TIGUKAT Object Management System: Initial Design and Current Directions*†

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

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We describe the TIGUKAT object management system that is under development at the Laboratory for Database Systems Research of 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 is a first-class object with well-defined behavior. The computational model supported is one of applying behaviors to objects. A query model has been developed for TIGUKAT that is complete with a formal object calculus, an equivalent object algebra and an object SQL language. The uniformity of the model permits modeling queries as objects, opening up the possibility of developing an extensible query optimizer. A prototype implementation of TIGUKAT, including the language and its optimizer is ongoing.

1 Introduction

The penetration of data management technology into new application areas with different 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 imaging systems. It is now commonly accepted that relational database management systems (RDBMSs), with their flat representation of data, are not sufficiently powerful to support these applications. 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. Specifically, the more important shortcomings of the relational model in meeting the requirements of these applications can be listed as follows [27]:

1. Relational systems deal with a single object type: a relation. A relation is used to model different real-world objects, but the semantics of this association is not part of the database. Furthermore, the attributes of a relation only from simple domains (numeric, character, date, etc.). The advanced applications cited above require explicit storage and manipulation of more abstract types (e.g., images, design documents, etc.) and the ability for the users to define their own application-specific types.

2. Additionally, relational systems capture only implicitly the relationships between the objects that various relations represent. These relationships are buried in the value relationships between attributes of relations. Thus, the schema is flat, consisting of a collection of relations. In applications where real-world entities are explicitly represented as abstract types, these relationships must be brought out explicitly, which

*The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., 895 Don Mills Road, North York, Ontario M3C 1W3.

†This research is supported by the Natural Science and Engineering Research Council of Canada under research grant OGP0951.

requires rich type systems for handling complex object structures with nested objects (e.g., a vehicle object containing an engine object).

3. Relational systems provide a declarative and (arguably) simple language for accessing the data. The applications mentioned above require richer languages that overcome the well-known "impedance mismatch" problem. This problem arises because of the differences in the level of abstraction between the relational languages and the programming languages with which they interact.

Object-oriented technology is the topic of intense study as the major candidate to successfully meet the requirements of advanced applications that require data management services. At the Laboratory for Database Systems Research of the University of Alberta, we are engaged in the design and development of an object-oriented DBMS (OODBMS), called TIGUKAT,[1] which follows the object-oriented methodology in its own design. Consequently, all database functionality is incorporated within an extensible object model.

Typically, OODBMS development has followed two streams in the past. The first is to extend object-oriented programming languages with DBMS features such as persistence and a query facility. The resulting systems are typically a merger between object-oriented and relational systems. Out of this approach there has emerged extensions to C++ (e.g., ObjectStore [21] and EXODUS [7]) and SmallTalk (e.g., GemStone [5]), among others. The second approach is to develop a language-independent object model and consistently extend it with DBMS features. TIGUKAT follows the second approach along with ORION [4], O_2 [3], and IRIS [11].

TIGUKAT adopts the functional/behavioral approach to access information about objects. This has been proposed in some prototype systems [25, 9, 24], however, most commercial systems provide some distinction between the properties (i.e., stored attributes) of an object and methods (i.e., operations or behaviors) that can be performed on them. TIGUKAT has a purely behavioral object model where the user interaction with the system is in terms of the application of behaviors to objects. In this way, full abstraction of modeled entities is accomplished since users do not have to differentiate between attributes and methods. We feel that a purely behavioral approach provides several benefits including consistency, understandability and portability. These are the major reasons for developing the behavioral model of TIGUKAT and for separating behaviors from their implementations (i.e., functions).

TIGUKAT's object model is uniform. Everything in the system, including types, classes, collections, behaviors, functions as well as meta-information, is a first-class object with well-defined behavior. Thus, there is no separation between objects and values, and the schema information is a natural part of the database that can be queried just like other objects. Current systems do not carry uniformity to the extent that we require for defining the model within itself and for integrating the query model, query optimizer, view manager, etc., as type and behavior extensions to the base model. Thus, we developed TIGUKAT to be completely uniform by strictly enforcing encapsulation of all information, by making behavior application the only operational semantics, and by allowing anything that can be described by a type with well-defined behavior to be an object. We have shown how the object model, query model, and optimizer are described in this way, thereby integrating them as a single consistent system.

Many commercial OODBMSs are currently adding declarative query capabilities. There has also been some theoretical work on formal object query models [1]. However, the query models of most OODBMSs describe object-oriented extensions to the relational query model which does not provide a good basis for object-oriented development. As a result, these extensions are usually only informally defined, and the extent of their functionality is somewhat limited and usually hard to describe. Quite different from any other system developed to date, TIGUKAT has a formal object query model incorporating both a formal object calculus and an object algebra with a proven equivalence. Furthermore, we developed a notion of safety based on the *evaluable* class of queries[2] [13], and give feasible algorithmic solutions to determine safety and to translate calculus expressions to equivalent algebra operators. We feel that this is a much better basis for implementing an object query model.

Although lacking in the formal aspects, several systems have defined languages for querying

[1] TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning "objects."

[2] The evaluable class of queries is arguably the largest decidable subclass of domain independent queries.

a database. There have been programming language extensions (GemStone/OPAL [5], O_2/O_2C [10]), SQL-like query languages with object-oriented extensions (IRIS/OSQL [16], ORION/SQL-like [18], O_2/RELOOP [8]), and extensions to QUEL (EXODUS/Excess [7]). TIGUKAT incorporates a complete SQL-like user language called TQL (TIGUKAT Query Language), an object definition language called TDL (TIGUKAT Definition Language) and a control language called TCL (TIGUKAT Control Language). An identifying characteristic of our language is that TQL is proven equivalent to the formal languages, which makes it easier to perform logical transformations and argue about its safety.

Query optimization is essential for efficient execution plans. A lot of research in extending relational query optimization techniques has been done in recent years (e.g., [31]). A more viable approach is to use object-oriented design to develop an extensible query optimizer that can evolve and improve its performance over time. An extensible query optimizer is under development for TQL, where the components of the optimizer such as cost functions, algebraic transformation rules, and the search strategy (i.e., the heuristics that determine when it is profitable to apply a certain rule) are all modeled as objects. Thus, multiple optimization strategies can be supported in the system, and adding new rules, heuristics, or cost functions, as well as changing the strategy used in optimizing a given query, can be accomplished completely within the model.

In the remainder of this paper, we provide a general overview of TIGUKAT, describe our current implementation efforts, and outline our future research directions. We start, in Section 2, with an overview of the TIGUKAT object model, presenting the primitive type system and highlighting its important features. A simplified geographic information system (GIS) is also presented as an example database application to demonstrate the features of TIGUKAT. Section 3 includes a discussion of the query model and the query language TQL. Several examples are presented to illustrate the features of the model and TQL. This is followed, in Section 4, with a presentation of the general architecture of the query optimizer as an extension to the primitive object model. In Section 5, we describe the architecture and current implementation status of our prototype. Finally, in Section 6, we conclude with a discussion of our current research directions.

