strings that are keys in the B-tree

will have dependency list entries indicating that the index must be modified when the string is.

The dictionary structure after an equality index on `address.state` has been added is shown in Figure 5. The dictionary structure after an equality index on `address.zip` has been added in shown in Figure 6. Both of these indexes share the component that indexes from `address` values to elements of the NSC. The creation of the index on `address.zip` does not require updating the B-tree of this component. The component now has two next-components. An insertion into the B-tree of the new component is made for each unique, non-nil key value in the first component. Note that the new index component is implemented as an identity index.

4.3.2. Index Removal

Consider removing the index on `address.state` from the dictionary structure of Figure 6. Since the first component of the index is used by another indexed path, only the second component should be deleted. In deleting the component, the entry that refers to the component must be removed from the dependency list of every object that appears as a value in the component's B-tree. Since the component is an equality component of class-kind `String`, the dependency list entry that refers to the component must be removed from every object that appears as a key value in the components's B-tree. The first component needs to be modified to indicate that it has only one

+ equality index on address.state

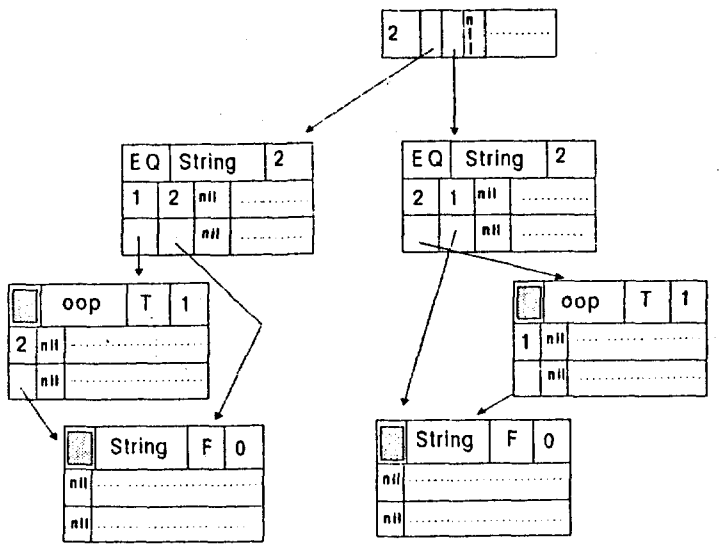


Figure 5

+ equality index on address.zip

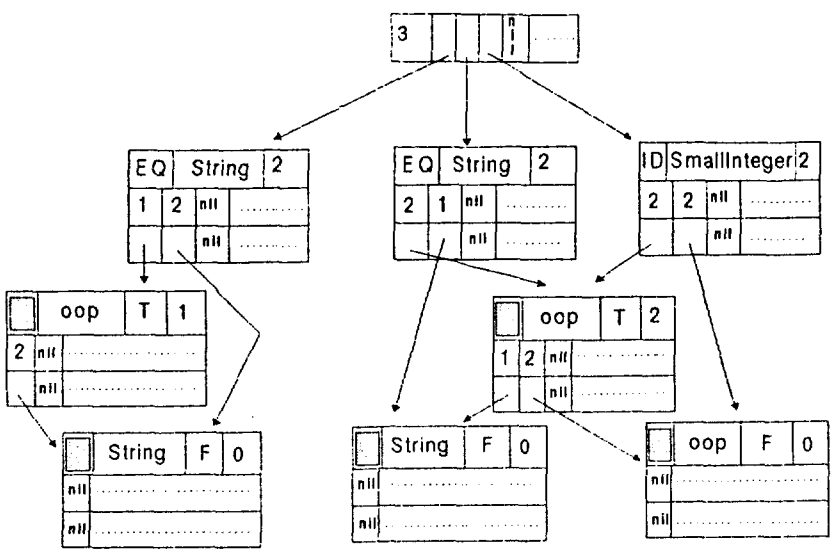


Figure 6

next-component. The resulting dictionary structure is shown in Figure 7.

