VLDB Journal, 4, 445-492 (1995), Malcolm Atkinson, Editor
©VLDB

TIGUKAT: A Uniform Behavioral Objectbase Management System

M. Tamer Özsu, Randal Peters, Duane Szafron, Boman Irani, Anna Lipka, and Adriana Muñoz

Received June, 1993; revised version received, May, 1994; accepted March, 1995.

Abstract. We describe the TIGUKAT objectbase management system, which is under development at the Laboratory for Database Systems Research at the University of Alberta. TIGUKAT has a novel object model, whose identifying characteristics include a purely behavioral semantics and a uniform approach to objects. Everything in the system, including types, classes, collections, behaviors, and functions, as well as meta-information, is a first-class object with well-defined behavior. In this way, the model abstracts everything, including traditional structural notions such as instance variables, method implementation, and schema definition, into a uniform semantics of behaviors on objects. Our emphasis in this article is on the object model, its implementation, the persistence model, and the query language. We also (briefly) present other database management functions that are under development such as the query optimizer, the version control system, and the transaction manager.

Key Words. Objectbase management, database management, reflective system, persistent storage system.

1. Introduction

The penetration of data management technology into new application areas with more demanding requirements than business data processing has generated a search for appropriate data models and system architectures to support these requirements. Some examples of these application areas are engineering design systems, knowledge base system applications, office information systems, and multimedia systems. It is now commonly accepted that relational database management systems (DBMSs),

M. Tamer Özsu, Ph.D., is Professor, *ozsu@cs.ualberta.ca*, Randal Peters, Ph.D., was graduate student, Duane Szafron, Ph.D., is Associate Professor, Boman Irani, M.Sc., Anna Lipka, M.Sc., and Adriana Muñoz, M.Sc., were graduate students, Laboratory for Database Systems Research Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. R. Peters is now Assistant Professor, University of Manitoba, B. Irani and A. Muñoz are with Bell Northern Research, and A. Lipka is a Ph.D. candidate at University of Toronto.

with their flat representation of data, do not have sufficient power to fulfill these requirements. The fundamental difficulty relates to the recognized semantic mismatch between the entities that are commonly encountered in these application domains and the representation provided by the underlying DBMS.

Object-oriented technology is the topic of intense study as the major candidate to successfully meet the requirements of advanced applications that use data management services. At the Laboratory for Database Systems Research at the University of Alberta, we are engaged in the design and development of an objectbase management system (OBMS)[1] called TIGUKAT,[2] which follows object-oriented methodology in its own design. Consequently, all database functionality is incorporated within an extensible object model. In this article, we provide a general overview of TIGUKAT with special emphasis on its object model, its implementation, and the persistence model. Some of the novel features of TIGUKAT are the following:

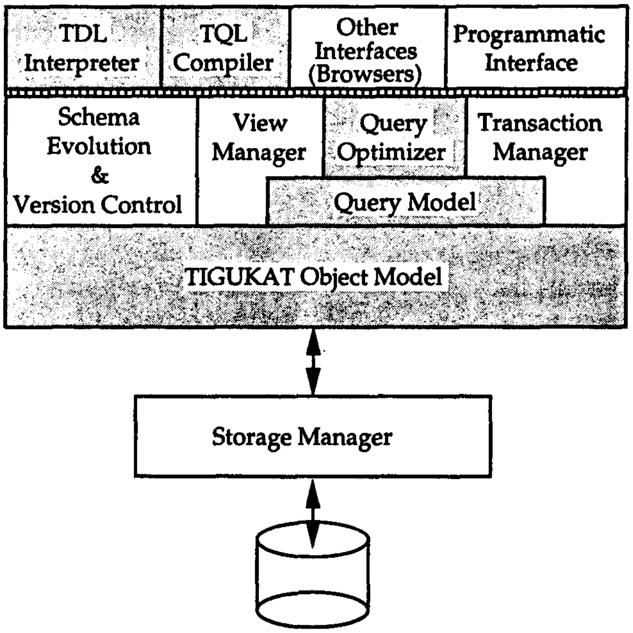
1. It has a purely behavioral object model where the user (a person or an application program) interacts with the system only by applying behaviors to objects. In this way, full abstraction of modeled entities is accomplished, since the users do not have to differentiate between attributes and methods.
2. Its object model is uniform. Everything in the system, including types, classes, collections, behaviors, functions, and meta-information, is a first-class object with well-defined behavior. Thus, there is no separation between objects and values, so the schema information is a natural part of the database that can be queried just like other objects.
3. This uniformity extends to other system entities (e.g., queries, transactions, views) which are treated as objects that can be created, stored, manipulated, and queried like any other object.

Two different approaches have been followed in the development of OBMSs. The first is to adopt the type system of an object-oriented programming language as the object model of the OBMS. For example, ObjectStore (Lamb et al., 1991) adopts the type system of C++ (Stroustrup, 1986), while Gemstone (Butterworth et al., 1991) follows the type system of Smalltalk (Goldberg and Robson, 1983). The second alternative is what is known as language-independent or generic object models where the OBMS defines its own object model, and appropriate mappings are provided from languages to this object model. TIGUKAT follows the second

1. We prefer to use the terms "objectbase" and "objectbase management system," rather than the more popular "object-oriented database" and "object-oriented database management system," since not only data in the traditional sense are managed, but objects in general, which includes things like code in addition to data.

2. TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

Figure 1. TIGUKAT System Architecture



approach as does, for example, O_2 (Deux et al., 1991). A database programming language is being designed, which is tightly integrated with the TIGUKAT object model. In addition, mappings will be provided from other programming languages.

TIGUKAT is an experimental system that is under constant development and revision. We have, therefore, chosen to follow an extensible system design approach. The uniformity of the model, which treats all system entities as objects, is the basis of TIGUKAT's extensibility. The general architecture of the system is depicted in Figure 1. To date, most of the development and implementation work has concentrated on the object model, the query model, and the implementation of query languages. The architectural framework of the query optimizer also has been developed (Muñoz, 1993); the details of the optimizer (e.g., the full set of transformation rules, and the detailed cost functions) have yet to be implemented, however.

The organization of this article is as follows. Section 2 provides an overview of the TIGUKAT object model, presenting the primitive type system. We include an example database application design to demonstrate the features of TIGUKAT. Section 3 describes some of the more important implementation design decisions and the approach we have taken. This is followed, in Section 4, with a description of the persistence model of TIGUKAT. Section 5 presents the query model with emphasis on the user-level languages. (A more detailed description of the object and query models are given in Peters et al., 1993*a*, Peters, 1994.) In Section 6, we provide a brief overview of our approach to providing the common database management functions such as query optimization, version management and transaction management. In Section 7, we end with a discussion of our future research directions.

2. Object Model

The TIGUKAT object model is defined behaviorally with a uniform object semantics. The model is *behavioral* in the sense that all access and manipulation of objects occurs through the application of behaviors (operations) on objects. The model is *uniform* in that every concept within the model has the status of a first-class object.

Uniformity in TIGUKAT is similar to the approaches of DAPLEX (Shipman, 1981) and its object-oriented counterpart OODAPLEX (Dayal, 1989). However, our definition of uniformity is complete in that it unconditionally extends over all forms of information, including the system components such as the schema, meta-information, query model, query optimizer, view manager, and transaction manager. We adopt another significant aspect of these models: their functional approach to defining behaviors. TIGUKAT enhances this approach by providing a separation of behavior, which is a semantic notion, from function, which is a means of implementing behavioral semantics.

The TIGUKAT model defines a number of primitive objects that include: *atomic entities* (e.g., reals, integers, strings); *types* for defining common features of objects; *behaviors* for specifying the semantics of the operations that may be performed on objects; *functions* for specifying the implementations of behaviors over various types;[3] *classes* for the automatic classification of objects based on their type;[4] *collections* for supporting general, heterogeneous, user-definable groupings of objects; and *higher-level constructs* to uniformly represent meta-information (i.e., schema) as objects with well-defined behavior. This last feature gives the system *reflective capabilities* (Peters and Özsu, 1993).

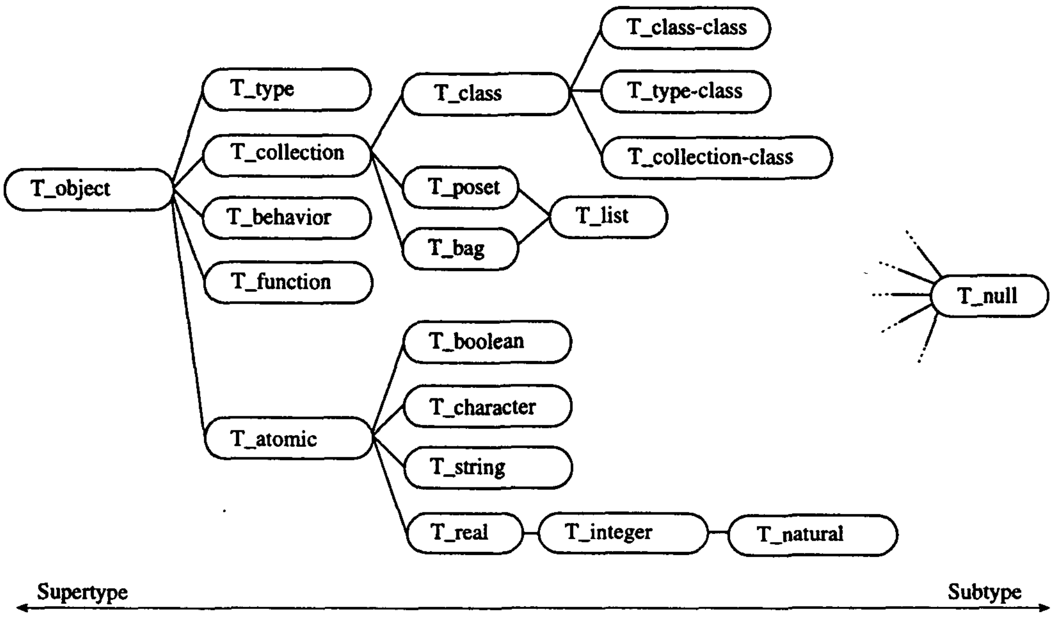
The primitive type system of TIGUKAT is shown in Figure 2 with the type `T_object` as the root of the lattice, and type `T_null` as the base. The type `T_null` defines objects that can be returned by behaviors when no other result is known (e.g., null, undefined). These are necessary because the result of every behavior application in TIGUKAT must be a reference to an object. There are no dangling references in TIGUKAT.

As a notational convenience, the prefix `T_` refers to a type, C_ refers to a class, **L_** refers to a collection, *B_* refers to a behavior, and *F_* refers to a function. Each prefix also has its own font variation for the string following it. For example, `T_city` is a type reference, C_city is a class reference, **L_historicSites** is a collection reference, *B_population* is a behavior reference, *F_calcPopulation* is a function reference, and a

3. Associations between behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.

4. Types and classes are separate constructs in TIGUKAT.

Figure 2. Primitive type system of TIGUKAT



reference such as Edmonton without any prefix represents some other application-specific object reference.

2.1 Behaviors and Functions

The access and manipulation of objects occurs exclusively through the application of behaviors. This is similar to the message-based approach of Smalltalk (Goldberg and Robson, 1983) and OODAPLEX (Dayal, 1989). Appendix A lists the signatures for the native behaviors defined by the primitive types of Figure 2.

We separate the definition of a behavior from its possible implementations, which are represented by TIGUKAT functions (corresponding to *methods* in other models). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types (known as *overloading* the behavior). This gives the model the ability to *dynamically bind* behaviors to implementations at run time (known as *late-binding*).

There are two kinds of implementations for behaviors. One is a *computed function*, which consists of runtime calls to executable code, and the other is a *stored function*, which is a reference to an existing object in the objectbase. Stored functions eliminate the need for instance variables, which limit reuse (Wirfs-Brock and Wilkerson, 1989*b*). The uniformity of TIGUKAT conceptually transforms each behavioral application into the invocation of a function, regardless of whether the function is stored or computed. This allows designers to concentrate on semantic responsibilities rather than on data attributes (Wirfs-Brock and Wilkerson, 1989*a*).

For example, the type designer is free to develop a purely behavioral specification of a type while the type implementor decides whether the behaviors are implemented by stored or computed functions.

The semantic definition of a behavior has many forms. A simple approach, common in other models, is a *signature* expression, consisting of a behavior name, parameter types, and a return type. Signatures are useful and necessary for describing the semantics of behaviors, but they are inadequate for characterizing the full semantics. For now, we assume that a proper semantic specification mechanism exists. In the current model design, a behavior is specified only by its signature. However, the extensibility of the model makes it easy to incorporate a more complete semantic specification when one is developed. The only extension required is to modify the implementation of the *B_semantics* behavior on `T_behavior` to correspond to the new, more complete semantics. We currently are investigating specification techniques and denotational semantics as a complete semantic description mechanism for behaviors.

Behaviors are applied to objects. The application of a behavior, say *B_population*, to an object, say Edmonton, using some arguments, say $a_1, \ldots ,a_n$, can be denoted by (*B_population* (Edmonton))($a_1, \ldots ,a_n$) or by use of the dot notation Edmonton.*B_population* ($a_1, \ldots ,a_n$). In either case, the object Edmonton is called the *receiver* of the behavior.

Behaviors are instances of the type `T_behavior` and functions are instances of the type `T_function`. We use an arrow "$\longrightarrow$" in function type specifications, and we curry multiple argument functions. A function type is of the form $\mathcal{A} \longrightarrow \mathcal{R}$ where $\mathcal{A}$ represents the argument type expression of the function, and $\mathcal{R}$ represents the result type. In general, the argument and result types may be any type specification, including a function type. Then, by currying, multiple argument functions may be specified.

