An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities

Laura M. Haas* Donald Kossmann† Edward L. Wimmers‡ Jun Yang§
IBM Almaden Research Center
San Jose, CA 95120

Abstract

Much of the world's nonstandard data resides in specialized data sources. This data must often be queried together with data from other sources to give users the information they desire. Queries that can span several specialized sources with diverse search capabilities pose new challenges for query optimization. This paper describes the design and illustrates the use of a general purpose optimizer that can be taught about the capabilities of a new data source. Our optimizer produces plans of high quality even in the presence of nonstandard data, strange methods, and unusual query processing capabilities, and makes it easy to add new sources of standard or nonstandard data.

1 Introduction

Much of the world's nonstandard data resides in specialized data sources. For example, videos are typically found in video servers that know how to manage quality of service issues; chemical structures are found in specialized chemical structure "databases"; and text is found in a variety of information retrieval and file systems. Many of these sources have specialized query processing capabilities. Some videos are now indexed by scene changes; chemical structure databases support substructure and similarity search; and information retrieval systems support content search of various degrees of sophistication. Furthermore, this data, though in specialized sources, does not exist in isolation. It often must be combined with other data from other sources to give users the information they desire.

Database middleware systems offer users the ability to combine data from multiple sources in a single query. Several projects are currently working on middleware to bridge sources of nonstandard data types [SAD+94, LP95, PGMW95]. Queries that can span several specialized sources with diverse search capabilities pose new challenges for query optimization in these projects. With multiple sources, the possible ways to execute any given query are many. Special-purpose optimizers that each understand

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* *laura@almaden.ibm.com*
† Current address: University of Passau, 94030 Passau, Germany; *kossmann@db.fmi.uni-passau.de*
‡ *wimmers@almaden.ibm.com*
§ Current address: Stanford University, Stanford, CA 94305; *junyang@db.stanford.edu*

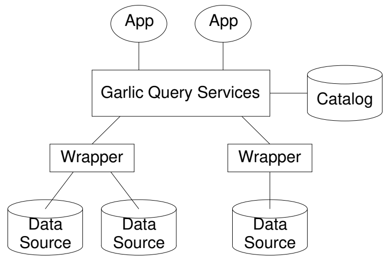

Figure 1: Garlic System Architecture

a particular set of data sources could be built, but due to the range of nonstandard data types and systems storing them, and the variety of combinations that users need, this is not a practical approach.

In the Garlic project [C+95], we are building a general purpose optimizer that can be taught about the capabilities of a new data source. We make it easy to add new sources of non-standard data and to exploit the special query processing abilities of sources. Our approach produces plans of high quality, while making it possible to include easily a broad range of data sources. In addition, we can both extend the system with new sources and evolve the descriptions of sources to capture further capabilities at any time.

This paper illustrates how our optimizer helps users relate together data of nonstandard types, and exploit the special query capabilities that may be associated with them. In the next section, we briefly discuss the middleware model we assume, and sketch our approach to optimization. A detailed description of our approach can be found in [KHWY96]. In Section 3, we give an example of how our optimizer would work for a query of chemical structure data. Section 4 summarizes the paper and includes a quick discussion of related work.

2 Optimization in a Middleware Architecture

The architecture of Figure 1 is common to many heterogeneous database systems, including TSIMMIS [PGMW95], DISCO [TRV96], Pegasus [SAD+94], DIOM [LP95], HERMES [ACPS96] and Garlic [C+95]. Data sources store data and provide functions to access and manipulate their data. These are pre-existing systems not to be disturbed by the middleware. A *wrapper* protects the data source from the middleware, and enables the middleware to use its internal protocols to access the source. The wrapper describes the data and capabilities of a source in a common data model. In Garlic, data is described using an object-oriented model based on the ODMG standard [Cat96, C+95]. Collections of objects are defined, and can serve as the targets of queries. Objects were chosen for their ability to model the full range of nonstandard data encountered. Wrappers provide methods to access the attributes of objects, and specialized search capabilities of the source are also encapsulated as methods. In this paper, we are concerned with middleware systems that promote "thin" wrappers, that is, systems in which the middleware does not mask the differences among sources, and instead, exploits the specialized capabilities of those sources.

