Integrating Dynamically-Fetched External Information into a DBMS for Semistructured Data*

Jason McHugh and Jennifer Widom
Department of Computer Science
Stanford University
{mchughj,widom}@db.stanford.edu

Abstract

We describe the *external data manager* component of the *Lore* database system for semistructured data. Lore's external data manager enables dynamic retrieval and integration of data from arbitrary, heterogeneous external sources during query processing. The distinction between Lore-resident and external data is invisible to the user. We introduce a flexible notion of *arguments* that limits the amount of data fetched from an external source, and we have incorporated optimizations to reduce the number of calls to an external source.

1 Introduction

Most research on querying distributed, heterogeneous, information sources takes one of two approaches. The first approach is *data warehousing*, where information is brought into a single repository for querying and analysis [Wid95, LW95]. Drawbacks with this approach are that changes to the original data may not be visible immediately, and that all information of interest must be replicated in advance. The second approach is to retrieve information *on-demand* as queries are posed [Wie92]. A significant drawback to this approach is that the information sources may be expensive or unavailable to query on a regular basis. In this paper we present the *external data manager* of the *Lore* system, which combines the two approaches in a clean and practical way. The external data manager provides a mechanism for dynamically fetching, caching, and querying data stored at any number of heterogeneous sources, and integrating it seamlessly with data resident in the Lore system. Because data integrated from multiple sources tends to be irregular and incomplete [PGMW95], the Lore database system, designed specifically for managing semistructured data [MAG+97], forms an excellent platform for our work. Furthermore, the need to specify an external source within the database in our approach also argues for using a semistructured DBMS, as will be seen later in the paper.

As an example, consider a database consisting of information about states and regions. While most geographic information is stable, some information, such as weather data, is best obtained dynamically from outside information sources. A data warehousing approach would require external weather data to be integrated into the local database every time it changed, regardless of whether it was needed by a user. In contrast, the on-demand approach would require that all data, including stable geographic information, be obtained from external sources. Our approach allows the stable information to be stored permanently in Lore, while the dynamic information is fetched (and cached) on demand when needed to answer a user's query. Furthermore, the stable geographic data, the integrated dynamic weather information, and the internal descriptions of the external weather sources, all may exhibit

*This work was supported by the Air Force Rome Laboratories and DARPA under Contracts F30602-95-C-0119 and F30602-96-1-031.

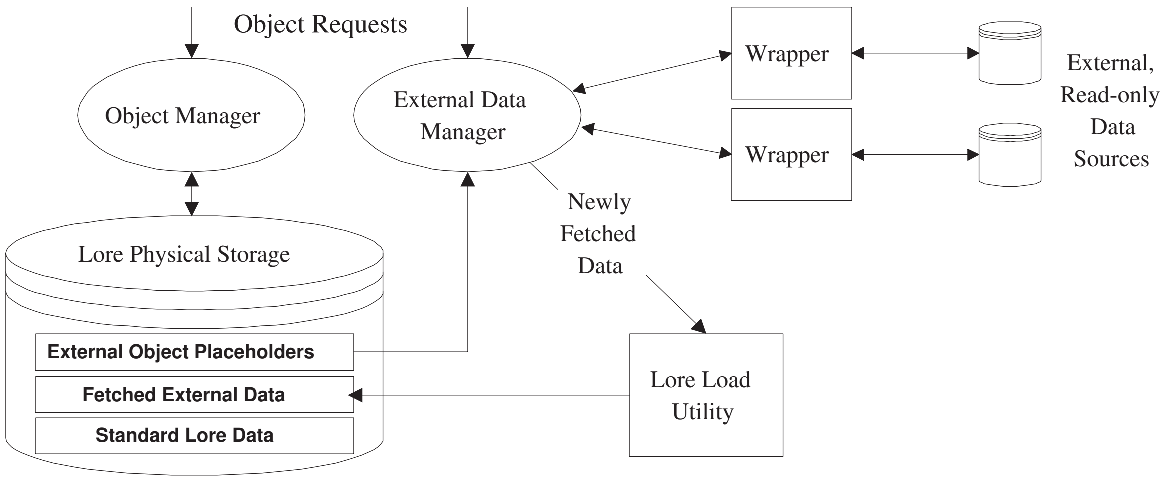

Figure 1: The External Data Manager Architecture

irregularity or incompleteness, arguing for using a DBMS such as Lore [Abi97].

