Function-Based Indexing for Object-Oriented Databases

by

Deborah Jing-Hwa Hwang

February 1994

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Function-Based Indexing
for Object-Oriented Databases

by

Deborah Jing-Hwa Hwang

Submitted to the Department of
Electrical Engineering and Computer Science on February 9, 1994
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in Computer Science

Abstract

Object-oriented databases have been developed to allow applications to store persistent data in the form of objects. As in conventional databases, object-oriented databases need to support queries that allow associative access to data by describing a property of the data of interest. In particular, they should support queries over user-defined sets based on user-defined functions that compute the property of interest. To make such *function-based queries* run faster, we must support *function-based indexes* where the keys are these computed properties. Function-based indexes are harder to maintain than indexes in conventional databases because mutations that affect keys can be done without accessing an element of the indexed set.

This dissertation presents a new function-based indexing scheme for object-oriented databases. The indexes are maintained by recording *registration information* for objects that indicate when index entries are affected and having operations that modify objects check for registrations to determine if an index needs to be updated. Our approach uses a combination of declared information that indicates how object methods affect each other and runtime information gathered during key computation, so that we register only the objects whose modifications can affect an index and only the operations that might affect an index entry check for registration information.

Recomputing index entries only when there is a modification that might affect an index comes at the cost of extra space for the registration information and extra time to check registration information during modifications. To quantify these costs, we simulated three implementations of our scheme on various performance benchmarks and analyzed several others. We show that function-based indexes are very useful in object-oriented databases and do not adversely affect system performance, but they do have a high space cost.

This work is being done in the context of the Thor distributed, object-oriented database system[42, 43]. Thor has features that complicate index use and maintenance. It has a client-server architecture with caching at both clients and servers. It also uses an optimistic concurrency control scheme. This dissertation presents a design for integrating index use and maintenance into Thor. We discuss the impact of adding indexes on Thor's system architecture and the impact that Thor's system architecture has on the performance of our indexing scheme, and we present *predicate validation*, a new optimistic concurrency control scheme.

Thesis supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Keywords: function-based indexing, function-based queries, object-oriented databases, registration, distributed systems, predicate validation

Acknowledgments

This dissertation would not have been possible without the support of many people throughout my graduate school career. First, I would like to thank my advisor Barbara Liskov for guiding my research and making sure I presented the work in a clear and concise manner. Next, I would like to thank my readers Bill Weihl and Dave Gifford who provided excellent feedback despite having very little time to review my work.

The people in the Programming Methodology group have been invaluable in providing support for this research. Special thanks to Mark Day, Umesh Maheshwari, and Atul Adya for building the TH pre-prototype that allowed me to explore the ideas in this thesis without building the whole system myself. Special thanks also to Dorothy Curtis for keeping Argus running even though we kept trying to break it, fixing bugs in the Alpha pclu compiler and debugger promptly, and helping me speed up my simulator.

I thank my MIT friends for understanding what graduate school at MIT is like, and I thank my non-MIT friends for making sure I ventured out into the "real world" occasionally.

I thank my family for the love and support that has been a source of strength throughout my life. And I thank Mike's family for welcoming me into their family.

My deepest appreciation goes to my husband Mike Ciholas who has been waiting patiently for me to finish and always said that I would finish even when I wasn't sure I could finish. This dissertation is dedicated to him.

Contents

List of Figures

Chapter 1

Introduction

Databases are used to manage large amounts of information that may be accessed by many different applications. They provide persistent and reliable storage, a uniform access model, and convenient retrieval facilities in the form of queries. They have proved to be particularly useful for various business applications such as inventory control and banking. The data generated by these applications tend to be regular and easily grouped into sets of records; thus the data can be stored in conventional databases.

Object-oriented databases have been developed to manage the data of applications such as CAD/CASE or office automation systems. These applications create data that are related in arbitrary and complex ways that do not fit well into fixed-size records. Instead of using records, the data in these applications are represented as objects. Objects encapsulate state and are declared to be of a particular *type*. The type of an object specifies the abstract *methods* that can be used to observe and modify the state of an object. Objects can contain an arbitrary number of direct references to other objects, and many objects can have references to the same object. In general, the object-oriented paradigm captures more semantically meaningful behavior and fits more naturally with the way the data in these applications are accessed. In addition, data from "traditional" database applications can be cast easily into the object-oriented paradigm.

Much of the success of conventional database systems is due to their support for queries. Queries provide users with a convenient way of identifying information of interest from a large collection using associative descriptions rather than identifying the data of interest exactly. For example, a query might be: "Select the employees from department E with a project manager whose yearly income is greater than $60,000." In a conventional database, this query would operate over a set of employee records by finding the records for the employees of department E and returning the ones that have project managers that make more than $60,000. To become as successful as conventional databases, object-oriented databases must also support queries[3].

Since queries are an important part of database use, *indexes* are used in conventional databases to make them run faster. This dissertation focuses on providing indexes in object-oriented databases. It makes two contributions. The first contribution is a very general function-based indexing scheme with more expressive power than previous schemes. We show through simulations that these indexes do make function-based queries run faster. We propose several different implementations and characterize the costs of maintenance for each one through simulation and analysis. We also propose several optimizations to our basic scheme.

The second contribution is a design for implementing function-based indexing in distributed systems where clients of a database are at different nodes from the database. We describe how to integrate index use and maintenance in these systems and analyze the performance impact of this system architecture on function-based indexing. To handle concurrent use and maintenance of indexes, we have developed *predicate validation*, an optimistic concurrency control algorithm suitable for integrating index use into optimistic concurrency control schemes. We have also developed an incremental index creation scheme.

The next two sections of this chapter discusses these contributions in more detail. The final section provides a roadmap for the rest of this dissertation.

1.1 Queries and Indexes

In a conventional database, the records representing a particular kind of information are grouped together into a single file or table. This is the only grouping supported directly by the system and is the basis for queries in these systems. By contrast, some object-oriented databases support user-defined sets that group together specific objects of the same type (for example, GemStone[12, 47, 48]), and we would like to support queries over these user-defined sets. In such a system, an object can be an element of more than one user-defined set. For example, consider a database containing objects representing employees and departments. Each department could contain a set of employees and if an employee worked for two departments, the sets for the two departments each would contain that employee.

In addition, to take the most advantage of the object-oriented paradigm, we want to provide *function-based* queries. In a function-based query, the property of interest is described as the result returned by a user-defined function on a set element. (I.e., for a function f, the value of the property of interest for a set element x is $f(x)$.) A user-defined function for such a query may perform arbitrary computation involving one or more parts of the state of the set element including calling methods of subobjects. For example, to answer the query, "Select the employees from set E with a project manager whose yearly income is greater than \$60,000," we would write a function f that computes an employee's project manager's income by first calling a method of an employee object to get a project object, then calling a method of the project object to get the manager employee object, and so forth.

We could compute a function-based query result by computing the function of interest for every element in a set and testing the function result to determine if the element should be included in the query result. However, this is very time-consuming for large sets. If we are going to compute many queries using the same function f, it would be useful to have a *function-based* index on E based on f. An index is a memoizing device that makes queries more efficient by mapping a property of interest (called the *key*) to the entities that have particular values for the property. In a function-based index, the properties of interest are the results of the index function computed for the set elements; thus an index I on set E using function f is a mapping of $< f(x), x >$ pairs, for each set element x in E. When a function-based query is computed using a function-based index, the property of interest does not have to be computed and the set elements that do not have property values of interest are not accessed, so queries run faster.

To maintain a function-based index, the database system must recognize when changes to objects affect the information stored in the index and recompute keys for the entries that have changed. Some changes are to elements of the indexed set (for example, an employee in E might change projects), but others are not (for example, the project manager of a project might change). In the object world, since an object may contain references to other objects and an object may be referenced by multiple objects, mutations that affect set indexes can occur without accessing the set element itself. Thus, index maintenance becomes more complex than in a conventional database.

This dissertation presents a new function-based indexing scheme that supports indexes over user-defined sets using user-defined functions. In our scheme, we try to minimizes the cost of index maintenance by only recomputing a key when a modification that can affect the key has been done. This is accomplished by *registering* any object that can be modified in a way that affects a key. That is, some information is associated with an object that indicates which keys are affected and when these keys need to be recomputed. When an object is modified, its registration information is checked and key recomputations are performed, if needed. Determining which objects should be registered is done using a combination of information declared in type specifications and information gathered while keys are being computed.

Providing indexes is a space/time tradeoff to make queries run faster in return for the space for the index. In addition, there is the cost of maintaining an index. We use space for the registration information and time to update the index when there is a modification that affects the index. To characterize the benefits of indexes and the cost of index maintenance, we simulated three implementations and analyze several others. Our results show that indexes are very useful in object-oriented databases and do not adversely effect system performance. We also find there are tradeoffs in the cost of maintaining function-based indexes stemming from the amount of registration information needed in each of the implementations and where it is stored, thus we have developed a framework for comparing space overhead of different indexing schemes.

Several schemes have been proposed for indexes in object-oriented databases. Compared to these schemes, our indexing scheme supports more expressive queries; it has the following desirable properties:

- It preserves abstraction and encapsulation. It supports queries that are expressed in terms of objects' methods, not in terms of objects' representations. Furthermore, the methods can do general computations, for example, the method that provides an employee's yearly income might compute it based on stored information about monthly income.

- It allows indexes to be based on user-defined functions. For example, employee objects probably do not have a method that returns an employee's project manager's income, but we can still maintain an index based on a function that computes this result.

- It allows indexes to be maintained for user-defined sets of objects. For example, we might maintain an index only for the set of employees in the engineering department and not for other departments in a business. Thus applications can incur the expense of index maintenance only when necessary.

- It supports multiple representations per type. Each set being indexed contains objects of some declared type. However, objects of a type need not have the same representation; there may be objects of the same type that have different representations. Furthermore, new implementations of types can be added to the database system dynamically.

Schemes that base queries on path expressions (for example, GemStone[47]) violate abstraction and encapsulation. They require knowledge of the implementation of a type to name the path of interest, constrain types to one representation, and require that the key values of interest be represented directly in the type's implementation. Method precomputation schemes[8, 10, 35, 34] allow queries based on method results, but they do not support indexes based on other functions and the only sets that can be indexed are type extents, sets that contains all the objects of a particular type.

1.2 Distributed Databases

We expect object-oriented databases of the future will be used in systems comprised of a network of computational nodes. In this environment, the database is a collection of *server* nodes storing the data, and there can be multiple *clients* (i.e., applications) accessing the data from other distinct client nodes. We expect that client nodes will cache data from the servers and do client computations locally, so that clients do not have to incur a network delay whenever they want to access their data. This also reduces the load at the servers. However, client caches are usually not large, so it is important that only the "interesting" data be cached at the client node. In such an environment, queries become an important tool in increasing client cache utilization and reducing the amount of network bandwidth wasted on transferring uninteresting objects by allowing clients to specify that only certain elements in a set are of interest; the system only transfers those elements rather than all of a set's elements.

Additional complexity in index use and maintenance arises when databases are distributed and clients cache data. There are questions about where computations associated with indexes take place. To answer a query, we may transfer an index to the client node. But an index may be a large object, and may cause more interesting data to be thrown out of the client's cache. The index may only be used once, so this is space inefficient. On the other hand, running queries at a server places additional load on the server. Likewise, key recomputation and index updates after modifications to registered objects can be done at the client or the server. Finally, there are questions about how concurrent use and maintenance of indexes interacts with concurrent access and modification of regular data objects.

One such object-oriented database system is Thor[42, 43], the context for our work. Thor performs all user computations at client nodes on cached copies and uses an optimistic concurrency control scheme. In addition, it transfers its objects from disk to the server cache in very large units. This dissertation describes how index use and maintenance can be integrated into the Thor system. Queries using indexes are run at the server. Key (re)computations are done at the client node, but actual updates to the index structure are also done at the server. *Predicate validation* is used to handle concurrent index use and maintenance. Predicate validation is a new optimistic concurrency control scheme that allows concurrent index use and maintenance with concurrent access and modification of regular data objects. In this scheme, index operations are represented by predicates and conflict detection is done on these predicates at transaction commit along with the regular conflict detection done on regular data objects. We also present a scheme for creating indexes incrementally. Incremental index creation is needed because long-running transactions may have difficulty committing due to the backward validation protocol used by Thor.

In addition, we discuss the impact of the Thor's system architecture on our performance evaluation. We extend our simulations to include transferring objects from disk in very large units. We conclude from the results that when objects are transferred in very large units, how objects are clustered on disk matters more than when they are transferred from disk in smaller units. We also evaluated the effect of the two-level architecture on our implementation schemes, and we conclude that the two-level architecture does not affect the relative merits or disadvantages of our implementation schemes.

1.3 Roadmap

Chapter 2 presents the design of our indexing scheme. We give a precise characterization of which objects need to be registered, describe the basic registration algorithm, describe the basic update algorithm to handle modifications that affect indexes, and give an informal argument of the correctness of this scheme.

In Chapter 3, we evaluate the performance of our indexing scheme. We describe several ways of organizing registration information. We present the results of simulating three of the schemes on various performance benchmarks and an analysis of the other schemes. We present a framework for comparing the space costs of indexing schemes, and we characterize the space cost of our scheme and discuss ways of reducing it.

We discuss work related to indexing in object-oriented databases in Chapter 4. We compare the expressive power of our function-based indexing scheme with path-based indexes and other function-based indexes and also analyze the costs of providing and maintaining these other indexes.

In Chapter 5, we present several optimizations to the basic algorithms. *Contained subobject analysis* allows the scheme to register fewer objects, while *lazy updates* and *deregistration* allow the scheme to do fewer index updates.

Chapter 6 discusses how indexes can be used and maintained in the Thor system. We describe where computations are done, how predicate validation works, and how to do incremental index creation. In addition, we discuss the impact of the Thor system architecture on the conclusions of our performance study.

We conclude with a summary of our work and a discussion of future work in Chapter 7.

Chapter 2

Function-Based Indexing

An index is a memoizing device. It maps *keys* which represent properties of interest to the entities that have those properties. Since we are concerned with function-based indexes, our keys are the result of computing a function $f(x)$ and the index stores a collection of pairs $< f(x), x >$, one for each element x in the indexed set. To maintain a function-based index, we must be able to detect when a modification to an object may affect the index. Our scheme is based on *registering* the objects that may affect an index. We determine which objects to register using a combination of static information that is declared and run-time information that is collected during key computation. Registration information is consulted during modifications to determine if it affects an index, and if so, the key for the affected entry is recomputed.

In this chapter, we present the basic design for our function-based indexing scheme. We begin the chapter by giving an example of the problem we are trying to solve and an overview of our solution to this problem. Then we present our object model followed by a brief description of our transactional model. We define the properties that a function must have to be useful as an index function in Section 2.4. We give a precise characterization of which objects need to be registered in Section 2.5 and describe the basic registration algorithm that registers these objects in Section 2.6. Section 2.7 discusses how registration happens during index creation and insertions, and the effect of deletions on registration information. Section 2.8 explains how modifications are handled in our scheme. Finally, we give an informal correctness argument that our scheme registers all the objects that it is suppose to register and that updates maintain indexes and registration information correctly.

2.1 Overview

Let us introduce the problem we are trying to solve with an example that we will use throughout this dissertation. Suppose we have a set of employee objects. Figure 2.1(a) shows part of this set. There are three kinds of objects in the figure:

1. Employee objects, e.g., objects α and β, contain two variables *proj_* and *income_* that refer to a project object and an income_info object, respectively.

2. Project objects, e.g., object π, contain a variable *manager_* that refers to a employee object.

3. Income_info objects, e.g., γ, contain variables *rate_* and *bonus_* that refer to integers.

Suppose we want to create an index on this set using a function *project_manager_income* that computes an employee's project manager's income. In creating the index, we run the function over every set element. When we compute $project\_manager\_income(\alpha)$, we access objects α, π, β, and γ. Now suppose later we modify π to refer to δ, as shown in Figure 2.1(b). Obviously, the entry in the index for α is no longer correct and *project_manager_income*(α) needs to be recomputed, but the problem is how do we determine when a recomputation is necessary.

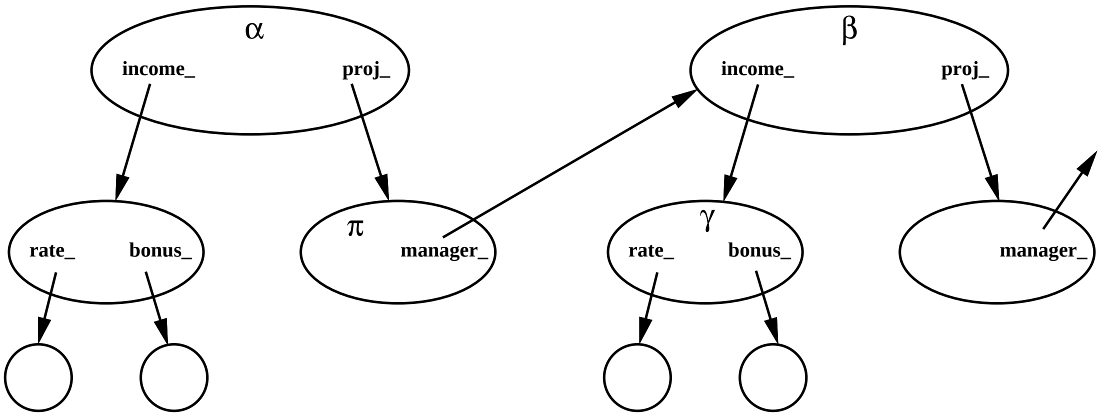
An obviously correct way of handling index maintenance is to recompute all indexes every time any object is modified. This is also obviously inefficient. To make indexing work well, maintenance should be done only when necessary. That is, if a modification does not affect any indexes, then do nothing. If a modification does affect an index, only recompute the part of the index that has changed in an automatic and efficient manner.

Our approach is to *register* the objects that if modified would cause an index entry to change. That is, we identify which objects can be modified in a way that causes index updates and maintain that information. Then index maintenance work is done only when a registered object is modified. The key to making the system work well is to register as few objects as possible. There are two aspects to minimizing the number of objects that are registered:

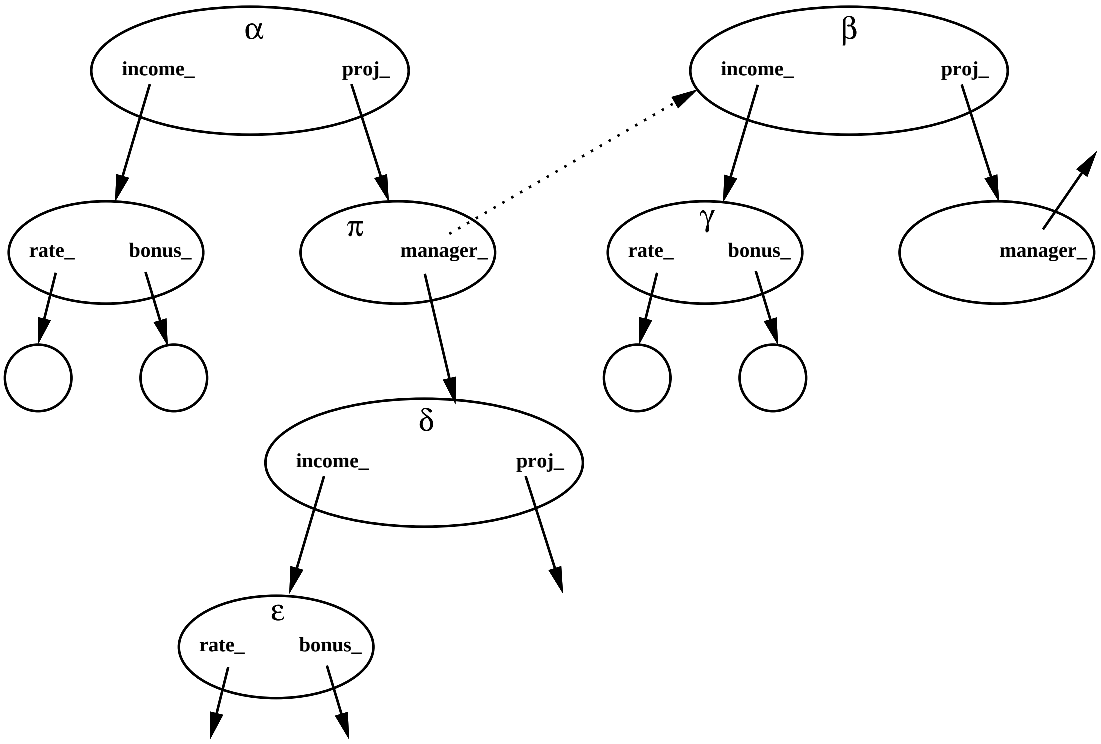
1. Recognizing the objects that (might) affect an index.

2. Recognizing the methods that (might) affect an index.

Clearly, the only objects that can affect an index are the ones that are accessed during key computations. However, not all of these objects need to be registered. For example, a local object holding temporary results does not need to be registered, nor does an object that cannot be modified. Our scheme uses two types of information to determine which objects need to be registered. First, we add dependency declarations that how object methods affect each other and using this information we only register objects that are accessed using methods that can be affected by other methods. Second, we create registration versions of the methods and



(a) Objects before modification.



(b) Objects after π is modified to refer to δ.

Figure 2.1: Example modification that affects an index. (Greek letters are OIDs.)

functions invoked during key computations that enter registration information into the system when necessary. We also create checking versions of the methods that do modifications that can affect the methods used in key computations. These checking versions consult the registration information and cause index updates if needed. Using these techniques, we try to minimize the number of registered objects and recomputations done in our scheme.

2.2 Object Model

Our object model is based on the Thor object model[42, 43]. Thor is a distributed, object-oriented database being developed by the Programming Methodology Group at the MIT Laboratory for Computer Science. Objects in Thor are specified and implemented in an application-independent language called Theta[22]. In the Thor model, there is a universe of persistent objects. Every object has a unique identity (an *OID*), an encapsulated state, and a set of methods that can be used to observe and modify the state. Objects can refer to other objects, allowing general data structures, such as lists, trees, and graphs, to be defined. Also, objects can be shared, i.e., many objects can refer to the same object.

Each object belongs to a type. A type is described by a *specification*. A specification defines the names and signatures of the methods for objects of that type. A specification also defines the meaning of a type. This includes a description of the behavior associated with calling each of the methods of the type. This semantic information is generally uninterpreted (and currently exists as comments). We will be augmenting specifications with information that will be interpreted by our indexing scheme. Information from type specifications is maintained online so that it is available when code is compiled.

We classify the methods of a type into two kinds: *observers*, which access but do not modify the state of their object, and *mutators*, which modify the state of their object. If all of an object's methods are observers, the object is *immutable* since there is no way to change it; in this case we also say that the type is immutable, since all of its objects are.

A specification for the `employee` type written in Theta is shown in Figure 2.2. The observers shown are *id*, *name*, *address*, *city*, *zip*, *yearly_income*, and *project*. The mutators shown are *set_name*, *set_monthly_rate*, *set_bonus*, and *set_project*. The `address_info`, `project`, `income_info`, and `medical_info` types have similar specifications. For our example, we are only interested in a few of the methods of the `project` and `income_info` types; the specifications for these methods are shown in Figure 2.3.

A type can be a *subtype* of zero or more other types. Its specification lists these *supertypes*. (Our `employee` type does not have any supertypes.) A subtype must have all the methods of its

```
employee = type

   % e refers to the receiver

   id () returns (int)
      % returns the id of e
   name () returns (string)
      % returns the name of e
   address () returns (address_info)
      % returns the home address of e
   city () returns (string)
       % returns the city that e lives in
   zip () returns (string)
       % returns the zip code of e's address
   yearly_income () returns (int)
      % returns the yearly income of e
   project () returns (project)
      % returns the project e works on
   set_name (new_name: string)
      % sets the name of e to new_name
   set_monthly_rate (new_rate: int)
      % sets the monthly rate paid to e to new_rate
   set_bonus (new_bonus: int)
      % sets the bonus paid to e to new_bonus
   set_project (new_project: project)
      % sets the project e works on to new_project
   :
   end employee
```

Figure 2.2: Specification of the `employee` type.

```
project = type

    % p refers to the receiver

    manager () returns (employee)
        % returns the manager of p
    set_manager (new_manager: employee)
        % sets the manager of p to new_manager
    :
    end project

income_info = type

    % i refers to the receiver

    rate () returns (int)
        % returns the rate of i
    bonus () returns (int)
        % returns the bonus of i
    set_rate (new_rate: int)
        % sets the rate of i to new_rate
    set_bonus (new_bonus: int)
        % sets the bonus of i to new_bonus
    :
    end income_info
```

Figure 2.3: Specifications of other methods used in the example.

supertypes (with signatures adjusted according to Cardelli's contra/covariance rules[13]), plus it may have additional methods.

A type is implemented by a *class*. The class defines a representation consisting of a set of *instance variables* and provides code to implement the type's methods and creators in terms of that representation. The instance variables may be data like integers or booleans that are stored within the object, or they may be references to other objects. Objects are encapsulated; only the object's methods have access to its instance variables. A type may be implemented by more than one class. Figures 2.4 and 2.5 show a class implementing the `employee` type and a creator for the class. This implementation has six instance variables. Note that the implementation makes method calls on some of the objects referenced by its instance variables, but the compiler does not need to know what class is being used to implement these objects, only what type they are. A class can be a *subclass* of another class, its *superclass*. (Our `employee_impl` class does not have a superclass.) A subclass can inherit instance variables and code from its superclass so that programmers can reuse code without duplicating it.

In addition to methods of user-defined types, there are stand-alone routines. Stand-alone routines are useful for defining computations that are not provided by methods. For example, an index using an employee's project manager's yearly income as a key requires a function that is not a method of `employee` objects. Figure 2.6 shows how this routine would be implemented.

`Set` is a built-in type in Theta. `Set` objects have the usual methods, for example, to insert and delete elements and to test for membership. The `set` type is parameterized by the element type, for example, `set[employee]` contains `employee` objects as elements. There can be many `set[T]` objects, for example, both the engineering department and the personnel department have `set[employee]` objects. Also, a `T` object might be an element in more than one `set[T]` object. `Set` objects are defined and maintained explicitly by users. Type "extents" (implicit sets containing all objects of a type) are not maintained; an application can maintain an extent explicitly using an ordinary set if desired.

2.3 Transactional Model

In our computational model, every interaction with the database occurs within an atomic transaction. Clients start a transaction, call methods of objects and other operations and routines, then try to *commit* any changes. Transactions provide serializability and atomicity. That is, the computational steps executed in a transaction appear to run in some serial order with respect to the computational steps done by other transactions, and either all changes done by a transaction to persistent objects are reflected in the database upon transaction commit or

```
employee_impl = class employee_ops

   id_: int
   name_: string
   address_: address_info
   income_: income_info
   proj_: project
   med_: med_info

   id () returns (int)
      return (id_)
      end id

   name () returns (string)
      return (name_)
      end name

   address () returns (address_info)
      return (address_)
      end address

   city () returns (string)
      return (address_.get_city ())
      end city

   zip () returns (string)
       return (address_.get_zip())
       end zip

   yearly_income () returns (int)
      return (income_.monthly_rate() * 12 + income_.bonus())
      end get_yearly_income

   project () returns (project)
      return (proj_)
      end get_project

   % continued in next figure
```

Figure 2.4: Class implementing the `employee` type.

```
% continued from previous figure

set_name (new_name: string)
    name_ := new_name
    end set_name

set_monthly_rate (new_rate: int)
    income_.set_monthly_rate (new_rate)
    end set_monthly_rate

set_bonus (new_bonus: int)
    income_.set_bonus (new_bonus)
    end set_bonus

set_project (new_project: project)
    proj_ := new_project
    end set_project

:

create (id: int, name: string, address: address_info,
        rate, bonus: int, proj: project) returns (employee)
    inc: income_info := income_info_impl.create(rate, bonus)
    med: med_info := med_info_impl.create()
    init {id_ := id, name_ := name, address_ := address, income_ := inc,
         proj_ := proj, med_ := med}
    end create

end employee_impl
```

Figure 2.5: Class implementing the `employee` type, continued. The **init** statement assigns values to the instance variables of a newly created object of the class and automatically forces a return of the new object.

```
project_manager_income (e: employee) returns (int)

    p: project := e.project ()
    m: employee := p.manager ()
    return (m.yearly_income ())
    end project_manager_income
```

Figure 2.6: An example index function.

the attempt to commit may fail, in which case the transaction *aborts* and none of the changes are reflected in the database. We assume that index use and maintenance also take place within this transactional framework. Thus any series of computational steps taken by our scheme is completed atomically; the steps are not interleaved with other concurrent transactions.

Indexes are created explicitly by a client in our model. The client must specify the set S and a function f that generates the keys when creating an index over S. For each element x of S, the system computes $f(x)$ and associates the result as the key for x in the index. For now, we will assume that the entire process runs within a single transaction. (We discuss incremental index creation in Chapter 6.) After transaction commit, the index will contain a entry for every element x of S.

2.4 Index Functions

Since an index is a memoizing device, not all functions can be used as index functions. We impose three requirements on index functions:

1. f must be a function of its argument: it must return the same result if called with the same argument in the same state.

2. f must be side-effect free: it must not modify any of the objects it uses in a way that would be detectable later. Note that this constraint is at the abstract type level. f could still perform *benevolent side effects* (that is, side effects that do not affect the abstract state of an object).

3. f must have a deterministic implementation: it must access the same objects each time it is computed between mutations that affect its result.

The first two requirements are necessary for indexes to make sense as a memoizing device. If f were not a function of its argument or side-effect free, then the result from one invocation would not be equivalent to the result from another invocation with the same argument. The third requirement is needed for our technique to work, as will be explained later. We do not believe it represents a significant loss of expressive power. (All other indexing schemes for object-oriented databases have the same restriction as explained in Chapter 4.)

To simplify the discussion, we will only consider index functions of the form:

$$f : \texttt{T} \rightarrow \texttt{b}$$

That is, we limit f to have only one argument of the set element type `T`. The type `b` must be immutable and must have the methods needed to maintain an index (for example, *less_than*, *equal*), but we will assume `b` to be one of the built-in types (for example, `int` or `string`). We will also assume that f to be a total function. We discuss the impact of relaxing these limitations in our discussion of future work in Chapter 7.

We will use the following notation in the rest of this dissertation: for an index I, we will refer to its index function as $I.f$, the set being indexed as $I.set$, and the collection of $< I.f(x), x >$ pairs as $I.data$.

2.5 Registration Information

Our approach to determining which objects to register is based on the *reachability* of an object from an element of an indexed set and knowing which observer was used in computing a key. Using this information, we register only those objects that if mutated might cause the index to change and do index maintenance only when a registered object is mutated in a way that may cause an index to change. More precisely, we only register objects that meet the following conditions:

1. They are accessed in computing the index function $I.f$.

2. They are *reachable* from the elements of the indexed set.

3. They have mutators that can affect the result of $I.f$.

The objects reachable from a set element are those that can be accessed by following references starting at the set element. For example, all objects shown in Figure 2.7 are reachable from `employee` object α. An index function $I.f$ might access additional, unreachable objects but these cannot affect the index because of our requirement that $I.f$ be a function. (The additional objects must either not have mutators that affect the results of $I.f$ or be temporary and local to $I.f$ and the routines it calls, for example, an array that holds some temporary information during a computation, or a global integer that is never changed.) Thus, only reachable objects can affect the index.

However, a particular index function probably does not access all reachable objects from a set element. $I.f$ accesses object y if it calls a method of y, directly or indirectly, during its computation. We will say that objects accessed by $I.f$ and reachable from x are *f-reachable* from x using index function $I.f$. The shaded objects in Figure 2.7 are f-reachable from `employee` object α where $I.f = \mathit{project\_manager\_income}$. Only the f-reachable objects of a set element x can affect the entry in I for x: objects $I.f$ does not access cannot affect its result either now, or in a future computation because of our requirement that $I.f$'s implementation be deterministic.

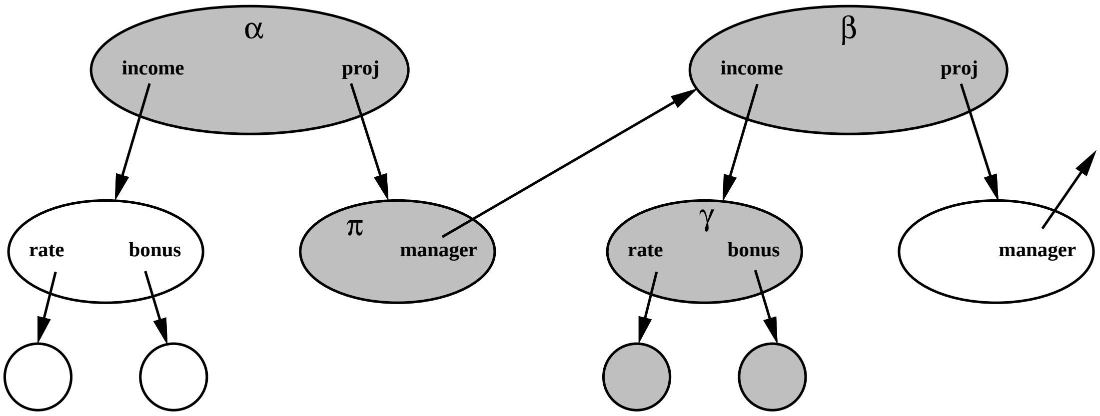


Figure 2.7: All objects shown are reachable from α. The shaded objects are f-reachable from α where f = *project_manager_income*. (Greek letters are OIDs.)

Stopping the analysis here would still register more objects than necessary. Immutable objects that are f-reachable do not have to be registered since there are no mutators to change their state. More generally, an object need not be registered if it is accessed only via observers that are not affected by any mutators.

We acquire this *dependency* information by adding declarations to a type's specification. These declarations indicate which observers are affected by which mutators; we declare that the observer *depends on* the mutator in this case. Figure 2.8 shows the specification of the `employee` type with these new dependency declarations. The specification indicates that the *yearly_income* method (which returns the current yearly income of the employee) depends on the *set_monthly_rate* and *set_bonus* methods (since these cause the yearly income to change) and the *project* method depends on the *set_project* method. In addition, the `project` type has dependencies between *project* and *set_project*, and the `income_info` type has dependencies between *bonus* and *set_bonus*, and *rate* and *set_rate*. We believe that these dependency declarations will not be hard to write. The effects that mutators have on observers are an important part of a type's meaning and should be obvious to the type definer.

Conceptually, whenever an object y is registered, a *registration tuple* of the form:

$$< y, m, x, i >$$

is added to a *registration table*. (We discuss various implementations of registration information in Chapter 3.) When such a tuple is contained in the registration table, it means that when mutator m of object y is invoked, m might affect the *key* paired with set element x in index i. It is worth noting that all of the information in the registration tuple helps us to avoid

```
employee = type

   % e refers to the receiver

   id () returns (int)
      % returns the id of e
   name () returns (string)
      % returns the name of e
      depends on set_name
   address () returns (address_info)
      % returns the home address of e
   city () returns (string)
       % returns the city that e lives in
   zip () returns (string)
       % returns the zip code of e's address
   yearly_income () returns (int)
      % returns the yearly income of e
      depends on set_monthly_rate, set_bonus
   project () returns (project)
      % returns the project e works on
      depends on set_project
   :

   set_name (new_name: string)
      % sets the name of e to new_name
   set_monthly_rate (new_rate: int)
      % sets the monthly rate paid to e to new_rate
   set_bonus (new_bonus: int)
      % sets the bonus paid to e to new_bonus
   set_project (new_project: project)
      % sets the project e works on to new_project
   :
   end employee
```

Figure 2.8: Specification of `employee` with dependency information.

doing unnecessary recomputations. Without the index i, we would not know which index to recompute; without the set element object x, we would have to recompute the whole index, rather than just one entry in it; and without the mutator, m, we would sometimes do a recomputation that was not needed. For example, this information allows us to avoid recomputing *project_manager_income* index entries when the *set_bonus* method modifies an `employee` object that is not a manager.

2.6 Registration Algorithm

Our registration algorithm works by keeping extra information as a key is computed. This extra information keeps track of f-reachability so that only f-reachable objects are considered for registrations. In our scheme, every observer O invoked by an index function has a registration version O_r. (O_r is produced by the compiler either at normal compile time or dynamically, upon creation of an index.) O_r takes the same arguments as O plus some extra ones: the set element x whose key is being computed, the index I that is being computed, and a set of objects R that contains all objects reached from x so far (this will be explained in more detail below). Each stand-alone function p also has a registration version p_r, if it is called in computing an index.

O_r (the registration version of observer or function O) does the following (with extra arguments x, I, and R):

1. If O is a method for some object $y \in R$,
   (a) For all mutators $m \in depends\_on(O)$, add a tuple $< y, m, x, I >$ to the registration table.
   (b) Add all objects referenced by the instance variables of y to R.
2. Run the body of O. For all observer or function calls p in O, call registration version p_r, passing x, I, and R as extra arguments. (If O does benevolent side effects, the mutators that it calls will also need registration versions.)
3. Return the same result as O.

Note that x and I are known because they are arguments to each O_r. A method's object is also known. (In Theta, the pseudo-variable *self* refers to this object.) *Depends_on*(O) is extracted from the dependency information given in the specification of y's type and is a set containing all the mutators O depends on. For example, *depends_on*(*yearly_income*) = {*set_monthly_rate*, *set_bonus*} for `employee` objects.

y	m	x	i
α	*set_project*	α	I
π	*set_manager*	α	I
β	*set_monthly_rate*	α	I
β	*set_bonus*	α	I
γ	*set_rate*	α	I
γ	*set_bonus*	α	I

Figure 2.9: Registration table after computing *project_manager_income*(α) for index I.

For our example, to use *project_manager_income* as an index function, we would need registration versions of the *project_manager_income* function, the `employee` observers *project* and *yearly_income*, the `project` observer *manager*, and the `income_info` observers *monthly_rate* and *bonus*. Figure 2.9 shows the registration tuples that are added to the registration table after computing $I.f_r(\alpha)$. Note that since object α is not a project manager, $I.f_r(\alpha)$ does not cause α to be registered for its *set_monthly_rate* or *set_bonus* methods.

2.7 Creation, Insertion, and Deletion

When a client creates an index, the system must compute $I.f(x)$ for every set element in creating $I.data$. In our scheme, the system uses $I.f_r$ (the registration version of $I.f$) for computing the key for each set element x; it passes in x, I, and $R = \{x\}$. The result of $I.f_r(x)$ is the *key* paired with x in $I.data$. After the creation transaction commits, all of the objects that can be mutated in a way that affects I will be registered.

When an object x is inserted into an indexed set, a key has to be computed for x and an entry added to I; if the indexed set has several indexes, we must do this for each index. In our scheme, we use $I.f_r$ to compute the key for x, passing in x, I, and $R = \{x\}$, so that after the transaction commits, the appropriate f-reachable objects from x are registered for I.

When an object x is deleted from an indexed set, for all indexes I on the set, $< I.f(x), x >$ is deleted from $I.data$ and all tuples of the form $< x, m?, x, I >$[1] are removed from the registration table. When a registration tuple $< y, m, x, I >$ is removed from the registration table, we say that y has been *deregistered* for m, x, and I. We deregister x when it is deleted from an indexed set because its keys no longer have to be maintained. However, note that not all of the obsolete registration information is being removed (for example, objects reachable

[1]The ? notation indicates a pattern variable.

from x may still be registered). Conceptually, we could specify that all tuples of the form $< y?, m?, x, I >$ are removed from the registration table, but as we will see in the next chapter, some implementations would make this difficult to achieve. Extra registrations do not affect correctness, but they do cause unnecessary index updates when mutations happen; we discuss deregistration in more detail in Chapter 5.

2.8 Mutations and Updates

Registration information is maintained so that when mutations occur, we can modify the affected indexes. Mutators that can affect an index should check the registration table for entries involving the object being mutated and the mutator being executed. If there is such an entry, the mutator should cause an index update. In addition, mutations may cause changes in the reachability graph, so we might need to update the registration table after mutations as well.

In our scheme, when an object y is registered for a mutator m, y's method dispatch vector entry for m is replaced with a checking version m_c that checks the registration set of y whenever m_c is invoked. (Mutator m_c is also produced by the compiler, either at the original compile time or when an object is registered). The checking version m_c looks for registration tuples of the form $< y, m, x?, I? >$. If there are any, and if the mutator is actually going to modify the object, the following is done:

1. Remove all tuples $< y, m, x?, I? >$ from the registration table. For each such tuple, if $x? \in I?.set$, $< key, x? >$ is removed from $I?.data$, where key is the current key value paired with $x?$ in $I?.data$, and the tuple is added to a list L of "affected indexes."

2. m_c does the actual mutation to its object.

3. For each tuple $< y, m, x, I >$ in L, call $I.f_r(x)$, passing it x, I, and $R = \{x\}$ as extra arguments, and then insert $< key', x >$ in $I.data$, where key' is the result of the computation.

The case of $x? \notin I?.set$ (in step 1) may happen since we are not removing all registration information related to x when an object is deleted from a set. This can happen two ways: some other mutator changes the reachability graph and the objects below the point of mutation still have registrations for x, but are no longer reachable from x, or x has been deleted from $I.set$. Note that if m itself changes the reachability graph, step 3 will register any newly f-reachable objects. Also note that by removing all $< y, m, x?, I? >$ tuples that we are deregistering y, so

y	m	x	I
α	*set_project*	α	I
π	*set_manager*	α	I
β	*set_monthly_rate*	α	I
β	*set_bonus*	α	I
γ	*set_rate*	α	I
γ	*set_bonus*	α	I
δ	*set_monthly_rate*	α	I
δ	*set_bonus*	α	I
ϵ	*set_rate*	α	I
ϵ	*set_bonus*	α	I

Figure 2.10: Registration table after recomputing *project_manager_income*(α) for index I.

if it turns out that y is no longer f-reachable from x?, it will not be reregistered and will not cause another unnecessary update. If y is still f-reachable from x, the recomputation of the key for x will reregister y.

In our example, suppose we invoke $\pi.set\_manager(\delta)$ that causes the example mutation that was shown in Figure 2.1(b). Since there is a registration tuple $< \pi,\ set\_manager, \alpha, I >$, the system would determine the current key value for α in I, remove the $I.data$ pair for α, do the mutation (setting π's *manager_* instance variable to refer to δ), compute $I.f_r(\alpha)$, and insert the appropriate new data pair into $I.data$. Figure 2.10 shows the registration table after this mutation. There are new registration tuples for δ and ϵ involving α from recomputing α's key after the mutation. Also, there are still registrations for β and γ involving α even though they no longer can affect α's key.

