Optimizing Queries across Diverse Data Sources

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

Abstract

Businesses today need to interrelate data stored in diverse systems with differing capabilities, ideally via a single high-level query interface. We present the design of a query optimizer for Garlic [C+95], a middleware system designed to integrate data from a broad range of data sources with very different query capabilities. Garlic's optimizer extends the rule-based approach of [Loh88] to work in a heterogeneous environment, by defining generic rules for the middleware and using wrapper-provided rules to encapsulate the capabilities of each data source. This approach offers great advantages in terms of plan quality, extensibility to new sources, incremental implementation of rules for new sources, and the ability to express the capabilities of a diverse set of sources. We describe the design and implementation of this optimizer, and illustrate its actions through an example.

1 Introduction

Businesses today rely on data stored in diverse systems with differing capabilities. Some data are in traditional database systems with a powerful query language and efficient indices for parametric data. Others are in spreadsheets and file systems with limited query capabilities, or in legacy application systems which provide specialized ways to access and manipulate data. The emergence of protocols such as CORBA, OLE DB and Java/JDBC makes it easier to access this range of sources, while database middleware systems or mediators [Wie93] offer the possibility of interrelating their data via a single high-level query interface. The first generation of commercial middleware systems has gained rapid acceptance in the marketplace. However, these products typically connect only a limited set of data sources, predominantly relational, and generally model all data sources as relational systems. This simplifies the middleware considerably, as it can assume that all the data sources have similar capabilities. The price of this simplification is that any specialized search or data manipulation capabilities of the underlying systems cannot be exploited when they are accessed through the middleware. Thus this first generation of middleware is not extensible to the arbitrary systems which may exist in a given business.

Several projects are addressing the problem of middleware for increasingly diverse systems [Day83, S+94, PGMW95, TRV96, LRO96]. Many of the data sources these systems integrate have limited or specialized query processing capabilities. Queries in this environment vary widely in performance depending on how and where their operations are executed. One key challenge for these systems is thus to develop a general-purpose *query optimizer* which can use information about the capabilities of a new data source to produce correct plans that efficiently answer queries ranging over data in multiple sources, with differing query capabilities. This paper takes up that challenge.

In this paper we present the design of a cost-based optimizer for heterogeneous middleware systems. We have implemented our approach in Garlic [C+95], a middleware system designed to integrate data from a broad range of data sources, with very different query capabilities. Our approach extends Lohman's [Loh88] grammar-like rules to work in a heterogeneous environment. Data sources are connected to the middleware engine via *wrappers*. The optimizer is given a set of rules that capture the engine's query execution strategies. Among these are several generic rules, which produce source-specific plans using matching wrapper-provided rules that encapsulate the capabilities of a particular data source. A normal dynamic-programming enumerator fires rules to generate all possible alternative execution plans for a query.

We have pursued and implemented our approach because it has several crucial advantages. First, since our optimizer is an extension of a standard optimizer we get all the benefits of advances in optimizer technology, as well as the benefits of considering the entire search space, leading to high quality, efficient plans. We believe ours is the first solution based on traditional dynamic-programming techniques. Second, the system is extensible. Regardless of their data model and query processing capabilities, new wrappers can be integrated without affecting other wrappers or the middleware. Third, wrappers can evolve gracefully.

*Current address: University of Passau, 94030 Passau, Germany
†Current address: Stanford University, Stanford, CA 94305

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

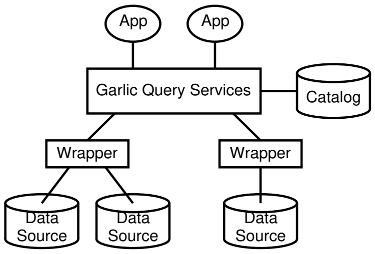

Figure 1: Garlic System Architecture

At any time, it is possible to refine or add wrapper rules to improve the performance of queries over the wrapper's data sources. Finally, this approach is extremely flexible, making it possible to integrate wrappers of strange data sources with unusual query processing capabilities.

The remainder of this paper is structured as follows: Section 2 describes the Garlic architecture. Section 3 presents the Garlic query optimizer and its built-in rules. Section 4 shows how easy it is to model the query behavior of diverse sources. Section 5 shows by example how the Garlic optimizer uses Garlic and wrapper rules to optimize a query across very different sources. Section 6 discusses related work, and Section 7 concludes the paper.

2 The Garlic System

Figure 1 shows the architecture of Garlic [C+95]. The architecture is typical of many heterogeneous database systems, e.g., [Day83, PGMW95, TRV96]. At the bottom are data sources, which store, access and manipulate data. Above every data source is a wrapper. A wrapper hides the details of the data source's interface and enables access to the data source using Garlic's internal protocols. The wrapper describes the data stored in the source using Garlic's data model, an object-oriented model based on the ODMG standard [Cat96, C+95]. Data in the source are viewed as objects, and Garlic refers to these objects using an OID it manufactures based on the source, the object's type, and a unique *key* determined by the wrapper. This OID allows Garlic to apply methods on objects; from the OID, Garlic can determine the appropriate wrapper, and the wrapper can locate the necessary data and apply the method. Wrappers provide methods to get the value of each attribute of an object, and to encapsulate any specialized search capabilities of the source. (These methods are typically implemented as commands in the native language or programming interface of the underlying source.) The wrapper also defines object *collections* which are the targets of queries in Garlic.

The wrapper further provides a description of its query processing capabilities in the form of a set of rules (encapsulated as *planning methods* [RS97]). Different sources may vary greatly in their query processing capabilities, and thus will provide different rules. A wrapper does not have to reflect the full query functionality of its data sources. However, in order for the data in that data source to be accessible through queries, some minimum functionality must be provided, *i.e.*, at least one access rule. We will discuss wrapper rules in Section 4.

