Compile-Time Path Expansion in Lore

Jason McHugh, Jennifer Widom
Stanford University
{mchughj,widom}@db.stanford.edu, http://www-db.stanford.edu

Abstract

Semistructured data usually is modeled as labeled directed graphs, and query languages are based on declarative *path expressions* that specify traversals through the graphs. *Regular* (or *generalized*) path expressions use regular expression operators to specify traversal patterns. Regular path expressions typically are evaluated at run-time by exploring the database graph. However, if the database includes a structural summary such as a *DataGuide*, then an alternative approach is to expand regular path expressions at compile-time using the structural summary, reducing the run-time overhead of database exploration. This paper describes algorithms for compile-time regular path expression expansion in the context of the *Lorel* query language for semistructured data, and reports on performance results conducted on the *Lore* system illustrating the benefits of compile-time expansion.

1 Introduction

Efficiently storing and querying *semistructured* data—data that need not adhere to a fixed schema—has emerged as an important topic in the database research community [Abi97, Bun97, Suc98]. Researchers have addressed the design of query languages for semistructured data, e.g., [AQM+97, BDHS96, FFLS97], the implementation of prototype database management systems for semistructured data, e.g., [FFLS97, MAG+97], and query optimization issues related to semistructured data and path expressions, e.g., [BDHS96, FLS98, FS98, GGT96, MW98]. With the recent emergence of *XML*, a proposed standard for exchanging information on the Web [LB97], and the remarkable similarity of XML to typical models for semistructured data, support for query languages for semistructured data—and the performance of such queries over large semistructured databases—is of increasing importance.

XML, as well as other semistructured data, can be stored in a database system by modeling the data as a labeled directed graph. Queries use *path expressions* to describe traversals through the labeled edges in the graph. The efficient evaluation of a query then hinges upon choosing efficient traversal schemes for the set of path expressions in the query [MW98]. A *regular* (or *generalized*) path expression uses regular expression operators to specify a (possibly recursive) path *pattern*. Regular path expressions are particularly useful when the structure of the data is irregular, changes often, or is not completely known to the user. Modeling semistructured data as labeled graphs and querying it using a language based on regular path expressions is common to most research in semistructured data, e.g., [BDHS96, FFLS97, MAG+97].

Run-time evaluation of regular path expressions can be expensive. In this paper we explore improving efficiency by performing compile-time expansion of regular path expressions based on a structural summary (*DataGuide* [GW97]) of the current database. Compile-time expansion incurs the cost of exploring the structural summary and rewriting the query, but it can eliminate significant amounts of unnecessary database exploration at run-time. We have implemented our algorithms in the *Lore* DBMS for semistructured data [MAG+97], and preliminary performance results confirming the benefits of the approach are reported.

Our work is similar in spirit, but not in details, to [FS98]. In [FS98], a cross-product is computed between a *graph schema*—a summary of the database that must be small and reside in memory—and a representation of the query. From this cross-product an expanded version of the query is

