An Object-Oriented View Onto Public, Heterogeneous Text Databases

Andreas Paepcke
Hewlett-Packard Laboratories
Palo Alto, CA, 94025

Abstract

Even though companies maintain highly-structured, traditional business data in relational databases, large amounts of information are available in semi-structured text sources, such as indexed online newspapers, patent information, literature citations or business profiles. This information is offered by commercial providers who maintain complete control over access language, schemas and update capabilities. One way to unify access to all of this material is to make it look like a collection of objects in an object-oriented database. We have prototyped such a view onto an information service that provides some 400 full-text, bibliographic and numeric databases. We explain how the illusion of object-orientedness is put together and how it is maintained in queries. We also show how we use the object-oriented approach to handle some classes of schema heterogeneity. Our query language is OSQL, an object-oriented extension to SQL.

1 Introduction

The desirability of convenient access to multiple, heterogeneous databases has long been recognized. The problem is getting more urgent as increasing numbers of databases come online and are made available not only to large MIS departments but to small businesses and individuals as well. We found that a large amount of important information is available from commercial providers in semi-structured text databases. These are sources that are subdivided into record-like fields. But the data within each field is usually not subjected to further structuring. Examples of such sources are online newspapers, patent disclosures, literature citations or business intelligence.

We designed and prototyped a solution that allows us to access one particular large set of publicly available text databases as if it were one object-oriented database.

After introducing in Section 3 the target information service we used for our prototype, we explain in Section 4 how we model its collection of databases in terms of types, functions and objects. In Section 5 we then show how this model allows the use of function overloading to address schema heterogeneity, in addition to affording unified access through OSQL, an object-oriented extension to SQL.

Section 6 lays out the advantages we gain from making our query translation and object materialization mechanisms object-oriented. We use type-allocated attributes and a hierarchy of *phrasebooks*, which mirror the data source hierarchy and hold the vocabulary information needed to drive the translation of queries from OSQL to the underlying search engines. Function overloading is used in the translation engine to manage different target languages.

2 Related work

In our choice of access language and the object-oriented approach to modeling underlying data sources we are most closely related to the Pegasus [1] project which uses extensions to Iris and OSQL [2, 3] to model relational and multi-media databases. In contrast, Mermaid [4] and MRDSM [5] presented a relational model to the user. The idea of adding an object layer on top of other data models is also discussed in [6].

Regarding techniques for schema integration, we may position the work described in this paper somewhere between the Multibase [7] and Mermaid efforts on one hand and the MRDSM effort on the other. Both Multibase and Mermaid called for a global schema to insulate the user from schema differences of underlying databases. MRDSM is based on the conviction that effective global schemas are too difficult to design and that multi-database access should instead be based on the principle of *interoperability*, resulting in a system of *federated* databases, rather than the illusion of one single database[1]. In our work we are attempting to find a synthesis of the two techniques in that we use a type hierarchy to model the databases and their schemas. By using inheritance we can reflect structural and semantic schema similarities in our model. But we can also accommodate "outlier" databases whose schemas are too dissimilar to shoehorn into one global schema. They simply occupy an isolated place in the type hierarchy.

While we believe that our approach affords progress in the area of schema integration, we only provide a framework for dealing with the problem. Plenty of difficulties remain when actually trying to handle semantic differences in any particular case. These are subject to research in their own right [9, 10, 11, 12].

Our focus on text databases is very different from systems like Multibase, which provided access to the relational and CODASYL models, and from the approaches taken in Mermaid or MRDSM which addressed relational sources only.

3 The Dialog Information Service

We chose *Dialog*[2] *Information Services, Inc.* to test our approach. This service provides about 1.4 terabyte of data distributed over some 400 databases. These include full-text, bibliographic, tabular/numeric and record-like directory entries. Full-text databases include many newspapers and trade magazines, such as the *Los Angeles Times* and *PC Week*. Examples of bibliographic databases are the card catalog of the *Library of Congress* and *Medline*, an extensive reference for medical literature. Tabular/numeric information includes census data, forecasts, corporate financial profiles and many others. Examples of sources with directory format are databases about businesses and a reference on *Books in Print*.