This scheme works well if computing $x? \notin I?.set$ and determining the current *key* are efficient. One way to achieve this is to maintain a hash table for a set that maps the OID of each element in the set to its key. In the absence of such a structure, we might like to remove obsolete registration tuples to prevent unneeded work (deregistration is discussed in Chapter 5), or we might keep a copy of the key in the registration tuple.

As an aside, we note that we try to maintain the method dispatch vector of a registered object so that it refers to checking versions for the mutators named in its registration tuples but to regular versions for other mutators so that no overhead is incurred for mutators that cannot affect an index. When there are no registrations, all of the objects of a particular class share the same method dispatch vector; thus when an object is registered for mutator m, it will need a copy of the dispatch vector that is the same as the original except that the entry for m will refer to m_c instead. If we are not careful, there will be a proliferation of dispatch vectors, so we will want to keep track of these new dispatch vectors and share them when possible. Also, when an object is deregistered for a mutator m and we want to convert back to the regular version of m, we need to find an already existing dispatch vector with the appropriate entries instead of creating a new one, if possible.

2.9 Correctness

For our scheme to be correct, at the end of any transaction, for a particular set and index there must be a single entry in the index for every element of the set that has a key that is equal to the result we would get if we executed the index function on the associated set element at transaction commit. We assume the function used to compute the index keys meets our requirements for an index function (that is, it is function of its argument, it is does not cause side effects, and it has a deterministic implementation) and that the dependency information in the specifications of the types used by the index function is correct (that is, all of the mutators that can affect an observer are listed in an observer's *depends_on* declaration).

Our index maintenance scheme is based on registration, so first we argue that $I.f_r(x)$ registers all the objects that can affect x's entry in I. The basic correctness condition is that the set of objects registered for x and I at any given time is a superset of the set of objects that affect x's entry in I at that time. This is true because when $I.f_r(x)$ is computed, all objects reachable from x that $I.f$ could access are in R, and every object in R will be registered if necessary. R is constructed inductively. Initially it contains x, the set element. Whenever a registration version of an observer of x is called, it will add all objects referenced by the instance variables of x to R; also, if the observer depends on any mutators, it will register x. If any observers of objects referenced by the instance variables are called, the call will go to the registration version of these observers and the subobjects will also be registered, if necessary, since they were added to R by the caller. The objects referenced by the subobjects' instance variables will be added to R, and so forth. Thus every f-reachable object from x that can be mutated in a way that affects an observer called in computing $I.f$ will be present in R and registered, if necessary. In addition, our restriction that $I.f$ have a deterministic implementation means that there cannot be any objects that affect the index function that are not accessed during this computation of $I.f$. Therefore, all objects that can affect the entry for x in the index are registered when $I.f(x)$ is computed.

Now we use an inductive argument to show that our scheme maintains indexes correctly. For the basis step of the argument, we must show that index creation is correct. Whenever we create an index, we begin with an empty index and then add to it a pair $< I.f(x), x >$ for each element x. The key is actually computed using $I.f_r(x)$. Since this computes the same result as $I.f$, the proper key is computed, and in addition, the appropriate objects are registered as argued above.

For the inductive step we must consider insertions, deletions, and mutations. We assume that when one of these occur, the index entries have the correct keys and all the necessary objects are registered. Insertion of x to the set simply adds the pair $< I.f_r(x), x >$ to the index; since the index entries were correct before and since we add the correct pair, the index is correct after the insertion. Furthermore, all the appropriate objects are registered since the necessary ones were registered before the insertion and the computation of the key using $I.f_r$ causes the appropriate additional registrations.

When an object x is deleted from a set we remove its index entry and deregister it (for itself as the set element). After deletion, the remaining entries in the index have correct keys since they were correct before the deletion, and all f-reachable objects from the set elements are registered since they were before; the only registration that has been removed is no longer needed.

When an f-reachable object y is mutated in a way that can affect x's entry I, it will be registered for x and I by the induction hypothesis that all objects have been registered correctly. Then effectively we delete x, and after the mutation we insert it. The net effect is correct because deletion and insertion are correct as shown above.

Objects that do not affect the index may also be registered. There are two ways in which excess registrations happen. First, there may be obsolete registrations, since we do not remove them, for example, when an element x is deleted from a set. In this case, the f-reachable subobjects of x are not deregistered, even though they can no longer affect an index entry. We discuss how to remove these obsolete registrations in Chapter 5. Second, even though an observer depends on a mutator in general, it may not depend on that mutator given the current state of the object. For example, suppose a counter object could only be incremented, and its observer *over_100* returns true if its value is over 100. In general, *over_100* depends on the *increment* mutator, but not if the current value of the counter is greater than 100. To determine this case, we would need to be able to prove this property and also monitor the object's state to know when the property's precondition was met. This type of program verification is beyond the scope of this dissertation.

Extra registrations in an object are not harmful, but they may cause unnecessary work. When the registered object is mutated the mutator first checks whether the "element" it is registered for is still in the set and if it is not, no change is made to the index. If the element is in the set, any recomputations will use the current state of the element and will compute the correct key for that element.

Chapter 3

Performance Evaluation

Support for indexes involves a time/space tradeoff: We trade space for the index in exchange for making queries run faster. In our scheme, we need extra space to store the registration tuples in addition to the index itself; these tuples also affect running time because of the time needed to move them between disk and primary storage and the extra time needed to perform registration checks and updates. To evaluate our indexing scheme, we need to get a sense of the tradeoffs involved between the benefits of using indexes and the cost of maintaining them. Our approach is to simulate three possible implementations on a variety of performance benchmarks and analyze several others. This chapter presents the results of the simulations and analysis.[1]

We begin by explaining our simulation model including what objects look like, the three implementation schemes to be simulated, and a system architecture. Next we present the database used in our simulations; it is based on the OO7 benchmark database[14]. In Section 3.3, we describe the benchmarks used to measure the benefit or overhead of our scheme on *queries*, *navigation*, and *updates*. In Section 3.4, we present our experimental framework. We discuss our hypotheses about the performance of function-based indexes in general and the performance of three implementations and present the database and configurations used in our simulations. The results of our simulations are presented in three parts. Section 3.5 shows the benefit of indexes to queries. Section 3.6 presents the results and our conclusions about the effect of disk organization on our benchmarks. Section 3.7 compares three implementation schemes on each of the benchmarks. The simulations are primarily concerned with the running time of our benchmarks. We observed that the space overhead for registration information is fairly high and how it is implemented affects each benchmark in different ways. In Section 3.8, we present a framework for analyzing the space overhead of indexing schemes, suggest several

[1]The simulator used in these experiments is written in CLU compiled using pclu, the portable CLU compiler, and was run on DEC Alpha machines.

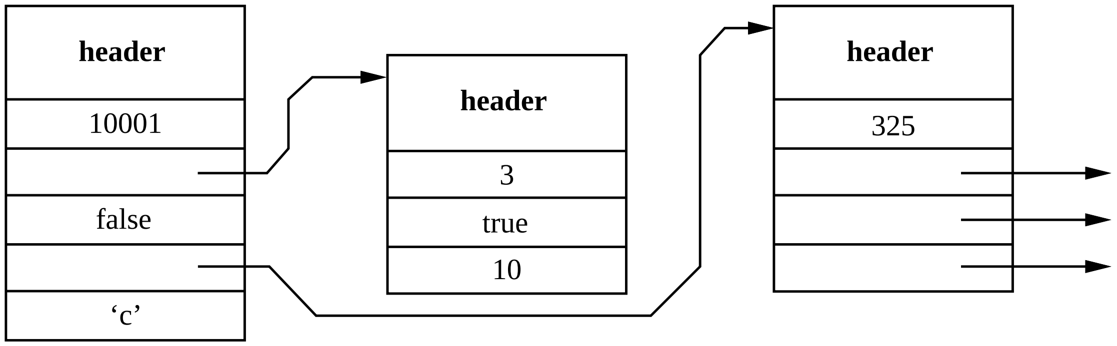


Figure 3.1: Example objects.

other implementations for our scheme with less space overhead than those simulated, and make some general conclusions about our indexing scheme.

3.1 Simulation Model

In this section, we present our simulation model. We begin by explaining how we model objects and indexes. Then we present three possible implementations for our scheme. Finally, we give a description of the system architecture that we simulated.

3.1.1 Objects

An object consists of some number of references and some amount of non-reference data (for example, integers and booleans). We are modeling a system with 64-bit addresses, so a reference takes 8 bytes. An object has a header of 16 bytes that contain information such as the object's unique id (OID) and a reference to its method dispatch vector. Some example objects are shown in Figure 3.1.

An index is modeled as a balanced tree of node objects. Each node of the tree has keys (assumed to fit into 8 bytes) alternating with values (which are references, also 8 bytes). We fixed the number of keys per node at 125, a number chosen to make each node have 2000 bytes of data. We compute the number of nodes in an index in the following manner:

1. The height of the tree is computed as $\lceil \log_{125}(\text{number of values}) \rceil$
2. Starting at the leaves of the tree, for each level of the tree, the number of nodes at the current level is ((total nodes at previous lower level - 1) div 125) + 1

The leaf nodes contain the $< key, x >$ pairs (where x is a reference to a set element) of an index data part. The "values" in higher-level nodes are references to index nodes at the next lower level.

3.1.2 Registration Implementations

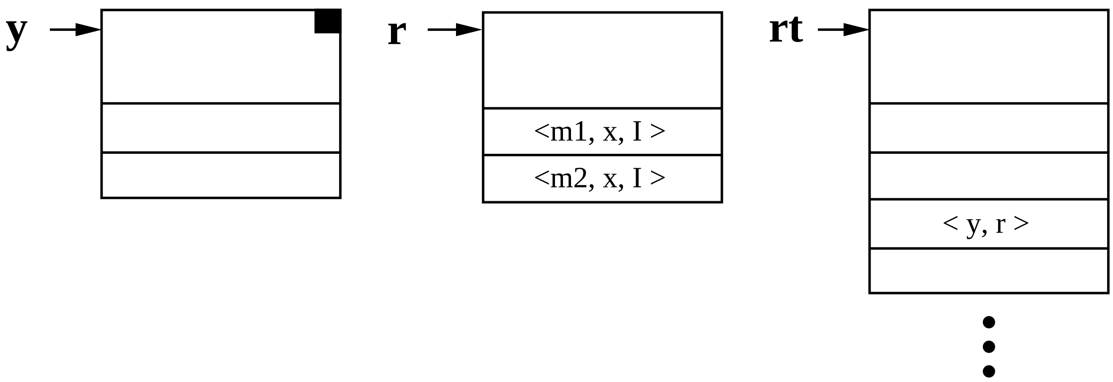
We simulate three implementation schemes. These schemes represent three points in the tradeoff between speeding up queries and slowing down updates. In all three schemes, we assume that there is a bit in the header of an object that indicates whether the object is registered or not. We believe this will not add any space cost since objects have unused bits in their headers. We also assume in the schemes that store registration information inside registered objects that there is no space overhead when objects are not registered.

The first implementation scheme is the *bit scheme*. This is a straightforward implementation of our design. The registration tuples for an object y (i.e., all tuples with y in the first field) are stored in a separate registration object as triples $< m, x, I >$ and this information is accessed by calling methods of the registration object. The registration table is a mapping of the registered objects' OIDs to their registration objects and is modeled as a regular index. Figure 3.2(a) shows this scheme pictorially; y is the registered object as indicated by the black box in the corner representing the header bit, r is the registration object for y, and rt is the registration table. Note that since the registration tuples are stored in an object, each registration object also carries a 16-byte header as extra space overhead.

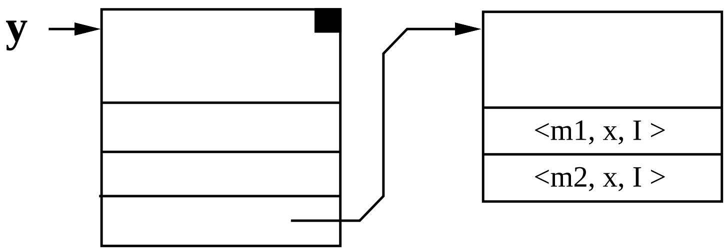
The second implementation scheme is the *pointer scheme*. In this scheme, as in the bit scheme, the registration tuples for a registered object are stored in a separate registration object. However, instead of a registration table, each registered object contains an extra (hidden) instance variable that refers to its registration object. This scheme is shown in Figure 3.2(b); y is the registered object and it stores a reference to its registration object. This scheme is interesting because it trades off an 8-byte reference in a data object to avoid storing a registration table, and allows us to find a registration object directly rather than doing a lookup in the registration table first.

The third implementation scheme is the *embedded scheme*. In this scheme, registration tuples $< m, x, I >$ are stored (directly) inside the registered object. Figure 3.2(c) shows this scheme; the registration tuples are stored as part of y. This scheme is attractive because there is no extra time overhead to find the registration tuples and there is no extra space overhead other than for the registration tuples.

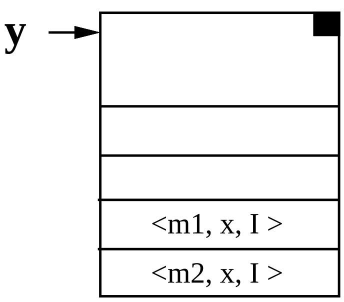
We pack the contents of a registration tuple $< m, I, x >$ into 12 bytes. (The most straightforward representation, one reference to each of m, I, and x, would take 24 bytes.) The reference



(a) Bit scheme



(b) Pointer scheme



(c) Embedded scheme

Figure 3.2: Three implementation schemes.

to x will be a full 8 bytes, but the mutator m can be a very small integer (an object probably does not have more than 256 mutators, so this information can be encoded into a 1-byte integer), and the index I can be encoded as well (for example, as an entry in an "index table" for the system).

3.1.3 System Architecture

The system we are simulating consists of a single client accessing a database on a single server with a disk and a primary memory cache; the server has 64-bit addresses. The server stores is objects on disk in units we call *segments*. Segments are fetched into the cache on demand and are removed using an LRU policy. We assume that the space taken up by a segment in the cache is the same as that taken up on disk. The size of the primary memory cache and the size of a segment are parameters to the simulation.

Note that in the pointer and embedded schemes where registration objects are separate from registered objects, we have a choice of whether to store a registration object in the same segment as the registered object or in a different segment. (This choice does not arise in the embedded scheme, since the registration information is being stored inside the registered object.) Our preliminary results showed that both the pointer and bit schemes with same segment storage performed worse than any of the remaining schemes. They exhibited worse space characteristics than the embedded scheme by spreading out the data objects among even more segments without the benefit to updates that the embedded scheme shows. This was especially true of the bit scheme with same segment storage as each access to a registration object still required a registration table lookup. As a result, we dismissed these cases and only modeled the pointer and bit schemes with registration objects stored in separate segments.

For execution time, we count the number of simulated time steps needed to complete a benchmark. The simulated time steps are meant to approximate machine cycles. We model disk access using two parameters, *disk overhead*, representing the average latency and seek time incurred during a disk access, and *disk transfer rate*. Disk overhead is expressed in milliseconds and disk transfer rate in megabytes per second. These real time parameters are converted to simulated time steps assuming some processor speed. Every disk access incurs both the disk overhead and the transfer time for one segment.

Each benchmark consists of a *trace*, a series of steps (OIDs) representing a series of method calls to objects in the database. The exact nature of these traces is described in Section 3.3. The time cost for a method call is modeled in the following manner:

1. Access to the cache is "free." That is, we assume that there is hardware support to

determine if an object is in the cache that essentially makes it free.

2. If the object is not in the cache, the segment that contains the object is brought into the cache, incurring a disk access. This may throw out a segment from the cache. If the victim segment has modified objects on it, it needs to be written to a "backing store" that is not part of the database storage.

3. Dispatching on an object's method dispatch table and executing a method body that accesses non-reference data is modeled as some fixed time cost. The default is 10 cycles, but this number can be set. (Note that methods that call methods of other objects are modeled by steps in the trace.)

3.2 Simulation Database

The simulated database is based on the OO7 benchmark database[14]. We model the small and medium versions of this database. (The large OO7 database is meant for multi-user tests; we did not do these tests, since we are interested in the basic performance of our scheme.) The OO7 database contains a part library of 500 *composite part* objects, each of which contains some information (like the part's id and build date), and references to a *document* object and the root of a graph of unshared *atomic part* objects. Each atomic part contains some information (like the part's id and build date), and 3, 6, or 9 references to other atomic parts. One reference per atomic part is used to connect the atomic parts into a ring. The other references are chosen randomly. Figure 3.3 shows a pictorial view of a composite part with the root of its atomic part graph and its document header. (The combination of small and medium sizes, and the number of references per atomic part create 6 different databases. We will refer to these databases as "small DB3," "medium DB9," etc. When the number of references per atomic part does not matter, we will just use "small DBs" and "medium DBs.")

In the small DBs, there are 20 atomic parts per composite part, while in the medium DBs, there are 200 atomic parts per composite part. For our simulation, we broke down document objects into a *document header* object that contains some information (like the document's id and date) and references to *document part* objects each containing 2000-byte pieces of text. In the small DBs, the document has 2000 bytes of text, so a document header refers to one document part, while in the medium DBs, a document has 20000 bytes of text, so a document header refers to 10 document parts. We did this to simplify our simulation; otherwise we would have had to simulate how the system deals with objects that are larger than a segment.

In addition to the part library, there is a design tree of assembly objects. At every level

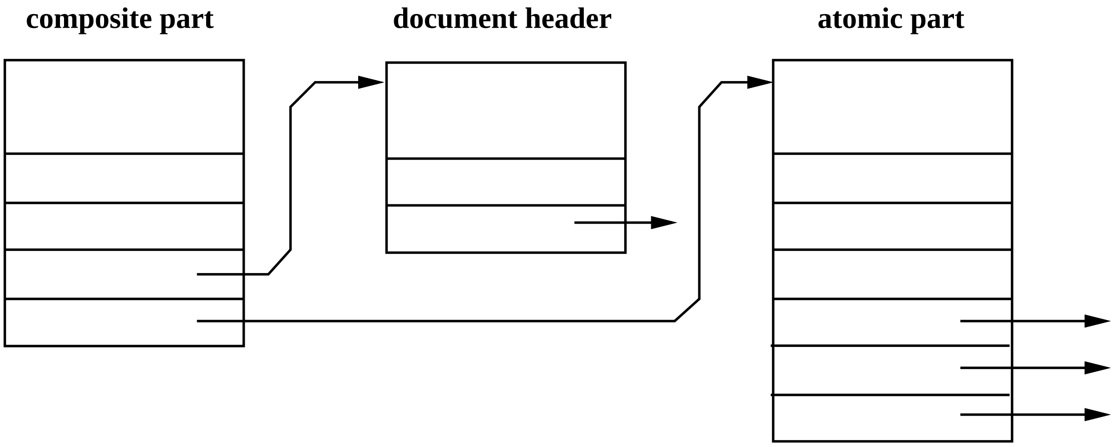


Figure 3.3: A composite part object with its document header and the root of its atomic part graph in the small DB3.

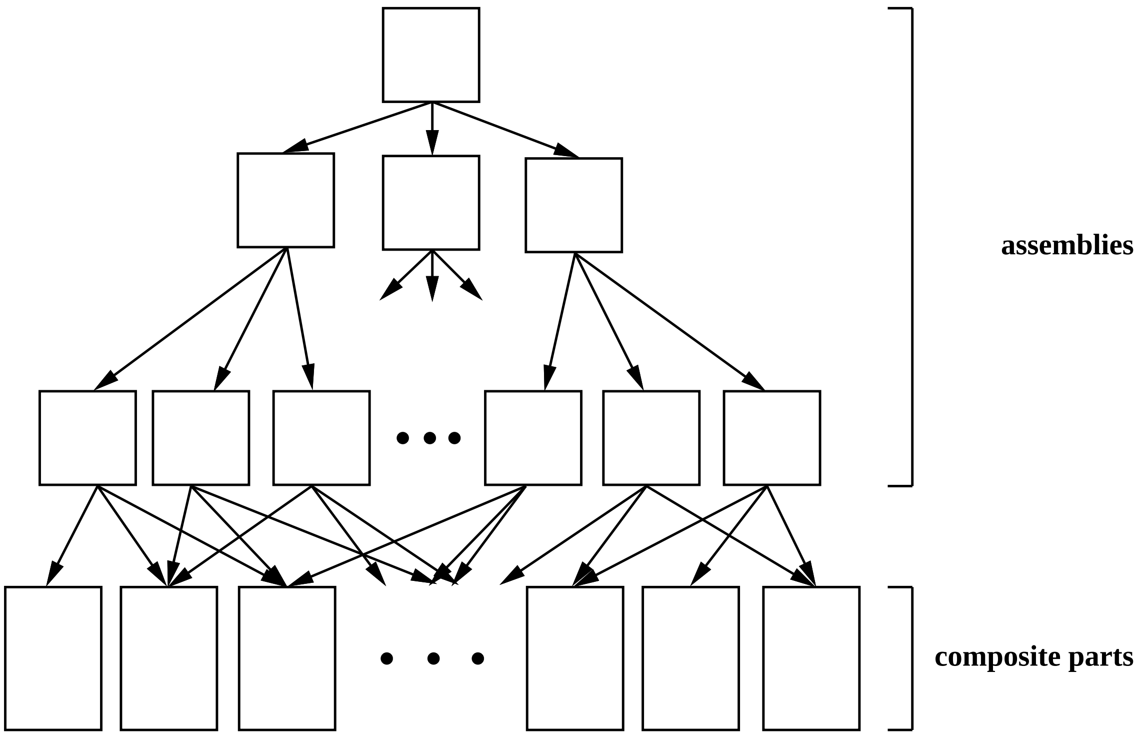


Figure 3.4: OO7 design tree.

	Size (bytes)			
	atomic part (3 refs)	atomic part (9 refs)	composite part	assembly
Object header	16	16	16	16
Non-reference data	24	24	16	8
Reference data	24	72	16	8
Total size	64	112	48	32

	Size (bytes)		
	document header (small)	document header (medium)	document part
Object header	16	16	16
Non-reference data	8	8	2000
Reference data	8	80	0
Total size	32	104	2016

Figure 3.5: Object sizes in the OO7 database.

	Average object size (bytes) excluding document parts
Small DB3	61
Small DB9	101
Medium DB3	64
Medium DB9	111

Figure 3.6: Average size of an object in each database excluding document parts.

of the tree except the base level, each assembly refers to 3 other (unshared) assemblies. An assembly at the base level refers to 3 randomly chosen composite parts (these can be shared among assemblies). The tree is 7 levels in the both the small and medium databases. Figure 3.4 shows a pictorial view of this assembly tree with only 3 levels; the full tree is much larger. Figure 3.5 shows the sizes of various objects in the OO7 database in our model. Figure 3.6 shows the average size of an object in each of the databases excluding the document part objects. We exclude the document part objects because they are never registered and are stored separately from the rest of the database in our simulations.

In most of the benchmarks, we model one index over a set containing all of the data objects (except document parts) based on the *date* method. For the small DBs, this index is 2 levels with one node for the root and 97 nodes at the leaf level for a total of 98 nodes. For the medium DBs, this index has 3 levels with one node for the root, 7 nodes at the middle level, and 817 nodes at the leaf level for a total of 825 nodes.

The objects in the database (except document parts) are clustered into segments in one of two ways. *OID order* models clustering based on reachability. In this clustering, objects are assigned to segments in the order they are created since we create them based on reachability. This ordering is beneficial to navigation since objects that refer to each other are closer together. *Date index order* models clustering based on the date index. In this ordering, objects are assigned to segments in the order of their placement in the date index. This ordering is beneficial to queries using the date index.

3.3 Benchmarks

Our goal is to measure the benefit or overhead of our scheme on *queries*, *navigation*, and *updates*. For queries, we measure the benefits and costs of having indexes on various kinds of queries. In navigation, there is only the cost of the space used by registration information when accessing registered object. Update costs include both the space used by registration information as well as the time needed to do one or more of the following steps: check that an object is registered, access registration information, compute the new key value and update the index, and update registration information. In this section, we describe the benchmarks we simulated to measure these benefits and costs. Figure 3.7 shows a summary of the our benchmarks.

3.3.1 Queries

The query benchmarks model various kinds of queries computed both with and without indexes. In these benchmarks, the keys are distributed uniformly over the set being indexed. Space for the result set is pre-allocated out of the cache to simulate it being kept in the cache for the duration of the run. Its size is proportional to the number of matches expected. There are two query benchmarks to measure the effect of function-based indexes in two situations. In the first query benchmark, we model queries over a set that contains most of the database and use an index function that accesses data present in the set element. We are interested in the cost of registration information when indexes are not used and the effectiveness of primary and secondary indexes for these types of queries. (A *primary* index is one that determines the clustering of the objects into segments. All other indexes are *secondary* indexes.) In the second query benchmark, we model queries over a set that contains relatively few objects of the total

1st Query Benchmark	All data objects (except document parts) are set elements
Query 1	Scanning without registrations
Query 2	Scanning in the presence of registrations
Query 3	Primary index use
Query 4	Secondary index use
2nd Query Benchmark	**Small, unclustered set of composite parts**
Query 5	Scanning with "field access" function
Query 6	Scanning with complex function
Query 7	Secondary index use
Navigation Benchmark	**Read-only traversal of design tree**
Navigation 1	No registrations, clustered in OID order
Navigation 2	No registrations, clustered in date index order
Navigation 3	With registrations, clustered in OID order
Update Benchmark	**Traversal of design tree with writes**
Update 1	No registration checking
Update 2	Only checking registration bit
Update 3	With checking and registration tuple access
Update 4	With checking, registration tuple access, and index updates
Update 5	With checking, registration tuple access, index updates, and registration tuple updates
Traversal types	
Full	Depth-first traversal of entire design tree
Path1	Depth-first traversal of one reference per level
Path2	Depth-first traversal of two references per level

Figure 3.7: Summary of simulation benchmarks.

database using an index function that accesses other objects in addition to the set element during its computation. This measures effectiveness of secondary indexes for these types of queries.

In the first query benchmark, all of the data objects except document parts are elements of the indexed set. The elements of the set are clustered in date index order. In the first three queries, the query is to find the elements that match a range of dates. In the fourth query, the query is to find the elements with a range of particular OIDs. These queries are run several times varying the percentage of matches. Measurements are taken for the following situations:

1. (Query 1) A query with no registered objects and using no indexes. Each object in the set is accessed in clustered order (i.e., the set elements are *scanned*) and a computation is done to determine inclusion.

2. (Query 2) A query using no indexes, but with registered objects. The computation is the same as Query 1.

3. (Query 3) A query using the date index. All set elements are registered. The index is consulted and the matches are accessed.

4. (Query 4) A query using the "OID index". All set elements are registered. We simulate an OID index to find matching OIDs. Then we sort the OIDs in segment order before accessing the matches.

Query 1 is the base case. There is no index, and there are no registrations. Query 2 measures the effect of registration tuples on queries not using indexes. Query 3 measures the utility of primary indexes. Query 4 measures the utility of secondary indexes. We sort the OIDs of the matches in segment order so that all of the matches in a segment will be accessed at the same time. Otherwise, if the cache is smaller than the total number of segments that need to be accessed, there could be thrashing when the query goes back to previous segments to access the additional elements.

The first query benchmark models a very "relational" sort of operation. Most of the database belongs to the set and the information is one of the "fields" of the set elements. We expect that many queries in object-oriented databases will differ in two ways: many sets will contain only a small percentage of the database and query functions will be complex computations involving other objects in addition to the set elements. To explore the utility of indexes for these types of queries, we run a second type of query benchmark. In this benchmark, we are interested in an analysis of a best-case scenario for function-based secondary indexes. We model the following scenario. There is a set containing the composite parts. We are interested in a function that takes a composite part as an argument and computes the number of 'e' characters in the document associated with a composite part by accessing each of the document's parts and counting the 'e' characters in the document part. (A character is 1 byte, but data is accessed in 4-byte words. Thus this function is modeled as 6500 cycles: one cycle each for a shift, mask, and comparison performed per byte over 2000 bytes plus one cycle for each increment when a byte matches; we assume on average one-fourth of the bytes checked match.) We run this query with and without an index. As a basis for comparison, we also run a query over the same set using the composite part's *date* method to model the "field" access type of function on a small set. The data objects are clustered in date order, so the composite parts and their documents are not clustered together. Except in the last query, there are no registrations. Measurements are taken for the following situations:

5. (Query 5) A query on a function that only accesses the date of a composite part without an index or registered objects.

6. (Query 6) A query on a function that accesses all of the document parts of the document contained in a composite part without an index or registered objects.

7. (Query 7) Query 6 using an index with registration tuples in the composite parts and document headers.

Query 5 is the used as a basis for comparison. It represents queries using the simplest kind of function. Query 6 is the base case for our complex function. Query 7 measures the utility of secondary indexes for queries over small sets using complex functions. The difference between Query 6 and Query 7 is the benefit that secondary indexes give to queries using complex functions.

3.3.2 Navigation

The navigation benchmark models non-query access to data that executes methods of an object and follows the references returned by methods. The traces in the navigation benchmarks are read-only traversals of the design tree of the OO7 database. The first traversal is a *full* traversal that starts with the root of the design tree and visits each object in the tree in a depth-first manner. When a composite part is reached, a depth-first search of its atomic part graph is done. (Since there are 729 base level assemblies and they each contain 3 references to composite parts, a composite part may be visited more than once. However, since the composite parts are assigned to base level assemblies randomly, not all of them may have been assigned;

Traversal	Trace sizes							
	Small DB3		Small DB9		Medium DB3		Medium DB9	
	# Steps	# Objs. Accessed	# Steps	# Objs. Accessed	# Steps	# Objs. Accessed	# Steps	# Objs. Accessed
Full	49207	11983	49207	11983	442867	100477	442867	101487
Path1	29	29	29	29	209	209	209	209
Path2	2943	2503	2943	2569	25983	23559	25983	23115

Figure 3.8: Trace sizes.

thus, those that are not assigned are not visited.) This benchmark also contains two path traversals. In the first path traversal (referred to as *path1*), we start at the root of the design tree and randomly pick one reference at each level of the tree to follow. When the base level is reached, one composite part is picked at random, and a depth-first search is done on its atomic part graph. The second path traversal (referred to as *path2*) is the same as the path1 traversal except that at each level two references are randomly chosen to be followed. The number of steps in each trace and the number of objects they access are shown in Figure 3.8.

For this benchmark, we run each traversal several times with a varying number of registration tuples per registered object. When objects are registered, all data objects (except the document parts) in the database are registered. For each run, three types of measurements were computed:

1. (Navigation 1) Traversal with no registered objects and clustered in OID order.

2. (Navigation 2) Traversal with no registered objects and clustered in date index order.

3. (Navigation 3) Traversal with registered objects and clustered in OID order.

Navigation 1 is the base case. Navigation 2 is used to measure the effect of clustering on navigation. Since objects that reference each other are stored closer to each other, OID order should make Navigation 1 faster (fewer disk accesses) than Navigation 2 where objects are stored in date index order. Navigation 3 measures the effect of registration tuples on navigation.

3.3.3 Updates

The update benchmark models a client that changes a field of a set element while navigating through the database. Writes are modeled in the following way. Checking the bit in the object header to determine if the object is registered has a cost of 2 cycles. In the embedded scheme, we assume that registration information is accessed as regular data and cost the same as executing a method body. In the pointer scheme, accessing a registration object is a normal object access and method call. (That is, we add a step to access the registration object into the trace.) For the bit scheme, the registration table must be accessed first to find the registration object associated with the object being written. This is modeled as an access to a node at every level of the registration table index to simulate walking down the tree to find the appropriate leaf. Then the registration object is accessed as in the pointer scheme.

Writes are recorded in a log that must be written to disk (during transaction commit) and applied in place after commit. We ignore reclustering problems; we assume that modifications do not affect object size, and model applying writes as simply writing the new information back to the old object location. (We will come back to this point in the comparison of the simulated schemes.) We assume that the writes from the log are sorted so that all the writes for a segment are applied together. We model new-value logging; the log record for a modified object is the same size as the object, since we assume modifications do not affect object size. (Change-value logging would reduce the size of the log, but the objects we are modeling are fairly small, so the difference would be minor. Also, in the system we are modeling, the time to write the log is dominated by disk accesses during computation, so the incremental savings of writing a smaller log would be relatively small.) In addition, if a write causes an index update, we have to write a log record for that update as well. We assume the log record for an index update is 32 bytes, 8 bytes each for the old key value, the new key value, the reference to the index I, and the reference to the set element x. We assume that a key value will fit into 8 bytes. We may be able to pack this information into fewer bytes, but this is less clear than with the registration tuple. Changes to registration information causes another log record if it is stored in a registration object as in the pointer and bit schemes. (Updates in the bit and pointer schemes may also require new registration objects to be created for data objects that are newly reachable and need to be registered. However, since we are not modeling new objects or objects that change size, we only model the case of registration information being modified and written back in place.)

The update benchmark uses the same traversals as the navigation benchmark (i.e., full, path1, and path2 traversals), and in addition, each step may be randomly chosen to be a write. As explained earlier, a write causes an access to the registration information of the accessed object if the object is registered. This access may result in no action taken (the write does not affect any indexes) or an update to an index (the write affects a key). In the case where the index is to be modified, there may also be changes in the registration information (the write changes the reachability graph).

We run each traversal several times varying the percentage of writes in the trace. In each case, the writes are uniformly distributed over the trace and larger percentages of writes include all writes done in smaller percentages of writes. The objects are clustered in OID order. For each run, five types of measurements are made:

1. (Update 1) Modifications in a system with no indexing.

2. (Update 2) Modifications done in our system, but with no registered objects.

3. (Update 3) Modifications on registered objects, but no index changes.

4. (Update 4) Modifications on registered objects that cause index updates.

5. (Update 5) Modifications on registered objects that cause index updates and registration updates.

All measurements include the time to write back the log and the segments with modified objects at transaction commit. In Update 1 and Update 2 there are no registrations. Update 1 measures the base case where a write does not check to see if an object is registered because there are no registered objects. Update 2 measures the incremental cost of checking for the registration bit in an object's header in our scheme. (Our scheme often avoids this cost since unregistered objects can use mutators that do not do checking.)

For Update 3, Update 4, and Update 5, all data objects (except document parts) are registered with one registration tuple each. Update 3 measures the incremental cost of accessing registration information even when there are no index updates. (Our scheme often avoids this cost since mutators that cannot affect an index generally do not check for registration information in the first place.) Update 4 measures the incremental time to compute the new key(s), write index update log records, and write back the segments containing the index nodes that have been updated. Update 5 measures the incremental time used in Update 4, as well as the time to write the registration object log records, if any, and the time to write back the segments containing the modified registration objects, if any.

3.4 Experiments

We are interested in answering three questions:

1. Are function-based indexes useful in an object-oriented database?

		Number of registration tuples per registered object						
		0	1		2		3	
		Size (bytes)	Size (bytes)	% Inc.	Size (bytes)	% Inc.	Size (bytes)	% Inc.
Emb.	Sm. DB3	1946776	2091892	7.5	2237008	14.9	2382124	22.4
	Sm. DB9	2425776	2571892	6.0	2717008	12.0	2862124	18.0
	Med. DB3	18280408	19505524	6.7	20730640	13.4	21955756	20.1
	Med. DB9	23080408	24305524	5.3	25530640	10.6	26755756	15.9
Ptr.	Sm. DB3	1946776	2382124	22.4	2527240	29.8	2672356	37.3
	Sm. DB9	2425776	2862124	18.0	3007240	24.0	3152356	30.0
	Med. DB3	18280408	21955756	20.1	23180872	26.8	24405988	33.5
	Med. DB9	23080408	26755756	15.9	27980872	21.3	29205988	26.5
Bit	Sm. DB3	1946776	2482948	27.5	2628364	35.0	2773180	42.4
	Sm. DB9	2425776	2962948	22.1	3108064	28.1	3253180	34.1
	Med. DB9	18280408	22802212	24.7	24027328	31.4	25252444	38.1
	Med. DB9	23080408	27602212	19.6	28827328	24.9	30052444	30.2

Figure 3.9: Total database size and percentage increase in size for databases, including 1 index, when all data objects (except document parts) are registered.

2. What is the effect of different segment sizes on our scheme?

3. How should function-based indexing be implemented?

In this section, we present our experimental framework. we describe the database configurations and system configurations that we chose to simulate. Then we discuss our hypotheses about each of these questions.

3.4.1 Database Configurations

We wanted to explore several points in the design space to test our hypotheses. We created four databases: small DB3, small DB9, medium DB3, and medium DB9. We did not run simulations on the DB6 databases because we could interpolate the results between the DB3 and DB9 databases.

Figure 3.9 shows the total number of bytes required for our databases, including one index, when all data objects (except document parts) are registered with varying numbers of registrations. We note that the pointer and bit schemes have fairly high space overhead, due to the object header overhead that every registration object incurs. In Section 3.8, we discuss other implementations with lower space overhead. However, we expect that disk space costs will continue to decrease making it feasible to cover our space overhead in return for the benefit that indexes provide. We also expect that in real systems many objects will not be registered at all, so that the average number of registrations tuples per object will be small. Also, as the average size of an object becomes larger, the space overhead of registering an object becomes a smaller percentage of the total database size as shown for the medium DBs where the atomic parts and documents are larger. In general, we simulated systems with one registration per registered object.

We configured the database so that data objects (except document parts) are clustered into *data segments* according to whatever clustering order (OID or data index) is chosen. The document parts are clustered together in separate *document segments*. The index nodes are clustered together in separate *index segments*. Registration objects in the pointer and bit schemes are clustered together in separate *registration segments*. The registration table nodes in the bit scheme are clustered together in separate *registration table segments* as well.

3.4.2 System Configurations

Processor speed is unlikely to have a qualitative effect on the results of our benchmarks, since they are I/O bound. In order to equalize the computational and I/O portions of our benchmarks, a processor would have to be much slower than what is currently available. A very fast processor would make the computational portions of the benchmarks faster relative to the I/O portions exacerbating the problem. We chose to simulate a 125Mhz processor. (That is, 1 millisecond = 12,500 simulated time steps.) This approximates a DEC Alpha, the hardware being used by the Thor project.

For disk speed, we chose three settings: slow, corresponding to a 30 millisecond overhead and a 2 megabytes per second transfer rate (this corresponds to mid-1992 off-the-shelf speeds[17]); medium, corresponding to a 15 millisecond overhead and a 5 megabytes per second transfer rate; and fast, corresponding to a 5 millisecond overhead and a 10 megabyte per second transfer rate. For segment sizes, we chose 2, 8, and 16 kilobytes (*2K*, *8K*, and *16K* segments, respectively).

For cache size, we chose 2 megabytes. This is large enough that the objects accessed in the small DBs will fit into the cache. Although we can simulate a cache that is large enough to contain the medium DBs (20 megabytes), we chose not to do so, because there will always be databases that do not fit into memory, and we wanted to run some experiments in this region of the design space.

We ran all benchmarks for each combination of parameters on the small DB3 and the small DB9 with a cold cache. After looking at the results, we concluded that disk speed did not affect the relative performances of the different implementation schemes in the system we are

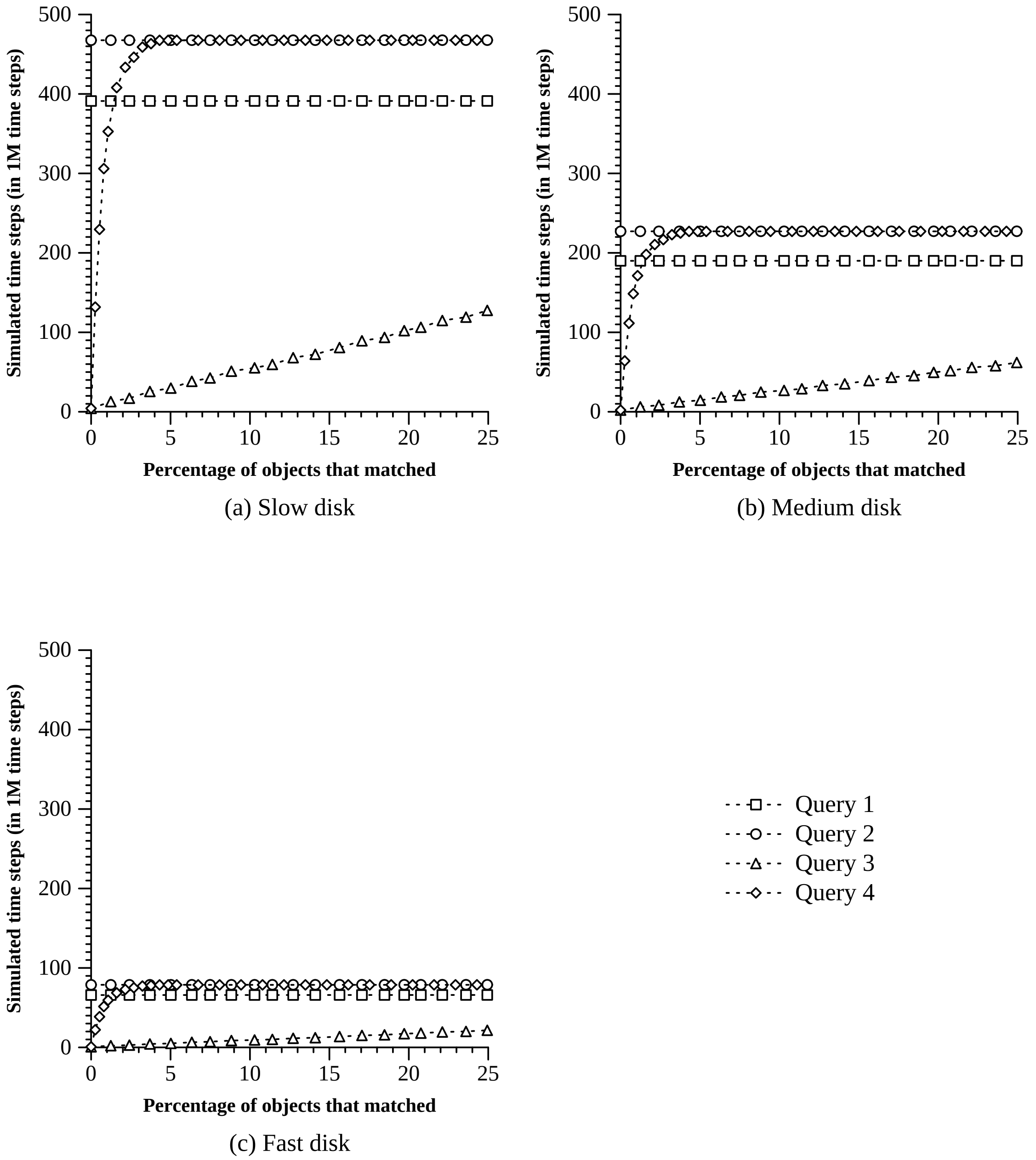


Figure 3.10: Effect of disk speed on embedded scheme query benchmark results.

modeling, since in all cases the computations are I/O bound. That is, the benchmarks completed in fewer simulated time steps and the graphs were flatter when simulating faster disk speeds, but the various crossover points were the same since the same number of segments needed to be accessed in each case. For example, Figure 3.10 shows the results for the embedded scheme on one of the query benchmarks for the different disk speeds. The results themselves will be explained in Section 3.5, but the point to note here is that the curve for Query 4 crosses over the curves for Query 1 and Query 2 at the same place along the x-axis in all three graphs. Since disk speed was no longer significant for our study, we did not run benchmarks with medium and fast disk speed settings for the medium DBs. All of the data we present in the rest of this dissertation are the results of using the slow disk speed setting.