A system catalog records the global schema. When a new data source is added to a Garlic system, it is associated with a wrapper. This association, as well as the data source's local schema and any available statistics for its data, is recorded in the catalog as part of the registration process for a data source. The catalog also contains information such as view definitions and information about the system configuration needed as input to the cost model during query optimization.

At the heart of Garlic are its query services, which play the same role as a mediator in the architecture of other systems [Wie93]. Garlic's query services have two major components: a query language processor, and a distributed query execution engine. The query language processor takes a query as input and obtains an execution plan for the query through parsing, semantic checking, query rewrite, and query optimization (as in Starburst [H+89]). The job of the optimizer is to construct and select an "optimal" plan for a given query, based on a cost model. Traditional query optimizers build plans based on detailed, built-in knowledge of the full set of execution strategies available and their costs. This is true even in distributed systems; the optimizer must know the capabilities and costs for each remote data source to decide which operations to execute at a source and which at the query site [FJK96]. Garlic, however, must be able to find good plans without built-in knowledge of data sources' capabilities and costs; how it accomplishes this is the subject of this paper.

Once the plan has been determined by the optimizer, its execution is coordinated by Garlic's query execution engine, which passes subqueries to the wrappers and assembles the final query result. Garlic's execution engine is a powerful system able to perform joins, apply predicates, invoke methods, sort, aggregate, and so on. This allows Garlic to compensate for functionality not present in the data sources or not reflected by their wrappers, and to execute itself those operations it can do more efficiently.

3 Query Optimization in Garlic

To optimize a query, Garlic uses a set of *STrategy Alternative Rules*, or *STARs* [Loh88], which construct plans that can be handled by Garlic's query engine. Garlic's enumerator fires appropriate STARs, following a dynamic programming model, to build plans for the query bottom-up. Garlic differs from [Loh88] in that some of Garlic's STARs are *generic*. These STARs are fired during enumeration when a piece of work is found that can or must be done by a wrapper. Generic STARs consult the appropriate wrapper to build their piece of the plan. From the resulting set of complete plans for the query, the optimizer selects the winning plan based on cost. This plan will then be translated into an executable (or interpretable) format.

Property	Description
Tables	set of tables that have been accessed and joined
Columns	set of columns of the output of the plan
Preds	set of predicates that have been applied in the plan
Source	where the output is produced; i.e., the *id* of a data source or Garlic's execution engine
Mat	TRUE if the output of the plan is materialized; FALSE otherwise
Order	a sort expr. if the tuples of the output are ordered; NIL otherwise
Cost	estimated cost of the plan
Card	estimated number of tuples of the output of the plan

Figure 2: Garlic Plan Properties

3.1 Plans in Garlic

Plans in Garlic are trees of operators, or *POPs* (Plan OPerators). Each POP works on one or more inputs, and produces some output (usually a stream of tuples). The input to a POP may include one or more streams of tuples. In a plan, these are produced by other POPs. Garlic's POPs include operators for join, sort, filter (to apply predicates), fetch (to retrieve data from a data source), temp (to make a temporary collection) and scan (to retrieve locally stored data). Garlic also provides a generic POP, called *PushDown*, which encapsulates work to be done at a data source.

Plans are characterized by a set of plan *properties*. Properties are a common way to track the work that is done in a plan [GD87, Loh88, M^+96]. It is particularly important to characterize plans with a fixed set of properties in Garlic, because Garlic plans are (in part) composed of generic *PushDown* POPs. The actual work being done by these POPs depends on the wrapper where the work takes place and the query, and is not understood by Garlic or any other wrapper in the system. However, the properties provide sufficient information about what is done to allow Garlic to properly incorporate the *PushDown* POP in a plan.

We characterize plans and their output by the eight properties described in Table 2. The properties of one POP are typically a function of the properties of its input POP(s), if any. Properties are computed as the POPs are created, by STARs. The properties assigned to a plan are the properties of the topmost POP of the plan. Most of these properties are equivalent to those used by optimizers of traditional database systems. An exception is the *Source* property. It is used to record where the output stream comes from (Garlic or a particular data source); the *Source* property is comparable to the *Site* property used by R^* [Loh88].

For example, Figure 3 shows one possible plan for executing the query "select m.Body from Inbox m, Classes c where m.Subject = c.Course and c.Prof = 'Aho' ", assuming *Inbox* is defined by a simple mail wrapper that only answers queries of the form "select OID from Inbox", and that *Classes* comes from a DB2 database. The leaves of the plan are both *PushDown* POPs, but with quite different properties. A *Fetch* POP retrieves from Mail the attributes

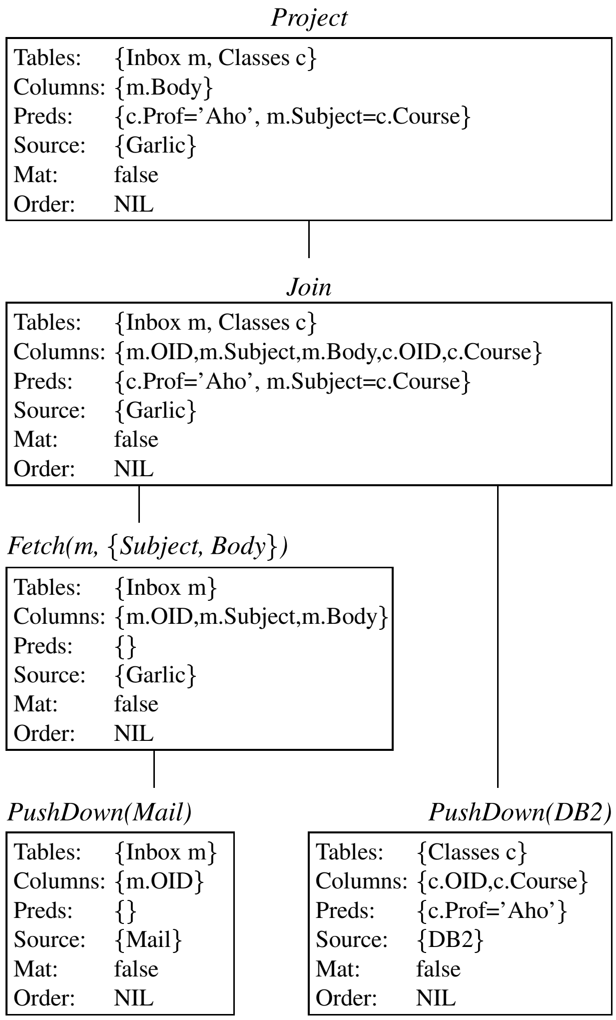

Figure 3: One Possible Query Plan for:
`SELECT m.Body FROM Inbox m,Classes c WHERE m.Subject=c.Course AND c.Prof='Aho'`

