Ozone: Integrating Structured and Semistructured Data*

Tirthankar Lahiri[1], Serge Abiteboul[2], Jennifer Widom[1]

[1] Stanford University, {tlahiri,widom}@db.stanford.edu
[2] INRIA Rocquencourt, serge.abiteboul@inria.fr

Abstract

Applications have an increasing need to manage *semistructured* data (such as data encoded in XML) along with conventional *structured* data. We extend the structured object database model ODMG and its query language OQL with the ability to handle semistructured data based on the OEM model and Lorel language, and we implement our extensions in a system called *Ozone*. In our approach, structured data may contain entry points to semistructured data, and vice-versa. The unified representation and querying of such "hybrid" data is the main contribution of our work. We retain strong typing and access to all properties of structured portions of the data while allowing flexible navigation of semistructured data without requiring full knowledge of structure. Ozone also enhances both ODMG/OQL and OEM/Lorel by virtue of their combination. For instance, Ozone allows OEM semantics to be applied to ODMG data, thus supporting semistructured-style navigation of structured data. Ozone also enables ODMG views of OEM data, allowing standard ODMG applications to access semistructured data without losing the benefits of structure. Ozone is implemented on top of the ODMG-compliant O_2 database system, and it fully supports our extensions to the ODMG model and OQL.

1 Introduction

Database management systems traditionally have used data models based on regular structures, such as the relational model [Cod70] or the object model [Cat94]. Meanwhile, the growth of the internet and the recent emergence of *XML* [LB97] have motivated research in the area of *semistructured* data models, e.g., [BDS95, FFLS97, PGMW95]. Semistructured data models are convenient for representing irregular, incomplete, or rapidly changing data. In this paper, we extend the standard well-structured model for object databases, the *ODMG* model [Cat94], and its query language, *OQL*, to integrate semistructured data with structured data. We present our implementation of the extended ODMG model and query language in a system called *Ozone*.

We will see that Ozone is well suited to handling *hybrid* data—data that is partially structured and partially semistructured. We expect hybrid data to become more common as more applications import data from the Web, and the integration of semistructured data within ODMG greatly simplifies the design of such applications. The exclusive use of a structured data model for hybrid data would miss the many advantages of a semistructured data model [Abi97, Bun97]—structured encodings of irregular or evolving semistructured data are generally complex and difficult to manage and evolve. On the other hand, exclusive use of a semistructured data model precludes strong typing and efficient implementation mechanisms for structured portions of the data. Our approach based on a hybrid data model provides the advantages of both worlds.

Our extension to the ODMG data model uses the *Object Exchange Model* (*OEM*) [PGMW95] to represent semistructured portions of the data, and it allows structured and semistructured data to be mixed together freely in the same physical database. Our OQL^S query language for Ozone is nearly identical to OQL [Cat94] but extends the semantics of OQL for querying hybrid data. An interesting feature of our approach is that it also enables structured data to be treated as semistructured data, if so desired, to allow navigation of structured data without

*This work was supported by the Air Force Rome Laboratories under DARPA Contract F30602-95-C-0119.

full structural knowledge. Conversely, it enables structured views on semistructured data, allowing standard ODMG applications access to semistructured data. We have implemented the full functionality of Ozone on top of the O_2 [BDK92] ODMG-compliant database management system (a product of ArdentSoftware Inc., `http://www.ardentsoftware.com`).

Related Work

Data models, query languages, and systems for semistructured data are areas of active research. Of particular interest and relevance, *eXtensible Markup Language* (*XML*) [LB97] is an emerging standard for Web data, and bears a close correspondence to semistructured data models introduced in research, e.g., [BDS95, FFLS97, PGMW95]. An example of a complete database management system for semistructured data is *Lore* [MAG+97], a repository for OEM data featuring the Lorel query language. Another system devoted to semistructured data is the *Strudel* Web site management system, which features the *StruQL* query language [FFLS97] and a data model similar to OEM. *UnQL* [BDHS96, BDS95] is a query language that allows queries on both the content and structure of a semistructured database and also uses a data model similar to Strudel and OEM. All of these data models, languages, and systems are dedicated to pure semistructured data. We know of no previous research that has explored the integration of structured and semistructured data as exhibited by Ozone. Note also that our query language OQL^S is supported by a complete implementation in the Ozone system.

There has been some work in extracting structural information and building structural summaries of semistructured databases. For example, [NAM98] shows how schema information can be extracted from OEM databases by typing semistructured data using Datalog. Structural properties of semistructured data can be described and enforced using *graph schemas* as shown in [BDFS97]. Structural summaries called *DataGuides* are used in the Lore system as described in [GW97]. These lines of research are dedicated to finding the structural properties of purely semistructured data and do not address the integration of structured and semistructured data as performed by Ozone.

The *OQL-doc* query language [ACC+97] is an example of an OQL extension with a semistructured flavor: it extends OQL to navigate document data without precise knowledge of its structure. However, OQL-doc still requires some form of structural specification (such as an XML or SGML *Document Type Definition* (*DTD*) [LB97]), so OQL-doc does not support the querying of arbitrary semistructured data.

2 Background and Motivating Example

2.1 The structured ODMG model and OQL

The ODMG data model is the accepted standard for object databases [Cat94]. ODMG has all the necessary features of object-orientation: classes with attributes and methods, subtyping, and inheritance. The basic primitives of the model are *objects* (values with unique identifiers) and *literals* (values without identifiers). All values in an ODMG database must have a valid *type* defined in the schema, and all values of an object type or *class* are members of a collection known as the *extent* for that class. Literal types include *atomic* types (e.g., integer, real, string, nil, etc.), *structured* types with labeled components (e.g. tuple(a:integer, b:real)), and *collection* types (set, bag, list, and array).

A *class* type encapsulates some ODMG type, and may define *methods* that specify the legal set of operations on objects belonging to the class. A class may also define *relationships* with other classes. Classes most commonly encapsulate structured types, and the different fields in the encapsulated structure (along with methods without arguments) denote the *attributes* of the class. ODMG defines the class Object to be the root of the class hierarchy. An attribute of a class in the ODMG model may have a literal type and therefore not have identity. *Named* objects and literals form entry points into an ODMG database.

The *Object Query Language*, or *OQL*, is a declarative query language for ODMG data. It is an *expression-oriented* query language: an OQL query is composed of one or more expressions or subqueries whose types can be inferred statically. Complex queries can be formed by composing expressions as long as the compositions respect the type system of ODMG. Details of the ODMG model and the OQL query language can be found in [Cat94], but are not essential for understanding this paper.

