Building a Federation of Heterogeneous (Non-Database) Repositories

Markus Tresch   Uwe Röhm
Institute of Information Systems
Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland
E-mail: {tresch,roehm}@inf.ethz.ch

1 Introduction

Today's database management systems are more and more criticized that they are huge monolithic blocks of software, expensive and difficult to handle, and therefore not suited for managing small personal data sets. As a consequence, still relatively few data is managed by database systems and the majority is stored in heterogeneous repositories, like for example file systems, spreadsheets etc.

We are developing a federated database management system that provides (i) a uniform view of and (ii) advanced database functionality over data externally stored in repositories. Our particular interest are along the following two main lines of investigation:

- **Service-Oriented Federation.** We propose a service-oriented federation layer, that provides selected database services, like for example data modeling, integrity constraints, query processing/optimization or transaction mechanisms, for externally stored data. The goal is to achieve a federation layer that is as thin as possible and as thick as necessary.

  Current approaches to federated databases are extended in that we allow component systems to be non-database management systems. In particular, we include repositories like file systems or "black-box" application systems, e.g. a library information system. We aim to export lacking database functionality to such repositories.

- **Gradual System Integration.** While integrating local repositories into a global federated system, only selected parts of the component systems are usually required. Furthermore, many repositories do not provide meta-data that is required to integrate their data.

  As a consequence, (semi-) automatic integration methods have in most cases shown to be impractical. We therefore propose a different approach to facilitate the integration of heterogeneous repositories into the federation. So called "object-likeness" (see also the likeness concept of [1]) is used to gradually integrate repositories and hence stepwise achieve more and more refined object-oriented views of external data.

In this short paper, we focus on providing distributed object management functionality for external non-database repositories and show how build a federated system that exports data modeling, query processing/optimization, and integrity constraints services. These investigations are part of the COSMOS research project of the database research group at ETH Zurich. Further information is available at http://www-dbs.inf.ethz.ch.

1.1 Running Example

As a running example, we consider BibTEX files which are used by the TEX system for storing literature references. We want to join the BibTEX entries stored in these files with the book records of our departmental computer science library system. The library system can be accessed via a WWW search interface. In-depth analysis of these two repository cases reveals several challenges:

1. **Iterator Interface to BibTEX Entries**
   Files in BibTEX format can be seen as lists of objects. Hence, a BibTEX entries can be accessed through an iterator interface which typically offers operations like getting the first/next object or deleting/inserting/updating the current object (BibTEX entries).

   Typical questions which can be answered by such an interface are "Retrieve the nth object." or "How many objects are available?". To get a more sophisticated iterator interface capable of retrieving an arbitrary object the type of objects stored must be specified.

2. **WWW-Form Interface to the Library System**
   The library system acts through its WWW search interface like a bag of objects (books). One can search for particular attributes and retrieve (a list of) whole objects. But the interface does not provide concepts like a current object or navigation along a given order. It is even not possible to ask for all attributes of the objects but only some of them.

   The facilities of such an interface are quite contrary to the above mentioned iterator interfaces. We cannot for example count the number of objects stored in library or ask for the nth object. One can just fill in the input form and retrieve an answer report. Typical questions are "Which objects have these specific values?".

The conclusions we can draw from our investigations are that BibTEX files can easily be transformed and integrated into a federation. Basically we have to provide some fundamental functions to iterate, access, and modify single BibTEX entries in such files. They are well suited for integration as their overall structure is like a list of records which can be directly modeled as sets of objects.

The library search interface however is less suited for integration. First this is a read only repository, second its interface is severely restricted in that it can only be asked for a limited subset of all possible attributes. For example, we can ask for the ISBN number of all books of a particular author, but we cannot ask for the author of a book with a given ISBN number. Although we can wrap the library system into a set of objects, the functionality is quite restricted and most access functions and all update functions are not supported. Obviously, we have to deal with quite different access functionalities.

In the sequel, we are going to discuss some selected issues of how to attack this repository heterogeneity. We will discuss methods and technologies for gradually integrating such systems into a federation.

2 Service-Oriented Federation

In this section, we describe the services of the federation layer. We focus an services for data modeling, query processing/optimization, and integrity constraints.

2.1 Data Modeling

The core of our federated database, the distributed object manager, uses an object-oriented data model to provide a uniform view of all external repositories as collections of objects that can be modeled, queried, and updated by a single language. This makes distribution and heterogeneity (and to some degree autonomy) of external repositories transparent.

The advantage of having an object-oriented common data model has widely been acknowledged [4]: it is semantically rich and allows a variety of abstraction mechanisms; it allows the behavior of objects to be captured through methods; it allows to integrate non-traditional databases and non-database repositories through behavioral mapping; finally the meta-class mechanism adds flexibility to the model and allows refinements of the model itself.