Subject and *Body* for each OID returned by the first *PushDown* POP, compensating for the inability of Mail to return these values directly[1]. Hence, *Fetch*'s properties include these two additional columns. Note that it has *Source* = 'Garlic', reflecting the fact that it will be executed by Garlic. The *Join* POP's properties reflect the two tables of its input streams, the union of the columns from those streams, and the predicate applied by its (second) input, as well as the join predicate. The final *Project* POP ensures that only the *Body* column is returned as specified in the query.

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. A *PushDown* POP is usually translated into a query or set of API calls to

[1] This is possible because (1) the assignment (and retrieval during query processing) of Garlic OIDs allows Garlic to go back to the data source to retrieve missing information and (2) wrappers must provide "get" methods for any attribute they define.

the wrapper's underlying data source. Wrappers are, however, free to translate the *PushDown* POPs in whatever way is appropriate for their system.

3.2 Using STARs to Produce Plans

Garlic's STARs are closely based on the work of [Loh88]; in fact, we have implemented the Garlic optimizer as an extension of the DB2 CS [G+93] version of STARs. We begin this section with a review of this work, and then focus on how we have extended STARs to meet Garlic's needs.

STARs can be seen as the production rules of a grammar that generates plans. We call the topmost non-terminal symbols of the grammar *roots*. A STAR determines how POPs can be combined in a plan. A simple STAR may build only a single POP, by invoking its constructor. The constructor allocates space for the POP, initializes various fields, and calls the property function to compute the properties of the new POP (including *Cardinality* and *Cost*).

Of course, few STARs are that simple. Most include a condition function; if the condition is true, then the STAR builds its plan, otherwise, no plan is built. Also, a single STAR may construct multiple POPs, and multiple plans. Multiple POPs are built by calling the POPs' constructors in sequence. Multiple plans result when the STAR is instantiated with a set parameter, and creates a plan for each element of the set—in this case, the condition (if any) is evaluated for every element of the set separately. Finally, STARs can also invoke other STARs. Thus, STARs are rules of the following form (where f_i is the name of a STAR or a POP):

$$\texttt{STAR}(\text{params}) ::= \forall e \in \textit{set}:\ f_1(f_2(\ldots), f_3(\ldots), \text{other args}) \quad [\text{if condition(args)}] \tag{1}$$

Note that when a STAR is instantiated, all properties of all the resulting plans are computed automatically, as the various POP constructors are called.

For example, the following STAR can be used to retrieve columns that are needed by some other STAR, but which have not yet been retrieved from the relevant wrapper.

$$\texttt{FetchCols}(T, C, \text{Plan}) ::= \textbf{Fetch}(T, C', \text{Plan}) \quad \text{if } C' \neq \emptyset,\ \ C' = C - \text{Plan}.\textit{Columns} \tag{2}$$

This STAR constructs a *Fetch* POP, if there are columns needed that are not already present in the properties of the input plan. It builds at most one plan, depending on the value of the condition function. In the following example, multiple plans may be returned (depending on the cardinality of the set of input plans), and multiple POPs are unconditionally constructed.

$$\texttt{DamStream}(\{\text{Plan}\}) ::= \forall p \in \{\text{Plan}\}:\ \textbf{Scan}(\textbf{Temp}(p)) \tag{3}$$

`DamStream` is called when an intermediate result must be stored. It is given a set of plans which produce that result, and adds *Scan* and *Temp* POPs to each. Examples of more complex STARs for a single-source DBMS can be found in [Loh88]. We will look at some of Garlic's more complex STARs in Section 3.5 below.

Garlic defines a fixed set of roots with fixed interfaces, corresponding to the different language functions it supports. There are roots for *select, group-by, insert, delete*, and *update*, which are invoked by the plan enumerator depending on the kind of query. In this paper we focus on *select-project-join* queries. These queries involve three kinds of roots: `AccessRoot` (STARs for single-collection accesses), `JoinRoot` (for joins) and `FinishRoot` (for ensuring that the plan is complete).

To allow the Garlic optimizer to plan queries when data comes from sources with differing query capabilities, Garlic includes several generic STARs. These STARs construct the generic *PushDown* POP described above. We will prefix the names of these generic STARs with *Repo* to remind us that they represent work that will take place in a data source (repository). There is a generic STAR corresponding to each root STAR (except `FinishRoot`, which is a purely Garlic function). Thus, there is a `RepoAccess` STAR and a `RepoJoin` STAR. When these STARs are instantiated, they invoke rules the wrapper may have provided, then use the results to build a *PushDown* POP and compute its properties. If there is no appropriate wrapper STAR, they simply return no plan. In many cases, Garlic will find other ways of accomplishing the same function.

We illustrate this using Garlic's `RepoAccess` STAR, shown in Figure 4. This STAR invokes the `plan_access` rule, if any, defined by the wrapper of the data source that contains the collection to be accessed. That rule returns a list of zero or more "wrapper plans". These are simply data structures, uninterpreted by Garlic, that provide information the wrapper needs to execute the access if Garlic requests it later. Also returned are the properties for each wrapper plan; these will typically be (a subset of) the properties requested when the STAR was instantiated. The Source property will be computed by the *ds* function provided by Garlic. The Garlic `RepoAccess` STAR uses these properties to set the properties of the *PushDown* POPs that it creates.