2 Object Model

2.1 Object Model Overview

The TIGUKAT object model [30] is defined *behaviorally* with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects. The model is *uniform* in that every component of information, including its semantics, is uniformly modeled as objects with well-defined behavior. Every element in the model has the status of a *first-class object*.

The primitive type system of TIGUKAT is shown in Figure 1. The primitive objects of the model include: *atomic entities* (reals, integers, strings, etc.), *types* for defining the common features of objects, *behaviors* for specifying the semantics of operations that may be performed on objects, *functions* for specifying implementations of behaviors,[3] *classes* for automatic grouping of objects based on type,[4] and *collections* for supporting general heterogeneous groupings of objects. In the remainder of the paper, the prefix `T_` refers to a type, **C_** refers to a class, **L_** refers to a collection, and *B_* refers to a behavior. For example, `T_person` is a type reference, **C_person** a class reference, **L_seniors** a collection reference, *B_age* a behavior reference, and a reference such as david without any prefix represents some other application specific reference. Some primitive types and behaviors are elaborated in this paper. For the complete model definition we refer the reader to [30].

Objects are defined as *(identity, state)* pairs where *identity* represents a unique, immutable object identity and *state* represents the information carried by the object. Thus, the model supports *strong object identity* [17]. This does not preclude application environments from having many *references* (or *denotations*) to objects, that need not be unique. The *state* of an object *encapsulates* the information carried by that object. Conceptually, every object is a *composite* object, meaning every object has references (not necessarily implemented as pointers) to other objects. For example, integers have behaviors that return objects, but they are not implemented as pointers.

The access and manipulation of an object's state occurs exclusively through the application of behav-

[3] Behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.

[4] Types and their extents are separate constructs in TIGUKAT.

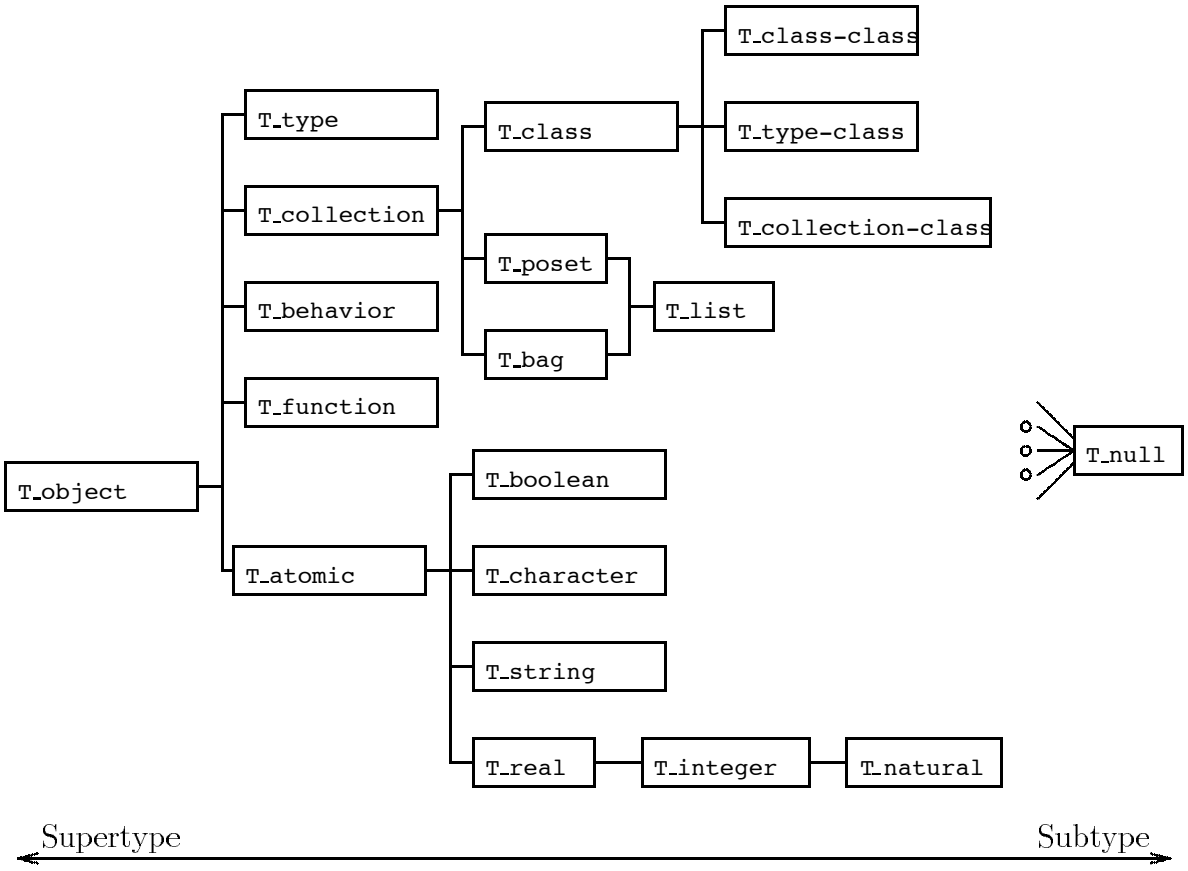

Figure 1: Primitive Type System

iors. An important primitive behavior defined on objects and used in the query model is *identity equality*, which compares two object references based solely on the identities of the objects they denote. This is the only equality defined by the primitive type system. Other notions of equality, such as structural equality based on object components, can be derived from identity equality.

The model separates the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*). A *type* specifies behaviors and encapsulates behavior implementations and state for objects created using that type as a template. The behaviors defined by a type describe the *interface* to the objects of that type. Types are organized into a lattice structure using the notion of *subtyping*, which promotes software reuse and incremental development. TIGUKAT supports *multiple subtyping*. Thus, the type structure is potentially a directed acyclic graph (DAG). However, this DAG is converted to a lattice by rooting it at type `T_object` and *lifting* with the base type `T_null`.

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental but distinct construct, responsible for managing the instances created using a specific type as a template. Every class is uniquely associated with a single type. A type must have an associated class before instances of that type can be created. The entire group of objects of a particular type is known as the *extent* of the type. This is separated into the notion of *deep extent*, which refers to all objects created from the given type or one of its subtypes, and the notion of *shallow extent*, which refers only to those objects created from the given type without considering its subtypes. In general, we use a class specification to mean the *deep extent* and explicitly denote the *shallow extent* with the suffix '+' when needed. For example, **C_person**

denotes all objects of type T_person or one of its subtypes, while C_person+ denotes only those objects specifically of type T_person.