The provider offers a unified boolean search language over all databases. This language is tightly integrated with the service's indexing which is available on all databases. A simple query over a movie database to find a film starring both Humphrey Bogart and Ingrid Berman could look like this:

```
begin 299
select ac=bogart and ac=bergman
```

The `begin 299` indicates that the following query(s) are to run over the movie database. The `ac=` constrains the search to the `actors/actresses` index.

[1] Reference [8] also introduces the distinction between these approaches.

[2] *Dialog* is a registered trademark.

Most of the service is managed through a terminal-based command line interface. Figure 1 shows how the user interacts with Dialog during a search session. A set of databases is selected[3]. All subsequent queries then range over these sources. Each query produces a (possibly empty) set of records which may be referred to by a name that is generated and returned to the user. These set handles are then used to display results in various formats. The movie query above returns:

```
               77  AC=BOGART
               64  AC=BERGMAN
      S1        1  AC=BOGART AND AC=BERGMAN
```

`S1` is the set handle. The middle column indicates that the result set has cardinality 1. We may view this single result record in various ways: we may ask for just one index field, such as the title:

```
type s1/ti/1
  .....
CASABLANCA.
```

We may also ask for a summary format which will return some subset of the available fields, or we may display the entire record.

Another purpose of set handles is query refinement. Consider the following query which retrieves movies with the word "tomatoes" in the title:

```
select tomatoes/ti
      S1        2  TOMATOES/TI
```

The two films found are *Fried Green Tomatoes* and *Attack of the Killer Tomatoes*. If we know that Jessica Tandy was involved in the movie we are looking for, we may now refine this query as follows:

```
select s1 and ac=Tandy
                2  S1
               24  AC=TANDY
      S2        1  S1 AND AC=TANDY
```

While this query language is common for all databases provided by the service, the schemas of the databases are not. The term *schema* in this context means the set of index fields which are searchable.

This concludes our introduction to the information service we set out to access. The following section explains at the conceptual level how we model the service's components through an object-oriented view.

[3] Dialog provides various mechanisms to aid in the selection of appropriate databases. Their description is beyond the scope of this paper.

Figure 1: The Dialog User Interaction Model

4 Modeling the data sources

We chose the object-oriented approach for several reasons: (1) we expect its power to allow future modeling of multi-media sources. (2) We wanted the ability to add additional sources to the system without disturbing existing pieces, while at the same time leveraging similarities with sources already imported. (3) The large number of databases being accessed requires the ability to categorize them. The intrinsic hierarchical structure of object-oriented modeling affords this.

OSQL provides us with multiple inheritance, polymorphism with classical dispatch[4], as well as stored, derived and foreign functions which are indistinguishable to the caller.

When building an object-oriented model over data in an OODBMS or over a computational task in object-oriented programming, three basic questions must be answered before the modeling task can begin in earnest:

- What is modeled as a type?
- What is modeled as a method/function?
- What is modeled as an object?

These questions are important because incorrect decisions in this context tend to make the subsequent modeling processes awkward. In our design we answered them as follows:

- Databases within the service are modeled as types.
- Index fields in databases are modeled as functions on these types.
- Records within databases are modeled as objects.

In the following subsections we explain each of these decisions.

4.1 The use of type

The decision to model databases as types was based on the observation that in object-oriented databases, types have three main roles.

One role is the grouping of items into categories. An `employee` type delineates a collection of similar items, namely employee objects. Similarly, each database within our data service contains items that are in some way similar. The *Los Angeles Times* database, for instance, contains articles that appeared in that newspaper, which gives them coherence as a set. We observe that we can apply this set-defining role of type towards our goal of bringing order into collections of databases. The Dialog service has done this without formal notions of type, and we show in Figure 2 how they have attempted to provide overviews of their available databases through hierarchical categorizations.

This brings us to a second role that types play in object-oriented databases, that of limiting the search space of queries: Dialog allows the user to specify a member of one of the higher-level tiers in Figure 2 as the range of a subsequent query. Specifying `Business` would, for instance, cause about 200 databases to be searched. All would contain information relevant to business-related information needs.

