PESTO: An Integrated Query/Browser for Object Databases

Michael Carey† Laura Haas† Vivek Maganty‡ John Williams†

†IBM Almaden Research Center
San Jose, CA
{carey, laura, williams}@almaden.ibm.com

‡Computer Sciences Department
University of Wisconsin, Madison, WI
maganty@informix.com

Abstract

This paper describes the design and implementation of PESTO (*P*ortable *E*xplorer of *ST*ructured *O*bjects), a user interface that supports browsing and querying of object databases. PESTO allows users to navigate the relationships that exist among objects. In addition, users can formulate complex object queries through an integrated query paradigm ("query-in-place") that presents querying as a natural extension of browsing. PESTO is designed to be portable to any object database system that supports a high-level query language; in addition, PESTO is extensible, providing hooks for specialized predicate formation and object display tools for new data types (e.g., images or text).

1 Introduction

The Garlic project at the IBM Almaden Research Center [Care95] is developing a system and associated tools for managing large quantities of heterogeneous multimedia information. The goal of Garlic is to permit both traditional and multimedia data residing in a variety of existing data repositories (including relational databases, document managers, image repositories, and files) to be presented to application developers and end users via a unified, object-oriented schema. The heterogeneous data can then be queried uniformly and manipulated using an object-oriented dialect of SQL. One component of this project, which is joint work between IBM Almaden and the University of Wisconsin, is the development of a graphical user interface called PESTO (*P*ortable *E*xplorer of *ST*ructured *O*bjects). We refer to the PESTO interface as a *query/browser*, as it marries navigational object browsing with declarative querying; it integrates browsing and querying via a "query-in-place" paradigm that provides a powerful yet natural user interface for exploring the contents of object databases.

The rest of this paper describes the design of PESTO. Like other object database browsers, PESTO provides a hypertext-like navigational capability that enables users to navigate the relationships between database objects. In addition, PESTO enables users to query the contents of an object database without having to write (or even look at) object queries; querying is treated as a natural and direct extension of browsing. As a result, the query facilities of PESTO are intuitive yet powerful, supporting basic query capabilities such as selections, value-based joins, negation, and disjunction; they also enable complex object queries involving structural predicates and universal quantification to be expressed in an intuitive manner. While PESTO has been developed as part of the Garlic project, it can be ported to any system that supports an object-based data model and a declarative query interface. In fact, in addition to Garlic, it currently runs on top of an SQL-based interface to the ObjectStore database system, described in [Kier95], and that version of PESTO was used to generate the figures and queries shown throughout the paper.

The remainder of this paper is organized as follows. Section 2 briefly reviews related work on user interfaces and explains how PESTO differs from existing database browsing and query tools. Section 3 describes the basic look and feel of PESTO, conveying its integrated approach to browsing and querying through the use of several examples; PESTO's support for iterative query refinement is also described briefly. Sec-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996**

tion 4 then discusses PESTO's query capabilities more technically, focusing on PESTO's support for complex OODB queries and on the semantics and the limitations of PESTO as a query language. Section 5 briefly highlights a few interesting details of PESTO's current implementation, including its portability features and the way that new data types (such as images) and associated predicate formation tools can be added. Finally, Section 6 summarizes the paper and discusses our future plans for PESTO.

2 Related Work

We begin here with a brief review of related work on graphical database interfaces. We then highlight ways that PESTO differs from this work.

2.1 Commercial Interfaces

Virtually all relational database system vendors offer graphical interfaces for their systems, and a number of other vendors sell graphical query and application development tools that run against multiple relational database engines. Existing commercial products have been influenced by early research in this area, including efforts such as Timber [Ston82], FORMANAGER [Yao84], FADS [Rowe85], and of course, the seminal work on QBE [Zloo77]. However, commercial relational front-ends such as Access, Paradox, Visualizer, BusinessObjects, and FindOut! have long since surpassed the early work due to advances in workstations, window systems, and user interface toolkits.

Most commercial relational front-ends allow users to choose among several "views" of their databases. For example, Access provides four: a datasheet view that displays data from a table or query in a tabular format, a form view that displays a single record at a time, a query view in which the user can form a new query or edit an old one, and a report view in which data is formatted, usually for printing. These views are separate from one another; for example, the query view, even in Paradox – which provides an excellent implementation of the Query-by-Example paradigm – is totally separate from the data views that are offered. Thus, a strong distinction is made in these products between the act of query formation and the act of browsing a query's result set. This is quite different from the query-in-place approach that PESTO takes; this difference is particularly important in the world of object databases, as Section 3 will show.

Some commercial relational front-ends now provide users with the ability to specify the "business objects" that they want to work with. Good examples of such systems are BusinessObjects 3.1 and Open Data's FindOut!. "Business objects" are typically just business-relevant views of the underlying data that are set up by a database administrator to make browsing and querying easier for end users; they should not be confused with the kinds of objects that object database systems are intended to manage. The closest these interfaces come to true object support is a feature of FindOut!, which uses the relationships defined among the object views to allow users to do simple navigation among their objects. However, even the Findout! interface provides no support for the more powerful kinds of object queries (e.g., queries over nested sets of inter-object references) that PESTO supports.

One other type of interface is also relevant to PESTO. Web browsers, e.g., Mosaic and Netscape, have taken the world by storm. This is due in part to their natural browsing paradigm and in part to the world that they provide access to. However, these interfaces are designed for a world without a schema, where the data model consists of objects and (mostly) untyped inter-object references; thus, they do not offer object query facilities. One of our goals in developing PESTO has been to provide a similarly friendly browsing interface, and to augment this interface with an equally natural paradigm for integrating querying and query refinement with browsing.

2.2 Related Research

In general, user interfaces have been neglected as a database research topic [Ston89, Ston93]. Still, a body of work exists in this area; a comprehensive survey and taxonomy of graphical user interfaces for database systems can be found in [Bati91]. Aside from the early work on graphical relational interfaces, most research has focused on the design of interfaces for databases based on richer data models such as E-R, semantic, and object-oriented models.