Objects of a particular type cannot exist without a class and every class is uniquely associated with a single type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Another unique feature of classes is that object creation occurs only through a class. Defining object, type, and class in this manner introduces a clear separation of these concepts. This separation is important when handling definitions of *abstract types*[5] that don't have any instances and therefore don't need to store any extent information. Other areas where this separation is useful is in type inferencing and schema evolution where type information (i.e., the schema) is manipulated into new structures and there is no need to be concerned with the overhead of classes.

We define a *collection* as a general user grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, no object creation may occur through a collection; object creation occurs only through classes. 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 subtype lattice whereas the management of collections is *explicit*, meaning the user is responsible for their extents. Finally, the elements of a class are homogeneous up to inclusion polymorphism, while a collection may be heterogeneous in the sense that it can contain objects that may be of different types. There is no equivalent of shallow extent for collections.

We define *class* as a subtype of *collection*. This introduces a clean semantics between the two and allows the model to use both constructs in an effective manner. For example, the targets and results of queries are typed collections of objects. This means targets also include classes, because of the specialization of classes on collections. This approach provides great flexibility and expressiveness in formulating queries and gives *closure* to the query model, which is often regarded as an important feature [32].

Two other fundamental notions are *behaviors* and the *functions* (known as *methods* in other models) that implement them. We clearly separate the definition of a behavior from its possible implementations (functions/methods). The benefit of this approach is that common behaviors over different types can have a different implementation for each of the types. This is direct support for behavior *overloading* and *late binding* of implementations to behaviors. These are recognized as a major advantage of object-oriented computing.

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of a behavior. The implementation of a particular behavior may vary over the types that support it. However, the semantics of the behavior remain consistent over all types supporting the behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavior application as the invocation of a function, regardless of whether the function is stored or computed. Functions are examined more closely in Section 3 where we show that queries are specialized functions and therefore carry all the semantics of function objects. This has the benefit of allowing queries to be used as the implementation of behaviors.

2.2 An Example System

We present a simple geographic information system (GIS) as a running example to demonstrate the power of TIGUKAT. This example is selected because it is among the application domains that can potentially benefit from the advanced features offered by object-oriented technology.

A type lattice for a simplified GIS is given in Figure 2. The example includes the root types of the various sub-lattices from the primitive type system to illustrate their relative positions in an extended application lattice. The GIS example defines abstract types for representing information on people and their dwellings. These include the types T_person, T_dwelling, and T_house. Geographic types to 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 win-

[5] These are more commonly known as *abstract classes*, but since class has a different meaning in our model we use the term *abstract types* instead.

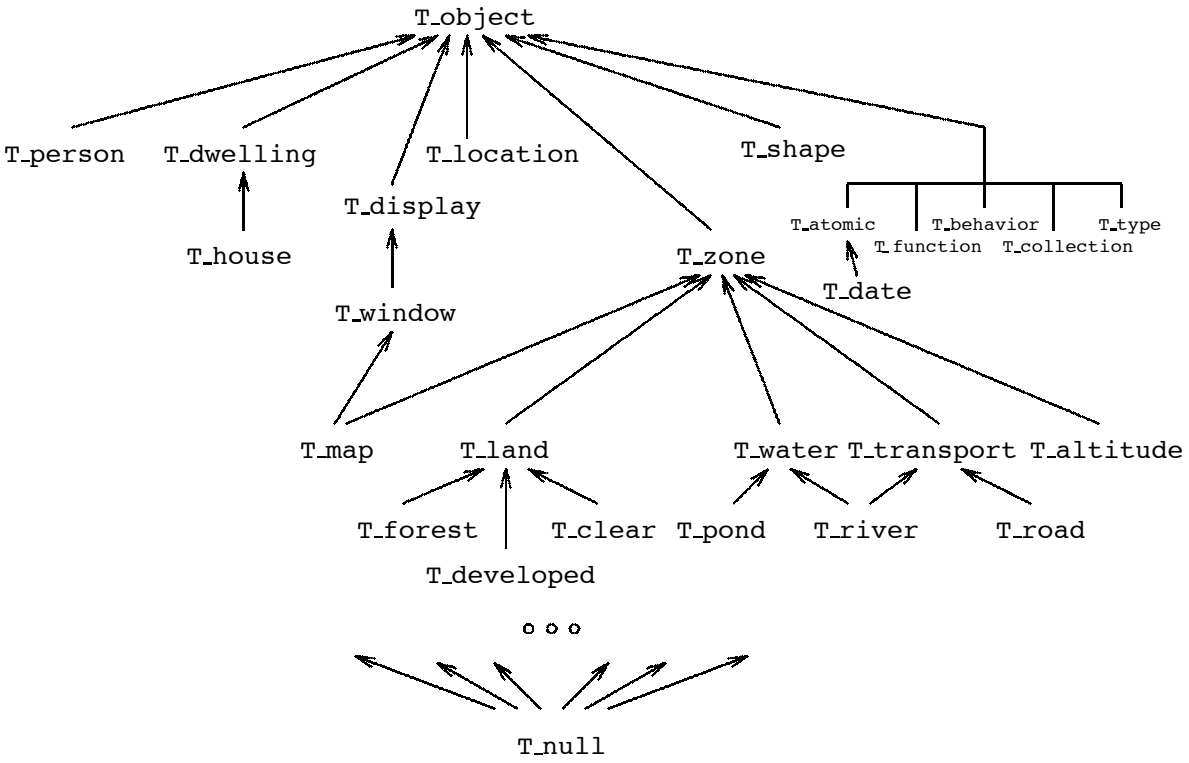

Figure 2: Type lattice for a simple geographic information system.

dow. Displayable types for presenting information on a graphical device are defined. These include the types `T_display` 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_shape` defines the geometric shape of the regions representing the various zones. For our purposes we use only this general type, but in more practical applications this type would be further specialized into subtypes representing polygons, polygons with holes, rectangles, squares, splines, and so on. Table 1 lists the signatures of the behaviors defined on the GIS specific types. The specification `T_coll`$\langle$`T`$\rangle$ where `T` is a type is used to denote a collection type whose members are of type `T`.

Type	Signatures
T_location	*B_latitude*: T_real
	B_longitude: T_real
T_display	*B_display*: T_display
T_window	*B_resize*: T_window
	B_drag: T_window
T_shape	
T_zone	*B_title*: T_string
	B_origin: T_location
	B_region: T_shape
	B_area: T_real
	B_proximity: T_zone $\rightarrow$ T_real
T_map	*B_resolution*: T_real
	B_orientation: T_real
	B_zones: T_coll$\langle$T_zone$\rangle$
T_land	*B_value*: T_real
T_water	*B_volume*: T_real
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_coll$\langle$T_person$\rangle$
T_dwelling	*B_address*: T_string
	B_inZone: T_land
T_house	*B_inZone*: T_developed
	B_mortgage: T_real

Table 1: Behavior signatures of GIS specific types.