The query services component of these systems must handle nonstandard data and query capabilities in planning and executing queries. Query services typically consist of a query language processor and an execution engine. The query processor obtains an execution plan for the query through some sequence of parsing, semantic checking, query rewrite, and query optimization, drawing on a system catalog to learn where data is stored, what wrapper it is associated with, its schema, any available statistics, and so on. The execution engine passes the sub-queries identified in the plan to the wrappers and assembles the final query result. A key feature of Garlic is that each wrapper provides a description of its source's query capabilities. This description is used by Garlic's query optimizer to create a set of feasible plans and to select "the best" for execution.

Garlic follows a traditional, dynamic programming approach to optimization [SAC+79]. Plans are trees of Plan Operators, or *POPs*, characterized by a fixed set of plan *properties*. These properties include *Cost*, *Tables*, *Columns*, and *Predicates*, where the latter three keep track of the collections and attributes accessed and the predicates applied by the plan, respectively. The enumerator builds plans for the query bottom-up in three phases, applying pruning to eliminate inefficient plans at every step. In the first phase, it creates plans to access individual collections used in the query. In the second phase, it iteratively combines plans to create join plans, considering all *bushy* join orders. Bushy plans are particularly efficient for distributed systems in many situations. Finally, the enumerator adds any POPs necessary to get complete query plans. The winning plan is chosen on the basis of a cost model that takes into account local processing costs, communication costs, and the costs to initiate a sub-query to a data source. The cost model should include the costs of expensive methods and predicates [HS93, CG96].

Garlic uses grammar-like *STrategy Alternative Rules* (*STARs*) as in [Loh88] as the input to enumeration. Each wrapper defines its own set of POPs, to describe the query processing capabilities it exports. The fixed set of properties used to describe POPs allows Garlic to handle plans that are (in part) composed of POPs whose specifications are unknown to Garlic because they are defined by wrappers. Every wrapper also has a set of STARs, which construct plans that can be handled by that wrapper, using the wrapper's POPs. Likewise, Garlic has STARs which construct plans using Garlic's POPs. The Garlic STARs use wrapper plans as building blocks, combining them to generate full plans for the query. The Garlic STARs require certain properties of the wrapper plans in order to combine them; if there are no plans with the necessary properties, other Garlic STARs are invoked to add Garlic POPs to existing plans until the correct properties are achieved. When a STAR is applied during enumeration, all properties of the resulting plans are computed.

We call the topmost non-terminal symbols of the grammar *roots*. While STARs and POPs are defined for every wrapper individually, Garlic defines a fixed set of roots with fixed interfaces, corresponding to the various language functions it supports. There are roots for *select, insert, delete*, and *update*. For example, an `AccessRoot` STAR models alternative ways to access a collection of objects from a data source. Wrappers may provide STARS for some or all of the roots. A wrapper must export at least one `AccessRoot` STAR if data in its data sources are to be accessible in queries. Since no data are currently stored in Garlic itself, there are no `AccessRoot` STARs defined for Garlic. Garlic, however, has several `JoinRoot` STARs that model the alternative ways to execute a join in Garlic's query engine. In addition, Garlic defines a `FinishRoot` STAR to complete plans, by adding POPs to enforce properties not yet taken care of by any wrapper or Garlic STARs. Further details of our approach to optimization can be found in [KHWY96].

Once the optimizer chooses a winning plan for the query, the plan is translated into an executable form. Garlic POPs are translated into operators that can be directly executed by the Garlic execution engine. Typically each Garlic POP is translated into a single executable operator. By contrast, an entire subtree of wrapper POPs is usually translated into a single query or API call to the wrapper's underlying data source. Wrappers are, however, free to translate POPs in whatever way is appropriate for their system.

3 Example: Optimization of Pharmaceutical Queries

In this section, we show how queries in Garlic are optimized, using as an example a pharmaceutical research application. In collaboration with chemists at the Almaden Research Center and a large pharmaceutical company, we are integrating a number of data sources including RDBMSs storing the results of biological assays, chemical structure databases, and text search engines containing paper abstracts and patents. Although many queries in this environment are quite complex and involve several data sources, we focus for brevity on queries to a single, nonstandard data source, a chemical structure database. We present POPs and STARs for this data source, and, for a sample query, show alternative plans that could be generated using those STARs in combination with Garlic's STARs. Many STARs, even for unusual data sources, have the same simple pattern, making them easy to write. Furthermore, we show that the differences in cost of alternative plans can be large, even when all data is in a single data source; hence it is important to have an optimizer that can enumerate all alternative plans. This is as true in an environment with a diversity of sources storing nonstandard data as in a standard relational environment.