One possibility for such an object-oriented data model would be to extend the ODMG data model [2] for distributed environments [5]. Based on former work done by our group [8], we are extending the developed object database language COOL* as described in [7]. The main advantages of COOL* over ODMG are its formally defined object algebra, the updatable view mechanism, and the generic update operations. As we will describe later, the object algebra serves as starting point for query optimization and the view mechanism is used for gradual integration of repositories. As an example, we show in Section 3, how to model BibTEX-Files and the library system using COOL*.

In order to wrap BibTEX and library entries with an object-oriented data model we have to do more than structural and behavioral conversion. An important problem arises of the necessity to maintain object identifications in the federated database.

An object identity (OID) assigns to any object in the federation a unique identifier. In general, an OID guarantees two properties: (a) there are at no time two objects with the same OID, (b) OIDs remain unchanged for an object's life time and are not reused after an object is deleted. Since not all external repositories, in particular the non-database systems, know about OIDs, we have to distinguish three sort of repositories.

- **Strong OID Repositories.** The repository has itself a notion of object identity which is maintained by the system and provides OIDs that fulfill properties (a) and (b). Examples of this category are object-oriented database systems.
- **Weak OID Repositories.** The repository has itself a notion of identifying objects using OIDs fulfilling only property (a). In this category fall all repositories with possibilities to define unique, value based keys, like for example relational databases. These systems identify each object by a unique combination of values. These values however may change over time or can be assigned later to another object.
- **No OID Repositories.** The repository has no notion of identification. One may ask for an object with particular properties and receive different answers at different times.

Applied to our example we see that BibTEX files and BibTEX entries provide weak object identification as the unique file name and the unique BibTEX key respectively may be reused for identifying files and BibTEX entries. The library system has strong object identification facilities as each book has its own unique signature. This key remains unchanged for an object's life time and will not be reused after a book is deleted (which does not occur that often anyway). Hence, both repositories provide a notion of a key attribute we can use to implement OIDs from.

2.2 Query Processing and Optimization

If we think about exporting database functionality to external (non-database) repositories, one of the most important services to provide is query processing. For example we want to be able to query BibTEX files for specific entries and even join these information with book entries of the library system.

Using the above presented distributed object manager we are already capable of formulating queries over different repositories in the common data model. Although some (non-database) repositories may not provide meta information about the data stored, it is up to the repositories' wrapper to present them as collections of objects which we than further refine by defining views on them.

An important issue of federated query processing arises from the heterogeneity of data sources. In general, the power of the single query languages will differ very vast. In the best case they dispose of special abilities like e.g. image querying by contents in a special image database system. In the worst case they offer less or even no common query facilities like for example the BibTEX file.

The federation layer's query service has to determine the individual query facilities of the repositories in order to translate the global query into efficient local sub-queries. Query facilities mean both the expressiveness of local query languages and the existence of further query processing engines like for example a local optimizer.

In non-database repositories most of the above observations are not transferable. File systems for example neither provide a query language, nor local query information (statistics) or an optimizer. The query interface of our CS department's library system also offers only very limited facilities. In fact, the federation layer does not know in detail, which possibilities the library system has. One may just guess that some indexes exist.

In both cases most of the query execution must be done by the federated database itself. To improve querying speed of such "simple" repositories, the federation layer must export physical design capabilities to such repositories. This means to build indexes in the federation over external stored data.

For example, it may create an index on the keys of the BibTEX entries in the file system. This index is managed by the federation and is used by the global query optimizer. In order to obtain query statistics the federation system can scan the repository to collect statistics and selectivity estimations.

Another possibility is the replication of external objects. All approaches introduce some redundancy into the system which must be dealt with.

Depending on the autonomy granted to the repositories further problems can arise. A global query may interfere with local queries which is not predictable from outside if a repository does not tolerate at least some global control. One solution is that the federated database must also deal with some sub-queries not be answered at all (or answered later) by the repository. There has been work done on that issue recently [6].

2.3 Integrity Constraints

One main purpose of a data model is the expression of integrity constraints that must apply to all instances of a repository. Whereas database systems usually have a data model and allow for integrity constraints that are checked and enforced by the system, non-database repositories in general lack of such a possibility. The integrity constraints service of the federation allows for modeling and enforcing integrity constraints for external repositories.

Integrity constraints in federated database systems can be local and/or are global constraints with respect to the following issues: Where is the constraint modeled (in a local repository schema, in the global federated schema, or equivalently at both places)? How many databases are involved in the constraint (one single local repository or the global federation by spanning multiple repositories)? Which system ensures the integrity constraint (a local repository system or the global federated system)?