As defined in more detail in Section 2.3, types are related to each other via *subtyping* (also referred to as *behavioral inheritance*). A behavior defined on a type `T_x` is *inherited* in the type if and only if the behavior is defined in a supertype of `T_x`. A behavior defined on a type `T_x` is *native* in the type if and only if the behavior is not defined in any supertype of `T_x`.

Inherited behaviors do not necessarily borrow their implementation from their supertypes (although this can be set as the system default). Therefore, we define a separate reuse mechanism for implementations called *implementation inheritance*. An implementation of a behavior in a type `T_x` is inherited if and only if the behavior is inherited and the function implementing the behavior in `T_x` is the same as a function implementing the behavior in a supertype of `T_x`. Otherwise, the implementation of the behavior is *redefined* (or *overridden*) in `T_x`.

TIGUKAT supports multiple subtyping. However, the separation of behaviors from functions introduces the need for separating behavioral inheritance from functional inheritance, and for defining separate conflict resolution schemes for both. Implementation inheritance conflicts are resolved using an approach similar to the one used in Modular Smalltalk (Wirfs-Brock and Wilkerson, 1988). Specifically, it is an error for a type to inherit two different implementations (i.e., two instances of `T_function`) for the same inherited behavior. The error is resolved by explicitly redefining[5] the `T_function` for that behavior. Note that one choice for redefinition is one of the two conflicting `T_functions`. No separate mechanism is required to solve inheritance conflicts between instance variables, because there are no instance variables. Stored function conflicts are resolved in the same uniform manner as computed function conflicts. Furthermore, in the context of a complete behavioral semantics, there are no behavioral inheritance conflicts. That is, the inherited behavior in the multiple supertypes will be semantically equivalent or not. When they are equal, only one behavior is defined in the subtype. When they are not equal, multiple behaviors are defined in the subtype.

2.2 Objects

An *object* is a fundamental primitive in TIGUKAT because the conceptual level of the model deals uniformly with objects. Objects are defined as unique (*identity, state*) pairs where *identity* represents a unique, immutable system-managed object identity (or OID), and *state* represents the information carried by the object. There are system defined mappings OID(o) and *state*(o) that accept an object o and return the OID or state of o, respectively. These are internal mappings used only by the system, and are not visible to the user. The existence of unique OIDs does not preclude application environments such as object programming languages from having many *references* (or *denotations*) to objects, which need not necessarily be unique and may even change, depending on the scoping rules of the application.

In TIGUKAT, every object can be viewed as a *composite* object, meaning that every object has references/relationships (not necessarily implemented as pointers) to other objects. These other objects are returned as results of behavior applications, but it does not matter whether the behaviors are implemented by stored or computed functions. For example, even integers are composite objects since they have behaviors that return objects.

Object existence, access, and manipulation in TIGUKAT is based on the notions of reference, scope, and lifetime. This is similar to other model proposals (e.g., Kent, 1990; Snyder, 1990; Fong et al., 1991) in that the only user-expressible representation of an object is a *reference* within a particular scope. A *scope* defines the visibility, access paths, and lifetime of object references. The *lifetime* of an object is independent of the *lifetime* of a reference to that object within a particular scope. That is, when a reference to an object disappears at the end of a scope, the object being referenced does not necessarily disappear along with it. This can depend on the definition of the scope, and the persistence of the object. From the database perspective, there is also the issue of explicit deletions and the dangling

5. Redefinition may be the explicit writing of a new function or simply choosing an existing function.

reference problem that follows. That is, when an object is explicitly deleted, all references to that object should no longer point to the object and somehow be invalidated. In TIGUKAT, every behavior application is a reference to an object. Thus, we do not invalidate references, but rather bind them to an object whose type is `T_null`. That is, when an object is explicitly deleted, the object is changed to an instance of type `T_null` (called undefined), so that all references to it remain valid. Garbage collection is used to reclaim the storage of deleted objects. The deletion semantics is explained in more detail in Section 4 since a similar approach is used when a persistent object is made transient. The similarity stems from the fact that subsequent programs will not see the persistent object that was made transient and it will appear as though the object was deleted. Another condition for object deletion and storage reclamation is if an object no longer has references through its class.

Operations on objects are performed through behaviors, and object access is specified through references. Therefore, an operation on an object reference in a particular scope represents the application of a behavior to the actual object that is referenced. We define several behaviors on the type `T_object` that are inherited by all types and, therefore, are applicable to every object. A mechanism is required to determine if two object references refer to the same object. This requirement is met by the behavior *B_equal*. For any two object references R_i and R_j, the result of $(B\_equal(R_i))(R_j)$ is true if and only if OID(R_i) and OID(R_j) map to the same object identity. The above operation is more commonly specified as $R_i = R_j$.

This is the only kind of equality that the primitive model defines. It is quite strong in that the only way two object references are equal is if they refer to the same object (with the same identity). Our notion of object equality is the same as "identity equal," defined in Khoshafian and Copeland (1986), or "0-equality" defined in Lecluse et al. (1988). We do not define, at this level, any notions of shallow or deep equality found in other models (Khoshafian and Copeland, 1986; Lecluse et al., 1988; Osborn, 1988) or in their extended versions, which determine equality at various levels (Shaw and Zdonik, 1990). These notions can be defined as equivalence relationships on the behavioral characteristics of objects and, therefore, should be left to customized interpretations at the behavioral level, rather than being part of the primitive model definition. For example, one may define person equality based on the equality of their social insurance numbers. The implementation of *B_equal* in a type `T_person` can be overridden to implement this semantics.

Objects in the model are strongly-typed[6] in the sense that each object is associated with a single type. A type defines all the behaviors applicable to the objects of the type. The *B_mapsto* behavior, when applied to object o, returns the type of that object. It is important in type-checking and query processing to know the type of

6. Note that this differs from another common meaning of strong typing, which refers to static type-checking.

an object (Straube and Özsu, 1990*b*).

Another behavior defined on `T_object` is the identity mapping behavior *B_self*, which maps every object to itself. That is, for any object *o*, *B_self*(*o*) = *o*. There are additional behaviors whose presentation depends on other primitive concepts. We introduce them as these concepts are defined.

2.3 Types

A *type* defines behaviors and encapsulates hidden behavioral implementations (including state) for objects created using the type as a template. The behaviors defined by a type describe the *interface* for the objects of that type. Types are organized into a lattice structure, using the notion of *subtyping*, which promotes software reuse and incremental type development. Since TIGUKAT supports multiple subtyping, the type structure is potentially a directed acyclic graph (DAG). However, this DAG is transformed to a lattice by rooting it at `T_object` and *lifting* with the primitive type `T_null`.

The uniformity of TIGUKAT implies that types also are objects with their own state and identity along with their own type. The type that describes all type objects is `T_type`, and it is accessible in the same manner as any object. Thus, in addition to serving as descriptions of objects, types are objects themselves, and the type `T_type` serves as a description for all other types, including itself. This is known as the **type:type** property (Cardelli, 1986) in programming languages. The state of a type object consists of a structural specification of its instances (a template), references to the encapsulated behaviors it defines, references to its subtypes and supertypes, and a reference to its associated class.

Two relationships on types have been identified (Özsu et al., 1994). One is the concept of a type *specializing* another type in a manner similar to what was described in Maier et al. (1989). The other is the more popular, and stronger, notion of explicitly creating a type to be a *subtype* of another type (Cardelli, 1984). A type `T_1` specializes a type `T_2` if `T_1` defines all the behaviors of `T_2` (and possibly more). A type `T_1` is explicitly created as a subtype of a type `T_2`, which means `T_1` specializes `T_2`, and all the instances of `T_1` are also instances of `T_2`. Thus, subtyping implies, specializes, and defines a subset inclusion relationship on type extents. Conversely, specializes does not imply subtyping. Furthermore, subtyping supports IS-A relationships between types whose consequence is *substitutability* (Shaw and Zdonik, 1990). Accordingly, an object of type `T_x` can be used (substituted) in any context specifying a supertype of `T_x`. Specialize on its own does not support substitutability. Specialize is a semantic property derived from the behaviors defined on types, while subtyping is an explicit use of this property to define a partial order on types and a subset inclusion relationship on their extents.

A behavior is required on types that determines the class of a given type. To create objects of a particular type, there must be a class associated with the type to manage its instances. However, types do not require an associated class if there are no instances of that type (e.g., *abstract types*). `T_type` defines behavior *B_classof* for accessing the unique class (if it exists) associated with a particular type. Primitive types such as `T_integer` and `T_real` also have associated classes (we refer to Peters, 1994, for a discussion of these classes).

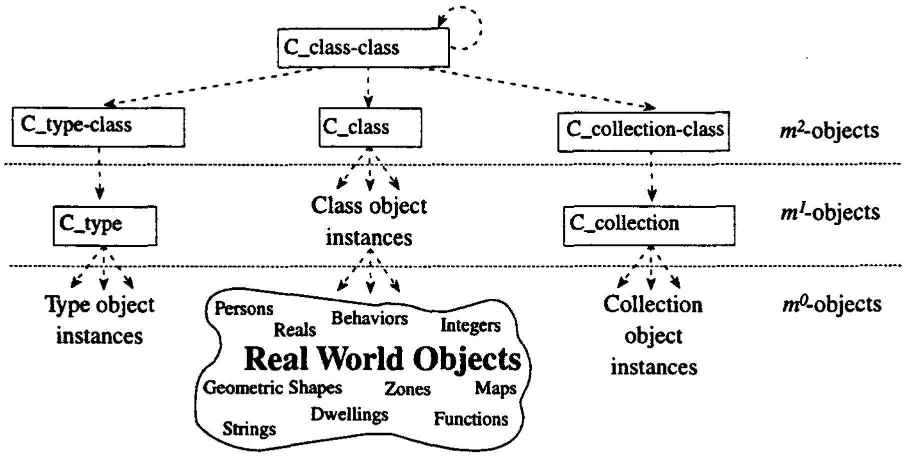
2.4 Classes and Collections

A *class* ties together the notions of *type* and *object instances*. The entire group of objects of a particular type, including its subtypes, is known as the *extent* of the type, and is managed by its *class*. We refer to this as the *deep extent*, and introduce a *shallow extent* that refers only to those objects created from the given type without considering its subtypes. The deep extent imposes a subset inclusion relationship on classes. We refer to this as *subclassing*, which has a direct relationship to subtyping on types. That is, a class **C_x** is a *subclass* of a class **C_y**; that is, the deep extent of **C_x** is a subset of the deep extent of **C_y**, if and only if the type associated with **C_x** is a subtype of the type associated with **C_y**.

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Another feature of classes is that object creation occurs only through a class using its associated type as a template for the creation. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes*, which imply *types*. Defining object, type, and class in this manner introduces a clear separation of these concepts. This separation is important in schema evolution, which manipulates type objects into new subtype relationships and need not be concerned with the overhead of classes. Furthermore, many object-oriented systems include *abstract types*, whose sole purpose is to serve as placeholders for common behaviors of subtypes and are never intended to have any instance objects. In these cases, there is no reason to manage classes for abstract types, because there are no instances of these types.

We define a *collection* as a general user-definable grouping construct (other constructs include *bags* for maintaining duplicates, *posets* for partially ordered collections, and *lists* that encompass both properties). A *collection* is similar to a class in that it groups objects, but it differs in the following respects. First, object creation may not occur through a collection; object creation occurs only through classes. This means that collections only form user-defined groupings of existing objects. Second, an object may exist in any number of collections, but is a member of the shallow extent of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the type lattice, whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (shallow extent), along with the extensions of its subtypes (deep extent). Therefore, the elements of a class are homogeneous up to inclusion polymorphism. A collection is heterogeneous in the sense that it can contain objects of types unrelated by subtyping. Furthermore, there is no distinction between shallow and deep extents for collections.

Figure 3. Three-tiered instance structure of TIGUKAT objects



In TIGUKAT, we define `T_class` as a subtype of `T_collection`, which introduces a clean semantics between the two, and allows the model to use them in a uniform way. For example, the targets and results of queries are typed collections of objects and, since classes are a subtype of collection, they may be used in queries as well. This approach provides flexibility and expressiveness in formulating queries and gives *closure* to the query model, which often is regarded as an important feature (Blakeley, 1991; Yu and Osborn, 1991).

2.5 Higher Level Constructs and Reflection

The types `T_class-class`, `T_type-class`, and `T_collection-class` in Figure 2 make up the *meta* type system. Their placement within the type lattice directly supports the extensibility of the model. The meta-model uniformly represents meta-information as first-class objects with well-defined behavior, and it maintains the behavior application abstraction on these constructs. This means that all properties of the model apply to this higher-level information uniformly. This property has been referred to as *reflection* (Peters and Özsu, 1993).

The higher-level objects are called *meta-objects* because they provide support for other objects. For example, `T_type` provides support for types, and **C_class** manages the class objects in the system. These meta-objects are uniformly managed by means of the primitives. This is possible through the introduction of higher level constructs called *meta-meta-objects*. Our model defines a three-tiered structure for managing objects, as depicted in Figure 3. Each box in the figure represents a class and the text within the box is the common reference name of that class. The dashed arrows represent instance relationships with the head of the arrow being the instance and the tail being the class to which it belongs.