3 Query Model and Language

3.1 Query Model Overview

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 behavioral extensions to the primitive model. We define a type `T_query` as a subtype of `T_function` in the primitive type system as illustrated in Figure 3. This means that queries have the status of *first-class objects* and inherit all the behaviors and semantics of objects. Moreover, queries are functions and can be used as implementations of behaviors, they can be compiled, they can be executed, and so on.

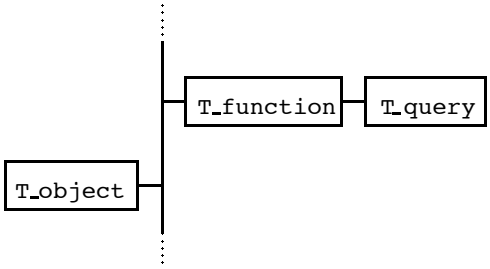

Figure 3: Query type extension to primitive type system.

Functions have source code associated with them and the source code for a query is a TQL statement as defined in Section 3.3. Functions have a behavior *B_compile* that compiles the code. For a query, this involves translating the query statement into an algebra tree, optimizing it, and generating an execution plan. Functions have a behavior *B_execute* that executes the compiled code. In general, for a query this means submitting the execution plan to the storage manager for processing. Furthermore, queries have specialized behaviors such as *B_result*, which is a reference to the materialized query result (i.e., the actual result collection itself). If this result is made persistent, then the query is said to be *stored* and does not need to be re-evaluated the next time it is called upon to *B_execute* itself. Other behaviors of a query relating to the extensible query optimizer include *B_initialPT* and *B_optimizedPT* for accessing the initial and optimized processing trees, *B_searchstrategy* for accessing the search strategy used for optimization, *B_transformations* for accessing the list of transformation rules used during optimization, *B_input-types* for accessing the types of the operand collections, *B_output-type* for accessing the type of the result collection, and several other behaviors for keeping various statistics about queries.

Incorporating queries as a specialization of functions is a very 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, i.e., 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 upon 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 is extensible in a uniform way since the type `T_query` can be further specialized by subtyping. This can be used to arrange the class of queries into additional subclasses, each with its own unique characteristics, and to incrementally establish the characteristics of new kinds of queries as they are developed. For example, we can subtype `T_query` into `T_adhocQuery` and `T_productionQuery` and then define different evaluation strategies for both. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies, while production queries are usually compiled once and then executed many times.

3.2 Formal Query Model

The languages for the query model include a complete object calculus, an equivalent object algebra, and an SQL-like user language.

The user query language (TQL) has a syntax based on the SQL *select-from-where* structure and formal semantics defined by the object calculus. Thus, it combines the power of the relational query languages

with object-oriented features. In this section, we give a brief overview of the calculus and algebra (see [28, 29] for additional details). The following section describes TQL.

Our object calculus has a logical foundation. It defines predicates on collections (essentially sets) of objects and a calculus query returns a collection of objects as a result. This gives the calculus the property of *closure*. The calculus incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavior applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has a rigorous definition of safety (based on the evaluable class of queries) and typing that is compile-time checkable. Moreover, it supports controlled creation and integration of new collections, types, and objects into the existing schema.

Like the calculus, our algebra is *closed* on collections. Algebraic operators are defined as behaviors on the primitive type `T_collection` which is part our uniform approach. These behaviors operate on collections and return a collection as a result. Thus, the algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. Composition of these behaviors accounts for the closure of the algebra.

The minimal set of algebraic operators defined in the system are as follows:

Set Operations The typical set **union**, **difference**, and **intersection** operators are defined.

Collapse (denoted $P \Downarrow$): Collapse takes a collection of collections P and performs the extended union over the element collections of P (i.e., it "flattens" P).

Select (denoted $P\ \sigma_F\ \langle Q_1, \ldots, Q_n \rangle$): This is a parameterized select operation that returns objects from the target collection P that satisfy the predicate F using other collections ($\langle Q_1, \ldots, Q_n \rangle$) as arguments.

Map (denoted $Q_1 \gg_{mop}\ \langle Q_2, \ldots, Q_n \rangle$): Map applies a function (denoted mop) of n arguments to each combination of objects $q_1, q_2, \ldots, q_n$ (where q_i ranges over collection Q_i) and returns a collection containing the results of the function application.

Project (denoted $P\ \Pi_{\mathcal{B}}$): This is a behavioral project operator that, when applied to collection P, returns the same set of objects, except that only the behaviors specified in the project list $\mathcal{B}$ are applicable to the objects. The result of this operation may create a supertype of the type of objects in the P.

Product (denoted $P \times Q$): Product returns a collection of product objects whose components are created from each permutation of the objects in the target collections P and Q. Product may initiate the creation of a new type along with a new class to maintain the product objects.

The additional operators that add to the expressiveness and brevity of the algebra, and provide many useful opportunities for optimization are as follows:

Reduce (denoted $P \triangle_{o_1,\ldots,o_n}$): Reduce is similar to the relational project operator in that it takes a collection of product objects P and eliminates some of the components ($o_1, \ldots, o_n$) of those product objects, returning a collection of product objects with a smaller degree.

Join (denoted $P \bowtie_F \langle Q_1, \ldots, Q_n \rangle$): Join is a special case of Product and produces a collection containing only those product objects that satisfy a predicate F whose arguments range over collections $P, Q_1, \ldots, Q_n$.

Generate Join (denoted $Q_1\ \gamma_g^o\ \langle Q_2, \ldots, Q_n \rangle$): Generate join has two possible results based on the nature of the generating atom g,[6] which is of the form $o\,\theta\,mop$ where θ is one of $\{=, \in\}$, and the mop function, which is a function over the elements of collections $Q_1, \ldots, Q_n$. Generate join produces a collection of product objects created from each permutation of the q_i's that is extended by an object in the following way. If θ is $=$, the mop function is applied to each permutation of the q_i's and the result is appended to the permutation, which forms the product object returned in the result collection. If θ is $\in$, the result of the mop function must be a collection. Each element of this collection is

[6] Generating atoms are defined as part of object calculus, which we do not specify formally in this paper. Equality atoms of the form $o = t$ or a membership atom $o \in t$, where o is an object variable, t is an appropriate term for the atom, and o does not appear in t, are called *generating atoms* for o.

appended to the q_i permutation and returned as a product object in the result collection.

A desirable property of an object query model is that the algebra and calculus be equivalent in expressive power, meaning that all queries expressed in one language can also be expressed in the other. Space limitations do not allow us to include them here, but in [29] we prove the equivalence of our object calculus and algebra in both directions and present the reduction of the user query language to the calculus. Moreover, the safety of our languages is also proven in that report.