2.2 The semistructured OEM model and Lorel

The *Object Exchange Model*, or *OEM*, is a self-describing semistructured data model, useful for representing irregular or dynamically evolving data [PGMW95]. OEM objects may either be *atomic*, containing atomic literal values (of type integer, real, string, binary, etc.), or *complex*, containing a set of labeled OEM subobjects. A complex OEM object may have any number of children (subobjects), including multiple children with the same label. Note that all OEM subobjects have identity, unlike ODMG class attributes. An OEM database may be viewed as a labeled directed graph, with complex OEM objects as internal nodes and atomic OEM objects as leaf nodes. Named OEM objects form entry points into an OEM database.

Lorel is a declarative query language for OEM data and is based on OQL. Some important features of Lorel are listed below. Details of OEM and Lorel can be found in [AQM+97], but again are not crucial to understanding this paper.

- *Path expressions:* Lorel queries navigate OEM databases using *path expressions*, which are sequences of labels that may also contain wildcards and regular expression operators. For instance, the query "Select D From A(.b|.c%)*.d D" selects all objects reachable from entry-point A by following zero or more edges each having either label b or a label beginning with the character c, followed by a single edge labeled d.

- *Automatic coercion:* Lorel attempts to coerce operands to compatible types whenever it performs a comparison or other operation on them. For instance, if X is an atomic OEM object with the string value "4", then for the evaluation of $X < 10$, Lorel coerces X to the integer value 4. If no such coercion is possible (for instance, if X were an image or a complex object) the predicate returns false. Lorel also coerces between sets and singleton values whenever appropriate.

- *No type errors:* To allow flexible navigation of semistructured data, Lorel never raises type errors. For instance, an attempted navigation from an OEM object using a nonexistent label simply produces an empty result, and a comparison between non-comparable values evaluates to false. Thus, any Lorel query can be executed on an OEM database with unknown or partially known structure, without the risk of run-time errors.

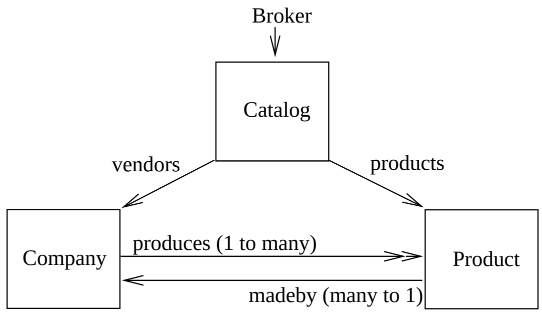

Figure 1: Structured ODMG classes in the retail-broker database

2.3 Example of hybrid data and queries

Our motivating example, used throughout the paper, considers a database behind a simplified on-line broker that sells products on behalf of different companies. There are three ODMG classes in this database: Catalog, Company, and Product. Class Catalog has one object, which represents the on-line catalog maintained by the broker. The object has two attributes: a vendors attribute of type set(Company), denoting the companies whose products are sold in the catalog, and a products attribute of type set(Product), denoting the products sold in the catalog. The Company class defines a one-to-many produces relationship with the class Product of type list(Product). This relationship specifies the list of products manufactured by the company, ordered by product number. Likewise, the Product class defines the inverse many-to-one madeby relationship with the class Company, denoting the product's manufacturer. The Company class contains other attributes such as name and address, and an inventory() method that takes a product name argument and returns the number of stocked units of the product of that name. The Product class contains other attributes such as name and prodnum (product number). The named object Broker of type Catalog provides an entry point to this database. Figure 1 depicts this schema without atomic attributes.

In addition to this structured data, let us suppose that we have product-specific XML information available for some products, e.g., drawn from Web sites of companies and analyst firms. This data might include manufacturer specifications (power ratings, weight, etc.), compatibility information if it applies (for instance, the strobes compatible with a particular camera), a listing of competing companies and products, etc. To integrate this XML data within our database, we enhance the Product class with a prodinfo attribute for this product-specific data. Since this data is likely to vary widely in format, we cannot easily use a fixed ODMG type for its representation, and it is much more convenient to use the semistructured OEM data model. Therefore, we let the prodinfo attribute be a "crossover point" (described below) from ODMG to OEM data.

There is also a need for referencing structured data from semistructured data. If a competing product (or company) or a compatible product appears in the broker's catalog, then it should be represented by a direct reference to the ODMG object representing that product or company. If the competing product or company is not part of the catalog, only then is a complex OEM object created to encode the XML data for that product or company.

An example OEM database graph for the prodinfo attribute of a product is shown in Figure 2. Note that in Figure 2, the competing product named "System12" is not part of the catalog database and therefore is represented by a (complex) OEM object; the other competing product and company are part of the catalog and are represented by references to Product and Company objects.

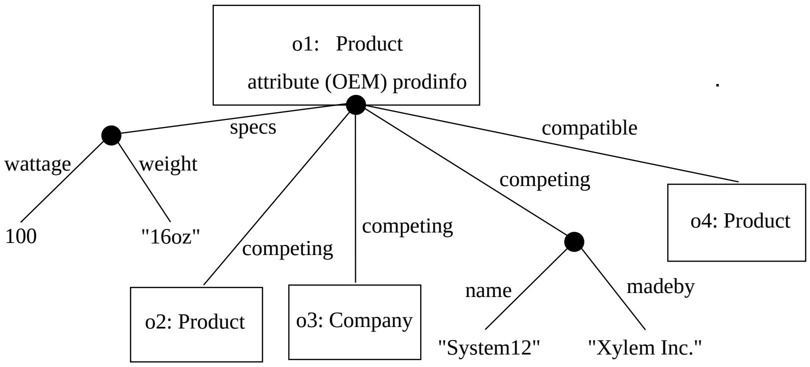

Figure 2: Example OEM graph for the prodinfo attribute of a Product object

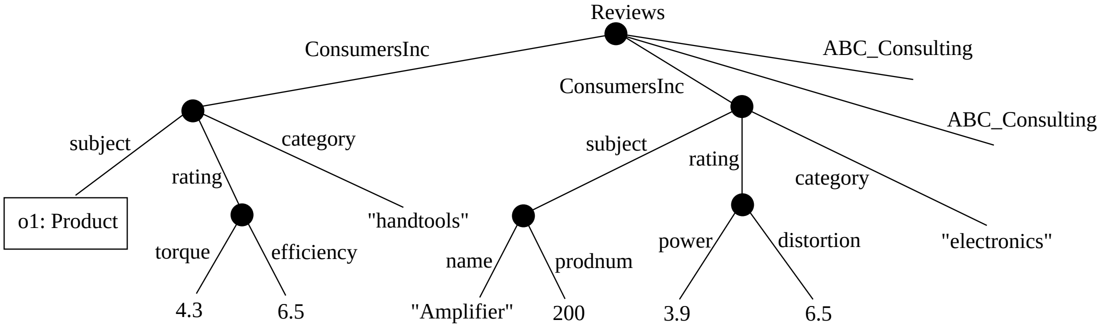

Figure 3: Semistructured Reviews data for the broker catalog

To continue with the example, let us also suppose that we have some review data available in XML for products and companies. The information is available from Web pages of different review agencies and varies in structure. We enhance our example database with a second entry point: the named object Reviews integrates all the XML review data from different agencies. Once again, the diverse and dynamic nature of this data means that it is better represented by the OEM data model than by any fixed ODMG type. Thus, Reviews is a complex OEM object integrating available reviews of companies and products. Here too we may reference structured data from semistructured data, since reviewed companies and products that are part of the catalog should be denoted by references to the ODMG objects representing them.