We note that there are high performance disk systems (e.g., RAID systems) that can transfer data faster than we modeled. However, such systems also require that data be organized carefully to take advantage of this transfer rate, since otherwise the cache will be filled with uninteresting data. This dissertation is concerned with the general characteristics of our indexing scheme, and modeling how our simulated database could be arranged on one of these disk systems is outside the scope of this dissertation.

Our results showed that segment size does not affect the relative performance of our implementations. We discuss the effect of segment size on overall performance on our benchmarks in Section 3.6. In all other discussions, we will present only the results for 8K segments.

Additionally, we ran the benchmarks for the small DB3 with 8K segments on a warm cache to measure the performance of our scheme when the entire database fits into memory and is preloaded into the cache. This was done to simulate our scheme for main memory databases.

3.4.3 Hypotheses

Our main hypothesis is that function-based indexing is very beneficial for queries in object-oriented databases. We expect that running function-based queries without the benefit of indexes will be very expensive, especially if the function accesses many objects.

Since for any particular disk, average latency is the same regardless of segment size (up to some maximum), the average time to read a byte off a disk depends on the segment size. Larger segments mean the latency overhead is amortized over more bytes so the average time per byte is lower than with smaller segment. We hypothesize that this makes larger segments better when we expect to read large amounts of data that are likely to be stored together, but if we are interested in small amounts of data spread out over many segments, then smaller segments are better since we do not waste time reading in large amounts of unusable data.

		# registered objects	Reg. info in data objects	Reg. objects	Reg. table
Emb.	Sm. DB	12093	145116	0	0
	Med. DB	102093	1225116	0	0
Ptr.	Sm. DB	12093	96744	338604	0
	Med. DB	102093	816744	2858604	0
Bit	Sm. DB	12093	0	338604	197568
	Med. DB	102093	0	2858604	1663200

Figure 3.11: Size (bytes) and breakdown of overhead for databases when all of the data objects (except document parts) are registered with one tuple.

For the comparison of the three implementation schemes, our hypothesis is that each of the implementation schemes is better for different types of computations due to the amount and placement of registration information in each of the schemes. The space overhead incurred by each scheme can be broken down into three categories: space taken up by registration information stored in a data object, space taken up by registration objects, and space taken up by a registration table. Figure 3.11 shows a breakdown of the space overhead for the small and medium DBs, including one index, when all data objects (except document parts) are registered with one registration tuple each.

Since all of the space overhead in the embedded scheme is stored inside data objects, we expect that it will perform poorly on queries and navigation. The registration tuples causes the database to be spread out over more segments, requiring more disk accesses to read in the same amount of data. In addition, as more registration tuples are added, the cost increases. On the other hand, we expect the embedded scheme to perform the best on updates, since the registration tuples are always available when a write occurs.

Likewise, we expect the bit scheme will add no overhead to queries and navigation, since the data objects are exactly the same whether or not there are any indexes. However, we expect that updates will suffer because the registration table must be accessed (which will cause one or more extra accesses that may result in disk accesses) and a registration object must be accessed in order to do a write.

We expect the pointer scheme to be somewhere in the middle for all types of benchmarks. The extra reference per registered object should have an effect on queries and navigation not unlike the embedded scheme with one registration tuple, but adding registration tuples to the registration object will not increase the cost for these computations. For updates, a registration object must be accessed when there is a write, but since the reference is stored in the data object, there is only one extra access rather than two or more as in the bit scheme.

3.5 Benefit of Indexes

The benefit of indexes can be seen in the results for the query benchmarks. For the purposes of showing the benefit of indexes, the graphs for the embedded and pointer schemes are nearly the same as for the bit scheme. Therefore, we will only show the graphs for the bit scheme in this section. Graphs including results for the pointer and embedded schemes will be shown later when we compare the performance of the three schemes. Also database size and segment size do not affect our conclusions, so we will only show the results for the small DB3 with 8K segments in this section.

Figure 3.12 shows the results of running the first query benchmark for the small DB3 with 8K segments using the bit scheme on a cold cache. The curves for Query 1 and Query 2 are the same in the bit scheme, so the result for Query 2 has been omitted from this graph. Recall that the queried set contains all of the data objects (except document parts), Query 1 is a query computed without an index by scanning the set, Query 3 is a query computed using a primary index, and Query 4 is a query computed using a secondary index. We see from the results of Query 3 that primary indexes are very useful as expected. The number of segments that need to be read in is proportional to the percentage of matches; thus only the minimum number of segments need to be read into the cache.

For secondary indexes, we are interested in the point at which the time to use an index crosses over the time to scan the entire set, that is, where the curve for Query 4 crosses the curve for Query 1. This compares using a secondary index with scanning a set (i.e., not using an index). We see that the percentage of matches where this happens is fairly low, about 4.5%. At this percentage of matches, there is one match per data segment, so we are reading in all of the segments as would happen when scanning the set.

Most of the cost of running queries with a cold cache is due to the time spent reading data segments from the disk. Figure 3.13 shows the results of running the first query benchmark with a warm cache. We see that primary indexes are still beneficial, since it allows us to avoid computing the *date* method for every element of the set. We also see that that secondary indexes are nearly as beneficial in this situation. (The curves are not exactly the same because the simulation still sorts the matches into segment order when using a secondary index. If we take out this cost, primary and secondary index use would be the same.)

Figure 3.14 shows the results of the second query benchmark for the small DB3 with 8K

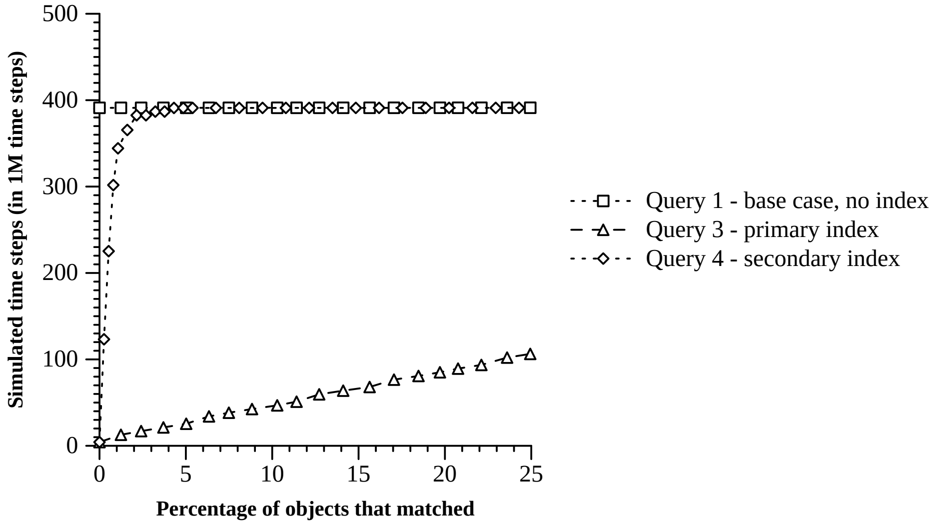


Figure 3.12: Results of Queries 1–4 on the small DB3 with 8K segments using bit scheme with a cold cache.

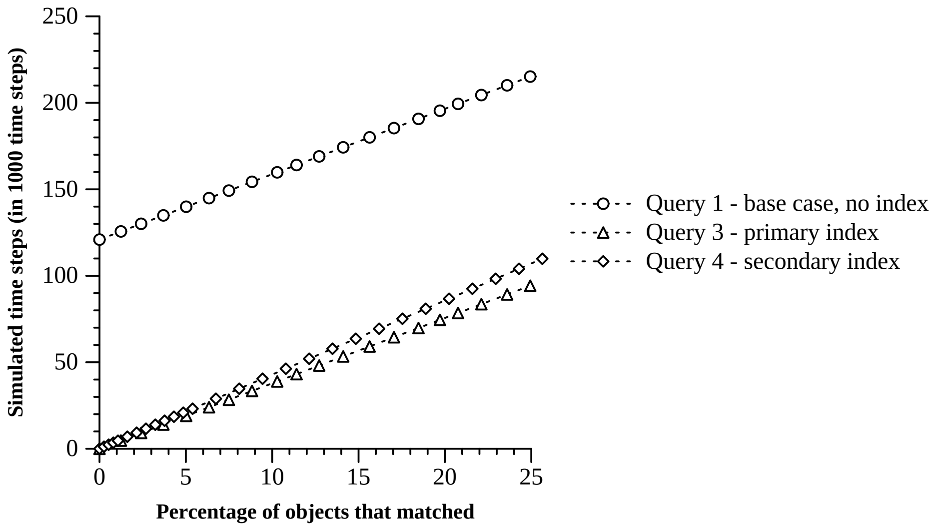


Figure 3.13: Results of Queries 1–4 on the small DB3 with 8K segments using bit scheme with a warm cache.

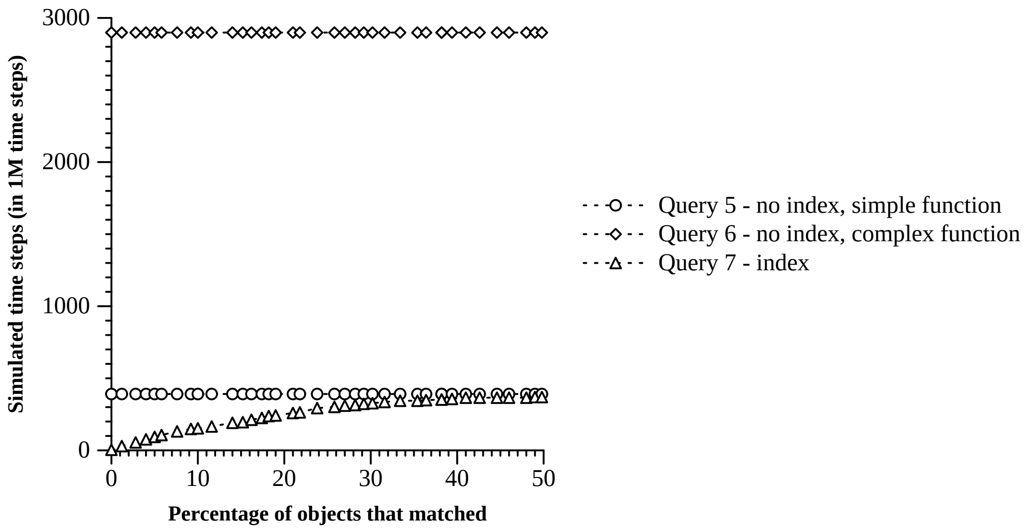


Figure 3.14: Results of Queries 5–7 for the small DB3 with 8K segments using bit scheme with a cold cache.

---

segments using the bit scheme with a cold cache. Recall that the queried set contains all composite parts, Query 5 scans the set and accesses a "field" of the composite part, Query 6 scans the set and computes a complex function counting the number of 'e' characters in the composite part's document, and Query 7 uses an index to compute the same query as Query 6. The immediate thing to notice is that the curve for Query 7 will never cross the curve for Query 6. Clearly in this situation, a secondary index is a big win even though the data objects accessed are scattered across various segments. This is because the index avoids reading in the document segments that are needed compute the query function.

We note that the crossover point for Query 7 with respect to Query 5 is very high in contrast to the similar type of query in the first benchmark. This is because the set is small, but (potentially) spread out over all of the segments of the entire database. At each percentage of matches, there are many fewer objects that match than in the first query benchmark, thus it takes a higher percentage of matches to reach the situation in which there is one match per data segment. Also, the index used in Query 7 basically makes accessing the results of a complex function equivalent to a method that accesses a "field" of an object, thus Query 5 represents the upper bound on the time steps for Query 7. As we approach a match percentage of 100%, Query 7 will have accessed the same data segments as Query 5.

Figure 3.15(a), on the left, shows the results of running the second query benchmark with a warm cache. We see that even when all of the data is resident in the cache, computing the complex function is significantly more expensive than computing a field access. The results

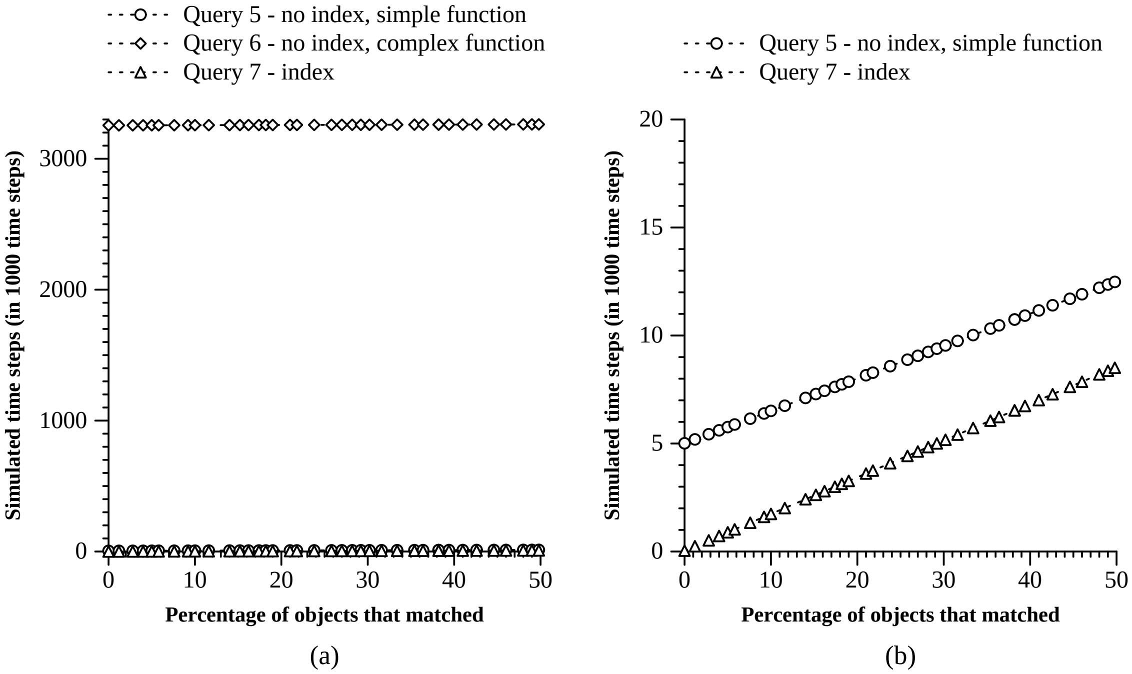


Figure 3.15: Results of Queries 5–7 for the small DB3 with 8K segments using bit scheme with a warm cache shown in graph (a). Graph (b) shows the details of the results of Query 5 and Query 7 on a different scale.

of Query 5 and Query 7 are not distinguishable at this scale. Figure 3.15(b), on the right, compares the results of Query 5 and Query 7 using a different scale to show the area of detail. We see that computing a query using an index is more efficient than computing a query without an index when the cache is warm.

From these results, we conclude that function-based indexes are useful for object-oriented databases in the following situations:

1. As a primary index of a large set with either a cold or warm cache.
2. As a secondary index of a large set with a warm cache.
3. As a secondary index of a large set with a cold cache when the percentage of expected matches is small.
4. As a secondary index of a small set with elements taken from a much larger set with either a warm or cold cache. These indexes are especially effective in speeding up queries using complex functions.

In addition, we note that the results of the second query benchmark show the benefits of precomputing results for either warm or cold caches. If we could access the keys we have already computed when the index function is invoked on a set element, the cost of that function call will become equivalent to a method that does a field access. We return to this issue in our discussion of future work in Chapter 7.

3.6 Effects of Segment Size and Clustering

Recall that we hypothesize that larger segments are better for performance when we expect to read large amounts of data that are likely to be stored together, but that if we are interested in small amounts of data spread out over many segments, smaller segments are better since we do not waste time reading in large amounts of unusable data. In this section, we compare the results of the bit scheme on the query and navigation benchmarks for several segment sizes and draw some general conclusions. We only consider runs with cold caches because the difference in segment size is not relevant once the data is resident; the fact that it was transferred into the cache in smaller or larger units no longer matters. As in the previous section, the results for the pointer and embedded schemes with respect to segment size and clustering are nearly the same as for the bit scheme and will not be shown. We did not run the benchmarks for the medium DBs using 2K segments, because they take a long time to run, and we did not think we would learn anything new after looking at the results for the small DBs using 2K segments.

3.6.1 Queries

For queries, differences in segment size are most noticeable when we compute the result by scanning the queried set (e.g., Query 1). Figure 3.16 shows the simulated time steps taken to complete Query 1 using the bit scheme on various segment sizes. We see that segment size is inversely related to total running time. When the segment size is larger, fewer time steps are taken to complete the queries than when the segment size is smaller. This effect is due to the fact that we are reading nearly all of the data objects into the cache to compute Query 1. Since every disk access incurs disk overhead time, the smaller segments mean we pay disk overhead more often. In particular, we see that the overhead of 2K segments is severe for the small DBs compared to 8K and 16K segments, and we see some of the same severity on the medium DBs going from 8K to 16K segments. Of course, the number of disk accesses needed to read the entire database could be reduced by various schemes, such as scanning multiple segments off the disk at each disk access. This type of disk behavior is captured somewhat by our use of larger segments. However, modeling this behavior precisely is beyond the scope of this dissertation.

We noted in the previous section that the crossover point for using a secondary index with a cold cache versus scanning the set (i.e., Query 4 versus Query 1) is fairly low for the small DB3

Seg. size	Query 1 execution time Time steps (millions)			
	Small DB3	Small DB9	Medium DB3	Medium DB9
2K	1425	2370	–	–
8K	391	642	3410	5917
16K	223	360	1899	3295

Figure 3.16: Execution time of Query 1 using the bit scheme.

Seg. size	Query 4 vs. Query 1 Crossover Percentages			
	Small DB3	Small DB9	Medium DB3	Medium DB9
2K	24	62	–	–
8K	4.5	8	61	74
16K	1.9	3.2	15	50

Figure 3.17: Query 4 versus Query 1 crossover percentages for bit scheme.

with 8K segments. Figure 3.17 shows the crossover points of Query 4 versus Query 1 using the bit scheme for the other databases and segment sizes. We see that the crossover points depend on the segment size and average object size. Systems with larger segments have crossover points at lower percentages than systems with smaller segments. Again, this has to do with the number of segments that need to be read in to answer the query. Larger segments create a situation in which it is more likely that there will be a match in every segment, whereas the opposite is true of smaller segments. This effect is also seen as objects get larger; for example, the crossover point for the small DB3 is lower than for the small DB9.

The running times for Query 5 (scanning a set of composite parts using a "field" access function) and Query 6 (scanning a set of composite parts using a complex function) are also affected by the segment size. Figure 3.18 shows the running times of Query 5 and Query 6. For the small DBs, the effect is the same as on the first query benchmark; larger segments allow the benchmark to complete in fewer time steps than smaller segments. Larger segments makes it more likely that there will be multiple matches per segment. However, for the medium DBs we see a different story. The benchmark takes more time steps with 16K segments than with 8K segments. This is because the larger database causes the matches to be spread out over more data segments, so that it is unlikely that there will be multiple matches per segment in Query 5 even if the segments are larger. As a result, more bytes are transferred into the cache

Seg. size	Comparison of Query 5 and Query 6 — Time steps (millions)							
	Small DB3		Small DB9		Medium DB3		Medium DB9	
	Qu. 5	Qu. 6	Qu. 5	Qu. 6	Qu. 5	Qu. 6	Qu. 5	Qu. 6
2K	1073	4070	1336	4593	–	–	–	–
8K	391	2899	633	3400	1653	24527	1827	24841
16K	222	2840	360	3095	1750	27100	2078	27665

Figure 3.18: Execution times for Query 5 and Query 6.

with larger segments than with smaller segments.

3.6.2 Navigation

Figure 3.19 shows the results of Navigation 1 (data clustered in OID order) and Navigation 2 (data clustered in date index order) on the full traversal using the bit scheme with a cold cache. The main point of this result is that navigation works fine when all of the data objects accessed fit in the cache. For the small DBs, it is still possible to navigate efficiently when the data is clustered in date index order. In this case, systems with larger segments take fewer time steps than systems with smaller segments, since we are navigating through most of the database.

However, when the accessed data objects do not fit in the cache, the clustering scheme is the main issue. The results for the medium DBs on Navigation 1 shows that navigation works fairly well when objects are clustered in OID order. Results for the medium DBs on Navigation 2 show that navigation performance degrades severely when objects are clustered in date index order. This is due to thrashing as the traversal hops to (potentially) different segments on each access and (potentially) causing segments with useful data to be thrown out before that data is accessed. In this case, we also see that that larger segments exacerbate this phenomenon.

The results of the path2 traversal are similar to the results of the full traversal. This is because the path2 traversal accesses enough objects for there to be objects in nearly every data segment. Clustering becomes an issue in the path1 traversal even for the small DBs. Figure 3.20 shows the results of Navigation 1 and Navigation 2 on the path1 traversal. When the database is clustered in date index order (Navigation 2), the performance of the path1 traversal degrades severely compared to its performance in the database clustered in OID order. This is because the path1 traversal accesses very few objects. They are likely to be clustered into a few segments when the database is clustered in OID order, but are likely to be in many different segments when the database is clustered in date index order, so Navigation 2 accesses many

Seg. size	Comparison of Navigation 1 and Navigation 2 (full traversal)							
	Time steps (millions)				Time steps (billions)			
	Small DB3		Small DB9		Medium DB3		Medium DB9	
	Nav. 1	Nav. 2	Nav. 1	Nav. 2	Nav. 1	Nav. 2	Nav. 1	Nav. 2
2K	1410	1422	2378	2366	–	–	–	–
8K	387	387	638	638	16.1	1088	27.7	1251
16K	218	218	355	355	12.6	1395	19.8	1503

Figure 3.19: Execution time of Navigation 1 and Navigation 2 on the full traversal. Note that time steps for the medium DBs are in billions.

Seg. size	Comparison of Navigation 1 and Navigation 2 (path1 traversal)							
	Time steps (millions)							
	Small DB3		Small DB9		Medium DB3		Medium DB9	
	Nav. 1	Nav. 2	Nav. 1	Nav. 2	Nav. 1	Nav. 2	Nav. 1	Nav. 2
2K	15	108	19	108	–	–	–	–
8K	13	102	17	111	21	672	26	680
16K	14	104	14	114	14	762	24	757

Figure 3.20: Execution time of Navigation 1 and Navigation 2 on the path1 traversal.

more segments than Navigation 1. We also see that larger segment sizes benefit Navigation 1 since it is likely that the objects it accesses will be clustered into fewer segments. The opposite is true for Navigation 2; since the objects that are accessed are spread out, they still end up in different segments and the larger segments bring more unused data into the cache.

3.6.3 Conclusion

The general conclusion we make is that segment size, clustering, and cache size are important factors in determining the running time for various computations. Segment size affects the time it takes to get accessed objects off disk. Clustering affects whether the objects that are accessed together are in the same segment. Cache size affects the number of segments that can be in the cache at any given time. Most of our benchmarks are computations that access nearly all of a database. When the cache size is large enough to hold all objects accessed, clustering is not a factor and large segments are beneficial by reducing disk overhead. When the cache is not large enough, clustering becomes the dominant factor. If the objects are clustered to match the pattern of access, large segments are again beneficial. If the objects are not clustered to match the pattern of access, large segments are detrimental, since they fill the cache with more unused objects than small segments, effectively making the cache smaller. Determining the optimal segment size, clustering, and cache size is dependent on the workload expected.

3.7 Comparison of Implementation Schemes

In this section, we compare the performance of each of the implementation schemes on each benchmark. Since segment size does not change the relative positions of the implementations, we show only the graphs for the simulations using 8K segments. Generally, database size also does not affect the relative performance our the implementations, so most of the graphs will be the results for the small DB3 except where database size makes a difference.

3.7.1 Queries

In this section, we compare the performance of the three implementations schemes on our query benchmarks. First, we note that the results of running the query benchmarks for the small DB3 with 8K segments with a warm cache show that that once all of the data is resident in memory, all schemes performed exactly the same, since they make the exact same computations.

For the first query benchmark, we begin by comparing their performance on Query 1 and Query 2 (scanning a large set without and with registration information, respectively). Figure 3.21 shows the results for the small DB3 with 8K segments with a cold cache. The differences in performance on Query 2 shows that registration information stored inside a data object has an effect on scanning. This is due to the effect that larger data objects have on clustering. The space overhead stored inside a data object causes the database to occupy more segments, so that reading in the entire set takes more time. As expected, the bit scheme has no degradation (i.e., Query 1 and Query 2 are exactly the same), since the objects continue to be clustered exactly the same in both cases. The extra reference per object in the pointer scheme is enough to disturb the clustering, but is slightly better than the embedded scheme with one registration tuple. There is a 13.0% increase in running time in the pointer scheme and a 16.4% increase in the embedded scheme. The pointer and bit schemes will not be affected by additional registration tuples. However, the embedded scheme will perform worse as more registration tuples are added since this increases the size of the data objects and will cause more disk accesses during scanning

Figure 3.22 shows a comparison of the results for Query 3 (primary index use) for the small DB3 with 8K segments with a cold cache. We note that the bit scheme is better than the pointer scheme which is slightly better than the embedded scheme with one registration tuple.

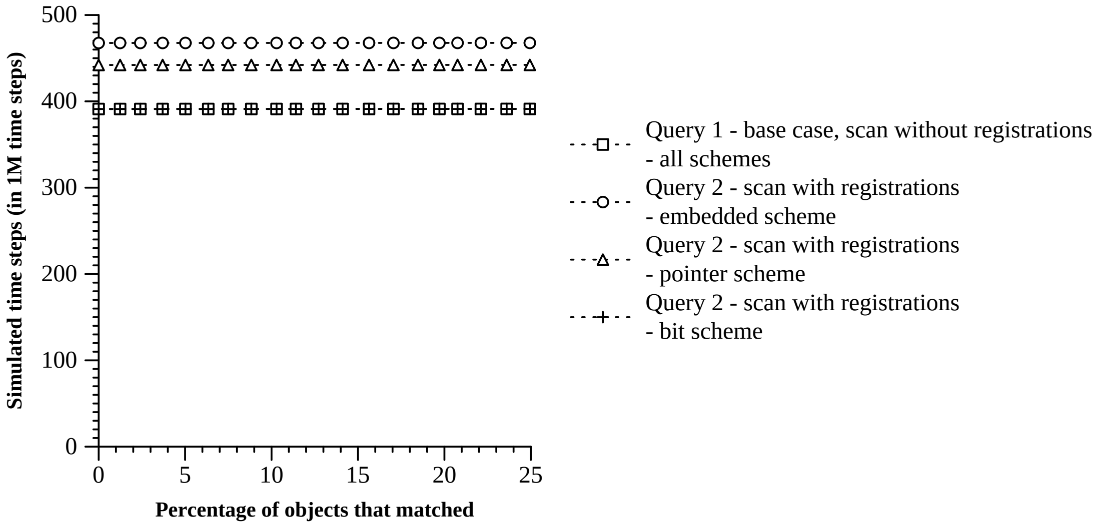


Figure 3.21: Comparison of results of Query 1 and Query 2 for the small DB3 with 8K segments with a cold cache.

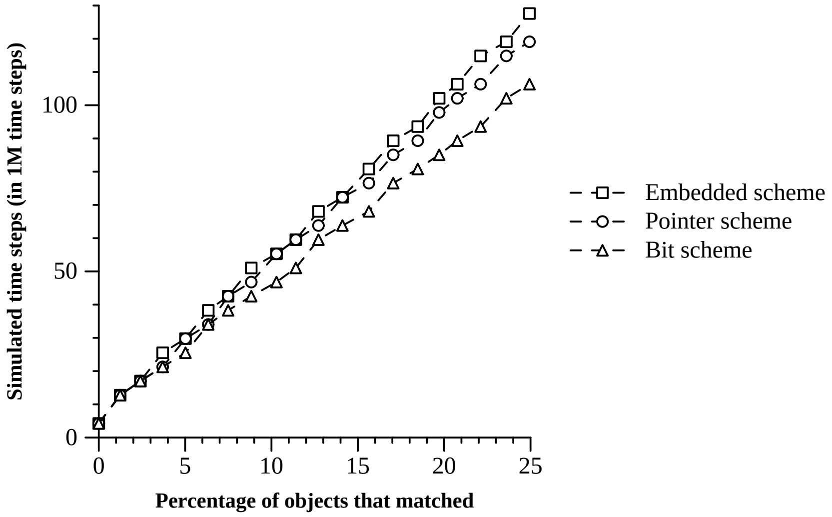


Figure 3.22: Comparison of results of Query 3 on the small DB3 with 8K segments with a cold cache.

Again, this is due to the effect that the pointer and embedded schemes have on the clustering of the data objects. Fewer objects fit into a segment than in the bit scheme, so both the pointer and embedded schemes have to read a few more data segments to get the same number of matches. Note that as we add more registrations per registered object, the performance of the embedded scheme will degrade whereas the pointer and bit schemes will not.

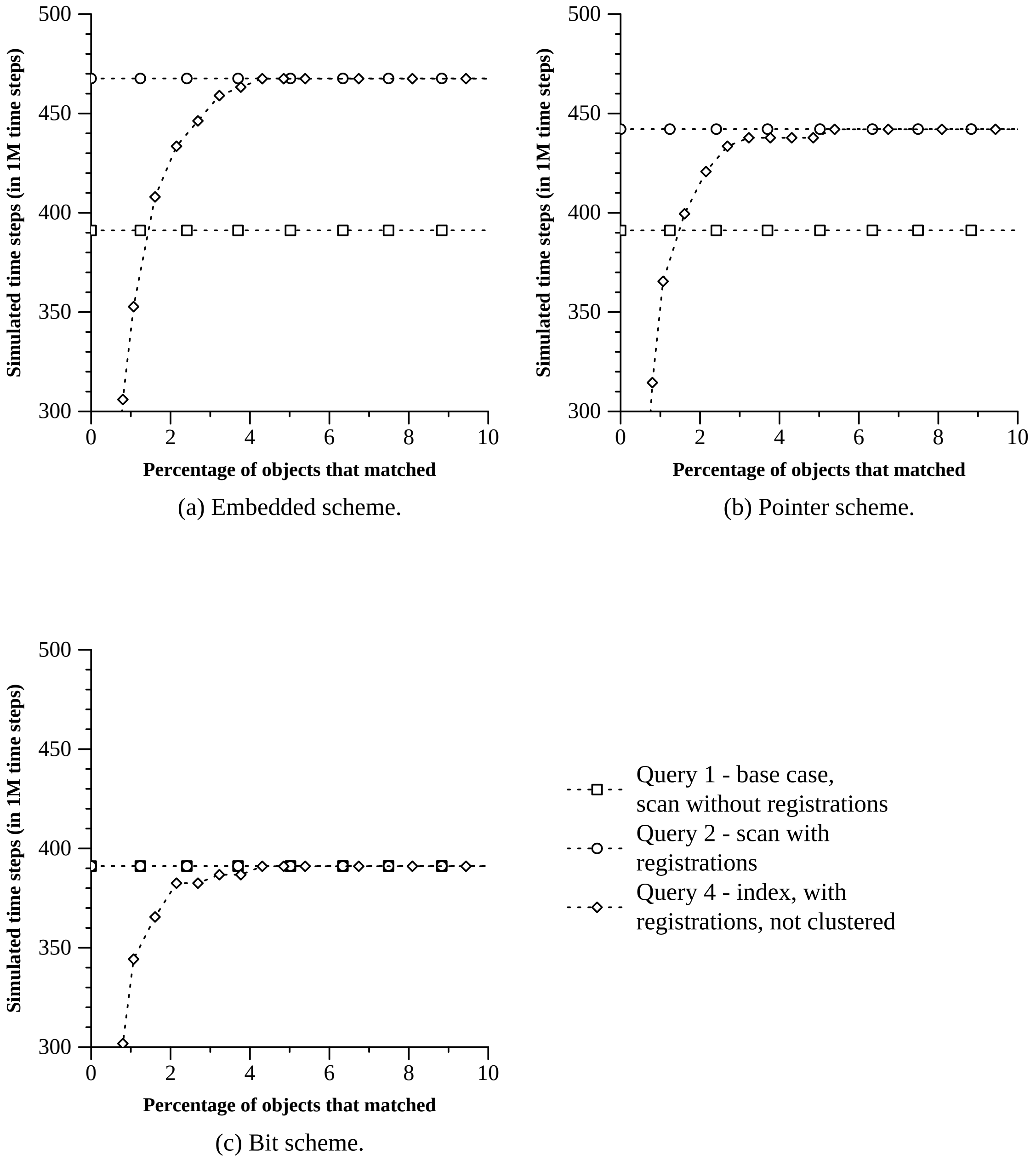
When we compare the schemes on Query 4 (secondary index use), we see that there are two meaningful crossover points. One is where the curve for Query 4 crosses the curve for Query 1; the other is where the curve for Query 4 crosses the curve for Query 2. In the first case, we are comparing secondary index use with not using indexes at all. In the second case, we are comparing secondary index use with scanning when there are registration tuples for another index, or with scanning instead of using the index. Figure 3.23 shows the area of interest in detail for each of the schemes for the small DB3 using 8K segments. In both cases, the main time cost for completing the query is from disk accesses and at the crossover points there are matches in every data segment. We find that the crossover points for secondary index use depend on the implementation scheme. We see that crossover is at about 1.5% with respect to Query 1 for the embedded and pointer schemes. With respect to Query 2, the embedded scheme crosses over at about 4.5% and the pointer scheme at about 5.5%. The crossover points in the bit scheme are the same, since Queries 1 and 2 are the same, at about 4.5%.

For the second query benchmark, we do not have to compare the results for Query 5 and Query 6 (scanning a small set with "field" access and complex functions, respectively); since there are no registrations, the schemes perform exactly the same for these queries. Figure 3.24 compares the performances on Query 7. We see that there is very little difference between the schemes. The reason for this is that there are relatively few registrations and the matches are not clustered together, so the registration information stored inside the data objects in each scheme has only a slight effect on the layout of the data compared to when there are no registrations.

3.7.2 Navigation

For navigation, we are interested in the effect of registration information on running time in each of the implementations. First, we note that running the benchmark with a warm cache shows that there is no difference in any of the schemes. Again, this is because all data is resident at the start of the traversals and all scheme do the exact same computations, so there is no difference whether or not there are any registrations.

Figure 3.25 shows the results of Navigation 1 and Navigation 3 (traversal without and with



(a) Embedded scheme.

(b) Pointer scheme.

(c) Bit scheme.

Figure 3.23: Detail of the results of Query 4 on the small DB3 with 8K segments with a cold cache.

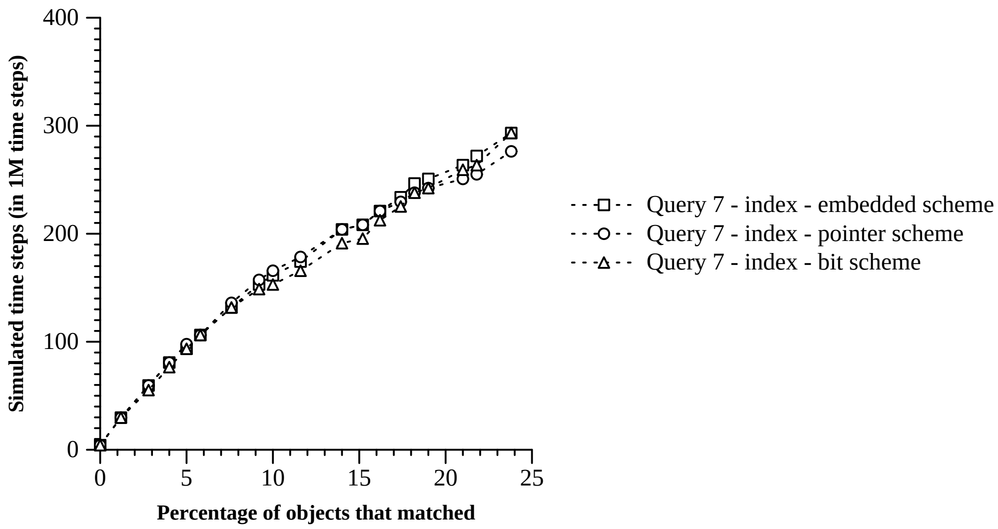


Figure 3.24: Comparison of results of Query 7 for the small DB3 with 8K segments.

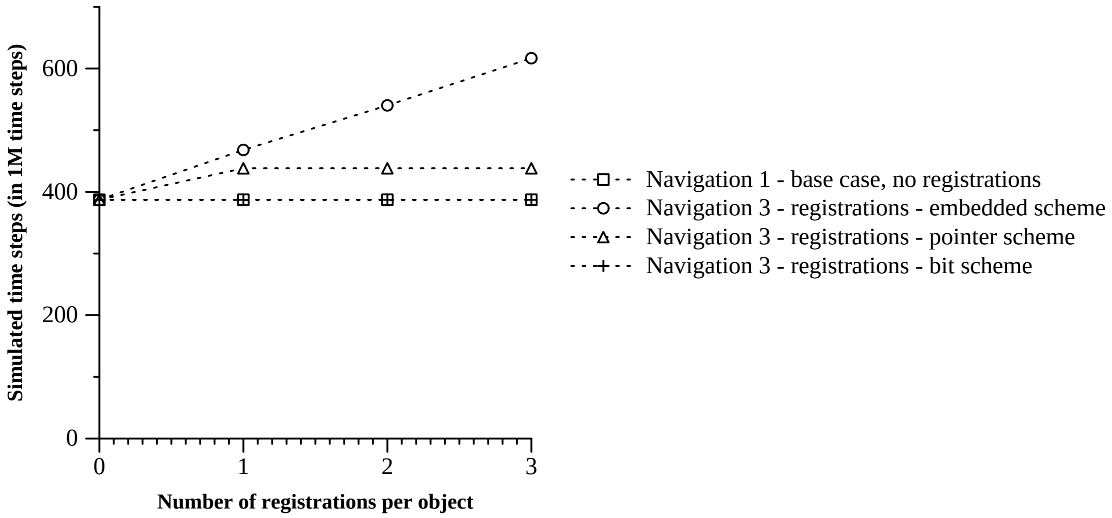


Figure 3.25: Comparison of results of Navigation 1 and Navigation 3 on the full traversal for small DB3 with 8K segments with a cold cache.

registration information, respectively) on the full traversal for the small DB3 with 8K segments with a cold cache. (Navigation 2 is only interesting with respect to segment size, so we omit the results for it in our comparison.) We see that the effect of registration information in Navigation 3 relative to Navigation 1 on the full traversal is the same as for Query 2 relative to Query 1. This is not surprising, since the full traversal accesses nearly the same data objects as the query.

As noted in Section 3.6.2, cache size and clustering are important for good navigation performance. Thus the main difference for each scheme is that a larger cache size is needed to store the data objects with registration overhead stored inside the registered object. As expected, the bit scheme is neutral since it adds no space overhead to the data objects, and the pointer scheme shows some degradation due to the extra reference per registered data object. The embedded scheme has the largest effect on cache size, and it is the only scheme where having more registration tuples per registration set affects the running of the traversal. This is because each additional registration tuple causes the registered data objects to become larger.

The results of the path1 traversal also show that the effect of registration information on running time is minimal for this type of query. Since so few objects are accessed, in many cases, there is no difference between Navigation 1 and Navigation 3. In some cases for the embedded and pointer schemes, Navigation 3 performs better than Navigation 1 due to the effect these schemes have on clustering combined with the fact that very few objects are being accessed. The change in clustering sometimes results in an object that was not in the same segment with the objects it references in Navigation 1, being pushed into the segment with the objects it references so that following these references does not cause a disk access in Navigation 3 that is present in Navigation 1.

3.7.3 Updates

For the previous benchmarks, the only differences in the schemes were due to the effect of registration information on clustering. In the update benchmarks, we expect many more differences due to extra computation and possible disk accesses to find the registration information during a write. Figure 3.26 shows the results of Update 1 and Update 2 on the full traversal for the small DB3 with 8K segments with a cold cache. Update 1 represents a system without indexes and measures the base cost of doing the update traversal. Update 2 measures the incremental cost in our scheme for checking the registration bit in object headers and is the same for all three schemes since there are no registrations. As expected, there is little difference between Update 1 and Update 2 as the cost of checking is totally dominated by the costs of

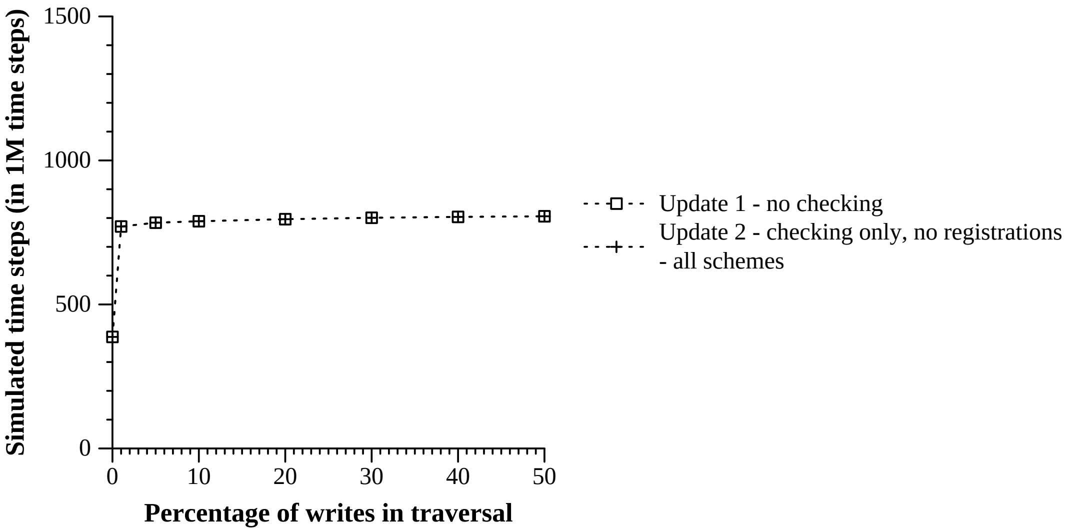


Figure 3.26: Comparison of results of Update 1 and Update 2 for the small DB3 with 8K segments with a cold cache.