Visual interfaces for database *browsing* are very attractive for systems based on richer data models, as such models make explicit the relationships between data objects in the database. Good examples of visual browsers that have been developed for such data models include KIVIEW [Motr88], Databrowse [Catt88], LID [Fogg84], and OdeView [Agra90]. KIVIEW introduced the notion of *synchronous* browsing of related objects; OdeView also emphasized this notion. Synchronous browsing is very important in PESTO as well. Unlike KIVIEW, where users explicitly indicate the synchronizing links, PESTO uses an implicit subwindowing approach that we believe is more convenient and intuitive. PESTO is closest in style to OdeView; we were heavily influenced by OdeView's browsing facilities. However, OdeView implements querying and browsing as separate mechanisms, as do virtually all other interfaces that we have seen. Also, OdeView requires object class definers to provide certain display- and query-oriented functions for their classes; in contrast, PESTO's object displays and query interactions are schema-driven and require no coding.

Work on an early graphical query interface for the functional data model, based on directly extending the QBE approach, can be found in [Heil85]. Object database interfaces that support graphical query formation are typified by Pasta-3 [Kunt89] and SNAP [Bryc86] (though SNAP could also be classified as a schema browser). In SNAP, queries are posed against the database *schema* to return browsable data, whereas the query context for PESTO users is at the data level. Pasta-3 provides a drag-and-drop paradigm for graphically forming and refining queries. Like Pasta-3, PESTO provides support for query refinement, but PESTO differs in its integration of querying and browsing. In Pasta-3, the query and browser windows are maintained separately. We feel that PESTO, with its support for querying at the instance level, and its blurred distinction between querying and result browsing, provides a more intuitive interface for exploring object databases.

In addition to these interfaces, a number of interfaces provide graphical support for browsing database schemas, including ISIS, GUIDE, SKI, and OPOSSUM (see [Care96] for references to these and other systems). Our work on PESTO has focused mainly on support for exploratory access to database objects; we expect to enhance PESTO's support for schema exploration later, drawing on related work in this area. Finally, the research literature on database user interfaces also includes papers on tools for building graphical interfaces, graphical primitives for manipulating object data, and novel ways of visualizing data. These papers are less directly relevant to PESTO.

2.3 What's New About PESTO?

There are two key differences between PESTO and the work just discussed. One difference is that, unlike the commercial tools, PESTO is designed for exploring object databases. Resulting challenges include the need to provide graphical support for path predicates, set-valued attributes, and method invocations, as well as continued support for *ad hoc* joins. These challenges are important, given the increasing commercial focus on object-oriented and object-relational data models. Few object-oriented database systems provide declarative *ad hoc* query support or graphical interfaces other than browsers. PESTO is unique in its support for advanced object query features.

Another important difference is PESTO's integrated *query-in-place* support for both querying and browsing. Existing graphical query tools make a sharp distinction between query formation and answer set browsing. In contrast, PESTO allows users to directly restrict the displayed data, much as some relational front-end tools (e.g., table browsers) permit updates in place. Moreover, PESTO generalizes existing notions of *filtered browsing* by allowing filters to be specified (1) in place, on the browse structure, (2) on any level(s) of nested sets, and (3) with arbitrarily complex predicates (including explicit and implicit joins). Consequently, PESTO users can simultaneously browse objects at some levels of a complex object structure while querying (filtering) the objects at other levels. PESTO also provides support for iterative query refinement throughout a query/browse session.

3 Browsing and Query-In-Place

In this section we describe PESTO's basic support for synchronized browsing, its notion of *query-in-place*, and its support for iterative query refinement. We begin by presenting the schema for a simple object database that will be used in our examples.

3.1 Example Schema

Given below is an object schema for a hypothetical university database. ODL, the ODMG-93 DDL [Catt94], has been used to express the schema. The class definitions should be largely self-explanatory; note that the schema accomodates both undergraduate and graduate students. Each class has an associated collection (its *extent*) in which all instances of the class and its subclasses are recorded.

```
class Student (extent Students)
{ Int ss_no;
  String last_name;
  String first_name;
  Float gpa;
  Ref<Professor> advisor;
  Set<Ref<Department>> major;
  Set<Ref<Course>> taking; }

class GradStudent: Student
(extent GradStudents)
{ String office;
  String phone; }

class Professor (extent Professors)
{ String last_name;
  String rank;
  String area;
  Int phd_year;
  Ref<Image> photo;
  Ref<Department> dept;
  Ref<Course> course;
  Set<Ref<Student>> advisees; }
```

```
class Department (extent Departments)
{ String name;
  Ref<Professor> chair;
  Set<Ref<Professor>> faculty;
  Set<Ref<Student>> majors; }

class Course (extent Courses)
{ String id;
  String name;
  Ref<Text> description;
  Ref<Professor> instructor;
  Set<Ref<Student>> takers; }
```

3.2 Browsing Complex Objects

To begin a PESTO query/browse session, the user selects a database to explore and chooses one or more collections from which to begin querying and browsing. The bottom left-hand corner of Figure 1 shows PESTO's startup window, titled "OO-SQL/QB." This window lists the available databases; in Figure 1 the user has chosen to explore the database called UnivDB, so the window also lists all the top-level collections in the UnivDB database. In Figure 1 the user has chosen to use the Students collection as an entry point for browsing. PESTO is entirely schema-driven, so the student window (labeled "Students0") shown on the left side of PESTO's "Data Browser" window in Figure 1 is the default display window that PESTO produces for Student objects. Immediately underneath this student window's pulldown menu bar and row of buttons is a label bar indicating the source of the window. It contains the label "Students0" to indicate that it is a window for browsing the Students collection; "0" is needed because a user can browse a collection independently through more than one window, in which case the additional student windows would be labeled "Students1," "Students2," and so on. This bar also tells which element of the collection is being examined and how large the collection is.