Figure 3 is a simplified example of this semistructured Reviews data. We assume that the reviews by a given agency reside under distinct subobjects of Reviews, and the names of the review agencies (ConsumersInc, ABC_Consulting, etc.) form the labels for these subobjects. For subsequent examples, we restrict ourselves to reviews by ConsumersInc. Reviews by this agency have a subject subobject denoting the subject of the review (either a product or a company), which may be a reference to the ODMG object representing the company or product, or may be a complex OEM object. Both cases are depicted in Figure 3.

Our overall example scenario consists of *hybrid* data. Some of the data is structured, such as the Product class without the prodinfo attribute, while some of the data is semistructured, such as the data reachable via a prodinfo attribute or via the Reviews entry point.

3 The Extended ODMG Data Model

Our basic extension to the ODMG data model to accommodate semistructured data is therefore relatively straightforward. We extend the ODMG model with a new built-in class type OEM. Using this OEM type, we can construct ODMG types that include semistructured data. For instance, we can define a partially semistructured Product class (as described above) with a prodinfo attribute of type OEM. There is no restriction on the use of the type OEM—it can be used freely in any ODMG type constructor, e.g., tuple(x:OEM, y:integer) and list(OEM) are both valid types.

Objects in the class OEM are of two categories: OEMcomplex and OEMatomic, representing complex and atomic OEM objects respectively.[3] An OEMcomplex object encapsulates a collection of (*label*,*value*) pairs, where *label* is a string and *value* is an OEM object. The original OEM data model specification included only unordered collections of subobjects [AQM+97, PGMW95], but XML, for example, is inherently ordered. Thus we allow complex OEM objects with either unordered or ordered subobjects in our data model, and refer to them as OEMcomplexset and OEMcomplexlist respectively.

To allow semistructured references to structured data, the value of an OEMatomic object may have any valid ODMG type (including OEM). Thus, apart from the ODMG atomic types integer, real, string, etc., the value of an OEMatomic object may for example be of type Product, tuple(a:integer, b:OEM), etc. When the content of an OEMatomic object is of type T, we will say that its type is OEM(T). Since OEM objects are actually untyped, OEM(T) denotes a "dynamic type" that does not impose any typing constraint. For example, an object of type OEM(integer) may be compared with an object of type OEM(string) or OEM(set(Product)) without raising a type error; further discussion of such operations is provided in Section 5.3. Intuitively, atomic OEM objects can be thought of as *untyped containers* for typed values. Note that an OEMatomic object of type OEM(OEM) can be used to store a reference to another OEM object (possibly external to the database). Also note that OEM(nil) is a valid type for an OEMatomic object, and we assume that there is a single named object OEMNil of this type in the OEM class.

4 Benefits of a Hybrid Approach

We now reinforce the benefits of a hybrid approach. An important advantage of our approach over a purely structured approach is that we can formulate queries on semistructured portions of the data without requiring full structural knowledge. With a purely structured approach, representation of XML data, for example, would require a different set of ODMG classes for each distinct XML DTD, possibly leading to complex schemas that the user would be required to have full knowledge of in order to formulate valid queries. Furthermore, modifications to the XML data might require expensive schema evolution operations. In contrast, the OEM model does not rely on a known schema, and the semantics of Lorel permits formulating queries without full knowledge of structure.

At the same time, an important benefit of our approach over a purely semistructured approach such as *Lore* [MAG+97] is that we are capable of exploiting structure when it is available. In particular, we can more easily take advantage of known query optimization techniques for structured data, and we can take advantage of strong typing when portions of the data are typed.

[3] These categories do not represent subclasses since, as we will see, OEM objects are untyped.

Finally, we can optionally apply the semantics of one data model to the other, so that the benefits of both models are available to us whether the data is structured or semistructured. For instance, treating ODMG data as OEM allows queries to be written without complete knowledge of the schema, while still retaining access to all ODMG properties (such as methods, indexes, etc.). On the other hand, we will show in Section 5.2 how our approach enables typed ODMG views of untyped OEM data, so that standard ODMG applications can access semistructured data using standard API's and structural optimizations.

5 The OQL^S Query Language

The query language for the Ozone system is *OQL^S*. OQL^S is not a new query language—except for some built-in functions and syntactic conveniences derived from Lorel, it is syntactically identical to OQL. The semantics of OQL^S on structured data is identical to OQL on standard ODMG data. OQL^S extends OQL with additional semantics that allow it to access semistructured data. The semistructured capabilities of OQL^S are mostly derived from Lorel, which is based on OQL but was designed specifically for querying pure semistructured data. Like Lorel, OQL^S allows querying of semistructured data without the possibility of run-time errors. OQL^S also provides new features necessary for the navigation of hybrid data: since OQL^S expressions can contain both structured and semistructured operands, OQL^S defines new semantics that allow such queries to be interpreted appropriately.

Space limitations preclude a complete specification for OQL^S in this paper. Since its syntax combines OQL and Lorel, interested readers are referred to [AQM$^+$97, Cat94]. In the remainder of this section we describe some of the more interesting aspects of the semantics of OQL^S, using simple self-explanatory queries to illustrate the points. In Section 5.1 we describe path expression "crossovers" from structured to semistructured data and vice-versa. In Section 5.2 we describe how our approach enables structured ODMG views over semistructured data. Section 5.3 discusses the semantics of OQL^S constructs such as arithmetic and logical expressions involving hybrid operands.

5.1 Path expression crossovers

When we evaluate a *path expression* in an OQL^S query (recall Section 2.2), a corresponding database path may involve all structured data, all semistructured data, or there may be *crossover points* that navigate from structured to semistructured data or vice-versa. Crossing boundaries from structured to semistructured data is fairly straightforward, since we can always identify the crossover points statically. For example, the following query selects the names of all competing products and companies for all products in the broker catalog from Section 2.3:

```
Select N
From   Broker.products P, P.prodinfo.competing C, C.name N
```

P is statically known to be of type Product, but prodinfo is an OEM attribute, and C is therefore of type OEM; prodinfo is thus a crossover point from structured to semistructured data.