Clearly there are many possible approaches that can be taken to integrate external data into a semistructured environment. Our main motivations in choosing the approach described below were to: (i) enable Lore to bring in data from a wide variety of external sources; (ii) make the distinction between local and external data invisible to the user; and (iii) introduce a variety of *argument types* and optimization techniques to limit the amount of data fetched from an external source. In Section 2 we introduce Lore and describe how the external data manager fits into the system. Further details on how we handle external data, especially our methods for reducing calls to external sources and reducing the amount of data retrieved, are described in Section 3. System status and future work are covered in Section 4. Due to space constraints, we do not provide a survey of related work, and a large number of details of our work are omitted from this paper. Nevertheless, this paper gives the reader the general functionality and structure of our external data manager.

2 Architecture

Lore (for *Lightweight Object Repository*) is a database management system designed specifically for semistructured data. For details on Lore and its architecture see [MAG+97]. Lore's data model is a simple, self-describing, nested object model called *OEM* (for *Object Exchange Model*) [PGMW95]. OEM data can be thought of as a labeled directed graph, where vertices are objects and directed edges represent object-subobject relationships. Leaf objects are *atomic*; all other objects are *complex*. Lore's query language, *Lorel* (for *Lore language*), is an extension of OQL [Cat94] supporting flexible path expressions for traversing semistructured data and extensive automatic coercion to relieve the user from the strict typing of OQL; for details see [AQM+96].

Lore's external data manager, and how it fits into the Lore system, is shown in Figure 1. Only the relevant portion of Lore is shown here; for the complete architecture see [MAG+97]. During query processing, requests for objects are sent from the query execution engine and are handled either by the *Object Manager* or the *External Data Manager*. The Object Manager services requests for data resident to the Lore database. The External Data Manager functions as the integrator between information stored at an external source and the local database—it is responsible for: (i) constructing requests to external sources based upon the current query and the local database state;

(ii) caching fetched external data along with information about the requests that generated it; and (iii) seamlessly integrating the external data during query processing.

The *wrapper* modules shown in Figure 1 accept requests from the external data manager (currently implemented as calls to the wrapper program) and translate them into specific commands for the external source. They also translate results from the external source into OEM before returning them to the external data manager. Since the related *Tsimmis* project at Stanford has focused on building wrappers that provide OEM interfaces to arbitrary data sources [PGGMU95], we are able to easily exploit such sources as external data in Lore.

The data stored within a Lore database can be divided into three categories: standard data, *External Object Placeholders*, and *Fetched External Data*. An external object placeholder, which is invisible to the user, specifies how Lore interacts with an external data source. Fetched external data consists of objects cached within Lore as a result of calls to external data sources. These objects are queried and retrieved just like standard Lore objects. In a Lore database, the external object placeholder and fetched external data for a single external source are stored as subobjects of a single object that we refer to as an *external object*. Details of the representation can be seen in Figure 2 and are described later.

The *Lore Load Utility* is a bulk loader designed to quickly load large amounts of data. It is shown in the figure because it is used by the External Data Manager to load into Lore the data returned from calls to external sources.

As a concrete example of how the components interact, consider the sample OEM database shown in Figure 2, ignoring for now the structure of the shaded (external) object and its subobjects. Suppose the query execution engine requests the `Name` subobject of the `State` object. Since the requested object is standard Lore data, the Object Manager handles the request. However, if the `Weather` subobject of `State` is requested, the request is handled by the External Data Manager, which may send a sequence of requests for information to the external source (further discussed in Section 3). Each external request is logged and the results are stored in the database via the load utility. After all requests are complete, the External Data Manager provides back to the query execution engine a set of objects corresponding to the relevant external information.

It is important to note that the external placeholder data, which in Figure 2 consists of all subobjects of the external object except the one labeled as "Fetched_Data", is used by the external data manager only and is not visible to the query execution engine or the user. External object placeholders are created by database administrators when a Lore database is created or when a new external source is "attached" to an existing database.

3 Details

Figure 2 illustrates a tiny portion of a geographic/weather database, as motivated in Section 1. The data includes some Lore-resident data about states and regions, along with a single external object (shaded in the figure) that fetches up-to-date weather information from an external source based on geographic area such as state or region, or (as will be seen) based on city within an area. The database includes information about a single state, "California", and a single region within the country "USA" which may be referred to either as "New England" or "Northeast".

As mentioned in Section 2, all subobjects of the shaded object except *Fetched_Data* constitute the external object placeholder. The *Wrapper* subobject specifies the program that interfaces between Lore and the external source. The *Quantum* subobject indicates the time interval (in seconds) until cached external information becomes "stale" and is no longer used. *Arguments* sent to the wrapper program (*Arg1*, *Arg2*, and *Arg3* in the figure) provide a way to qualify the information requested.