---

bringing the objects into the cache in the first place.

For Update 3, Update 4, and Update 5, recall that all data objects (except document objects) are set elements are registered with one registration tuple each. For these updates, it is necessary to read an object's registration tuples when it is written. Update 3 measures the incremental effect of reading the registration tuples. Figure 3.27 shows a comparison of the three schemes and Update 1. For the embedded scheme, reading registration tuples is trivial, since they are part of the registered object. However, since the data objects are spread out among more segments, reading all of the data objects into the cache takes longer in Update 3 than in Update 2. The incremental increase in running time is about 21%. For the pointer and bit schemes, the registration object for a written registered object must be accessed as a regular object to bring it into the cache. This causes more overhead than in the embedded scheme, but is mitigated somewhat by the fact that the data objects are clustered more tightly in these schemes. In addition, the bit scheme requires the registration table to be read into the cache. However, since the data objects themselves cluster into fewer segments than in the pointer scheme and the registration table is likely to stay in the cache, the effect is about the same as in the pointer scheme (about a 35% increase).

Figure 3.28 shows the results of Update 3, Update 4, and Update 5 for the small DB3 with 8K segments with a cold cache for each of the schemes. The difference between Update 3 and Update 4 is due to updating the index (i.e., writing update records into the log and

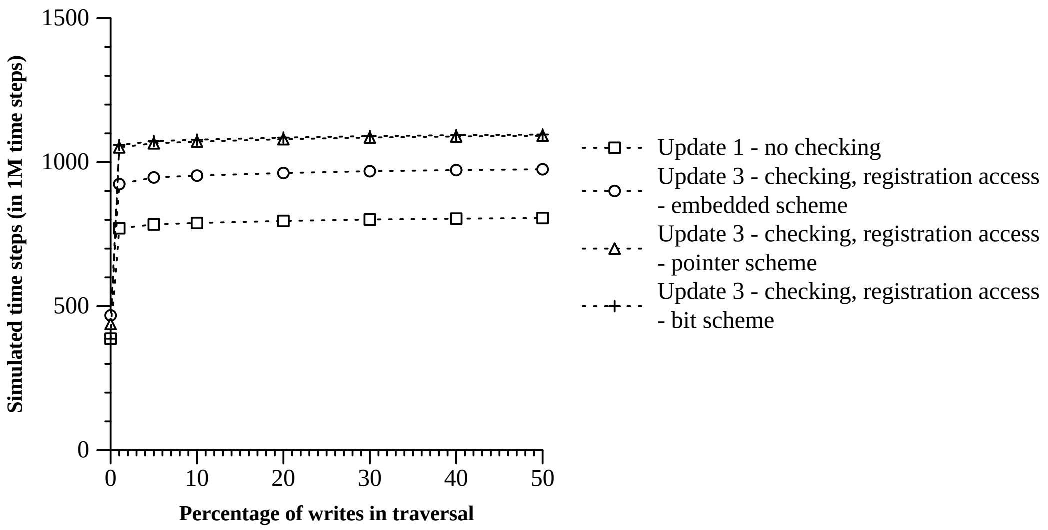
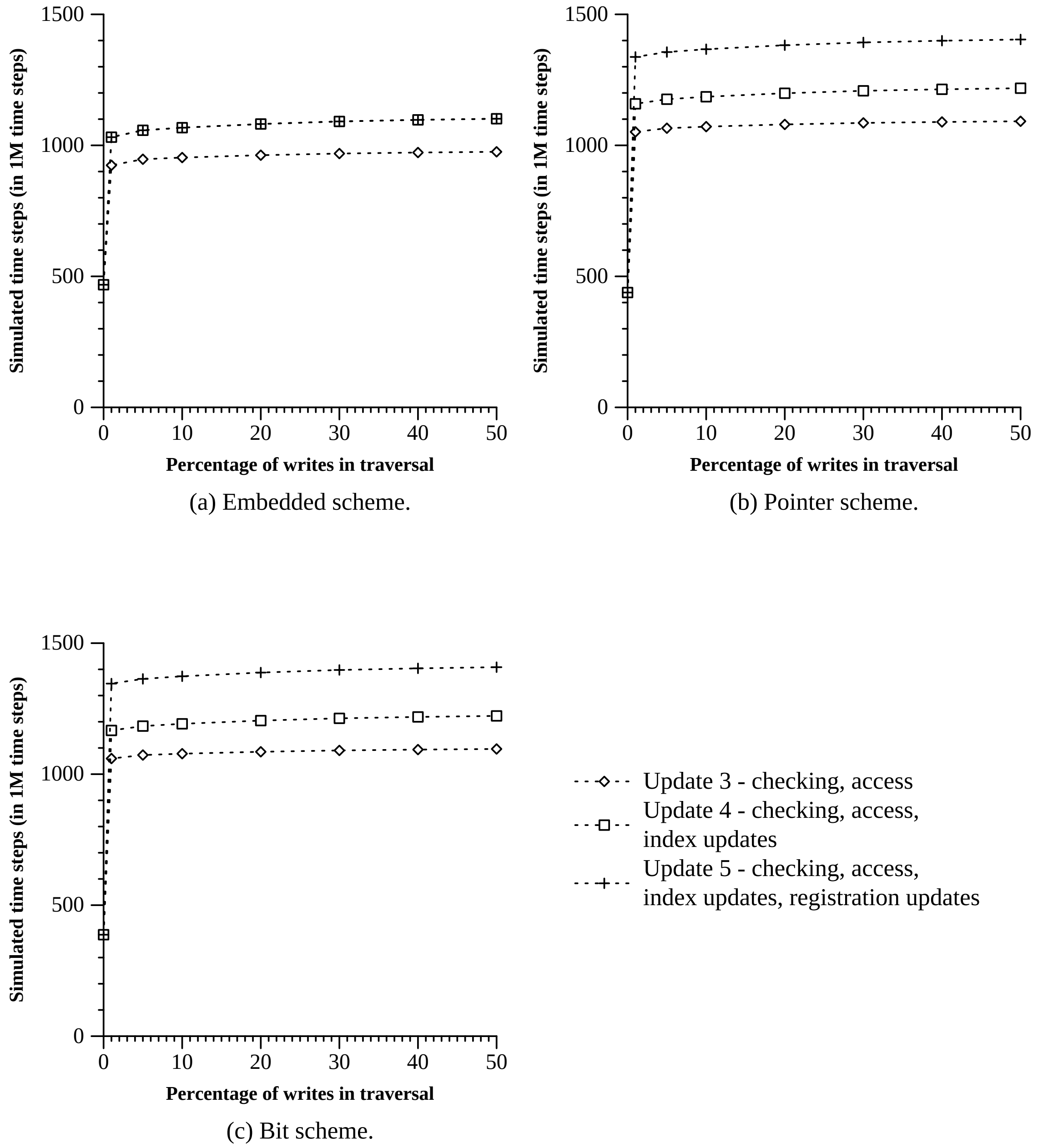


Figure 3.27: Comparison of results of Update 3 for the small DB3 with 8K segments with a cold cache.

---

writing back the segments that hold the modified index nodes), which has the same cost for any of the schemes. The differences between Update 4 and Update 5 measures the incremental effect of having to update registration tuples. Here the embedded scheme has a much smaller incremental cost with respect to the other schemes. Since registration tuples are already part of the registered object the log entry for the registered object already contains the new version of the registration tuples (since we are using new-value logging). As a result, there is no discernible difference between Update 4 and Update 5 for the embedded scheme. The results for the pointer and bit schemes show the incremental cost of writing the log records for the modified registration objects and then writing the segments containing these objects. For these schemes, there is about a 15% incremental increase in running time between Update 4 and Update 5.

Other than the advantage that the embedded scheme has on Update 5, the schemes perform comparably for the small DB3 configuration. This is because everything fits into the cache (the data objects, the registration sets, and the registration table, if any), so there is only one disk access per segment accessed. Figure 3.29 shows the results of the update benchmark on the full traversal for the medium DB9. This database does not fit into the cache, thus some accesses cause segments to be thrown out of the cache. We observe two interesting phenomena. First, we note that the incremental increase of doing index updates and registration updates over just checking the registration tuples as a percentage of total running time has diminished to a small amount (a few percent) when the database is this large. This is because the cost of doing index

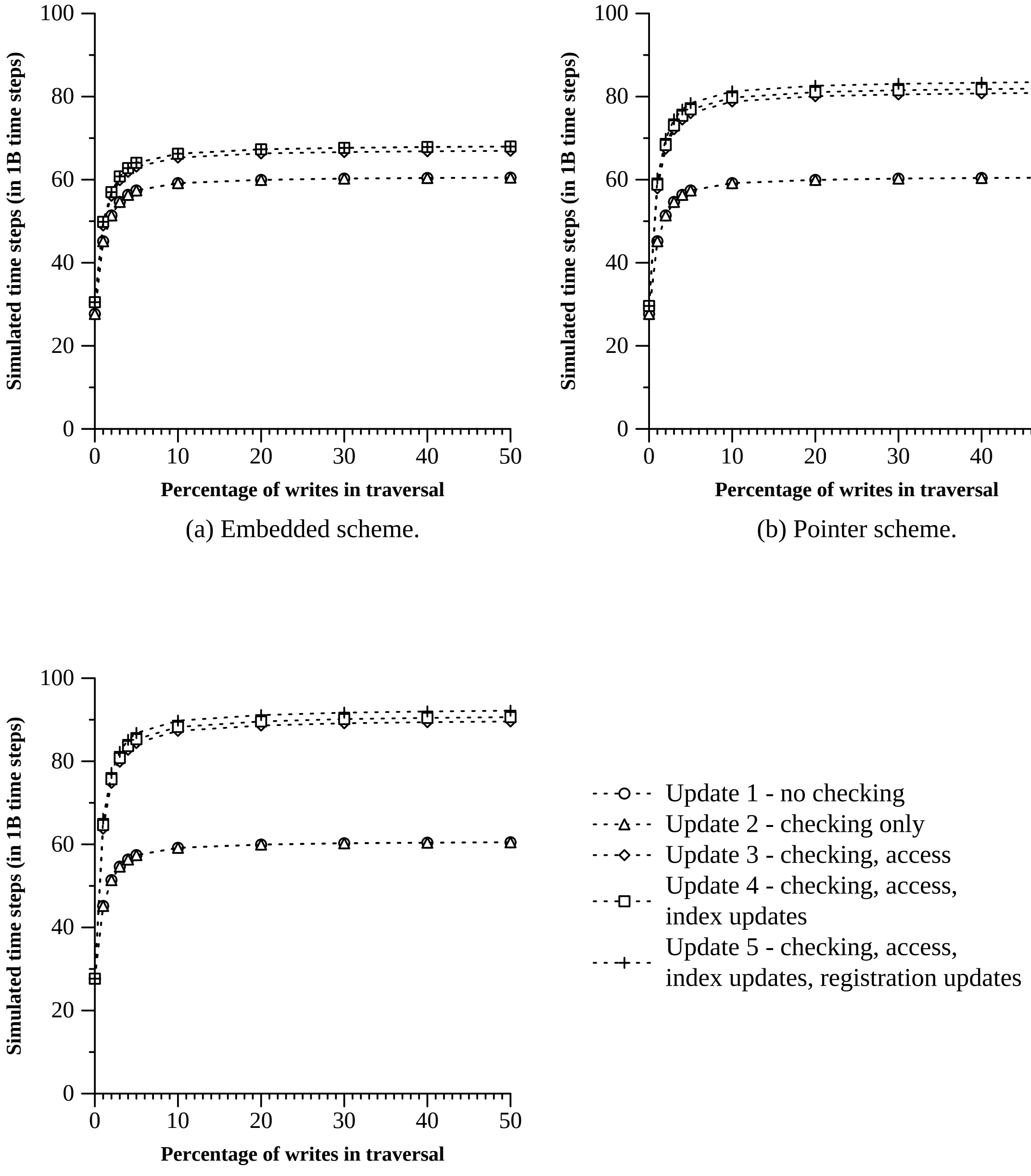


(a) Embedded scheme.

(b) Pointer scheme.

(c) Bit scheme.

Figure 3.28: Results of Updates 3–5 on full traversal for the small DB3 with 8K segments.



(a) Embedded scheme.

(b) Pointer scheme.

(c) Bit scheme.

Figure 3.29: Results of update benchmark on full traversal for the medium DB9 with 8K segments. Note that the y-axis is in billions of time steps.

updates and registration updates (writing update log records, writing index segments, and in the pointer and bit schemes, writing registration object log records, and registration segments) is dominated by the I/O costs of doing the traversal.

Second, we see greater differences between the three schemes on Update 3 (writes to registered objects). The embedded scheme performs the best, followed by the pointer scheme, then the bit scheme. The embedded scheme has an incremental increase in running time from Update 2 to Update 3 of only 12% percent while the pointer scheme has an increase of 35%, and the bit scheme has a very large increase of 50%. This is due to the difference in sizes that each of the schemes require for this database. The embedded scheme increase is due to having more data segments to access, but as we saw in the small DB3 case, this is the primary cost for updates in the embedded scheme. In the pointer and bit schemes registration objects take up more space than the equivalent registration tuples in the embedded scheme, so that a larger cache would be necessary to hold the same amount of data objects and registrations than in the embedded scheme. In addition, the bit scheme needs extra space for the registration table. Also, since there are many registration objects per registration segment, the registration segments tend to stay in the cache (as does the registration table segments) since there are many accesses to them, causing data segments to be thrown out earlier and more often than in the embedded scheme.

We note that the curves for these results are basically flat. This is due to two factors: reading in the entire database over the course of the traversal and the long length of the full traversal. Even when only 1% of the steps are writes, this translates to 506 writes to 497 objects in the small DBs and 4364 writes to 4270 objects in the medium DBs. Thus there are enough modified objects to have nearly one modified object per data segment read. The results for the path2 traversal are similar since the traversal accesses multiple objects per data segment. The path1 traversal accesses very few objects. Figure 3.30 shows the results of the path1 traversal on the small DB3 with 8K segments with a cold cache. Here we see a situation where each write has an incremental cost, but otherwise the results are about the same as for the full and path2 traversals. As expected, the embedded scheme performs the best, followed the pointer scheme, then the bit scheme.

Running the update benchmark for the small DB3 with 8K segments with a warm cache does not significantly alter the relative performance of our schemes. The reason for this is that we cannot avoid the cost of writing back data segments, registration segments, and the log. Thus disk I/O still dominates the update benchmark even when the cache is warm.

A type of update benchmark that we did not model is when a client modifies objects that

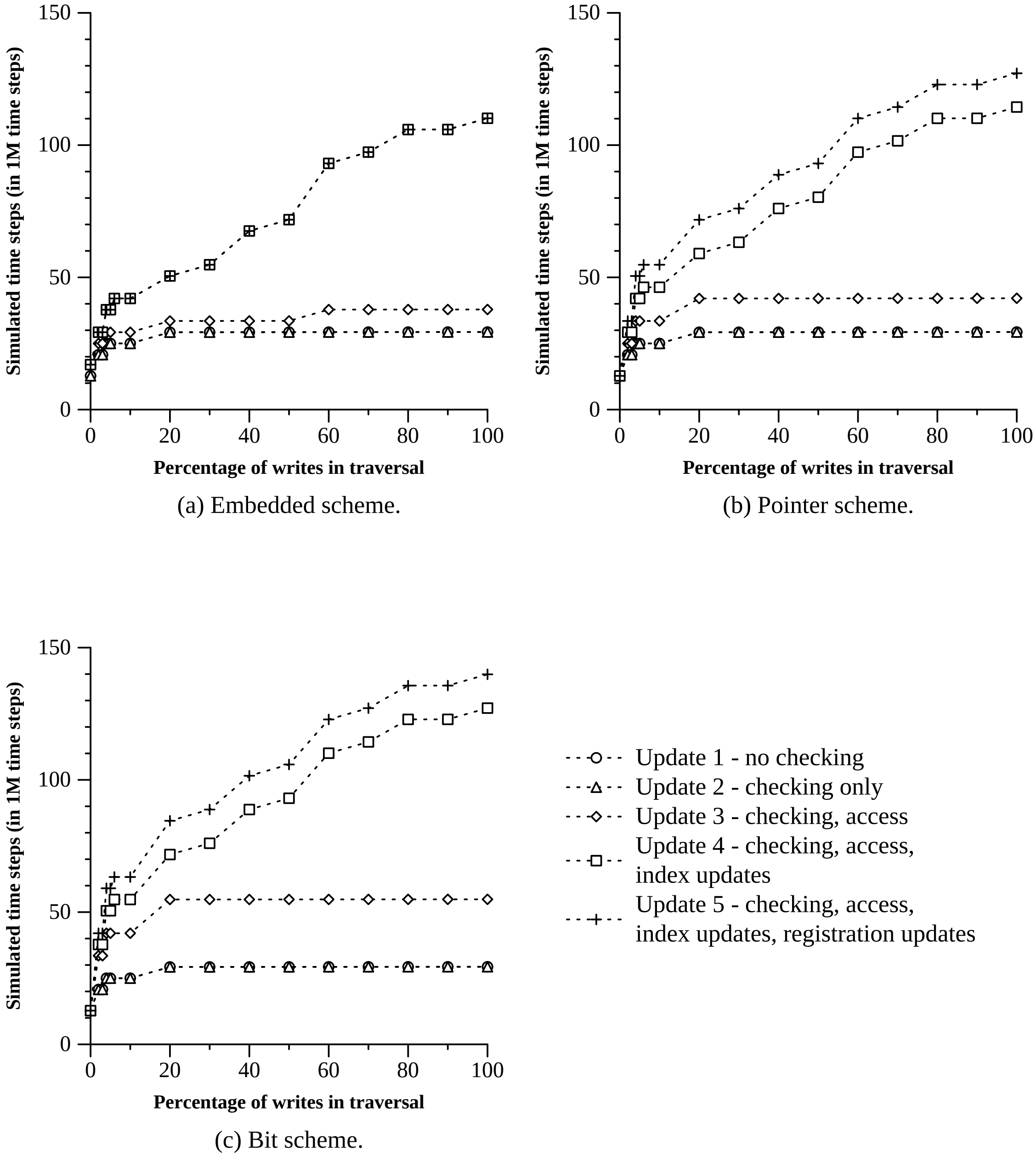


Figure 3.30: Results of update benchmark on path1 traversal for the small DB3 with 8K segments with a cold cache.

cause index updates, but the modified objects are not set elements. We expect that the results from such a benchmark would be similar to those above, since the dominant cost for these updates is the disk I/O. The cost of finding registration information is the same as in the simulated update benchmark. Computing the old key and recomputing the new key for the affected index entries will require accessing the set elements, but this can only increase the disk access overhead and will happen in all three schemes equally.

Another situation that we did not model was updates that add (or delete) registration tuples. In the system we are modeling, changes in object size might be handled by trying to reallocate the object into space left in its current segment and failing that, allocating the object into a different segment with sufficient space. We can discuss what happens in our schemes when new registration objects are created or when registration tuples are added in such a scheme. In the bit scheme, when a new registration object is created the system must allocate the new registration object and update the registration table. Updating the table might cause a node to split, but this is not likely. When a registration tuple is added to a registration object, the registration object increases in size and may have to be written to another segment.

In the pointer scheme, the initial registration causes the registered object to become larger and this could cause the registered object to be moved from its current segment, destroying any clustering that might be present. But once the reference has been added, the object will not become larger; thus we could recluster the objects after an update and they would stay clustered (assuming that the data object does not have operations that change its size). As in the bit scheme, when a new registration object is created, the system must allocate the new registration object, and when a registration tuple is added to a registration object, the registration object increases in size and may have to be written to another segment.

In the embedded scheme, registration tuples are added directly into the registered object. As pointed out before, this changes the size of the registered object and can cause the clustering to be destroyed. Again, we could recluster the objects after an update, but unlike the pointer scheme, this may happen every time a registration tuple is added rather than just the first time an object is registered. Thus there is a down-side to the embedded scheme during updates.

Index maintenance is costly in all systems. We are concerned with the additional expense associated with using and maintaining registration information in our scheme. In the embedded scheme, the main cost to updates is having to read and write larger data objects, and as we can see from our results, this cost is relatively modest. The pointer and bit scheme are substantially more expensive than the embedded scheme on updates, but there is little difference between them provided the cache is large enough in the case of the bit scheme to accommodate its additional registration table.

3.8 Space Analysis

We begin this section by summarizing the results of our simulation experiments. The query benchmarks show that indexes are an important optimization for queries in an object-oriented database, especially in the cases of primary indexes and complex index functions. If the overall workload is mostly queries, indexing will be a great benefit. Index maintenance adds some cost to updates, but unless updates are a very high percentage of overall workload, the benefits of indexes outweigh the cost of updates. Indexes may not be very useful when updates are common in any system, so perhaps indexes should not be created in these situations in the first place. Traditionally, databases have been used primarily to read information through queries with few updates. We expect this to continue in object-oriented databases with navigation added into the workloads as another way of reading information.

Recall that our hypothesis about our implementations schemes is that the different implementations would perform well in different situations. In particular, we expected that the embedded scheme would be best for updates but the worst for navigation and queries, and the bit scheme would be best for queries and navigation, but worst for updates, with the pointer scheme in the middle for all three benchmarks. This relative ordering is borne out by our experiments.

In the rest of this section we present a framework for analyzing the space requirements of indexing schemes, and we suggest other implementations for our scheme that are more space efficient than the implementations we simulated. In this analysis, we are only interested in the space overhead due to registration tuples and associated overhead for finding and maintaining them. We ignore the space overhead of the index since it is the same for all schemes.

The space overhead of the registration information in our indexing scheme (RS) is characterized by the following formula:

$$RS = R + O$$

where R is the total space taken up by registration tuples and O is the total space taken up by overhead needed for finding the registration tuples.

R is the same for all implementations schemes:

$$R = (N * L) * T * d$$

where

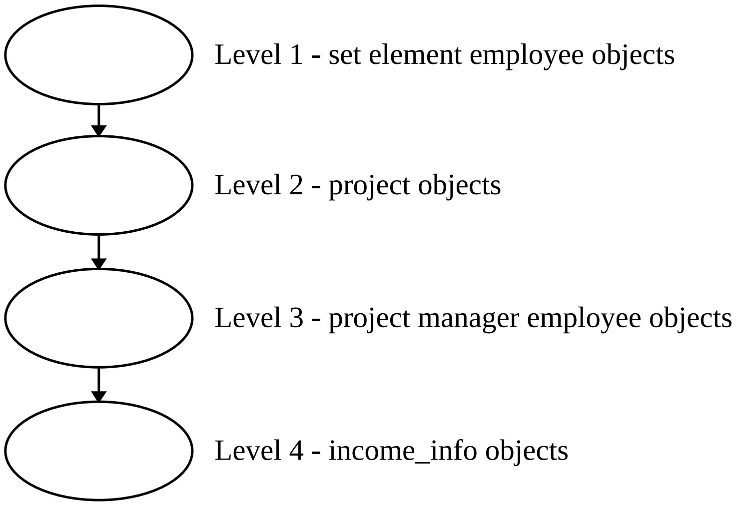


Figure 3.31: Level graph for *project_manager_income*.

N = number of set elements.

L = number of objects registered per set element.

T = size of a registration tuple.

d = average number of mutators that affect the observers used in key computation.

L can also be thought of as the number of *levels* in the object graph representing each access done by the index function. Figure 3.31 shows that *project_manager_income* has four levels, since it accesses a set element `employee` object, a `project object`, a project manager `employee` object, and an `income_info` object. For our analysis, we assume that all objects have the same structure, so that L is a constant for a particular index.

The $(N*L)$ factor comes from the fact that each object that is registered must have information about the set element that it is registered for. For example, for our *project_manager_income* index, if three `employee` objects, α, β, and δ, in the indexed set refer to the same `project` object π, then there will be three registrations for π, namely $< \pi$, *set_manager*, $\alpha, I >$, $< \pi$, *set_manager*, $\beta, I >$, and $< \pi$, *set_manager*, $\delta, I >$. In addition, the project manager `employee` object and the `income_info` object will also each have three registrations.

O depends on the implementation scheme. For the embedded scheme, $O_{emb} = 0$, since there is no overhead. In the pointer and bit schemes, O depends on the number of accessed objects, since each registered object has a reference or registration table entry, and a registration object. To describe the overhead of the pointer and bit schemes, we introduce the following notation:

M = number of non-set elements that are accessed during key computations.

H = size of object header

P = size of a reference

B = size of registration table entry

Thus, $O_{ptr} = (N + M) * (P + H)$, i.e., each registered object incurs overhead for a reference and a registration object header), and $O_{bit} = (N + M) * (B + H)$, i.e., each registered object incurs overhead for a registration table entry and a registration object header.

We would like to minimize RS, but not at the expense of making updates very expensive. (Thus eliminating registration information altogether and recomputing all indexes whenever there is a mutation is not acceptable.) To reduce the space overhead of registration information RS, we can try to reduce both terms, R and O. For R, we cannot change the $(N * L)$ factor except by registering fewer objects. We already avoid registering immutable objects, or objects that are accessed by observers that do not depend on any mutators. For example, for the *project_manager_income* index, if `project` objects were immutable or did not have a *set_manager* mutator, they would not be registered. We discuss another way of avoiding registrations in Chapter 5.

We can change the d and T factor of R. We assumed in our simulations that a registration tuple was a $< m, x, I >$ triple. Storing m in each registration tuple means d can be greater than one. For example, in the *project_manager_income* index, both project manager `employee` objects and their `income_info` objects have two registrations per set element since they each have two mutators that can affect the index function. We might ask if it worth storing m in a registration tuple. Suppose instead that we kept a table for each index I that mapped classes to the mutators of the class that could affect the index. The checking versions of mutators could look at this information to see if they need to cause an update. (I.e., when a checking version of a mutator runs, it first checks if it is in the mapping for its class for the indexes named in the registration tuples of its object.) In our example, the *project_manager_income* index would have a mapping < `employee_impl`, { *set_project*, *set_monthly_rate*, *set_bonus* } > indicating there are three mutators of the `employee_impl` class that can affect the result of the *project_manager_income* function. This scheme would allow us to always store one $< x, I >$ pair for each registered object and drop the d factor from our formula.

Note that we lose some precision with this scheme and may cause unnecessary updates. For example, suppose `employee` object α is registered for the *project_manager_income* because it is in the indexed set, but it is not a project manager. When we store m, the only time a mutation of α causes an update is when its *set_project* mutator is called. Using the new scheme, suppose that we call the *set_bonus* mutator on α. Since we only change the method dispatch vector to the checking version of mutators that can affect an index, if the *project_manager_income* index is the only index being maintained on the set, α will be using the regular version of *set_bonus* and will not check, so this will not cause updates. However, if the set also has an index for the *yearly_income* observer, α will be registered for the *yearly_income* index, since *yearly_income* depends on *set_bonus*, and its method dispatch will point to the checking version of *set_bonus* as well. In this case, when *set_bonus* on α is called, the checking version looks up information for both the *project_manager_income* and *yearly_income* indexes and determines that entries in both indexes need to be recomputed. However, the recomputation for the *project_manager_income* index is unnecessary, since α is not a project manager. Depending on how many indexes have overlapping mutator sets, the space savings in registration information from using this scheme may or may not be worth the extra recomputations.

Let us assume that we will use the above scheme. In addition to removing the d factor from R, moving mutator information to the index also allows us to have smaller registration tuples that are just $< x, I >$ pairs, thus reducing the T factor of R. Originally, $T = 12$ in our simulations. The reference for x still takes 8 bytes, but we expect in a 64-bit address system that not all bits of a reference will be significant (e.g., 48 bits should be sufficient), so we can store the index id using some of the bits of the reference to x, creating a registration tuple that is 8 bytes.

We can further reduce T for the common case when a registration is for the set element itself. That is, x in the registration tuple is the registered object. In this case, we can store just $< I >$ and know that the set element is the implicit x for this registration tuple. We can store $< I >$ in 4 bytes (assuming that data needs to be stored on a word boundary). A data object might have both 4-byte and 8-byte registration tuples, so we use one bit in the registration tuple to indicate if a registration tuple is the short version (with size $T_{short} = 4$) or the long version (with size $T_{long} = 8$).

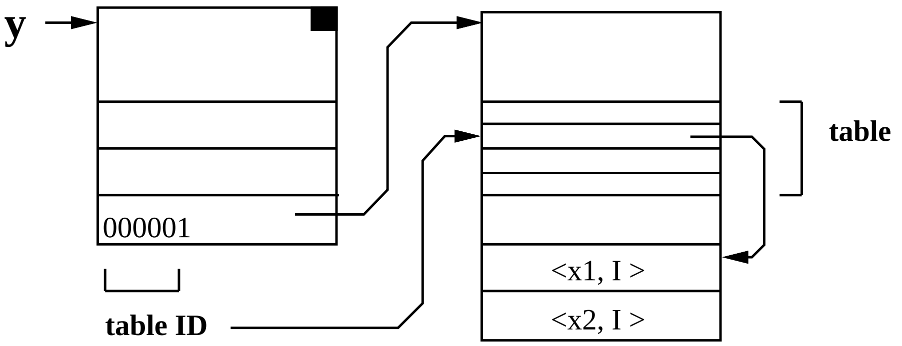
Using all of these techniques, we can now express R with the following formula:

$$R = N * T_{short} + N * (L - 1) * T_{long}$$

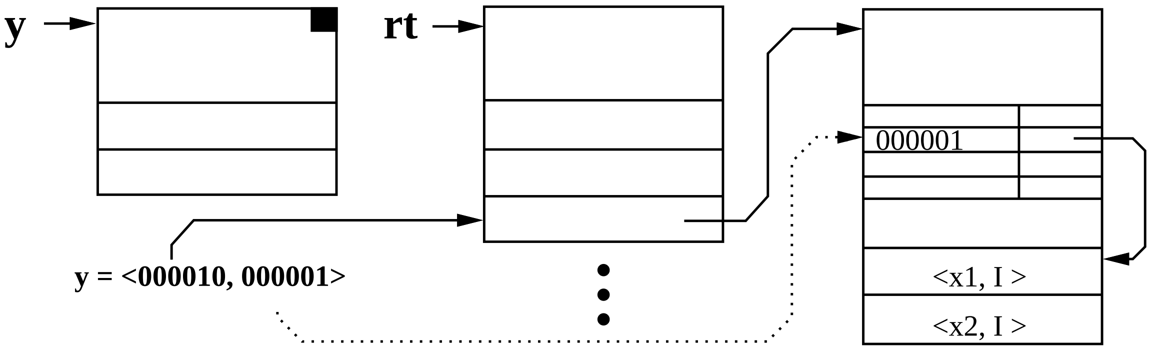
That is, there one short registration tuple for the top-level registration in each of the N set elements and one long registration tuple for every other registration.

Now we consider the O term of RS. One component of O in both the pointer and bit schemes is the object header for each registration object. To reduce this cost, we can combine the registration tuples from many objects into one registration object so that the amortized cost of the registration object header is small enough for us to ignore it. However, we must add some overhead back into the equation in order to find the registration tuples for a specific registered object. For the pointer scheme, one possible implementation would be to store a small "table" at the top of each registration object that keeps track of where the registration tuples for a particular data object starts. The table would be indexed directly and each entry would be 2 bytes. An entry would indicate an offset into the registration object. (We assume that the offsets in the registration object tables are for word boundaries, so that 2 bytes can encode enough locations in the registration object.) Since we expect that changes that affect the number of registrations will be rare, we assume that all of the space between the offset mapped in table entry i and the offset in table entry $i+1$ contain registration tuples for the registered object that uses table entry i, and whenever registration tuples are added to or removed from the registration object, we repack and rewrite the entire registration object. When accessing a registration object, a "table ID" needs to be supplied to find the correct table entry, so we need to store or calculate a table ID for each registered object. We expect that we can store the table ID directly within the reference to the registration object. Figure 3.32(a) shows this scheme pictorially. The registered object is y, and its reference to a registration object contains bits that are interpreted as y's table ID into the registration object's table. The overhead of the pointer scheme is now: $O_{ptr} = (N + M) * P'$, where $P' = 10$, 8 bytes for the reference and 2 bytes for the registration object table entry.

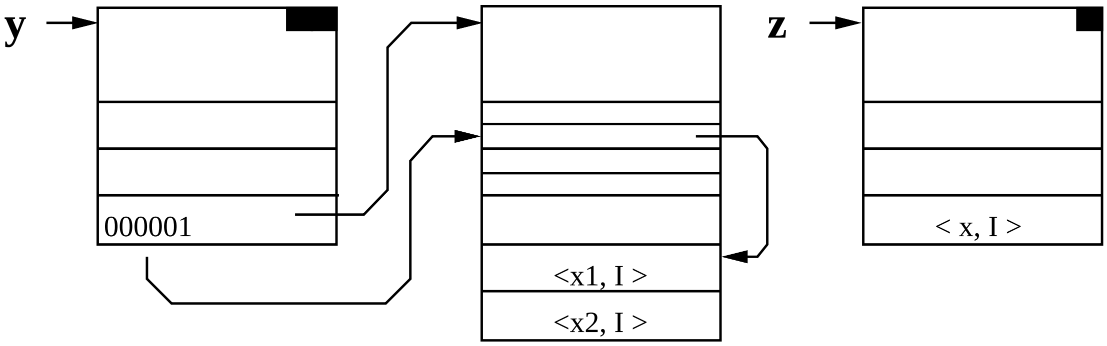
We can use the same kind of special reference in the registration table entries of the bit scheme to point to registration objects. However, we might like to reduce the size of the registration table itself. One way is store the registration tuples for data objects in one segment in one registration object and use segment IDs to index directly into the registration table. That is, object references are really <*segment_id*, *address*> pairs, and we use the segment ID to map directly to an entry in the registration table that contains the reference to the registration object for that segment. As in the pointer scheme, we need a way to find the registration tuples for the specific registered object when accessing a registration object. However, there is no place to store a table ID, so we will keep a table at the top of a registration object that maps the address portion of the registered object's reference to an offset into the registration object. As in the pointer scheme, we assume that registration objects rarely change size so when they do, we repack and rewrite them. We keep the table in sorted order by address and also keep the registration tuples packed in order according to the table. To find a registered object's registration tuples, the registration object table is searched using binary search to find the table entry for the registered object's address and then the offset is followed to find the



(a) New pointer scheme



(b) New bit scheme



(c) Hybrid scheme

Figure 3.32: Three new implementation schemes.

registration tuples.

Figure 3.32(b) shows this scheme. Reference y is shown as a <*seg_id*, *addr*> pair. *Seg_id* is used to index the registration table rt to find a registration object. *Addr* is then used to find the offset of y's registration tuples in this registration object. The space cost of the registration table for this scheme is amortized, so we will ignore it in our overhead calculations. Each registered object has an entry in a registration object table for an address and an offset. The address is 4 bytes and the offset is 2 bytes, so the new overhead for each registered object, B', is 6 bytes, and $O_{bit} = (N + M) * B'$

We observe in the pointer scheme that paying the overhead factor for one registration tuple is costly; we might as well store the registration tuple in the space taken up by the reference. Thus, we devise a *hybrid scheme* where if a data object only has one registration, the registration tuple is stored directly in the object, and if a data object has more than one registration, it stores a reference to a registration object. Since we need to distinguish whether a registered object is storing a registration tuple or a reference, we use another bit in the object's header to indicate this. Figure 3.32(c) shows both kinds of registered objects. Both objects y and z are registered. Object y has two registrations so both bits in its header are set and it has a special reference as in the pointer scheme. Object z has only one long registration, so it has only one bit set in its header and stores the registration tuple directly. The overhead for this scheme is $O_{hyb} = S * P'$, where S is the number of objects that have more than one registration.

In these new implementation schemes, we assume that if a registration object becomes larger than a segment that it is marked as special and there is code to handle the overflow (into another registration object). Overflow can happen if too many objects' registrations are put into one registration object, or if one registered object has more registrations than will fit into a segment. The latter case probably will be rare and can happen equally in either scheme. Overflow from putting too many objects' registrations into a registration object is less likely to happen in the pointer and hybrid schemes than in the bit scheme, since we have explicit control over how many objects' registration tuples go into one registration object in these schemes, whereas in the bit scheme, the clustering in a segment determines where registrations are stored.

Now we can compare the space used by our new implementation schemes. We note that $S \leq (N + M)$, so that the hybrid scheme is never worse than the pointer scheme; therefore, we will not consider the pointer scheme any longer and will consider only the embedded, bit, and hybrid schemes. The formulas for the registration space overhead of each scheme is:

$$RS_{emb} = N * T_{short} + N * (L - 1) * T_{long}$$

$$RS_{bit} = N * T_{short} + N * (L-1) * T_{long} + (M+N) * B'$$

$$RS_{hyb} = N * T_{short} + N * (L-1) * T_{long} + S * P'$$

If the index function uses only data in the set element, the hybrid scheme clearly uses less space than the bit scheme and is the same as the embedded scheme, since each set element will have only one registration tuple, so $S = 0$. For more complex functions, the space costs for the hybrid and bit schemes depend on the sharing structure of the objects accessed during key computations.

By proposing a hypothetical set, plugging in numbers for sizes, and modeling sharing in different amounts at different points in the key computation, we can graph the registration overhead space functions for each scheme to compare them. Our graphs will be for a set with 1024 elements (i.e., $N = 1024$). To make the modeling tractable, we assume that the index function accesses five objects in a linear path with the key being an integer in the last accessed object. We assume that no object appears in more than one level. That is, objects can be shared at the same level (e.g., many `employee` objects in the indexed set can share the same `project` object), but the objects are not shared between levels. (e.g., a set element x never appears as an object at a lower level when a key is computed for another set element z; note that this can happen in our *project_manager_income* function if the project manager is also an element of the indexed set). We introduce the following notation:

L_1, L_2, $\ldots =$ number of distinct objects at each level of the index function; for example., for *project_manager_income*, L_1 would be the number of set elements, L_2 would be the number of `project` objects accessed, etc.

For the index function under consideration, there are five levels and $M = \sum L1, \ldots, L5$. In general, $M \leq \sum L1, \ldots, Ln$, since objects that are used in more than one level would be counted at each level.

The x-axis in our graphs represents the *sharing factor* at the specified level. We use the notation "L$i = n$" to mean that n objects at level $i-1$ refer to the same object at level i. For example, if L2 = 1, each set element refers to a distinct object at level 2, while if L2 = 2, two set elements refer to the same object at level 2. By definition, L1 = 1, since there is no level zero.

Figure 3.33 shows the amount of space overhead required by each implementation for various sharing factors at the second level (L2) assuming that there is no sharing at lower levels (i.e., L3 = L4 = L5 = 1). Thus at each lower level, there are the same number of objects as at the

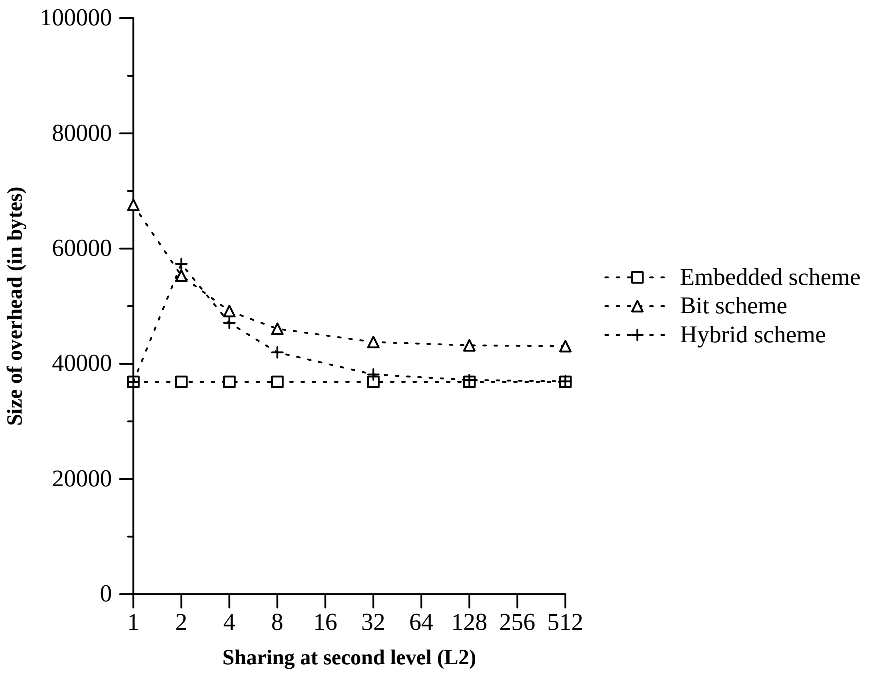


Figure 3.33: Comparison of space overhead on a 5-level path function with sharing at the second level for a set with 1024 elements (L1 = L3 = L4 = L5 = 1).

first level (i.e., $L_2 = L_3 = L_4 = L_5$). The embedded scheme is not affected by sharing. For the hybrid scheme, when L2 = 1, there is no sharing at all, so S = 0; when L2 $\neq$ 1, all of the objects at the second level are shared, so all objects at lower levels will also be accessed during multiple key computations and have more than one registration, thus S = $\sum L_2, \ldots, L_5$. The results show that the hybrid scheme is like the embedded scheme when there is no sharing, since each object has just one registration tuple. When there is a little sharing, the hybrid scheme costs increase until L2 = 2, since there are increasing numbers of registered objects with more than one registration. Since we are modeling even distribution of sharing, L2 = 2 is the worst case for the hybrid scheme, since it is the minimum amount of sharing to cause every non-set element to have two registration tuples, which results in the maximum number of registration objects. As sharing increases beyond a factor of 2, the hybrid scheme asymptotically approaches the embedded scheme, because there are fewer and fewer registration objects and the overhead O_{ptr} is being amortized over many registration tuples. The bit scheme benefits from sharing since there are fewer registration objects, but it takes up more space than the hybrid scheme in this scenario, except when L2 = 2, though not significantly more except when there is no sharing at all.