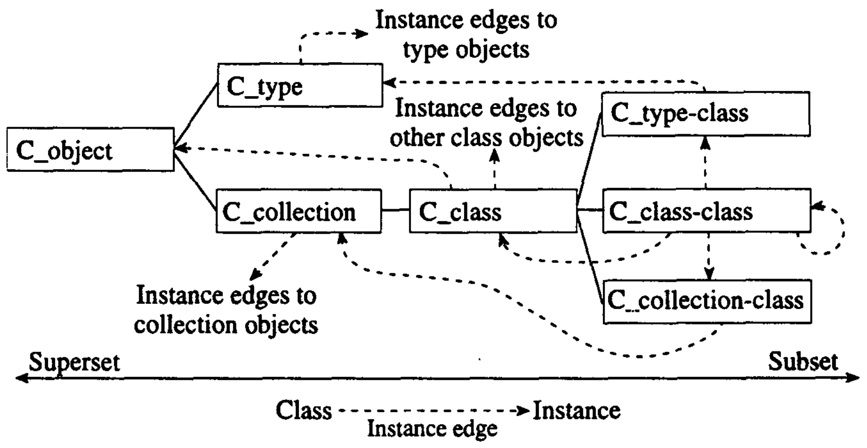
The lowest level of our instance structure consists of the "normal" objects that depict real world entities, such as integers, dwellings, maps, and behaviors. Type and collection objects also reside at this level, which illustrates the uniformity in TIGUKAT. We define this level as m^0 and classify its objects as m^0-objects. The second level defines the class objects whose associated types maintain schema information for the objects below it. These include **C_type**, **C_collection**, and most other classes in the system. This level is denoted as m^1, and its objects are m^1-objects. At this level, classes maintain the objects of the system (objects cannot exist without classes), and they are associated with types that define the schema information of their instances (classes cannot exist without types). Thus, classes represent the binding management between objects and the operations that can be performed on them as defined by their type. The upper-most level consists of the meta-meta-information (labeled m^2), which defines the functionality of the m^1-objects and is used to give definitional properties to these objects. The structure is closed off at this level because the m^2-class **C_class-class** is an instance of itself, as illustrated by the looped instance edge.

In the following discussion, we show the interactions among the various levels of the structure, and how they contribute to the uniformity of TIGUKAT, which in turn forms the foundation for reflection. We refer the reader to the primitive type lattice in Figure 2, and to a portion of its companion primitive class lattice shown in Figure 4. Each **C_x** class in Figure 4 is associated with the corresponding **T_x** type in Figure 2.

Figure 4 illustrates the subset inclusion and instance structure of some of the m^0, m^1, and m^2-objects in relation to one another. Starting from the left-side of the lattice structure, we explain the relationships between these classes and their instances. The class **C_object** is an m^1-object that maintains all the objects in the objectbase (i.e., every object is in the deep extent of class **C_object**). Two other m^1-objects in the figure are subclasses of **C_object**, namely, **C_type** and **C_collection.** These two classes maintain the instances of types and collections, respectively. Class **C_collection** is further subclassed by the m^2-object **C_class**, because every object that is a class is also a collection of objects. For example, the class **C_city** is an instance of the class **C_class**, as well as a collection of city objects. The deep extent of **C_class** manages all classes in the system, such as **C_object**, **C_type**, and **C_city**. Finally, **C_class** is subclassed by m^2-objects **C_type-class**, **C_class-class**, and **C_collection-class**. Intuitively, **C_type-class** is a class whose instances are classes that manage type objects. Similarly, **C_class-class** is a class whose instances are classes that manage class objects and **C_collection-class** is a class whose instances are classes that manage collection objects.

This meta-architecture is sufficient for managing all objects, including meta-information, in a uniform way. This provides the foundation for reflective capabilities such as the support for class behaviors and reflective queries. To support class

Figure 4. Subclass and instance structure of m^1 and m^2 objects



behaviors, each class can be made an instance of its own meta-class, instead of the common meta-class **C_class**. For example, to define a class behavior *B_averageAge* on **C_person** that computes the average age of the persons, we can uniformly extend the meta-model by creating a type `T_person-class` as a subtype of `T_class`, defining the behavior *B_averageAge* on `T_person-class`, creating an m^2-class **C_person-class** as the associated class of `T_person-class`, and creating **C_person** as an instance of **C_person-class**. Now, we can create person instances of **C_person** in the usual way, and *B_averageAge* is applicable to **C_person** and returns the average age of all persons in **C_person**. We can define many other class behaviors on `T_person-class`, including various object creation and initialization behaviors. This approach is in contrast to the usual way of making **C_person** a direct instance of **C_class**. If this is done, it is difficult to define class behaviors for **C_person** since **C_class** typically has many class instances and any class behavior defined on `T_class` would apply to all class objects. Our approach is superior to an approach that defined an extra m^2-class for every class (e.g., Smalltalk), since it has smaller space overhead.

More powerful extensions also are possible. For example, although **C_person-class** is a separate m^2-class for **C_person**, it can be used to group other related classes, such as **C_student** and **C_employee**, simply by creating them as instances of this class. Behavior *B_averageAge* would then be applicable to all these additional classes. Our approach provides a good balance between the flexibility of defining class behaviors and the efficiency of grouping common classes under a single m^2-class.

Reflective queries can be expressed naturally in TIGUKAT without any meta-level extensions to the query languages. The reason is that the query model incorporates the behavioral paradigm of the object model and, since the meta-system is uniformly represented by objects with well-defined behaviors, the meta-objects can be used in queries just like any other objects. For example, it is natural (through behavior applications) to express a query that returns the types that define a behavior *B_age* with the same implementation as one of its supertypes. Additional examples include a query that returns a collection of all types with no associated classes (i.e., all abstract types), a query that returns types that define a certain implementation for a certain behavior, and a query that returns the classes that have a greater cardinality than all other collections in the system. Moreover, we can use reflection to infer the result type of a query during its execution. An example reflective query is given in Section 5.

Our meta-class structure is similar to ObjVlisp (Cointe, 1987) and is a generalization of the one-to-one class/meta-class architecture of Smalltalk (Goldberg and Robson, 1989). The generalization of Smalltalk stems from the fact that we do not necessarily define an m^2-class for every class, which is required in Smalltalk. We can group several classes under a common m^2-class. Full details of the reflective features of TIGUKAT and its comparison with other meta-models were presented in Peters and Özsu (1993).

The introduction of the m^2-objects adds a level of abstraction to the model that encapsulates the schema as first-class objects. The benefit of this approach is that the entire model is consistently and uniformly defined within itself. Every object has well-defined behavior and, therefore, we can uniformly apply behaviors to the higher-level objects.

2.6 Temporality

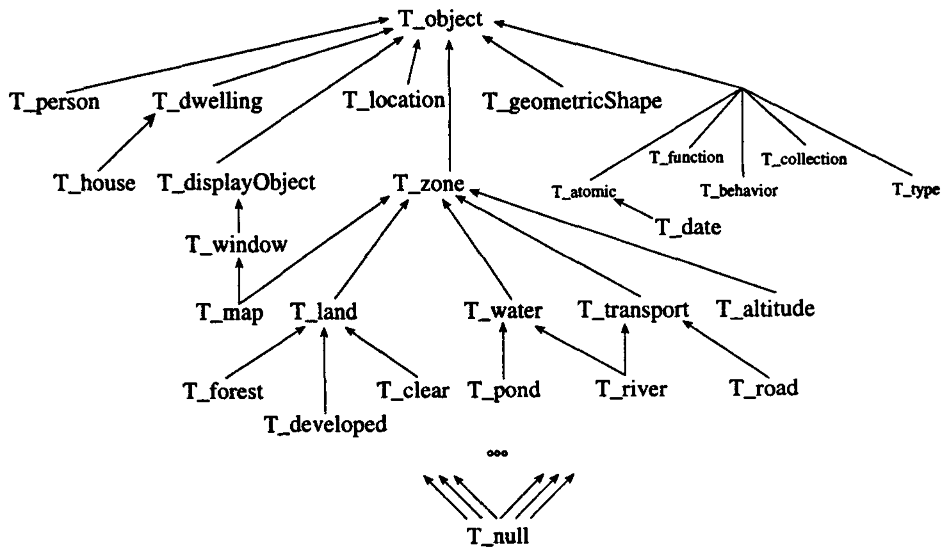
Temporality is introduced into TIGUKAT through an extensible set of primitive *time* types. A rich set of behaviors is defined on these types to model the various notions of time elegantly (Goralwalla and Özsu, 1993).

We use the concept of a *timeline* to represent an axis over which time can be specified. A timeline comprises a collection of time references. A time reference is a means by which time can be specified (e.g., 5 seconds, t3, July 31, [1967,1968], 3 years, 9:17:54:20). We have identified three basic types of time references: a time *instant* (e.g., *moment, chronon*), a time *span* (*duration*), and a time *interval*. These are used to construct instant, span, and interval timelines.

We can model different kinds of timelines depending on (1) their domain (*discrete, dense,* or *continuous*), (2) their boundedness (*bound* or *infinite*), and (3) their ordering (*linear* or *branching*). Any combination of these three features is possible in forming a timeline. This gives applications built on TIGUKAT substantial flexibility in choosing timelines to suit their needs.

Behavior histories are used to manage the properties of objects over time. A subtype of `T_behavior` is introduced to specialize behaviors with temporal qualities for managing histories. Instances of this subtype are called *temporal behaviors*. Temporal behaviors specialize non-temporal behaviors and, thus, encompass all the functionality of non-temporal behaviors. This introduces *temporal transparency* in the sense that a temporal behavior can be used anywhere a non-temporal behavior is expected. In other words, a user unconcerned with temporality can use temporal behaviors as though they were non-temporal. This has the benefit of integrating

Figure 5. Type lattice for a simple geographic information system



temporal applications smoothly into an existing system.

Temporal behaviors can manage independently *valid time* histories (when a value for the behavior is valid) and *transaction time* histories (when a value for a behavior is committed to the objectbase). Our approach adheres to the well-recognized orthogonal nature of the these two times (Snodgrass and Ahn, 1985), and allows us to support valid time, transaction time, and bitemporal models.

2.7 Example System Design

In this section, we present the design of a simplified geographic information system (GIS). This example is used throughout this article to demonstrate various features of TIGUKAT. The GIS example is selected because it usually is listed among the application domains, which require the advanced features offered by object-oriented technology.

A type lattice for a simplified GIS is given in Figure 5. The example includes the root types of the various sub-lattices from the primitive type system, which illustrates their relative position in an extended application lattice. The GIS example defines abstract types that represent information on people and their dwellings. These include the types `T_person`, `T_dwelling`, and `T_house`. Geographic types that store information about the locations of dwellings and their surrounding areas are defined. These include the type `T_location`, the type `T_zone`, along with its subtypes that categorize the various zones of a geographic area, and the type `T_map`, which defines a collection of zones suitable for displaying in a window. Displayable types that present information on a graphical device are defined. These include the types

T_displayObject and T_window, which are application-independent, along with the type T_map, which is the only GIS application-specific object that can be displayed. Finally, the type T_geometricShape defines the geometric shape of the regions representing the various zones. For our purposes we only use this general type but, in more practical applications, this type would be further specialized into subtypes representing, for example, polygons, polygons with holes, rectangles, squares, and splines. Table 1 lists the signatures of the behaviors defined on GIS-specific types.

3. Implementation Considerations

The persistence issues related to the implementation of TIGUKAT are discussed in the next section. In this section, we discuss some of the other issues that arise in the implementation of a uniform and generic object model such as TIGUKAT. There are three issues that we discuss: the implementation of the primitive type system, behavior application, and the implementation of behavioral and implementation inheritance. For more details, see Irani (1993).

3.1 Implementation of Primitive Type System

TIGUKAT is implemented in g++, which is GNU's implementation of C++. However, since TIGUKAT has a generic object model, there is no one-to-one mapping between TIGUKAT types and C++ classes (i.e., we do not create a C++ class for each TIGUKAT type that is defined). Instead, there is a single foundation C++ class, TgObject, which is the principal template for instantiating all TIGUKAT objects. That is, every TIGUKAT object (type, class, behavior, collection, function, instance, atomic, and other primitive or user-defined objects) is an instance of this fundamental C++ class. This approach ensures the uniform representation of all objects in the system, since they may each be treated as an instance of TgObject. The TIGUKAT semantics is embedded within the TgObject structure. Following this approach, the TIGUKAT model is implemented within itself.

From the structural viewpoint, every instance of TgObject comprises an array of records (Figure 6). These can be thought of as the *attributes* (data fields) of that particular instance. TgObject is a dynamic array where each element is either an integer, a character, or TgObject. Integers, reals, and characters are stored directly, while all other objects, including the atomic objects such as sets, strings, bags, lists, and posets, have only their references stored in the slots (this ensures efficient use of memory). For any object, the first slot always contains a pointer to that object's type, which was the template used for its creation. Thus, in line with the model, every object carries knowledge about its type.