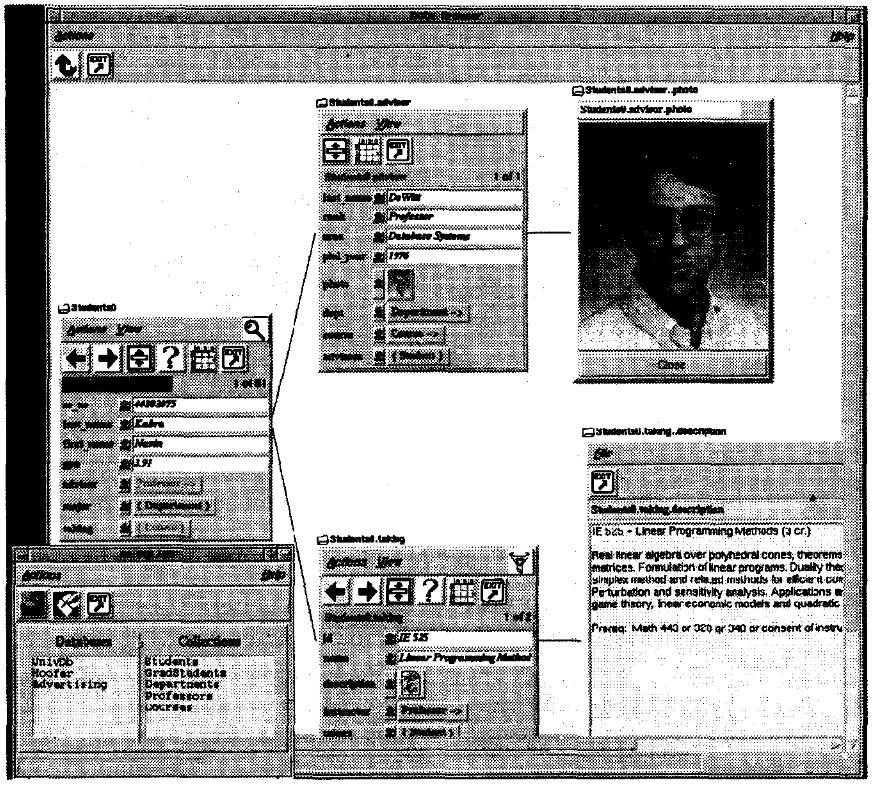
Each attribute of the Student class is visible in the default student window[1], and this window currently displays a Student object that was returned by the underlying object database system. Attributes of primitive data types, e.g., *ss_no* and *last_name*, are shown as being contained in the Student object. Attributes that are references or collections of references are presented as buttons that will bring up other windows for displaying the object(s) targeted by the reference(s). The *advisor* attribute is an example of a reference attribute, and in Figure 1 the user has clicked on it to bring up a window for the referenced Professor object; the user then repeated this action on the advisor window's *photo* attribute to bring up the advisor's photograph as well. The user has clicked on the button for the *taking* attribute to bring up a course window (labeled "Students0.taking," as per the window labeling convention described below), which currently shows the first of several courses that the displayed student is enrolled in, and on the course window's *description* button to bring up the associated Text object describing that course.[2] Each window created by navigating via button clicks from the student window has a label bar that indicates how it depends on the other windows (e.g., "Students0.advisor" for the professor window, "Students0.advisor.photo" for the associated Image window); in addition, lines are drawn to visually connect dependent windows in the browse area.

Near the top left of Figure 1's student window are PREVIOUS and NEXT arrow buttons for stepping through the objects in the Students collection. When a collection window is displaying the first (last) object in the collection, the PREVIOUS (NEXT) arrow is disabled and grayed out; otherwise, both arrows are enabled and shown in solid black. Because a student can take several courses, the course window also has PREVIOUS and NEXT buttons. This window represents a nested collection of objects, and at present it contains the first element of the taking set for the *currently displayed student* (Navin Kabra). The NEXT and PREVIOUS arrows of this window can be used to browse through the set of courses that this student is taking. The other windows shown have no NEXT/PREVIOUS arrows because they were derived from single-valued references, so each has just one associated object. When dependent windows are being displayed, browsing is synchronous (as in KIVIEW and OdeView) – when the user clicks on the NEXT (PREVIOUS) arrow in the student window, this window will advance to the next (previous) object in the Students collection. When this happens, the contents of the dependent windows change as well – the professor window displays that student's advisor, the course window displays the first of that student's courses, and so on. The course window's NEXT and PREVIOUS arrows can then be used to browse through the set of courses that this next student is enrolled in.

Continuing across the top row of buttons in the student window in Figure 1, the button to the right of the NEXT arrow is the EXPAND button for the window.

[1] Default display windows can be customized by using a pulldown menu to selectively hide attributes.

[2] By default, a PESTO reference button contains the referenced class name and an indication of whether it is for a reference or a set of references (Student.advisor versus Student.taking). Optionally, a class definer can override this by providing a special icon for references to a given class (Professor.photo, Course.description). A custom displayer can also be provided for the instances of a given class if desired. The image and text data types in Figure 1 illustrate such overrides.



IE 525 - Linear Programming Methods (3 cr.)

Real linear algebra over polyhedral cones, theorems
matrices. Formulation of linear programs. Duality thec
simplex method and related methods for efficient com
Perturbation and sensitivity analysis. Applications a
game theory, linear economic models and quadratic

Prereq: Math 440 or 320 or 340 or consent of instru

Figure 1: Synchronously browsing students.

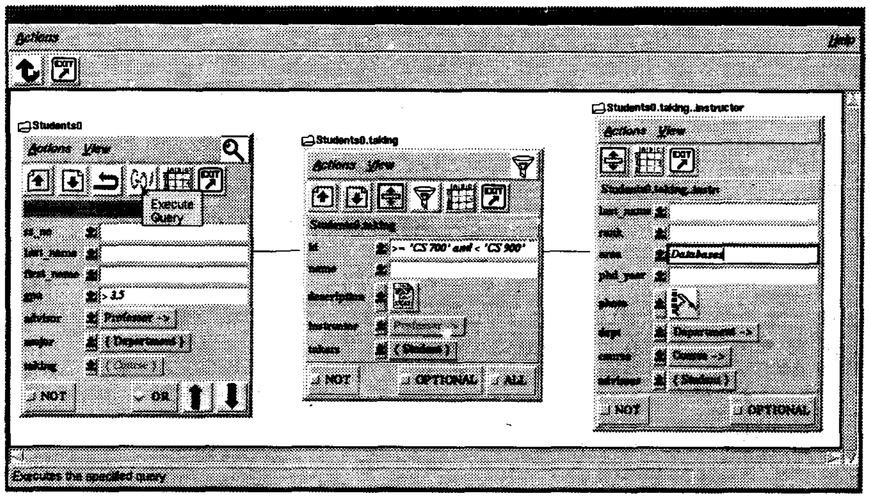
This button gives the user the option of seeing the window's objects as a member of their most specific applicable class – for example, clicking on the student window's EXPAND button will allow the user to view the extra information available for those students that are graduate students. To the right of the EXPAND button is the QUERY (?) button, which is discussed in the next subsection. Next to the QUERY button is the TABLE button, which will allow the user to see multiple objects of a single collection at a time. At the far right is the EXIT button, which closes the window and its dependent sub-windows.