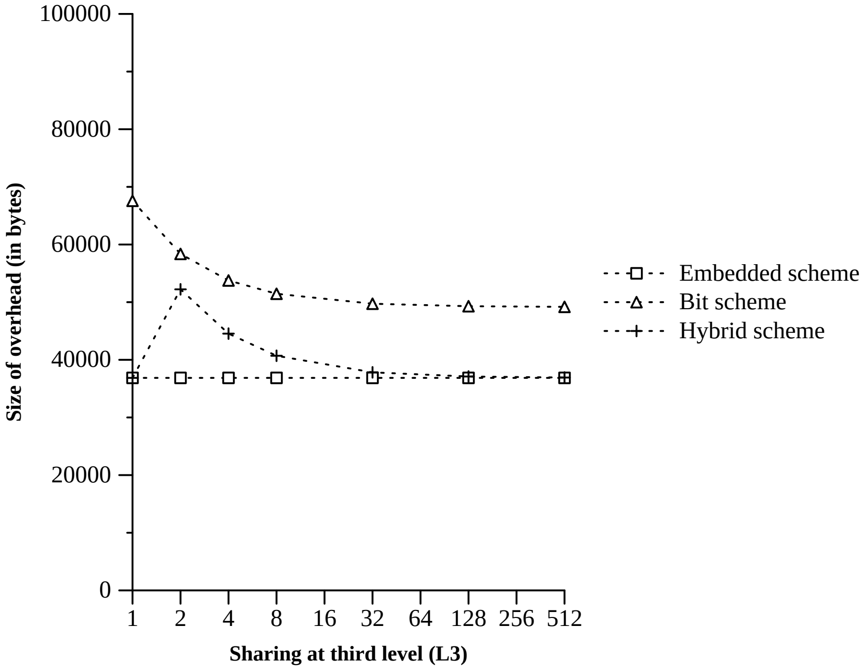


Figure 3.34: Comparison of space overhead on a 5-level path function with sharing at the third level for a set with 1024 elements (L1 = L2 = L4 = L5 = 1).

Figure 3.34 shows the amount of space overhead required by each of the new implementations when there is no sharing at the second level (i.e., L2 = 1) for various sharing factors at the third level (L3), assuming that there is no sharing at lower levels (i.e., L4 = L5 = 1). Thus at the second level, there are N objects (i.e., $L_2 = N$), and at the fourth and lower levels there are the same number of objects as at the third level (i.e., $L_3 = L_4 = L_5$). For the hybrid scheme, since L2 = 1, the objects at the second level are not shared, and when L3 = 1 there is no sharing at all and S = 0; when L3 $\neq$ 1, all of the objects starting at the third level are shared, thus S = $\sum L_3, \ldots, L_5$. The results here show that the hybrid scheme takes advantage of the lack of sharing at the second level by storing the registration tuple directly in the registered objects at this level as in the embedded scheme. The bit scheme cannot do this, since it must store a registered object's registration tuples in a registration object, even if there is only one. Thus, the bit scheme incurs more overhead than in the previous graph, since there are more registered objects, while the hybrid scheme incurs less overhead, and at this level of sharing, the hybrid scheme always incurs less overhead than the bit scheme. If we do not introduce sharing until even lower levels, the gap between the bit scheme and the hybrid scheme becomes even larger, since there are more objects with only one registration when sharing is introduced at lower levels.

We note that in both graphs, the hybrid scheme has a significant jump in space overhead whenever we move from no sharing (a sharing factor of 1) to a sharing factor 2 and then as sharing increases the overhead approaches that of the embedded scheme again. As we pointed out above, a sharing factor of two is the worst case for the hybrid scheme. If we feel that this case is a likely scenario, we might flatten the curve by storing two registration tuples directly in a registered object and only switching to a registration object if there are three or more registrations for a data object. We would expect curves for this scheme to be similar to the hybrid scheme curves shown, but with peaks at a sharing factor of three that are lower.

We can extrapolate the effect of these schemes on the time performance of our benchmarks from our simulations. The bit scheme would continue to show no effect on the query and navigation benchmarks. Since a short registration tuple is 4 bytes, and a long registration tuple or a reference is 8 bytes, we would expect the hybrid scheme to perform better than the original pointer scheme on the queries and navigation. The performance of the embedded scheme would be similar.

Judging the effect of the new implementations on updates is more difficult. We note that we have reduced the registration space overhead for our scheme by trading off time to do updates for this reduction in space. The mutator check now has to access the index class-to-mutator map, and every registration tuple must have a bit checked to see if it long or short. In the hybrid scheme, we also have to check a second bit in the header to determine if a registered object has an embedded registration tuple or a reference. In the reference case, we have to mask the table ID bits before accessing the registration object and then access the registration object table entry before finding the registration tuples. In the bit scheme, there is still an access to a registration table (albeit a smaller one) plus searching for a table entry for the registered object's address before accessing the registration tuples.

On the other hand, in the new implementations there is less data to read in (and possibly write out). The savings in disk I/O might be substantial and mitigate the extra computation to find and process registration information. The bit scheme nearly always requires more space than the hybrid scheme, but the cost of finding and processing registration information is well-bounded, and if sharing is high in the level graph, the additional space in not too large. The hybrid scheme usually requires less space than the bit scheme, but the actual cost is sensitive to the sharing structure of the accessed objects. It costs the most space when sharing is high in the level graph but the sharing factor is low and evenly distributed. However, we might expect that the common case is that most objects have one registration, thus update performance of the hybrid scheme might be close to the embedded scheme.

We have shown that the embedded scheme has the minimal amount of space overhead. However, whenever an index is added to the system in the embedded scheme, data objects become larger. As we saw in our results, storing registration information inside an object has a negative affect on query and navigation performance. In general, we may be willing to accept a small performance degradation due to larger data objects for the benefits of indexing. However, there is no bound on the number of registrations that an object may have, so in the embedded scheme, queries and navigation become slower and slower as more indexes are added. Thus, we conclude that an embedded scheme is not a suitable implementation for our indexing scheme.

The bit scheme has superior performance on queries and navigation, but has greater overall space requirements and poorer update performance. The hybrid scheme is a good compromise between the embedded and bit schemes. It is not quite as good as the bit scheme on queries and navigation, but may often give update performance like the embedded scheme, and it takes up less space than the bit scheme in most cases. If an application can accept slightly more cost to queries and navigation, switching to a registration object after two registrations rather than one registration will make the cost of an index in the hybrid scheme more predictable. Thus we conclude that either implementation scheme is suitable for our indexes. The hybrid scheme is a suitable implementation scheme if space is tight. If maximum query and navigation performance is needed and space is not tight, the bit scheme may be worth its cost in poorer update performance.

As a final note, we stated in Chapter 2 that the set element x in our registration tuple $< x, I >$ was used to avoid having to recompute the entire index rather than just the index entry for x. Figure 3.35 shows an example of where it might be more efficient to only record the registration for I rather than a registration tuple for each individual set element. Suppose object ϕ is f-reachable using index function $I.f$ (as indicated by the dashed arrows from the set elements to ϕ) and accessed using an observer that depends on mutator m. This will add a registration tuple $< x, I >$ for every set element for ϕ. When m is run on ϕ every index entry in I will have to be recomputed, thus the fact that we keep information for every set element has not saved us any work. In this case, it is clear that registering ϕ only for I would be more efficient. Of course, it is fairly rare for one object to be f-reachable from all set elements. However, one object might be f-reachable from many set elements, and it might still be reasonable to only record I in the object's registration tuple. To determine when this might be useful, we would need to trade off the space for the multiple registrations (and the time to process the registration information) with the time spent recomputing the index entries that were not affected by the mutation.

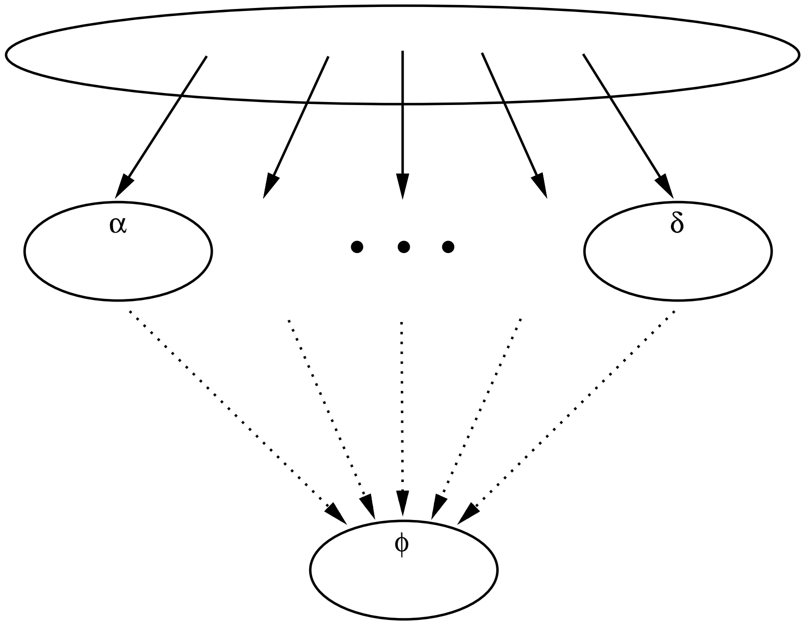


Figure 3.35: Example of a heavily shared f-reachable object. (Greek letters are OIDs.)

Chapter 4

Related Work

Indexing is an integral part of how a database functions. As databases have been extended beyond the traditional relational model, new systems have tried to extend indexing to accommodate the new extensions. There are two major lines of research into extending databases: traditional relational databases extended to incorporate objects and object references, and object systems extended to support database-style queries on collections.

Extended relational databases are relational database systems that have added constructs to handle more complex data types than can be modeled with records of built-in base types. Some representative systems are POSTGRES[53, 55, 56, 57, 58, 59], Starburst[28, 29, 45, 54, 61], Genesis[5, 6, 7], and Exodus[15, 16]. In general, these systems allow users to define new "base" types with richer sets of operations than the built-in types. For example, a user could define a `box` type with operations that compute and compare the areas of boxes. However, the goal of these systems is to retain the relational model and theory. This leads to user-defined base types that are immutable and the record continuing to be the only mutable entity. (That is, one can mutate a record with a box field to have a different box value, but one cannot mutate the box itself.) Thus, the question of mutating an object below the level of a set element does not arise. Sometimes these systems allow more general types of record fields, but do not allow indexing on these fields. For example, POSTGRES allows mutable fields in the form of collections of records as a field value, but does not allow indexes on these fields. So even with a more complex data model, indexing in these extended systems is largely the same as in traditional relational databases.

The work done in object-oriented databases is more closely related to our work. Such systems start with a general object-oriented programming language and add support for fast associative access to collections of objects. Some representative systems are GemStone[12, 47, 48], O_2[23, 24], Orion[4, 36, 37], and ObjectStore[40, 52]. Work specifically dealing with indexing falls into two categories:

1. Indexing schemes based on the structure of an object (that is, the values of instance variables). We will call these *path-based* indexing schemes.

2. Indexing schemes based on the result of an object's operations. Like our scheme, these are *function-based* indexing schemes.

In this chapter, we compare our indexing scheme with these other schemes. First, we survey several path-based indexing schemes. Then we look at the work being done in function-based indexing. To make our discussion more concrete, we will use the three element set of `employee` objects implemented by the `employee_impl` class shown in Figure 4.1 as an example. This figure shows only the objects that are f-reachable from the `employee` elements using *project_manager_income*. Each object has a unique identifier that is shown as a Greek letter. For convenience, we have set the project manager to be one of the set elements (which would indicate that the project manager is his or her own project manager).

4.1 Path-based Indexing

Most earlier work on index maintenance in object-oriented systems has concerned an approach in which a *path expression* is used as the basis for indexing. A path expression consists of a variable V followed by one or more instance variable names using dot notation, e.g., *V.income_.bonus_* for our `employee` objects. One can think of a path expression as a limited index function in which the only kinds of calls that can be made are *get_* methods that return the objects referenced by the instance variables of the object implementations. Path-based indexing appeared first in GemStone[47] with subsequent implementations proposed by Bertino and Kim[11] and Valduriez[60]. Note that path-based indexing guarantees that every time a path expression is evaluated, it accesses the same objects.

There are several disadvantages in using path-based indexing. Path-based indexing violates abstraction and encapsulation: Users must know an object's representation to state a query. The GemStone designers rationalized this design by arguing that user-defined types usually include methods for accessing the type's instance variables anyway[47], but this is not true for many types. Path-based indexing also has limited expressive power. The "function" of a path expression consists only of a sequence of instance variable lookups. This eliminates computed values, like our *project_manager_income* example. Furthermore, because the instance variables are exposed, multiple implementations are not possible; all objects of the type must have the same instance variables. We believe that in systems of the future, these limitations will be not

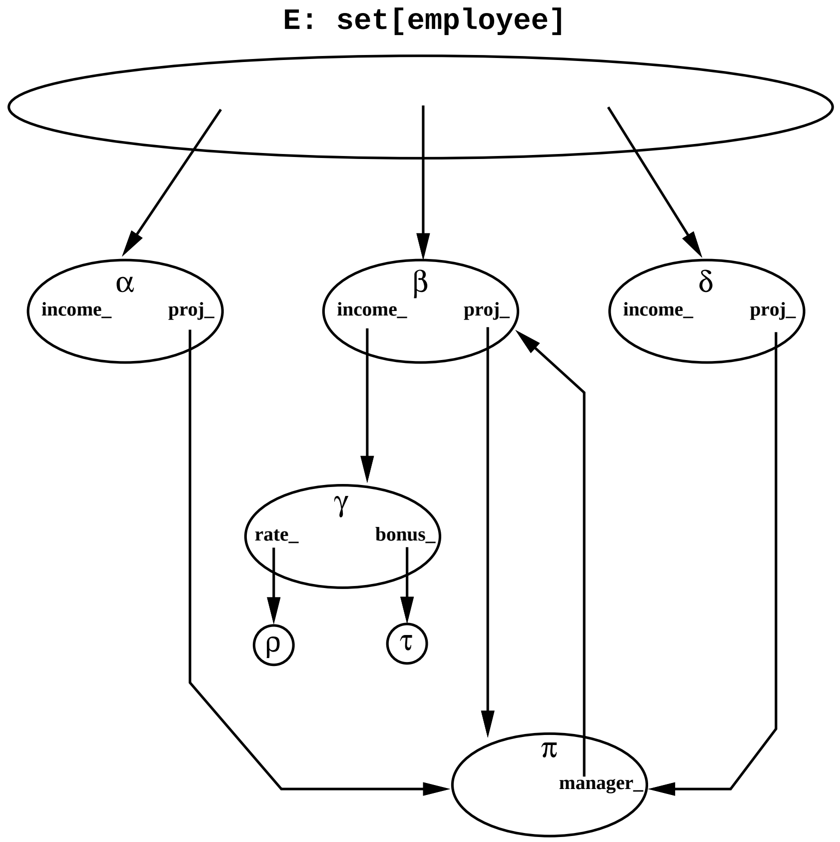


Figure 4.1: A set of three `employee` objects. Only the f-reachable objects are shown, where f = *project_manager_income*. (Greek letters are OIDs).

be acceptable. For example, the proposed ODMG standard defines queries based on function results[18].

Nevertheless, it is interesting to study the implementations of path-based indexes. We are interested in whether the restrictions imposed by path-based indexing make efficient implementations possible, and if so, whether these schemes can be adapted for our scheme. We will compare both time and space requirements of these schemes with ours. However, since our evaluation shows that our scheme has high space overhead, but not too great an impact on performance, we are primarily interested in comparing space requirements.

Various strategies have been proposed for index maintenance for path-based indexes; a survey can be found in [9]. Of these the *multi-index* is the most commonly used scheme; it was originally proposed for GemStone[47] and is also used in the O_2[24, 23] and ObjectStore[40, 52] databases. We will cover the GemStone scheme in some detail, then briefly cover the *nested index* and *path index* of Bertino and Kim[11] and the *join index* of Valduriez[60], emphasizing their differences from the GemStone scheme and each other.

4.1.1 Multi-indexes (GemStone)

In the GemStone scheme[47], the notion of a *link* in a path expression is introduced. A link is conceptually a connection between parts of a path expression separated by a single dot. A path expression of the form $set\_el.ivar_1 \ldots ivar_n$ has n links in it between each $ivar_{i-1}$ and $ivar_i$ (where set_el is considered to be $ivar_0$). For example, the path expression *set_el.proj_.manager_.income_.rate_* has links between *set_el* and *proj_*, *proj_* and *manager_*, *manager_* and *income_*, and *income_* and *rate_*.

The GemStone scheme is to generate an index data part for each link mapping the objects referred to by $ivar_i$ to the objects referred to by $ivar_{i-1}$. This means that for any path expression of n instance variables, there are n data parts, and the *ith* data part would contain pairs $< ivar_i, ivar_{i-1} >$. (For notational convenience, we will use an instance variable name as if it were the OID of the object it refers to.) We will refer to the index data part whose pair values are the set elements as the *top-level data part.* For an index using the path expression *set_el.proj_.manager_.income_.rate_*, GemStone would construct four index data parts with mappings $<$ *proj_*, *set_el* $>$ (the top-level data part for this index), $<$ *manager_*, *proj_* $>$, $<$ *income_*, *manager_* $>$ and $<$ *rate_*, *income_* $>$.

The index data parts are organized together by a data structure called an *index entry.* The index entry is an array of component entries in the order of the links of the path expression being indexed. Each component entry contains its index data part and the component entry of the next link. The index entry, component entries, and the data parts together comprise an index. Figure 4.2 shows the index entry using the path expression *set_el.proj_.manager_.income_.rate_* after it has been created for the example set in Figure 4.1. The index entry's component entry array has four elements, which are the component entries containing the four index data parts described above.

Objects that are used in computing a path expression are registered. Objects have a *dependency list* of $<$ *ivar*, *component_entry* $>$ pairs. Each pair indicates that if instance variable *ivar* changes, then the object's entry in the data part corresponding to the component entry needs to be recomputed. A pair is added whenever an object's entry is entered into a data part with a non-`nil`[1] key.

Entries are added to data parts in the following manner. When an object (other than `nil`) is inserted into an indexed set, for every index entry, the object is traced along the path expression to insert pairs into the data parts when needed. A set element and its key are always inserted into the top-level data part even if its key is `nil` or it is already present in the data part. This is so every non-`nil` element of the set is represented in the top-level data part. (I.e., an indexed collection can be a bag rather than a set.)

At each link, if the key is non-`nil` and unique (that is, it does not already exist as a key in the data part), tracing repeats and the pair generated by the next link is inserted into the data part of the component entry for that link. If the key is `nil`, the insertions stop since there is no value for the next link. If the key is non-unique, insertions stop since the pairs generated by the rest of the path expression are already in the later data parts. Tracing stops when the end of the path expression is reached, if it has not stopped before. The data parts shown in Figure 4.2 reflect the index after all three objects in the sample set have been inserted. In addition, each of our objects would have been registered for some of their instance variables when their entries were entered into these data parts. For example, when object α was entered into the *set_el.proj_.manager_.income_.rate_* index, $<$ *proj_*, **L1** $>$ was entered into object α's dependency list.

When a new index is created that uses a path expression that shares links with a path expression that already has an index, only the data parts for the new links are created. For example, if we add an index using *set_el.proj_.manager_.income_.bonus_* to our example set, only an additional component entry with an index data part of $<$ *bonus_*, *income_* $>$ pairs would be created along with the new index entry. Figure 4.3 shows how the new index would fit in with

---

[1] `Nil` is an object of type null; it has no operations and no state. GemStone allows `nil` to be a valid object for any type, thus `nil` can be inserted into any set. Also, any instance variable may refer to `nil`.

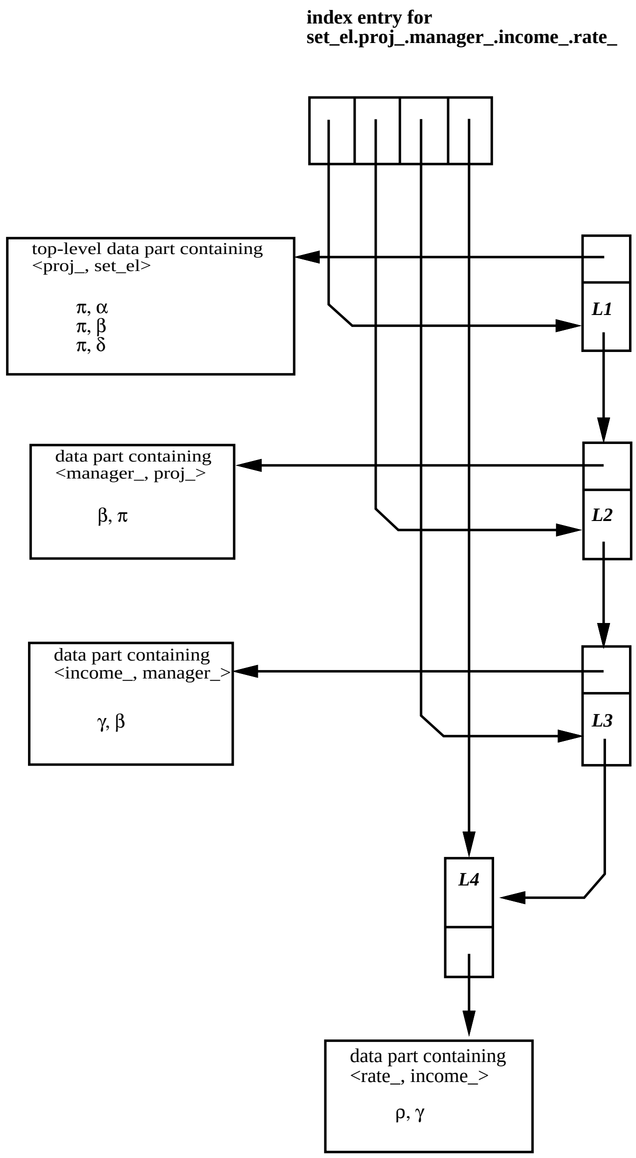


Figure 4.2: Example of a single index in GemStone using path expression *set_el.proj_.manager_.income_.rate_*. (Greek letters are OIDs.)

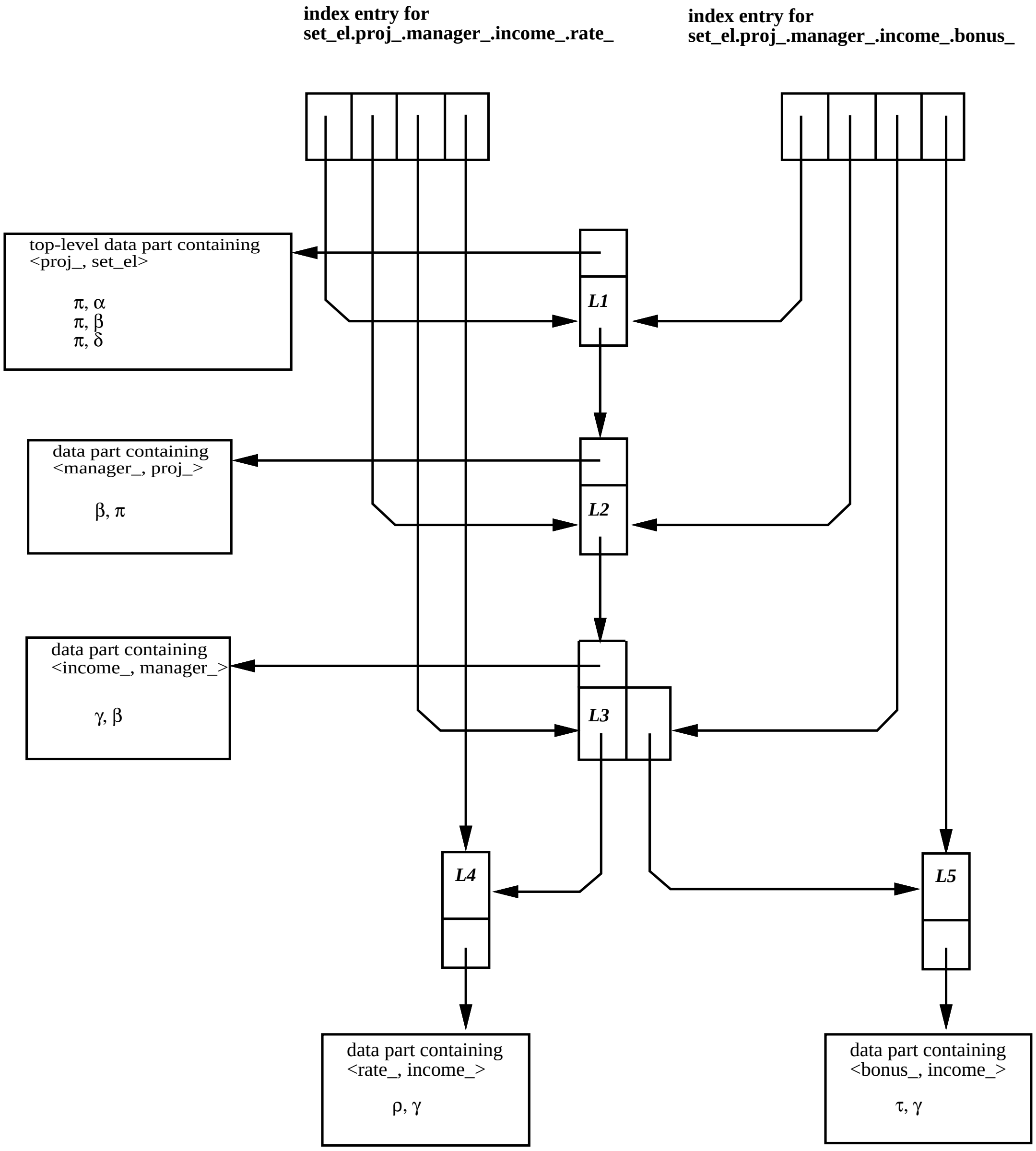


Figure 4.3: Example of two indexes in GemStone that share links in their path expressions. (Greek letters are OIDs.)

the previous index. Note that the index entry for *set_el.proj_.manager_.income_.bonus_* has the same component entries as the index entry for *set_el.proj_.manager_.income_.rate_* except for the last component entry. Also note that **L3** now has two component entries that could be the next one in a path expression involving that link.

When an object is deleted from an indexed set, the procedure is basically reversed. First, the pair in the top-level data part is deleted. If the pair is the last one with a particular key, then the deletion continues to the next links as indicated by the next component entry array. If the key of the pair that is deleted is not the last one in a data part, then deletion stops. That is, there are still objects in the set that refer to the key object, so the rest of its structure must remain in the later index data parts. For example, in Figure 4.3, if we were to delete object δ from the example set, only the $< \pi, \delta >$ pair in the top-level data part would be deleted and then the deletion would stop since there are other pairs with π as a key (namely $< \pi, \alpha >$ and $< \pi, \beta >$). Although the designers of GemStone do not state so explicitly, we assume that the objects deleted from the index are also deregistered.

Mutations are handled similarly to our scheme. It is a deletion followed by an insertion after the mutation. However, the tracing starts at the level of the object that contains the instance variable being changed rather than at the set element. The system can do this because it knows exactly what pieces need to be retraced, namely the the rest of the path expression. When an instance variable *ivar* of object x is changed, first the $< y, x >$ pair, where y is the object that *ivar* refers to, is deleted from the index data part that depends on the instance variable being changed (as indicated by the object's dependency list), then the object is modified, and then it is reinserted into the same data parts that it was previously deleted from. (Recall that deletions and insertions may propagate to lower levels.) If the instance variable changed was in a set element, then all duplicate entries need to be deleted and reinserted with the new key as well. (Duplicates are possible in the top-level index data part because GemStone allows bags to be indexed as well as sets.)

When a query using an index is executed, a lookup is done in the data part of the last component in the path expression to find the values associated with the key(s) in question. If this is the top-level data part, the lookup is finished since the values are the set elements. If not, the values are sorted and then a lookup is done in the data part of the previous component using the values as keys. This process is repeated until the top level data part is reached.

We analyze the space overhead of GemStone scheme in the same manner used in Section 3.8. We compare it with the hybrid scheme, since we concluded that the hybrid scheme was the most suitable implementation of our scheme. Recall our notation:

N = number of set elements.

M = number of non-set elements that are accessed during key computations.

L = number of objects registered per set element.

$L_1, L_2, \ldots$ = number of distinct objects at each level of the index function.

T_{short} = size of set element registration tuple $< I >$.

T_{long} = size of registration tuple $< x, I >$.

S = number of objects that have more than one registration tuple in the hybrid scheme.

P' = size of overhead in hybrid scheme.

The registration information for the hybrid scheme is $RS_{hyb} = N * T_{short} + N * (L-1) * T_{long} + S * P'$. For the GemStone scheme, we assume a dependency list entry in a registered object is the same size as our short registration (T_{short}). There is only one registration per accessed object regardless of the number of times it is accessed during key computations, so the formula for registration information in the GemStone scheme is $(N + M) * T_{short}$, usually a much smaller number than for our hybrid scheme. However, the dependency list entry is not enough to do an update. The intermediate indexes must be accessed, so we must take the space for index(es) into consideration in our comparison.

We assumed in our simulations that each set element had a distinct key and a key occupied 8 bytes for a total of 16 bytes per index entry. This overstates the size of an index, since often many objects have the same key and we expect keys to be like integers, which only occupy 4 bytes in Thor. We can model an index more precisely with the following function:

$$
\begin{aligned}
B(N,K) &= \text{size of an index for } N \text{ objects evenly mapped to } K \text{ keys} \\
&= K * (4 + ((N/K) * 8))
\end{aligned}
$$

That is, there are K entries in the index, each of which maps a 4-byte key to a set of N/K object references. In GemStone, only the lowest-level index contains entries for key values mapped to the objects. In the intermediate level indexes, the "keys" are references, so the size of an intermediate level index is computed as follows:

$$
\begin{aligned}
E(N,M) &= \text{size of an index for } N \text{ objects evenly mapped to } M \text{ references} \\
&= M * (8 + ((N/M) * 8))
\end{aligned}
$$

We ignore the index structure overhead. The intermediate level indexes in GemStone are likely to be hash tables. Since lookups to these tables should always succeed, we assume that they are implemented using an algorithm like Brent's variation of open addressing with double hashing[38] that performs well and uses almost no space overhead even when the table is full, so that the size of the index is the sum of the sizes of the entries. Our index and the lowest-level GemStone index are likely to be B-trees. The GemStone index may be smaller than our index since it only has entries for the objects that directly contain the keys, but the size of the B-tree will be dominated by the entries, so we consider only the entries.

The formula for space overhead in GemStone depends on the level p of the path expression being indexed: $(N + M) * P + E(N, L_2) + E(L_2, L_3) + \cdots + E(L_{p-1}, L_p) + B(L_p, K)$. Note that if the path expression has only one level (i.e., the keys are in the set elements), then $M = 0$, the middle terms drop out, and $M_p = N$ in the last term. For the hybrid scheme, the space overhead for registration information and the index is: $N * T_{short} + N * (L - 1) * T_{long} + S * O + B(N, K)$. For a path expression in the hybrid scheme, $L = p$.

If the path being indexed has one level, the overhead in Gemstone is the same as in our hybrid scheme, $N * T_{short} + B(N, K)$, since $S = 0$. For paths with more than one level, the the amount of sharing determines whether the GemStone scheme or the hybrid scheme require more space. As in the previous chapter, we can graph the space overhead for the GemStone scheme on an index function that accesses five objects in a linear path, i.e., a path expression of the form $V.a.b.c.d.e$, and compare it with the space overhead in the hybrid scheme for the same path. Recall that we assume that no object appears in more than one level and that there is no sharing at the first level (L1 = 1). We use the same values as before ($T_{short} = 4$, $T_{long} = 8$, $P' = 10$). In addition, we assume that the sharing factor of keys is 1 with respect to the number of objects at the lowest level (i.e., $K = L_5$).

Figure 4.4 compares the GemStone scheme and the hybrid scheme when there is sharing at the second level (L2). We see that when there is no sharing (L2 = 1), the GemStone scheme has much higher space overhead than the hybrid scheme, but when the sharing increases the space overhead in the GemStone scheme is reduced rapidly. The crossover point occurs when the L2 sharing factor is less than 2. The reason for this is that the GemStone scheme coalesces the registration tuples for the shared objects into one tuple and the intermediate level indexes all have many less than N entries. In contrast, in the hybrid scheme, each tuple is still present and $S = M$ since all non-element objects have more than one registration when the sharing is at the second level.

Figure 4.5 is a comparison of the two schemes when there is sharing only at the third

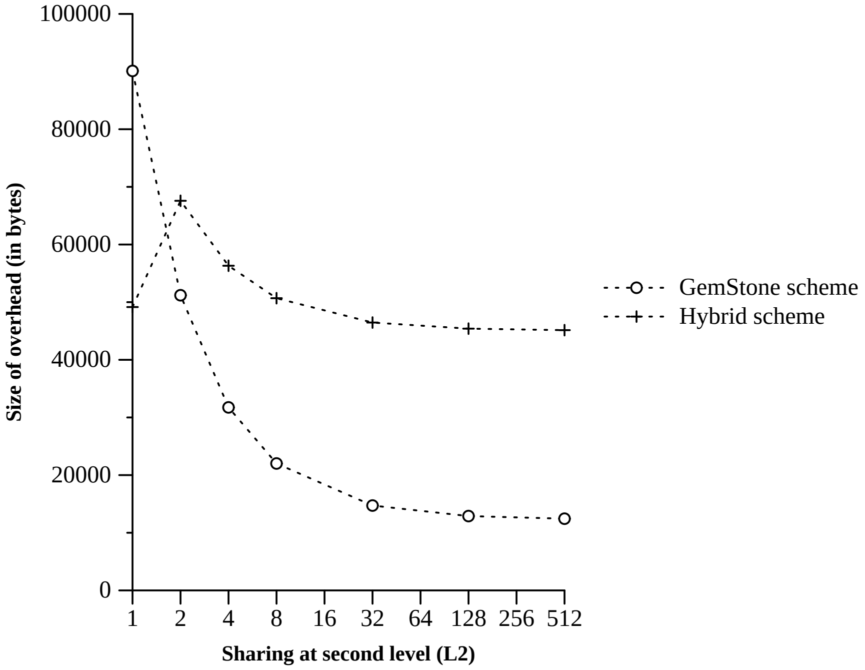


Figure 4.4: Comparison of GemStone space overhead on a 5-level path function with sharing at the second level for a set with 1024 elements (L1 = L3 = L4 = L5 = 1).

---

level (L3). Here we see that the crossover point at which the GemStone scheme incurs less space overhead than the hybrid scheme is when the L3 sharing is about 2, even though the first two intermediate level index have N entries. Figure 4.6 shows that if there is no sharing until the fourth level (L4), the GemStone scheme has higher space overhead than the hybrid scheme. The reason for this is that M has become fairly large $(3 * N + L_4 + L_5)$, and the first three intermediate level indexes have N entries each, thus offsetting the space savings from the sharing at the lower levels.

A query in the GemStone scheme has a computational component proportional to the number of levels in the path expression, since there are as many index data part lookups as there are index data parts. This probably would not affect its performance on our query benchmark with a cold cache, since these the index data parts are likely to be in memory and disk accesses to bring in the data objects dominate the cost of the benchmark. We would expect a running time between the original pointer scheme and the bit schemes, since each registered object has a 4-byte registration tuple, and somewhat better than the hybrid scheme since some of the data objects in the hybrid scheme will have 8-byte registration tuples or references. However, if the cache is warm, the computational component is the running time of a query. In this case, the running time of the hybrid scheme will be less than the GemStone scheme, since we only have

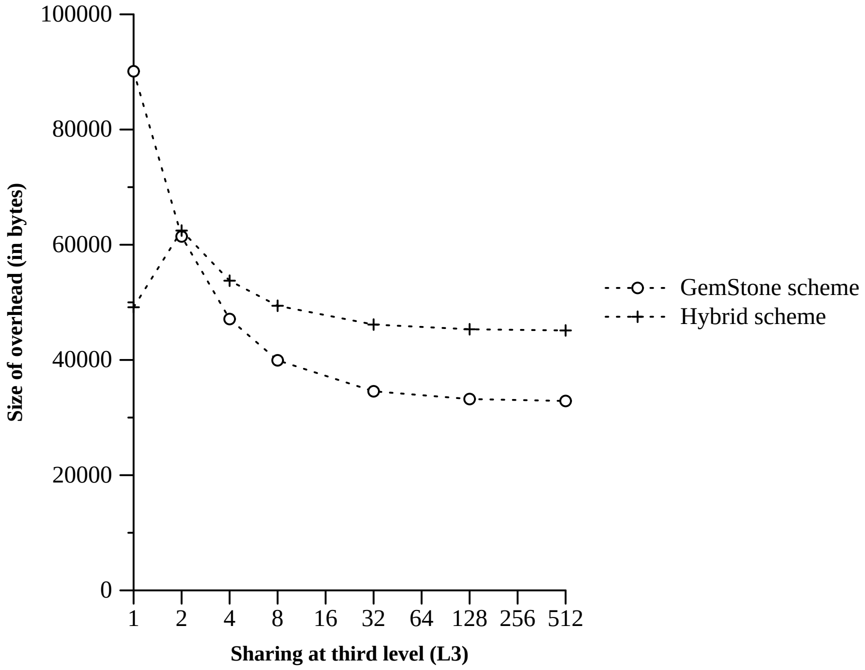


Figure 4.5: Comparison of GemStone space overhead on a 5-level path function with sharing at the third level for a set with 1024 elements (L1 = L2 = L4 = L5 = 1).

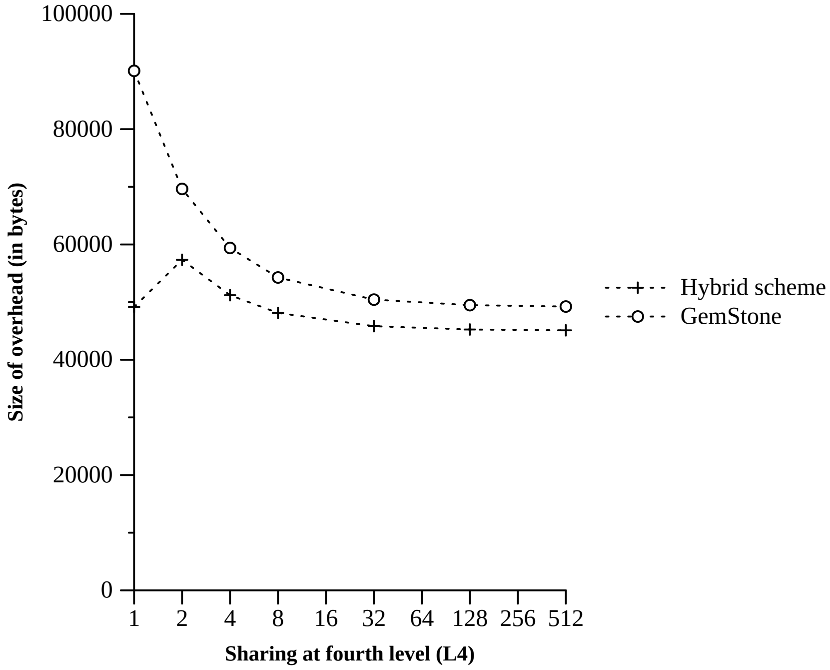


Figure 4.6: Comparison of GemStone space overhead on a 5-level path function with sharing at the fourth level for a set with 1024 elements (L1 = L2 = L3 = L5 = 1).

to do one lookup.

For navigation, the effect of registrations in the GemStone scheme also would be between the original pointer scheme and the bit schemes with a cold cache as in the query benchmark, and would not affect performance when the cache is warm. For updates, there is no clear comparison. In the hybrid scheme, we might have to access the mutated object's registration object (if it is shared) and then access the objects used in recomputing the key starting at the set element. Updates in GemStone only access the objects below the mutated objects rather than starting from the set element (and only if they are not shared). However, in order to do this, the lower-level index data parts associated with these objects must be accessed, bringing in not just the registration information for the object being mutated but also the registration information for all the other objects at that level. Depending on the sharing, these index data parts could be large.

Our conclusion is that the Gemstone scheme is very good for indexing path expressions. The implementation is especially appealing because the number of registrations is proportional to the number of accessed objects rather than the number of set elements times the number of registrations for each set element. In addition, limiting index updates to those parts directly affected by the mutation of an object is attractive. For example, a mutation of the *rate_* instance variable in a *income_* object will cause only one deletion and one insertion in the lowest index data part of the *set_el.proj_.manager_.income_.rate_* index. Thus, we might like to adapt this organization for our scheme.

However, it is not clear whether we can take advantage of this organization. The index data structure for path-based indexing is straightforward because the graph of objects accessed is a straight line. Each object is accessed once in order from root to leaf, and the key is computed by a series of *get_* instance variable methods with the last one returning the key. Thus, one can start in middle of the object graph for a path expression, find the new key, and update the index in a straightforward manner. In function-based indexing, the graph of objects accessed is a general one, so that the index data structure would be more complex than that for path-based indexing. More importantly, the object graph can be traversed in an arbitrary manner with the ends of the paths combined with each other in computing a key. We might be able to determine which paths are being mutated, but the values at the ends of the paths do not tell us anything about the effect of this new path value on the overall computation of the key. For example, even if we could determine that the *rate_* instance variable of object γ was the one that changed because of a mutation, we would not necessarily know what its effect is on the computation of *project_manager_income* for object α. We could attach a function to the graph at that point that does the right recomputation using the new value, but trying to determine the effect of a particular change to the result of an index function is likely to require a user to write the needed function. This type of scheme would be similar to the compensating actions in the GOM system[34, 35] that we will see later in this chapter.

In addition, our object model allows multiple representations of a type, thus the object graph for a function may not be identical for each element in the indexed set; we would need to provide an additional group of index data parts for each distinct graph structure. (Note that it is not only the possibility that the set elements have different representations, but also the possibility that each subobject of set elements with the same representation might have different representations.) Managing this proliferation of data structures would be tricky.