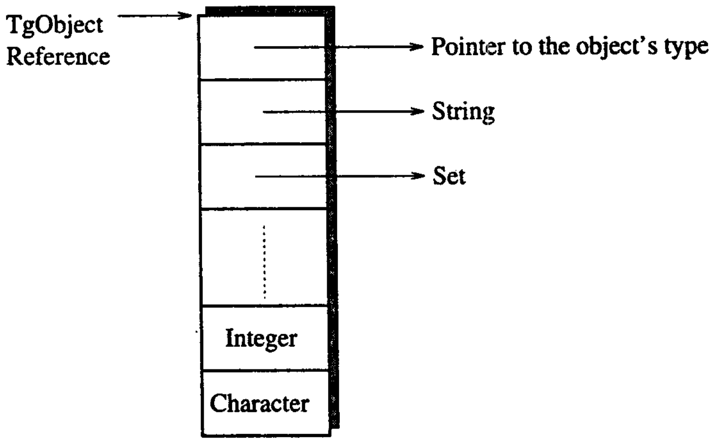
To implement uniform treatment of everything as first-class objects, we have implemented different kinds of C++ object instances in the system (viz., type, class, object, behavior, function, collection, and atomic objects). Although these template instances are all TgObjects, they differ in their structural contents. For

Table 1. Behavior signatures pertaining to example specific types of Figure 5

Type	Signatures	
T_location	*B_latitude:*	T_real
	B_longitude:	T_real
T_displayObject	*B_display:*	T_displayObject
T_window	*B_resize:*	T_window
	B_drag:	T_window
T_geometricShape		
T_zone	*B_title:*	T_string
	B_origin:	T_location
	B_region:	T_geometricShape
	B_area:	T_real
	B_proximity:	T_zone $\rightarrow$ T_real
T_map	*B_resolution:*	T_real
	B_orientation:	T_real
	B_zones:	T_collection<T_zone>
T_land	*B_Pollutants:*	T_collection<T_string>
T_water	*B_volume:*	T_real
	B_Pollutants:	T_collection<T_string>
T_transport	*B_efficiency:*	T_real
T_altitude	*B_low:*	T_integer
	B_high:	T_integer
T_person	*B_name:*	T_string
	B_birthDate:	T_date
	B_age:	T_natural
	B_residence:	T_dwelling
	B_spouse:	T_person
	B_children:	T_person $\rightarrow$ T_collection<T_person>
T_dwelling	*B_address:*	T_string
	B_inZone:	T_land
T_house	*B_inZone:*	T_developed*
	B_mortgage:	T_real

*Behavior was refined from supertype T_dwelling.

Figure 6. Representation of generic `TgObject` structure



example, a type object has a fixed number of slots dedicated for maintaining its information, such as its corresponding class (implemented as a reference to another C++ instance that is a *class* object), its subtypes set (reference to a C++ set instance), and its supertypes. We do not discuss the detailed data structures of each of these objects; we only discuss the structure of *type* objects, since this information is relevant to the subsequent discussion on behavior application and inheritance implementation.

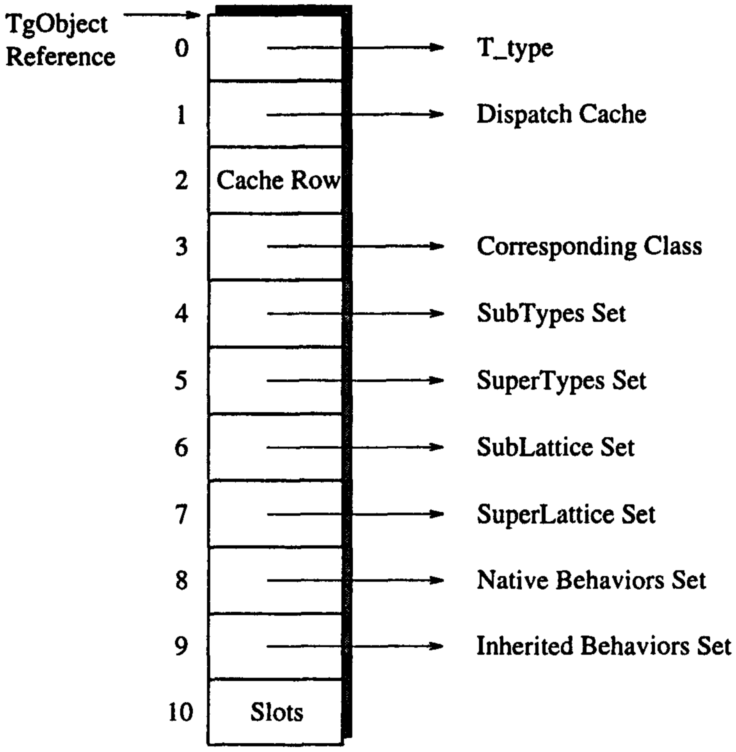
3.2 Behavior Application

Dispatching is the process by which the application of a behavior on an object (message sending) is bound to a particular function (implementation of that behavior). In the event that the applied behavior's implementation is not clearly evident (as a result of subtyping), the right function associated with that applied behavior for the type of the receiver object must be invoked. This requires what is called *dynamic binding*. Thus, behavior application involves the retrieval and application of an appropriate piece of binary code that is contingent on the receiver's type and the selector for that behavior.

Dispatching may be considered a special case of what is called *resolution* (Zdonik and Maier, 1990). Resolution has been defined as a runtime interpretation process that selects a particular value from a possibly ambiguous set of values. Method dispatch (behavior application), hence, seeks to select an appropriate function object (method) whose code needs to be executed, from a set of function objects, each of which implement the same named behavior object over different types. To correctly make this decision, some additional information (actual type of the receiver and the method selector) relevant to the context is required.

Since behavior application is such a fundamental operation in TIGUKAT, it is important to have an efficient dispatch implementation. We have opted for a

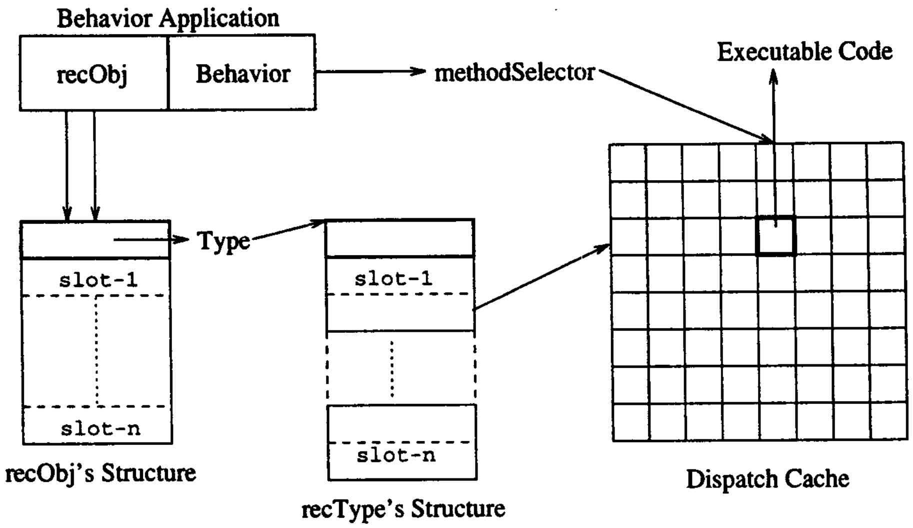
Figure 7. Type object's structure



relatively simple but fast mechanism at the cost of bearing the consequential memory overhead. The system maintains a *dispatch cache*, which consists of a slot for each behavior-type pair that exists in the system. This cache is a statically allocated volatile structure that needs to be reinitialized on program startup. The size of this lookup table is proportional to the total number of unique behaviors in the system, and to the total number of types in existence. We sacrifice memory usage for quick response time during execution (André and Royer, 1992; Dixon et al., 1989), but an incremental coloring algorithm could be used to drastically reduce memory consumption. We have not implemented this optimization in the current version of TIGUKAT.

Each entry in the dispatch cache is a function pointer to some executable code, which implements that behavior (column) for the concerned type (row). Every unique behavior has a unique integer mapping associated with it. This integer mapping, the *method selector*, provides access to the appropriate column of the cache. That column is said to "belong" to the behavior. The addresses stored in the slots of this column may be different or identical, depending on which of the subtypes have inherited the same implementation of that behavior, and which have had that behavior redefined, overridden, or *reassociated* (associated with a different function). The process of filling the cache row with appropriate values during the

Figure 8. Behavior application process



creation of a new type has been termed *implementation inheritance*, and our system handles it automatically up to a certain degree of complexity as discussed in the next subsection.

Behaviors may be reassociated with functions at any time (redefinition of behaviors), which makes it imperative that we support the dynamic binding of behaviors and perform dispatch *on the fly*. Although it is evident that static (compile time) dispatching is more efficient (Cattell, 1991), this seldom will be possible in our system. The reference to an object of a particular type may potentially be referencing an object of any of this type's subtypes. The ambiguity about which function should be invoked can only be resolved at runtime when knowledge becomes available about which type's instance is being referenced. Thus, the actual type of a receiver object needs to be identified prior to function execution. We note that, although dynamic binding might render static type checking difficult, it does not entirely preclude it.

The behavior application process for computed functions in TIGUKAT involves the following procedure. With reference to Figure 8, given an object, say *recObj*, as the receiver of a particular message, we extract its type, say *recType*, which is readily available since every object knows its type. All types have knowledge of their unique cache row (Figure 7). From the applied behavior object we extract the *methodSelector.* This integer value indexes into a unique column in the dispatch cache. The slot in the determined row and column contains the address of the function code to be executed. The list of arguments passed to the behavior is supplied to the function after relevant type checking is done. Behavior application is conveniently reduced to the execution of a single line of code:

JMP (*recObj* ⟶ *recType* ⟶ *dispatchCache* [*methodSelector*])

where *recObj* is a pointer to the object on which the behavior is to be applied (receiver object reference), *recType* is the receiver object's type, *dispatchCache* is the matrix of executable addresses, and *methodSelector* gives access to the appropriate column in the dispatch cache. Therefore, the two basic requisites for binding an executable piece of code to the applied behavior at runtime are the type of the receiver object and the method selector for the behavior.

3.3 Behavioral and Implementation Inheritance

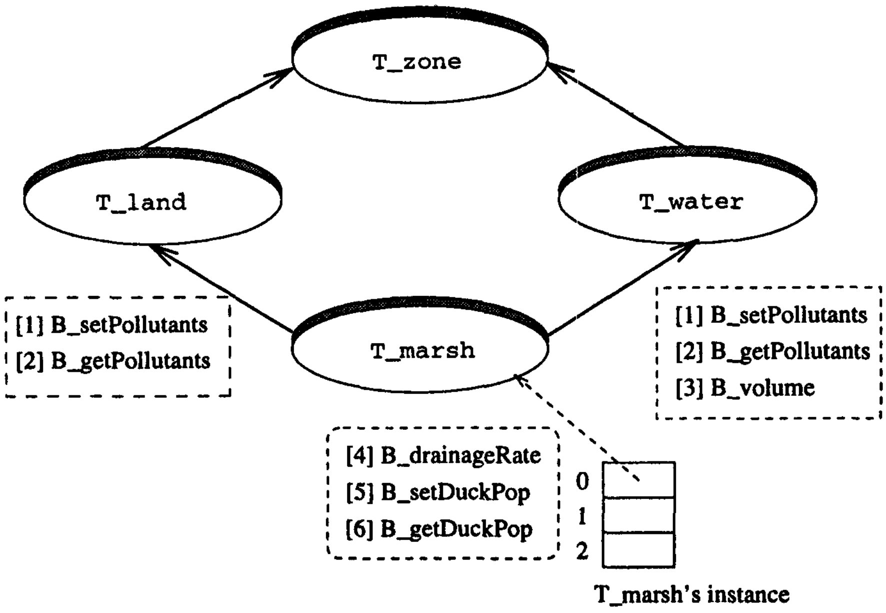
As indicated in Section 2.1, two kinds of inheritance are supported by TIGUKAT: *behavioral* and *implementation* inheritance. The implementation strategy for behavioral inheritance (subtyping) involves taking the union of the interface sets of all the types declared as immediate supertypes of the new type being created. This set forms the contents of the new type's *inherited* set, and comprises the minimum set of behaviors to which all objects of this type should conform. The nature of the functions with which these behaviors have been associated is of no consequence to the behavioral inheritance mechanism. The implemented algorithm iterates through the relevant interfaces, and selects all the behaviors with unique signatures as candidates for insertion into the new type's inherited set. This is a relatively straightforward technique.

Implementation inheritance facilitates code reuse by ensuring that all code is at a level where the maximum number of types can share it (Atkinson et al., 1989). If only single inheritance is present, the inherited set of the new type is precisely the contents of the interface set of its sole supertype. No conflict resolution is necessary, and all entries in the dispatch cache and the supplementary cache are merely duplicated in the row allocated for the new type for the complete set of inherited behaviors. This implies that all implementations (function addresses) for the inherited set of behaviors are inherited, too. However, the type implementor[7] has the liberty to reassociate any or all of these inherited behaviors.

With multiple inheritance, the situation is more complex, because conflict resolution has to take place. Figure 9 depicts an inheritance graph with multiple subtyping. The arrows indicate a subtyping relationship from the tail to the head, and the dotted arrow indicates an instance of the type. The dashed boxes contain the interface sets of the corresponding types, while the matrices DC and SC are the dispatch cache and the auxiliary cache, respectively. The auxiliary cache SC is a bit cache that records whether a function is stored or computed. Execution of

7. We identify three classes of users: The *type specifier* is the person who designs the inheritance hierarchy for the user application. The *type implementor* is the one who actually implements this required hierarchy using TDL. The *end user* refers to the person or application program that may query the existing system and instantiate new objects, but may not be authorized to modify the existing type structure.

Figure 9. Implementation inheritance requiring conflict resolution



	1	2	3	4	5	6
T_zone						
T_land	1	1				
T_water						
T_marsh	1	1			2	2

DC

	1	2	3	4	5	6
T_zone						
T_land	s	s				
T_water	c	c	c			
T_marsh	s	s	c	c	s	s

SC

the stored function simply sets or gets the contents of one of the receiver's slots without executing any code. In this case, the slot number, rather than the function address, is stored in DC.

Consider the GIS example that we introduced earlier. We create a new type `T_marsh` as a subtype of `T_land` and `T_water`, with the native behaviors *B_drainageRate* (to calculate the rate of water leaving or entering the marsh) and *B_DuckPop* (to store the population of ducks in the marsh). This inheritance structure has a clash in behaviors that the system is unable to resolve automatically, and requires the type implementor's intervention. The conflict resolution policy fails because the behaviors *B_setPollutants* and *B_getPollutants* are defined in the interfaces of both the direct supertypes (`T_land` and `T_water` are immediate supertypes of `T_marsh`, and have conflicting implementations associated in each of these types, being computed in `T_water` but stored in `T_land`, as depicted in auxiliary cache SC). We have assumed that the type implementor opted for the stored implementations to be inherited and therefore each instance of `T_marsh` requires a total of three slots:

slot 0 holds the reference to the type, slot 1 holds the reference to the collection of pollutants, and slot 2 holds the value of the duck population.