3.1 STARs for a Chemical Structure Database

The chemical structure database maintains collections of molecules. This data source (and its wrapper) can handle two kinds of queries: *substructure* queries, which return the key (i.e., the *l_number* field) of all molecules that contain a certain substructure, say, "$CC(C)C$"[1], and *similarity* queries. Given a sample molecule, similarity queries compute a score in the range of [0,1] for every molecule, measuring how similar each is to the sample molecule; the query returns all molecules of a collection ordered by this score. Thus, the first kind of query corresponds to the application of a boolean predicate (i.e., filter) to a collection of molecules whereas the second kind of query corresponds to ranking molecules of a collection in the same way as is done for web pages in a WWW search engine or for images in an image processing system.

In addition to these two kinds of queries, the wrapper of the chemical structure database can handle method calls. Method calls can be made to fetch the value of an attribute of a molecule, to determine whether a specific molecule (specified by its *l_number*) contains a certain substructure, or to determine the similarity score of a specific molecule as compared to a sample molecule.

To model the ability to execute substructure queries, our chemical wrapper defines the STAR shown in Figure 2. The STAR is an `AccessRoot` STAR used in the first phase of plan enumeration for a Garlic *select* query (i.e., a read-only query). Like all `AccessRoot` STARs, it takes three parameters: (1) T, which specifies a collection of molecules used in the query; (2) C, the set of all expressions used in the various clauses of the query; and (3) P, the set of predicates found in the *where* clause of the query.

$$\texttt{AccessRoot}(T, C, P) = \forall p \in P : \textbf{M\_Select}(T, p)$$

Conditions: p is a *substructure* predicate

Figure 2: substructure `AccessRoot` STAR

The STAR generates a list of alternative plans to retrieve the molecules of collection T. Each plan consists of a single *M_Select* POP which models filtering the collection by a substructure predicate. The semantics of an *M_Select* POP are not known to the Garlic optimizer, but the wrapper sets the plan properties to inform the optimizer which parts of the query are handled by this plan. In this case, the *Column* property would be set to $\{l\_number\}$ specifying that only the key of the molecules is returned; the *Predicate* property would be set to $\{p\}$ specifying that the substructure predicate p has been applied, and of course, the wrapper's cost model would be consulted in order to compute

[1]All chemical formulas in this paper are presented in *SMILES* notation, which is emerging as a standard description language for chemical structures.

the estimated *cost* property of the *M_Select* plan; Garlic's optimizer uses the *cost* property in order to carry out pruning and to determine the winning plan. Ultimately, the *M_Select* POP of the winning plan would be translated by the wrapper at execution time into a sub-query to the chemical structure database.

The *M_Select* POP can only apply a single substructure predicate at a time. This is because the chemical structure database can only apply a single predicate at a time (this is a good example of a data source with both nonstandard data and query capabilities). For queries that have several substructure predicates, the STAR will enumerate a separate, alternative access plan for every individual predicate. Depending on the selectivity of the various predicates, and the expense of a method call to test substructure, these alternative access plans can have quite different costs. For queries that have no substructure predicates, the STAR will fail and return no plan (other STARs would be applicable for such queries). Furthermore, the *M_Select* POP does not compute scores for similarity search; this is because the chemical structure data source cannot handle substructure predicates and similarity search in a single query. Finally, the conditions on the STAR guarantee that the wrapper will only be given substructure predicates; this is because this data source cannot evaluate predicates such as *l_number* < "M38". These predicates would have to be evaluated by Garlic after retrieving the raw data from the source.

$$\texttt{AccessRoot}(T, C, P) = \forall e \in C : \mathbf{M\_Rank}(T, e)$$

Conditions: e is a *similarity* expression

Figure 3: Similarity `AccessRoot` STAR

The STAR of Figure 3 models that similarity searches can be carried out by the chemical structure database. Again, the STAR is an `AccessRoot` STAR, and it takes the same three parameters as the substructure `AccessRoot` STAR. Rather than generating alternative plans with an *M_Select* POP, this STAR generates alternative plans with an *M_Rank* POP. To model the execution of a similarity search, the properties of the *M_Rank* POP are set in the following way. The *Column* property is set to contain *l_number* and e, the scoring expression on which the similarity search is based. The *Predicate* property is set to $\emptyset$ as no predicates can be evaluated by the wrapper or data source during a similarity search. Unlike substructure queries, similarity searches return results ordered by score; accordingly the *M_Rank* POP sets the *Order* property to e DESC. Finally, of course, a special cost function would be invoked to compute the *cost* property of *M_Rank* plans. As for substructure predicates above, only one score expression can be evaluated as part of a similarity search at the data source.