4.3.3. NSC Insertion

If an object other than `nil` is inserted into an NSC, the index dictionary and each of the dictionary's entries is examined. For each unique first element of an `indexComponentPath`, an insertion is made into the component. This operation involves an insertion into the component's B-tree, and an insertion into the dependency list of the object added to the NSC.

When an insertion whose key value is non-`nil` is made into an index component, its B-tree is checked for the presence of the key value. If the key value is not found, then insertions are also made into the next-components.

Consider the insertion of an object whose `name` value is `nil`, whose `address.zip` value is 11598, and whose `address.state` value is `nil`, into an NSC with the dictionary structure of Figure 6. An insertion will be made into the first component of the index on `name.first`. The insertion will not propagate into the second component since the path suffix `name.first` is undefined with respect to the inserted object.

An insertion will be made into the first component that is shared by the remaining two indexed paths. If this is the first insertion into the component's B-tree for an employee with this address, then the insertion will propagate to both next-components.

4.3.4. NSC Deletion

If an object other than `nil` is deleted from an NSC, then the index dictionary and each of its elements is examined. For each unique first element of an `indexComponentPath`, a deletion is made from the component. When the deletion of the last occurrence of a key value is made, deletions propagate to the component's next-components.

Consider the deletion of the object inserted above. A deletion will be performed on the first component of the indexed path `name.first`. The deletion will not propagate to the next-component since the key value deleted was `nil`. A deletion will be made from the first component shared by the remaining two paths. If the deletion leaves no other entries with the same key value as the one deleted (if no remaining employees have the identical address), then the deletion is propagated to the two next-components for `State and Zip`.

4.3.5. Object Modification

When the value of an object at a given offset is modified, then a deletion followed by an insertion is made for each index component that is dependent upon the value of the object stored at that offset. When the component is not the first component of an `indexComponentPath` (when `intoAnNSC` is `false`), the deletion of single entry followed by the insertion of a single entry for each dependent component will do. (Note that an index component can't be a first component for one path and non-first for another.) If the dependent component references an NSC, then every occurrence of the object, old value pair in the component's B-tree must be deleted. If n occurrences are deleted then n occurrences of the object, new value pair are inserted. The propagation of these insertions and deletions is handled in the same manner as described for NSC insertion and deletion.

When a byte object with a non-nil dependency list is modified each index component on its dependency list is modified. Each entry in a dependent component's B-tree with a key value identical to the byte object is deleted from the B-tree. After the modification, each of the deleted entries is reinserted.

4.4. Indexed Lookups

Identity indexes directly support identity (==) lookups. Equality indexes and identity indexes on `Boolean`, `Character` and `SmallInteger`, directly support =, >, >=, <, <= and range lookups. The only differences between evaluating these lookups is in the initial access to the last index component of an indexed path. The evaluation of an indexed lookup begins with a B-tree lookup in the last index component of the indexed path's index component path. If the indexed path is of length one, then the lookup is complete. Otherwise, the following sequence is repeated $n-1$ times for an indexed path of length n. Sort the result of the previous B-tree lookup by OOP. Using the sorted list of OOPs, perform a lookup on the B-tree of the previous index component for the preceding link in the path.