3.3 Query-In-Place

While browsing all students and their courses might be of interest to some UnivDB users, many would prefer to browse only a selected subset of the students, or perhaps only a selected subset of their courses. This can be done by utilizing PESTO's *query-in-place* features to perform *filtered browsing* – i.e., by specifying and then browsing a subset of a given collection of objects. PESTO allows any collection-valued window – whether top-level or nested – to be the target of a query, and browsing the result objects of a query has the look and feel of browsing an unrestricted object collection (except that only objects satisfying the query specification are visible).

As an example, suppose a user is browsing students together with their courses and associated instructors, but really only cares about students with grade point averages over 3.5 who are taking one or more computer science graduate courses from instructors whose research area is databases. To restrict the set of students being browsed, the user can use the QUERY button in the student window to put it (and its dependent windows) in query mode; the boxes associated with each attribute value become predicate entry boxes. The user can then type their GPA predicate into the student window, graduate-level CS course predicate into the course window, and instructor area predicate into the professor window, as shown in Figure 2. Clicking GO tells PESTO to execute the query and return to browse mode to view the results.[3] The resulting browse state is similar to basic synchronous browsing. Now, however, the set of browsable Student objects is restricted to those satisfying the user's criteria, and the magnifying glass icon in the student window will be highlighted to remind the user that the query is active there. The NEXT and PREVIOUS buttons in the student window enable the user to browse through all students that satisfy the query predicate. The NEXT and PREVIOUS buttons in the course window still enable the user to browse through any/all courses that the current student is taking, not just graduate CS

[3]The box that says "Execute Query" appears in the figure because PESTO's "balloon help" feature has been enabled.



gpa: > 3.5

id: >= 'CS 700' and < 'CS 900'

area: Databases

Executes the specified query

Figure 2: Students with high GPAs who are taking a CS graduate course taught by a database professor.

courses taught by database professors, as the purpose of the predicate was only to restrict the set of Students being browsed.

In the example above, the user wished to see only those elements in a top-level collection (e.g., Students) that satisfy some criteria. Because PESTO is intended for object databases, where objects can contain nested collections of (e.g., sets of references to) other objects, PESTO permits queries to be specified in *any* window associated with a collection – whether it is a top-level collection or a dependent collection. In contrast to the previous example, suppose that the user wishes to browse *all* students, but to see only their CS courses taught by database professors. Starting from the same browse state as the previous example, Figure 3 shows how this can be done. This time, the user clicks on the QUERY button of the course window to place the course window and its subordinate professor window in query mode; the student window will remain in browse mode this time. The user enters the desired predicates on courses and their instructors, as shown in the figure, and then hits GO to launch the course query. In the resulting browse state, the student window can be used to browse through all of the students, as in unfiltered browsing. However, the dependent course window now provides browse access, for each student, just to courses that are CS courses taught by database professors. The filter icon in the course window is highlighted as a reminder that this window is showing only a filtered subset of its associated objects (like the magnifying glass for a top-level collection). The user is now browsing a top-level collection (Students) in an unfiltered manner, but filtering the contents of a nested collection (Students.taking).

In general, PESTO allows simultaneous queries (filters) at as many levels as a user wishes. This generality is one of PESTO's most useful and unique features, as it allows users iteratively to browse through a complex database and narrow their browsing scope as they go. Filter predicates can be added to any window associated with a collection, at any level of nesting, at any time during a browse session. Thus, users can begin by synchronously browsing a collection, decide that they don't want to see everything, add a filter, decide that they don't really want to see all of the objects in a given subwindow, add a filter to limit the objects being browsed there, and so on.

In addition to this query-in-place support for objects at multiple levels of nesting, PESTO provides a short cut for an important special case. Consider again the query shown in Figure 2, where the user asked to browse students with high GPAs who are enrolled in a graduate CS course taught by a database professor. PESTO's response to this query allowed the user to synchronously browse these students and *all* of the courses that they are taking (together with the instructors of those courses). This is what the user wanted in our earlier example. However, a different user might wish to use the students' course criteria (graduate CS courses taught by database instructors) to filter the courses as well – thereby viewing *only* those courses that caused each student to be selected for browsing. To support the case where a user wants to place an existential predicate on a nested collection, and to place the identical query predicate on the nested collection itself, PESTO provides a FILTER button in nested collection windows whenever a query on an ancestor window causes them to enter query mode. To use it, the user would proceed as in Figure 2 – but before saying GO, the user would click the FILTER button in the course window to ask PESTO to filter the Course objects as well.

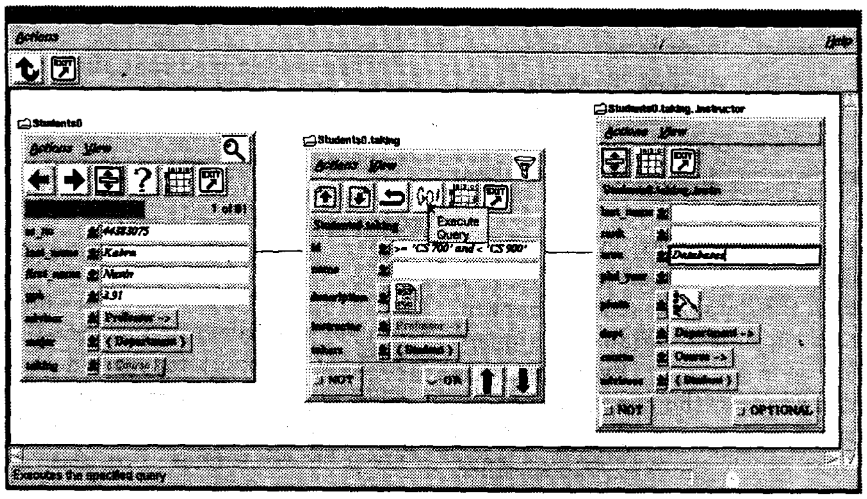


Figure 3: All students plus their CS graduate courses taught by database professors (if any).