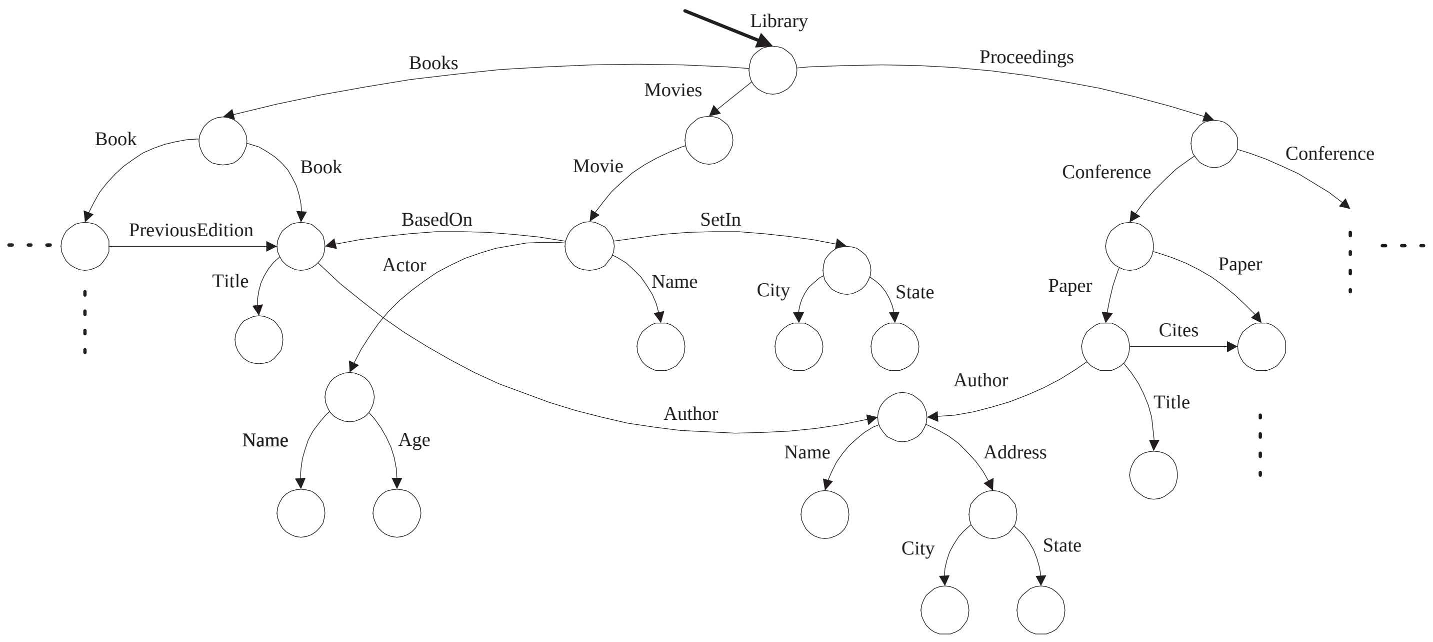

Figure 1: Semistructured library database

produced that is expected to execute more efficiently than the original. Our algorithm traverses the DataGuide (which corresponds roughly in our work to their graph schema) in order to rewrite the query. We do not require that the DataGuide is small since a full cross-product is not formed. We also do not require that the DataGuide reside in memory, and we thus must consider the I/O cost incurred when portions of the DataGuide are read from disk during query rewrite. We also address some additional query rewrites not covered in [FS98].

2 Setting and Examples

Our work is conducted in the context of *Lore*, a database management system designed specifically for storing and querying semistructured data [MAG$^+$97]. Lore's data model is the *Object Exchange Model* (*OEM*) [PGMW95]. Data in this model can be thought of as a labeled directed graph, with objects as vertices and edges representing the object-subobject relationship. Details on OEM and Lore can be found in [MAG$^+$97]. The query language supported by Lore is *Lorel* [AQM$^+$97]. Lorel is an extension of OQL supporting (among other features) declarative regular path expressions for traversing semistructured data. Since semistructured data need not conform to a predefined schema, in Lore we have introduced *DataGuide*s [GW97], which replace the traditional schema with a dynamically-generated structural summary of the database. The DataGuide stores information about all unique label paths in the current database graph.

Figure 1 shows a sample OEM database of library information with atomic values omitted. Because in our sample data we primarily show only one representative for each kind of information, with the exception of the two `Book` and `Conference` edges, this graph also serves as the DataGuide for our running example. Of course an actual database, including the one used for our experiments, would have a large number of books, movies, conferences, etc.

2.1 Lorel Path Expressions

In Lorel, a path expression specifies a traversal through the database graph. For example, "`Library.Books b, b.Book x`" is a path expression that binds to variable b all `Books` subobjects of the `Library` object, and to variable x all `Book` subobjects for a given object bound to b. A path expression is made up of a sequence of path *components*, where a component is a triple of the form $\langle$*SourceVar*, *SubPath*, *DestVar*$\rangle$.[1] The source variable can either be a variable or a *name*. A name uniquely identifies a single object and is used as an entry point into the graph. A *SubPath* is either a single label, or a subpath with regular expression operators that cannot be decomposed further.