For purposes of this paper, we assume that wrappers construct their plans using STARs. Note, however, that since Garlic does not interpret the wrapper plans (only their properties), wrappers are actually free to construct their plans however they wish, as long as the interface to Garlic is STAR-like. Interested readers may consult [RS97] for the wrapper's perspective on this process. STARs provide a useful means of capturing the wrappers' query capabilities, regardless of implementation. Thus, when we need to characterize the work done in a plan by a wrapper, we will use "wrapper STARs" and "wrapper POPs" to do so. We will use wrapper STAR names that start with *plan_* and are all lower case in order to distinguish wrapper STARs from Garlic STARs.

$$\texttt{RepoAccess}(T, C, P) ::= \forall p \in \texttt{plan\_access}(T, C, P) : \textbf{PushDown}(p)$$

Condition: `plan_access`(T, C, P) has been defined by the wrapper of the data source that stores T.
Functions: none

Figure 4: Garlic's `RepoAccess` STAR
T a table; C columns of T used in the query; P restrictions on T defined in the query

3.3 Plan Enumeration and Dynamic Programming

Garlic's cost-based [S+79] optimizer enumerates plans by invoking the appropriate root STARs of Section 3.2. Plans for *select* queries are enumerated bottom up in three phases. In the first phase, the enumerator applies the `AccessRoot` STAR to every collection used in the query. Since at this time Garlic stores no data, `AccessRoot` basically serves to call `RepoAccess`.

In the second phase, the enumerator applies the `JoinRoot` STAR, which invokes the `RepoJoin` STAR as well as various other join STARs, each of which represents one Garlic join method. It applies the `JoinRoot` STAR iteratively, passing it two plans and a join predicate each time. Initially, each plan is one of those enumerated in phase one for a single table access. When all possible two-way join plans have been examined, the enumerator invokes the `JoinRoot` STAR to combine single table access plans with two-way join plans to create the three-way joins, and so on, until plans which join all the collections of the query have been created. The enumerator considers all *bushy* join orders. Since Garlic is a distributed system, bushy plans are particularly efficient in many situations.

Garlic's optimizer employs dynamic programming in order to find the best plan with reasonable effort [S+79]. In every step of plan enumeration, Garlic's optimizer applies pruning; that is, the optimizer does not use plan *A* as a building block for other, more complex plans if *A* has higher cost than another plan and *A*'s properties are a subset of that plan's. Only plans whose properties are included in a cheaper plan's are pruned; for example, if Plan 1 has higher cost than Plan 2, but the *Source* of Plan 1 is Garlic (i.e., *Source* property is "Garlic") and the *Source* of Plan 2 is some data source, then Plan 1 may not be pruned because it might be a building block for a winning plan that executes most operators of the query in Garlic's query engine.

In the third phase, the enumerator applies Garlic's `FinishRoot` STAR to get a final query plan that includes all projections, selections and orderings specified in the query and not so far achieved. When this rule completes, all remaining plans will have the same properties, and the least cost plan is chosen for execution.

3.4 Costing Plans

In Garlic, the cost of a plan is the sum of local processing costs, communications costs, and the costs to initiate subqueries and methods. The communication costs and the costs to initiate subqueries and methods are estimated by Garlic functions using constants stored in Garlic's catalog. The local processing costs of the operators of Garlic's query engine are estimated by a cost model provided by Garlic. This model includes CPU and I/O costs, and models fairly closely the actions of the Garlic execution engine. The local processing costs of wrappers and their data sources, however, must be estimated by cost models that are defined for each wrapper individually because there is no universal, generic cost model that is valid for all wrappers and all data sources. We are working on a framework to help wrapper writers create these models. Today, they must be hand-written and hand-calibrated.

An important parameter of any kind of cost model is the *Cardinality* of input and output collections. As with other properties, *Cardinality* is computed after every application of a STAR. Cardinality depends on logical operations of the query, so wrapper writers need not implement functions that compute this property. However, they must provide ways to gather statistics on the cardinality of the stored collections, and on values of their attributes.

3.5 More Complex Garlic STARs

We now describe the Garlic join STARs. Garlic's `JoinRoot` STAR, which is applied in the second phase of plan enumeration, is defined in Figure 5. It specifies that joins can be evaluated in Garlic in one of three ways: (1) by pushing the join down to a data source, (2) via a nested-loop join in Garlic, or (3) by means of a *bind join* (defined below). For each of these three join methods, Garlic defines a separate STAR which is called by Garlic's `JoinRoot` STAR in order to produce the corresponding join plan.

The simplest of the actual join STARs is `RepoJoin` (Figure 6). This STAR produces plans in which the join is done by a data source if that source's wrapper has a `plan_join` STAR and if both the outer and inner of the join are available at the data source. Like the `RepoAccess` STAR, Garlic's `RepoJoin` STAR creates a generic *PushDown* POP to track the properties of the wrapper plan.