Now consider Figure 2 not primarily as a categorization but as a type hierarchy. We see that the correspondence between categorization and type/subtype relationships can now be used in queries:

[4] Classical dispatch has the resolution of calls to overloaded functions controlled by the type of the first argument to the call.

Figure 2: Excerpt of the Database Categorization/Type Hierarchy

```
select article
   for each Business article
   where ...
```

`Business` provides the necessary typing of the variable `article`. The query translation process can then use this to determine the databases over which the subsequent `where` clause is to be applied.

In addition to categorizing information items by the sources they come from, we can use other criteria, such as their topic area. Since we follow the Iris model of allowing an object to belong to multiple types, such a topic hierarchy may live side by side with the source-based hierarchy and can provide the basis for a topic index.

The third role of type in an object-oriented system is to organize behavior[5]. The organizing principle is that of inheritance: functions are defined on types, and objects of those types or their subtypes may be used as arguments to these functions[6]. The applicability — and consequently availability — of a function in a call with some set of actual arguments is therefore controlled by the shape of the type hierarchy. In the following section we show how this interaction of functions and types can be put to use in the context of our problem.

4.2 The use of function

Databases in Dialog are indexed in semantically meaningful ways. All `Newspapers`, for instance, have index fields such as `caption`, `lead paragraph` and `headline`. The *Washington Post* database has some additional indexes, such as `column name` and `series title`. By modeling indexes as functions defined on types, we reflect the fact that indexes are properties of databases. In addition, we can put the behavior-organizing role of types to use. The indexes of `Newspapers` and the `WashingtonPost` may be captured through inheritance:

Type `Newspaper`
Functions:
  Caption
  LeadParagraph
  SectionHeading
  Headline

Type `WashingtonPost(Newspaper)`:
Functions:
  ColumnName
  SeriesTitle

The expression `WashingtonPost(Newspaper)` indicates that `WashingtonPost` is a subtype of `Newspaper`. `ColumnName` and `SeriesTitle` are available to `WashingtonPost` only. In addition, the `Newspaper` functions are inherited as well.

With this much of the model in place, we can formulate a complete query to retrieve articles in the *Washington Post* which contain 'Cuba' in the lead paragraph and 'Castro' as a person's name elsewhere.

```
select article
   for each WashingtonPost article
   where LeadParagraph(article) = "Cuba"
     and NamedPerson(article) = "Castro"
```

If we wanted the query to range over more than just the *Washington Post*, we could type `article` to be `Newspaper`, in which case all newspaper databases would be searched[7].

[5] Note that this includes organization of attributes because encapsulation requires that attribute values associated with objects should just be accessed as results of functions.

[6] There are, of course, many variations on how functionality is organized in different object-oriented systems. Reference [13] introduces some of them.

[7] NamedPerson is an index that allows a search to distinguish

This query is translated to `begin 146 ; select Cuba/lp AND Castro/na`.

In response to this query we must return a set of articles that match our search criteria. This brings us to the concept of an object in our model.

4.3 The Use of object

Items in databases are modeled as objects. Articles in the *Washington Post* database are therefore instances of type `WashingtonPost`. Note that this does not imply an *a priori* instantiation of all items in all data sources! We merely think of the items as objects when formulating queries. Only when items are found in the data source and need to be made available do we 'materialize' objects. We make an instance of the appropriate type. Then we extract the item's text from the underlying service and associate it with the new object. We may then apply the same functions to the object that we applied earlier as part of the query. Examples: `LeadParagraph(<object>)` would return the text of the lead paragraph, `NamedPerson(<object>)` would return a list of all names of persons in the article.

5 Schema integration through function overloading

Unifying the query access to a set of data sources is only the first step towards making the sources available. The differences in schema still pose a formidable problem, although its impact is not as severe for the text sources of our service as it would be for relational databases[8]. We may distinguish between structural and semantic heterogeneity in schemas. In our case of a collection of text sources, a structural discrepancy could be the absence of an index field in one data source that is present in another. Such a discrepancy is of practical importance especially when the two data sources have semantically related contents. If, for instance, one bibliographic database offers an index on book editors, a searcher would expect other bibliographic databases to provide this index as well.

between the use of words to identify a person and other uses. Example: 'Miller' used as a name vs. as a profession.