A regular path expression is a path expression where one or more of the path components contains a subpath with a regular expression operator applied to it. For example, the path component $\langle x$, (.PreviousEdition)*, $y\rangle$ will bind to y all descendents of x that are reachable by following 0 or more edges labeled `PreviousEdition`. The path component $\langle x$, (.Cites|.PreviousEdition), $y\rangle$ will bind to y both `Cites` and `PreviousEdition` subobjects of x. In addition to the Kleene closure (*) and alternation (|) operators, Lorel includes operators for "one-or-more" closure (+) and optionality (?), and the subpath "#" is used to match 0 or more edges with any label.[2]

2.2 Examples

Here are three Lorel queries containing regular path expressions, intended for the database of Figure 1.

```
(1) Select x, y from Library.# x, x(.Cites|.PreviousEdition|.Sequel)+ y
(2) Select y, z from Library.Proceedings c, c.# x, x.Title y, x.Cites z
(3) Select y from Library.# x, x.Author y
    Where exists z in x.Title: z like "%stand%"
```

The first query returns database objects x together with other objects y that were "influenced" by x. Here influenced means that there exists a path of length ≥ 1 of `Cites`, `PreviousEdition`, or `Sequel` edges. The second query returns all titles of objects in `Proceedings` subgraphs along with the entries that they cite. Note that x can be bound to any descendent of c, but x must have `Title` and `Cites` subobjects in order to satisfy bindings for variables y and z. The third query includes the SQL-style "like" operator and returns all authors of objects that have "stand" in their title.

3 Compile-time Rewrites

We now introduce two schemes for eliminating regular path expressions at compile-time: *path expansion* (for eliminating *, +, and ? operators) and *alternation elimination* (for eliminating | operators).

3.1 Path Expansion

We use the DataGuide to replace all regular expression operators *, +, ? with alternations (|) representing possible paths in the current database. The rewrite algorithm effectively executes the entire regular path expression over the DataGuide and records the label paths that are seen.

[1]Lorel allows many shortcuts that eliminate variables and are convenient in practice. In the Lore implementation, all shortcuts are transformed into the fully instantiated path components that we describe here.

[2]Lorel also supports "label wildcards", and in fact "#" is an abbreviation for "(%)*" where % matches any label [AQM+97]. Label wildcards are expanded into alternations efficiently by the Lore preprocessor and thus are not considered further in this paper.

Since the DataGuide is typically much smaller than the database, we can efficiently replace regular path expression operators with the actual set of possible matching paths, eliminating unnecessary exploration of the full database at run-time. Cycles must be handled carefully: the semantics of Lorel is to traverse a data cycle no more than once when evaluating a path expression with a closure operator, and that same semantics must be preserved when we eliminate closure at compile-time. Note that we are assuming a "compile-and-run" scenario, with concurrency control ensuring that the structure of the database remains stable between compilation and execution. Obviously if the DataGuide can change between compile-time and run-time then our approach does not apply.

As an example, consider the subpath "`Library.# x, x.Author y`" in example query 3 (above). The path component $\langle Library, \#, x\rangle$ binds to x any descendent of the named object *Library*. However, the path is further restricted by the second path component $\langle x, \text{Author}, y\rangle$. By performing compile-time expansion of the #, we can ignore at run-time those database paths that don't lead to an `Author` subobject. For example, by applying this path to the DataGuide for Figure 1, we determine that the # can match only `Proceedings.Conference.Paper` and `Books.Book`, yielding an equivalent path expression "`Library(.Proceedings.Conference.Paper|.Books.Book) x, x.Author y`." The expansion of paths with + and ? proceeds similarly.

By expanding a regular path expression at compile-time using the DataGuide, we are guaranteed to visit, at run-time, a subset of the objects we would have visited with the original path expression. If the DataGuide is small and resides in memory then the expansion itself will be very fast and almost certainly worthwhile. However, when the DataGuide is large and may reside partially (or completely) on disk, it is less obvious that the cost associated with compile-time expansion, plus the cost of evaluating the expanded path expression, will be less than run-time evaluation of the original path expression. In Section 4 we empirically evaluate the expansion tradeoffs. We have some preliminary ideas on an efficient data structure that stores certain path statistics tuned to enable the optimizer to quickly choose whether or not to perform compile-time path expansion. Simple heuristics based on, say, the size of the DataGuide are not sufficient, but detailed coverage of this topic is beyond the scope of this paper.