Both STARs of the chemical wrapper are very simple; nevertheless, they are sufficient to construct good plans for queries over molecules, as will be shown in the next subsection. In general, most wrapper STARs are simple because wrapper STARs model "what" sub-queries can be handled by a wrapper and its data sources; wrapper STARs need not model "how" these sub-queries are actually executed. For example, the STARs of Figures 2 and 3 do not enumerate alternative plans using different indexes since access path selection is carried out autonomously by the chemical structure database system. Since Garlic STARs, like the rules for plan generation of any standard dbms, must specify how Garlic executes its portion of a plan, they are generally significantly more complex. However, Garlic's optimizer is completely general; in other words, Garlic STARs are written once (by the developers!), and no new Garlic STARs are needed to add a new data source, only the simpler wrapper STARs.

3.2 Example Query and Plans

In the following, we will show how wrapper STARs work together with Garlic's pre-defined STARs to generate alternative plans for a query. As discussed in the previous section, the chemical wrapper and its data source can do only one thing at a time: either apply a substructure predicate or run a similarity search. Garlic makes it possible to ask queries that cannot be evaluated natively by the data source. For example, Garlic can answer queries with both a substructure predicate and a similarity search, by issuing method calls or executing joins to combine partial query results returned from the chemical wrapper. To illustrate this point, we use a query that asks for poisonous molecules containing "*c1ccccc1*" (benzene), that are similar to a "*CC(C)C*" molecule (isobutane).

select m.l_number, m.similarTo("*CC(C)C*") as relevance
from PoisonousMolecules m
where m.containsSubStructure("*c1ccccc1*")
order by relevance desc

Figure 4 shows three alternative plans for this query. In the first phase of enumeration, the Garlic optimizer uses the STARs of the previous section to generate *M_Select* and *M_Rank* POPs. In the second and third phases, the Garlic STARs are fired to complete the plans. The POPs generated by the Garlic STARs are denoted by "G_" in Figure 4 and are fairly self-explanatory. The *G_Pushdown* POP submits a sub-query to a wrapper and receives results from the wrapper. *G_Fetch* fetches so-far unretrieved or uncomputed columns (e.g., scoring expressions) using method calls. *G_Filter* filters out rows, applying methods as needed to do so. *G_Join* is a logical join operator that combines two input streams and *G_Sort* orders the incoming stream of values.

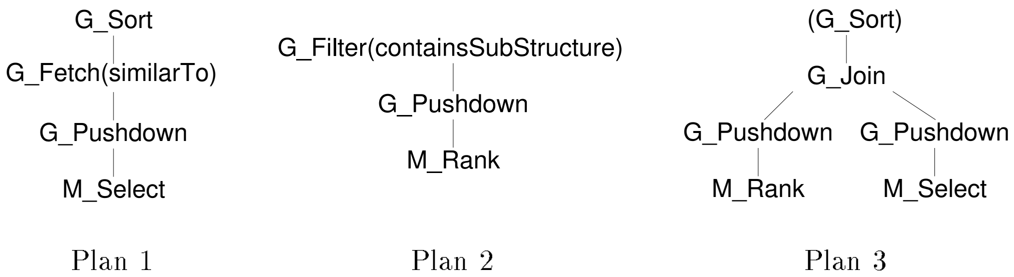

Figure 4: Alternative Query Evaluation Plans

Plan 1 of Figure 4 would be executed as follows: run the substructure query in the chemical structure source, then for each qualifying molecule call a method to compute its similarity score, and sort the resulting ⟨molecule, score⟩ pairs by score. Plan 2 carries out the similarity search in the chemical structure source, and then filters out molecules that contain a "*c1ccccc1*" structure, using *containsSubStructure* method calls to compute the truth value of the predicate. This second plan does not require a final sort in Garlic as the molecules are already in the right order as a result of the similarity search. Plan 3 executes both a similarity search and a substructure query in the molecule database and then does a self-join in Garlic to assemble the final query result; depending on the join method used to implement the G_Join POP final sorting of the molecules might be necessary. Plan 3 is generated by special Garlic self-join STARs that combine different access plans produced by wrapper STARs.