[8] This is because the use of index fields is optional. One may search for a string occurring 'anywhere in the record' of a database. A search for the string 'Miller' over two databases may therefore be successful even if one of the databases does not offer the NamedPerson index. Not taking advantage of index fields makes searches less effective in that it decreases retrieval precision, but some progress will still be made.

We can use this same example to illustrate semantic schema heterogeneity in the case of semi-structured text: one bibliographic database might use `Author` to cover both article producers and editors of books, while another source might distinguish between them by providing an `Editor` index. The meaning of the author fields is thereby affected. Another typical example of semantic heterogeneity are differences in unit of measurements: a European architectural data source might provide building dimensions in the metric system, while similar American databases contain English units.

The modeling of index fields as functions gives us an opportunity to provide some schema integration both structurally and semantically.

5.1 Removing syntactic field name differences

The simplest case occurs when two data sources have semantically equivalent fields with differing names. A bibliographic database might have a field named `Author` while a newspaper database calls its equivalent field `Byline`. In our model each of these fields will be represented by a function on its respective data source. By naming both of these functions `Author`, we have fixed the problem. The implementations of these overloaded functions provide the correct mapping during the query translation process.

Figure 3 illustrates the technique[9]. We show the two types `Bibliography` and `Newspaper` with some of the functions defined on them. The `Author` function on `Bibliography` holds the translation to use when searching for authors over bibliographic databases: `au=`. The `Author` function on type `Newspaper` holds the translation to use when searching over newspapers that use an abbreviation of the word "byline" as the field name. Similarly, the figure shows how we could hide the difference between the field names `Title` and `Headline`.

5.2 Supplying missing index fields

In some cases it may be possible to maintain the illusion that some index field is present when it is actually not provided by the underlying data source. If we frequently work with a movie database that provides a

[9] For clarity we will use some fictitious examples and simplify syntactic details. In particular, the Dialog service has done a good job at avoiding gratuitous inconsistencies in names of semantically equivalent fields. Syntactic integration of the kind described here will therefore be more relevant when we include textual data sources outside of Dialog.

Figure 3: Removing Syntactic Field Differences

Summary field, we might find it convenient to have the same field available for newspaper databases we work with regularly. Even though newspapers do not provide such a convenience, we can define a function on the newspaper type that represents newspapers in our model. We would call the function Summary, thereby overloading the movie database type's function. Its implementation might search over the newspapers' Title and LeadParagraph fields and return them to provide a summary capability. For the searcher the difference between such a derived and a 'built-in' function is transparent.

Figure 4 illustrates this example.

5.3 Resolving semantic discrepancies

Semantic discrepancies are the most difficult to smooth out. One difficulty this problem poses beyond the problems of the previous two sections is that we need to make adjustments not only when translating high-level OSQL queries into underlying service queries, but that we must also operate on data flowing out of the service to the user. Our model helps us because it is functional. This means that the places where translation activities must occur are well defined. Consider, for instance, the following query:

```
select compound
   for each ChemicalHandbook compound
   where NumberOfElements(compound) > 15
     and BoilingPoint(compound) > 120C
     and MeltingPoint(compound) < -10C
```

Assuming that the chemical handbook which is modeled by type ChemicalHandbook deals in units of Kelvin, the BoilingPoint and MeltingPoint functions on that type would have the job of converting the values of 120 and -10 from Celsius to Kelvin. The function that materializes objects would have to do the conversion in the other direction.

We now turn away from the user-level view to describe some parts of our implementation.

6 Object technology in the implementation

Both the user's model and the implementation of this model use the same object system, WS-IRIS which is a persistent, in-memory object store [14, 15]. This allows us to unify the user view and the implementation. The central idea is that functions used by end users in queries are overloaded in the implementation with functions that "do the right thing" for those user-level function invocations. This section describes the approach as applied to implementation-related information maintenance, the translation process and interactions with materialized objects.