Garlic's `NestedLoopJoin` STAR is shown in Figure 7. Using a plan for the outer (T_1) and a plan for the inner (T_2) as building blocks, it constructs a new plan with a *NLJ* POP at the root and a *Scan* POP to iteratively read the inner, which is materialized via a *Temp* POP. The third parameter of *NLJ* is the set of join predicates. For the *NLJ* POP to function, all the attributes needed to evaluate those predicates must have been retrieved. To ensure this, we use a variant of the `FetchCols` STAR defined in Section 3.2,

```
JoinRoot(T1, T2, P) ::= RepoJoin(T1, T2, P)
JoinRoot(T1, T2, P) ::= NestedLoopJoin(T1, T2, P)
JoinRoot(T1, T2, P) ::= BindJoin(T1, T2, P)
```

Conditions: none
Functions: none

Figure 5: Garlic's JoinRoot STARs

RepoJoin(T_1, T_2, P) ::=
$\forall p \in$ **plan_join**(T_1, T_2, P) : **PushDown**(p)

C.: T_1.*Source* = T_2.*Source*; T_1.*Source* $\neq$ 'Garlic';
plan_join(T_1,T_2,P) defined by the wrapper of T_1.*Source*
F.: none

Figure 6: Garlic's RepoJoin STAR

NestedLoopJoin(T_1, T_2, P) ::=
NLJ(FetchCols(T_1, *NeedAttr*(T_1, P)),
Scan(**Temp**(FetchCols(T_2, *NeedAttr*(T_2, P)))),
P)

C.: none
F.: *NeedAttr*(*Plan*, *Preds*) computes the attributes of collections of *Plan* that are needed to compute the predicates in *Preds*.

Figure 7: Garlic's NestedLoopJoin STAR

BindJoin(T_1, T_2, P) ::= $\forall p \in$ **plan_bind**(T_2, P) :
Bind(FetchCols(T_1, *NeedAttr*(T_1, P)), **PushDown**(p))

C.: T_2.*Source* $\neq$ 'Garlic'
plan_bind(T, P) defined by the wrapper of T_2.*Source*
F.: *NeedAttr* as in Figure 7.

Figure 8: Garlic's BindJoin STAR
T_1, T_2 plans for outer and inner; P potential join predicates

which returns the Plan without an attached *Fetch* POP if no columns are missing. The ability to invoke other STARs to enforce certain properties is powerful; it allows Garlic to detect discrepancies between what a plan provides and what is needed, and to compensate. Thus, Garlic can provide powerful queries against even very limited data sources.

The third Garlic join rule, the one for bind joins, is shown in Figure 8. A *bind join* is a nested loop join in which Garlic passes intermediate results (e.g., values for the join predicate) from the outer objects to the wrapper for the inner, which uses these results to filter the data it returns. If the intermediate results are small and indexes are available at data sources, bindings can significantly reduce the amount of work done by a data source. Furthermore, bindings can reduce communication cost in the same way that a semi-join does in distributed databases. On the other hand, bindings result in poor plans if intermediate results are large: high processing costs at Garlic's query engine, the wrapper and the data source, plus high communication costs to ship intermediate results. Therefore, binding plans should be enumerated and costs evaluated in addition to the other two alternatives. The BindJoin STAR checks that the wrapper for the data source which produces the inner plan accepts bindings (provides a plan_bind STAR), and if so, asks the wrapper to re-plan the inner with the additional bind predicates. For each resulting wrapper plan, the BindJoin STAR produces a new *PushDown* POP as the inner. Using our variant of FetchCols, BindJoin ensures that all the attribute values needed from the outer for the join predicates are retrieved, so that the *Bind* POP can pass them to the inner.

3.6 Discussion

We have implemented the STAR framework, and STARs and cost models for wrappers of several data sources, including DB2, Oracle, ObjectStore, an image processing system called QBIC [N+93], two Lotus Notes databases, and two Web sources. Our implementation extends the DB2 CS V2 optimizer with the STARs and POPs described above. During plan enumeration, the RepoAccess STAR is invoked once for each collection in the query, and invokes the appropriate wrapper's plan_access STAR. All of Garlic's join STARs are applied in every step of the second phase of plan enumeration to ensure that all possibilities are considered. However, the conditions on the RepoJoin and BindJoin rules ensure that they will return plans only when such plans are possible.

In the current system, all STARs and POPs are implemented in C++. An alternative would be to implement STARs as declarative rules and interpret the STARs as proposed in [LFL88]. This might simplify the implementation of STARs, especially for wrapper writers; hard-coding all STARs in C++, however, provides significantly better performance during plan enumeration.

Our approach to optimization has several key advantages. It is a simple extension of traditional optimizer technology, allowing us to both enumerate a full set of plans and to take advantage of any and all advances in optimization and execution strategies. Since we enumerate all possible plans, we are guaranteed to find the optimal plan as defined by our cost model; as with all optimizers, however, this may not be the actual best execution plan if the cost model used by the optimizer is not sufficiently accurate. The extensions we make are isolated and few in number, consisting of one generic *PushDown* POP and a few generic STARs.

As a further consequence of this design, our system is extremely flexible. Wrappers for new data sources can be added at any time without considering the capabilities of other wrappers, and without changing the optimizer code. Because Garlic does not have to understand the wrapper plans, relying only on a fixed set of properties to describe them, a wide range of data sources can be wrapped. These sources may differ in data model and vary widely in query processing abilities, yet no special properties have been added to deal with heterogeneity.

Finally, STARs are a powerful construct for a distributed system. In addition to standard relational function, Garlic's STARs can handle approximate search, replicated collections, and gateways [K+96]. An example involving approximate search is given in Section 4.

4 Modeling Wrapper Query Capabilities Using STARs

In addition to making optimization simple for Garlic, the STAR framework makes it easy to describe wrapper query capabilities, and allows wrappers to start simply, and evolve over time. While Garlic STARs may be complicated, due in part to their use of other STARs to enforce needed properties, wrapper STARs tend to be simple. Indeed, we have found no need for wrapper STARs to invoke other STARs, or even to build multiple wrapper POPs. In this section, we demonstrate the power and simplicity of the STAR framework for heterogeneous systems, by means of an example involving three very different data sources. In the next section, we extend our example to show how the Garlic optimizer would optimize a query involving these three sources.

Consider a university with a relational database storing basic information on each course offered, course descriptions in a special text store, and an on-line complaint mechanism that sends mail to an ombudsman. These three sources (relational, text, and mail) are integrated using Garlic. In the following, we provide relevant details of these wrappers and define STARs for them.