3.4 Iterative Query Refinement

PESTO includes features that allow iterative query refinement throughout a query/browse session. PESTO provides support for returning to and then editing the previous query. Also supported is a query history mechanism that enables older queries to be located, refined, and resubmitted. Together, these features ensure that PESTO users can incrementally focus their attention on the objects of interest and can back up and try again if they discover that they have taken a "wrong turn."

3.4.1 Refining the Last Query

Whenever a user enters query mode, the latest query is recalled to the screen, ready for incremental editing. For example, if the user's query in Figure 2 turned out to be too selective, e.g., due to an overly high choice of GPA, the user could hit QUERY and modify that portion of the query predicate; hitting GO then launches the newly modified query. Also, PESTO allows aborting a query without losing the browse state. If a user gets part way through specifying a new query, and then decides not to proceed with it after all, the REVERT button (the "turn back" arrow visible when a window is in query mode) returns to the exact browse state – with the same objects in the same windows – that existed right before entering query mode.

These query refinement features also serve another purpose – recovery from disorientation during a complex query/browse session. After entering a query, although a magnifying glass or filter icon indicates that the query is active, no details of the query predicate are displayed while the results are being browsed. If the user forgets the details of their query, they can reenter query mode by pushing the QUERY button in the query's target window – causing PESTO to redisplay the query. Having refreshed their memory, they can then revert back to browsing.

3.4.2 Query History Support

Being able to edit the most recent query addresses the most common case, but it doesn't handle cases where a user makes an exploratory mistake (or a series of such mistakes) and wishes to back out through several queries – returning to an earlier, perhaps more successful query. To handle such cases, PESTO includes a query history feature when windows are in query mode. PESTO remembers the last M queries that targeted each window. The UP and DOWN arrow buttons in a query window can be used to walk back and forth through its history to select a previous query to edit. Once the desired query has been found and edited, it is appended to the window's query history as its most-recently-issued query.

4 Semantics & Complex Queries

In this section, we discuss PESTO's conceptual model and complex query capabilities.

4.1 Object-Oriented User Interaction

It should be clear that the interaction model that PESTO presents to its users is heavily *object-oriented*; there is a direct correspondence between each window in PESTO's browse area and an object in the underlying database.[4] PESTO's query-in-place approach is intended to allow users to directly examine and manipulate the displayed objects. As a result, in browse mode, each window keeps track of the OID of its displayed object; windows represent "live" objects. We

[4] One exception will be discussed later – *ad hoc* join queries lead to "synchronizer" windows that do not directly correspond to stored objects.

will not discuss updates or method invocation in detail here, but the PESTO design allows users to update an object displayed in a browse window by choosing an update action and then directly editing its contents; similarly, users can invoke an object's methods from a pulldown list provided in the display window's menu options.

PESTO's behavior (and semantics) in query mode is based on a similarly tight correspondence between windows and objects in the database. Each window in query mode represents an object variable in a query formed through the user's actions. Every query is rooted at a particular window, the one that the user placed in query mode, and its meaning is "let me browse the objects that satisfy the predicate specified here and in any/all dependent windows." Each top-level collection window represents an object quantifier over its associated collection in the resulting query. Each dependent collection window represents a quantifier that ranges over the appropriate collection in its parent window's current object. Dependent reference windows can be thought of similarly (i.e., as one-element collections), though most object query languages support path expressions that make explicit quantifiers unnecessary in this case.

To make this clear, the following is the OO-SQL [Kier95] that PESTO generates and associates with the student window when the user hits GO in Figure 2:

```
select distinct S, S.*  from Students S
where S.gpa > 3.5 and exists (
  select T from (S.taking) T
  where (T.id >= 'CS 700'
  and T.id < 'CS 900'
  and T.instructor is not null
  and T.instructor..area = 'Databases'));
```

As an aside, this discussion highlights a key difference between PESTO's query capabilities and those of QBE and its commercial descendants. QBE makes a strong distinction between queries and results – tabular forms are used to specify queries; in general, a separate result table is needed to browse the query results. In contrast, PESTO's query criteria are entered, in place, into browse windows that have simply been placed in query mode. PESTO displays the results in the same windows, thus retaining their complex object structure and enabling their component objects to be "live." As mentioned earlier, PESTO allows users to simultaneously browse at some levels of a complex object structure while querying at others. This is another feature that distinguishes PESTO from QBE-like interfaces.

4.2 Expressing Complex Queries

Earlier, we claimed that PESTO makes it "easy" for users to express complex queries. We now examine this claim by explaining how PESTO supports a wide variety of object queries. We then discuss certain kinds of queries that PESTO cannot express (and why).

4.2.1 Select/Project Queries

The most basic form of query in PESTO is roughly equivalent to a relational select/project query. PESTO's support for filtered browsing of a top-level collection, combined with its support for hiding some of the attributes of the objects to be displayed in a given browse window, provides this level of expressive power. As illustrated earlier, users specify their selection criteria by entering values or expressions into the attribute entry boxes of the collection's display window after putting it in query mode. Users can enter values to specify equality predicates (or SQL-style *like* patterns for strings), and PESTO supports a simple predicate language that has the usual comparators and logical connectives (*and*, *or*, and *not*) to allow complex predicates to be expressed for individual attributes. (E.g., see the course *id* predicate in Figure 2.) Between attribute boxes in a given window, PESTO's semantics are conjunctive, as one might expect. Finally, PESTO supports complex negative selection predicates by providing a NOT button in each query window; when pressed, this button negates the entire predicate implied by that window (and any dependent windows).