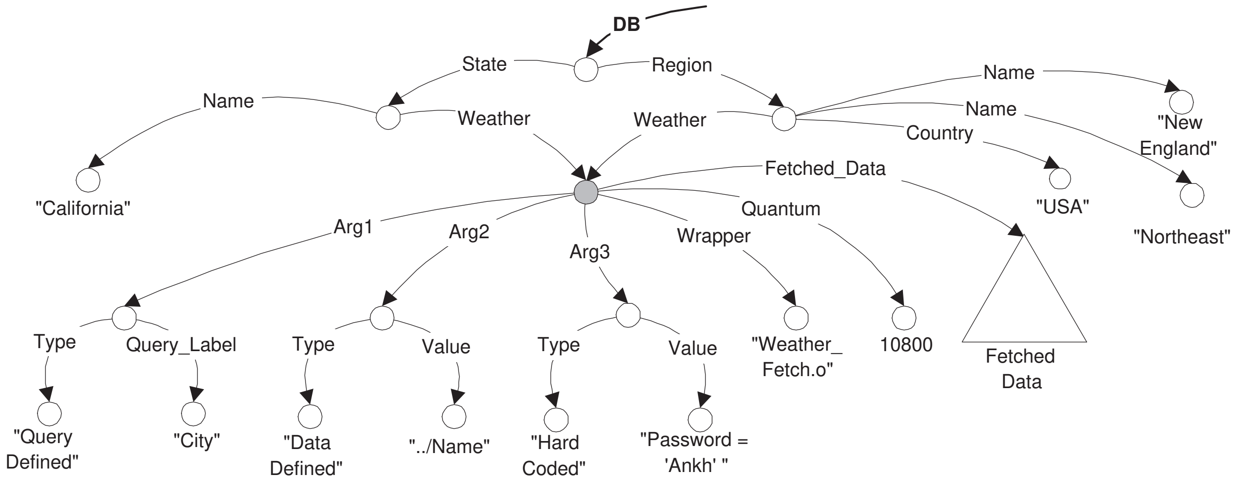

Figure 2: An Example OEM Database with an External Object

Ideally, arguments limit the data fetched from an external source to that which is immediately useful in answering the current query, thus: (i) reducing the cost associated with shipping data from an external location; (ii) reducing the amount of external data stored within Lore; and (iii) speeding up query processing by reducing the number of objects examined. Arguments sent to the external source can come from three places: the query being processed (*query-defined arguments*), values of Lore-resident objects (*data-defined arguments*), and constant values tied to the external object (*hard-coded arguments*). When dealing with arguments we must consider: (i) how argument values are extracted from the user's query and the current database state; (ii) how to combine argument values into a set to be sent to the source; and (iii) how to construct a sequence of such sets.

3.1 Single Argument Values

First, we consider how single argument values are extracted. The values of query-defined and data-defined arguments rely on the concept of *current path* to the external object. During query processing, the database is traversed from the root (edge *DB* in our example), and when the query processor encounters an object there is a specific path it has followed to get there [MAG+97]. In our example, the current path for the external object would have labels `DB.State.Weather` or `DB.Region.Weather`. The current path is readily available from the query execution engine.

We first describe a somewhat simplified approach for generating arguments, consistent with our (simplified) figure. We then explain the full generality of our approach. Hard-coded arguments, such as *Arg3* in the figure, supply their value directly from within the external object placeholder. A query-defined argument, such as *Arg1* in the figure, always includes a *Query_Label* subobject in its placeholder. The labels making up the current path, followed by the specified *Query_Label* say L, make up a path expression P. If the `where` clause of the current query contains $P = v$ for some constant value v, then $L = v$ becomes a value for the query-defined argument. A data-defined argument can either point directly to an atomic object in the database whose value will be sent as an argument (not illustrated in the figure), or it can specify objects via a "relative path" through the database as illustrated by *Arg2*. The relative path is evaluated with respect to the current path, resulting in zero or more objects whose values become arguments.

To be more concrete, suppose in Figure 2 the query processor calls the external data manager for the shaded external object via the current path labeled `DB.Region.Weather`. Consider *Arg1*. The *Query_Label* subobject with value "City" specifies that if the query being processed has any predicate of the form "`DB.Region.Weather.City = X`" in its `where` clause, then "*City = X*" is an argument value. In *Arg2*, the *Value* subobject specifies the relative path expression (in the Unix style) `"../Name"`. Based on the current path, the possible data-defined argument values are "*Name = 'New England'* " and "*Name = 'Northeast'* ". Finally, *Arg3* is a hard-coded argument specifying "*Password = 'Ankh'* ".