The mail wrapper exports a *Complaints* collection of objects of type *Message*. Messages each have *Sender, Date, Body* and *Subject* attributes. The wrapper provides only the ability to iterate through a collection, retrieving the OIDs. To model this ability, it defines the simple `plan_access` STAR shown in Figure 9. Like every `plan_access` STAR, this STAR takes as parameters the identifier of a collection (T), a set of attributes (C), and a set of predicates (P) that are used in the query. Regardless of C and P, this STAR always returns one plan consisting of a single *Quantifier* POP. The *Quantifier* POP models the execution of the query "select OID from T" in the data source that stores T. The values of the properties (except *cost* and *cardinality*) of the *Quantifier* POP are defined in Table 1; the `RepoAccess` STAR would get these values from the wrapper plan to create its *PushDown* POP. Query plans generated using this STAR are executed as follows: the OIDs of all messages of a collection are passed from the wrapper to Garlic's execution engine, which uses method calls to the wrapper to get the attributes of the messages.

The simple STAR of Figure 9 could be used as a starting point for wrappers of many different sources. (There is nothing Mail-specific about it.) This STAR guarantees that any query that accesses data from one of a wrapper's

`plan_access`(T, C, P) = **Quantifier**$(T, ds(T))$

C.: none
F.: $ds(T)$ returns the *id* of the data source that stores T.

Figure 9: Mail Wrapper STAR

`plan_access`(T, C, P) = **R_Scan**$(T, C, P, ds(T))$

C.: none
F.: $ds(T)$ returns the id of the rel. data source that stores T.

`plan_bind`$(T, C, P, plan)$ =
R_Scan$(T, C, P \cup plan.Preds, ds(T))$

C.: none
F.: *ds* as defined above.

`plan_join`(T_1, T_2, P) = **R_Join**(T_1, T_2, P)

C.: T_1*.Source* = T_2*.Source*
F.: none

Figure 10: Relational Wrapper STARs

sources can be processed, but it does not model a wrapper's query processing capability, and therefore, plans generated by this STAR often show poor performance. Initially a wrapper writer might define only this STAR to integrate a source quickly; later (s)he could add more powerful STARs to improve performance. For example, we could initially use this STAR to integrate the relational database, and then, once we had made the relational data accessible, replace it with the STARs of Figure 10 to exploit the relational engine's query processing power, improving performance.

The relational wrapper exports a *Classes* collection. *Class* objects have attributes *Course, Professor,* etc. The relational data source supports the usual relational operations, and the wrapper provides STARs for access, bind and join. These STARs are shown in Figure 10. They construct a set of POPs which model the relational source's operations. Their properties are given in Table 1. `plan_access` generates an *R_Scan* POP which models the execution of a single-table query, aggressively applying all predicates and retrieving all necessary columns. `plan_bind` also builds an *R_Scan* POP, adding the binding predicates to the set. Finally, `plan_join` constructs an *R_Join* POP, which models the relational source's ability to join two tables, again applying all predicates and fetching all columns.

The text wrapper exports a single collection, *Descrs*, which contains objects of type *Blurb*, with attributes *Name* and *Description*. The text data source supports single-collection queries with methods of the form *contains(string)* or *is_about(string)* modeling its search capabilities. *contains* returns a boolean value, depending on whether the document it is applied to contains the words in the string. *is_about(string)* returns a rank between 0 and

	Table (t)	*Column* (c)	*Preds* (p)	*Order* (o)	*Mat* (m)	*Source* (s)
Quantifier(T, S)	T	*oid*	$\emptyset$	NIL	FALSE	S
T_Rank(T, C, e, P, S)	T	$C\cup$ *score*(e)	P	*score*(e)	FALSE	S
T_Scan(T, C, P, S)	T	C	P	NIL	FALSE	S
R_Scan(T, C, P, S)	T	C	P	NIL	FALSE	S
R_Join(T_1, T_2, P)	$T_1.t \cup T_2.t$	$T_1.c \cup T_2.c$	$T_1.p \cup T_2.p \cup P$	NIL	FALSE	$T_1.s$

Table 1: Properties (except cost and cardinality) of POPs used in Wrapper STARs

T a collection; S an id of a data source; e an *is_about* predicate; C a set of attributes; P a set of preds; T_1, T_2 plans

`plan_access`$(T, C, P) =$ **T_Scan**$(T, C, P_t, ds(T))$

C.: $P_t \subseteq P$ are all predicates of the form *contains(string)* or *Name = string*.
F.: $ds(T)$ returns the *id* of the text data source that stores T.

`plan_access`$(T, C, P) =$
$\forall e \in C$: **T_Rank**$(T, C, e, P_t, ds(T))$

C.: e is an *is_about* expression on T. $P_t \subseteq P$ as above.
F.: *ds* as above.

Figure 11: Text Wrapper STARs

1 indicating how closely the document matches the terms in the argument string. STARs defining this wrapper's plans are found in Figure 11. The POPs for these STARs are also described in Table 1. Note that this wrapper provides two `plan_access` rules: one, which produces a *T_Scan* POP, simply scans the documents, returning whatever attributes are asked for, and applying any "contains" or other String predicates, and the other, which produces the *T_Rank* POP, returns the results in order of rank computed as a result of an *is_about* method in the *order by* clause.

¿From these three examples, we can see that the basic query power of wrappers and data sources with vastly different querying abilities can be modeled easily with a handful of simple, single-POP STARs. There are two reasons why wrapper STARs can be so simple. First, Garlic provides a powerful query engine which can make up for missing query function in the wrappers. Second, wrapper STARs model "what" can be executed by a wrapper, not "how". For example, the relational wrapper exported a simple `plan_join` STAR to model that joins can be executed by its data sources; it did not need to enumerate alternative plans with different join methods because plans with an *R_Join* POP are translated into a multi-table (SQL) query, and the optimizer of the relational data source automatically determines the most efficient join methods. Precise modeling of join methods may be required in the wrapper's cost model in order to estimate the cost of join processing in the data source, but it is not required in the wrapper's STARs.