4.2.2 Complex Queries on Objects

References in an object database can be null-valued; e.g., a student may not have an advisor yet, or a self-paced course may not have an instructor. PESTO must therefore provide users with a way to specify whether or not they want the parent object to qualify when a query involves predicates on a dependent object and the reference is null. By default, dependent objects are said to be *mandatory*, meaning that a parent object is considered to satisfy its full query predicate only if a dependent object exists and satisfies its portion of the query predicate. For example, consider again the query of Figure 3, where, for each student, the user wants to browse CS graduate courses taught by database professors. Courses with a null *instructor* attribute will not be browsable in the result, as they do not qualify by default. To allow users to opt to see such courses as well, PESTO includes the button labeled OPTIONAL in the dependent professor window of Figure 3. Pressing this button, which appears in every dependent window of a query window, makes the existence of a dependent object *op-*

tional there instead of mandatory. This same idea applies to dependent collection windows. Back in Figure 2, the course *id* predicate and the instructor *area* predicate combine to specify that the user wishes to see students that are taking at least one CS graduate course from a database professor. If the OPTIONAL button were pressed in the course window there, the user would also see students who are taking no courses at all.

Collection-valued reference attributes raise another question – does the user want to require the predicate to be true for *some* members of the collection, or for *all* of its members? The former case corresponds to existential quantification, and the latter to universal quantification. To support both cases, PESTO provides an additional button, labeled ALL, whenever a dependent window corresponds to a collection-valued attribute. By default, a predicate in such a window has existential (or "some") semantics; pushing the ALL button means "this predicate must hold for *all* objects in this nested collection." For example, the query shown in Figure 4 means that the user wants to browse CS courses in which all enrolled students have high GPAs and an advisor, if any, who is a full professor.

PESTO's provision of OPTIONAL and ALL makes it possible for users to ask queries that would otherwise be complicated to express in an object query syntax. Compare Figure 4 to what the user would need to say in OO-SQL to achieve the same semantics:

```
select distinct C, C.*
from Courses C
where C.id like 'CS %'
and not exists (
  select T from (C.takers) T
  where not ( T.gpa > 3.5
  and (T.advisor is null or
       T.advisor..rank = 'Professor')));
```

Finally, it is worth noting that the ALL and NOT buttons in a dependent window can be combined to say "none." NOT helps to form the window's local predicate, while ALL modifies the type of quantifier associated with the window. For example, by pressing the course window's ALL and NOT buttons in Figure 4, the user could instead ask to browse CS courses in which *none* of the enrolled students have both high GPAs and an advisor, if any, who is a full professor (i.e., CS courses in which *all* enrolled students do *not* have both high GPAs and an advisor, if any, who is a full professor).

4.2.3 Disjunctive Queries

As mentioned earlier, PESTO supports a small predicate language for specifying conditions on attributes. Simple disjunctive predicates can be specified using this facility. To support a wider range of disjunctive queries, queries that would otherwise fall outside the expressive power of PESTO (unlike their conjunctive counterparts), PESTO provides "power users" with the notion of an *OR-stack* that can be associated with a collection window when it is the target of a query.

As an example, suppose that a user wishes to browse students with high GPAs who are taking CS courses taught by a database professor *or* with low GPAs who are taking English courses from a professor named "Smith." To express this request, the user would enter the first part of this disjunctive query just as in Figure 2. However, instead of pushing GO at this point, the user will instead push the OR button in the student window, thereby activating its OR-stack. In response, PESTO records the first part of the predicate and then clears the student window and its dependent windows so that another disjunct may be entered. The user can now fill in the rest of the conditions. The OR button is highlighted to indicate the presence of multiple disjuncts on the OR-stack, and the UP and DOWN arrows to the right of the OR button can be used to navigate the OR-stack; this allows each of the query's disjuncts to be viewed and modified while the window is in query mode.

4.2.4 *Ad Hoc* Joins (Links & Synchronizers)

As described in [Bati91], many graphical query tools that support object-like queries, such as E-R-based tools, only permit joins along paths in the database that are supported by existing relationships. In contrast, PESTO supports value-based linking as well – and in a way that fits naturally with the rest of PESTO's object query facilities. We introduce the concept of a *synchronizer*, which is a transient object resulting from a query that binds together two or more otherwise unrelated database objects. To users, PESTO synchronizers look like database objects that contain only reference attributes; they provide a handle for conveniently and *synchronously* browsing the results of an *ad hoc* join query in a way that is consistent with the treatment of other browsed collections. Such a concept is not necessary (and therefore not found) in relational query tools, as users of those tools browse result tables containing attribute data that has been copied out of the query's source tables. In contrast, PESTO's synchronizers enable the user to retain links to the underlying (live) objects in the database. As a result, all of PESTO's query-in-place and query refinement features work when a user is querying and/or browsing via synchronizers as well.

Suppose that a user wishes to browse student/professor pairs where the student and the pro-

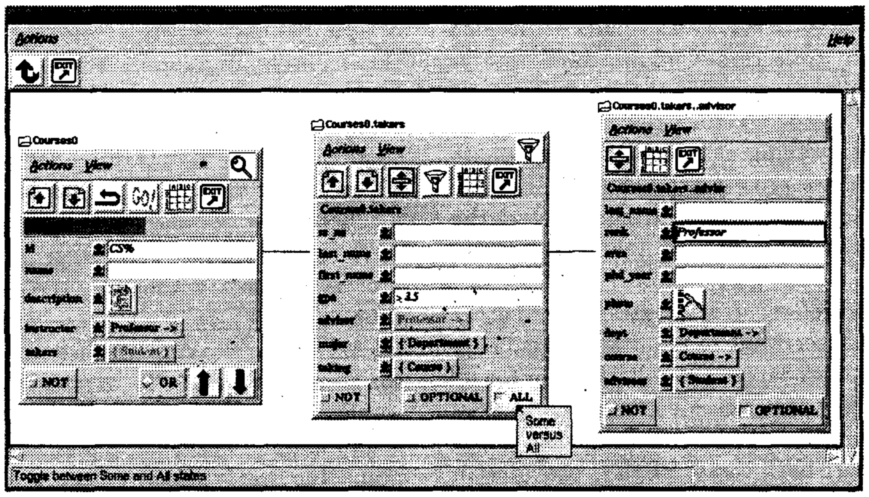


Figure 4: CS courses in which *all* students have high GPAs and full professors (if any) for advisors.

fessor have the same last name. They begin by opening independent top-level windows on the Students and Professors collections and placing one of them in query mode. To specify the join criterion, they *link* the *last_name* attributes by clicking on *last_name*, first in the window in query mode and then in the other window. PESTO detects the link request, highlighting the source and target attribute names as they are selected. When the link target is selected and found to be in an independent window, PESTO creates a synchronizer window, placing both previously independent windows under its control and in query mode. PESTO also displays a menu bar at the top of the browse area that shows the linking (comparison) operators that make sense given the selected link attributes' data type. Figure 5 shows the screen at this stage of the join process.