We iterate over each of the behavior objects in the inherited interface of `T_marsh` generated during behavioral inheritance. If a behavior exists in only one supertype's interface, this signifies a conflict-free condition; thus, no conflict resolution is required. The implementation for that behavior may be safely inherited together with its associated function (stored or computed). The appropriate entry in the supplementary cache, indicating a stored or computed association, is inserted. If the association is with a computed function, then the address of that function is also inserted into the dispatch cache. All the stored functions will possess a NULL entry in the dispatch cache until class creation time. At that time, slots will be assigned to all the stored functions, one slot per pair of set-get accessors. This may require a reallocation of slots to behaviors, which is entirely system managed.

For each conflicting behavior, the conflict resolution policy has to be applied. The supplementary cache values for that behavior are examined. If they happen to indicate a computed function for all the conflicting supertypes, the values of the addresses of the functions from the dispatch cache are examined. If these are identical for each of the types in the set of conflicting supertypes, then this behavior's implementation is safely inherited and the corresponding address is inserted into the dispatch cache. A *computed* indication is placed in the supplementary cache.

If the conflicting behavior is implemented by a stored function in all the supertype entries, the corresponding value of `T_function` is examined for each type. If these match, then a *stored* indicator is placed in the supplementary cache, and a NULL is entered into the dispatch cache. Recall that, for all the stored functions, the dispatch cache will hold the corresponding slot number to access (an identical value for each paired set-get) instead of the address of the executable code. These slot numbers will be inferred and allocated during class creation only, at which time it will be possible to determine the total number of all the associated stored functions.

In the event that an inherited behavior is associated with a stored function in one of the supertypes, and a computed function in another, or if there is mismatch in the values of function pointers, then no conflict resolution is possible by the system, and a NULL is entered in both caches. It is the type implementor's responsibility to associate this behavior with an appropriate implementation of his choice, or to specify which of the supertype's implementations is to be inherited. A message-requesting intervention will be displayed. The cache values for this behavior must be inserted (i.e., each behavior must be associated with some function) before class creation so that the newly established type is considered functionally complete.

4. Persistence Model

A fundamental decision governing OBMS implementation is the strategy employed for managing persistent objects. *Persistence* is defined as the ability of an object to survive across multiple application program executions, and a *persistent object* is one that has this property. The persistence model of TIGUKAT adheres to the following principles:

1. *Persistence is transparent to the user.* TIGUKAT Query Language (TQL) and TIGUKAT Control Language (TCL) provide a declarative specification for indicating that an object is persistent. Users do not perform any explicit input/output operations, and they do not open and close files. TIGUKAT coordinates with the low-level storage manager to provide persistence transparently.
2. *Persistence is orthogonal to the type of an object* (Atkinson and Buneman, 1987). A type can be made persistent or transient. The instances of a type can be either persistent or transient. The only dependency is that, if an object is made persistent, then its type must also be made persistent because an object cannot exist without a type. These are described as persistence side-effects (PSEs) below.
3. *Persistence is independent of the query model* (Atkinson and Buneman, 1987). Queries do not differentiate between transient and persistent objects. Both are queried in a uniform way, using the same language constructs. This principle is followed in the development of a programming language interface to TIGUKAT.

Five basic approaches to persistence have been identified (Zdonik and Maier, 1990). The first strategy requires that a decision about persistence be made prior to object creation. Depending on whether a persistent or transient object is needed, an appropriate object creation routine is invoked on the object. Thus, there are separate routines for creating transient and persistent objects.

The second approach is called *reachability*-based persistence. This methodology, pioneered in PS-Algol (Atkinson et al., 1983) and incorporated by O_2 (Bancilhon et al., 1992), requires that persistent objects *hang* off a persistent root via a direct or indirect reference. When an object *o* is made persistent, all objects in the transitive closure (i.e., reachable from *o*) are made persistent. Object *o* becomes a root for persistence. In this scheme, every object reachable from a root is made persistent or transient when the root is made persistent or transient.

The third approach is *allocation*-based persistence. This approach restricts the persistence of an object by requiring it to be allocated within a persistent container (collection) during object creation. This requires the existence of a persistent storage space with variables naming locations within that space. Objects written into persistent variables are guaranteed to be persistent as long as they are maintained in the persistent variable. ObjectStore (Lamb et al., 1991) takes this approach, although it renders garbage collection difficult due to the dangling references problem.

The fourth approach is *type*-based persistence where some types are declared to be persistent, and an object is persistent if it is an instance of a persistent type. The E language (Richardson and Carey, 1989; Richardson et al., 1989; Schuh et al., 1990) uses a similar approach, and maintains a parallel hierarchy of persistent and corresponding non-persistent types.

The fifth approach (which we follow) associates persistence with individual objects, and requires explicit declaration of persistence, which may occur anytime during an object's existence. We define primitive behaviors *B_persistent* and *B_transient* on `T_object`, which are applicable to all objects in the system. These behaviors coerce the receiver object to be persistent or transient, respectively. The TIGUKAT user languages provide declarative constructs for making individual objects or collections of objects persistent or transient. The system translates these requests to applications of *B_persistent* or *B_transient* on the affected objects.

We opted for object-based persistence because it best maintains the uniformity of object access, and does not restrict the use of types for persistent or non-persistent purposes. Any object created during a session (either a query session or an application program execution session encapsulated as a transaction) can be explicitly made persistent (or transient) at any time during the session. Thus, all TIGUKAT objects are *potentially persistent.*

The support for persistence is a behavioral extension to the model. Behaviors *B_persistent* and *B_transient* are added to the type `T_object` and, thus, are applicable to any object. This clarifies the fact that all objects are potentially persistent (or transient) in TIGUKAT. The language constructs for persistence in TQL and TCL invoke these behaviors.

TIGUKAT queries operate on collections and return collections as results. Since collections are objects, we permit the existence of persistent as well as transient collections, which can contain a mixture of both transient and persistent member objects. The transient members of a collection must cease to exist at the end of a particular session, even if the collection is persistent. On the other hand, the persistent members of a collection must continue to exist in their respective class extents after a particular session ends. This is true even if the collection is transient and ceases to exist at the end of a session. This does not cause a problem, because the persistent objects in a transient collection reside in the (persistent) class associated with the type of these objects. Thus, these objects are available following the session even if the collection is not. All collections generated as a result of query execution are initially transient. The semantics of handling each case of transience and persistence of objects, collections, classes, and types are described by implementations for the *B_persistent* and *B_transient* behaviors, which we discuss below.

Coercing an object to be persistent could result in *persistence side-effects* (PSEs), which propagates persistence to type and class objects related to the original object. The *persistence matrix* (Figure 10) depicts the various alternative strategies involved in making a TIGUKAT object persistent. Reading across rows, a "+" entry indicates a PSE, while a "−" entry indicates PSE-free persistence (the diagonal entries are not a concern and, therefore, are PSE-free). Making a *type* persistent is PSE-free. Its corresponding class and instances, if they exist, are not required to be persistent.

Figure 10. The persistence matrix

	Type	Class	Inst
Type	×	−	−
Class	+	×	−
Inst	+	+	×

If a *class* object is made persistent, a PSE occurs, which makes its corresponding type persistent as well. However, the instances of this class do not need to be made persistent. The final case is when a particular instance object is made persistent. This causes PSEs that make both its class and its corresponding type persistent. This protects against the object being stored as a persistent instance of a transient type and, sometime later, being erroneously accessed as an instance of a non-existent type (if its transient type disappears in the meantime). The primitive types, classes, behaviors, and functions are, by default, perpetually persistent and cannot be deleted. This is necessary for the integrity of the system.

The complement of the persistence matrix is the *transience matrix* (not shown). This matrix is derived from the repercussions of making persistent objects transient (by applying a behavior *B_transient* defined on `T_object` for example). The effects are precisely the opposite of those described in the persistence matrix (i.e., making an instance transient will not affect its type or class; making a class transient does not affect its corresponding type, but all its instances will be made transient; and making a type transient will make its corresponding class and all its instances transient).

This model of persistence is fairly low-level, and the referential integrity among objects is a problem to consider. In particular, when a transient object disappears, how are dangling references to this object handled?

One approach is to offload the responsibility onto the application programmer, who must update references to transient objects before the end of a session. This approach is unacceptable for obvious reasons. Another approach is to use reachability persistence, which disallows persistent objects to reference transient ones, since, when an object is made persistent, the transitive closure of all objects reachable from that object are made persistent as well. Reachability-based persistence is not useful in a uniform model like TIGUKAT because, conceptually, all objects in the entire objectbase are reachable from any object. Consider an arbitrary object. Since every object knows its type, the type of the object is reachable, and must be made persistent. Every type knows its class and, therefore, the class is reachable, and should be made persistent. Every class knows its instances, and all instances of that class are made persistent. Every type knows it supertypes and subtypes and, thus, the class/instance persistence propagates over the entire lattice and makes all objects persistent.

The approach that we use has the net effect of transforming transient objects into perpetually persistent undefined objects at the end of a session (or transaction).

This always is safe because undefined is an instance of `T_null`, which is a subtype of all types. The substitutability property allows us to use undefined anywhere an instance of a supertype is used.

Operations on a TIGUKAT objectbase occur within a given user session (which will be modeled as a transaction when the programming language is developed). A session defines a scope for the transience of objects. There are **save** (commit) and **quit** (abort) statements that can be used in a session. In this sense, a session serves as a simple, flat transaction model. At the end of a session, all transient objects are logically replaced by the perpetually persistent undefined object. This can be efficiently implemented by pointer swizzling. That is, we modify the OID mapping so that it appears as though the transient object was written to stable storage at the location where the persistent undefined object exists. Then, all persistent objects that referenced the transient object will now reference the persistent undefined object and there will be no dangling references.

In this approach, there is the potential for wasted stable storage when a persistent object is made transient. The transformation to the persistent undefined object occurs as usual, but we must somehow reclaim the storage occupied by the object when it was persistent. With a central OID to disk address mapping we can simply update this mapping, and reclaim the storage immediately. If, however, objects hold the disk addresses directly, then there may be other persistent objects that reference the old disk address and we cannot simply reclaim the space without updating these references. In this case, a garbage collector can be used to manage reference counts and reclaim the storage after all references have been updated. In the meantime, the storage must be transformed into a persistent undefined object so that objects referencing it will not see the old persistent object, but rather the undefined object. This transformation easily is implemented by encoding the information in the header of the old object on disk.

Our approach to single-object persistence and the maintenance of the PSEs are described in the *F_makePersistent* and *F_makeTransient* functions below that serve as implementations for the *B_persistent* and *B_transient* behaviors defined on `T_object`.

```
F_makePersistent (o)
    This is the implementation of the B_persistent behavior defined on T_object.
    INPUT: An object o to be made persistent.
    if object o is transient then
        Call storage manager to write o to stable storage and update log
        Apply B_persistent to the type and class of o:
          o. B_mapsto. B_persistent
          o. B_mapsto. B_classof. B_persistent
        if object o is a class then
          Apply B_persistent to the associated type of the class:
            o. B_typeof. B_persistent
        endif
    endif
```

The recursion of the *F_makePersistent* implementation is ended by making primitive T_type and primitive classes **C_type** and **C_class-class** perpetually persistent. T_type and **C_type** represent the end of the type chain, while **C_class-class** represents the end of the meta-class chain. Note that these are the minimal primitive persistent objects. In practice, the recursion is ended much sooner, because many more primitive objects are perpetually persistent (like **C_class**, for example).

At commit time (or the end of a session), the transaction management facility ensures that persistent objects are written out to stable storage. No changes are made to persistent objects with respect to references to transient objects. Dangling references are avoided by the transformation described above.

The implementation for the *B_transient* behavior is as follows:

F_makeTransient (*o*)

This is the implementation of the *B_transient* behavior defined on T_object.

INPUT: An object *o* to be made transient.

```
if object o is persistent then
    Call the storage manager to mark object o transient and update log
    if object o is a class then
      Apply B_transient to every member in the shallow extent of the class
    endif
    if object o is a type then
      Apply B_transient to the associated class of the type:
        o.B_classof.B_transient
    endif
endif
```

At commit time, all transient objects are replaced by the persistent undefined object. This ensures that there will be no dangling references to the transient objects, because persistent objects that reference the transient object will now reference the persistent undefined object.

The explicit deletion semantics for persistent and transient objects are closely related to the *F_makeTransient* implementation and the transient-to-undefined object transformation. The reason is that, when an object is explicitly deleted, there is still the problem of dangling references to consider. The *B_drop* behavior defined on T_object can be used to explicitly delete an object. The deletion semantics is related to schema evolution when the object to be dropped is part of the schema (i.e., a type, class, collection, behavior, or function). Schema evolution is beyond the scope of this article, but is addressed in Peters (1994).