3.3 TIGUKAT Query Language

The main function of the TIGUKAT user language is to support the definition, manipulation, and retrieval of objects in an object base. The language consists of three parts: the TIGUKAT Definition Language (TDL) which supports the definition of metaobjects (types, collections, classes, behaviors, and functions), the TIGUKAT Query Language (TQL), which is used to manipulate and retrieve objects, and the TIGUKAT Control Language (TCL), which supports the session specific operations (open, close, save, etc.). Only TQL is presented in this paper; the complete specification of all languages is given in [29].

TQL is based on the SQL paradigm. We adopt this paradigm for various reasons. Most importantly, SQL is accepted as the standard language in relational databases, and current work on SQL3 attempts to extend its syntax and its semantics to fulfill requirements of object-oriented systems [12]. 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. In addition, TQL extends the basic SQL structure by accepting path expressions (i.e., implicit joins [19]) in the *select, from*, and *where* clauses. Object equality, defined by the primitive system, provides support for explicit joins. The results of queries can be queried, since queries operate on collections and always return finite collections as results. Therefore, query statements can be used in the *from* and *where* clauses of other queries (i.e., nested queries or subqueries). Furthermore, objects can be queried regardless of whether they are persistent or transient. We note that the syntax for the application of aggregate functions is not explicitly supported in the current implementation of TQL. However, as the underlying model is purely behavioral, these functions may be defined as behaviors on the primitive type `T_collection` and can be applied to any collection including those returned as a result of a query.

There are four basic TQL operations: **select, insert, delete**, and **update**, and three binary operations: **union, minus**, and **intersect**. In this paper, 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:[7]

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

The *select clause* in this statement identifies the objects that are 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 free variables in object calculus formulas. The *into clause* declares a reference to a new collection. If the *into clause* is not specified, a new collection is created; however, there is no reference to it. The *from clause* declares the range of object variables appearing in the *select* and *where* clauses. The basic form of the from clause consists of a list of variables, followed by the keyword **in**, followed by a reference to a collection. The *where clause* defines a Boolean formula that must be satisfied by objects in the result of the query. The basic form of the *where* clause consists of a number of atoms connected by logical operations **and, or, not**. An atom is either an expression using equality (*term = term*) or membership (*variable* $\in$ *term*), where a term is a path expression or a variable. The following examples give a feel for the basic *select-from-where* structure. A query is first expressed in TQL (T:), followed by a corresponding object calculus expression (C:), and then an equivalent algebraic expression (A:). In the algebraic expressions, we subscript an operand collection by the variable that ranges over it. This variable is used as a symbolic reference to the elements of the collection. We also use symbols $R1$, $R2$

[7] All bold words and characters correspond to terminal symbols of the language (keywords, special characters, etc.). 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.

as temporary results and "$\leftarrow$" for assignment[8]. Furthermore, we use the arithmetic notation for operations $o.B\_greaterThan(p)$, $o.B\_elementOf(p)$, etc., instead of the Boolean path expressions.

Example 3.1 Return land zones valued over \$100,000 or cover an area over 1000 units.

T: **select** o
from o **in C_land**
where $(o.B\_value() > 100000)$
or $(o.B\_area() > 1000)$
C: $\{o \mid \mathbf{C\_land}(o)$
$\wedge (o.B\_value > 100000 \vee o.B\_area > 1000)\}$
A: $\mathbf{C\_land}_o\ \sigma_{[o.B\_value > 100000\ \vee\ o.B\_area > 1000]}$

Example 3.2 Return all zones where people live (the zones are generated from person objects).

T: **select** o
from q **in C_person**
where $(o = q.B\_residence().B\_inZone())$
C: $\{o \mid \exists q(\mathbf{C\_person}(q)$
$\wedge\ o = q.B\_residence.B\_inZone)\}$
A: $\left(\mathbf{C\_person}_q\ \gamma^o_{o=q.B\_residence.B\_inZone}\right) \triangle_q$

The collection reference in the *from clause* may be followed by a plus '**+**'. When the collection reference is a class, the plus indicates that the shallow extent of the class is used as the range. In absence of the plus, the deep extent of the class is used. The plus has no affect on collections. In Example 3.1, the deep extent of class **C_land** is used as the range of o. If it were specified as **C_land**+, then the shallow extent would be used.

The range of a variable in the *from clause* can be the collection returned as the result of a subquery, where a subquery can be either given explicitly or as a reference to a query object. We illustrate the use of a subquery in the examples below.

Two additional constructs are defined for the where clause in order to specify existential and universal quantification. The *exists predicate* consists of the keyword **exists**, followed by a collection reference, which may be a subquery. The *exists predicate* is *true* if the referenced collection is non-empty. The following example shows the use of *exists* with a subquery as the collection reference.

[8]Space limitations do not allow us to specify the algebraic expression as a single formula. Simple back substitution can be used to reconstruct this formula.

Example 3.3 Return the maps with areas where senior citizens live.

T: **select** o
from o **in C_map**
where exists (**select** p
from p **in C_person**, q **in C_dwelling**
where $(p.B\_age() \geq 65$
and $q = p.B\_residence()$
and $q.B\_inZone() \in o.B\_zones()))$
C: $\{\ o \mid \mathbf{C\_map}(o) \wedge \exists p(\mathbf{C\_person}(p)$
$\wedge \exists q(\mathbf{C\_dwelling}(q) \wedge p.B\_age \geq 65$
$\wedge\ q = p.B\_residence$
$\wedge q.B\_inZone \in o.B\_zones))\}$
A: $R1 \leftarrow \mathbf{C\_person}_p\ \sigma_{p.B\_age \geq 65}$
$\left(\mathbf{C\_map}_o \bowtie_F \langle \mathbf{C\_dwelling}_q,\ R1_p \rangle\right) \triangle_{p,q}$
where F is the predicate:
$q = p.B\_residence \wedge q.B\_inZone \in o.B\_zones$

The universal quantifier is expressed by the *forAll predicate*, which consists of the keyword **forAll**, followed by a *range variable list* and a *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 following example query illustrates the use of *forAll*. The subquery defines the range of p and $p.B\_low > 5000$ defines the Boolean formula that must be satisfied by every p in the range in order for an o to be returned.

Example 3.4 Return all maps that describe areas strictly above 5000 feet.

T: **select** o
from o **in C_map**
where forAll p **in** (**select** q
from q **in C_altitude**
where $q \in o.B\_zones())$
$p.B\_low() > 5000$
C: $\{\ o \mid \mathbf{C\_map}(o) \wedge \forall p(\neg \mathbf{C\_altitude}(p)$
$\vee\ \neg(p \in o.B\_zones) \vee p.B\_low > 5000)\}.$
A: $R1 \leftarrow \mathbf{C\_altitude}_p\ \sigma_{\neg(p.B\_low > 5000)}$
$\mathbf{C\_map} - \left(\left(\mathbf{C\_map}_o \bowtie_{p \in o.B\_zones} R1_p\right) \triangle_p\right)$

In the previous examples, path expressions only occur in the *where clause*. Path expression can also occur in the *select* and *from* clauses of a query. The

following example illustrates the use of path expressions in the *select* clause.