Semistructured to structured crossover is more complicated, and we focus on this case. It is not possible to identify such crossover points statically without detailed knowledge of the structure of a hybrid database. To define the semantics of queries with transparent crossover from semistructured to structured data, we introduce (below) the logical concept of *OEM proxy* for encapsulating structured data in OEM objects. We also discuss in Section 5.1.1 an explicit form of crossover for users who do have detailed knowledge of structure.

In the purely semistructured OEM data model, atomic OEM objects are leaf nodes in the database graph. The result of attempting to navigate an edge from an atomic OEM object is defined in Lorel to be the empty set, i.e., the result of evaluating $X.label$ is empty if X is not a complex OEM object. However, in our extended ODMG model, OEMatomic objects are containers for values with any ODMG type (recall Section 3), so in addition to containing atomic values, they may provide semistructured crossover to structured data.

Thus, OQL^S extends Lorel path expressions with the ability to navigate structured data encapsulated by OEMatomic objects, in addition to navigating semistructured data represented by OEMcomplex objects. In our running example, some of the objects in the Reviews graph labeled subject are OEMatomic objects of type OEM(Product) and OEM(Company). Thus, the following query has a semistructured to structured crossover since some of the bindings for C are OEM(Company) and OEM(Product) objects:

```
Select A
From   Reviews.ConsumersInc R, R.subject C, C.address A
```

For C bindings of type OEM(Company), the evaluation of C.address generates the *OEM proxy* (defined below) of the address attribute in the Company class. For C bindings of type OEM(Product), the evaluation of C.address yields the empty set.

To allow flexible navigation of structured data from semistructured data, at the same time retaining access to all properties of the structured data, we define the logical notion of *OEM proxy* objects, as follows:

Definition 5.1 (OEM Proxy) An OEM proxy object is a temporary OEM object created (perhaps only logically) to encapsulate a value of any ODMG type. It is an OEMatomic object that serves as a *proxy* or a surrogate for the value it encapsulates. □

Semistructured to structured crossover is accomplished by (logically) creating OEM proxy objects, perhaps recursively, to contain the result of navigating past an OEMatomic object. It is important to note that this concept of OEM proxy is a logical rather than a physical concept. It specifies how a query over hybrid data is to be interpreted, but does not specify anything about the actual implementation of the query processor. All we require is that the result of a query should be the same as the result that would be produced if proxy objects were actually created for every navigation past every OEMatomic object. For C bindings of type OEM(Company) in our example query, the corresponding A bindings are OEM proxy objects of type OEM(tuple(street:string, city:string, zip:integer)) encapsulating the address attribute of the corresponding Company objects.

The general algorithm for evaluating $X.l$ when X is an OEMatomic object of type OEM(T) encapsulating the value Y follows. Consider the different cases for T:

1. *T is an atomic ODMG type* (i.e., one of integer, real, char, string, boolean, binary, or nil): For any label l, the result of evaluating $X.l$ is the empty set.

2. *T is a tuple type:* For all non-collection fields in the tuple whose labels match l (note that a label with wildcards can match more than one field) a proxy object is created encapsulating the value of the field. If the label of a collection-valued field matches l, proxies are created encapsulating each element of the collection. The result of evaluating $X.l$ is the set of all such proxies.

3. *T is a collection:* If T is a set or a bag type, $X.l$ returns a set. This set is empty unless the label l is the specific label item. For this built-in system label, the value of $X.l$ is a set of OEM proxies encapsulating the elements in the collection Y. If T is a list or an array type, $X.l$ is evaluated similarly, except that the ordering of the elements of Y is preserved by returning a list of proxies instead of a set. Note that navigation past such objects requires some knowledge of the type T, since the user needs to use label item (or a wildcard) to navigate below the encapsulated collection.

4. *T is a class*: Here, Y is an object encapsulating some value—let Z be an OEM proxy for that value. The result of evaluating $X.l$ is a set including the OEM proxies obtained by evaluating $Z.l$ (by recursively applying these rules), the OEM proxies encapsulating the values of any relationships with names matching l, and the OEM proxies encapsulating the results of invoking any methods with names matching l. If T is OEM, X is a reference to an OEM object Y, and the result of $X.l$ is the same as the result of evaluating $Y.l$, i.e., automatic dereferencing is performed.

To illustrate these rules, consider the following query, which selects the names of all products manufactured by all companies reviewed by ConsumersInc:

```
Select N
From   Reviews.ConsumersInc R, R.subject C,
       C.produces P, P.name N
```

Here, for those C bindings that are of type OEM(Company), the corresponding P bindings are proxies of type OEM(Product), encapsulating the elements of the relationship produces in the Company class. Finally, the N bindings encapsulate the name attributes of the Product objects encapsulated by the P bindings, and the type of these N bindings is OEM(string). Proxies thus allow "transparent navigation" to structured data from semistructured data.

Queries should be able to access all properties of structured data referenced by semistructured data, and OQL^S therefore allows queries to invoke methods with arguments on structured objects encapsulated by OEMatomic objects. The expression $X.m(arg_1, arg_2, \ldots, arg_n)$ applies the method $m()$ with the specified list of arguments to the object encapsulated by X. If X is of type OEM(T), the result of this expression is a set containing the OEMatomic object encapsulating the return value of the method, provided T is a class that has a method of this name and with formal parameters whose types match the types of the actual parameters $arg_1, arg_2, \ldots arg_n$. If T is not a class, or if it is a class without a matching method, or if X is of type OEMcomplex, this expression returns the empty set. As an example, the following query selects the inventories of "camera1" for all companies in the catalog and reviewed by ConsumersInc:

```
Select C.inventory("camera1")
From   Reviews.ConsumersInc R, R.subject C,
```

For those C bindings that are not of type OEM(Company), the evaluation of the expression C.inventory("camera1") yields the empty set. For C bindings of type OEM(Company), the evaluation of C.inventory("camera1") returns a singleton set containing an OEM(integer) object encapsulating the result value.

OQL allows casts on objects, and as a ramification of our approach, OQL^S allows any object to be cast to the type OEM by creating the appropriate proxy for the object, allowing semistructured-style querying of structured data. For instance, an object C of type Company can be cast to an OEM proxy object of type OEM(Company) by the expression (OEM) C. Once this casting is performed, the proxy can be queried without full knowledge of its structure. This casting approach is useful when users have approximate knowledge of the structure of some (possibly very complex) structured ODMG data but prefer not to study its schema in detail. Queries with casts to OEM are also useful when the structure of the database changes frequently and users want the same query to run against the changing database without errors.

5.1.1 Semistructured to structured crossover by explicit coercion

OQL^S provides another mechanism for accessing structured data from semistructured data: a modified form of casting that extracts the structured value encapsulated by an OEMatomic object. Although OEM proxies allow all properties of structured data contained in OEMatomic objects to be accessed without casts, casts enable static type checking. Furthermore, casting a semistructured operand to its true structured type may also provide performance advantages, since it may allow the query processor to exploit the known structure of the data.