6.1 Data structures for translation information

Consider the following query which searches in a medical bibliography *Medline* for articles by J. Smith that address clinical depression:

```
select article
  for each Medline article
  where descriptor(article) = "depression"
  and   author(article) = "Smith, J."
```

In order to process this query we must open a connection to the service and inform it that the following query is to range over the *Medline* database. Then the translated query must be transmitted to the information service that provides *Medline*. Here is the translation of the above query to the *Dialog* target language:

```
begin 152
select de=depression and au=Smith, J.
```

There are three categories of data needed by our implementation to perform this translation:

1. We must maintain run-time information about the status of the connection to the service.

Figure 4: Providing Missing Fields

2. We must be able to map the type identifier `Medline` to the Dialog database Medline identifier "`File 152`".

3. We must know that searches over article descriptors are effected through the expression "`de`" and that authors are searched for through "`au`".

The first two items are administrative *global information* which we keep as part of the database type hierarchy in type-allocated attributes. We call the kind of information exemplified by the third item *translation vocabulary*, and we maintain it in *phrasebooks*.

6.1.1 Phrasebooks for translation vocabulary

Given the diversity of sources in even a single information service, we cannot use a single translation table for organizing the vocabulary needed to drive the translation from OSQL to the underlying search engines. We instead need a data structure that can accommodate the differences in target schemas, while allowing us to take advantage of schema similarities. We use hierarchies of *phrasebooks* for this purpose.

A phrasebook is an object that is used to find strings to send to the underlying service to effect the correct search behavior. In our example query this is the mapping from the virtual database functions `descriptor` and `author` to the Dialog search expressions "`de`" and "`au`", respectively.

We organize phrasebooks using type hierarchies which mirror the hierarchy of databases. A single instance is made of each terminal in the phrasebook hierarchy. This instance is used by the query translator during the query compilation process to find the native search expressions. In order for the translator to find phrasebook instances, they are associated with their corresponding terminal in the database hierarchy by a type-allocated reference pointer in `phrasebook`. Figure 5 illustrates this.

Figure 6 shows a subset of the methods available on the phrasebook hierarchy of Figure 5.

The expression:

```
(author (phrasebook (gettypenamed 'Medline)))
```

would, for instance, return "`au`", which the translation process uses to translate the `author(article)` part of our example query. To translate function invocations of the form `func-name(arg)`, the translator therefore isolates `func-name` in the query and determines the type of `arg`. It obtains the phrasebook object associated with that type and applies method `func-name` to it to obtain the translation phrase. This is repeated for all function invocations, and the substrings obtained from the phrasebooks are pieced together to eventually form the complete search string.

Note in Figure 6 that we use inheritance to reflect regularities in the target schemas: *descriptors* are indexes available on all *Dialog* databases, and they all have the same field name: `de`. The `descriptor` method is therefore inherited by all Dialog phrasebooks and can be used in all queries over the Dialog service. If for some reason the field name (and therefore the translation) of the `descriptor` index were different for one of the databases, we would simply override the inherited `descriptor` phrasebook attribute to return the correct translation for the irregular database type.

6.2 The translation process

In the previous section we showed how we satisfy the information needs of the translation process. In this section we describe how we use function overloading to manage the dynamics of code generation for diverse target languages.

Since the Dialog service presents a single query language, the translation process is uniform for the databases of this one service. But we need mechanisms for choosing the correct translation algorithms when searching in other services. Our database type hierarchy affords this naturally because the service itself is at its root. We simply make `compileQuery` a method on that root type. This method takes an OSQL query string as an argument, and it returns a

Figure 5: Phrasebook Objects Associated with Database Types

CompiledQuery object. These objects contain information such as the names of databases involved in the query, the translated search string and various administrative items. An instance of DialogService can therefore be used for the compilation. If we add a new service to our system, we make that service and its member databases a sibling of DialogService and overload the compileQuery function, adding a new method that handles translations for that service.

Query execution is handled analogously, through an overloaded function ExecuteQuery which is defined on service types and takes a CompiledQuery object as an argument. It returns a result scan object[10] or a set of materialized objects, depending on how the OSQL query was formulated.