Example 3.5 Return the dollar values of the zones that people live in.

T: **select** $p.B\_residence().B\_inZone().B\_value()$
from p **in C_person**
C: $\{ o \mid \exists p(\mathbf{C\_person}(p)$
$\wedge o = p.B\_residence.B\_inZone.B\_value)\}$.
A: $\left(\mathbf{C\_person}_p\ \gamma^o_{o=p.B\_residence.B\_inZone.B\_value}\ \right) \triangle_p$
This has a simplier form using map as follows:
$\mathbf{C\_person}_o \gg_{o.B\_residence.B\_inZone.B\_value}$

The following example shows a path expression in the *from* clause and also introduces the *indexed expressions* of the *select* clause. Indexed expressions are used to specify behavioral projections in TQL.

Example 3.6 Return the zones that are part of some map and are within 10 units of water. Project the result over *B_title* and *B_area*.

T: **select** $o[B\_title, B\_area]$
from p **in C_map**, o **in** $p.B\_zones$, q **in C_water**
where $o.B\_proximity(q) < 10$
C: $\{ o[B\_title, B\_area] \mid \exists p \exists q(\mathbf{C\_map}(p)$
$\wedge\ \mathbf{C\_water}(q) \wedge o \in p.B\_zones$
$\wedge\ o.B\_proximity(q) < 10)\}$.
A: $R1 \leftarrow \mathbf{C\_map}_p\ \gamma^o_{o \in p.B\_zones}$
$R2 \leftarrow \left(R1_{p,o} \bowtie_{o.B\_proximity(q)<10} \mathbf{C\_water}_q\right) \triangle_{p,q}$
$R2\ \Pi_{B\_title,B\_name}$

Joins can be defined in TQL by specifying a list of expressions in the *select* clause. The result is a collection of product objects whose components correspond to the objects returned by each expression in the list. Each expression can be independently indexed to project behaviors on that particular component in the resulting product objects. The following example illustrates a join.

Example 3.7 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

T: **select** $p,\ q.B\_title()$
from p **in C_person**, q **in C_map**
where $p.B\_residence().B\_inZone() \in q.B\_zones()$
C: $\{p, o \mid \mathbf{C\_person}(p) \wedge \exists q(\mathbf{C\_map}(q)$
$\wedge\ o = q.B\_title$
$\wedge\ p.B\_residence.B\_inZone \in q.B\_zones)\}$
A: $\left(\mathbf{C\_person}_p \bowtie_F \left(\mathbf{C\_map}_q\ \gamma^o_{o=q.B\_title}\right)_{q,o}\right) \triangle_q$
where F is the predicate:
$p.B\_residence.B\_inZone \in q.B\_zones$

In addition to the *select statement*, TQL supports three binary operations: **union, minus**, and **intersect**. The syntax of these statements is defined below. Their semantics correspond to the typical set operations on the given collection references.

<*collection ref*> **union** <*collection ref*>
<*collection ref*> **minus** <*collection ref*>
<*collection ref*> **intersect** <*collection ref*>

TQL has a proven equivalence to the formal languages, which makes it easy to perform logical transformations and argue about its safety. The theorems and proofs of equivalence are given in [29].

4 Query Optimizer

TIGUKAT query optimizer follows the philosophy of representing system concepts as objects and is along the lines of [22]. The search space, the search strategy and the cost functions are modeled as objects (see Figure 4). The incorporation of these components of the optimizer into the type system provide extensibility via the basic object-oriented principle of subtyping and specialization.

There are several design decisions that need to be discussed. First, is the modeling of the algebraic operators as objects. These are defined as behaviors on `T_collection`. In the type lattice, they appear as instances of `T_algebra`, which is a subtype of `T_behavior` (instances are shown as circles in Figure 4 and types are shown as rectangles). The execution algorithms for these operators are appropriately modeled as functions (as instances of `T_algOp`). `T_algOp` is defined as a subtype of `T_context`, which is a type that models functions, all of whose arguments have been marshaled. We note that there may be many different algorithms to implement each algebraic operator (e.g., nested loop join, merge-sort join, and hash join). Thus, there may be many implementation functions as instances of `T_algOp`.

The second design decision deals with the compilation of query results in the translation of a TQL expression to an algebraic expression. Algebra expressions are commonly represented as processing trees (PTs) [20]. In relational systems, a processing

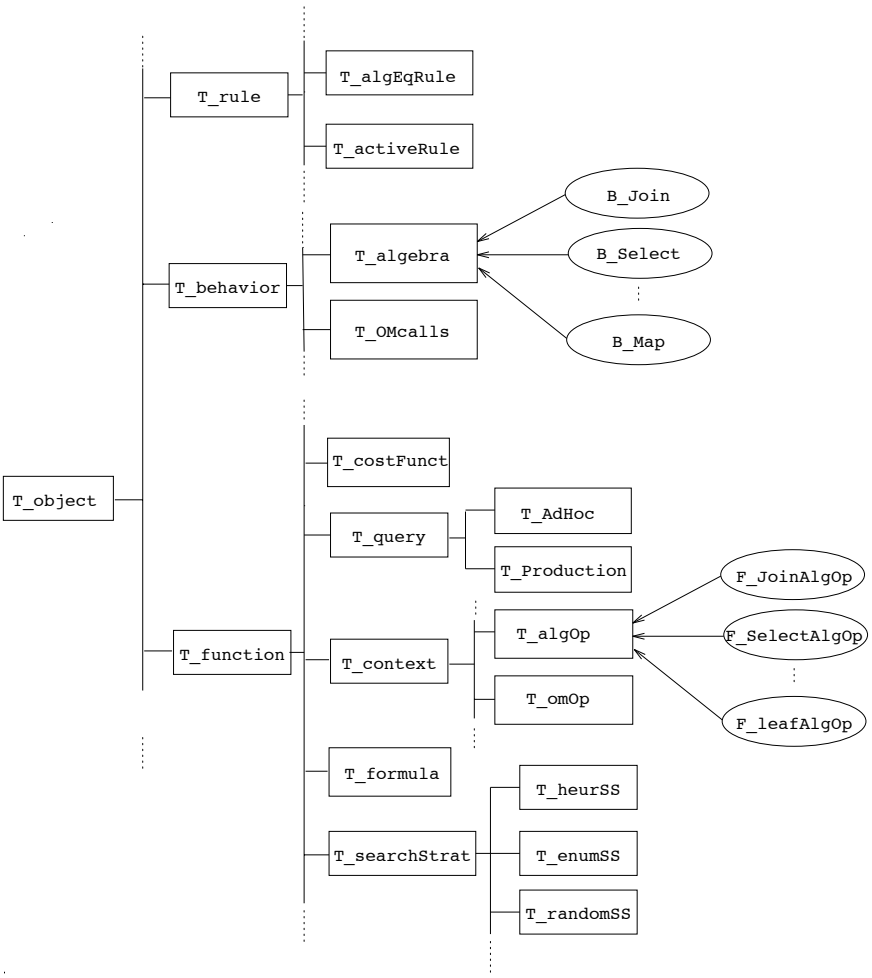

Figure 4: Optimizer as part of the type system

tree is a labelled tree where the leaf nodes represent relations and the intermediate nodes correspond to relational algebra operators. The edges represent temporary results generated by these operators. In our case, processing trees are constructed so that the nodes uniformly correspond to the functions implementing the algebraic operators. A PT is defined recursively with the root node of a PT being an algebraic operator of type `T_algOp` whose children are all of type `T_algOp` as well. In this way, a PT is modeled as an object.