3.2 Alternation Elimination

We can also eliminate alternation operators in regular path expressions by introducing either a `union` operator or a disjunct in the `where` clause. If the alternation appears in the `From` clause, e.g., "`From Library(.Book|.Movie) x, x.Title y`", then we can rewrite this clause as "`From ((Library.Book) union (Library.Movie)) x, x.Title y`". This transformation can be applied as many times as necessary, and if `union` is implemented properly it will not introduce any computational or I/O overhead to any execution strategies. Once an alternation is replaced with a `union` we can consider the following query rewrite:

$$\begin{array}{l} \text{Select s} \\ \text{From ...,((x.Book) } \texttt{union} \text{ (x.Movie)) y,...} \\ \text{Where w} \end{array} \rightarrow \left(\begin{array}{l} \text{Select s} \\ \text{From ...,x.Book y,...} \\ \text{Where w} \end{array} \right) \texttt{Union} \left(\begin{array}{l} \text{Select s} \\ \text{From ...,x.Movie y,...} \\ \text{Where w} \end{array} \right)$$

Here we have replaced the `union` expression in the `From` clause with two queries connected via a `union`. In Section 4 we give examples showing when this transformation is advantageous.

When the alternation appears in the `Where` clause, such as "`Where exists y in x(.Subject |.Keyword): y = 'Semistructured'`", we can rewrite it using a `union` and make the transformation as above, or we can rewrite it using disjunction: "`Where exists y in x.Subject: y = 'Semistructured' or exists z in x.Keyword: z = 'Semistructured'`". Note that in general, to introduce disjunction, the `Where` clause must first be expressed in disjunctive normal form,

Path Expression	Compile-time Expansion	Query Execution	Total	Execution of Original Path
Library.#.Name	0.35	28.74	29.09	54.10
Library.#.Title	0.21	10.0	10.21	33.43
Library.Books.Book.#.Name	0.07	5.37	5.44	10.05
Library.Movies.#.Title	0.1	2.84	2.94	12.30
Library.Movies.Movie.Actor.#	0.01	3.07	3.08	3.12

Table 1: Execution times for small database

Path Expression	Compile-time Expansion	Query Execution	Total	Execution of Original Path
Library.#.Title	0.77	102.65	103.42	412.13
Library.Books.Book.#.Name	0.18	99.10	99.28	126.74
Library.Movies.#.Title	0.24	83.73	83.97	223.10
Library.Movies.Movie.Actor.#	0.01	35.4	35.41	37.93

Table 2: Execution times for larger, cyclic database

and care must be taken to ensure that quantification and scope remain correct. The importance of this transformation is that it allows us to take advantage of an index created over `Keyword` objects when no corresponding index exists for `Subject` objects, or vice-versa.

4 Implementation and Performance

In this section we empirically evaluate the benefits of the compile-time rewrites described above. We have extended the Lore system to expand path expressions containing #, +, and ?, using the DataGuide. Lore does not currently support a general query rewrite mechanism to implement the `union` rewrite described in Section 3.2, so we have hand-fed the original and rewritten queries in order to evaluate the effectiveness of the transformation.

4.1 Path Expansion

In our first set of experiments we compare execution times for path expressions containing #, with and without path expansion. (Recall that # matches any label path of length 0 or more.) We use synthetic library databases similar to Figure 1. Atomic values are unimportant since we are concerned primarily with paths through the database. Our database generator creates graphs using parameters such as number of books, average number of authors, percentage of books that are sequels, etc. In Tables 1 and 2 we present execution times (in seconds) for path expression evaluation with and without path expansion. The numbers in Table 1 were generated by running Lore over a small library database of about 31,028 objects and 42,270 edges. In this database there are 2,000 books, 4,000 authors and actors, and 1,000 movies. The data is not tree-structured, but there are few cycles in the graph, and the DataGuide consists of about 100 objects. No indexes are used. For the execution times given in Table 2 the database contains 132,727 objects and 245,335 edges, with 10,000 books, 20,000 authors and actors, and 10,000 movies. Even though the data follows the same general form as the first database, the data is generally more cycle, and the DataGuide is about double the size of the first DataGuide. Again no indexes are used.