In OQL, a standard cast on a structured operand "$(T)X$" may produce a runtime error if the X binding is not a subtype of T. However, this approach is not suitable for casts on OEM objects, since it contradicts our philosophy of not mandating structural knowledge of semistructured data. Therefore, OQL^S provides a separate mechanism for performing casts on OEM objects without type error, through the built-in Coerce function defined as follows:

Definition 5.2 (The Coerce function) Let O be an OEM object. The value of Coerce(C, O) is the singleton set set((C) X) if O is an OEMatomic object encapsulating an object X in class C (or a subclass of C). Otherwise the value of Coerce(C, O) is the empty set. □

As an example, the following query selects the products of all Company subjects of reviews by ConsumersInc. Since the type of the C bindings is known to be Company, the type of the result returned by the query can be determined statically to be set(list(Product)):

```
Select P
From   Reviews.ConsumersInc R, R.subject S,
       Coerce(Company, S) C, C.produces P
```

In the future we may also consider a more powerful "case" construct that successively tries to coerce an OEM object into a structured object from a list of classes, returning the first non-empty result obtained.

5.2 Structured access to semistructured data

A powerful feature of OQL^S is its support for *structured views* of semistructured data. Intuitively, structured data can be synthesized from semistructured operands when the semistructured data is known to exhibit some regularity, e.g., based on an XML DTD or from an analysis of the data. Such structured views may provide faster access paths for queries (e.g., via standard indexes), and the structured results can be exported to standard ODMG applications that do not understand the OEM model, and that may use API's such as Java or C++ bindings to access the database.

The synthesis of structured data from semistructured data is accomplished in OQL^S (once again, without the possibility of type error) using the built-in Construct function defined as follows:

Definition 5.3 (The Construct function) Let O be an OEM object. Construct(T, O) returns a value of type set(T) that is either a singleton set containing a value of type T constructed from the OEM object O, or the empty set if no such construction is possible. □

The Construct function may be viewed as a rich coercion function from OEM to a given type. If O is an OEMatomic object, then Construct behaves similarly to Coerce in Definition 5.2 above. If O is an OEMcomplex object, then Construct creates a structured ODMG tuple. Construct(T, O) is defined recursively as follows:

1. If O is an OEMatomic object of type OEM(T'), and if T' is identical to or is coercible to a subtype of T, then a singleton set containing the value encapsulated by O is returned.

2. If T is a class encapsulating a type T', and if Construct(T', O) $= \{v\}$, then Construct(T, O) is a singleton set containing a new T object encapsulating the value v.

3. If T is a tuple type, then each field labeled l must be constructed:

   (a) If l is a collection of values of type T', then for each l-labeled subobject O' of O, we evaluate Construct(T', O'). The result v_l is a collection of the non-empty results of this evaluation. If O is an OEMcomplexlist object and l is an ordered collection, the ordering of the subobjects of O is preserved in the construction. Otherwise, an arbitrary order is used for the resulting collection v_l.

   (b) l has a non-collection type T': The construction is successful if there is exactly one l-child O' of O and if Construct(T', O') $= \{v_l\}$.

   Finally, Construct(T, O) is a singleton set containing the tuple with value v_l for each l-field in the tuple.

4. If T is a collection of values of type T', then for each subobject O' of O with the reserved label item, we evaluate Construct(T', O'). The result is a collection of the non-empty results of these evaluations. Once again, if T is an ordered collection, the ordering of the subobjects of O is preserved in the construction; otherwise an arbitrary order is produced.

5. In all other cases, Construct returns an empty set.

As an example, let us suppose (simplistically) that we know that the manufacturer specifications for electrical products in our broker catalog always includes an integer wattage value and a real weight value. We define a class Espec encapsulating the type tuple(wattage:integer, weight:real). The query below selects a structured set of specifications:

```
Select E
From   Broker.products P, P.prodinfo.specs S, Construct(Espec, S) E
```

The type of the S bindings is OEM, and the type of the E bindings is Espec. The result of the query is therefore of type set(Espec). Thus, the result is a set of structured objects that may be materialized for indexing purposes, and may easily be exported to a Java or C++ application. While this example is very simple (and in fact a similar effect could have been achieved by using a tuple constructor in the Select clause), it does illustrate the general principle, which is to create structured views over portions of the data that are semistructured but have known components.

5.3 Semantics of mixed expressions

In OQL, expressions (queries) can be composed to form more complex expressions as long as the expressions have types that can be composed legally. OQL provides numerous operators for compositions, such as arithmetic operators (e.g., $+$), comparison operators (e.g., $<$), boolean operators (e.g., AND), set operators (e.g., UNION), indexing operators (e.g., $\langle$*list_name*$\rangle$[$\langle$*position*$\rangle$]), etc. (See [Cat94] for an exhaustive specification of all OQL compositions.) OQL^S extends the composition rules of OQL to allow semistructured and structured expressions to be mixed freely in such compositions. We refer to expressions that include a semistructured subexpression as *mixed expressions*. Space limitations preclude an exhaustive treatment of all possible OQL^S expressions in this paper, but several important aspects of the interpretation of mixed expressions are highlighted in the remainder of this section.

Run-time coercion is used in the evaluation of mixed expressions

Faithful to the Lorel philosophy, the OQL^S query processor evaluates mixed expressions by attempting to coerce their subexpressions to types that can be composed legally. As an example, we consider the interpretation of compositions involving the comparison operator "$<$".

In OQL, the expression $(X < Y)$ is legal provided X and Y both have types integer or real (interpreted as arithmetic comparison), string (interpreted as string comparison), boolean (interpreted as boolean comparison), or set(T) or bag(T) (interpreted as set inclusion). A type error is raised in OQL for all other cases. OQL^S additionally allows X, Y, or both to be of type OEM, and the type of the mixed boolean expression $(X < Y)$ in that case is also OEM: OEM(true), OEM(false), or OEMNil (recall Section 3 for definitions of these types). For instance, the value of (OEM(4) $<$ 5) and (OEM(4) $<$ "5") are both OEM(true). The value of (OEM(4) $<$ set(1, 3)) is OEMNil since the two operands cannot be coerced into comparable types.

OEMNil is used to implement three-valued logic for mixed boolean expressions

Mixed boolean expressions are evaluated in OQL^S according to the rules of three-valued logic, similar to Lorel and just as NULL values are treated in SQL [MS93]. There are two important aspects to the use of OEMNil for implementing three-valued logic: First, if the Where clause of a query is a mixed-boolean expression, a value of OEMNil for the Where clause is interpreted as false. Second, if a query returns a collection of OEM objects, any OEMNil values are filtered out from the result; however, OEMNil values may appear in OEM components of structured query results.