In this case, the user selects the "=" comparator. At this point, PESTO is ready for the user to enter any additional query predicates and press GO to execute the query. Additional predicates could be other links or path predicates rooted at either (or both) of the collections being joined. Note that the join query as a whole is rooted in the synchronizer window, which is where the GO button is located. Once the user hits GO, the synchronizer can be used to sequence through the results. It behaves like any other top-level object window with respect to result browsing; when its NEXT (PREVIOUS) button is pressed, PESTO will show the user the next (previous) student/professor pair whose last names match. When the user is done browsing the join results, pushing the EXIT button in the synchronizer window will free the student and professor windows (making them each independent top-level browse windows again) and remove the synchronizer from the screen.

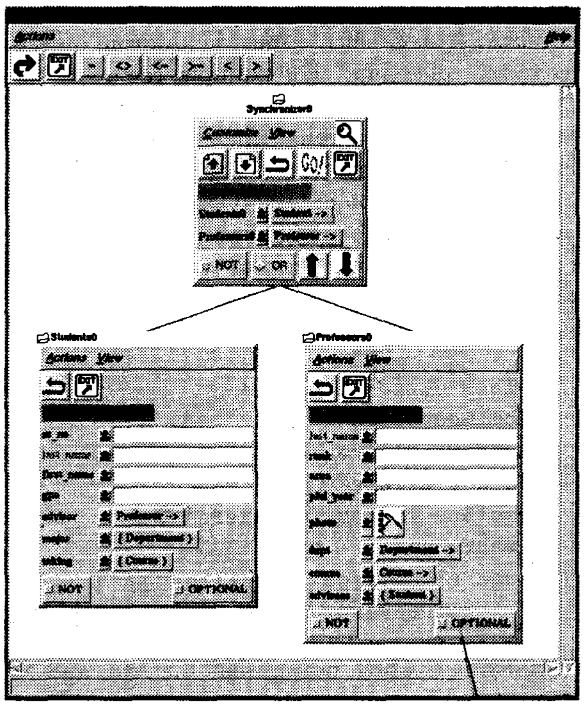


Figure 5: Student/Professor join (in progress).

PESTO's links can support a variety of queries. In addition to joins of independent collections, links can be used to join across existing dependent and top-level windows, in which case no top-level synchronizer window is created. Moreover, links can be made between simple attributes, pairs of reference attributes, a reference attribute and a nested collection of references, or a simple attribute value and a nested collection of values. By combining links with the features for negation and universal quantification, many complex queries can be expressed, such as "browse the students who are taking no courses taught by their advisor."

4.2.5 Type-Specific Predicates

Last but not least, PESTO also allows for specialized data types, like Text or Image data, that require type-specific query predicates when formulating queries (e.g., approximate image matching, or keyword searches for text). PESTO was heavily influenced here by the approach that the Garlic project's object SQL dialect takes to handling such predicates [Care95]; namely, they are expressed as method calls. PESTO supports an extensible set of type-specific methods called *pickers* for such data types. The class definer for a specialized data type is permitted to add a picker function that PESTO can call. This function interacts with the user, presumably by putting up a window, and is expected to return a string containing a method call or a boolean expression made up of method calls. (See [Cody95, Care96] for more about this approach.)

4.2.6 Limitations

PESTO is quite powerful in terms of the types of object queries that users can ask. Of course, while it supports quite a range of queries, it is not a relationally complete query facility (nor is it intended to be). It can express many of the queries that OO-SQL [Kier95] supports, including queries that involve selection, projection, path expressions, nested sets, existential and universal quantification, value-based joins, and type-specific predicates. In addition, we are prototyping extensions to support sorted results for collection windows, aggregates, and certain kinds of method calls in queries. All of this work is driven by PESTO's user interaction model – namely, that windows on the screen, in almost all cases, should represent live objects drawn from collections in the underlying database. This model is important for PESTO's intended usage, which is to support interactive exploration and manipulation of the contents of object databases. In addition, there is an important tradeoff to be made between the expressiveness of PESTO as a query language and its usability and consistency as a tool for non-expert users; we do not wish to sacrifice the latter in favor of the former.

So what are the limitations and/or oddities of PESTO as a query language? Due to PESTO's object-oriented interaction model, relational projection is not actually supported, though it can be simulated via attribute hiding. Also, relational joins – which create new objects out of attribute data copied from pairs of other objects – are not supported, though matching and pairing is supported via links and synchronizers. The relational union operation is also not supported. PESTO's OR-stack allows users to specify a limited class of union queries, where the unioned subqueries all range over the same underlying collection. However, PESTO is not powerful enough to express unions that combine objects from multiple collections, though this could potentially be addressed using a synchronizer-like notion. In any case, we do not intend for PESTO to ever support the unioning of objects of different types that are union-compatible only in the relational sense; that would be wholly incompatible with our object-oriented user interaction model. Finally, while we plan to add some degree of support to PESTO for sorting, aggregates, and grouping, the power of these facilities will again be limited to what can be done within the confines of our object-oriented model.

5 Implementation Details

To enable fast prototyping, PESTO was implemented at the University of Wisconsin using Tcl/Tk [Oust94]. PESTO currently runs on the Garlic system [Care95], which supports an object-extended dialect of SQL. Garlic runs under AIX and today provides object-based access to data managed by DB2/6000, ObjectStore, and the QBIC [Nibl93] image manager. PESTO also runs on the OO-SQL interface to ObjectStore [Kier95]. This section touches on two aspects of PESTO's current implementation, portability and support for content-based queries involving specialized data types; more details can be found in [Care96].