Key	Query
A	Select x From Library.Books y, y.Book x where x.(Title|Keyword) = "Armageddon"
B	(Select x From Library.Books y, y.Book x Where x.Title = "Armageddon") `Union` (Select x From Library.Books y, y.Book x Where x.Keyword = "Armageddon")

Table 3: Key for Table 4

Query	Execution Time	Notes
A	5.9	No index used.
B	4.5	Index created and used over `Keyword`. No index over `Title`.
B	4.9	Index created and used over `Title`. No index over `Keyword`.
A	0.017	Index created and used for both `Title` and `Keyword`.
B	0.019	Index created and used for both `Title` and `Keyword`.

Table 4: Execution times for alternation elimination

Tables 1 and 2 show that compile-time path expansion can reduce overall execution time by up to 75%. For these databases and DataGuides, the time to perform expansion is dwarfed in all cases by actual query execution time. As can be seen, there is essentially no benefit to expanding the # operator when it appears at the end of the path, i.e., the last row in each table, since in this case expansion does not prune any paths from consideration at run-time. (The small difference in query execution times is due to a more efficient implementation of the physical *scan* operator used by the transformed path expression.)

Clearly we can construct a database with a very large DataGuide, where the cost of exploring the DataGuide does outweigh the run-time benefit of compile-time expansion. For example, we generated a database whose DataGuide's size was close to the size of the database due to very unstructured data. The time to execute a sample regular path expression was 156 seconds, which was faster than the time required to expand the path expression (46 seconds) and then execute the expanded path expression (150 seconds).

4.2 Alternation Elimination

To test the effectiveness of replacing alternation with `union`, we ran the experiments reported in Tables 3 and 4. Table 3 shows our original (A) and rewritten (B) queries. We present the transformation of alternation in the `Where` clause into a `union` operator (with subsequent query rewrite), rather than disjunction, because the performance improvement is more significant. Table 4 shows execution times (in seconds) with a variety of indexes over the smaller library database.

The experiments in Table 4 show one situation in which it is beneficial to eliminate alternation. In Lore, *value indexes* may be used to quickly locate atomic objects with specific values and incoming labels (e.g., objects with incoming label `Title` and value "Armageddon"). We can then traverse backwards through the graph to match the path expression being evaluated. For details see [MW98]. In our example queries, if a value index exists for `Title` or `Keyword` objects but not both, then it would be extremely difficult for the optimizer to exploit just one index in the evaluation of query A. Query B, however, can take advantage of a single index when it exists. For example, with an index over `Keyword` objects the rewritten query ran about 24% faster than the original. The speedup using a `Title` index was less because typically there are many `Keyword` subobjects for a movie, but usually only one `Title`, thus requiring less search when a `Title` index is not present. If both indexes are present then the optimizer selects a more sophisticated query execution strategy (a combination of traversing down through the graph and upwards from the indexed values) for both queries, with significantly faster execution times as shown in the last two lines of Table 4.

In general terms, the advantage of transforming alternation into a `union` expression, and then into two queries connected via a `union`, is that even though some redundant path traversals may occur in the rewritten query, its two subqueries can be optimized independently and thus can use very different execution strategies. Transforming alternation into disjunction in our particular example has a less dramatic effect. However, the same general principle of separating execution strategies applies when the path expression operands to the alternation are longer.

One simple way to decide whether the rewritten query is likely to be advantageous is to send both queries to the physical query plan generator and compare estimated cost [MW98]. As future work we intend to investigate more efficient heuristics to make this decision.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

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

[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.

[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.

[FLS98] D. Florescu, A. Levy, and D. Suciu. Query optimization algorithm for semistructured data. Technical report, AT&T Laboratories, June 1998.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Orlando, Florida, February 1998.

[GGT96] G. Gardarin, J. Gruser, and Z. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 390–401, Bombay, India, 1996.

[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, 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.

[MW98] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Stanford University Database Group, August 1998. Document is available as `http://www-db.stanford.edu/pub/papers/qo.ps`.

[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.

[Suc98] D. Suciu. Semistructured data and XML, September 1998. Manuscript Draft.