Modeling Constraints. In order to understand to power and the limitations of this approach, we classify integrity constraints in federated databases as follows:

- **Locally modeled constraints** are known to the repository schema only. Consequently, these constraints must always be enforced by the repository system itself, since the global federation is not aware of them. These constraints can restrict their own local database only. They may occur if either the local repository does not export the constraint to the federation, or the federated data model does not have the ability to model the constraint. Updates to the federated schema may violate these local constraints.

  Consider as an example of a locally modeled constraint our BibTEX files, where no two files with equal filename are allowed within the same directory, and the global federation does not express this uniqueness of filenames.

- **Globally modeled constraints** are modeled in the global federated schema only. These constraints must always be enforced by the global federation layer, since the local repository does not know about them. They may be violated by updates local to the repository. Globally modeled constraints are distinguished in single- and multi-repository constraints.

  Single-repository constraints in the global schema refer to one single underlying repository, that is, they restrict the data of one local database. They occur if the federation adds a constraint to a repository because the repository itself is not capable in modeling the constraint. Consider as an example again the file system, where files may or may not have execution permission, but there is no possibility to prevent that somebody sets text files, like e.g. BibTEX files, to be executable. The federated system may define such a constraint for the file system.

  Multi-repository constraints span multiple repositories. They are used to define inter-database constraints, that become important after databases start cooperating. An example of a multi-repository constraint modeled in the global schema are referential integrities between books of the BibTEX files and the departmental library.

- **Locally and globally modeled constraints** are equivalently modeled at both places, in the repository schema as well as in the federated schema. Since these constraints are modeled at the local repository too, they must always be single repository constraints. Locally and globally modeled constraints may be defined bottom-up, first in a repository schema and then translated to the federation, or top-down, first in the global schema an then translated to the corresponding local repository schema.

Enforcing Constraints. Constraints modeled in a local and the global schema at the same time are distinguished in locally vs. globally enforced constraints:

- **Locally enforced constraints** are ensured by the underlying repository. The global system acts like a client, that is, for any update on the global schema that may violate the constraint, the federation asks the local repository to approve the operation.

An example of locally ensured constraints are referential integrities in relational databases that are modeled and ensured in the relational repository, and the constraint is just translated to the global schema.

- **Globally enforced constraints** are ensured by the global system. Although the repository does know of this constraint, it cannot ensure it itself. The federated system acts like a constraint server and ensurer, that is asked by the repository to approve local updates.

  An example of globally ensured single-repository constraints are referential integrities of repositories that do not know themselves of the notion of referential integrity. The global system adds this notion at the global level.

It turns out that enforcing these variety of constraints demands a rather flexible mechanism. Some constraints require a very restrictive handling of violations. They must be rejected by the system, that is, an exception is raised and the operation fails. An example of a strict constraint is the write protection of a file. Any update trying to delete this file is rejected and an error message is prompted. Other constraints allow for a more constructive handling, such that updates that try to violate these constraints successfully terminate, but afterwards the systems invokes either a correcting or compensating action.

Correcting actions do some follow-up updates that transform the repository to the next well defined and correct state, which may be different from the pre-update state. An example of correcting constructive constraints are views. Updates may insert objects into a view, even if the object does not fulfill the condition for being in the view. The system allows this update but starts afterwards a corrective action, that recomputes the view.

Compensating actions undo the update and completely re-establish the state before the update. This is a special case of corrective actions. An example of a compensating constructive constraint is, if an update deletes the value of a data item that is defined to be not-null. This update may succeed but the system afterwards sets the data item back to its previous value.

An interesting approach of how to enforce local and global integrity constraints in a federated system is to create monitors that observe local update operations and send messages to the federation in order to provide global coordination [3].

3 Gradual System Integration

As previously mentioned, gradual step-wise integration of external repositories is a main issues in this project. This stems from the experience that wrapping external repositories is an industrious but uninspired piece of work that can never be fully automated. Nevertheless, we aim to make it as easy as possible by introducing a concept called “object-likeness”. The core idea is that we use object-views to define external heterogeneous data to look like sets of objects.

We now show how stepwise system integration can be achieved in our example using the facilities of our distributed database language COOL*.

(1) The first step in wrapping an external repository is to assign a logical name and to specify a repository category. For example, `BibTeXDB` is defined to be a repository of category `FileSystem` and `CSLibDB` of category `WWWInterface`. This information determines the basic behavior of the repository as well as the primitive functions to initialize, connect, and disconnect to the external system. Further typical repository categories are `RelationalDB` or `ObjectDB`.

```
define repository BibTeXDB as EXTERNAL(FileSystem)

define repository CSLibDB as EXTERNAL(WWWInterface)
```

As a result of the above definitions, two new object classes are created named `Objects@BibTeXDB` and `Objects@CSLibDB` respectively. So far, these classes are not related to each other, and it is not yet defined how to access their objects.