The application of equivalence preserving transformation rules that are defined for the algebraic operators result in the generation of a family of PTs for a given query. The essence of query optimization is to find the PT within this family that corresponds to the most efficient execution plan for the query. The family of PTs is commonly called the *search space* and the algorithm that determines how this space is searched for the "best" PT is the *search strategy*. Usually these two are coupled together. However, in an extensible query optimizer such as ours, they need to be decoupled. We model search strategies as objects; specifically, `T_searchStrat` is defined as a subtype of `T_function` (see Figure 4), and is further specialized into enumerated search strategies (`T_enumSS`), randomized search strategies (`T_randomSS`), and heuristics-based strategies (`T_heurSS`).

The final optimization-related concept that needs to be incorporated into the model is the cost function. Cost functions determine the cost of executing the query according to each PT. We model them through `T_costFunct`, which is a subtype of

`T_function`.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer the extensibility inherent in object models. The optimizer implements a control strategy that associates a search strategy and a cost function to each query. The database administrator has the option of defining new cost functions and new search strategies or transformation functions for new classes of queries as they are developed.

5 Prototype Implementation

The TIGUKAT object model is implemented on top of EXODUS [6]. EXODUS is a fairly full-fledged system providing many aspects of DBMS functionality. We actually use only its storage system (called EXODUS Storage Manager – ESM for short) to provide persistence for TIGUKAT objects.

The general architecture of our implementation is shown in Figure 5. The shaded modules are currently being implemented. The other modules are included to indicate the general architectural set-up and the on-going research work.

ESM is at the lowest end of the system responsible for persistent storage on disk. ESM supports a client-server topology where the client module is linked with the host application program and interacts with the server, which may be running on the same machine or a different one. The TIGUKAT object model implementation is linked with ESM's client module to form an executable module that requires the server process to be running to perform persistent I/O operations. The TIGUKAT object model implementation consists of a library of TIGUKAT object implementations comprising the complete primitive object system, including behaviors, functions, and macros for atomic object creation.

The current prototype is implemented in C++. Since TIGUKAT has a language-independent object model, the type system of TIGUKAT is not mapped to that of C++. Instead, a single foundation C++ class, `TgObject`, is defined as the principal template for instantiation of all other objects. This approach ensures the uniform representation of all objects in the system, since they may each be treated as an instance of `TgObject`. The semantics of TIGUKAT objects is buried within the `TgObject` structure. Following this approach, the TIGUKAT model can be implemented using any programming language that suffices in building the foundation primitive system (e.g., SmallTalk [14]).

An important problem to solve is behavior application, since a particular behavior has to be bound to a particular function corresponding to the implementation of that behavior. Subtyping and redefinition of implementation introduce difficulties in determining a corresponding function and requires dynamic binding. Behavior application involves retrieving an appropriate function based on the behavior and the type of the receiver, and executing this function using the receiver and any argument objects. Since behavior application is fundamental to TIGUKAT, we have opted for a relatively simple, but fast mechanism, at the cost of bearing the consequential memory overhead. The system maintains a *dispatch cache* that is essentially a matrix with behaviors along one axis and types along the other. Each entry represents an implementation for a possible (behavior, type) combination. This cache is a statically allocated, volatile structure that needs to be reinitialized on program startup. The size of the cache is proportional to the total number of unique behaviors in the system and the total number of types in existence. We sacrifice memory usage for quick response time during execution, but as proposed in [2], an incremental coloring algorithm would help to substantially reduce this excessive memory consumption. We have not yet incorporated this optimization into the current prototype.

A further complication is introduced due to the separation of the stored and computed functions. Although the object model does not distinguish between stored and computed functions at the conceptual level, this distinction needs to be addressed in the implementation. Consider that some behavior is associated with a stored function. On invocation, that function requires access to a memory location (data field or slot) within the physical structure of the object it was invoked on. The function either places an object into this slot or retrieves one from it. The stored function accesses the concerned memory location via primitive system provided *set* and *get* accessor functions. Note that for computed functions, these slots have pointers to code rather than values. Due to subtyping, the number of slots and their addresses may change. To solve the slot access problem, we have chosen to implement a *supplementary cache* that contains information about which behaviors are associated with stored functions and which are associated with computed functions. The