6.3 Interacting with materialized objects

A result object of a query can be materialized into our environment by making an instance of the result variable's type and placing the incoming text into one or more of this new instance's attributes. In our example query this would be an instance of type Medline. Our current protoytpe "lumps" an entire incoming information record into one attribute called contents. This approach makes the materialization process roughly as fast as the transfer path to the service allows. A disadvantage of the approach is that we lose the structure that is available in the service. For example, if the information item is a newspaper article, we can no longer easily extract just the author or title or publication date without parsing efforts on our part.

In our next prototype we plan to add the ability to apply the same functions to materialized objects that we can use in our queries. Given a materialized instance of Medline, we will then be able to apply, for example, the function author to it to extract just that piece of information. For our example, this will be the third use of the function author: it was used in the formulation of the query, it was used again to access the correct translation in the phrasebook object and it will be the way to extract information from the materialized object. This use of function overloading makes it much easier to understand and to extend the system because related functions have the same name.

7 Conclusion

We have introduced and critiqued a technique for accessing semi-structured text sources through an object-oriented data model. One clear advantage of

[10] A *result scan object* is a handle to the result set. It answers to functions such as nextObj or nthObj which extract objects from the result set.

Figure 6: Functions/Attributes on the Phrasebook Hierarchy

our approach arises from the fact that we may run OSQL queries over data sources of different models. Our approach also allows users to categorize data sources by contents. It allows us to model schemas in a way that preserves similarities while acknowledging that some schemas are too different to be seen as part of a global view over all participating databases. Function overloading allows us to 'fix' some schema differences, and we presented examples of this.

Acknowledgments

We thank Witold Litwin for suggesting this work and for his continued interest and encouragement. Tore Risch provided WS-IRIS, our database through which we model and access the underlying service and into which we materialize information. Vicki O'Day conducted interviews with research librarians that helped us understand the extensions we will need to fill real-life requirements. Robin Jeffries provided feedback to earlier drafts of this paper. Finally, Jurgen Annevelink was a critical listener and sounding board for early ideas.

References

[1] R. Ahmed, P. Desmedt, W. Kent, A. Rafii, W. Litwin, and M. Shan. The Pegasus project: Information management in a heterogeneous database environment. In *IEEE COMPCON*, 1991.

[2] D. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. Lochovsky, editors, *Object-oriented Concepts, Databases, and Applications*. Addison-Wesley Publishing Company, 1989.

[3] William Kent. Important features of Iris OSQL. *Computer Standards & Interfaces*, 13(1-3), Oct 1991.

[4] M. Templeton, D. Brill, A. Hwang, I. Kameny, and E. Lund. An overview of the MERMAID system – a frontend to heterogenous databases. In *Electronics and Aerospace Conference and Exposition, IEEE EASCON*. Institute of Electrical and Electronics Engineers, 1983.

[5] Witold Litwin and Abdelaziz Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(12), December 1986.

[6] Gio Wiederhold. Views, objects and databases. *IEEE Computer Magazine*, 19(12):37–44, December 1986.

[7] T. Landers and R. Rosenberg. An overview of Multibase. In H. Schneider, editor, *Distributed Data Bases*. North-Holland Publishing Company, 1982.

[8] Peter Fankhauser, Withold Litwin, Erich J. Neuhold, and Michael Schrefl. Global view definition and multidatabase languages - two approaches to database integration. In *Proceedings of the European Teleinformatics Conference (EUTECO 88)*. North Holland, 1988.

[9] Linda G. DeMichiel. *Performing Database Operations Over Mismatched Domains*. PhD thesis, Stanford University, 1989.

[10] Amar Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, New York, 1989. This is a collection of papers on the subject.

[11] Heinz Ulrich Hoppe, Karin Ammersbach, Barbara Lutes-Schaab, and Gaby Zinssmeister. EXPRESS: An experimental interface for factual information retrieval. In *Proceedings of the Thirteenth International Conference on Research and Development in Information Retrieval*, 1990.

[12] William Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, 1991.

[13] R.P. Gabriel, J.L. White, and D.G. Bobrow. Clos: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9), September 1991.

[14] Witold Litwin and Tore Risch. Main memory oriented optimization of oo queries using typed datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), December 1992.

[15] T. Risch and G. Wiederhold. Building adaptive applications using active mediators. In *DEXA 91.* Springer Verlag, 1991.