The only difference in the transient object and deleted object semantics is the timing of events. When an object is deleted (whether it be transient or persistent), it is immediately replaced by the persistent undefined object, rather than at the end of a session, as is the case for transient objects. A simplified implementation of the *B_drop* behavior for deleting objects is defined as follows and the similarities to *F_makeTransient* are apparent:

```
F_deleteObject (o)
    A simplified implementation of the B_drop behavior defined on T_object.
    INPUT: An object o to be deleted.
    Call the storage manager to mark object o as deleted and update log
    if object o is a class then
        Apply B_drop to every member in the shallow extent of the class
    endif
    if object o is a type then
        Apply B_drop to the associated class of the type:
          o.B_classof.B_drop
    endif
    Perform schema evolution operations if o is a schema object
```

The single-object persistence approach can be transitively applied to all objects referenced by the object being made persistent. This can proceed to any number of levels until the transitive closure is reached. Thus, we can identify the boundaries for the transitive application of persistence. The lower bound is when only a single object is made persistent (our approach). The upper bound is when all objects in the transitive closure are made persistent (reachability persistence). In a finite objectbase, there are a finite number of levels between these two boundaries. We call the lower limit *0-persistence*, the upper limit *n-persistence*, and any level between these two *i-persistence*. For example, the **persistent all** construct of TQL and TCL performs *1-persistence* on a collection argument. That is, the collection and all of its members (1 level of reference) are made persistent. We show that, in a uniform model like TIGUKAT, the transitive closure from any object is the entire objectbase and, so, *n-persistence* is not useful.

5. Query Model and Language

An identifying characteristic of the TIGUKAT query model is that it is a direct extension to the object model. In other words, it is defined by type and behavior extensions to the primitive model. We define a type `T_query` as a subtype of `T_function` in the primitive type system. This means that queries have the status of *first-class objects*, and they inherit all the behaviors and semantics of objects. Moreover, queries are functions and can be used as implementations of behaviors, they can be compiled, and they can be executed.

Incorporating queries as a specialization of functions is a natural and uniform way of extending the object model to include declarative query capabilities. The major benefits of this approach are as follows:

1. Queries are *first-class objects*, meaning they support the uniform semantics of objects, they are maintained within the objectbase as another kind of object, and they are accessible through the behavioral paradigm of the object model.
2. Since queries are objects, they can be queried, and can be operated on by other behaviors. This is useful in generating statistics about the performance of queries, and in defining a uniform extensible query optimizer.
3. Queries are uniformly integrated with the operational semantics of the model and, thus, queries can be used as implementations of behaviors (i.e., the result of applying a behavior to an object can trigger the execution of a query).
4. The query model can be extended by subtyping `T_query`. This can be used to specialize the notion of queries into additional types that can be incrementally introduced and developed as new kinds of queries are discovered. For example, we subtype `T_query` into `T_adhocQuery` and `T_productionQuery` and then define different evaluation strategies for both in the query optimizer. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies while production queries are compiled once and executed many times.

The languages for the query model include a complete object calculus, an equivalent object algebra, and an SQL3-like user language. The TIGUKAT object calculus is a first-order predicate language. Predicates of the calculus are defined on collections (essentially sets) of objects, and a calculus expression returns a collection of objects as a result. This gives the language *closure*. The calculus includes a function symbol for behavior evaluation to incorporate the behavioral paradigm of the object model. This allows the specification of *path expressions* (or *implicit joins*) in calculus formulas. The calculus is object-creating, and supports a controlled creation and integration of new collections, classes, types, and objects into the existing schema.

The safety of the calculus is based on the *evaluable* class of queries (Gelder and Topor, 1991), which is arguably the largest decidable subclass of the domain independent class (Makowsky, 1981). We extend this class by making use of equivalence ($=$) and membership ($\in$) operators in queries for *object generation*. This alleviates the need for explicit *range* specifications for those variables that can be generated from the given operators.

The TIGUKAT object algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. Algebraic operators are modeled as behaviors on the primitive type `T_collection`. Like the calculus, the algebra is *closed* in that every algebraic operator works on collections and returns a collection as a result.

The operators of the algebra include typical set operations, a collapse operator for flattening nested collections, a select for returning objects that satisfy a predicate, an operator for applying a series of behaviors to a collection of objects, an operator to project behaviors, an operator for unconditionally combining objects, a join for combining objects based on a join predicate, a generating join for producing objects from other objects and joining the generated objects with the ones from which they were generated, and a reduction operator for separating joined objects into their original components.

The first-order expressiveness of the calculus, its safety, as well as the equivalence of the calculus and algebra were proven elsewhere (Peters, 1994; Peters et al., 1993*b*). In this context, a calculus expression is considered safe if it can be evaluated in finite time, and produces finite output (Ozsoyoglu and Wang, 1989; Peters, 1994). The remainder of this section describes the user language of TIGUKAT, with a focus on its constructs for managing persistence and querying the objectbase.

The main function of the TIGUKAT language is to support the definition, manipulation, and retrieval of persistent (and transient) objects in an objectbase. The language consists of three parts: the TIGUKAT Definition Language (TDL), which supports the definition of meta-objects (i.e., types, collections, classes, behaviors, and functions), the TQL, which is used to manipulate and retrieve objects, and the TCL, which supports the session specific operations (e.g., open, close, save). We focus on TQL and TCL in this article (the complete specification of all three languages was given in Peters et al., 1993*b*; Lipka, 1993).

TQL has a syntax based on the SQL3 *select-from-where* structure, and a formal semantics dictated by the TIGUKAT object calculus. Thus, TQL combines the power of declarative query languages with object-oriented features in the form of the international data-speak of SQL. The broad acceptance of SQL as a standard query language in relational databases, together with the current efforts on SQL3 to extend the syntax and semantics with object-oriented features (Gallagher, 1992) are the main motivations for our SQL basis.

The semantics of TQL is defined in terms of the object calculus. In fact, there is a complete reduction from TQL to the object calculus (Lipka, 1993). In addition, TQL accepts path expressions (implicit joins; Kim et al., 1989) in the *select, from,* and *where* clauses. Object equality is defined on the primitive type `T_object`, thus explicit joins are also supported by TQL. The results of queries can be queried, since queries operate on collections, and always return a finite collection as a result. Queries can be used in the *from* and *where* clauses of other queries (i.e., nested queries). Objects can be queried regardless of whether they are persistent or transient.

Note that the syntax for the application of aggregate functions is not explicitly supported in the current implementation of TQL. However, because the underlying model is purely behavioral, these functions are defined as behaviors on the `T_collection` primitive type, and can be applied to any collection including those returned as a result of a query.

TQL consists of the four basic operations: **select**, **insert**, **delete**, and **update**, along with three binary operations: **union**, **minus**, and **intersect**. In this article, we only discuss the *select, union, minus,* and *intersect* statements.

The basic query statement of TQL is the select statement, which has the following syntax.[8]

<select statement>: **select** *<object variable list>*
[into [persistent [all]] *<collection name>***]**
from *<range variable list>*
[where *<boolean formula>***]**

The *select clause* in this statement identifies objects to be returned in a new collection. There can be one or more object variables of different formats (constant, variables, path expressions, or index variables) in this clause. They correspond to the free variables in an object calculus formula. The *into clause* declares a reference to a new collection that will hold the result. This collection optionally can be made persistent by specifying the **persistent** keyword. This does not make the members of the collection persistent; to do this, the **all** keyword must be specified as well. If the *into clause* is not specified, a new transient collection is created. There is no reference to this collection, and it disappears at the end of a query. In this case, the result cannot be retained for later use by another query. It can be printed only to the screen, for example. The *from clause* declares the ranges of object variables in the *select* and *where* clauses. Every object variable can range over an existing collection or a collection returned as the result of a subquery. A subquery is a nested *select-from-where* clause that can be given explicitly or specified as a reference to an existing query object. A range variable statement in the from clause is as follows:

<range variable> : *<identifier list>* **in** *<collection reference>* **[−]**
<collection reference>: *<term>* | **(***<query statement>***)**

The collection reference in the range variable definition can be followed by a minus −, which refers to the shallow extent of a class, which is a collection of objects[9] The default is the deep extent for classes. The *term* in the collection reference definition is either a constant reference, a variable reference, or a path expression.

The *where clause* defines a boolean formula that must be satisfied by the objects returned by a query. Boolean formulas have the following syntax:

8. The notation used throughout this section is as follows: all bold words and characters correspond to terminal symbols of the language (e.g., keywords, special characters). Nonterminal symbols are enclosed between < and >. Vertical bar | separates alternatives. The square brackets [ ] enclose optional material which consists of one or more items separated by vertical bars.

9. In earlier articles, we used the plus + sign for the shallow extent. However, it was pointed out to us by the referees that this was counter-intuitive, because the shallow extent actually reduces the cardinality of the range. We have, therefore, changed the symbol to −.

<boolean formula>: *<atom>*
| **not** *<boolean formula>*
| *<boolean formula>* **and** *<boolean formula>*
| *<boolean formula>* **or** *<boolean formula>*
| (*<boolean formula>*)
| *<exists predicate>*
| *<forAll predicate>*
| *<boolean path expression>*

where an atom is defined as follows:
<atom>: *<term>* = *<term>* | *<identifier>* ∈ *<term>*

and a *term* is either a variable reference, a constant reference, or a path expression.

Two special predicates are added to TQL boolean formulas to represent existential and universal quantification. The existential quantifier is expressed by the *exists predicate* of the form:

<exists predicate>: **exists** *<collection reference>*

The *exists predicate* is true if the referenced collection is not empty. The universal quantifier is expressed by the *forAll predicate*, which has the following structure:
<forAll predicate>: **forAll** *<range variable list>* *<boolean formula>*

The syntax of the *range variable list* is the same as in the *from clause* of the select statement. It defines variables that range over specified collections. The boolean formula is evaluated for every possible binding of the variables in this list. Thus, the entire *forAll predicate* is true if, for every element in every collection in the range variable list, the boolean formula is satisfied.

The last component of the boolean formula definition is the *boolean path expression* defined simply as:

<path expression> = **TRUE/FALSE**

To avoid such an artificial construct, we include a boolean path expression in the definition of a TQL formula under two conditions. First, all invoked functions are assumed to be *side-effect-free* (which is a common assumption in many object query models) and, second, the result type of the entire path expression must be a boolean type.

There is a sizeable literature on object query models and languages. This continues to be an active area of research with many language and query model definitions. We do not provide a detailed comparison of our model and language with others. We refer the interested reader to Mitchell et al. (1993) for an overview of these languages and models.

The following queries on the GIS example objectbase illustrate the expressive constructs of TQL and how the persistence of results are specified.

Example 1. Return the zones that are part of some map, and are within 10 units from water. Project the result over *B_title* and *B_area*. Place the result into a persistent collection called **L_floodZones** and make all members persistent.

> **select** *o* [*B_title, B_area*]
> **into persistent all L_floodZones**
> **from** *p* **in C_map,** *o* **in** *p. B_zones* (), *q* **in C_water**
> **where** *o. B_proximity* (*q*) $<$ 10

Example 2. Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map. The result is a transient collection that disappears at the completion of the query.

> **select** *p, q. B_title* ()
> **from** *p* **in C_person,** *q* **in C_map**
> **where** *p. B_residence* ().*B_inZone* () $\in$ *q. B_zones* ()

The following is an example of a reflective query and illustrates that no new constructs are needed in the language to query the schema.

Example 3. Return the types that define the behavior *B_age* with the same implementation as one of its supertypes. Place the result into a persistent collection called **L_inheritedAgeTypes**, but do not make the members persistent.

> **select** *t*
> **into persistent L_inheritedAgeTypes**
> **from** *t* **in C_type,** *r* **in** *t. B_supertypes*()
> **where** *B_age* $\in$ *t. B_interface* () **and** *B_age* $\in$ *r. B_interface* ()
>   **and** *B_age. B_implementation* (*t*) = *B_age. B_implementation* (*r*)

TQL also supports three binary operations: **union**, **minus**, and **intersect**. The syntax of these statements is specified below. The <*collection reference*> field can be specified as a subquery or as a reference to an existing (transient or persistent) collection.

> <*collection reference* > **union** <*collection reference* >
> <*collection reference* > **minus** <*collection reference* >
> <*collection reference* > **intersect** <*collection reference* >

TQL has a proven equivalence to the formal languages, making it easy to perform logical transformations and argue about its safety. The theorems and proofs of equivalence were given in Lipka (1993).

The TIGUKAT control language (TCL) defines statements for controlling operations within an objectbase session. In the absence of a computationally complete programming language, TCL serves to provide a scope for execution and interaction with a TIGUKAT objectbase.

Since everything in TIGUKAT is treated as a first-class object, sessions are also represented by objects. Specifically, session objects are instances of `T_session` type, which is a direct subtype of `T_object`. Every TIGUKAT user has at least one instance of `T_session`, which is referred to as a *root session*. Other sessions can be opened and manipulated from this session by issuing session-specific TCL operations: **open**, **close**, **save** (commit), and **quit** (abort). TCL also provides an assignment statement for creating object references, as well as two forms of a *persistent* operation whose syntax is as follows:

1. **persistent** *<object reference list>*
2. **persistent all** *<collection reference>*

The semantics of the first form is to make every object in the given object reference list persistent according to the rules defined in Section 4. The second form requires the argument to be a collection. The semantics is to make the collection persistent and all of its members persistent as well.

The inverse operations of the **persistent** statements are the **transient** statements whose syntax is as follows:

1. **transient** *<object reference list>*
2. **transient all** *<collection reference>*

6. Other DBMS Functionalities

In addition to the powerful object and query models that TIGUKAT provides, the system is enhanced by a number of other functions commonly associated with DBMSs. In this section, we provide a brief overview of three functions that have been under development: query optimizer, the versioning scheme, and the transaction manager.