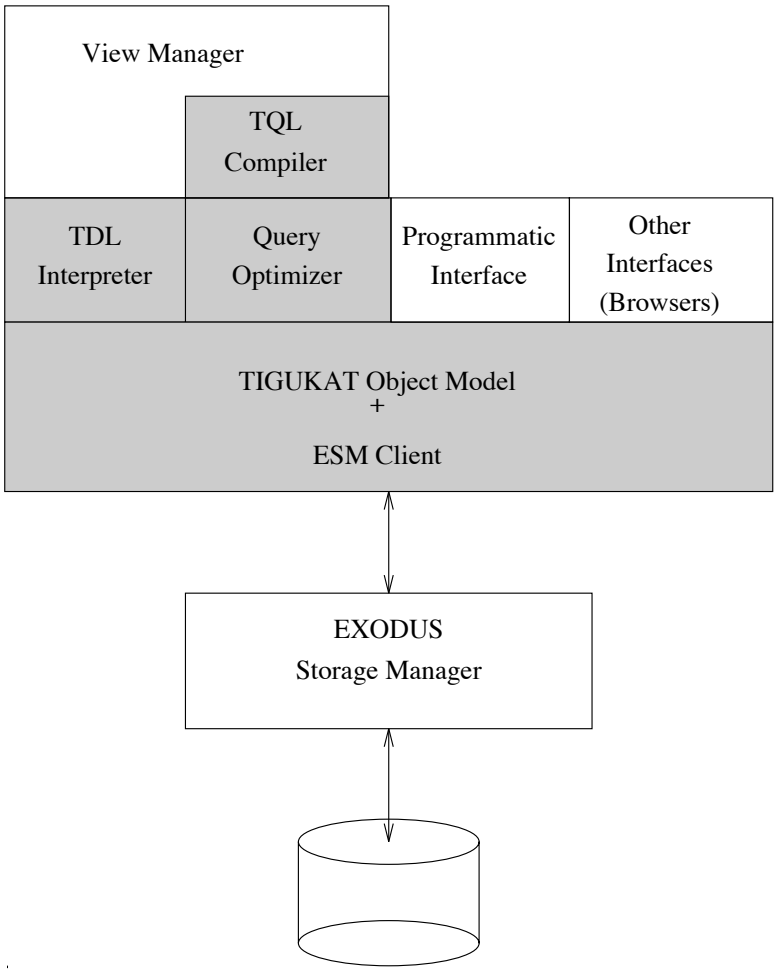

Figure 5: The TIGUKAT System Architecture

supplementary cache is a matrix similar to the dispatch cache. For every entry in the dispatch cache, there is a corresponding entry in the supplementary cache that indicates the stored or computed nature of the associated function. The supplementary cache makes its information content available to the type creation process to resolve conflicts. The implementation inheritance mechanism (part of the type creation process) uses this information and attempts to resolve which of the inherited behaviors should be associated with stored functions and which ones with computed functions. For stored functions, information about *set*/*get* pairings are maintained in a third cache matrix.

The current prototype is a proof of concept implementation to demonstrate the feasibility of implementing an extensible and uniform core object system. Several performance issues were addressed during its development as described in this section. However, there are still many others to consider and we are working on these as part of the future research. The full implementation of the TIGUKAT object model is presented in [15], the implementation of the language compiler is discussed in [23] and the details of the query optimizer can be found in [26].

6 Conclusions

In this paper, we provide an overview of the TIGUKAT object management system under development at the Laboratory for Database Systems Research of the University of Alberta. TIGUKAT has a uniform behavioral object model where everything

is a first-class object with well-defined behavior, and the only means of access to the object base is through behavior application.

The TIGUKAT object model is implemented on top of EXODUS. We have also defined a formal 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 its implementation is ongoing. The optimizer is being developed as a uniform extension to object model and will therefore be integrated with the model, just as the query model has been.

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

Biographies

M. Tamer Ozsu (ozsu@cs.ualberta.ca) is a Professor of Computing Science at the University of Alberta. His research interests are in distributed databases, object-oriented systems and interoperability issues. He is the author, co-author or co-editor of a number of texts including *Principles of Distributed Database Systems* (Prentice-Hall, 1993), *Distributed Object Management* (Morgan Kaufmann, 1993) and *Advances in Object-Oriented Database Systems* (Springer-Verlag, 1994). He holds the McCalla Research Professorship for 1993/94.

Randal Peters (randal@cs.ualberta.ca) is currently studying for his Ph.D. degree in Computing Science at the University of Alberta. He developed the TIGUKAT object model and query model, and is currently working on incorporating schema evolution and view mangement into TIGUKAT. He is planning to defend his Thesis in the fall of 1993.

Boman Irani (bomani@bnr.ca) obtained his M.Sc. degree in Computing Science from the University of Alberta in 1993. He was responsible for the implementation of the TIGUKAT object model. He is currently with BNR in Ottawa.

Anna Lipka obtained her M.Sc. degree in Computing Science from the University of Alberta in 1993. Her responsibilities included the design and implementation of the TIGUKAT user languages. She is currently studying for her Ph.D. degree at the University of Toronto. She can presently be reached at internet address anna@cs.ualberta.ca.

Adriana Muñoz (adriana@cs.ualberta.ca) is currently studying for her M.Sc. degree in Computing Science at the University of Alberta. She has designed and is implementing the query optimizer for the TIGUKAT query language.

Duane Szafron (duane@cs.ualberta.ca) is an Assistant Professor of Computing Science at the University of Alberta. His research interests include object-oriented computing, programming environments and user interfaces. He received a Ph.D. from the University of Waterloo, and a B.Sc. and M.Sc. from the University of Regina.

References

[1] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical report, INRIA, France, 1993.

[2] P. André and J. Royer. Optimizing Method Search with Lookup Caches and Coloring. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 110–123, September 1992.

[3] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2: An Object-Oriented Database System. In *Proc. of the 2nd Int'l Workshop on Object-Oriented Database Systems*, pages 1–22. Springer Verlag, September 1988.

[4] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *ACM*

Transactions on Office Information Systems, 5(1):3–26, January 1987.

[5] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[6] M. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. The Architecture of the EXODUS Extensible DBMS. In M. Stonebraker, editor, *Readings in Database Systems*, pages 488–501. Morgan Kaufmann, 1988.

[7] M. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 413–423, September 1988.

[8] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP: An Algebra Based Query Language for an Object-Oriented Database System. *Data & Knowledge Engineering*, 5:333–352, 1990.

[9] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. of the 2nd Int'l Workshop on Database Programming Languages*, pages 80–102, June 1989.

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

[11] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[12] L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proc. 1st International Conference on Information and Knowledge Management*, pages 17–26, November 1992.

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

[14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 2 edition, 1989.

[15] B.B. Irani. 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.

[16] W. Kent. Important Features of Iris OSQL. *Computer Standards & Interfaces*, 13:201–206, 1991.

[17] S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA '86 Conf. Proc.*, pages 406–416, September 1986.

[18] K. Kim, W. Kim, and A. Dale. Cyclic query processing in object-oriented databases. In *Proc. 5th Int. Conf. on Data Engineering*, pages 564–571, 1989.

[19] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[20] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. 12th Int. Conf. on Very Large Databases*, pages 128,137, 1986.

[21] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM*, 34(10):50–63, October 1991.

[22] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. 17th Int. Conf. on Very Large Databases*, pages 363–373, 1991.

[23] A.P. Lipka. 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.

[24] F. Manola and A.P. Buchmann. A Functional/Relational Object-Oriented Model for Distributed Object Management. Technical

Memorandum TM-0331-11-90-165, GTE Laboratories Incorporated, December 1990.

[25] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In K.R. Dittrich and U. Dayal, editors, *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Computer Science Press, 1986.

[26] A. Muñoz. The design and implementation of an object-oriented query optimizer for TIGUKAT. Master's thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, 1994. (in preparation).

[27] M.T. Özsu, U. Dayal, and P. Valduriez (eds.). An introduction to distributed object management. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, 1994.

[28] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System. In *Proc. of the Second Int'l. Conf. on Information and Knowledge Management*, pages 403–412, November 1993. A full version of this paper is available as University of Alberta Technical Report TR93-01.

[29] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The Query Model and Query Language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, June 1993.

[30] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR92-14, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, October 1992.

[31] D.D. Straube and M.T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[32] L. Yu and S.L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proc. 7th Int. Conf. on Data Engineering*, pages 670–677, 1991.