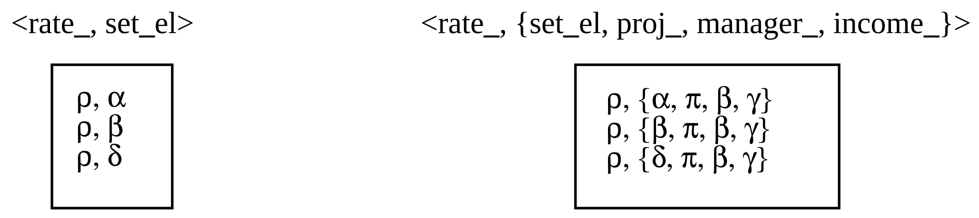
Finally, we note that in our scheme, we do not necessarily register all of the objects that are accessed. We do not register objects that are immutable or when the observer used does not depend on any mutators, so $L = p$ is only an upper bound on the number of objects registered for a path expression function and $N * L$ is the upper bound on the number of registration tuples we must store. In the GemStone scheme, all M objects are registered whether or not they can be mutated in a way that affects an index. If many of these objects are immutable or accessed by an observer that does not depend on any mutators, the space for registration tuples in our scheme will be closer to the space used in the GemStone scheme.

4.1.2 Nested indexes and path indexes

Since the GemStone scheme incurs extra space overhead for the lower-level index data parts, it is an interesting question whether a different index data part organization using less space could be used to implement path-based indexes. Two different index data part organizations that would reduce the number of index data parts have been proposed by Bertino and Kim: the nested index and the path index[11]. Bertino and Kim also compared these schemes and the GemStone scheme using an analytical cost model for both space overhead and time of execution[11]. We will come back to their conclusions after we present the alternate implementations.

A nested index has only one data part that maps the final object of a path expression to its set element. For example, for an index using *set_el.proj_.manager_.income_.rate_* as its path expression, the data part consists of only $<$ *rate_*, *set_el* $>$ pairs as is the case in our scheme. Figure 4.7(a) shows the data part for this nested index on our sample set.

Insertions and deletions are straightforward. The set element is traversed along the path expression and the final object of the path expression and the set element are inserted or deleted



(a) Nested index (b) Path index

Figure 4.7: Data parts for nested and path indexes using path expression *set_el.proj_.manager_.income_.rate_*.

---

from the data part for that index.

Although registration is not discussed, we infer from the examples in [10] and [11] that Bertino and Kim's model of the world allows dependency information to be stored with the type rather than with each object, and it is a pair $<$ *ivar*, *index* $>$ that indicates *index* depends on the *ivar* instance variable of all objects of the type. As a result, the nested index scheme is very inefficient during updates; it traces down the path from the mutated object to the key, but then to find the set elements that are affected, the set elements that are associated with the key must be examined to determine if the mutated object is a subobject in their path. (Different paths may end in the same key, so only the set elements that have the mutated object in their paths are the ones that need to have their index entries updated.) This is especially inefficient if the mutated object is more than two levels down.

The update problem of the nested index stems from not having enough registration information; thus the set elements that might be affected by a mutation must be searched to make sure they are the right ones. Although this scheme would have no effect on queries or navigation, its effect on updates is very detrimental. Thus we conclude that the nested index scheme is not practical.

Path indexes were proposed to overcome the update inefficiency in nested indexes. A path index only has one data part as well, but instead of mapping the final object of the path to the set element, it maps the final object to the set of all of the objects along the path expression. (We will call these *value sets*.) For an index using the path expression *set_el.proj_.manager_.income_.rate_*, the data part would be $<$ *rate_*, {*set_el*, *proj_*, *manager_*, *income_*} $>$ pairs. Figure 4.7(b) shows the data part for this path index on our example set. Like the nested index, there is registration information on the basis of type that determines if a mutator has affected an index entry.

Insertions and deletions are straightforward here as well. The set element is traversed along the path expression and all objects accessed are gathered into the value set that is associated with the final object in the path in the data part. The appropriate data entry is inserted or deleted, respectively.

When a mutation occurs, the key at the end of the mutated path is found by tracing the path from the mutated object to the final object. Since all of the objects on a path are expressed in the path index, this structure can be used to determine the set element without testing all of the set elements to determine if the mutated object is a subobject by finding the value sets that have the final object as their keys. For all value sets containing the mutated object, the path starting at the mutated object has been changed and must be updated with the result of tracing the new path from the mutated object down to the final object. The higher-level objects of the paths do not need to be accessed.

The path index also supports a different type of query that Bertino calls a "projection on path instantiation." A general example of such a query would be one that has two selection predicates, one on the final object of the index path expression, and one on an instance variable of an object in the same path expression. For example, "Give me all employees where *set_el.proj_.manager_.income_.rate_* $>$ 10 and *set_el.proj_.manager_.name_* $=$ 'Smith'." This query could be answered by looking up 10 in the *set_el.proj_.manager_.income_.rate_* index, then accessing all of the *manager_* objects in the value sets associated with 10 for *name_* $=$ "Smith." This is faster than normal processing since there is no need to follow references from the set elements to find the *manager_* objects of interest (though there still will be a need to follow references from the *manager_* object to the *name_* object to determine if *name_* $=$ "Smith"). This could be done in the multi-index scheme as well, if the lookup procedure saved the objects gathered from the appropriate data level lookup.

The path index fixes the nested index update problem by storing the path for each set element. Each reference can be viewed as a registration for its object. The number of references in the path index is $N * L$, because it store a reference for every level for each set element, as in our scheme. Like our scheme, if there is little sharing, a path index takes up less space than in a GemStone index. When there is lots of sharing a path index has much higher space overhead than a GemStone index, since the entries in a path index store the same information in their value sets (as in our example, where all three value sets contain π, β, and γ). We observe that this common information could have been stored separately and shared to save space, but in general the sharing may happen at several levels, so it is not clear how to organize the common information. For example, another `project` object π' might also refer to β as its *manager_*, so the value sets of set elements using π' would be $\{\pi', \beta, \gamma\}$ and there is now a question of whether just β and γ should be grouped together and referenced in each value set, or whether there should be two groupings $\{\pi, \beta, \gamma\}$ and $\{\pi', \beta, \gamma\}$ and whether these two groupings should reference the group of β and γ instead of naming them both explicitly.

The path index scheme is an interesting implementation because it stores the registration information for an index inside the index's entries. Since there is no registration information in an object, the path index scheme has no effect on navigation. The index entries are larger because they store value sets instead of just a reference to an element, so a path index is larger than an index in our scheme. This may affect queries if the index can no longer be kept in memory. For updates, there is a check for the registration information in the type on every mutation, though this information can probably be kept in memory, or the objects that need to check can be marked with a bit in the header as in our scheme. Finding the key requires accessing only from the mutated object down, but finding the index entries that are affected requires checking the value sets of the matches for the mutated object. Recomputing the index entry only requires accessing from the mutated object down along the new path.

It is not clear how we might use this organization. We can store references to the objects accessed, but the main problem is that we do not have an easy way to find the affected key in order to lookup the registration information. Also, as in the GemStone scheme, we may not be able to determine what the new key is without recomputing it from the top. However, if we keep precomputed results, this organization might be useful, because the information about which objects may be needed for recomputation is stored in one place, and the system might be able to use this information to preload the cache with these objects to make updates run faster.

4.1.3 Join indexes

Join indexes were originally designed for distributed relational databases to make joins of sets on different machines more efficient[60]. Bertino generalizes this technique for object-oriented databases[10]. A join index is basically a GemStone index with corresponding index data parts mapping objects to each other in the opposite direction. For example, for the path expression *set_el.proj_.manager_.income_.rate_*, not only are there the four data parts as created in the GemStone scheme but also data parts of < *set_el*, *proj_* >, < *proj_*, *manager_* >, < *manager_*, *income_* > and < *income_*, *rate_* > pairs. Though this information can be easily obtained by traversing an object, the reverse index data parts would allow the system to go "directly" to the final object rather than accessing objects along the path from the set element to the final object in order to find the value of the final object (i.e., they support path precomputation). A claim is made that these extra index data parts would make object traversal more efficient in the cases where the objects along the path are not co-located on a single server. Bertino also notes that this idea can be applied to nested indexes for similar benefits.

This scheme provides more expressive power to find the result of a path expression without accessing the objects by using more space for the extra index data parts. However, this extra expressive power is paid for by extra space for the reverse index data parts and in time during updates when the extra index data parts must be updated. Otherwise the analysis of the purely indexing aspects of this scheme would be similar to that for the GemStone scheme since we would only be using the half of the index data parts that would be present in GemStone.

4.2 Function-based Indexing

There are two function-based indexing schemes that have been proposed: *function materialization*, implemented in the GOM system[34, 35], and *method precomputation* proposed by Bertino[8, 10]. The idea in both schemes is that the result of a method that involves a computation can be precomputed and stored as an additional instance variable for an object. As an added benefit, these results are used to provide an index based on the precomputed method. The precomputed results need to be recomputed when there are object modifications in the same situations where an index entry would be recomputed in our scheme; therefore these systems must solve the same set of problems that we have solved.

In this section, we will examine these schemes in some detail. We are interested in how they differ from our scheme both in their expressive power and in their implementation details. In addition, we briefly discuss the Cactis system[31, 32], an entity-relationship database that maintains derived attributes.

4.2.1 Function materialization

The scheme most closely related to ours is the GOM scheme[34, 35]. The designers of the GOM scheme are primarily interested in storing the results of invocations of an object's method so that when the method is called again, the result is already available without any computation. However, as they point out, one can also use this information to find the objects that have a particular result for the method as well. The ability to return precomputed results is expressive power that our scheme does not have. In addition, the GOM scheme allows the precomputation of methods that have arguments (i.e., arguments other than the implied set element). Both of these ideas are extensions that we would like to explore for our scheme; they will be discussed briefly as future work in Chapter 7. To make the comparisons here easier, we will assume GOM indexes only for methods without arguments and will ignore the space requirements to support the precomputed results.

As an indexing scheme, the GOM scheme is not as expressive as our scheme. The only sets that can be indexed are type extents (sets that contain all objects of a particular type). This makes sense since the GOM designers are mainly interested in eliminating method computations. With respect to indexing for queries, this is a very "relational" view of the world where the only sets of interest are the sets containing all of the records of a particular type. Subsets are computed on the fly by describing them as the records from the main set that have a certain property. We expect that users of object-oriented databases will want more flexibility to create user-defined sets that match their application domains without encoding the differences as extra fields. Typically, these sets will not encompass all the objects of a particular type, so applications will not want to pay for indexing that they are not using. For example, to model a business it might be useful to have several sets of employees, one for each department.

Also the only functions that can be used as index functions are (observer) methods. This a very static view of the world; it expects that the only functions of interest are the ones thought of by the designers of the type. However, it is unlikely that a type designer can think of all of the computations that a client might want to do. Our *project_manager_income* function is one example, as is the function we used in our second query benchmark that given a composite part, counts the number of 'e' characters in its associated document.

Since a precomputed method can be an arbitrary computation, a precomputed result for an object and the corresponding index entry can be affected by mutations of objects other than set elements as in our scheme. The description of the GOM implementation starts with a very inefficient scheme and incrementally adds "optimizations" to achieve acceptable performance. The optimizations that bring the GOM system to a state that is roughly equivalent to our basic scheme are incorporated in our description here. Other optimizations will be discussed in Chapter 5 along with the optimizations to our scheme.

Conceptually, a GOM object is a tuple with fields for its instance variables and fields for its precomputed method results. The tuples representing objects of the same type are stored together in a table (i.e., the type extent), thus an index based on a method is just an index over the field containing the method's precomputed result. Queries using precomputed methods are computed using these indexes. Figure 4.8 shows our example set with one precomputed method. (We will assume that *project_manager_income* is a method of `employee` objects.) The precomputed results of a type are stored separately from the instance variables values in a table

OID	id_	name_	address_	income_	proj_	med_	*project_manager_income*
α	$\cdots$	$\cdots$	$\cdots$	$\cdots$	π	$\cdots$	75000
β	$\cdots$	$\cdots$	$\cdots$	γ	π	$\cdots$	75000
δ	$\cdots$	$\cdots$	$\cdots$	$\cdots$	π	$\cdots$	75000

Figure 4.8: Example GOM type extent for `employee` objects with one precomputed method.

keyed by OID.

Indexes in the GOM scheme are maintained using a registration technique like ours. Objects are registered by collecting the set of all accessed objects during the precomputation of a method O of a set element x and then informing an index manager of this set along with x and O. The index manager then adds registration tuples of the form $< y, O, x >$ for each object y in the accessed object set. The registration table is a separate global table keyed on y. Figure 4.9 shows the registration table for our example set. In addition, each accessed object stores the name of the precomputed method that accessed it. For our example set and index function, all of the (f-reachable) objects shown will have *project_manager_income* in their precomputed method lists. This information is used in index maintenance.

A static analysis determines which mutators need to check for registrations in the following way. When an observer is to be precomputed, the system examines all the *code* for all the types used in computing that observer (i.e., this includes the code of any subobjects of the set element that it accesses) and makes a conservative approximation about what instance variables are used in computing a method. This analysis is based on computing all the possible path expressions that an index function function might possibly traverse. Then the system rewrites the code of the primitive mutators of these instance variables (i.e., the *set_* instance variable methods) to check for registration information and notify an index manager when an object that was used in a precomputation has been modified, if necessary. This code is recompiled, and from then on, all objects of the type run this code, whether or not they participate in any indexes. In addition, for each primitive mutator, a list of all precomputed methods that the mutator may affect is associated with the mutator. For example, the *set_rate* mutator of the `income_info` type would have *project_manager_income* in its list. This list is compiled into the new mutator code and is used in index maintenance

When a modified mutator of y is executed, the registration information for y is checked if the intersection of the mutator's list of precomputed methods and y's list of precomputed methods that have accessed it is not empty. If the intersection is empty, y has not been accessed by a

y	O	x
α	*project_manager_income*	α
β	*project_manager_income*	β
δ	*project_manager_income*	δ
π	*project_manager_income*	α
π	*project_manager_income*	β
π	*project_manager_income*	δ
γ	*project_manager_income*	α
γ	*project_manager_income*	β
γ	*project_manager_income*	δ

Figure 4.9: Example GOM registration table.

method precomputation; therefore mutations to it cannot affect any indexes. If the intersection is not empty, then y was accessed during a method precomputation and the index manager is informed of the mutation of y. For example, if *set_rate* is called on γ, the index manager call will be informed because both the list for the *set_rate* mutator of the `income_info` type and the list in γ contain *project_manager_income*. However, if we mutate a "random" `income_info` object σ in the same way, the index manager will not be informed because σ's list will not contain *project_manager_income*.

When the index manager is notified of a mutation for an object y, it looks in the registration table for any entries for y; if there are any it marks the affected index entries as invalid and arranges for recomputations. These recomputations are done *lazily*, that is they are only done when the index needs to be used again. We will discuss a lazy update optimization to our scheme in the next chapter.

The GOM scheme registers more objects than necessary. For example, immutable objects will be registered, as will objects that are being accessed by observers that do not depend on any mutators. In addition, the GOM scheme causes more recomputations than our scheme. A mutator that sets an instance variable may not change the result of an observer that reads that instance variable (e.g., rotating a square changes its coordinates, but does not change its area; also, the observer might do a benevolent side-effect). The reason is that analysis of instance variables only approximates what we get from our *depends_on* declarations. The instance variable analysis is computing dependency at the concrete level. At this level, *get_* instance variable observers depend on *set_* instance variable mutators. Our *depends_on* relation is at the abstract level; we are interested in dependencies in the abstract state rather than in the concrete implementation. The designers of GOM realized that their technique registered too many objects and suggested an optimization that we will discuss in the next chapter with our subobject containment analysis optimization. In addition, they also developed the concept of a *compensating action* that is run when a particular instance variable is modified that takes the old key value for the set element that has been affected and the new instance variable value and computes the new key value without accessing other objects. A compensating action must be written explicitly by a database maintainer, and this technique only applies in limited cases, since not all modifications will result in a functional transformation of the old key to a new key.

It is instructive to compare the implementation of GOM to our scheme to see if they benefit from providing indexes only for type extents using methods. We will only consider the portions that are directly related to providing indexing over functions and will not consider the portions related to providing precomputed results. The indexes in both schemes are the same size, so we consider only registration space overhead. Each registered data object in GOM stores the precomputed method that accessed it, that is the registration information stored in an object is $< O >$. As in our scheme, this would take 4 bytes (T_{short}). As in our scheme, there are $N * L$ registration tuples, since each access causes a registration tuple to be stored. Since registration tuples are stored in a separate location in a table, the entire tuple $< y, O, x >$ must be stored. O can be encoded in the reference for x as in our new pointer scheme, so that each tuple is 16 bytes. Thus the registration space overhead in the GOM scheme is: $(N + M) * T_{short} + (N * L) * 16$. For the 5-level path function example we used in our previous analyses, the GOM scheme uses much more registration space overhead than the hybrid scheme in any situation. Also note that to find the entries in the registration table for an object y efficiently, some kind of index would have to be provided.

Computationally, the two schemes are about the same. The 4 bytes of registration information stored in each data object will cause query and navigation performance between the bit schemes and the original pointer scheme. For updates, the GOM scheme compiles the list of precomputed methods that a mutator can affect into the mutator's code, but it still needs to compute the intersection of this list with the object's list of precomputed methods. This computation is more expensive than the header bit check in our scheme, though our technique could be used in the GOM scheme to avoid the full check during mutations of unregistered objects as well. Note that in our scheme, we do not change a mutator of an object to the checking version unless it is registered for that mutator. As a result, our scheme often does not check when an object is unregistered.

In summary, the GOM scheme is a method precomputation scheme that uses precomputed results as the keys for an index over the precomputed method. The ability to provide precomputed results is extra expressive power that our scheme does not have. However, it requires additional space that is not present in our scheme. As an indexing scheme, the GOM scheme has less expressive power than our scheme, since it is limited to indexing over type extents using methods.

The implementation of the GOM scheme is based on the same fundamental ideas as our scheme though the details are different. Objects are registered when they are accessed during key computation. Dependency information is used to determine what mutators need to check for registration information and when index updates are necessary.

The dependency information in the GOM scheme is not as precise as in our scheme, because it is inferred from the concrete implementation of a type, whereas dependency information in our scheme is declared at the abstract level. This can cause the GOM scheme to register more objects than in our scheme. These extra registrations take up space and can cause unnecessary updates. In addition, because the dependency information is inferred from the concrete implementation of a type, the GOM system restricts types to have only one implementation. Multiple implementations of a type would be hard to support in the GOM scheme, since the instance variable analysis would have to be done for each possible combination of implementations that could be used by the index function. If many types are used and they each have many implementations, the analysis might take a long time. Also, when a new implementation is added to the system, this analysis would have to be repeated to include the new implementation.

Another implementation difference is that the registration table in the GOM scheme is stored and managed in a central location. This is reasonable in a single server system, but Thor is distributed, and in particular, the objects that affect an index entry may not be stored at the same server. In this type of system, it would be better if the registration information for an object is stored at the same server as the registered object, so we would not use a centralized implementation.

4.2.2 Method pre-computation

Bertino's method pre-computation scheme[8, 10] is another scheme for storing precomputed results of methods and using these results as keys of an index. Bertino's scheme has the same expressive power as the GOM scheme, except that it does not support methods with arguments. It provides the expressive power to return precomputed results, but as an indexing scheme, it can be applied only to type extents and only methods can be index functions. As we did for the GOM scheme, we will only be concerned with the aspects of this scheme that support indexing functionality and will not consider support for precomputed results in our comparison.

Bertino also uses a registration scheme. Objects are registered whenever their instance variables are used to access the subobject referred to by the instance variable. (For convenience, we will say the computation accesses the instance variable when we mean it accesses the object referenced by the instance variable.) Like the GOM scheme, this is analogous to assuming dependency between *get_* instance variable observers and *set_* instance variable mutators. Unlike our scheme and the GOM scheme, there is no static analysis; there is no attempt to limit registration checking only to those mutators that can possibly affect a precomputed method result, so all mutators of an object must check for registrations.

Registration information for a registered object y is stored in the following forms:

- Two flags, im and em, for each instance variable indicating whether it has been accessed during a method precomputation. The im flag indicates that a precomputed method of y accessed the instance variable; the em flag indicates that a precomputed method of a different object accessed the instance variable.
- A list $\mathcal{M}$ of precomputed methods records. An entry has the result of the precomputation, a flag indicating whether the result is valid, and information about which instance variables of y were accessed.
- A list $\mathcal{P}$ of "parent" records. Parents of y are those objects that called a method of y during a method precomputation for another object. An entry contains a pointer to the parent, and a set of records that contain a method identifier for the precomputed method, a flag that indicates whether the entry is valid, a flag that indicates whether the parent is a set element storing the precomputed result, and information about which instance variables of y were accessed.

The flags im and em are set as one would expect. The initial entry for a parent gets added to $\mathcal{P}$ whenever a method of y is called by the parent during a method precomputation of some other object for the first time. Subsequent records are added to the entry for the parent if other precomputed functions access the same object through the parent. Parent record entries are marked as valid whenever the parent access y during a precomputation. The entries in $\mathcal{P}$ are used during index invalidation.

Index entries are not recomputed when a mutation happens. Instead, they are only marked as invalid. Invalidation happens as follows. Every time an instance variable $ivar$ of object y is changed, $ivar$'s flags are checked. If neither flag is set, the mutation happens normally. If the im flag is set, the entries in $\mathcal{M}$ for precomputed method O that indicate they depend on $ivar$ are found. The result for O in each of these entries is marked as invalid, and the corresponding index entry in the index for O is replaced with an entry for the set element associating with it the undefined key, "Ω."

If *ivar*'s *em* flag is set, each valid entry in $\mathcal{P}$ that indicates that it depends on *ivar* for some precomputed method O is used to find set elements whose result are no longer valid. This is done by following the parent pointers up the object graph, backtracking along the path taken by the original precomputation. The backtracking works as follows. For each parent record entry, do the following:

1. Access parent p.

2. Find the instance variable $ivar_p$ of p refers to y.

3. Find all valid parent record entries in p's $\mathcal{P}$ that depend on $ivar_p$ and invalidate them. If entry indicates that parent is a set element x, access x, mark the result for O in x's $\mathcal{M}$ as invalid, and the corresponding index entry in the index for O is replaced with an entry associating x with the undefined key. Otherwise, repeat backtracking at the next level.

By invalidating the parent record entries during backtracking, the scheme cuts short invalidations coming up from other branches of the f-reachability graph. We assume that if the scheme discovers that p no longer refers to y from any of its instance variables, it terminates.

The invalidation scheme is incomplete because it assumes that a parent object is never both a set element and part of the path in computing a key for another set element. Suppose that `project` objects have a method *num_employees* that returns the number of employees in a project and we have an index over the function shown in Figure 4.10(a) that computes the number of employees in the employee's project manager's project. Figure 4.10(b) shows an example where an object's parent is a set element containing a precomputed method result and part of a path used in precomputing the method result for another set element. The parent record entry created for f in `project` object π should indicate that β is a parent that does not contain the precomputed result because it is accessed in computing $f(\alpha)$, but it should also indicate that β is a parent that contains the precomputed method result because it is accessed along the same path in computing $f(\beta)$.

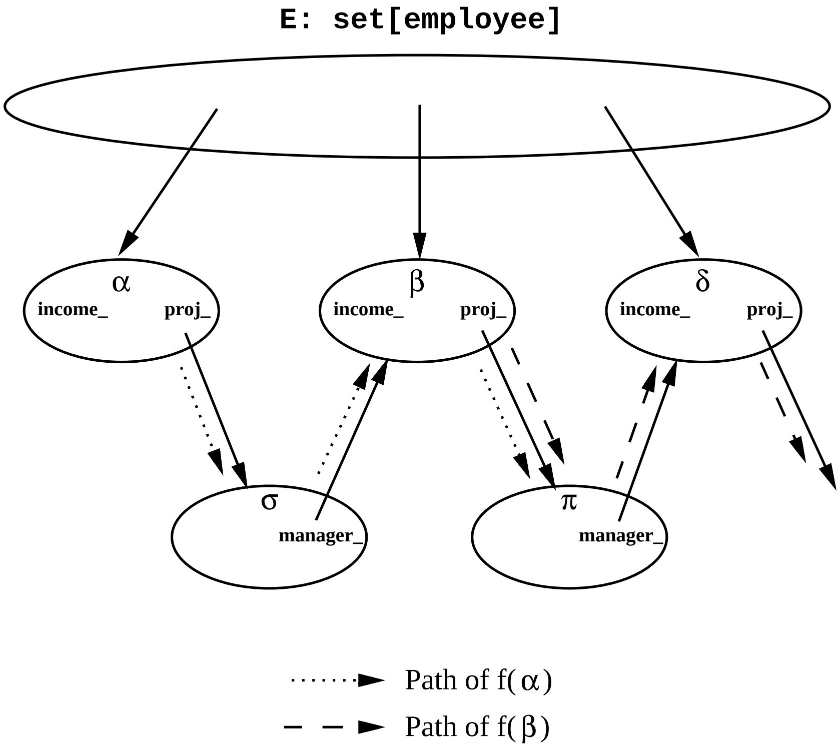
The invalidation scheme is also incomplete because it assumes that there will be only one instance variable in the parent that refers to the child. The algorithm can be easily generalized to do a complete check of all instance variables that refer to the child and then find any parent record entries that depend on any of these instance variables.

When a query using a precomputed method O is run, the index for O is consulted, and all elements with keys that satisfy the query automatically belong to the result set of the query.

f (e: employee) **returns** (int)

p: project := e.project ()
m: employee := p.manager ()
mp: project := m.project ()
return (mp.num_employees ())
end f

(a) Index function



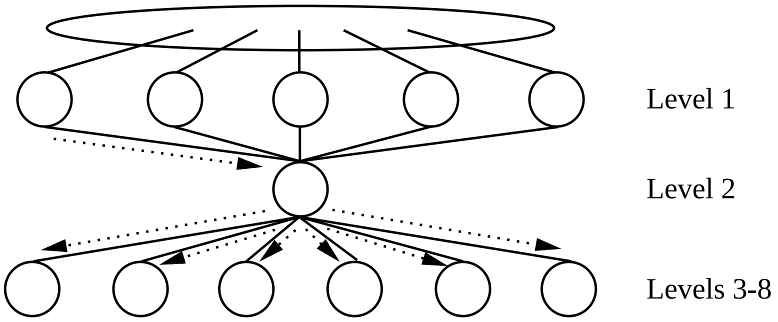
(b) Example set

Figure 4.10: Example where a parent is a set element that contains a precomputed method result and is part of a path used to precompute the method result for another set element. Object π is access from β during the computation of both $f(\alpha)$ and $f(\beta)$, so its parent record entry for β should indicate that β is both a parent with a precomputed method result and a parent used in precomputing the method result of another set element.

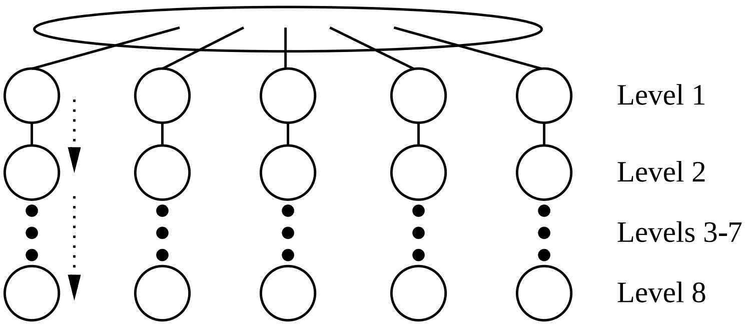
In addition, O is run on any set elements with undefined keys to test for inclusion. Each result is stored in the set element's entry for O in $\mathcal{M}$, and index entries are entered associating the set elements with the result.

We can characterize the registration information space overhead in the following way. There are 2 bits per instance variable for their flags. These bits can be stored in a bit vector. (We do not want to store these bits within a reference because that would cause extra work on every access to mask them out.) The actual size of the flag vector depends on the number of instance variables, but since we assume that data is aligned on a word boundary, it would have to be a multiple of 4 bytes. We will assume 4 bytes for the flag vector; this will encode the flag bits for 16 instance variables, which seems like a reasonable number. A method record has a method identifier, a method result, a validity flag, and dependency information. (We count the space for the method result because it is needed during updates.) Dependency information can be represented as a bit vector, 1 bit per instance variable, so it is 2 bytes. The method identifier is 2 bytes (and can probably also encode the validity flag) and a method result is 4 bytes, so a method record can be stored in 8 bytes. A parent record has a method identifier, a parent pointer, a validity flag, and dependency information. We probably cannot encode both a method identifier and the dependency bit vector into the parent pointer, so parent records are 12 bytes (8 bytes for the parent pointer and 4 bytes for the method identifier and dependency bit vector, the validity flag again encoded into the method identifier). Note that there is only one parent record entry per parent (per index) regardless of how many times the object is accessed from that parent during the key computations for an index. However, an object may be accessed by more than one parent during key computation, so the number of parent records depends on the sharing structure of the elements in the set. Thus to add an index in this scheme, the registration space overhead is $(N + M) * 4 + N * 8 + X * 12$, for the flag vectors, the method records in the set elements, and the parent records, where X varies depending on the sharing structure.

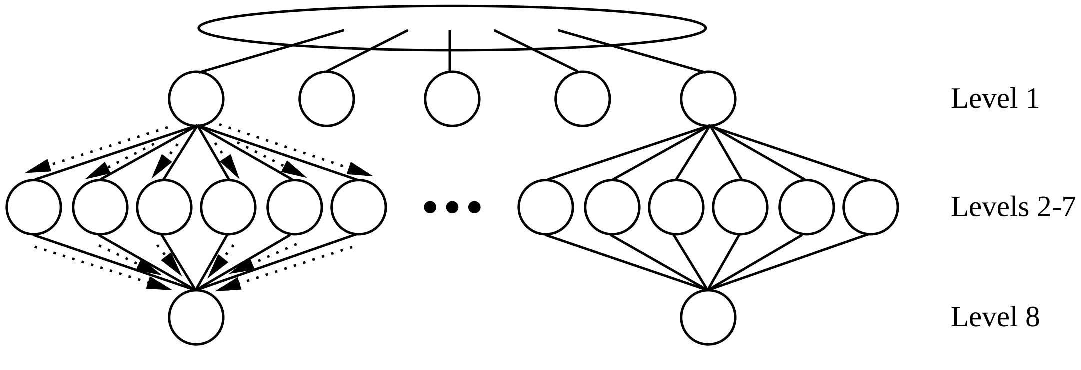
Figure 4.11 shows three scenarios for which the number of registrations in Bertino's scheme (method records plus parent records) is less, equal to, or greater than in our scheme. In each scenario, there are $N = 5$ elements and $L = 8$ (the dotted arrows show the path of the index function for the first set element), thus our scheme needs 40 registrations. For Bertino's scheme, there is always one method record in each set element. In scenario (a), the objects at levels 2 through 8 are shared by all set elements. There are 11 parent records for this scenario, 5 in the object at level 2 and one in each of the objects at levels 3 through 8, for a total of 16 registrations. In scenario (b), the index function is a path expression. This scenario needs 35



(a) Our scheme: 40 registrations, Bertino's scheme: 16 registrations



(b) Our scheme: 40 registrations, Bertino's scheme: 40 registrations



(c) Our scheme: 40 registrations, Bertino's scheme: 65 registrations

Figure 4.11: Comparison of the number of registrations in our scheme and Bertino's scheme in three sharing scenarios. The dotted arrows indicate the path taken by the index function in computing the key for the first set element.

parent records, one in each object at every level, for a total of 40 registrations. In scenario (c), the key computation for each element traverses an unshared, complex f-reachability graph. In particular, the object at the bottom of the f-reachability graph is accessed through 6 different parents. There are 60 parent records in this scenario, one in each of the objects at levels 2 through 7 and 6 in each object at level 8 (one for each of its parents), for a total of 65 registrations.

There are no implementations of this scheme, but we can do some rough comparisons on the effect of its overhead on performance. Bertino suggests that the registration information can be embedded directly in an object because the space overhead is not too large. Assuming the sharing does not cause a large number of parent records, the effect of this implementation on queries and navigation should be between our original pointer scheme and our original embedded scheme, since registrations are 8 or 12 bytes. The effect of registration information on index maintenance is likely to be similar to the embedded scheme since it is available directly. The number of objects accessed during an update backtrack depends on how far down the graph the mutation take place. If we assume, on average, a mutation takes place near a leaf, then the difference between the schemes depends on how bushy the object graph traversed by the precomputation is. Invalidation in this scheme may only visit the direct path from the leaf to the set element while our old key computation would visit all the objects in the graph, though if the index function accessed the mutated object from many parents, each backtracking will access each one. Recomputing the new key will be the same in both schemes.

Bertino also suggests that this scheme could be implemented by storing all of a precomputed method's results and parent records for an entire index into a separate object and having objects registered for the precomputed method point to it so that the objects along the path do not have to be accessed. (The flag vector would still be stored in an object.) This implementation may cause unnecessary invalidations, since without accessing an object, we cannot determine which of its instance variables actually refer to the child object. If the path no longer exists (e.g. the instance variable that used to refer to the child object now refers to a different object), we may still find a path in the registration object and do unnecessary invalidations. In addition, if more than one instance variable can refer to the same child object (i.e., the instance variables have the same type), we would have to assume that any of these instance variables may refer to the child object and any method records or parent records that depend on them must be invalidated.

This scheme is interesting because it allows the system to backtrack up the object graph along the path that was taken in computing the index key to find the set element and the old key rather than recomputing the old key starting at the set element. However, to do so, this scheme may register more objects than in our scheme. It must register objects that cannot be mutated in a way that affects the index since it needs to keep the entire path in order to do the backtracking. In addition, all of an object's mutators must check a pair of instance variable flags every time there is a change to an instance variable to determine if it was used in a method precomputation. These inefficiencies are due to determining dependency at the concrete level among instance variable accesses and modifications rather than at the abstract level between observers and mutators of the type. However, as we have shown, this scheme sometimes uses less space than our scheme since in some situations it can take advantage of the sharing among objects, and it would be interesting to see if we can combine our dependency information at the abstract level with this scheme to provide a more GemStone-like implementation for our scheme.

4.2.3 Cactis

The Cactis system is an entity-relationship database[31, 32]. In the entity-relationship model, every object has a type that specifies some number of attributes, which are values like integers and booleans, and some number of relationships that relate two (or more) objects and may also have attributes. Cactis supports both intrinsic attributes (i.e., a regular value) and derived attributes, where the value is the result of an arbitrary computation that can use the values of other (possibly derived) attributes of the object or its relationships. In addition, each relationship attribute has a direction (also declared in the type) so that one object in the relationship is a transmitter and may set the relationship attribute while the other other object is a receiver and may access the attribute. Thus, the attributes of a transmitter object can be made accessible to the receiver objects to which it is related. Likewise, attributes of an object γ can be made accessible to an object α that is not directly related to γ if there is a path of objects and relationships that transfer the attribute of γ to α.

The Cactis system stores the results of computing derived attributes so that they need not be computed on every access. Since the attributes used to compute a derived attribute may themselves be derived, when an attribute of an object is modified, any number of other attributes in both the modified object and other objects may need to be recomputed. Determining which attributes are affected when an attribute changes is done in the following way. Each type keeps a list of all the relationships its objects participate in and a dependency vector for each attribute indicating which other attributes of the object or its relationships the attribute depends on. When an attribute is modified, all other attributes (of the object) that depend on it are marked as out-of-date. For every relationship attribute that depends on the modified attribute (directly or indirectly), the objects that are related to the modified object by that relationship are found and any of their attributes that depend on the relationship attribute are marked as out-of-date, and so forth along the path of objects and relationships until there are no more attributes that are affected by the change. Out-of-date attributes are recomputed when they are accessed or if they are marked as "important," meaning that they should be recomputed immediately.

The Cactis scheme is similar to Bertino's method precomputation scheme. An object's intrinsic attributes can be thought of as instance variables that contain non-reference data and derived attributes as precomputed method results. Relationships can be thought of as a pair of references between the related objects and a relationship attribute as an instance variable of the receiver object. The functions that define the derived attributes are the precomputed methods. When a modification is done to an "instance variable", the invalidation phase backtracks along the paths of all computations that accessed that instance variable as in Bertino's scheme. The main difference is that precomputed method results are maintained and used at every level of the computation.

The designers of Cactis indicate that there is a 65-bit per attribute overhead stored in every object in the system, but other implementation details are not described. Therefore, it is not possible to evaluate the space and time performance of this scheme.

Chapter 5

Optimizations

For function-based indexing to work well, the impact of registration and updates on the system must be minimized. Both aspects of our scheme, registration and updates, can benefit from optimizations:

1. The registration algorithm still registers more objects than necessary. In particular, objects may have subobjects that can be mutated only by calling a method of the enclosing object and therefore need not be registered.

2. Updates are inefficient in two ways: an update may do work that will be overwritten before the new index entry is used; and obsolete registrations cause unneeded recomputations.

In this chapter, we describe three optimizations to our indexing scheme:

1. Contained subobject analysis — statically identifying subobjects that can be mutated only by calling a method of the enclosing object.

2. Lazy updates — doing index entry recomputation when needed or in the background rather then immediately upon a mutation of an object that affects the index entry.

3. Deregistration — garbage collecting obsolete registration information to reduce unnecessary updates.

5.1 Contained Subobjects

One way to register fewer objects is to recognize *contained subobjects*. Intuitively, an object contains a subobject if mutations to the subobject can occur only within methods of the containing object. For example, an `income_info` object might be contained by a containing `employee` object (since it is created inside the containing `employee` object, and `employee` objects do not have methods that return the `income_info` object). However, the `project` object

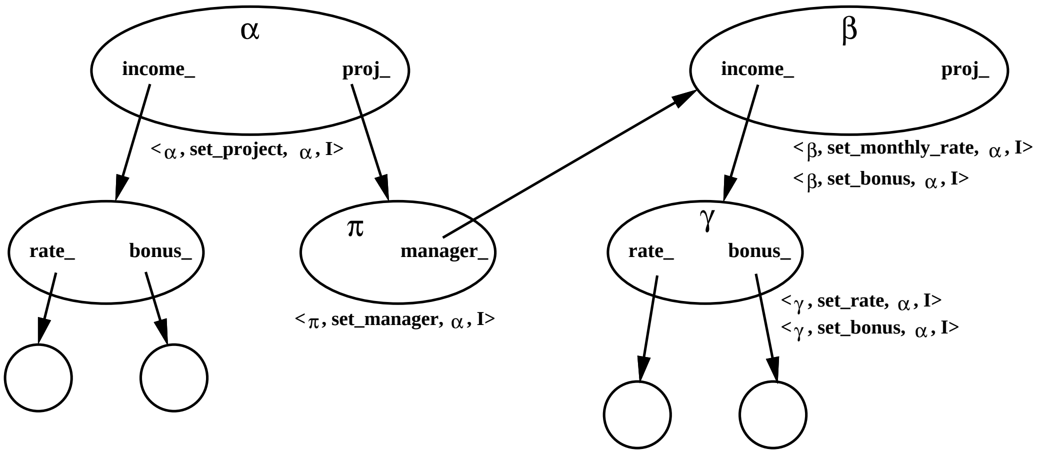


Figure 5.1: Example with registrations from *project_manager_income*(α). (Greek letters are OIDs.)

---

referred to within an `employee` object is not contained, since it can be returned by the *project* method and then mutated via its *set_manager* method.

To see how we could use the knowledge of contained subobjects to register fewer objects, consider our example in Figure 5.1, showing the registration tuples that our scheme generates after computing $I.f_r(\alpha)$ next to each object, and suppose object γ is contained in object β. With the scheme described so far, both β and γ have registrations that will cause index maintenance work when β.*set_bonus* runs. Index maintenance work will be done twice, once inside γ.*set_bonus*, and also in β.*set_bonus*. Thus half the work is wasted; in general, there may be even more wasted work because the mutator of the containing object may mutate several contained subobjects.

The work done in γ.*set_bonus* is totally unnecessary because β.*set_bonus* does what is needed and no other code can call γ.*set_bonus*. This fact holds whenever there is a contained subobject. So if we can recognize contained subobjects we can save both index maintenance work and the space for the (unneeded) registration tuples in these subobjects. (The lazy update scheme presented below can avoid some of this wasted work for contained subobjects, but not registering these objects is even better.) This optimization cannot be done for subobjects that are not contained, however, since they might be modified directly (without an intervening call on a mutator of the containing object).

In this section, we present an algorithm for computing subobject containment. Then we show how containment information can be used to register fewer objects. Finally, we discuss an optimization in the GOM scheme that is related to our subobject containment analysis.

5.1.1 Containment Algorithm

Containment is a property of a class. All of the objects created from a particular class have the same properties with respect to containment. In particular, containment is a property of the subobjects reachable from the instance variables of a class. Any object y that is only reachable from the instance variables of another object x is contained by x because only the methods of x can call the methods of y. We capture this notion by dividing the "world" as viewed by a class into two parts: the part that is reachable only from an object's instance variables and the externally accessible objects reachable from the environment. In this view, contained subobjects are those objects that are reachable only from the instance variables. Our goal is to determine if any of a class's instance variables refer to such subobjects.

To determine if an instance variable refers to a contained subobject, we analyze the code of a class. For languages like Theta that support a subclassing mechanism, we assume that we can "rewrite" a class's creators and methods to include any code that is inherited from any superclasses and their superclasses, etc., to allow us to analyze all of the code executed in a class. We assume that the instance variables of a class are encapsulated. That is, only the code in the class has access to the instance variables, and no other code can access the instance variables, including the code of any subclasses of the class being analyzed. We also assume that methods and procedures do not store any own data (i.e., data that is retained between invocations).

Aliasing makes our job more complex. In the example shown in Figure 5.2, suppose x is an instance variable, y is a local variable, and a is an argument variable (thus a refers to an object that is reachable from the environment), and we execute an assignment like $y := x[3]$ (i.e., x is an array and now y refers to one of its elements). Figure 5.2(a) shows the result of this assignment. Now suppose we call a method of y's object with a's object as an argument (e.g., $y.foo(a)$). A possible result (shown in Figure 5.2(b)) is that a's object is now reachable from y's object, and therefore, y's object might not be contained. But since y is an alias for a part of x, it is also the case that a's object might now be reachable from x's object, so that x's object also might not be contained, even though the invocation of did not mention x.