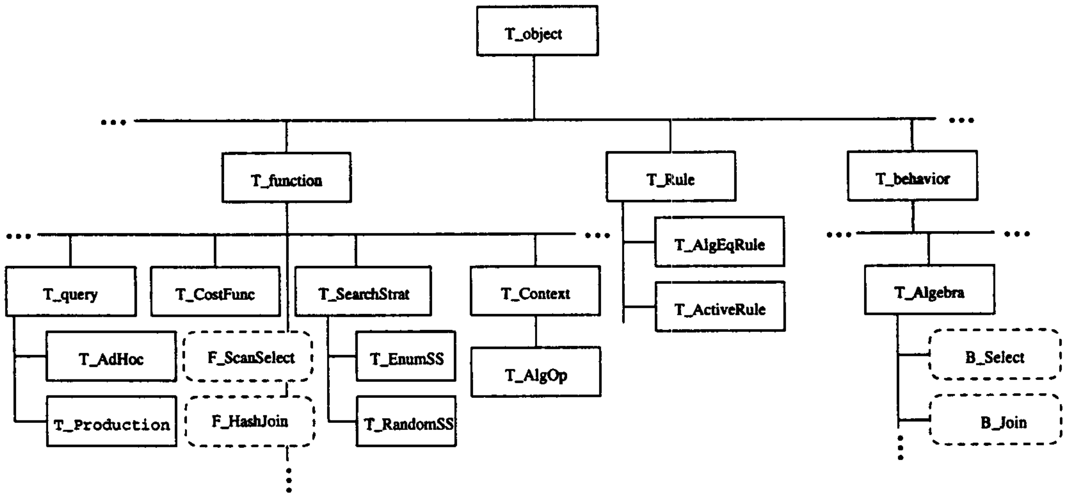
6.1 Query Optimizer

The goal of query optimization is the choice of the "optimum" execution plan for a query from a set of equivalent execution plans specified as algebraic expressions. The set of equivalent execution plans are obtained by the application of algebraic transformation rules and the optimum strategy is the one with the lowest cost according to a cost function. Thus, to characterize a query optimizer, three things need to be specified:

1. The transformation rules that generate the alternative query expressions, which constitute the *search space*;
2. A *search algorithm* that allows one to move from one state (i.e., execution plan) to another in the search space; and
3. The *cost function* that is applied to each state.

The TIGUKAT query optimizer (Muñoz, 1993) follows the philosophy of representing system concepts as objects, and is along the lines of Lanzelotte and Valduriez (1991). The search space, the search strategy, and the cost function are modeled as objects

Figure 11. Optimizer as part of type system



(Figure 11). The incorporation of these components of the optimizer into the type system provide extensibility via the basic object-oriented principle of subtyping and specialization.

The states in the search space are modeled as processing trees (PT) whose leaf nodes are references to collections, and whose non-leaf nodes denote behavior applications whose results are other objects. Those nodes that correspond to algebraic operator behaviors return temporary collections as result.

Algebraic operators (e.g., *B_Select, B_Join*) are defined as behaviors of the `T_collection` type. They are modeled as instances (shown as dashed boxes in Figure 11) of type `T_algebra`, which is a subtype of type `T_behavior`. The implementation (execution) algorithms for these algebraic operators are modeled as function objects (e.g., *F_HashJoin, F_ScanSelect*). These implementation functions cannot be used as nodes of a PT, since these nodes should represent execution functions all of whose arguments have been marshalled. Therefore, `T_AlgOp` is defined as a type whose instances are functions with marshalled arguments, and represent nodes of a PT. In this fashion, each node of a PT represents a specific execution algorithm for an algebra expression.

Search strategies are similarly modeled as objects, but separate from the search space. `T_SearchStrat` is defined as a subtype of type `T_function` and it can, in turn, be specialized. Figure 11 shows the specialization of `T_SearchStrat` into enumerated search strategies `T_EnumSS` and randomized search strategies `T_RandomSS`. The algebraic transformation rules that control the movement of the search strategy through the search space are implemented as instances of `T_AlgEqRule`, which is a subtype of T_Rule.

Cost functions (instances of T_CostFunc) are defined as special types of functions, making them first-class objects. Each function is associated a cost through the behavior *B_costFunction*. Application of this behavior to a function object *f* (i.e., *f. B_costFunction*) returns another function object *g* of type T_CostFunc, which implements the computation of the cost of executing function *f*. This allows definition of parameterized cost functions, whose values are dependent upon a number of factors.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer with the extensibility inherent in object models. The optimizer basically implements a control strategy that associates a search strategy and a cost function to each query.

6.2 Versioning

Traditionally, a *version* of a particular modeled entity (e.g., object, type, schema, objectbase) is perceived as a state of that entity as it existed at a particular time during its evolution. *Version control* is the ability to effectively and selectively manage versions of entities. For example, engineering design applications may track versions of components that have been put into production, stock market, and taxation analysis applications, and may use versions of a futures model to evaluate "what if" scenarios, and to provide alternate futures scenarios. Collaborative systems may have different design teams working on different versions of an overall design, and a system may even version the schema as it evolves, so that old and new objects can coexist in the system without having to perform conversions on the instances of the schema (Skarra and Zdonik, 1986). Some researchers have separated user-level versions from system-level versions, and then limited the version model to encompass user-level versions only (Sciore, 1994). With uniform object models such as TIGUKAT, both user-level and system-level versions can seamlessly coexist, and a single version model suffices to support both. The version model developed for TIGUKAT (Peters et al., 1995) uniformly supports both user-level and system-level versions.

Temporal behaviors and branching time (i.e., branching behavior histories) are the framework for version support in TIGUKAT. A behavior can be temporal or non-temporal. The non-temporal behaviors maintain the most recent (i.e., snapshot) results, while the temporal behaviors maintain a history of results as the behavior changed over time. This history may be represented by a linear time-model or a branching time-model. We propose to use the latter where each branch represents an alternate future (or version) of the behavior history. The unique aspects and advantages of our approach are the following:

1. The model is general in that it can be applied to any history tracking system that incorporates branching time. For example, it can be used on both valid time and transaction time as long as (1) they are modeled as histories, and (2) branching time is supported. Other systems support valid and transaction time

histories (Rose and Segev, 1991; Dayal and Wu, 1992), however, branching time is not directly supported in these systems (branching time is supportable in the model discussed in Dayal and Wu, 1992), but the burden of developing a branching model is left up to the user).

2. A portion of a behavior history (called a *version slice*) can be defined by specifying a start time and an end time on the history timeline. A version slice denotes the initial history of a temporal behavior for a given version, and only that portion of the original behavior history is visible in the version. This is useful for excluding parts of the behavior history from the version. Version slicing is unique in that other temporal versioning models define a version based on the entire behavior history up to a certain end time.
3. Each version slice can spawn an independent branch on the timeline after the end of a slice. This is useful since it allows the behavior to temporally evolve along this branch, independent of any other versions. We are unaware of any other model that allows version slices and versions to temporally evolve independent of other versions in this manner.
4. A version slice can *mirror* or *copy* the portion of the history on which it is defined. A *mirrored* slice reflects all changes to the slice in both the original and the version (i.e., updates to the version or the original within the slice are visible to both). A *copied slice* is a separate, independent copy of the original behavior history that becomes part of the new version (i.e., the original and the version have their own copy of the slice, and updates to the version or the original within the slice are not visible in the other).
5. The version model is general and, when incorporated into a uniform object model like TIGUKAT, system-level versions such as versions of schema and versions of the entire objectbase can be modeled in addition to user-level versions without the need for extensions. This unifies user-level versions and system-level versions within a single framework.

We have completed the design of the branching time version model, defined a uniform behavioral representation of this model within TIGUKAT, and developed user language support for managing versions. The versioning approach has been mapped to other approaches such as versions of types, versions of schema, and versions of the entire objectbase, which are useful for schema evolution. This signifies the uniform feature of the version model as an underlying framework to support all types of versioning approaches. We are currently undertaking the implementation of the version model.

6.3 Transactions

Conventional transaction management involves the synchronization of simple read/write access to a shared database in an environment that is not failure-free. Both the transaction models and the synchronization principles that are used in these environments are simple compared to those that are needed in OBMSs. The complexity of the application domains that the OBMS technology is expected to serve is reflected in the type of transaction management support that they require. In these systems, there is a recognized need for more general and powerful transaction models (Elmagarmid, 1992). An overview of transaction management concerns in OBMSs was given in Özsu (1994).

One important characteristic of the relational data model—which is the basis of most current commercial systems—is its lack of a clear update semantics. The model, as it was originally defined, clearly spells out how the data in a relational database are to be retrieved (by means of the relational algebra operators), but it does not specify what it really means to update the database. The consequence is that the consistency definitions and the transaction management techniques are orthogonal to the data model. It is possible—and indeed it is common—to apply the same techniques to non-relational DBMSs or even to non-DBMS storage systems.

The independence of the developed techniques from the data model may be considered an advantage, since the effort can be amortized over a number of different applications. Indeed, the existing transaction management work on OBMSs have exploited this independence by porting the well-known techniques to the new system structures. During this porting process, the peculiarities of OBMSs such as class (type) lattice structures, composite objects, and object groupings (class extents) are accommodated, but the techniques are essentially the same.

In TIGUKAT, we are taking a different approach. It is our claim that, in OBMSs, it is not only desirable to model update semantics within the object model, but it is indeed essential for the correct operation of these systems. The arguments are as follows:

1. In OBMSs, both data and operations on data (which are called methods, behaviors, or operations in various object models) are stored. Queries that access an object-oriented database refer to these operations as part of their predicates. In other words, the execution of these queries invokes various operations defined on the classes (types). To guarantee the safety of the query expressions, existing query processing approaches restrict these operations to be side-effect free, in effect disallowing them to update the database. This is a severe restriction that should be relaxed by the incorporation of update semantics into the query safety definitions.
2. Transactions in OBMSs affect the type (class) lattices. Thus, there is a direct relationship between dynamic schema evolution and transaction management. Many of the conventional techniques employ locking on this lattice to accommodate these changes. However, locks (even multi-granularity locks) severely restrict concurrency. The definition of what it means to update an objectbase, and the definition of conflicts based on this definition of update semantics would allow more concurrency.

   It is interesting to note again the relationship between changes to the type (class) lattice and query processing. In the absence of a clear definition of update semantics and its incorporation into the query processing methodology, most of the current query processors assume that the database schema (i.e., the type (class) lattice) is static during the execution of a query (Staube and Özsu, 1990*a*).

3. Since TIGUKAT treats all system entities, including the database schema (i.e., meta-objects) and queries, as objects that can themselves be queried, it is only natural to model transactions as objects. However, since transactions are basically constructs that change the state of the database, their effects on the database need to be clearly specified.

   Within this context, it should also be noted that the application domains that require the services of OBMSs tend to have somewhat different transaction management requirements, both in terms of transaction models and in terms of consistency constraints. Modeling transactions as objects enables the application of the well-known object-oriented techniques of specialization and subtyping to create various different types of transaction managers. This gives the system extensibility.
4. Some of the requirements require rule support and active database capabilities. Rules themselves execute as transactions, which may spawn other transactions. It has been argued that rules should be modeled as objects (Dayal et al., 1988), but if that is the case then, certainly, transactions should be modeled as objects too.

Consequently, we are now working to define the update semantics of the TIGUKAT object model, and are investigating a powerful transaction model (which may better be called a *workflow*, following more current terminology) that meets the requirements of the application domains that OBMSs are likely to serve, and is modeled in the system as objects. The concurrency control algorithms that are appropriate for these models exploit the semantics of operations and provide flexibility to the type implementors in defining the concurrent execution semantics. Our work in this area is relatively recent and more concrete results will be reported in future articles.

7. Conclusions and Future Directions

In this article, we provide an overview of the TIGUKAT objectbase management system under development at the Laboratory for Database Systems Research at the University of Alberta. TIGUKAT has a uniform behavioral object model where everything is a first-class object, and the only means of accessing the objectbase is through behavior application.

We have defined a query model for the system, complete with an object calculus, an object algebra, and a user language. The user-language consists of a definition language, a session language, and an SQL-based query language. The interpreters for the first two, and the compiler for the last one, have been implemented. An extensible query optimizer has been defined, and a type system to support this architecture has been implemented. The optimizer is being developed as a uniform extension to the object model and, therefore, will be integrated with the model just like the query model has been.

Current work on the system is progressing along five lines: (1) the incorporation of time into the object and query models, (2) the definition of the update semantics for the model, (3) the development of a view manager, (4) the development of storage structures to support query optimization (i.e., indexing and clustering issues), and (5) the definition of a transaction model and its incorporation into the model.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under reserach grants OGP0951 and OGP8191, as well as by the Canadian Institute for Telecommunications Research (CITR), a federally funded Centre of Excellence.

The authors thank the anonymous referees as well as Malcolm Atkinson for numerous suggestions that significantly improved the article.

Technical reports and papers related to TIGUKAT are available via the World Wide Web at *http://web.cs.ualberta.ca/~database/tigukat.html.*

References

Atkinson, M. and Buneman, P. Types and persistence in database programming languages. *ACM Computer Surveys,* 19(2):105-190, 1987.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, P.W., and Morrison, R. An approach to persistent programming. *The Computer Journal,* 26(4):360-365, 1983.

Atkinson, M., Bancilhon, F., DeWitt, D.J., Dittrich, K., Maier, D., and Zdonik, S. The object-oriented database system manifesto. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases,* Kyoto, Japan, 1989.

André, P. and Royer, J. Optimizing method search with lookup caches and incremental coloring. *OOPSLA Conference Proceedings,* Vancouver, 1992.

Bancilhon, F., Delobel, C., and Kanellakis, P. *Building an Object-Oriented Database System: The Story of O_2.* Menlo Park, CA: Morgan Kaufmann, 1992.

Blakeley, J.A. DARPA open object-oriented database preliminary module specification: Object query module. Technical report, Texas Instruments, December, 1991.

Butterworth, P., Otis, A., and Stein, J. The Gemstone object database management system. *Communications of the ACM,* 34(10):64-77, 1991.