As each POP of each plan is added, its cost property would be computed. The cost model of the chemical wrapper would be consulted to estimate the cost of the *M_Select* and *M_Rank* sub-queries (i.e., to compute the *cost* property of these POPs), and Garlic's cost model would be consulted to estimate the cost of joins and sorting in Garlic, the overhead of method calls, and the communication costs. Clearly, the differences in cost between the three plans can be large so that it is important to enumerate and cost out all three of them. If, say, 10,000 poisonous molecules are stored in the chemical structure data source, Plan 2 would induce the overhead of 10,000 substructure method calls while executing the *G_Filter* in order to find the qualifying molecules composed of benzene. The overhead of method calls can be substantial in this environment because every method call requires sending a request to the data source and possibly setting up internal structures in the data source to evaluate the request. Depending on the number of benzene molecules (i.e., the selectivity of the substructure predicate) significantly fewer method calls are issued for Plan 1: if, say, 1,000 benzene molecules exist, then only 1,000 *similarTo* method calls are required as part of the *G_Fetch* operation of this plan. The reduction in the number of method calls comes, however, at the cost of sorting the 1,000 resulting molecules in Garlic. The use of method calls is completely avoided in Plan 3, but Plan 3 requires paying the price of an additional join in Garlic.

4 Conclusions

The optimizer described and illustrated in the previous sections has been implemented as part of the Garlic project. Further details of its implementation can be found in [KHWY96]. This optimizer is based on traditional, well-understood optimization technology, and can handle queries to data sources with standard and nonstandard data and search capabilities. To accomplish this, our optimizer extends the traditional approach by allowing strategy alternative rules and cost models to be defined separately for each wrapper. Because rules for wrappers are typically quite simple, and because rules for different wrappers are defined separately, the system is easily extensible, and can support a broad range of wrappers for data sources with diverse and specialized capabilities. Since the optimizer is cost-based and enumerates the entire space of feasible plans, it finds good plans even in the presence of nonstandard data, strange methods, and unusual query processing capabilities.

Only recently have other projects addressed the problem of optimization in this environment [FRV95, Qia96, PGH96, TRV97]. Most take the approach of decomposing, or rewriting, the query into wrapper-specific pieces, and heuristically pushing down maximal pieces to the wrapper, though of course the actual means of decomposition varies. An exception is the optimizer of DISCO, which also does a cost-based optimization [TRV97]. However, DISCO enumerates plans as though all wrappers could handle any query, then uses wrapper grammars to filter out impossible plans. The Garlic optimizer, by contrast, enumerates only valid plans.

We have chosen a Selinger-style, dynamic programming approach to optimization because of its proven efficiency in finding good plans. We have followed Lohman's STAR framework, because of the advantages it offers in terms of extensibility. We believe that our work extends these benefits into the heterogeneous environment of sources of nonstandard data in a simple and compelling way. New wrappers of nonstandard sources are typically up and running in Garlic in a matter of days, and initial performance results reported in [KHWY96] indicate that the dynamic programming approach to optimization will work as well for this environment as it has worked in the past for standard relational data.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[FRV95] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proc. CoopIS*, 1995.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[KHWY96] D. Kossmann, L. Haas, E. Wimmers, and J. Yang. I can do that! using wrapper input for query opitmization in heterogeneous middleware systems, October 1996. Submitted for publication.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[LP95] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *Proc. ACM Int'l. Conf. on Information and Knowledge Management*, 1995.

[PGH96] Y. Papkonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami, Fl, USA, December 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[Qia96] X. Qian. Query folding. In *Proc. IEEE Conf. on Data Engineering*, pages 48–55, New Orleans, LA, USA, 1996.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SAD+94] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, chapter 32. ACM Press (Addison-Wesley publishers), Reading, MA, USA, 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. ICDCS*, 1996.

[TRV97] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling acccess to distributed heterogeneous databases in DISCO. *ACM Trans. Comp. Syst.*, 1997. To appear.