To determine when two variables may be aliases to objects that are reachable from one another, we adapt Larus and Hilfinger's *alias graph* construction algorithm[41]. Briefly, alias graph construction is a data flow computation that produces a conservative approximation of the aliases visible at any point in a program. The roots of an alias graph are variables names and the interior nodes represent storage areas. Edges are added (or deleted) by applying transformation rules associated with each statement or expression that causes changes in aliasing. (For example,

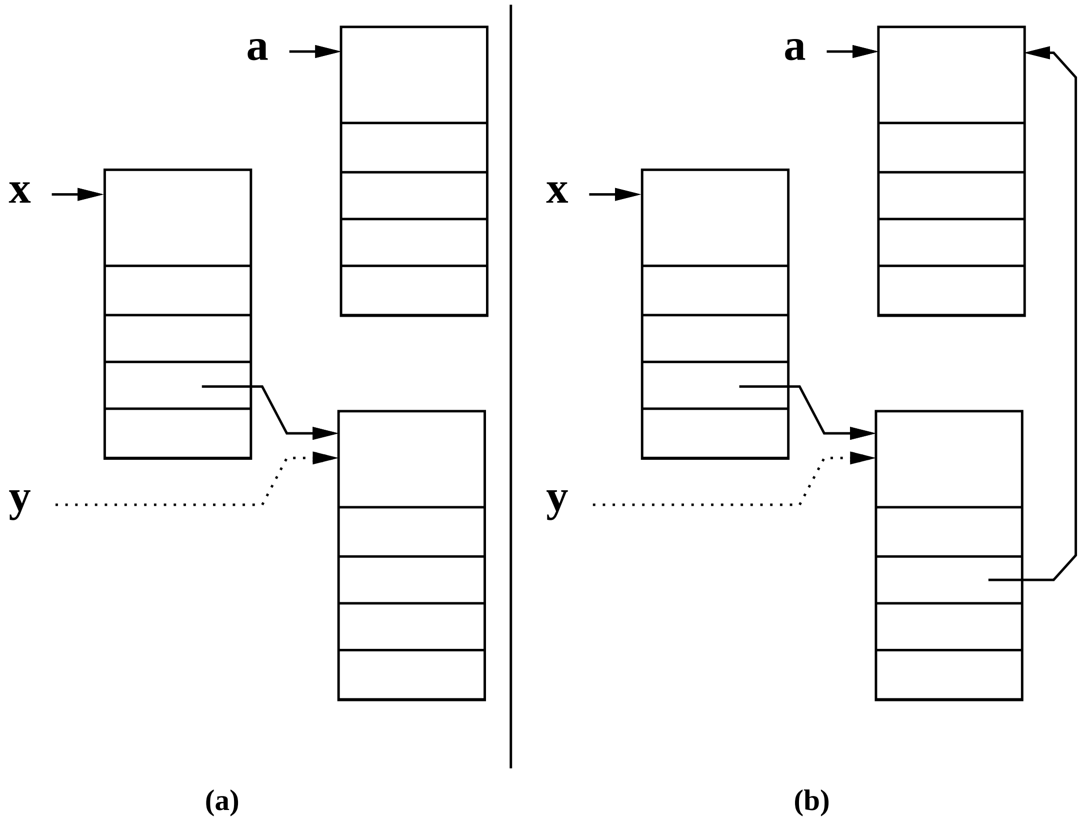


Figure 5.2: An example of how aliasing can happen: (a) after executing $y := x[3]$, (b) a possible outcome of $y.foo(a)$.

assignments or procedure calls.) Two (or more) variables are aliased if there is a path from each variable to a common node.

Adapting the alias graph construction algorithm for our purposes is straightforward. Alias graph construction applies only to variables that are references, so variables of built-in immutable types like `int` or `char` (i.e., those that are not represented as references) are not considered in the analysis. In the original algorithm, a function call $f(a_1, \ldots, a_n)$ with unknown aliasing effects causes all variables aliased to the argument variables to become aliases for the $\perp$ node that represents a global storage area. Since there are no global variables in Theta, a function call can only cause aliasing among local storage areas (i.e., those that are reachable from the arguments). We adapt the algorithm by creating a node to represent the result of the function call and all variables aliased to the arguments of the call become aliases to this node, instead of the $\perp$ node. We also provide special transformation rules for method calls of built-in types like `array` and `sequence`, since there is only one system-provided implementation of these types, and we know the effects of the methods of these types on aliasing. Method calls of the form $x.m(a_1, \ldots, a_n)$ are treated as function calls of the form $m(x, a_1, \ldots, a_n)$ with unknown aliasing effects except for method calls to *self*, e.g., *self*.$m(a_1, \ldots, a_n)$. Method calls to *self* are treated as function calls of the form $m(a_1, \ldots, a_n)$ and analyzed using interprocedural alias analysis on the code for m in the class. Calls to local routines defined in the class are also analyzed using interprocedural alias analysis. Note that we do not take advantage of the power of the algorithm to detect aliases in Lisp-like structures, since Theta does not have these data structures.

Using the alias graph construction algorithm, we can compute what aliases are caused by a particular creator or method. Clearly, any instance variable that becomes an alias for an object that an argument references is no longer contained. However, analyzing each creator or method individually is not enough, since one method or creator may cause aliasing among instance variables and another might alias one of these instance variables to an argument variable or return it as a result. For example, consider the class fragment in Figure 5.3. From inspection, we can see that method *combine* causes aliasing between instance variables x and y, but no other aliasing, so we conclude that x and y still refer to contained subobjects after calls to *combine*. Likewise, we see that method *expose_x* causes us to conclude that x does not refer to a contained subobject after a call to *expose_x*, but since y is never mentioned in the body of *expose_x*, we conclude that y still refers to a contained subobject after calls to *expose_x*. However, this is incorrect, since if we execute *combine* and then *expose_x*, y is aliased to x when we call *expose_x*, so we should conclude that y might not refer to a contained subobject after

```
example_impl = class example

    x_: array[employee]
    y_: employee
    :

    combine (i: int)
        y_ := x_[i]
        end combine

    expose_x () returns (array[employee])
        return (x_)
        end expose_x

    :
    end example_impl
```

Figure 5.3: Example class with aliasing from one method affecting another method.

calls to *expose_x*. Thus, in addition to analyzing each creator or method, we must also compute the transitive closure of aliasing exposed by each of the alias graphs.

We compute subobject containment in the following manner. For each creator or method f_i in the class, we analyze the body of f_i using the alias graph construction algorithm. The initial graph for each analysis is constructed as follows:

- (Reference) argument variables all point to the $\perp$ node.
- The pseudo-variable *self* points to the $\perp$ node.
- Each (reference) instance variable i points to its own node.
- Each local variable points to its own node.

The $\perp$ node represents the storage area of the environment and as far as we are concerned, all arguments are aliases for it. *Self* also points to the $\perp$ node, since the environment has an alias for it (otherwise, the object would be garbage). We use this initial graph to construct an alias graph A_i that describes the aliases that exist just before the creator or method returns. If there is a return statement, the returned result (assuming it is a reference) is represented as a node and is aliased to the $\perp$ node.

We summarize the aliasing information from each graph in an $N+1$ by $N+1$ matrix M, where N is the number of (reference) instance variables in the class. M represents aliasing of instance variables (1 to N) to each other and to the environment ($N+1$). Initially, the entries on the diagonal (i.e., entries $M[i,i]$, for $i = 1$ to $N+1$) are set to 1 (i.e., every variable is an alias to itself) and all other entries in the matrix are set to 0. After each alias graph A_i is constructed, if instance variable i is aliased to an argument variable in A_i (i.e., there is a path from i to the $\perp$ node), then $M[i, N+1]$ and $M[N+1, i]$ are set to 1. If instance variable i is aliased to instance variable j in A_i (i.e., there exists a path from i and a path from j to a common node), then $M[i,j]$ and $M[j,i]$ are set to 1.

After processing each alias graph, M^*, the transitive closure of M, is computed. (An algorithm for computing transitive closure can be found in [20].) The last column of M^* contains the information we need. If $M^*[i, N+1] = 0$, then instance variable i refers to a contained subobject, since it has not been aliased to the environment, and if $M^*[i, N+1] = 1$, then instance variable i does not refer to a contained subobject, since it has become aliased to the environment. If all of a class's instance variables refer to a contained subobject, we say the class is *completely contained.*

5.1.2 Using Containment Information

The compiler can take advantage of contained subobjects when producing registration versions of observers. It can avoid calling the registration versions of the contained subobject observers altogether, since it knows that there cannot be any outside mutations that can affect the state of the subobject. The basic algorithm for any registration version of observer O_r would become:

1. If O is an observer for some object y,
    (a) If $y \in R$, register y, if necessary, that is, if there is some mutator m that O depends on.
    (b) Add the uncontained subobjects of y to R.
2. For all calls p in O, if p is a method of a contained subobject, call the regular version of p. Otherwise call the registration version p_r, passing x, I, and R. (I.e. p is a stand-alone routine, a method of an uncontained subobject, or a method of a local object.)

In the best case scenario, when all of a set's elements are completely contained immutable objects, there is no registration information associated with indexes on that set at all. For set elements that are completely contained but mutable, O_r's would add registration tuples for the set elements x only, but otherwise would be the same as the original O's, thus none of the f-reachable objects of x would be registered.

More likely, only some of an object's subobjects are contained. For our `employee` objects implemented by the `employee_impl` class, `project` objects are obviously not contained within the `employee` objects since the *set_project* method will return them. However, the `income_info` objects are. Thus, we would only have to register the `employee` object if the *yearly_income* observer is used as an index function, and call regular methods on the `income_info` object and its subobjects, if it had any. For our example (Figure 5.1), we would no longer have registration tuples for `income-info` object γ.

5.1.3 GOM

As we saw in Chapter 4, the basic GOM system[34, 35] registers more objects than necessary to maintain function-based indexes. The GOM designers realized this, so they proposed something akin to our contained subobject analysis called "strict encapsulation" to try to avoid registering some of these objects. Strict encapsulation corresponds to complete containment in our analysis. Under the GOM definition, a class that is strictly encapsulated must meet the following properties:

1. *Get_* and *set_* instance variables methods of the type must not be named as methods in type interface.

2. All subobjects must be created during the initialization of the containing object.

3. No method returns references to subobjects.

This definition is incomplete because it does not say anything about the types of arguments to mutators. In particular, it does not rule out a mutator that takes an object a as an argument and calls a subobject y's method passing in a as an argument or calls a method of a passing in y as an argument. Either case might cause aliasing between objects reachable from an instance variable and objects reachable from the environment. Also, it is not clear what is meant by the third point; we are not sure if the GOM designers mean to include only the instance variables, or any reference to objects reachable from the instance variables. As we saw in our aliasing example, if we only look at the instance variables, a local variable can become an alias to objects reachable from an instance variable. When such a local variable is used in a return statement, the assumed containment property is violated.

To make use of this optimization in the GOM scheme, a database programmer examines the code of a class and determines whether it is "strictly encapsulated" and then tells the system explicitly that a high-level mutator only affects certain high-level observer results. Then the static analysis of this code moves the recomputation decision into the high-level mutator rather

```
set_rate_and_bonus (e: employee, new_rate, new_bonus: int)

    e.set_monthly_rate (new_rate)
    e.set_bonus (new_bonus)
    end set_rate_and_bonus
```

Figure 5.4: An example mutator with multiple parts.

than in the primitive *set_* instance variable methods that it calls. This is more like what our *depends_on* specification provides, but is done in an ad-hoc manner. Also note that the GOM designers only allow this "optimization" to be done on completely contained objects. We believe our scheme is safer; the type designers only have to worry about semantics and registration versions of observers that take advantage of containment are produced automatically.

5.2 Lazy updates

The basic update algorithm recomputes $I.f(x)$ whenever there is a mutation affecting an index. This may cause updates that are immediately overwritten. For example, consider the routine *set_rate_and_bonus* shown in Figure 5.4. The recomputation done during the call of *set_monthly_rate* is not necessary because of the recomputation done during the call of *set_bonus*.

One way to avoid unnecessary updates is to do the recomputation lazily. Both the GOM system[34, 35] and Bertino's scheme[8, 10] have lazy updates. In a lazy update scheme, an *I.data* entry would be *valid* if $key = I.f(x)$; otherwise it would be *invalid*. A mutator of a registered object would invalidate the current index data entry for x, but not recompute it. At some later time, the system would do the recomputation and revalidate the index data entry for x. It could wait to do the recomputation until the next time a query tries to use the index, but this will delay the query. Alternatively, it could do recomputations in the background.

The basic algorithm can be extended to do lazy updates in the following way. The entries of $I.data$ have an extra boolean field $valid$, which is true if $key = I.f(x)$ and false otherwise. A mutator m for an object y finding registration $< x, m, I >$ would behave as follows:

1. $valid$ is set to false in $I.data$ entry $<$ *old_key*, x, $valid >$.

2. remove the registration tuple from y.

3. m does the actual mutation to y.

Note that we still leave unspecified how we would find *old_key*. Some time later, the system would do the recomputation. For an $I.data$ entry $< key, x, false >$, the following is done:

1. Remove the data entry from $I.data$
2. Call $I.f_r(x)$ passing it x, I, $R = \{x\}$ as arguments.
3. Add $<$ *new_key*, x, *true* $>$ to $I.data$ where *new_key* is the result from step 2.

The system has some choices as to when to do the actual recomputation. It can do a recomputation of invalid index entries whenever there is a query that tries to use an index. Both the GOM system[35, 34] and Bertino's scheme[8, 10] do this. Since they are primarily interested in returning results of method calls quickly, it makes sense to delay the recomputations of invalidated results while calls are being made to other objects. However, in our scheme, queries are the only thing we are concerned with and waiting to recompute entries until a query wants to use the index would cause delays for queries trying to use indexes with invalidated entries. To try to prevent such delays, the system can do recomputations in the background by keeping a list of all invalidated entries. When the system recomputes the new index key, the entry is removed from the list.

The list of invalidated entries could also be used to determine if a particular index entry has already been invalidated, so that a subsequent mutator that affects it would not actually cause an access to $I.data$. The GOM system combines lazy updates with deregistration (described below) of the mutated object to avoid multiple accesses to an invalid $I.data$ entry caused by multiple mutations to the same object. Using a list of invalidated entries would be more general by avoiding multiple accesses to an invalid $I.data$ entry caused by subsequent mutations to *any* object that affects the index key of an element x rather than just multiple mutations to the same object.

5.3 Deregistration

When an object can no longer affect an index (because it is no longer f-reachable from a set element), we could remove its registration tuples for that index. We will call this process *deregistration*. Deregistering objects saves work by eliminating useless checks and recomputations, and saves space by removing obsolete registration tuples.

Deregistration can be accomplished by calling a deregistration version of a function or observer. The deregistration version O_d takes as extra arguments a set element x and an index I; it is also produced by the compiler and is basically the opposite of the registration version:

1. If O is a method for object y, remove all of y's registrations of the form $< m?, x, I >$.

2. Run the body of O. For all observer or function calls p in O, call registration version p_d, passing x and I as extra arguments.

3. Return the same result as O.

Now it is straightforward to see how we could deregister every object registered when $I.f_r(x)$ was invoked. We simply call $I.f_d(x)$. This is guaranteed to deregister all the objects that $I.f_r(x)$ registered because we require that $I.f$ have a deterministic implementation. Thus $I.f_d(x)$ will access the exact same objects that $I.f_r(x)$ did and remove the appropriate registration tuples. Note that it would be straightforward to make use of containment information to speed up this computation. We just make regular calls on the blue instance variables rather than the deregistration calls.

In Section 2.8, we deregistered an object when it was mutated, and in Section 2.7, we deregistered an object when it was deleted from an indexed set. In addition, we should call $I.f_d(x)$ when we delete a set element x, since the registration information for I in objects f-reachable from x is no longer needed and the subobjects of x are likely to be in the cache. We will call this scheme *partial deregistration*.

An alternative scheme is to do *full deregistration*. In addition to running $I.f_d$ when an object is deleted from the set, we also run it on the set element x when a registered object y is mutated. In this scheme, when a checking mutator m_c of y is executed, the following happens (assuming we are doing lazy updates):

1. Remove all tuples $< m, x?, I? >$ from y's registrations set. For each such tuple, call $I?.f_d(x)$ passing it $x?$ and $I?$ as extra arguments and invalidate $< key, x? >$ in $I?.data$.

2. Do the actual mutation to y.

The new registrations for the x's and I's will be done later, when the new key is computed for x using $I.f_r$. Note that it is important that $I.f_d$ be called before the mutation to y in case the mutation changes the f-reachability graph from x. If we want to do lazy deregistration as well (i.e., call $I.f_d$ later or in the background), we would have arrange for a copy of the old version of y to be used when calling $I.f_d$.

Full deregistration is attractive if key must be calculated explicitly to find the index data entry $< key, x >$ for invalidation. We would have to run $I.f$ anyway and could run $I.f_d$ instead. In addition, mutators no longer need to check for $x \in I.set$, since there will no longer be obsolete registrations for x. Thus, this scheme is also attractive if testing for set membership is inefficient.

If it is easy to find key and test for $x \in I.set$, partial deregistration is probably more efficient than full deregistration. For partial deregistration, the main problem is the unnecessary invalidations caused by obsolete registrations. However, if a mutation does not affect reachability, which is true quite often, there are no obsolete registration tuples. Also, if a mutation causes objects to become totally inaccessible, partial deregistration would not waste time removing registrations from objects that will be garbage collected anyway.

Chapter 6

Indexes in Thor

This research is being done in the context of Thor, an object-oriented database system[42, 43]. Thor is designed to support heterogeneous applications that concurrently share objects over a heterogeneous distributed system. Thor is still under development, so its system architecture changes from time to time, and study of alternative designs in several areas is an important part of our research. The research in this dissertation is being used as part of our on-going evaluation of the implementation. Part of this evaluation includes determining if design decisions made earlier continue to be reasonable as we add queries and indexes to the system.

In this chapter, we explore issues related to index use and maintenance in the Thor system architecture. We will not address the question of query processing, that is, how queries are described and processed into execution plans. We assume that query descriptions in Thor will be in some standard notation (e.g., OQL[18]), and many others have done research in generating (optimized) execution plans both for databases in general[33] and specifically for distributed databases[2, 19, 62]. Also, we will not address how to handle indexes for sets that have elements on more than one server (i.e., *distributed sets*). We leave this issue as an area for future research.

We begin by describing the current Thor system architecture to provide some background. We concentrate on those areas that have an impact on indexing or where indexing has an impact on a design decision. Some of the areas are still under investigation and in these places we describe some of the alternatives.

There are many issues that arise in using and maintaining indexes in the Thor system. These issues can be divided into two broad categories: those that arise from adding indexes to Thor and those that arise from the impact of the Thor system architecture on the performance of index use and maintenance. In Sections 6.2 and 6.3, we enumerate some of these issues and look at them in more detail.

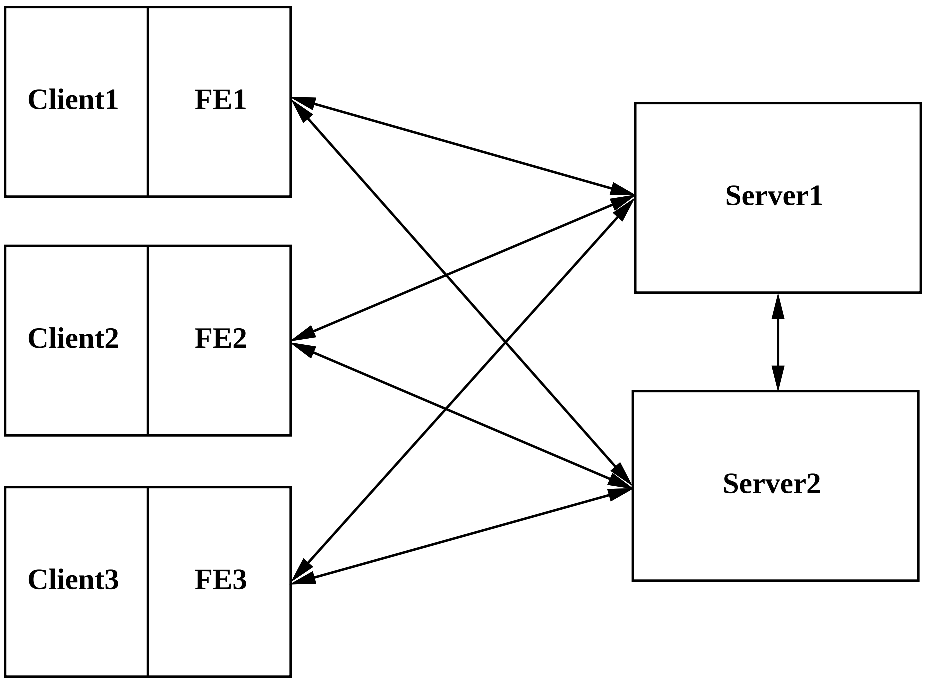


Figure 6.1: A representative Thor configuration. The arrows represent communication between nodes.

6.1 Thor System Architecture

Thor is an object-oriented database system. It supports sharing of objects by clients (applications) written in different languages. Clients of the database access objects by calling the methods of the objects within transactions. Thor transactions are serializable and atomic. That is, transactions in the system appear to run in some serial order and either all changes done by a transaction to persistent objects are reflected in the database upon transaction commit, or none of the changes are reflected in the database upon transaction abort. Persistence in Thor is based on reachability from designated persistent roots.

The Thor system is intended to run in a distributed environment. It is implemented using *clients* and *servers*. Servers provide persistent storage for objects. *Front-ends* (*FEs*) process client requests for operations to be run on persistent objects. An FE is created each time a client opens a *session* to the database; there is one FE for each client. When the session ends, the FE is destroyed. The servers and FEs are not generally co-located on the same physical node, though they may be; an FE is usually co-located with its client, though it may not be. We will assume that servers and FEs are on different nodes, and an FE and its client are on the same node for the rest of our discussion. Figure 6.1 shows a representative configuration.

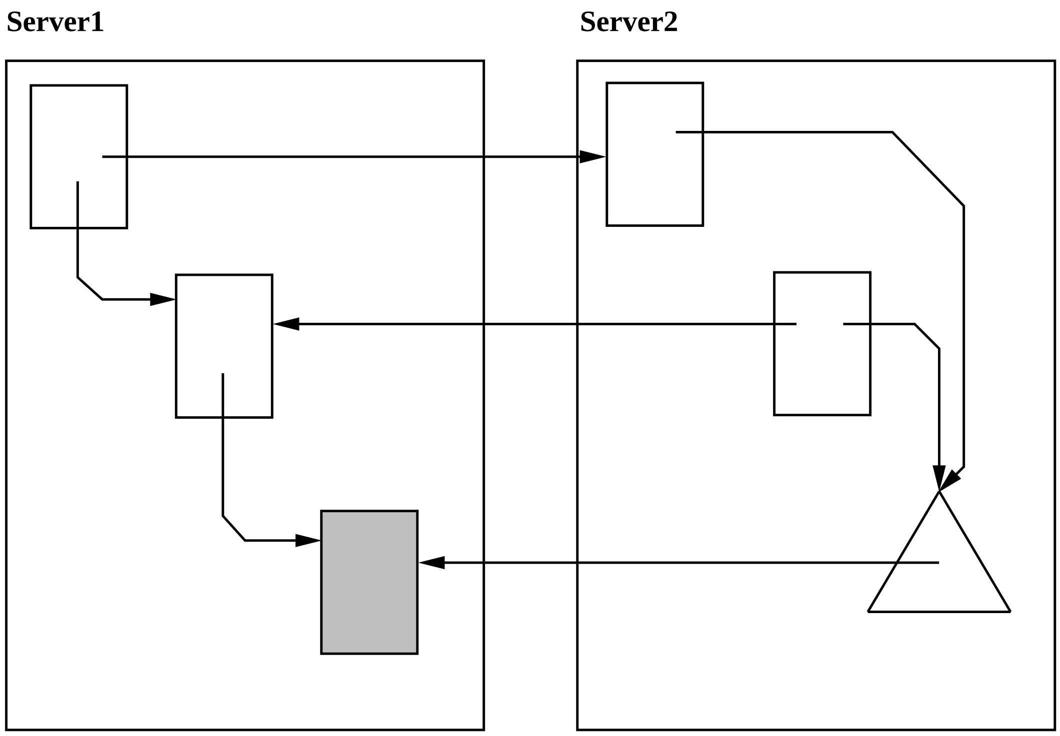


Figure 6.2: Objects at Thor servers. The triangle is a server-surrogate for the shaded object.

Every persistent object resides at one server. Persistent objects refer to each other using an "external reference" (called an *xref*). An xref consists of the identity of the server that (may) store the object and the object's "local" address at that server. An object that has moved leaves behind a surrogate containing its new xref. An example is shown in Figure 6.2. Server surrogates are eventually removed during server garbage collection after all objects that referred to the moved object by its old xref have been modified to contain its new xref[46]. Since objects can move from server to server, a persistent object also has a system-wide unique identifier (OID) assigned to it.

The servers store their persistent objects in very large disk segments. Tentatively, we have chosen a segment size of 64 kilobytes (64K), though this is a subject of current research[26]. The segments are fetched into the server cache on demand and are removed using an LRU policy. The server cache is assumed to be very large (perhaps up to 1 gigabyte in size); thus we use large segments for more efficient use of the disk. However, since multiple FEs can access objects at a server, the server cache space devoted to any given FE may still be fairly small.

Although persistent objects reside at the servers, client requests are carried out at the FE on copies of the objects. These copies are cached in primary memory at the FE. The FE cache is assumed to be much smaller than at a server. If the FE does not already have a copy of an object when a client wants to run one of its methods, the FE sends a fetch request to the server that stores the object and a copy of the object is sent over to the FE. The server returns a group of objects containing the requested object and some other, *prefetched* objects as well. The prefetched objects are "related" to the requested object. Currently, we are only prefetching the objects referred to (directly) by the requested object that are in the same segment as the requested object. This scheme might be extended to more levels of the object graph rooted at the requested object and to objects in other segments that are already in the cache.

Objects at the FE refer to each other by local virtual memory addresses. When a persistent object copy is brought to an FE, its references must be *swizzled* before being used. That is, its xrefs must be converted into virtual memory addresses. Currently, we are looking at two forms of swizzling[50]: *object swizzling*, where all of the xrefs in an object are swizzled when the object is accessed for the first time, and *pointer swizzling*, where an individual xref is swizzled when it is used for the first time to access the object it refers to. In object swizzling, a xref for an object that is not present in the FE's cache is swizzled to refer to an *empty* FE-surrogate, which contains the xref of the object it represents. Access to an empty FE-surrogate causes a fetch request for its object, and the FE-surrogate is *filled* with the requested object's virtual memory address. Subsequent accesses to the FE-surrogate are forwarded to its object. Filled surrogates are removed during FE garbage collection. Figure 6.3 shows an example of what happens when an FE fetches a object in the object swizzling scheme. When **FE1** fetches object A, copy α is sent to **FE1** along with prefetched copy π. After α and π have been swizzled, α has a reference to *pi*, and *pi* has has a reference to an FE-surrogate that contains the xref of object Γ. In the pointer swizzling scheme, an access to a reference that has not been swizzled causes the FE to look for the object in its cache and if it is present, the reference is converted to the virtual memory address of the object and marked as swizzled. If the object referred to is not present, a fetch request for the object is made and swizzling proceeds when the requested object is installed in the FE's cache. In either scheme, objects may be *shrunk* (that is, turned into empty FE-surrogates) to reclaim space in the FE cache[21].

Clients can create new objects. Initially, new objects live only at the FE. A new object can be made persistent only by mutating a persistent object to refer to the new object. As part of committing the surrounding transaction, a copy of the new object is sent to a server. If the transaction commits, the new object is stored at a server and is assigned an OID and an xref; the FE is informed of the new object's OID and xref so that it can update its copy. If the transaction aborts, the server discards the new object. Whether the transaction commits or

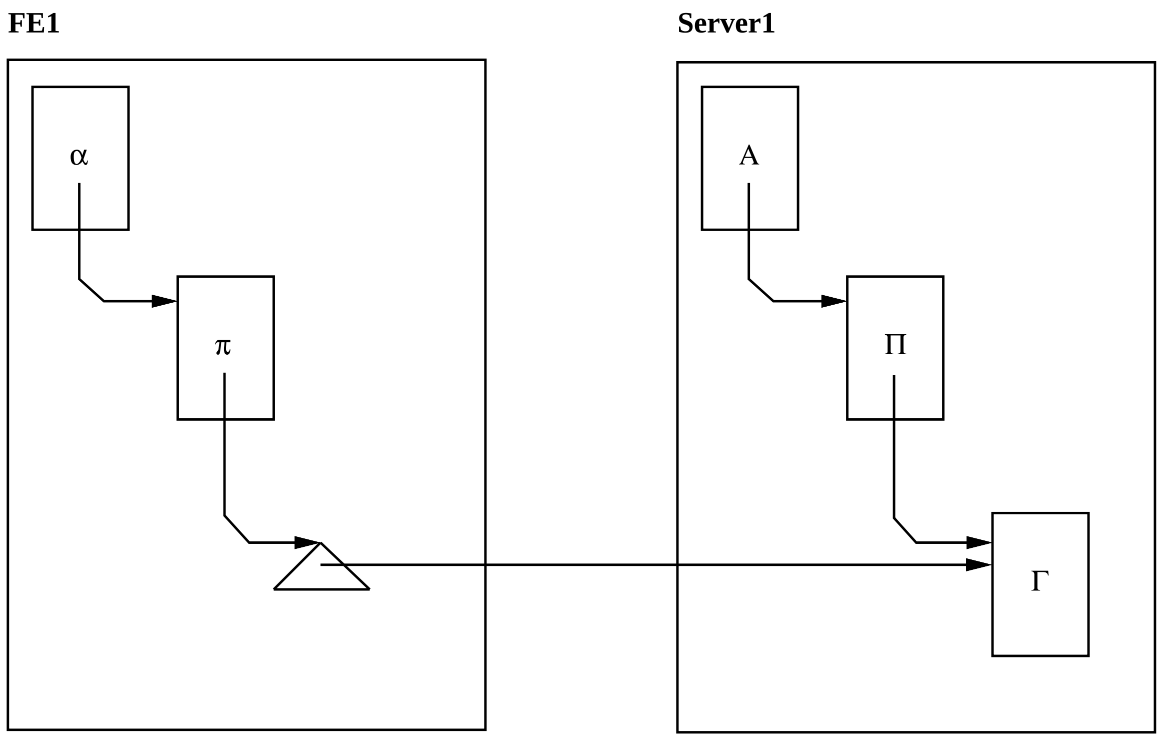


Figure 6.3: Objects at a Thor FE after object A is fetched by FE1. Uppercase Greek letters are the OIDs of the objects. Lowercase Greek letters denote a copy of the object with the corresponding uppercase OID. The triangle is an FE-surrogate for object Γ.

aborts, the new object still exists at the FE. Objects that are not made persistent by the time a client finishes a session with the database are destroyed when the FE is destroyed.

Concurrent access to objects in Thor is handled using an optimistic concurrency control mechanism[39]. In our current design[1], every persistent object has a version number (v#) that is advanced every time a transaction that modifies the object commits. An object's current v# is stored with the object and is copied to an FE when the object is fetched. The FE keeps track of what objects are read and modified during a transaction. When the client tries to commit a transaction, the FE sends the v#'s of the objects read by the transaction and the v#'s and new versions of the objects modified by the transaction to one of the servers storing these objects to be *validated.*

If multiple servers are involved, the initially contacted server acts as the coordinator of a two-phase commit protocol[27], sending the participants the v#'s of all the objects used by the transaction; otherwise committing can be done locally. Validation is done by checking the v#'s of the objects used in the transaction against the current v#'s of the objects. If any do not match, the transaction must abort. This is because the transaction has read a version of an object that "conflicts" with writes done by committed transactions. (Note that this is a backwards validation technique favoring writers over readers.) Since Thor allows multiple servers to commit transactions concurrently, the validation step is more complicated than just described, but the notion of when of two transactions have done conflicting reads and writes is the same. Details of the full protocol can be found in [1]. We will discuss our work only in the context of a single-server system, since this case covers the basic concurrency control mechanism and we believe validation for index operations can be added to the full protocol in a straightforward manner.

Assuming a single-server system, if all v#'s match, the server selects a v# larger than any of the current ones and this is written into objects when their new versions are stored stably. After this happens, the transaction is said to have *committed.* When a transaction commits, the FE is informed of the new v#'s in the commit reply; it then updates the v#'s of the modified objects it has in its cache. When a transaction aborts, the server informs the FE about the objects that had stale v#'s, and the FE shrinks them. The FE also restores its copies of other modified objects back to their state before the transaction began.

This optimistic scheme can cause extra aborts not present in locking schemes, since FEs may have copies that are stale. To prevent this from happening, the servers in Thor keep track of which FEs have copies of which objects and send *invalidation* messages to FEs holding copies of objects that have been modified. Invalidation is done by shrinking the invalid object.

If an object has already been read by the current transaction, the transaction must be aborted. Thus invalidation also has the added benefit of causing early aborts of transactions that cannot possibly commit, avoiding the full commit protocol for them.

Thor uses a primary copy replication scheme[51] to make the servers highly available. We are not concerned with the exact details of the scheme except to note that modifications are applied to the server cache's copies at transaction commit, and they are flushed to the physical storage of a server in the background by an "apply" process as in the Harp file system[44].

6.2 Adding Indexes to Thor

The Thor system architecture as described above is inadequate for using indexes to answer queries. Some questions that arise include the following:

1. How do we integrate index use with two-level caching? In particular, are indexes (and sets) fetched to FEs like regular objects?

2. How do we integrate index maintenance with two-level caching? In particular, are changes to indexes committed like changes to regular objects?

3. How are concurrent use and maintenance of sets and indexes handled?

4. Can index creation on a very large set be assured in a system using optimistic concurrency control?

Thor's two-level caching allows us to make choices about where these computations take place. Currently, there is a strict division between the servers and the FEs. All user computation happens at the FE, and servers only store their objects and perform system functions like the transaction commit protocol. Our general philosophy in integrating indexes into Thor is to minimize the changes we would have to make to the current system, though we will discuss some of the other options. In particular, we will choose to perform user computations at the FE in all cases, although we will point out cases where it might be worthwhile to make a different decision.

Query computations using an index are different than method computations, because we are using a system data structure instead of a user data structure. Section 6.2.1 focuses on how indexes (and sets) behave differently from regular objects in Thor and describes how indexes are used to answer queries.

Modifications to registered objects cause two additional computations: a computation to recompute the new key and a computation to update the index entry. Key computations involve user computations, so we choose to perform them at the FE. Updates only involve the index, so we choose to have them performed at the server. Section 6.2.2 discusses the issues and reasoning behind these choices.

Concurrent use and maintenance of indexes should not cause erroneous behavior or unnecessary aborts. It is well-known that ordinary concurrency control mechanisms based on locking must be extended to handle indexes[25]. To our knowledge, this has not been done for optimistic concurrency control mechanisms. We have developed a new form of optimistic concurrency control, *predicate validation*, that is suitable for allowing concurrent use and maintenance of indexes in systems using optimistic concurrency control. Like its namesake, predicate locking[25], in predicate validation, index operations are converted to read and write predicates that describe the objects of interest. Unlike predicate locking, predicate validation does conflict resolution at transaction commit. Consistent with the backward validation scheme of Thor, it aborts a transaction if the results of its read predicates have been invalidated and if a transaction commits, uses its write predicates to invalidate conflicting operations of active transactions. Predicate validation is described in Section 6.2.3.

Creating an index on a very large set will take a long time. We have to access every element of the set in order to build an index. The likelihood that another transaction will mutate one of the set elements while the index is being built increases as a set get larger. For very large sets, we may never be able to commit the index-creating transaction. Section 6.2.4 describes a way to build indexes on very large sets incrementally. In this scheme, we periodically commit the computations of $I.f_r(x)$ even though the index has not been completed and any index updates that these registrations cause are handled specially.

6.2.1 Queries

Queries can be computed with or without using an index. We consider how to compute queries using indexes first. In our discussion, we assume that an index is stored at the same server as the indexed set and that registration information for an object is stored at the same server as the object.

Efficient queries will be more important in Thor than in more conventional systems because of its two-level architecture. In addition to providing associative access for clients, queries reduce the amount of data that needs to be sent to the FE by identifying the objects of interest. To make queries efficient in Thor, set and index objects cannot be treated like regular objects. The elements of set objects should not be prefetched when the set object is fetched. Similarly, if an index is fetched, the set elements it references should not be prefetched. We envision that when a set object is fetched, what is sent to the FE is a description of the set containing, for example, the number of elements it has and a list of what indexes it has; this information is consulted during query processing.

Even without prefetching the set elements, copying an index to the FE still may be inefficient, since indexes for very large sets are themselves large objects. In addition, in the object swizzling case, references to set elements in the index causes FE-surrogates to be created for them, even if they are not fetched to the FE, and would increase the space cost of copying the index to the FE. On the other hand, sometimes we might like to bring the entire index over to the FE. For example, for a series of queries using the same index, we can trade off the space the index takes up with the time consumed by multiple network accesses. Thus, we may want to answer queries using indexes at either the server or the FE. We will call index usage at the server *remote* and index usage at the FE *local.* The question of how to determine which mode of usage would be the most advantageous for a particular application is part of query processing, and it is not addressed in this dissertation. However, we believe that the case for copying an index to the FE will be uncommon, and we will implement only remote indexes for Thor initially.

Using an index remotely requires that we enrich the interface between the servers and the FEs. The server interface must be extended with operations that allow an FE to request a computation using an index. These computations could involve just one use of an index (a *match request*), or they could be entire query execution plans, for example, if all of the indexes used in answering a query are stored at the same server. The server interface should be able to handle either case. We must also add computational capability at the server to run these index operations. We justify adding this complexity to the server by noting that index use is a system activity and involves no user code. Also, as we will see later, special processing must be done by the server at commit time to handle index use (either remotely or locally).

The results of remote index use can be kept at the server or sent back to the FE. A client is likely to want to invoke methods on the elements of the result set, since generally computing a query result set is only the first step in a computation. We might want to keep the result set at the server if it is still fairly large and subsequent computations would reduce the number of objects of real interest to the client computation. However, this would require running user code at the server. Thus, following our general philosophy, we choose to send result sets and their elements back to the FE in reply to remote index use.

A query that cannot be answered using an index must be computed by iteration and function application. Under regular object processing, all of the set's elements would need to be copied to the FE. If only a few objects of a large set actually match, this represents a particularly inefficient use of network bandwidth and FE cache space. The non-matching objects need not have been sent over in the first place, and they take up space in the FE cache even though they will not be used, causing more frequent garbage collections. We might like to have the server run this computation instead, but again, this requires user computation at the server, so for now, we will compute queries that do not use indexes at the FEs in the straightforward manner.

6.2.2 Updates

Thor's two-level cache structure gives us design choices of where computations related to index maintenance take place. There are two computations: (re)computing keys and updating indexes. We consider each computation in turn.

(Re)computing new keys requires that object methods be invoked. These are user computations, so in accordance with our general philosophy, we choose to have these computations done at the FE. In addition, transaction semantics also requires that any modifications done by a transaction be observable by later computations done by the same transaction. Indexes are no exception; thus the system must filter query results for a transaction against any modifications done by the transaction that affect the result. For example, this means that if a set element x that is registered for index I is mutated by a transaction T, the result of a subsequent query by T on the set using I must reflect the current state of x. For *insert* and *update* operations, this means that the index key $(I.f(x))$ must be (re)computed if a subsequent query is done using the modified index. If we do not keep precomputed results, it is also necessary for the *delete* operation to compute the key of the deleted set element and for the *update* operation to compute the old key for the modified set element. (If a query is made using I before a (re)computation is completed, the system will have to delay returning the result until the (re)computation is completed and the result has been filtered.)

When an index is updated, we could copy it to the FE to do the modifications, or we can have the work done at the server. We choose to this work at the server; the FE notifies the server of the updates to the index at transaction commit. The semantics of transactions requires us to delay doing modifications to the persistent copies of an object until a transaction has committed, so that we cannot apply index updates to the persistent index object until the transaction actually commits. There is also a question of exactly what is the physical representation of an index and how it is related to updates. We can imagine that what an FE sends back at commit time for an index that has been updated is not a physical description of the changes (as is done for regular objects), but rather a logical description. We choose to have the server install the updates to the index in the background after the transaction commits as part of the regular background apply process.

6.2.3 Predicate Validation

Indexes are used to avoid reading every element in a set in order to determine if the element satisfies some predicate. There are two problems in trying to use and maintain indexes concurrently with regular objects in Thor. First, not actually reading a set element leads to problems, since regular concurrency control is done at the object level. If we do not read a set element, we cannot detect if it has been mutated by another transaction. And if we do not read the set structure either, we cannot detect if something has been inserted or deleted. Even if we could know a set element was mutated, it may not matter to the query.

In particular, the problem is one of *phantoms*, first identified by Eswaran, *et al.*[25]. Conceptually, a query not only reads the elements in its result set but also all of the "elements" that are not in its result set. A phantom is a non-existent result set element. It is important to know what result set elements do not exist at the time of a query computation because if one is created and committed before the query commits, and it should have been included in the query result, the query should abort. For example, if a transaction does the example query, "Select the employees from department E with a project manager whose yearly income is greater than \$60,000," there should be a conflict if another transaction commits a modification to a set element x such that $project\_manager\_income(x)$ now returns \$65000, rather than \$55000, since x should now be included the result set of the query. However, since x was not fetched, x's v# was not read when the query was computed, and there is no information for the regular concurrency control mechanism to generate a conflict. Similarly, if a transaction inserts a new element x into E where $project\_manager\_income(x) = \$65000$ and commits, our example query should be aborted.

The second problem is that treating the index structure as a regular object may lead to undesirable behavior. Conflicts at the physical level may cause unneeded aborts. For example, a transaction that inserts a new `employee` object x into E where $project\_manager\_income(x) = 50000$ should not conflict with a transaction that does our example query, since the insertion of x does not affect the result of the query. This should be the case even if the index entry for x is written in one of the index nodes read to answer the query. If index nodes are treated as regular objects, this would cause a v# conflict and the transaction doing the query would have to abort. Thus, indexes need to support higher concurrency than regular objects.

We can solve both problems by doing conflict resolution at the logical operation level. In our case, the logical entities are predicates. The basic idea is that index operations are modeled as predicates that describe the elements of the set that are of interest. Queries are predicates that describe the objects that have been read (and not been read). Index maintenance operations are predicates that describe objects that have been modified. A conflict occurs whenever the commit an index maintenance operation would change the result computed for a query that has not yet committed. This scheme is consistent with the Thor optimistic concurrency control: instead of preventing predicate conflicts by preventing such accesses through locking, it identifies predicate conflicts after they have happened, and prevents transactions that have invalidated results from committing. We call this scheme *predicate validation* because it is an optimistic version of Eswaran, *et al.*'s predicate locking technique[25].