These examples also demonstrate three further advantages of our approach. First, we defined a simple minimal STAR that might be the first STAR a wrapper would export. This makes it easy to get a wrapper up and running. Second, wrapper writers can add STARs or alternatives for an existing STAR at any time, to expose more wrapper query functionality to Garlic. This makes it easy to modify and evolve wrappers. Third, each wrapper's STARs were defined independently of the others', and without affecting Garlic STARs or Garlic's query services, making it easy to add new wrappers to the system. Modeling power, low "entry-cost" for writing wrappers, evolvability, and extensibility are key advantages of our approach.

5 Optimizing a Query

To see how the whole framework works, we now describe how a query against the sources of Section 4 would be processed by the Garlic optimizer using Garlic's built-in STARs (Section 3) and the wrapper STARs defined above. Suppose that the ombudsman has just received a complaint about an Ancient Studies course. She remembers receiving a number of complaints about courses concerning the ancient world recently, and wants to see what faculty are involved. She poses the following query:

```
SELECT     C.Course, C.Prof
FROM       Classes C, Descrs D, Complaints P
WHERE      D.Name = C.Course AND
           C.Course = P.Subject
ORDER BY D.is_about("ancient world, Greece, Rome")
```

In phase one of optimization, Garlic's `AccessRoot` STAR is invoked once for each collection of the query. In each case, it invokes the appropriate wrapper's `plan_access` STAR, and then creates a *PushDown* POP. This results in four plans, shown in Figure 12, one from each of the Mail and Relational wrappers, and two from the Text Wrapper. Their properties will be those of the wrapper POPs in Table 1.

In phase two, Garlic's `JoinRoot` STAR is fired, first to make all possible two-collection joins, and then to look at all three-collection plans. This entails four calls to `JoinRoot` to join *Classes* and *Descrs* (one with each of the plans for *Descrs* as the outer, and two with *Classes* as the outer, using the different plans for *Descrs* as the inners), four more for *Descrs* and *Complaints*, and two for joining *Classes* and *Complaints*. Each time it is called,

P1: PushDown(R_Scan(Classes,{Course,Prof}, $\emptyset$, RDB))
P2: PushDown(Quantifier(Complaints, Mail))
P3: PushDown(T_Scan(Descrs, {Name,score}, $\emptyset$, Text))
P4: PushDown(T_Rank(Descrs, {Name,score}, $\emptyset$, Text))

Figure 12: Plans from Phase 1 of Optimization

P5: NLJ(P1, Scan(Temp(P3)), {Course = Name})
P6: NLJ(P4, Scan(Temp(P1)), {Course = Name})
P7: NLJ(P3,Scan(Temp(Fetch(P2,Subject))),{Subject=Name})
P8: NLJ(P4,Scan(Temp(Fetch(P2,Subject))),{Subject=Name})
P9: Bind(Fetch(P2,Subject), P1 + {Course = Name})

Figure 13: Two-Way Join Plans Surviving Pruning

P10: NLJ(P5, Scan(Temp(Fetch(P2,Subject))),
{Subject=Course})
P11: NLJ(P4, Scan(Temp(P9)), {Name = Course})

Figure 14: Three-Way Join Plans Surviving Pruning

`JoinRoot` instantiates all three Garlic join rules. For this query, `RepoJoin` never returns any plans, as no two collections are co-located. `NestedLoopJoin` always returns a plan, as Garlic can always perform the join, so ten nested loop plans are returned. Since only the relational wrapper defines a `plan_bind` STAR, `BindJoin` returns a plan only when *Classes* is the inner. This occurs in three plans, so in total, thirteen join plans are considered in this phase. However, only five plans survive pruning (Figure 13). The others are eliminated because they have the same properties as another plan, and cost at least as much.

Note that each plan of Figure 13 builds on the plans of Figure 12. For example, plan P5 combines plans P1 and P3, storing the results of P3, and adding the join operator with a scan of the new collection. Plan P8 similarly builds on plans P4 and P2, but discovers that it needs to add a fetch of *subject* before making the temporary collection, in order to apply the join predicate during join processing.

Plans P7 and P8 demonstrate the benefits of extending well-known optimizer technology. Both plans apply a join predicate that did not appear in the query, but could be deduced from it by taking the transitive closure of the predicates [G^+93]. These plans required no new rules, nor did the new, generic Garlic rules disturb them; the existing optimizer computed transitive closures of predicates, and the Garlic optimizer therefore (automatically) does so.

In the next step of phase two, these two-way join plans will be combined with the single-table access plans from phase one to generate the three-way joins. In this phase, fourteen plans are created, but only two survive pruning, one ordered by *is_about* (P11) and one not ordered (P10). These two plans, shown in Figure 14, are the input to phase three. In this phase, the `FinishRoot` STAR is invoked to complete both plans. P11 is already complete, so it is returned as is, but `FinishRoot` adds a *Sort* POP to P10 to complete it. As both plans now have the same properties, a winner is chosen on the basis of cost.

6 Related Work

Despite its importance, there is little related work on optimization and decomposition of queries across data sources with different query capabilities. Some systems use query rewrite rules to decompose a query, but have no cost model to evaluate alternative plans (e.g., [FRV95]). [CS93] uses rewrite rules to generate alternative versions of a query involving foreign tables and functions. Each version can then be optimized, and the least cost plan overall is chosen. Most work on cost-based query optimization in heterogeneous systems is limited to specific classes of data sources [DKS92, GST96]. The works most closely related to ours are [TRV96] (DISCO) and [PGH96]. These two approaches also use grammars to describe the capabilities of wrappers; however, the types of grammars used and how they are used are significantly different.