(2) In the second step, views are defined that present selected data of the external repositories as object sets. For example, views `Files` or `BibEntries` of repository `BibTeXDB` represent bibliographic entries and files. View `Books` of repository `CSLibDB` represents items of the library.

```
define view Files as EXTERNAL(BibTeXDB)
  on iterate    do ...
  on retrieve   do ...
  on create     do ...
  on delete     do ...

define view BibEntries as EXTERNAL(BibTeXDB)
  on iterate    do ...
  on retrieve   do ...
  on create     do ...
  on delete     do ...

define view Books as EXTERNAL(CSLibDB)
  on retrieve   do ...
```

For each view, functions must be defined that are used to iterate and retrieve on these object sets. In addition, functions to `create` a new object and to `delete` an existing object from/to are required. Because repository `CSLibDB` is wrapped through a read-only WWW interface, view `Books` will have a `retrieve` function only. Notice also that this view does not have an `iterate` function, because as we discussed earlier, iteration through this particular WWW interface is not possible.

(3) Views are further refined with the definition of attributes. We define attributes file name `fname` and file extension `fext` for view `Files` as externally stored values of type string. For each attribute, functions to set and get their value must be specified. For set-valued attributes, additional functions to add/remove values to/from the attribute must be defined. Notice, that no functions to change the values of the view `Books` are defined, since this repository is read-only.

```
define view Files'
  as extend[fname: EXTERNAL(string),
            fext:  EXTERNAL(string), ...](Files)
  on get[fname](o)        do ...
  on set[fname:= v](o)    do ...
  ...

define view BibEntries' as
  extend[key, author: EXTERNAL(string),
         year:        EXTERNAL(integer),
         ...
         infile:      EXTERNAL(string)](BibEntries)
  on get[key](o)          do ...
  on set[key:= v](o)      do ...
  ...
  on get[infile](o)       do ...
  on set[infile:= v](o)   do ...

define view Books'
  as extend[SI,AU,TI,SE,PU,PL,YR,IS: EXTERNAL(string)](Books)
  on get[SI](o)           do ...
  on get[AU](o)           do ...
  ...
```

(4) The attribute `fext` can be used to define a specialized view `BibFiles` as a selection of those files, having file extension "bib".

```
define view BibFiles
   as select[fext = "bib"](Files')
```

(6) To model the relationship between "BibEntries" and "BibFiles", view `BibFiles` is extended with relationship `entries` returning a set of objects representing all bibliography entries of that file. Vice versa,

a relationship `bibfile` is defined, returning the object representing the file of which the bibliography entry is a part of.

```
define view BibFiles'
   as extend[entries: set of BibEntries] (BibFiles)
   on get[entries](o)        do select[inname(b) = fname(o)](b: BibEntries)
   on add[x](entries(o))     do set[bibfile:= o](x)
   on remove[x](entries(o))  do set[bibfile:= NULL](x)

define view BibEntries'' as
  extend[bibfile: BibFiles](BibEntries')
  on get[bibfile](o)      do pick(select[infile(o)=fname(b)](b:BibFiles))
  on set[bibfile:= v](o)  do set[infile:= fname(v)](o)
```

Since these are all views (derived information of external data) all relationships between objects must be derived too, that is, implemented using query expressions. Notice that functions to add/remove objects are defined for the set-valued relationship `entries` instead of one function to set its value.

4 Summary and Outlook

We have described our vision of an open service-oriented federation layer that export database functionality to data externally stored in heterogeneous non-database repositories.

In the future, we will continue implementing our prototype. We are particularly interested in general concepts for easy wrapper specification, global integrity management and distributed query processing / optimization. Another aspect of on-going work will be the application of our proposed service-oriented federation to real-world problems and the preparation of industrial utilization.

References

[1] S. Blott, L. Relly, and H.-J. Schek. An open abstract-object storage system. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Montreal, Canada, June 1996. ACM Press.

[2] R. Cattell, editor. *The Object Database Standard: ODMG-93 (Release 1.2)*. Morgan Kaufmann, San Francisco, 1994.

[3] M. C. Norrie, W. Schaad, H.-J. Schek, and M. Wunderli. CIM Trough Database Coordination. In *Proc. Int. Conf. on Data and Knowledge Systems*, May 1994.

[4] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object orientation in multidatabase systems. *ACM Computing Surveys*, 27(3), June 1995.

[5] E. Radeke. Extending odmg for federated database systems. In *Proc. 7th Int'l Conf. on Database and Expert Systems Applications*, Zurich, Switzerland, Sept. 1986.

[6] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Proc. IEEE Int'l Conf. on Distributed Computing Systems*, 1996.

[7] M. Tresch. Principles of distributed object database languages. Technical Report 248, ETH Zürich, Dept. of Computer Science, July 1996.

[8] M. Tresch and M. H. Scholl. A classification of multi-database languages. In *Proc. 3rd Int'l Conf. on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, Sept. 1994. IEEE Computer Society Press.