The latter point is illustrated by the following two queries, which have identical From clauses but differ in their Select clauses.

```
Select P.prodnum + 100              Select tuple(prod:P, newpnum:P.prodnum+100)
From   Reviews.ConsumersInc R,      From   Reviews.ConsumersInc R,
       R.subject P                         R.subject P
```

In both queries, since OEM entry point Reviews is used, variable P is of type OEM. If for a particular P binding passed to the Select clause, the value of P.prodnum $+$ 100 is OEMNil (because P.prodnum cannot be coerced to an integer or a real, e..g, it has string value "123A"), then the value OEMNil is discarded from the query result. On the other hand, the query on the right has a (structured) Select clause of type tuple(prod:OEM, newpnum:OEM). For this query, a similar P binding would produce an OEMNil value for the newpnum component of the

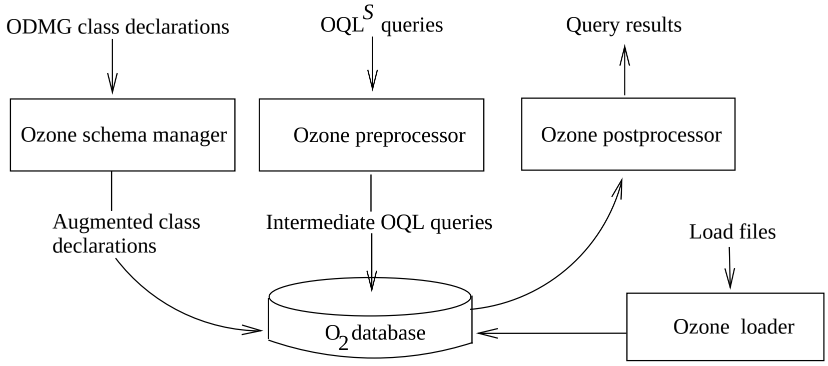

Figure 4: Architecture of the Ozone system

tuple. That OEMNil value is retained in the result, since it is part of a tuple in which the prod component has the non-nil value P.

OEM operands can be used in any expression without type error

Since an OEM object can be a container for any type, OQL^S allows OEM operands to be used in any expression without type error. An OEM expression may therefore be used in a query as an atomic value, a collection, a structure, or a class. For instance, in OQL, the index expression $X[Y]$ is a legitimate expression only when X is of type string, list(T), or array(T), and when Y is an integer. In OQL^S, $X[Y]$ is also a legitimate expression when X, Y, or both are of type OEM. Consider when X is of type OEM. If Y is an integer or an OEMatomic object encapsulating an integer (or a string coercible to an integer), and if X is of type OEM(string), then $X[Y]$ is of type OEM(char) and encapsulates the Y^{th} character in the string. If X is of type OEM(list(T)) or OEM(array(T)), then $X[Y]$ has type OEM(T) and encapsulates the Y^{th} element in the ordered collection encapsulated by X. Finally, if X is of type OEMcomplexlist, then $X[Y]$ returns the Y^{th} subobject of X. For all other types of X, the value of $X[Y]$ is OEMNil.

6 Implementation

The Ozone system is fully implemented on top of the ODMG-compliant O_2 object database system. An *Ozone database* is simply an O_2 database whose schema has been enhanced by a *schema manager* module that adds predefined classes for the storage and management of OEM objects. OQL^S queries are compiled by a *preprocessor* module into intermediate OQL queries on the Ozone database, and query results are presented to the user via a *postprocessor* module. Ozone also provides a *loader* module for bulk-loading semistructured data from files, and the semistructured data in the load files may include references to structured data. See Figure 4 for a depiction of the overall architecture. In this section, we first describe how OEM objects are represented in an Ozone database, and then describe the main aspects of the translation from OQL^S to OQL. Space limitations preclude a complete description of our implementation.

6.1 Representation of OEM objects in O_2

Ozone defines a class OEM to represent OEM objects in O_2. This class is the base class for OEM objects, and the different kinds of OEM objects introduced in Section 3 are represented by subclasses of OEM, as follows.

6.1.1 Complex OEM objects

The class OEMcomplex represents complex OEM objects. This class has two subclasses for representing ordered and unordered complex OEM objects: OEMcomplexset and OEMcomplexlist. Since complex OEM objects are collections of (*label*, *value*) pairs, the types encapsulated by these two classes are set(tuple(label:string, value:OEM)) and list(tuple(label:string, value:OEM)) respectively.

6.1.2 Atomic OEM objects encapsulating atomic values

Atomic OEM objects encapsulating atomic values also are represented by subclasses of OEM. For example, the class OEM_integer (encapsulating the type integer) represents OEM(integer) objects. The class OEM_Object (encapsulating the class Object) represents the type OEM(Object). (Recall that Object is a supertype of all classes in ODMG.) The remaining atomic classes are OEM_real, OEM_boolean, OEM_char, OEM_string, OEM_binary, and OEM_OEM (encapsulating the class OEM and representing the type OEM(OEM)). The class OEM itself encapsulates the type nil, and the extent for this class consists of the single named object OEMNil. The classes described here and in Section 6.1.1 are the *fixed classes* of Ozone—they are present in every Ozone schema.

6.1.3 Atomic OEM objects encapsulating ODMG objects

Recall from Section 3 that an atomic OEM object can encapsulate a value of any ODMG type. We distinguish between two cases for non-atomic types: classes (this section), and non-atomic literal types such as tuples or collections (Section 6.1.4).

When C is a class, atomic OEM objects of type OEM(C) could be represented as instances of the class OEM_Object (Section 6.1.2): since Object is the root of the ODMG class hierarchy, the class OEM_Object can store references to objects in any class. However, the use of a single class has performance limitations, since the exact types of encapsulated objects would have to be determined at run-time through potentially expensive schema lookups. Therefore, for performance reasons, Ozone defines a *proxy class* OEM_C for representing atomic OEM objects of type OEM(C) for each user-defined class C. Operations on an object belonging to a proxy class need not consult the schema and can exploit the structural properties of the encapsulated structured object (whose exact type is known from the proxy class). The Field() method, described later in Section 6.2.1, is example of an operation that can exploit structure through this approach. In our running example, the Ozone schema manager defines proxy classes OEM_Catalog (encapsulating the class Catalog), OEM_Product (encapsulating the class Product), and OEM_Company (encapsulating the class Company). These classes represent the types OEM(Catalog), OEM(Product), and OEM(Company).

6.1.4 Atomic OEM objects encapsulating non-atomic literals

Atomic OEM objects encapsulating non-atomic literal values (tuple, set, list, etc.) could be represented by equivalent complex OEM objects. For instance, an atomic OEM object encapsulating the value tuple(name:"foo", oid:4) could be represented in Ozone by an equivalent OEMcomplexset object with two children:("name", OEM_string("foo")) and ("oid", OEM_Integer(3)).