DISCO addresses problems beyond the scope of Garlic, with an emphasis on operating when not all data sources are available. DISCO uses a wrapper grammar to *match* queries. The DISCO optimizer enumerates query plans as if wrappers could handle any kind of query, then uses the wrapper grammar to parse each plan to determine whether it can be handled by the wrapper. Thus, DISCO enumerates *all* plans, including many invalid ones. The Garlic optimizer, by contrast, constructs only valid plans, and it is quicker to construct a plan using STARs than to parse a plan using a grammar.

[PGH96] proposes a set of algorithms that decompose a query based on a novel *relational query description language* that describes the capabilities of wrappers. Their algorithms push down as much work as possible to wrappers to minimize the amount of processing in the middleware system's query engine. However, this work gives no guidance on how to execute the remaining query pieces in the middleware, or how to choose between alternative plans.

Recently, other ways to describe capabilities of heterogeneous wrappers or data sources have been proposed. In [LRO96], *capability records* are used to describe which bindings can be passed to a source. However, the *capability record* mechanism is not powerful enough to describe the capabilities of, say, Garlic's relational or image wrappers. In other work, *views* are used to describe which queries can be handled by a wrapper/data source; e.g., [Qia96, LRU96]. While flexible, decomposing a query using views requires solving the query subsumption problem. Thus, these approaches are typically limited to simple conjunctive queries and cannot easily be extended to handle ordering, grouping, or aggregate functions.

7 Conclusion

In this paper, we presented the design of a query optimizer for heterogeneous middleware systems designed to integrate data sources with different data models and query processing capabilities. A query optimizer is a critical component of any such middleware system, because differences in cost between alternative plans for executing a query can easily be several orders of magnitude, and there are generally many possible plans. Our optimizer is based on dynamic programming and Lohman's *STrategy Alternative Rules,* or *STAR*s. We have extended Lohman's approach to encompass generic and wrapper STARs, and implemented this in the Garlic middleware system. Garlic uses STARs to construct its query execution plans, in which a generic *PushDown* POP represents work done by a data source. Garlic's generic STARs construct *PushDown* POPs and invoke wrapper-provided STARs to construct the wrapper portion of the plan. We illustrated our approach with both Garlic and wrapper STARs, and described how they would be used to optimize a query. In a small set of experiments [K+96], we have further shown the importance of optimization in this environment, and how alternative wrapper STARs impact query processing in Garlic.

The advantages of our approach lie in its extensibility and evolvability, the expressiveness of the powerful STAR syntax, the simplicity of wrapper STARs, and the fact that it can be implemented as an extension of an existing optimizer, leading to high quality plans. The approach is extensible, as new wrappers and their STARs can be integrated without affecting other wrappers or Garlic's query engine. The STAR syntax is powerful, as it enables wrapper writers to precisely model the capabilities of wrappers even for very unusual data sources. It is typically easy to define STARs because STARs simply model "what" kind of queries can be handled by a wrapper rather than specifying precisely "how" these queries are executed by the data sources. The approach is efficient, as it employs well-known optimization techniques such as dynamic programming with pruning to find good plans with reasonable effort.

In the future, we want to continue to integrate and experiment with new kinds of data sources in order to get more general insight into the design tradeoffs for wrapper STARs. We are considering wrappers for a digital library product, and for OLE automation servers. We are also examining whether we can develop cost models for broad classes of data sources, so that modeling the cost of wrapper plans can be simplified for the wrapper writer.

8 Acknowledgments

We thank Guy Lohman, Yannis Papakonstantinou and Anthony Tomasic for their helpful comments, and our Garlic teammates for their support and assistance.

References

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. *In IEEE RIDE Workshop*, Taipeh, 1995.

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

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. *In VLDB Conf.*, Dublin, 1993.

[Day83] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. *In VLDB Conf.*, Florence, 1983.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. *In VLDB Conf.*, Vancouver, 1992.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *In ACM SIGMOD Conf.*, Montreal, 1996.

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

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. *In ACM SIGMOD Conf.*, San Francisco, 1987.

[G+93] P. Gassner et al. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16(3), 1993.

[GST96] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. *In VLDB Conf.*, Bombay, 1996.

[H+89] L. Haas et al. Extensible query processing in starburst. *In ACM SIGMOD Conf.*, Portland, 1989.

[K+96] D. Kossmann et al. I can do that! using wrapper input for query optimization in heterogeneous middleware systems. Technical report, IBM Almaden, 1996.

[LFL88] M. Lee, J. Freytag, and G. Lohman. Implementing an interpreter for functional rules in a query optimizer. *In VLDB Conf.*, Los Angeles, 1988.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. *In ACM SIGMOD Conf.*, Chicago, 1988.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. *In VLDB Conf.*, Bombay, 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external query processors. *In ACM PODS Conf.*, Montreal, 1996.

[M+96] W. McKenna et al. EROC: a toolkit for building NEATO query optimizers. *In VLDB Conf.*, Bombay, 1996.

[N+93] W. Niblack et al. The QBIC project: Querying images by content using color, texture and shap. *In SPIE*, San Jose, 1993.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. *In IEEE PDIS Conf.*, Miami, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. *In IEEE ICDE Conf.*, Taipeh, 1995.

[Qia96] X. Qian. Query folding. *In IEEE ICDE Conf.*, New Orleans, 1996.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. *In VLDB Conf.*, Athens, 1997.

[S+79] P. Selinger et al. Access path selection in a relational database management system. *In ACM SIGMOD Conf.*, Boston, 1979.

[S+94] M.-C. Shan et al. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, chapter 32. ACM Press, Reading, 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling acccess to distributed heterogeneous databases in DISCO, 1996. Submitted for publication.

[Wie93] G. Wiederhold. Intelligent integration of information. *In ACM SIGMOD Conf.*, Washington, DC, 1993.