Cardelli, L. A semantics of multiple inheritance. In: Kahn, G., MacQueen, D., and Plotkin, G., eds. *Semantics of Data Types, Volume 173 of Lecture Notes in Computer Science,* Berlin: Springer Verlag, 1984, pp. 51-67.

Cardelli, L. A polymorphic λ-calculus with Type:Type. Research Report 10, DEC Systems Research Center, May 1986.

Cattell, R.G. *Object Data Management*. Reading, MA: Addison Wesley, 1991.

Cointe, P. Metaclasses are first class: The ObjVlisp model. *OOPSLA Conference Proceedings,* 1987.

Dayal, U. Queries and views in an object-oriented data model. *Proceedings of the Second International Workshop on Database Programming Languages,* Gleneden Beach, OR, 1989.

Dayal, U., Buchmann, A., and McCarthy, D. Rules are objects too: A knowledge model for an active object-oriented database system. *Proceedings of the Second International Workshop on Object-Oriented Database Systems,* Bad Munster am Stein-Ebernburg, 1988.

Dayal, U. and Wu, G. A uniform approach to processing temporal queries. *Proceedings of the Eighth International Conference on Data Engineering,* Phoenix, AZ, 1992.

Deux, O., et al. The O_2 system. *Communications of the ACM,* 34(10):34-48, 1991.

Dixon, R., McKee, T., Schweizer, P., and Vaughan, M. A fast method dispatcher for compiled languages with multiple inheritance. *OOPSLA Conference Proceedings,* New Orleans, LA, 1989.

Elmagarmid, A.K., ed. *Transaction Models for Advanced Database Applications.* Menlo Park, CA: Morgan Kaufmann, 1992.

Fong, E., Kent, W., Moore, K., and Thompson, C. X3/SPARC/DBSSG/OODBTG Final Report. Technical report, NIST, September 1991.

Gallagher, L.J. Object SQL: Language extensions for object data management. *Proceedings of the First International Conference on Information and Knowledge Management,* Baltimore, MD, 1992.

Gelder, A.V. and Topor, R.W. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems,* 16(2):235-278, 1991.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Reading, MA: Addison Wesley, 1983.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Second edition. Reading, MA: Addison-Wesley, 1989.

Goralwalla, I. and Özsu, M.T. Temporal extensions to a uniform behavioral object model. *Proceedings of the Twelfth International Conference on Entity-Relationship Approach,* Arlington, TX, 1993.

Irani, B.B. Implementation of the TIGUKAT object model. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report, TR93-10.

Kent, W. A Rigorous Model of Object Reference, Identity and Existence. Technical Report HPL-90-31, Hewlett Packard Labs, April 1990.

Khoshafian, S.N. and Copeland, G.P. Object identity. *OOPSLA Conference Proceedings,* Portland, OR, 1986.

Kim, W., Ballou, N., Chou, H.T., Garza, J.F., and Woelk, D. Features of the ORION object-oriented database system. In: Kim, W. and Lochovsky, F.H., eds. *Object-Oriented Concepts, Databases, and Applications.* Reading, MA: Addison Wesley, 1989.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Communications of the ACM,* 34(10):50-63, 1991.

Lanzelotte, R. and Valduriez, P. Extending the search strategy in a query optimizer. *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, 1991.

Lecluse, C., Richard, P., and Velez, F. O_2, an Object-Oriented Data Model. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* Chicago, IL, 1988.

Lipka, A.P. The design and implementation of TIGUKAT user languages. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR93-11.

Maier, D., Zhu, J., and Ohkawa, H. Features of the TEDM object model. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases,* Kyoto, Japan, 1989.

Makowsky, J.A. Characterizing data base dependencies. *Proceedings of the Eighth Colloquium on Automata, Languages and Programming,* Location?, 1981.

Mitchell, G., Zdonik, S.B., and Dayal, U. Optimization of object-oriented query languages: Problems and approaches. In: Dogac, A., Özsu, M.T., and Biliris, A., eds. *Advances in Object-Oriented Database Systems,* Berlin: Springer Verlag, 1993.

Muñoz, A. Extensible query optimizer architecture for TIGUKAT. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR94-01.

Osborn, S.L. Identity, equality and query optimization. In: Dittrich, K.R., ed. *Advances in Object-Oriented Database Systems, Volume 334 of Lecture Notes in Computer Science,* Berlin: Springer Verlag, 1988, pp. 346-351.

Ozsoyoglu, G. and Wang, H. A relational calculus with set operators, its safety, and equivalent graphical languages. *IEEE Transactions on Software Engineering,* SE-15(9):1038-1052, 1989.

Özsu, M.T. Transaction models and transaction management in object-oriented database management systems. In: Dogac, A., Özsu, M.T., Biliris, A., and Sellis, T., eds. *Advances in Object-Oriented Database Systems.* Berlin: Springer Verlag, 1994.

Özsu, M.T., Straube, D.D., and Peters, R. Query processing issues in object-oriented knowledge base systems. In: Petry, F.E. and Delcambre, L.M., eds. *Intelligent Database Technology: Approaches and Applications, Advances in Databases and Artificial Intelligence,* Volume 1. Greenwich, CT: JAI Press, 1994, pp. 79-144.

Peters, R.J. TIGUKAT: A uniform behavioral objectbase management system. PhD thesis, University of Alberta, Edmonton, Alberta, Canada, 1994. Available as University of Alberta Technical Report TR94-06.

Peters, R.J., Goralwalla, I., and Özsu, M.T. A Unified Version Model Based on Branching Time. Technical Report, University of Manitoba, 1995.

Peters, R.J., Lipka, A., Özsu, M.T., and Szafron, D. An extensible query model and its languages for a uniform behavioral object management system. *Proceedings of the Second International Conference on Information and Knowledge Management,* Washington, DC, 1993*a*.

Peters, R.J., Lipka, A., Özsu, M.T., and Szafron, D. The query model and query language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, January, 1993*b*.

Peters, R.J. and Özsu, M.T. Reflection in a uniform behavioral object model. *Proceedings of the Twelfth International Conference on the Entity-Relationship Approach,* Arlington, TX, 1993.

Richardson, J. and Carey, M. Persistence in the E language: Issues and implementation. *Software–Practice & Experience,* 19(12):1115-1150, 1989.

Richardson, J., Carey, M., and Schuh, D. The design of the E programming language. Technical Report 824, University of Wisconsin, February 1989.

Rose, E. and Segev, A. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. *Proceedings of the Tenth International Conference on the Entity-Relationship Approach,* 1991.

Schuh, D.T., Carey, M.J., and DeWitt, D.J. Persistence in E revisited–Implementation Experiences. In: Implementing persistent object bases: Principles and practice. *Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1990.

Sciore, E. Versioning and configuration management in an object-oriented data model. *The VLDB Journal,* 3(1):77-106, 1994.

Shaw, G. and Zdonik, S.B. A query algebra for object-oriented databases. *Proceedings of the Sixth International Conference on Data Engineering,* 1990.

Shipman, D.W. The functional data model and the language DAPLEX. *ACM Transactions on Database Systems,* 6(1):140-173, 1981.

Skarra, A.H. and Zdonik, S.B. The management of changing types in an object-oriented database. *OOPSLA Conference Proceedings*, 1986.

Snodgrass, R. and Ahn, I. A taxonomy of time in databases. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* 1985.

Snyder, A. An abstract object model for object-oriented systems. Technical Report HPL-90-22, Hewlett Packard Labs, April 1990.

Straube, D.D. and Özsu, M.T. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems,* 8(4):387-430, 1990*a*.

Straube, D.D. and Özsu, M.T. Type consistency of queries in an object-oriented database system. *Proceedings of the ECOOP/OOPSLA Conference,* Ottawa, Canada, 1990*b*.

Stroustrup, B. *The C++ Programming Language*. Reading, MA: Addison Wesley, 1986.

Wirfs-Brock, A. and Wilkerson, B. An overview of modular Smalltalk. *OOPSLA Conference Proceedings*, San Diego, CA, 1988.

Wirfs-Brock, A. and Wilkerson, B. Object-Oriented Design: A Responsibility-Driven Approach. *OOPSLA Conference Proceedings*, New Orleans, LA, 1989*a*.

Wirfs-Brock, A. and Wilkerson, B. Variables limit reusability. *Journal of Object-Oriented Programming*, 2(1):34-40, 1989*b*.

Yu, L. and Osborn, S.L. An evaluation framework for algebraic object-oriented query models. *Proceedings of the Seventh International Conference on Data Engineering*, Kyoto, Japan, 1991.

Zdonik, S.B. and Maier, D. Fundamentals of object-oriented databases. In: Zdonik, S.B. and Maier, D., eds. *Readings in Object-Oriented Databases*, Menlo Park, CA: Morgan-Kaufman, 1990.

Appendix A: Primitive Type System

The following tables show the signatures of the behaviors for the non-atomic types (except the container types), the signatures of the behaviors for the container types, and the signatures of the behaviors for the atomic types. The receiver type of a behavior is excluded, because the receiver must be an object of a type that is compatible with the type defining the behavior. The notation `T_collection<T>` is used to define a collection type whose members are of type T. The type specifications for the behaviors are the *most general* types. Types for some of the behaviors are revised in the subtypes. For example, the result type *B_self* is always the type of the receiver object and the result type *B_new* is always the membership type of the receiver class.

Table 2. Behavior signatures of non-atomic types of primitive type system

Type	Signatures	
T_object	*B_self:*	T_object
	B_mapsto:	T_type
	B_conformsTo:	T_type → T_boolean
	B_equal:	T_object → T_boolean
	B_notequal:	T_object → T_boolean
	B_persistent:	T_object
	B_transient:	T_object
	B_newprod:	T_list<T_object> → T_list<T_set<T_behavior>> → T_object
T_type	*B_interface:*	T_set<T_behavior>
	B_native:	T_set<T_behavior>
	B_inherited:	T_set<T_behavior>
	B_specialize:	T_type → T_boolean
	B_subtype:	T_type → T_boolean
	B_subtypes:	T_set<T_type>
	B_supertypes:	T_set<T_type>
	B_sub-lattice:	T_poset<T_type>
	B_super-lattice:	T_poset<T_type>
	B_classof:	T_class
	B_tmeet:	T_set<T_type> → T_type
	B_tjoin:	T_set<T_type> → T_type
	B_tproduct:	T_list<T_type> → T_type
T_product	*B_compTypes:*	T_list<T_type>
T_behavior	*B_name:*	T_string
	B_argTypes:	T_list<T_type>
	B_resultType:	T_type → T_type
	B_semantics:	T_object
	B_associate:	T_type → T_function → T_behavior
	B_implementation:	T_type → T_function
	B_primitiveApply:	T_object → T_object
	B_apply:	T_object → T_list → T_object
	B_defines:	T_set<T_type>
T_function	*B_argTypes:*	T_list<T_type>
	B_resultType:	T_type
	B_source:	T_object
	B_primitiveExecute:	T_object → T_object
	B_basicExecute:	T_list → T_object
	B_execute:	T_list → T_object
	B_compile:	T_object
	B_executable:	T_object

Table 3. Behavior signatures of container types of primitive type system

Type	Signatures	
T_collection	*B_memberType:*	T_type
	B_union:	T_collection → T_collection
	B_diff:	T_collection → T_collection
	B_intersect:	T_collection → T_collection
	B_collapse:	T_collection
	B_select:	T_string → T_list<T_collection> → T_collection
	B_project:	T_set<T_behavior> → T_collection
	B_map:	T_string → T_list<T_collection> → T_collection
	B_product:	T_set<T_collection> → T_collection
	B_join:	T_string → T_list<T_collection> → T_collection
	B_genjoin:	T_string → T_list<T_collection> → T_collection
	B_setEqual:	T_collection → T_boolean
	B_containedBy:	T_collection → T_boolean
	B_cardinality:	T_natural
	B_elementOf:	T_object → T_boolean
	B_insert:	T_object → T_collection
	B_delete:	T_object → T_collection
T_bag	*B_occurrences:*	T_object → T_natural
	B_count:	T_natural
	Inherited behaviors refined to preserve duplicates	
T_poset	*B_ordered:*	T_object → T_object → T_boolean
	B_ordering:	T_behavior
	Inherited behaviors refined to preserve ordering	
T_list	*B_first:*	T_object
	B_last:	T_object
	B_next:	T_object
	B_previous:	T_object
	Inherited Behaviors refined to preserve duplicates and ordering	
T_class	*B_deepExtent:*	T_collection
	B_new:	T_object
T_class-class	*B_new:*	T_type → T_class
T_type-class	*B_new:*	T_set<T_type> → T_set<T_behavior> → T_type
T_collection-class	*B_new:*	T_type → T_collection
T_product-class	*B_new:*	T_list<T_object> → T_object

Table 4. Behavior signatures of atomic types of primitive type system

Type	Signatures	
T_atomic		
T_boolean	*B_not:*	T_boolean
	B_or:	T_boolean → T_boolean
	B_if:	T_object → T_object → T_object
	B_and:	T_boolean → T_boolean
	B_xor:	T_boolean → T_boolean
T_character	*B_ord:*	T_natural
T_string	*B_car:*	T_character
	B_cdr:	T_string
	B_concat:	T_string → T_string
T_real	*B_succ:*	T_real
	B_pred:	T_real
	B_add:	T_real → T_real
	B_subtract:	T_real → T_real
	B_multiply:	T_real → T_real
	B_divide:	T_real → T_real
	B_trunc:	T_integer
	B_round:	T_integer
	B_lessThan:	T_real → T_boolean
	B_lessThanEQ:	T_real → T_boolean
	B_greaterThan:	T_real → T_boolean
	B_greaterThanEQ:	T_real → T_boolean
T_integer	Behaviors from T_real refined to work on integers	
T_naturals	Behaviors from T_integer refined to work on naturals	