For performance reasons once again, Ozone defines additional OEM subclasses encapsulating the types of non-atomic class properties (attributes, relationships, and methods) since these are

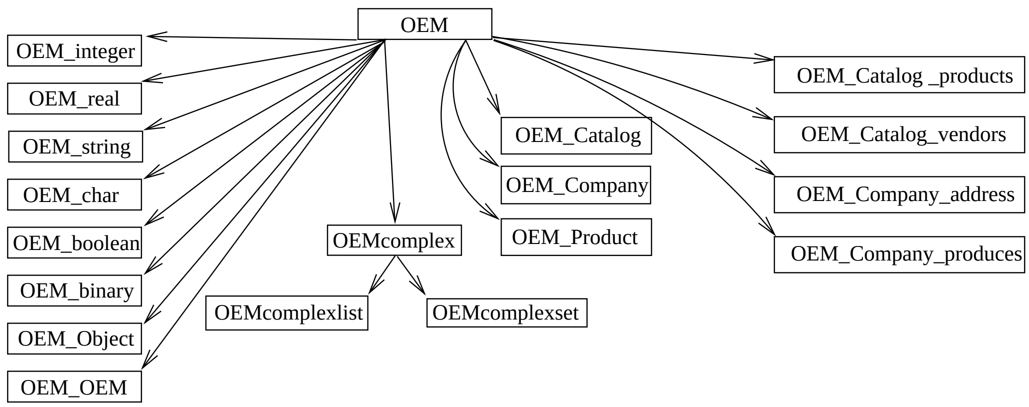

Figure 5: Classes added by Ozone to our example schema

the non-atomic literal types that are most commonly encountered in queries. For each such $\langle property \rangle$ of type P in each class C, Ozone creates a new *auxiliary class* $\mathsf{OEM\_}C\_\langle property\rangle$ to represent atomic OEM objects of type $\mathsf{OEM}(P)$. As with proxy classes (Section 6.1.3), a query on an auxiliary class object is faster than a query over an equivalent OEMcomplex object representing the same data. Of course, it is not possible to define auxiliary classes encapsulating all possible non-atomic types, since the space of temporary literal types that can be synthesized by queries and subqueries is infinite. For values of such types, Ozone must create equivalent OEM objects.

Referring again to our running example, the produces relationship of Company has the non-atomic literal type list(Product), and the Ozone schema manager therefore creates the auxiliary class OEM_Company_produces encapsulating the type list(Product). This auxiliary class represents the type OEM(list(Product)). The class Company also has the non-atomic address attribute, and the schema manager creates the auxiliary class OEM_Company_address. Similarly, the schema manager creates the classes OEM_Catalog_vendors and OEM_Catalog_products for the vendors attribute and the products attribute in the Catalog class.

The complete set of fixed, proxy, and auxiliary classes added by the Ozone schema manager to the schema of Section 2.3 is depicted in Figure 5. Note that this class hierarchy is invisible to the user; the type OEM is the only user-visible type added by Ozone.

6.2 Translation from OQL^S to OQL

OQL^S expressions involving OEM operands are implemented by methods defined in the class OEM. This class defines methods for navigation, user-specified coercion, the Construct function described in Section 5.2, and performing different kinds of unary and binary operations (such as arithmetic, boolean, comparison, and set operations). In the remainder of this section, we first describe the use of methods for implementing path expressions in OQL^S. Then we describe the implementation of user-specified coercion and construction. We conclude with a brief illustration of how methods are used to implement mixed expressions.

6.2.1 Implementation of OQL^S path expressions

Structured data is queried by OQL^S in exactly the same way as by OQL on a standard O_2 database, i.e., an OQL^S query over structured (pure ODMG) data does not need to be modified by the Ozone preprocessor. However, navigation of OEM objects is performed by methods such as the Field() method, which takes a label argument and produces the set of children with matching labels. Other navigational methods are discussed briefly at the end of this section. Let us illustrate how the Ozone preprocessor rewrites OQL^S queries on OEM objects using the Field() method. The following OQL^S query (on the left) that selects all compatible products for all products in the broker catalog is translated by the Ozone preprocessor to the OQL query shown on the right:[4]

```
Select C                              Select C
From   Broker.products P,             From   P In Broker.products,
       P.prodinfo.compatible C               C In P.prodinfo.Field("compatible")
```

For complex OEM objects (ordered or unordered) the implementation of the Field() method is straightforward: it iterates through the encapsulated collection of (*label*, *value*) pairs and retrieves all children whose labels match the label argument to the method.

For atomic OEM classes encapsulating non-atomic values, the definition of the Field() method is designed to be consistent with the rules for navigating past atomic OEM objects as defined in Section 5.1. For each proxy or auxiliary OEM class, Ozone automatically generates its Field() method (at class definition time) using the O_2 *metaschema*—an API allowing schema lookups and manipulations within an application. The Field() method matches the label argument with each attribute, relationship, and method (without arguments) in the class and returns OEM objects encapsulating any matching property or the return value of any matching method. Atomic properties are returned as instances of corresponding atomic OEM classes (for instance, an attribute of type integer would be returned in an object of type OEM_integer), while non-atomic properties are returned as instances of proxy or auxiliary classes (for instance, the produces relationship of a Company object would be returned in an object of type OEM_Company_produces).

As described in Section 5.1, OQL^S allows method invocations on OEM objects with the following semantics: if the OEM object encapsulates an object in a class with a matching method, the method is invoked and a singleton set containing the OEM proxy object encapsulating the method's return value is generated, otherwise the empty set is returned. Methods without arguments are handled in Ozone in the same way as class attributes as described above. Methods with one or more arguments are handled by the Invoke() method whose signature is:

set(OEM) Invoke(string methodName, list(string) argstext)

Note that any valid OQL^S expression can be used as an argument to a method on an OEM object. The argstext parameter lists the actual query texts for these expressions. In a proxy OEM class, if methodName matches any method in the encapsulated class, Invoke() uses the Ozone preprocessor to translate each OQL^S expression in argstext into its intermediate OQL form. If the argument types match the method's signature, Invoke() uses the O_2 API to invoke the method on the encapsulated ODMG object (the intermediate OQL expressions are used as arguments in this call). For example, an OQL^S query invoking the inventory() method on all subjects of reviews by ConsumersInc is shown below, together with its intermediate OQL form:

```
Select C.inventory("camera1")    Select C.Invoke("inventory",list(""camera1""))
From   Reviews.ConsumersInc R,   From   R In Reviews.Field("ConsumersInc"),
       R.subject C                      C In R.Field("subject")
```

[4]We translate all range variables to the "V In..." style, since O_2's version of OQL supports only this form.

For any C binding that is of type OEM(Company), the Invoke() method applies the inventory() method with the argument "camera1" to the encapsulated Company object. Since this argument is an atomic OQL expression, it does not need preprocessing. For such C bindings, Invoke() returns a set containing a single OEM_integer object storing the result of applying the method. For all other C bindings, Invoke() returns the empty set.