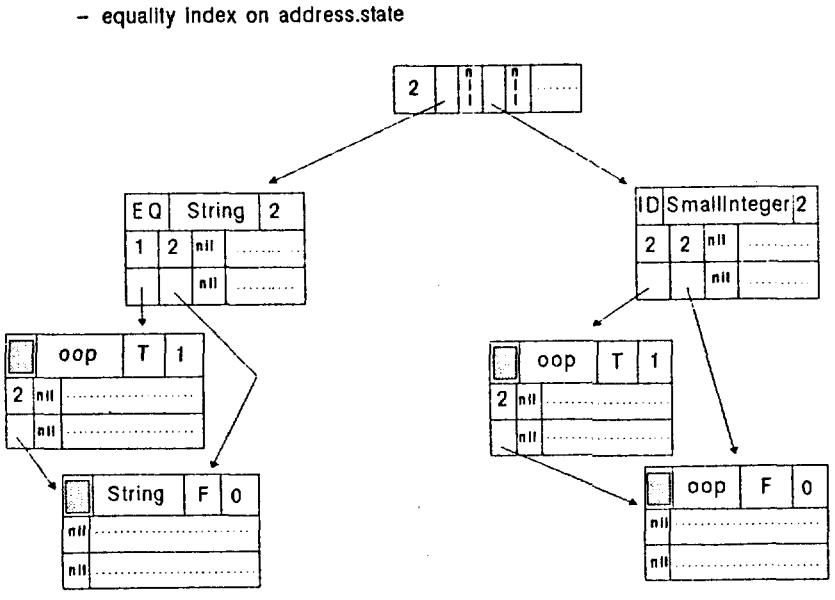


Figure 7

Consider the evaluation of the term *A*.name.first = 'Jones', where *A* is an EmployeeBag object with an equality index on name.first. Using the B-tree from the second component of the indexed path, all those names with a first value of 'Jones' are found. These name values are then sorted by OOP By performing an incremental search of the B-tree of the first component, using the sorted list of name values as lookup keys, the elements of *A* whose name values have a first value of 'Jones' are found.

By not having index entries for nil elements of an NSC, and *not propagating entries for* nil key values to next-components, indexed lookup never return elements of the NSC for which a path is undefined. (Actually, we do insert index entries for nil NSC elements when the path is of zero length.) Thus, to find those elements of an NSC for which a path is undefined, one forms an NSC containing the values present in the first index component of the path and performs a set difference of it from the indexed NSC. The values present in the in the first index component are exactly those for which the path is defined.

4.5. The Query Language

We have chosen to provide associative access through a limited calculus sublanguage. However, we have been careful in constructing the language so that associative queries can be viewed procedurally as OPAL code. We support selection on collections with NSC implementations — subclasses of Set and Bag. Selection conditions are conjunctions of comparisons, where the comparisons are between path expressions and other path expressions or literals. While simple conjunctive selections might seem limited, we note that about the same support for associative access is supplied at the logical level in Cypress [Ca] and in the internal representation of Adaplex queries [CFLR], although those systems, as some others [ZW], select from classes rather than collections. In an object-oriented model, there is no need for many of the joins used in relational systems, as these joins often serve to recompose entities that were decomposed for data normalization. Entities are not decomposed in the first place in an object-oriented model; most joins are replaced by path-tracing, which we support.

An associative query is a variation on a select expression:

```
Emps select:
    {anEmp | anEmp.worksIn.deptName = 'Marketing'
&
    anEmp.salary > anEmp.worksIn.manager.salary}
```

We have extended all of OPAL to allow path expressions. The meaning of the above query is the same as for the corresponding OPAL expression with a regular block.

```
Emps select:
    [anEmp | anEmp.worksIn.deptName = 'Marketing'
&
 anEmp.salary > anEmp.worksIn.manager.salary]
```

Thus there is little impedance mismatch between OPAL and its query sublanguage.

5. Related Work

Experimental extensions of System/R to support complex design objects have dealt with the problem of indexing [HL, LP, PKLM]. There, complex objects are built of a root tuple, plus a tree of component tuples. The resulting object model differs in a fundamental aspect from ours in that the component tuples are dependent on the root tuple. Those component tuples are removed when the root tuple is removed, and they are not shared with other complex objects. (Later versions of the work allow external references to component tuples, but do not enforce referential integrity [Da] for such references.) The notion of dependent component objects shows up in other models [Gr, BB, Ni, We].

Each complex object is composed from tuples of several relations; these relations can be indexed on values actually stored in the tuple. In the hierarchy of component tuples, each tuple has a reference to its parent tuple, and may have references to other component tuples in the same object, or to roots of other objects. Further, each root tuple maintains an index to its component tuples at all levels, to aid in traversing from parent to child tuple, and for moving or copying the entire object. The techniques for indexing complex objects in System/R were not directly applicable to our problem, since component objects in GemStone can be *arbitrarily shared and are not dependent*.