We begin our explanation of predicate validation for indexes in Thor by describing the predicates used in indexing. Then we present two versions of the protocol. The first version is for Thor when only remote match requests are allowed. It is very simple and applicable to conventional systems as well as Thor. Although we believe that remote use will be the dominant mode of use (and probably the only one we will implement), for completeness, we also present a second version of predicate validation that extends the protocol to allow local index use.

Index Predicates. There is only one kind of read predicate in our indexing scheme, which we will call a *match request*. Match requests are tuples of the form $< I, \mathit{low\_key}, \mathit{hi\_key} >$ where either or both of the keys may be infinity. They correspond to a single use of an index and represent the objects of $I.set$ that have keys generated by $I.f$ in the range *low_key* to *high_key*. (Complex queries are converted to multiple match requests during query processing.)

There are three kinds of write predicates, *insert*, *delete*, and *update*, corresponding to the three index maintenance operations. Insert write predicates are tuples of the form $< I$, insert, $\mathit{key}, x >$ and delete write predicates are tuples of the form $< I$, delete, $\mathit{key}, x >$, where x is the set element that was inserted or deleted, I is the index that needs to be updated, and *key* is the result of $I.f(x)$. Update write predicates are tuples of the form $< I$, update, $\mathit{old\_key}, \mathit{new\_key}, x >$ where *old_key* is the key paired with x before the mutation and *new_key* is the key to be paired with x after the mutation.

The read and write predicates conflict in the following straightforward ways:

- An insert write predicate $< I$, insert, $\mathit{key}, x >$ or a delete write predicate $< I$, delete, $\mathit{key}, x >$ conflicts with a match request $< I, \mathit{low\_key}, \mathit{hi\_key} >$ if $\mathit{low\_key} \leq \mathit{key} \leq \mathit{hi\_key}$.

  This is what one would expect. An insert or a delete conflicts with a match request whenever the key of its object falls in the range of the match request.

- An update write predicate $< I$, update, *old_key*, *new_key*, $x >$ conflicts with a match request $< I$, *low_key*, *hi_key* $>$ if $((old\_key < low\_key$ or $hi\_key < old\_key)$ and $(low\_key \leq key \leq hi\_key))$ or $((low\_key \leq old\_key \leq hi\_key)$ and $(new\_key < low\_key$ or $hi\_key < new\_key))$

  Updates are a little trickier. An update conflicts with a match request whenever *old_key* is in the range of the match request and *new_key* is not, or vice versa. This is because if both keys are in the match request range or both keys are out of the match request range, the update has no effect on x's status with respect to the match request result.

We assume for now that we recompute the old key whenever there is an index update. This means that two transactions that make modifications that affect an index entry for x will always conflict during regular object validation, since both transactions will read all objects f-reachable from x before any mutations. We will discuss the effect of using precomputed results as old keys at the end of this section.

Simple Version. The goal of predicate validation is to detect when a transaction that has used an index for a match request should abort because another transaction has committed a conflicting update. To do this, the system must keep track of the order in which index operations are done. The assumption that match requests are being answered only at the server leads to a very simple protocol, because the server becomes the centralized point that serializes the match requests and updates for an index.

Simple predicate validation in a single-server Thor system works as follows. The server keeps a `use-list` per transaction that keeps track of all the match requests that have been made by that transaction. The entries are of the form $< m\_req, valid >$, where m_req is the match request that was answered and $valid = true$ if the match request result has not been invalidated by some committed transaction. (That is, no committed transaction has done a modification that caused a conflicting update to the index used by m_req.) The server adds an entry to a `use-list` whenever it does a match request for a transaction; at that point, the entry is marked as valid.

The FE keeps a modification list, `t-mod-list`, for its transaction. The entries in `t-mod-list` are write predicates $< w\_pred >$ that represent index updates (to be) done by the transaction. Entries are added to `t-mod-list` when the transaction does a modification to a registered object that causes an index update. The list is used by the FE to filter queries involving that index. For example, if there is a `t-mod-list` entry with update w_pred, then any query using $w\_pred.I$[1] must be filtered.

At commit time, in addition to the v#'s of read and modified regular objects and the new versions of modified regular objects, the FE sends the transaction's `t-mod-list` to the server. Regular object validation is done first, and if any have invalid v#'s, the transaction aborts. Otherwise, the following additional steps are taken:

1. For each entry $< m\_req, valid >$ in the transaction's `use-list`, check if $valid = true$. If all entries are valid, the transaction may commit. If any entries are invalid (i.e., $valid = false$, then the transaction must abort.

2. If the transaction does commit, for each entry $< w\_pred >$ in its `t-mod-list`, find all `use-list` entries $< m\_req, true >$ for the same index, i.e., $w\_pred.I = m\_req.I$. For each such `use-list` entry, if w_pred conflicts with m_req, set $valid = false$.

Then the new versions of the regular objects and the transaction's `t-mod-list` are written to the log to be applied later and the transaction's `use-list` and `t-mod-list` are discarded at the server. The FE discards its `t-mod-list` when it learns of the transaction's outcome.

Note that we can cause early aborts of transactions that have invalid match results. Whenever an entry in a transaction's `use-list` is invalidated, we can send an invalidation message the FE managing that transaction, causing the FE to abort the transaction. (Since an FE only manages one transaction at a time, there is a one-to-one mapping between transactions and FEs).

Our predicate validation mechanism must satisfy the following condition to be correct:

> Any use of an index I by a transaction T must reflect all modifications done to I by transactions that commit before T.

Since we are assuming a single-server system, in the backwards validation scheme of Thor, transactions commit in the order they request to commit. We assume that when a transaction commits, it does so in a critical section in which all of the indexes used or modified by the transaction and their associated information are locked so that other transactions cannot use them for match requests or to commit.

To see that the simple version of predicate validation satisfies our correctness condition, consider that when a transaction T does a match request, the current value of an index I reflects the modifications of any transactions that have committed before that point. Thus,

[1] The $w\_pred.I$ notation means the index I of the w_pred tuple. Likewise for $m\_req.I$.

we only need to be concerned with transactions that commit after that point and before T commits. Such a transaction T' will commit while T's match request is in the server's `use-list`. Therefore, T''s modifications will be compared to T's match requests, and if any conflict with T's match requests, those match requests will be marked as invalid. The modifications that do not conflict with T's match requests do not matter, because they do not affect the match results. When T attempts to commit, it will be unable to do so if any of its `use-list` entries have been invalidated. Thus, if T commits, its match requests are certain to have reflected the modifications of all transactions that committed before it, because none of the modifications made after its match requests conflicted with them.

Extended Version. Extending predicate validation to handle local index use makes the protocol more complex. The server can no longer serialize uses and updates to an index because it does not see the match requests answered at FEs. In particular, we cannot mark local match requests as invalid when transactions commit. We could require that the FE inform the server of any match requests it answers. Then the server could maintain the `use-lists` as in the simple version and the protocol would be the same. However, this scheme has very little advantage over just having match requests answered at the server.

Instead, we delay marking match requests: we have the server keep track of the committed updates done to an index since a copy of the index was given out, and we compare them with match requests made by committing transactions during the validation step of the commit protocol. However, since different FEs may have copies that reflect differing numbers of updates, we introduce a version number $V\#$ for indexes to distinguish the copies used to answer match requests. The server gives each committing transaction a timestamp that reflects its commit order. (I.e., for all transactions, $T1$ and $T2$, if $T1$ is serialized before $T2$, then $T2$'s timestamp is later then $T1$'s timestamp.) When a transaction succeeds in committing, its timestamp is stored as the V# of any indexes it updates. We also keep track of the V# of an index used to answer a match request. Since V#'s are the timestamps of committed transactions, a match request answered using an index with a V# of v can be invalidated only by a modification made by a transaction whose timestamp is later than v.

The extended version of predicate validation for a single-server Thor system works in the following way. FEs continue to keep their own modification list, `t-mod-list`. As in the simple version, the entries in `t-mod-list` are write predicates $< w\_pred >$ that represent index updates (to be) done by the transaction. Entries are added to `t-mod-list` when the transaction does a modification to a registered object that causes an index update, and the list is used to filter match requests.

FEs also keep an `f-use-list` containing entries for every match request made (both remotely and locally). The entries in this `f-use-list` are of the form $< m\_req, v >$ where m_req is the match request that was made and v is the V# of $m\_req.I$ when the match request result was computed. When an FE does a remote match request, the server computes the result and sends it back to the FE along with the V# of $m\_req.I$ at the time the result was computed. This V# is the v associated with the match request in the transaction's `f-use-list`. When an FE does a local match request, if it does not have a copy of the index, the index is fetched. The server sends a copy of the index that contains its current V#. Any match requests answered using this copy are associated with this V# in the transaction's `f-use-list`.

Servers keep a modification list, `mod-list`, that keeps track of the modifications that have been done. An entry in `mod-list` is a tuple $< w\_pred, v >$ where w_pred is the write predicate corresponding to the index update and v is the V# of $w\_pred.I$ after the update. Entries are added to `mod-list` at transaction commit as explained below.

At commit time, in addition to the v#'s of read and modified regular objects and the new versions of modified regular objects, the FE sends the transaction's `f-use-list` and `t-mod-list` to the server. As in the simple version, the server does regular object validation first, and if there are any invalid v#'s, the transaction aborts. Otherwise, the following additional steps are taken:

1. For each entry $< m\_req, v >$ in the transaction's `f-use-list`, v is compared against $m\_req.I$'s current V#. If all of the V#'s are the same, the transaction may commit.

   If any V# is not the same, for each such entry $< m\_req, v_m >$ in the transaction's `f-use-list`, find all the entries $< w\_req, v_w >$ in `mod-list` where $w\_pred.I = m\_req.I$ and $v_w > v_m$. If any of these `mod-list` entries have write predicates w_pred that conflict with match request m_req, the transaction aborts. If there are no conflicts, the transaction may commit.

2. If the transaction does commit, all of the indexes appearing in its `t-mod-list` have their V#'s set to the transaction's timestamp as explained above, and the entries in `t-mod-list` are added to `mod-list` with the transaction's timestamp as well.

Then the new versions of the regular objects and the transaction's `t-mod-list` are written to the log to be applied later as in the simple version. The server discards the `f-mod-list`, and the FE discards its `f-mod-list` and `t-mod-list` when it is informed of the outcome of the transaction.

The `mod-list` needs to be garbage collected periodically, since otherwise it will grow without bound. We note that we only compare entries in `mod-list` that have v's greater than those of committing match requests. Thus, if we can determine a lower bound LB on the v's of these match requests, we can safely discard any modifications in `mod-list` whose v's are less than or equal to LB. We can determine LB in the following way. The server maintains a `V#-table` (per index) that maps FEs to V#'s. Whenever it sends V# information for a match request using index I to an FE, it updates the entry for the FE in I's `V#-table` with the minimum of the current value for the FE in `V#-table` and the V# being sent over. Thus, when a remote match request using I it done, I's `V#-table` is updated using the current V# of I, and when a copy of I is sent to an FE, the V# of the copy is used in updating I's `V#-table`. A `V#-table` also keeps track of whether or not an FE has a copy of the index. When a transaction commits (or aborts), if the FE does not have a copy of I, the FE's value in I's `V#-table` is advanced to ∞. Entries for an FE are removed from `V#-tables` when it is destroyed. The lower bound of the entries in all `V#-tables` serves as the LB for trimming `mod-list`.

If an FE keeps an index copy for a long time, it may become very out-of-date. The FE can bring its copy up-to-date by asking the server to send any updates that have been made to the index since the FE received the copy along with the current V# and apply the updates to the copy. Then the FE can advance the V# of its copy to the sent V#. Or the FE can invalidate its copy and request a new copy. Also, note that if an FE keeps an index copy for a long time, it eventually prevents the server from garbage collecting `mod-list` entries, since the LB will not advance past the V# of the copy. The server can be pro-active in this case and ask FEs to invalidate their copies of an index.

Our correctness condition for the extended version of predicate validation is the same as for the simple version. We must ensure that when a transaction T commits, its match results reflect the modifications done by all transactions that committed before it. When T does a match request, the v associated with it identifies all transactions whose modifications are reflected in the match result, namely all those with timestamps less than or equal to v, so we only need to worry about modifications made by transactions with timestamps later than v. The server checks for such modifications by examining `mod-list`. If we assume for the moment that information is never thrown away, it is clear that any such modification will be in `mod-list`, since all modifications are added to `mod-list`. We guarantee that we never throw away needed information, because we only discard modifications after their v's are less than the v's of the match requests of active transactions; it is safe to discard those modifications because they are already reflected in the match results computed by active transactions.

Using precomputed results for old keys. When we use precomputed results as old keys, we do not recompute the old key when we need to update an index. (Of course, we need a way of finding the old key when given an element of the indexed set. For example, we might keep a table that maps every element in the set to its key.) Using precomputed keys has an interesting consequence for predicate validation.

Conflicting updates will still be detected by the regular object concurrency control mechanism when we use precomputed results as old keys. However, some updates that appear to conflict (and do conflict when we recompute old keys) actually do not, and by using precomputed keys, we can allow the later transactions to commit. For example, suppose transaction $T1$ mutates the f-reachability graph from x at an object y, while transaction $T2$ mutates the f-reachability graph for x at an object z which is above y in the original f-reachability graph for x. Suppose $T1$ commits first; it removes the old entry for x in index I, enters a new entry for x in I with the new key, and enters the new key in the precomputed result table for x. Now suppose $T2$ tries to commit. The recomputation done by $T2$ does not conflict with $T1$ because it never reads y. Also, the key for x is correct with respect to the modification being committed by $T2$, since it does not depend on the modification made by $T1$. So we should allow $T2$ to commit. The only thing we need to ensure is that when $T2$ reads the precomputed result table, it gets the new key computed by $T1$, so that it can remove the index entry for x that $T1$ entered. ($T2$ must remove the entry entered by $T1$, since otherwise the index invariant that there is only one and only one entry for each set element will be violated.) We can do this by having the apply process read the table right before it applies the index update and then immediately write the new key into the table afterwards. Since the apply process installs modifications in the order that transactions commit, when it does $T2$'s modification, the table entry for x will have the new key computed by $T1$.

6.2.4 Incremental Index Creation

It should be simple to create an index: iterate through the set, compute $I.f_r(x)$ for each element x, and create the index data part. However, creating an index on a very large set is likely to take a long time. If we run index creation as a single transaction using the regular Thor concurrency control, the index creating transaction will read every set element and every object f-reachable from the set elements. As the length of time that index creation takes increases, the likelihood increases that there will be a transaction that commits a modification on one of the accessed objects, causing the index creating transaction to abort.

Thus for large sets, we would like to build indexes incrementally. As with regular index maintenance, there are two kinds of events we need to detect: changes in set membership and mutations to registered objects. These two situations are handled by our indexing scheme, and we can take advantage of code already in place to do incremental index creation. We will assume only one index creation is being done at any given time to a particular set, though we allow creation processes to run concurrently on different sets. Perhaps this constraint is enforced by "locking" the set for index creation.

Algorithm. To support incremental index creation[2], indexes have two states, *open* and *closed*. An open index is a "normal" index. A closed index is just a skeleton of an index; its data part is initially empty. It behaves like an open one except that it cannot be used to answer queries (this is done by not listing it in the set's index list). In particular, it can be named in registration tuples. Updates to closed indexes happen when there are set operations on the indexed set or when objects registered for the closed index are mutated in a way that matters. The server does special handling of these updates as we will see below.

The process for incrementally creating an index for a set S using index function $I.f$ is as follows.

1. Run a transaction to create an empty, closed index I on S.

2. Run a transaction, that for some number N of set elements x, computes $I.f_r(x)$, and insert a $< key, x >$ pair, where $key = I.f_r(x)$, and into $I.data$. Repeat until keys for all of the set elements have been computed. We will call these registration transactions.

3. Run a transaction to fix up the index (explained below), install the index data part, and open the index (i.e., the index is added to S index list).

Note that conflicts between a transaction of the index creating process and a modification by another transaction will be resolved in favor of the transaction that tries to commit first. However, since the transactions of the index creating process are short, we do not expect these conflicts to happen often. If a transaction of the index creating process does abort, the transaction is run again.

N should be chosen to make it likely that registration transactions will not encounter any conflicts. $N = 1$ certainly meets this criteria, but is likely to be too inefficient, though we note that registration transactions are likely to involve only one server on the assumption that, most of the time, an f-reachable object will be stored at the same server as the set element that it can

[2]This scheme is similar in spirit to the ARIES system's incremental index creation scheme[49], but since ARIES is a relational database, the details are completely different.

affect. Thus most of the time, the non-distributed case of the commit protocol will be run. The groups of N objects could be chosen so that all of the objects accessed are in the same segment. We could "lock out" access to a segment of element objects to guarantee that the registration transaction will not encounter conflicts. This might cause other transactions to wait, but the waiting would not be as long as if we locked the segment during the entire creation process.

Concurrent modifications to the set or mutations to registered objects for an closed index are handled like operations that affect normal indexes. Even though the index is closed, since we committed the transaction to add the closed index, the system knows about the index. Thus, updates to a closed index arrive at the server in the transaction's `t-mod-list` as described in Section 6.2.3. When the apply process encounters an update to a closed index, it adds the entry to a special `i-mod-list` for the index. Fixing up the newly-created index before opening it is straightforward; we just apply the updates in the `i-mod-list` for the index.

Thor Implementation. Various parts of the index creating process can be run at either the FE or the server. As with index maintenance, the keys $(I.f_r(x))$ are computed at the FE. Although this scheme will require all of the set elements to be copied to the FE, we note that fetching can be overlapped with computation (e.g., by streaming the elements to the FE) and that once a registration transaction computing the key for a set element x commits, the FE can shrink x. Thus the index creating process only sees the initial network delay for the first elements to arrive at the FE, and the FE does not have to store many objects even if the set is large as long as we set N to an appropriate value.

We can minimize disk costs at the server by sending the set elements to the FE in clustered order where possible. This maximizes the numbers of elements that are brought into the server cache with each disk read.

The creation of the index data part is done at the server. The server already knows how to apply index modifications for regular index maintenance, so we format the results of the registration transactions as insert entries for the new index, but we need to keep them separate from the `t-mod-list` since we do not want them to be added to the closed index's `i-mod-list`). Creating the index at the server avoids having to send the index's `i-mod-list` to the FE when it is time to fix it up. In addition, the index is likely to become large, so the space overhead of keeping it in the FE cache and the cost of copying it back to the server may be quite high.

6.3 Performance

The Thor system architecture is novel. It has many features that can affect the performance our indexing scheme. This section addresses the following questions about performance:

1. What is the effect of very large segments?

2. What is the effect of two-level caching? In particular, what is the impact of swizzling and prefetching on the proposed implementation schemes?

We are interested both in the general effect these features have on computations and any specific effects they have on our proposed implementation schemes. Section 6.3.1 describes the results of extending our performance evaluation to 64K segments. Section 6.3.2 explores the effects that two-level caching may have on our implementations.

6.3.1 Very Large Segments

As we saw in Chapter 3, segment size has an impact on the performance of our benchmarks. Recall that our general conclusion was that for computations accessing the entire database, large segments were beneficial unless the cache was not large enough to hold all of the accessed segments. In this case, clustering mattered and when clustering did not match the pattern of access, large segments were detrimental. The segments in Thor are much larger than in conventional systems. To explore the effect of very large segments, we ran each of our benchmarks using 64K segments. This section reports our results.

For the first query benchmark (queries over a large portion of the database), the results for a system using 64K segments with a cold cache continued the trends reported earlier. Figure 6.4 extends our previous results for Query 1 (scanning without registrations) using the bit scheme with entries for 64K segments, and it shows that 64K segments are very beneficial when reading the entire database. Similarly, the trend for the crossover points for secondary index use (Query 4) to become lower as segment sizes increase continued since the likelihood of one match per data segment is much higher with very large segments. The crossover points for 64K segments are very low relative to the other segment sizes studied (about 0.5% for the small DB3).

On the second query benchmark (queries over a small, unclustered subset of the database), the results for 64K segments continue all of the trends reported earlier except for one. The difference can be seen in Figure 6.5. We see that the trend for larger segments to make Query 6 complete in fewer time steps is no longer true, although it should be more likely that there are multiple composite parts per segment since there are only 32 segments holding the data objects,

Seg. size	Query 1 execution time — Time steps (millions)			
	Small DB3	Small DB9	Medium DB3	Medium DB9
2K	1425	2370	–	–
8K	391	642	3410	5917
16K	223	360	1899	3295
64K	100	154	777	1346

Figure 6.4: Execution time of Query 1 using the bit scheme including 64K segments.

Seg. size	Comparison of Query 5 and Query 6 — Time steps (millions)							
	Small DB3		Small DB9		Medium DB3		Medium DB9	
	Qu. 5	Qu. 6	Qu. 5	Qu. 6	Qu. 5	Qu. 6	Qu. 5	Qu. 6
2K	1073	4070	1336	4593	–	–	–	–
8K	391	2899	633	3400	1653	24527	1827	24841
16K	222	2840	360	3095	1750	27100	2078	27665
64K	100	4041	154	4087	2788	43803	3234	44700

Figure 6.5: Execution time of Query 5 and Query 6 including 64K segments.

and the 2-megabyte cache holds 32 segments. However, it turns out the cache is too small. It can hold only 31 data segments because the result set also takes up some space, effectively the size of one segment. This is an artificial situation that can be remedied by having a slightly larger cache, but it does point out the impact that very large segments have on cache size. If the data accessed is not clustered very well, each disk access will bring in many unused data objects and may cause segments with useful objects to be thrown out when this might not happen with smaller segment sizes.

The results of the navigation benchmarks using 64K segments followed the trends reported earlier. In particular, when all of the accessed objects do not fit into the cache, traversals where the database is clustered in OID order continue to perform well, while traversals where the database is clustered in date index order perform worse.

We conclude from our results that 64K segments make clustering issues even more important due to the large number of objects that can fit into a segment. We would like it to be the case that most of the objects in a segment are "related" in some way, but as we have seen, it is not usually possible to have them clustered so that both queries and navigation are efficient. The optimal clustering of a set depends on the expected workload. In addition, we conclude that indexes are very important for queries over small, unclustered subsets of the database using complex functions when segments are very large, especially in large databases. The objects used in computing the function are unlikely to be clustered in any meaningful way and actually accessing them will fill the cache with many unused objects. Using an index to answer such a query is much more efficient.

6.3.2 Two-level Cache Structure

Since the Thor system is still under development, its architecture changes radically from time to time. We felt it was not interesting to simulate a version of Thor that may never exist. However, the two-level cache structure is an integral part of the Thor architecture that will not be changed, so we discuss the likely effect of this structure on how computations are run in the (new) embedded, (new) bit, and hybrid implementation schemes in this section.

For queries and navigation, we are interested in the incremental increase in space needed at the FE cache that is caused by the two-level cache structure for each implementation scheme. The first thing we note is that since we expect workloads to be mostly queries and navigation, we do not want registration information to be prefetched. Since registration information needs to be interpreted differently than regular data (e.g., checking if a tuple is long or short, or masking the table ID out of a reference to a registration object), both the server and the FE can handle them specially. In particular, we do not want the server to send any of the objects that are referenced in registration tuples in the prefetch group of a data object, and we do not want to swizzle these references into FE-surrogates at the FE.

Given this special handling, the relative performance of our implementation schemes on queries and navigation will be the same as before. The bit scheme will perform the best since the data objects take up less space than in the embedded and hybrid schemes. Having smaller data objects results in both less network delay to transfer the objects of interest to the FE cache and less space used in the FE cache.

In addition, the performance of navigation also depends on our ability to prefetch the right "related" objects. Different prefetching policies will have different effects on each implementation scheme. Under the current scheme where we only prefetch from the same segment as the requested object, the effect each implementation scheme has on clustering will have the most effect on performance. We would expect that since there are fewer objects in a segment in the embedded and hybrid schemes that fewer objects will be candidates for prefetching; thus their performance might be worse than the bit scheme. On the other hand, since segments are very large relative to object size, this may not be noticeable. In any case, a more permissive policy that took objects from many segments might cancel this effect by allowing the prefetch group in the embedded and hybrid schemes to include all objects that would have been part of the prefetch group in the bit scheme under the current policy.

For updates, we are interested in the number of extra fetches that are necessary to find registration information and finding the set elements of the affected entries. Updates in all three schemes incur an access to the table for an index that maps a class to the mutators that affect the index. These maps are likely to be small, so we can fetch them to the FE and keep them in the FE cache. This will happen only once per index, so they will probably have little impact on overall performance, and their effect is the same for all three schemes.

In the embedded scheme, the registration tuples are already present in a registered object, so there are no extra fetches for finding them. In some cases, there may not be any extra fetches to find a set element, e.g., if the registered object being modified is the set element or the set element is already at the FE. In the case where a set element is not already at the FE, we have to fetch it, and of course, we would like the prefetch group to contain the objects that the set element references, and so forth. If there are several registration tuples, we would like to fetch all of the affected set elements at the same time rather than one at a time, so we would need a way of indicating a group fetch to the server. Note that the embedded scheme avoids extra fetches at the cost of wasted space in the FE cache for registration information in registered objects that are not being modified.

When registrations are embedded in the hybrid scheme, it performs like the embedded scheme on updates. When the registered object has a reference, we need to access a registration object (to check for registrations) and then the set element. We have two choices for the access to a registration object. We can fetch the registration object to the FE cache, or we can avoid caching registration objects at the FE by adding an operation to the server that checks for registrations. If we cache the registration object at the FE, we might avoid some fetches as in the embedded scheme, e.g., when another object uses the same registration object and the set element(s) in its registration tuple(s) are already at the FE. However, registration objects may be large, and it is not clear if we will be able to cluster registration information into them in a way that will avoid the extra fetches for other registration objects. In addition, even if the registration object is present, we may still have to fetch the set element(s). Having a server operation to do registration checks saves space in the FE cache, but it means that an extra fetch is always needed. However, since the FE is doing a mutation (otherwise it would not have requested a registration check), we can prefetch the set elements named in the tuples of interest and some of their "related" objects, if they are at the same server, and not already at the FE, in the reply, thus avoiding the possible extra fetch for the set element. Therefore, there is just one fetch per update.

Hopefully, in the hybrid scheme, many objects will have embedded registration tuples, so that most of the time it will perform like the embedded scheme on updates. In particular, we believe set elements are more likely to have embedded registration tuples, since they probably are not shared as much as the lower-level objects. For other registered objects, the embedded registrations still may cause an extra fetch for the set element anyway, so having the server do registration checks when there is a reference to a registration object is probably a better tradeoff than caching registration objects at the FE.

For the bit scheme, we have the same tradeoff of space in the FE cache versus extra fetches that we have in the hybrid scheme when registered objects have a reference to a registration object. We would not want to cache the entire registration table, but we can cache parts of it (e.g., the entries for segments that have been accessed recently). Note that registration objects in this scheme are likely to be larger than in the hybrid scheme, so more space is potentially wasted if we cache registration objects. Also note that we must cache both registration table entries and registration objects to avoid any extra fetches, because there is no information in a registered object that will tell us which registration object is the one that contains its registration tuples, unlike the direct reference in the hybrid scheme. On the other hand, data objects in the bit scheme are smaller than in the hybrid scheme, so perhaps we can afford space to cache registration information. Then again, smaller data objects allow more data to be cached and as a result computations go faster, so perhaps it is better to use this extra space to cache more useful objects.

Our analysis does not change our evaluation of our implementation schemes. The bit scheme is still better for navigation and queries, so if the greater space requirement can be met, it is a very attractive choice. However, the hybrid scheme's performance on queries and navigation is not too far behind the bit scheme, and it probably performs better on updates. As we saw in Chapter 3, if there is moderate amount of sharing, the hybrid scheme requires less space than the bit scheme, thus it might be the best all-around choice. However, we do not have any experience with real situations, so it is hard to be certain that our assumptions are valid. These schemes should be evaluated more carefully with simulation studies like those done in Chapter 3 when the Thor architecture is stable.

Chapter 7

Conclusion

Associative access to data is an important service that databases provide to clients. Queries provide clients a way of identifying the data of interest by describing a property that the data must have. In object-oriented databases, we expect as clients become more sophisticated, they will want to compute function-based queries using user-defined functions over user-defined sets that are maintained by the client. A function-based query is one that expresses the property of interest as the result of applying a function to the elements of the set. Indexes are used to optimize query computations by providing a mapping of property values (keys) to the objects that have those property values.

This dissertation has presented a new function-based indexing scheme for object-oriented databases. Our indexing scheme supports more expressive queries than other schemes proposed for object-oriented databases. It preserves abstraction and encapsulation, so that set elements can have more than one implementation. It supports indexes over user-defined sets. And it supports indexes using user-defined index functions that are not observer methods of the set elements.

When an index is created over a set, the key associated with a set element is computed by invoking the index function on the set element. We register objects during key computation by recording the information needed to do index maintenance. This information is a tuple $< y, m, I, x >$ that indicates that mutator m of object y can affect the entry in index I for set element x. During a mutation, registration information is checked and appropriate updates are made to affected indexes. Registrations are limited to just the objects that if modified might cause the index to change, and updates to indexes happen only when changes that may affect the index occur. We try to minimize the number of registered objects and recomputations done by our scheme by requiring that type specifications declare dependency information between mutators and observers. If an object is accessed during key computation using an observer that does not depend on any mutators, it is not registered, and only the objects that have been registered have mutators that check for registration information.

In Chapter 3, we simulated three possible implementations of the registration information: a bit scheme where a registration table maps a data object to a registration object containing its registration tuples; a pointer scheme where each registered data object contains a reference to a registration object containing its registration tuples; and an embedded scheme where registration tuples are stored directly inside the registered object. Our results showed that function-based indexes allow function-based queries to be computed more efficiently than without an index. Indexes are especially useful when the index is a primary index, when is is a secondary index and the query is expected to have a low percentage of matches, and when is is a secondary index over a small set.

Our simulations showed that registration information stored inside data objects affects query and navigation performance when the cache is cold, because it causes data objects to become larger. This results in more disk accesses to bring in the same number of data objects. Thus, the bit scheme has superior performance to the pointer and embedded schemes on queries and navigation. Update performance is affected by the placement and overall size of registration information. The embedded scheme has superior performance on updates, since registration information is readily available and is of minimum size. However, the embedded scheme has unpredictable effects on system performance since the increase in size of the data objects cannot be bounded, so we concluded that an embedded scheme is an unsuitable implementation.

The number of registration tuples that must be kept for an index in our basic scheme is proportional to the number of set elements times the number of registrations per set element. In addition, if we do not store registrations tuples directly in a data object, there is space overhead for organizing the registration information elsewhere. We developed a framework for characterizing the space overhead in indexing schemes and analyzed each implementation. We proposed ways of reducing the size of registration information and suggested alternate implementations that reduce the overhead associated with keeping registration information outside of data objects. We concluded that a hybrid scheme where a data object stores either a small (fixed) number of registration tuples or a reference to a registration object is a suitable implementation when space is tight, and that the bit scheme can be used if maximum query and navigation performance is needed, and its greater space requirement can be met.

We proposed three optimizations to our basic scheme. The number of registrations are reduced by our contained subobject analysis, which allows us to avoid registering objects that can only be accessed inside some other object. When many mutations affect the same index entry, lazy updates avoid recomputations that overwrite new key values before they are used. And we can remove obsolete registrations through deregistration, reducing the number of unnecessary recomputations.

Finally, we presented a design for integrating function-based indexes into the Thor object-oriented database system. Thor is distributed with a two-level architecture of FEs and servers. Our design tries to minimize the changes to the current Thor system. We enriched the Thor server interface so that queries computed using indexes can be run at the server. For updates, key (re)computations are performed at the FE, while modifications to the index itself are done at the server. We developed predicate validation, a new optimistic concurrency control mechanism, to integrate concurrent index use and maintenance with the regular Thor optimistic concurrency control protocol. In predicate validation, index operations are represented as predicates and conflict detection is done in terms of these predicates. The backwards validation technique used by Thor makes it difficult for long transactions to commit, so we also developed an incremental index creation algorithm.

We also discussed the impact that Thor's system architecture has on our performance evaluation. Thor uses very large segments, so we extended our simulation study to include very large segments. We found that clustering is more important in systems with very large segments, since much more data is brought into the cache with each disk access. We also evaluated the effect of the two-level architecture on our proposed index implementations. We concluded that the two-level architecture does not change the conclusion of our performance evaluation: the hybrid scheme is still a reasonable choice, and the bit scheme can be used if the space required is worth the maximum performance on queries and navigation.

7.1 Future Work

These are several areas where this work can be extended. They can be divided into three categories: implementation issues, support for increased expressive power, and other miscellaneous ideas. We discuss them briefly in this section.

7.1.1 Implementation

Registration information. As we showed in Chapter 3, our scheme has a high space requirement that is proportional the number of set elements times the number of registrations needed for each set element. The GemStone implementation[47] only uses space proportional to the number of accessed objects, but it works only for path expressions. Bertino's method precomputation scheme[8, 10] attempts to provide a path-based implementation of a function-based indexing scheme, but registers too many objects because dependency is determined at the concrete level of instance variables rather than at the abstract level of methods. It would be interesting to see if we could combine our technique of using dependency at the abstract level to register fewer objects with Bertino's scheme to produce a GemStone-like implementation for our indexing scheme.

More declarative information. It would also be interesting to explore the question of whether other kinds of declarative information can be included in type specifications to reduce the number of registered objects. For example, if aliasing information were available for all types, we could do a better job of subobject containment analysis.

Simulating Thor. The analysis of our indexing scheme for Thor in Chapter 6 is based on assumptions that we have not tested. It would be interesting to simulate our indexing scheme under various workloads in the Thor architecture.

Distributed sets. In a distributed database like Thor, all of a set's elements may not be stored at the same server. There will always be a set object that refers to all of the elements, so we could still use the basic scheme and store an index at the same node as the set object. However, an interesting question is whether an index for a distributed set could be distributed across the servers that store the elements of the set, for example, if the index is very large and would take up too much space at one server. We also might be able speed up query processing, because partial match requests could be done in parallel. The major issues are how to partition indexes for distributed sets and how to keep these indexes up-to-date in the presence of the partitioning.

7.1.2 Expressive Power

Precomputed results. As we pointed out in Chapter 4, the GOM scheme[34, 35] and Bertino's method precomputation scheme[8, 10] have additional expressive power to return precomputed method results. As we saw in Chapter 3, this can be very beneficial for computations that call a complex function that accesses many objects. Thus we might like to provide the same functionality for elements of an indexed set. In addition, precomputed results would reduce the cost of updates in our index scheme, since we would not have to recompute the old key.

The main cost of providing precomputed results is the overhead of storing the results in a way that makes them easy to access. If this were not so, then computing the result would be just as efficient as looking it up. Storing the results inside the data object avoids an extra access to find the precomputed result, but as we have noted before, this has an impact on query and navigation performance by making data objects larger. If results are stored outside a data object, we would have to analyze the cost of accessing the result versus computing the result again, to determine if remembering the result is worthwhile. For example, for a very complex function that accesses many objects, it still may be more efficient to access the precomputed result rather than compute the function. In addition, if a precomputed result is not stored with the data object, some form of predicate validation may be necessary to allow concurrent use and maintenance of the precomputed result.

Generalized index functions. In this dissertation, we assumed index functions are total, only take one argument, the set element, and return a result of a built-in type. We might like to relax these assumptions. The GemStone scheme[47] allows path expressions that return keys of user-defined type. The keys of the index are OIDs, and the only queries allowed are ones that select on the identity of a key. However, some user-defined types have "natural" ordering properties, for example, a type representing complex numbers. We might like to support range queries for keys of these types, but that would require running user code during index creation, queries, and updates to determine how two objects compare. Our design for Thor would have to be reevaluated, since index construction, queries, and updates are done at the server.

We might like to allow index functions that are not total. In Theta, this would allow index functions that can signaled an exception. For example, the *manager* method of a `project` object may signal *no_manager* if there is no current manager of the project. We can use Bertino's idea of an "undefined" key[8, 10] and associate it with any set elements whose key computations signal an exception. Entries with undefined keys would not match any query.

To extend the registration algorithm to support functions with multiple arguments, we would add any arguments to the initial reachability set R. However, as in the GOM scheme[34, 35], support for additional arguments would require that we remember the arguments that are used to compute a key as part of our registration information. This would increase the size of registration information, and it is not clear if the benefit of saving these results will be enough to warrant the extra space overhead. It is also not clear if the results of these types of functions make sense as index keys, or if they are only useful as precomputed results.

7.1.3 Miscellaneous

Views and constraints. Indexes are system-defined derived data. We believe that our registration technique can be used to support user-defined derived data such as views and constraints. A view is a result set of a query that tracks changes in the base sets used to compute the query. To maintain a view efficiently, we do not want to recompute the entire query whenever there are mutations, but rather just the part that has been affected. A constraint is a computed property of some particular objects that must always hold. We only want to check that the constraint is maintained if there is reason to believe that the objects involved have changed in a way that might affect the property. Views and constraints could be maintained by registering objects as the derived data is computed; when registered objects change, the derived data is invalidated or recomputed. As in index maintenance, the main issues are determining what information should be in a registration tuple and when it needs to be checked.

Predicate validation for user-defined types. User-defined types may be able to provide more concurrency in an optimistic scheme if conflicts are determined on the basis of semantic information rather than on the reading and writing of physical versions[30]. It would be interesting to explore whether predicates and conflict rules between predicates can capture this semantic information in a compact way.

References

[1] Atul Adya. Transaction management for mobile objects using optimistic concurrency control. Master's thesis, Massachusetts Institute of Technology, February 1994.

[2] P. M. G. Apers, A. R. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, SE-9(1):57–68, January 1983.

[3] M. Atkinson et al. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[4] Jay Banerjee et al. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[5] D. S. Batory et al. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1729, November 1988. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[6] D. S. Batory and C. C. Gotleib. A unifying model of physical databases. *ACM Transactions on Database Systems*, 7(4):509–539, December 1982.

[7] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems*, 13(3):231–262, September 1988. Also University of Texas at Austin Techreport TR-86-24.

[8] E. Bertino. Method precomputation in object-oriented databases. In *Proceedings of the ACM-SIOIS and IEEE-TC-OA International Conference on Organizational Computing Systems*, pages 199–212, Atlanta, Georgia, November 1991.

[9] E. Bertino. A survey of indexing techniques for object-oriented databases. In J. C. Freytag, G. Vossen, and D. Maier, editors, *Query Processing for Advanced Database Applications.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993. Forthcoming.

[10] E. Bertino and A. Quarati. An approach to support method invocations in object-oriented queries. In *Proceedings of the 2nd IEEE International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 163–168, Tempe, Arizona, February 1992.

[11] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[12] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[13] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[14] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[15] Michael J. Carey et al. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, August 1986.

[16] Michael J. Carey et al. The EXODUS extensible DBMS project: An overview. Technical Report 808, Computer Science Department, University of Wisconsin - Madison, 1988. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[17] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. In *Proceedings of the USENIX File Systems Workshop*, pages 79–91, 1992.

[18] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[19] W. W. Chu and P. Hurley. Optimal query processing for distributed database systems. *IEEE Transactions on Computers*, C-31(9):835–850, September 1982.

[20] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, Cambridge, MA and New York, 1990.

[21] Mark Day. *Managing a Cache of Swizzled Persistent Objects.* PhD thesis, Massachusetts Institute of Technology, 1994. Forthcoming.

[22] Mark Day et al. *Theta Reference Manual.* Programming Methodology Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1994.

[23] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[24] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[25] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 11(11):624–633, 1976.

[26] Sanjay Ghemawat. *Disk Management for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1994. Forthcoming.

[27] J. N. Gray. *Notes on Database Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, New York, 1978.

[28] Laura M. Haas et al. Extensible query processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 377–388, Portland, OR, June 1989. Also IBM Almaden Research Center Research Report RJ 6610 (63921).

[29] Laura M. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990. Also IBM Almaden Research Center Research Report RJ 7278 (68535).

[30] Maurice Herlihy. Optimistic concurrency control for abstract data types. In *Fifth ACM Principles of Distributed Computing Conference*, 1986.

[31] Scott E. Hudson and Roger King. CACTIS: A database system for specifying functionally-defined data. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 26–37, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[32] Scott E. Hudson and Roger King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[33] Mathias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[34] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases. Technical Report 28/90, Universität Karlsruhe, October 1990.

[35] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 258–267, Denver, Colorado, May 1991.

[36] Won Kim et al. Integrating an object-oriented programming system with a database system. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–152, San Diego, CA, September 1988.

[37] Won Kim et al. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, June 1989.

[38] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[39] H. T. Kung and J. T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[40] Charles Lamb et al. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[41] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, Atlanta, GA, June 1988.

[42] Barbara Liskov. Preliminary design of the Thor object-oriented database system, March 1992. Available as Programming Methodology Group Memo 74, MIT Laboratory for Computer Science.

[43] Barbara Liskov et al. A highly available object repository for use in a heterogeneous distributed system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems Design, Implementation, and Use*, pages 255–266, Martha's Vineyard, MA, September 1990.

[44] Barbara Liskov et al. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 226–238, Asilomar Conference Center, Pacific Grove, CA, 1991.

[45] Guy M. Lohman et al. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[46] Umesh Maheshwari. Distributed garbage collection in a client-server, transactional, persistent object store. Technical Report MIT/LCS/TR-574, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1993. Master's thesis.

[47] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[48] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[49] C. Mohan and Inderpal Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 361–370, San Diego, California, June 1992.

[50] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(3), August 1992.

[51] Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 8–17, Toronto, Ontario, Canada, 1988.

[52] Jack Orenstein et al. Query processing in the ObjectStore database system. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 403–412, San Diego, CA, June 1992.

[53] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES data model. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 83–96, Brighton, England, September 1987. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

[54] P. Schwarz et al. Extensibility in the Starburst database system. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 85–92, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[55] Michael Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering*, pages 262–269, Los Angeles, CA, February 1986.

[56] Michael Stonebraker. Object management in Postgres using procedures. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, pages 66–72, Asilomar Conference Center, Pacific Grove, CA, September 1986.

[57] Michael Stonebraker et al. Application of abstract data types and abstract indices to CAD data bases. In *Engineering Design Applications, Proceedings from SIGMOD Database Week*, pages 107–113, San Jose, CA, May 1983.

[58] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[59] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[60] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, October 1987.

[61] P. F. Wilms et al. Incorporating data types in an extensible database architecture. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 180–192, Jerusalem, Israel, June 1988.

[62] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.