Path expressions with wildcards and regular expression operators

Recall that path expressions in OQL^S may contain wildcards and regular expression operators [AQM+97]. Wildcards are supported by the Field() method, whose label argument may contain wildcards. Regular expression operators are implemented through standard set operations and three additional navigational methods in the OEM class:

- Closure(): X.Closure("foo") returns the set of all objects reachable from the object X by following zero or more edges labeled "foo".
- UnionClosure(): X.UnionClosure(set("foo_1",..., "foo_n")) returns the set of all objects reachable from the object X by following zero or more edges whose labels match any one of "foo_1",..., "foo_n".
- NestedClosure(): X.NestedClosure(query) returns the set of all objects obtained by zero or more executions of query. Variable self is used in the query to reference the object on which the method is invoked.

As an example, the following translation uses the NestedClosure and Closure methods as well as a Union operation to implement a nested regular expression.[5]

```
Select D                  Select D
From   A(.B|(.C)*)* D     From   D in A.NestedClosure(
                                      "Select X
                                       From   X in (self.Field("B")
                                                    Union
                                                    self.Closure("C"))")
```

6.2.2 Implementation of user-specified coercion and construction

As described in Section 5.1, OQL^S allows an object X in any class C to be converted into an OEM proxy object of type OEM(C) through the user-specified cast expression (OEM) X. This coercion is implemented in Ozone simply by creating an appropriate object in the proxy class for C. For instance, if X is of type Company, (OEM) X is translated by the preprocessor to OEM_Company(X), which creates a proxy for a Company object.

OQL^S also allows an OEM object X to be coerced explicitly to any class C using the expression Coerce(C, X). This Coerce function of Definition 5.2 is implemented as a method Coerce in the class OEM. One difficulty is that the result of the Coerce function is of type set(C), where C can be any ODMG class in the schema, i.e., we have introduced polymorphism that is not supported by ODMG. In our implementation, method Coerce therefore has the fixed return type set(Object), which is suitable for returning objects of any type. The preprocessor then inserts an OQL cast to obtain objects of the proper type.

[5] In this particular example the regular expression could be simplified to `A(.B|.C)*`, in which case the translation uses only the UnionClosure method, but we translate the unsimplified expression for illustrative purposes.

Finally, as described in Section 5.2, OQL^S allows the construction of a structured ODMG value of type T from an OEM object O using the expression Construct(T, O). The present Ozone prototype requires T to be a class type and implements the Construct function as a method Construct in the OEM class. For reasons analogous to the Coerce method, the return type of the method is set(Object). Method Construct uses the O_2 metaschema to create an object in the class C, then attempts to construct the different attributes of the object using the rules described in Section 5.2. If the construction is not successful, the empty set is returned. Once again, the preprocessor must insert an OQL cast to obtain objects in the specified class.

6.2.3 Implementation of mixed expressions

Mixed expressions involving OEM operands (Section 5.3) are implemented through methods. We will illustrate the comparison operator "$<$" as an example. In OQL^S, an OEM object can be compared with values of the following atomic ODMG types: integer, real, boolean, and string. For these types the OEM class defines the comparison methods Less_integer(integer value), Less_real(real value), etc. An OEM object also can be compared with another OEM object (the comparison is interpreted at run-time based on the exact types of the values encapsulated by the two objects), and for this purpose the Less_OEM(OEM value) method is provided. The "$<$" operator also denotes set containment. The present Ozone prototype defines set comparison methods only for those unordered collections types that have corresponding auxiliary classes. In our example schema, we thus define the method Less_set_Product(set(Product) value) since OEM_Catalog_products is of type set(Product), and Less_set_Company(set(Company) value) since OEM_Catalog_vendors is of type set(Company). The return type of all of these comparison methods is OEM, and the return value is always one of OEM_boolean(true), OEM_boolean(false), or OEMNil.

As examples of the use of comparison methods, let X be an OEM object. Ozone translates the expression $(X < 5)$ to X.Less_integer(5) and the expression ($X <$ "abc") to X.Less_string("abc"). If Y is of type set(Company), the expression $(X < Y)$ is translated to X.Less_set_Company(Y). Thus, the decision of which comparison method should be invoked is made statically.

7 Conclusions

We have extended ODMG with the ability to integrate semistructured data with structured data in a single database, and we have extended the semantics of OQL to allow queries over such hybrid data. As far as we know, our work is the first to provide true integration of semistructured and structured data with a unified query language. We feel that this direction of work is particularly important as more and more structured data sources incorporate semistructured XML information, and vice-versa. We have built *Ozone*, a system that implements our ODMG and OQL extensions on top of the O_2 object-oriented database system. Our future work will proceed in the following directions:

- *Ozone performance:* We plan to investigate optimizations that would allow the navigation of OEM proxies for structured data to be as fast as standard OQL. We also plan to study general optimizations for navigating semistructured data [MW99] in the context of Ozone, and explore the use of mechanisms provided by O_2 (such as clustering and indexing) to improve performance. Finally, we would like to detect regularity in semistructured data, and determine whether we can exploit such regularity by using ODMG classes to represent such data [NAM98].

- *Object-Relational Ozone:* We plan to define a similar semistructured extension to the object-relational data model [SM96], and define semantics for the SQL-3 query language in order to query hybrid (object-relational plus semistructured) data.
- *Applications:* We intend to study a suite of applications that can take advantage of our hybrid approach, in order to identify any missing functionality and performance bottlenecks. We also plan to investigate general design issues for hybrid-data applications.

Acknowledgments

We thank the members of the Lore group at Stanford University for their feedback on the design of Ozone. We also acknowledge Ardent Software for providing us with a copy of the O_2 database system. Dallan Quass was influential in evolving Lorel to an OQL-like form, and Sophie Cluet and Jerome Simeon provided help with numerous questions on O_2 and OQL. Tirthankar Lahiri thanks Oracle Corporation for sponsoring his doctoral studies at Stanford University.[6]

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[ACC+97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1):5–19, 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 335–350, Delphi, Greece, January 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Francisco, California, 1992.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of the 1995 International Workshop on Database Programming Languages (DBPL)*, 1995.

[6] Oracle Corporation is not responsible in any way for the content of this paper.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997. Tutorial.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard: ODMG–93*. Morgan Kaufmann, San Francisco, California, 1994.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a Web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[LB97] R. Light and T. Bray. *Presenting XML*. Sams, Indianapolis, Indiana, September 1997.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MS93] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco, California, 1993.

[MW99] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the Twenty-Fifth International Conference on Very Large Databases*, Edinburgh, Scotland, September 1999. To appear.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, May 1998.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[SM96] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, San Francisco, California, April 1996.