Adaplex [Ch+, CFLR] provides a model similar to GemStone, but again with a significant difference. Entities (objects) may belong to multiple types (classes), unlike GemStone where every object is an instance of a single class. Other models share this multiple multiple-membership property with Adaplex [Zd84, Zd85, DKL]. Since an entity can acquire mappings (attributes) from all the various types it belongs to, the Adaplex designers have chosed to decompose the storage representation of an entity into a logical record for each type to which the entity belongs. (The logical records for an entity can be clustered on physical storage.) Each connected component of the type hierarchy has an entity dictionary — much like our object table — which maps entity identifiers to logical records. The collection of logical records for a given type can be indexed, but on data values only (not entities) and hence not on the substructure of entities. Adaplex allows declarations that two mappings invert each other (such as manages and manager between Employee and Department) to support access from an entity to all other entities containing the first entity as the value for a particular mapping. Note that the individual link indexes in GemStone in essence maintain such an inverse mapping for all objects in a collection, although the inverse mapping is not named.

We also note that Adaplex tightly couples its procedural data language with the host language at the expression level, but preprocesses the host language to extract data accesses and encapsulate them in non-procedural "envelopes". In Cypress [Ca], entities are maintained separately from information about entities (relationships). Entities in a domain (class) are indexed by identity, and relationships can also be indexed. Further, a linked list can be maintained for an entity and all relationship records in which it appears.

An extension to Ingres allows a programmer to add new data types and index support for them [SBG]. However, Ingres treats instances of those types as uninterpreted sequences of bits, so instances of such types can not reference other database entities directly. A successor to Ingres, Postgres [SR], makes some provision for objects, but does so through storing QUEL and C procedures as attribute values. Since complex objects are something the application designer implements on top of Postgres, its hard for the system to give any direct support to indexing complex objects.

6. Conclusion

Indexing as described in Sections 3 and 4 has been implemented in GemStone. Initial instrumentation shows a 300-fold improvement in performance when using indexed access to select a single element from a collection of 10,000 elements using the associative query

```
Emps select:
      {anEmp | anEmp.address.street.name =
            '99936.............AIBA}.
```

This improvement is relative to the corresponding selection block

```
Emps select:
        [anEmp | anEmp.address.street.name =
                '99936.............AIBA].
```

We are in the process of performing benchmarks using the Wisconsin benchmark [BDT].

7. Acknowledgements

The authors would like to thank all those at Servio Logic who have contributed to the GemStone project. We would also like to thank Allen Otis and Alan Purdy for comments on previous drafts of this paper.

8. Bibliography and Trademarks

[BB] Batory, D., and A. Buckman, Molecular objects, abstract data types and data models: a framework, *Proc. Conference on Very Large Databases*, 1984.

[BDT] Bitton, D., D.J. Dewitt, and C. Turbyfill, Benchmark database systems a systematic approach, Computer Science Technical Report #526, University of Wisconsin-Madison, 1983.

[Br+] Brodie, M., B. Blaustein, U. Dayal, F. Maniola, and A. Rosenthal, CAD-CAM database management, *Database Engineering* 7:2, June 1984.

[Ca] Catell, R.G.G., Design and implementation of a relationship-entity-datum model, Xerox CSL 83-4, May 1983.

[Ch+] Chan, A., A. Danberg, S. Fox, W.-T. K. Lin, A. Nori, and D. Ries, Storage and access structures to support a semantic data model, *Proc. Conference on Very Large Databases*, September 1982.

[CFLR] Chan, A., A.A. Fox, W.-T. K. Lin, and D. Ries, Design of an ADA compatible local database manager (LDM), TR CCA 81-09, Computer Corporation of America, November 1981.

[CK] Copeland, G., and S.N. Koshafian, A decomposition storage model, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1985.

[CM] Copeland, G., and D. Maier, Making Smalltalk a database system, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1984.

[Da] Date, C.J., *An Introduction to Database Systems, Volume 2*, Addison-Wesley, 1983.