5.1 Portability

For portability, PESTO uses a small set of Tcl procedures to interact with the underlying object database system. This interface is called PASTA (*P*ortability *A*bstraction for *ST*ructured *A*rchives). It allows PESTO to connect to an object database, discover its root collections, obtain type information, set up a portal for accessing the objects in a collection, submit a query and get a portal for its answer set, sequence through a portal's objects, retrieve an object's content by OID, apply a method, and so on.

5.2 Specialized Data Types

As mentioned earlier, PESTO allows implementors of specialized classes, e.g., text, image, and other multimedia data types, to provide customized displayers and/or predicate formation tools for their classes. We refer to customized predicate formers tools as *pickers*. The implementor of a new displayer or picker informs PESTO of its presence by adding it to an internal Tcl table. Adding a picker requires the provision of a set of four Tcl routines; this interface has been used in Garlic to import the QBIC image query interface. When a user clicks a reference button (in query mode) for a data type with its own picker, PESTO calls the picker's "pick" routine to cause a type-specific picker window to appear; the user then interacts with this window. When the user hits GO in the query's top-level window, PESTO calls the picker's "get" routine to request a query string (which typically applies a matching method of some sort to a multimedia object) that PESTO can *and* into the query; "get" also causes the picker window to unmap itself from the display. To support PESTO's query history, the picker interface includes a "repick" routine to reestablish the picker window state from a previously returned query fragment. Finally, pickers are shut down by PESTO via an "exit" function.

6 Conclusions

We have described PESTO, a query/browser for object databases. Like other browsers, PESTO allows users to navigate a database by following the relationships among its objects. In addition, PESTO supports a *query-in-place* paradigm that allows users to query as well as browse the database in a natural and integrated fashion, without actually writing object queries. We believe that PESTO's query facilities strike a good balance, being intuitive yet powerful; support is provided for path predicates, structural and value-based joins, universal quantification, negation, and a variety of complex conjunctive and disjunctive predicate forms. In addition to its object query features, PESTO is extensible (to support new data types) and portable across object database systems.

Our future plans include rewriting PESTO in Java and adding support for additional features, e.g., end-user customization, updates, methods, tabular collection views, and alternative paradigms for displaying a nested collection hierarchy. Important long-term plans include a usability study, comparing PESTO's ease of use against that of textual object-oriented query languages, and formal work to characterize PESTO's query power.

7 Acknowledgements

Rakesh Agrawal was a key participant in early design discussions that gave birth to PESTO. John Thomas implemented the PASTA protocol for Garlic, and Mary Tork Roth found many bugs in early versions of PESTO. Bill Cody and Denis Lee made the PESTO/QBIC connection happen. Jerry Kiernan provided access, at the University of Wisconsin, to his OO-SQL prototype. Finally, Roger King and Eben Haber kindly provided us with pointers to some of their favorite database user interface papers.

References

[Agra90] R. Agrawal, N. Gehani, and J. Srinivasan, "Ode-View: The Graphical Interface to Ode," *Proc. ACM SIGMOD Conf.*, May 1990.

[Bati91] C. Batini *et al*, *Visual Query Systems*, Tech. Rep. 04.91, U. di Roma, March 1991.

[Bryc86] D. Bryce and R. Hull, "SNAP: A Graphics-based Schema Manager," *Proc. IEEE Data Eng. Conf.*, 1986.

[Care95] M. Carey *et al*, "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach," *Proc. 1995 IEEE RIDE Workshop*, March 1995.

[Care96] M. Carey *et al*, *PESTO: An Integrated Query/Browser for Object Databases*, Res. Rep. No. RJ10016, IBM Almaden Research Center, March 1996.

[Cody95] W. Cody *et al*, "Querying Multimedia Data from Multiple Repositories by Content: The Garlic Project," *Proc.IFIP 2.6 Working Conference on Visual Database Systems - 3*, March 1995.

[Catt88] T. Rogers and R. Cattell, "Entity-Relationship Database User Interfaces," in *Readings in Database Systems*, M. Stonebraker (ed.), Morgan Kaufman, 1988.

[Catt94] R. Cattell (ed.), *The Object Database Standard: ODMG-93 (Release 1.1)*, Morgan Kaufman, 1994.

[Fogg84] D. Fogg, "Lessons from a "Living In a Database" Graphical Query Interface," *Proc. ACM SIGMOD Conf.*, June 1984.

[Heil85] S. Heiler and A. Rosenthal, "G-Whiz, A Visual Interface for the Functional Model with Recursion," *Proc. 11th VLDB Conf.*, Aug. 1985.

[Kier95] J. Kiernan and M. Carey, "Extending SQL-92 for OODB Access: Design and Implementation Experience," *Proc. ACM OOPSLA Conf.*, Oct. 1995.

[Kunt89] M. Kuntz and R. Melchert, "Pasta-3's Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power," *Proc. 15th VLDB Conf.*, Aug. 1989.

[Motr88] A. Motro, A. D'Atri, and L. Tarantino, "The Design of KIVIEW: An Object-Oriented Browser," *Proc. 2nd Int'l. Conf. on Expert Database Sys.*, April 1988.

[Nibl93] W. Niblack et al., "The QBIC Project: Querying Images by Content Using Color, Texture and Shape," *Proc. SPIE*, Feb. 1993.

[Oust94] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[Rowe85] L. Rowe, "Fill-in-the-Form Programming," *Proc. 11th VLDB Conf.*, Aug. 1985.

[Ston82] M. Stonebraker and J. Kalash, "TIMBER - A Sophisticated Relational Browser," *Proc. 8th VLDB Conf.*, Sept. 1982.

[Ston89] M. Stonebraker and E. Neuhold, *The Laguna Beach Report*, ICSI Tech. Rep. No. 1, Berkeley, CA, June 1989.

[Ston93] M. Stonebraker *et al*, "DBMS Research at a Crossroads: The Vienna Update," *Proc. 19th VLDB Conf.*, Aug. 1993.

[Yao84] S.B. Yao *et al*, "FORMANAGER: An Office Forms Management System," *ACM Trans. on Office Info. Sys.*, 2(3), July 1984.

[Zloo77] M. Zloof, "Query By Example," *IBM Sys. J.* 16, Dec. 1977.