As mentioned above, our description of argument generation has so far been somewhat simplified. In the general case, each argument descriptor in the external object placeholder may include an optional *Tag* and *Operator* subobject, with atomic values t and op respectively. If these are included, once we have obtained an argument value v as described above, the actual argument sent to the source is "t op v". (Thus, in the simplified examples above, we have assumed tags *City*, *Name*, and *Password*, and operator =, although they are not shown in the figure.) In query-defined arguments, the relevant operator in the query must match the operator in the argument descriptor. However, as a further enhancement, a special operator `match` is permitted for query-defined arguments, which causes the corresponding operator in the query to be sent as part of the argument: In our example, if the query included "`DB.Region.Weather.City like` X" instead of "`DB.Region.Weather.City = X`", then the argument sent to the external source would be "*City like X*". This feature allows us to easily exploit operators available in external sources. Finally, if no *Tag* or *Operator* is specified, then the argument is sent without qualification, which does make sense for certain types of external sources.

3.2 Argument Sets and Calls to External Source

A single call to the external data manager may result in multiple calls to the external source due to multiple bindings for data-defined and/or query-defined arguments. A sequence of argument sets is generated, and each argument set results in (at most) one call to the external source. Pseudocode to generate argument sets is shown in Figure 3. Without loss of generality, assume that the query is in disjunctive normal form. In line 2 the algorithm extracts the hard-coded arguments, since these will be included in all argument sets. Line 3 begins a loop that will iterate over each disjunct in the query. In line 4 the query-defined arguments for the current iteration are extracted. They will be included in every argument set generated by lines 5 through 26, which handle data-defined arguments. If there are no data-defined arguments (the `If` on line 5) then the argument set consists solely of any hard-coded and query-defined arguments. Otherwise, line 8 begins a loop that packages the hard-coded, query-defined, and all possible combinations of data-defined values into a single argument set, which is added to the final sequence in line 14. The data-defined values are extracted by a sequence of GetFirstValue (line 9) and GetNextValue (line 23) calls, which simulate a general counting mechanism that builds argument sets that are a combination of all possible values.

Once argument sets have been computed, to actually obtain the information from the external source at most one call is made to the source for each argument set (based on optimizations described in the next subsection). In addition to the newly fetched information, previously fetched information also is used when we finally evaluate the query, to ensure that we provide the most complete answer possible based on the information we have obtained from the source. (Consider, for example, a query in which no argument sets are generated.) If previously fetched information is stale,

```
Set CreateArgumentSets(ExternalObject EO, Disjuncts disjuncts) {
   // Initialize the return value to NULL
   Set ArgumentSet = NULL;
   // Hard coded arguments never change
   HCA = HardCodedArgs(EO.HardCodedArgs);

   // For each disjunct in the query
   For each Disjunct, D, do {
      // First get the query-defined arguments
      QDA = QueryDefinedArgs (EO.QueryDefinedArgs, D);

      // Check for existence of data-defined arguments
      if (# DataDefined Args in EO = 0) then
        ArgumentSet += PackageArguments(HCA, QDA, NULL);
      else {
        // Now reset all data-defined arguments to their first values
        For I = 1 to # DataDefined Args in EO
           DDA[I] = GetFirstValue(EO.DataDefinedArgs, i);

        // Do a pairwise join between all possible argument values
        curDDA = 0;
        While (curDDA <= # DataDefined Args) {
            For each possible value, V, of the first DataDefined Arg {
              DDA[1] = V;
              // Package all arguments into a set and add to the sequence
              ArgumentSet += PackageArguments(HCA, QDA, DDA);
           }
           // Now determine the next combination of data-defined values
           curDDA = 2;
           While ((the Current value for DDA[curDDA] is the last value) AND
                  (curDDA <= # DataDefined Args)) {
              ResetToFirstValue(EO.DataDefinedArgs, curDDA)
              DDA[curDDA] = GetFirstValue(EO.DataDefinedArgs, curDDA);
              curDDA++;
           }
           // End case for this disjunct is when we have created argument
           // sets for all possible data-defined combinations
           if (curDDA <= # DataDefined Args)
              DDA[curDDA] = GetNextValue(EO.DataDefinedArgs, curDDA);
        }  // End of while more data-defined value combos exist
      }  // End of If data-defined arguments exist
   }  // End of For Each Disjunct
   return ArgumentSet;
}
```