[DK] Dolk, D.R., and B.R. Konsynski, Knowledge representation for model management systems. *IEEE Transactions on Software Engineering*, 10:6, November 1984.

[Ea] Eastman, C.M., System facilities for CAD databases, *Proc. IEEE 17th Design Automation Conference*, June 1980.

[GR] Goldberg, A., and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[Gr] Gray, M., Databases for computer-aided design, In *New Applications of Databases*, G.Garadarin, E. Gelenbe eds., Academic Press, 1984.

[Ha] Haynie, M.N., The relational/network hybrid data model for design automation databases, *Proc. IEEE 18th Design Automation Conference*, 1981.

[HL] Haskin, R.L., and R.A. Lorie, On extending the functions of a relational database system, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1982.

[JSW] Johnson, H.R., J.E. Schweitzer, and E.R. Warkentire, A DBMS facility for handling structural engineering entities, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Kr] Krasner, G., *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[LP] Lorie, R., and W. Plouffe, Complex objects and their use in design transactions, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[Ma] Maier, D., *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[MOP] Maier, D., A. Otis, and A. Purdy, Object-oriented database development at Servio Logic, *Database Engineering*8:4, December 1985.

[MP] Maier, D., and D. Price, Data model requirements for engineering applications, *Proc. International Workshop on Expert Database Systems*, 1984.

[MSOP] Maier, D., J. Stein, A. Otis, and A. Purdy, Development of an ojbect-oriented DBMS, To appear: ACM Conference On Object Oriented Programming Systems, Languages, and Applications, Portland, Oregon, September 1986.

[Mo] Morgenstern, M., Active Databases as a paradign for enhanced computing environments, *Proc. Conference on Very Large Databases*,1983.

[Ni] Nierstrasz, O.M., An object-oriented system, In *Office Automation: Concepts and Tools*, D.C. Tsichritzis, ed., Springer-Verlag, 1985.

[PKLM] Plouffe, W., W. Kim, R. Lorie, and D. Mc Nabb, A database system for engineering design, *Database Engineering*, 7:2, June 1984.

[Pu] Purdy, A., personal communication.

[RS] Rowe, L.A., and K.A. Shoens, Data abstraction, views and updates in RIGEL, *Proc. ACM/SIGMOD International Conference on the Management of Data*, 1979.

[Si] T.W. Sidle, Weaknesses of commercial data base management systems in engineering applications, *Proc. IEEE 17th Design Automation Conference*, June 1980.

[SBG] Stonebraker, M., B. Rubenstein, and A. Guttman, Applications of abstract data types and abstract indices to CAD data bases, *Engineering Design Application Proceedings* from SIGMOD Database Week, May 1983.

[SR] Stonebraker, M., and L. Rowe, The design of POSTGRES, Berkely TR ERL 85/95, November 1985.

[We] Weisner, S.P., An object-oriented protocol for managing data, *Database Engineering*, 8:4, December 1985.

[Zd84] Zdonik, S.B., Object management systems concepts, *Proc. ACM SIGOA Conference on Office Information Systems*, 1984.

[ZW] Zdonik, Z.B., and P. Wegner, Towards object-oriented database environments, Brown Univeristy TR, 1985.

Trademarks

Proceedings

1986 International Workshop on Object-Oriented Database Systems

KLAUS DITTRICH AND
UMESHWAR DAYAL, EDITORS

Sponsored by ACM Special Interest Group on Management of Data
IEEE CS Technical Committee on Database Engineering

In cooperation with Gesellschaft für Informatik, West Germany
FZI, University of Karlsruhe, West Germany
IIMAS, Mexico

IEEE Computer Society Order Number 734
Library of Congress Number 86-45866
IEEE Catalog Number 86TH0161-0
ISBN 0-8186-0734-3
ACM Order Number 472861

SEPTEMBER 23-26, 1986
ASILOMAR CONFERENCE CENTER
PACIFIC GROVE, CALIFORNIA

Association for Computing Machinery
THE COMPUTER SOCIETY OF THE IEEE
COMPUTER SOCIETY PRESS