Figure 3: Pseudocode to generate argument sets

1	Name = “California”, Password = “Ankh”
2	Name = “California”, City = “Middletown”, Password = “Ankh”

Figure 4: Argument Sets Generated Via the Current Path with Labels DB.State.Weather

1	Name = “Northeast”, City = “Middletown”, Password = “Ankh”
2	Name = “New England”, City = “Middletown”, Password = “Ankh”
3	Name = “Northeast”, City = “Smithville”, Password = “Ankh”
4	Name = “New England”, City = “Smithville”, Password = “Ankh”

Figure 5: Argument Sets Generated Via the Current Path with Labels DB.Region.Weather

it is refetched, based on the original query-defined arguments and current values for the other arguments.

3.3 Optimizations

It is clear that many calls to an external source can quickly dominate query processing time, so we incorporate certain optimizations to limit the number of calls to an external source. We make the assumption that more arguments result in less fetched data.[1]. Thus, we order the sequence of calls so that less restrictive argument sets are sent to the external source first. This is done by ordering the query disjuncts passed into `CreateArgumentSets` (Figure 3) so that those disjuncts which provide more query-defined values will come later in the generated sequence of argument sets: the disjuncts in the query are sorted by increasing number of query-defined arguments for the current path. By doing so we increase the odds that a given fetch will subsume a later fetch in the sequence. Argument sets used previously (by the current query or an earlier one) are tracked, and we determine when previously fetched (non-stale) information is guaranteed to include all data that would be fetched by the current argument set. This optimization occurs after the sequence of argument sets are generated. In such cases, the subsumed argument set is discarded.

[1] Although this property may not seem obvious at first, most external sources do behave in this manner. Consider, for example, a relational database with selection conditions, or a web site with an attribute-value forms interface.

3.4 Example

Suppose we are given the following Lorel query, which (take our word for it [AQM+96]) asks for weather information in cities named “Middletown” or in cities named “Smithville” located in a region of the USA.

```
Select DB(.State|.Region).Weather
Where DB(.State|.Region).Weather.City =
 "Middletown" or (DB.Region.Country = "USA"
 and DB.Region.Weather.City = "Smithville")
```

The path expression `DB(.State|.Region)` matches both *State* and *Region* subobjects of *DB*. Suppose the query is posed over the database in Figure 2, and consider the case when the external object is encountered with current path `DB.State.Weather`. The argument sets generated by the algorithm given previously are shown (in the order that the sets are generated) in Figure 4. The algorithm considers the second disjunct first, since it provides no query-defined arguments for the current path. The data-defined argument is evaluated with respect to the current path yielding a single value; this value is paired with the hard-coded argument, resulting in argument set 1. Argument set 2 is based on the first disjunct of the query. Here we have the same two arguments as set 1, but we also have a query-defined argument for *City*. Notice that argument set 2 is subsumed by set 1, and therefore set 2 will not result in a call to the external source.

Now suppose the external object is encountered with current path `DB.Region.Weather`. The argument sets generated by the algorithm are shown in Figure 5. Here both disjuncts provide one query-defined argument, so we can consider them in either order. The first two argument sets are generated from the first disjunct in the query, based on two matching values for the data-defined argument, a single query-defined argument, and the single hard-coded argument. Argument sets 3 and 4 are similarly generated from the query's second disjunct.

4 System Status and Future Work

The external data manager described in this paper is fully implemented within the Lore database system. We are currently extending the number of Lore applications that use external data in order to exercise the system, and we are contemplating appropriate performance measurements. We are also considering the following extensions.

- Currently cached external data becomes stale based on a simple timeout mechanism. For those external sources with a triggering mechanism [WC96], we would like cached data to become stale based on notifications from the source to the external data manager that the data has changed.
- Our current optimization techniques eliminate calls to external sources based on complete subsumption: a call is eliminated if it is guaranteed return strictly less information than a previous one (as shown in Example 3.1). We plan to explore how to "share" information fetched in multiple calls when the information is known to bear an intersection but not complete subsumption relationship.
- Lore contains a sophisticated index manager [MAG+97], but currently it indexes Lore-resident data only. We plan to explore how to properly index fetched external as well as external data not currently resident in Lore.

References

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

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), November 1996.

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

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing, IEEE Data Engineering Bulletin,* 18(2), June 1995.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. Technical report, Stanford University Database Group, February 1997.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, Singapore, December 1995.

[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.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1996.

[Wid95] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, November 1995.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.