Chapter 9
Multidatabase Query Processing

In the previous three chapters, we have considered query processing in tighly-coupled homogeneous distributed database systems. As we discussed in Chapter 1, these systems are logically integrated and provide a single image of the database, even though they are physically distributed. In this chapter, we concentrate on query processing in multidatabase systems that provide interoperability among a set of DBMSs. This is only one part of the more general *interoperability* problem. Distributed applications pose major requirements regarding the databases they access, in particular, the ability to access legacy data as well as newly developed databases. Thus, providing integrated access to multiple, distributed databases and other heterogeneous data sources has become a topic of increasing interest and focus.

Many of the distributed query processing and optimization techniques carry over to multidatabase systems, but there are important differences. Recall from Chapter 6 that we characterized distributed query processing in four steps: query decomposition, data localization, global optimization, and local optimization. The nature of multidatabase systems requires slightly different steps and different techniques. The component DBMSs may be autonomous and have different database languages and query processing capabilities. Thus, a multi-DBMS layer (see Figure 1.17) is necessary to communicate with component DBMSs in an effective way, and this requires additional query processing steps (Figure 9.1). Furthermore, there may be many component DBMSs, each of which may exhibit different behavior, thereby posing new requirements for more adaptive query processing techniques.

This chapter is organized as follows. In Section 9.1 we introduce in more detail the main issues in multidatabase query processing. Assuming the mediator/wrapper architecture, we describe the multidatabase query processing architecture in Section 9.2. Section 9.3 describes the techniques for rewriting queries using multidatabase views. Section 9.4 describes multidatabase query optimization and execution, in particular, heterogeneous cost modeling, heterogeneous query optimization, and adaptive query processing. Section 9.5 describes query translation and execution at the wrappers, in particular, the techniques for translating queries for execution by the component DBMSs and for generating and managing wrappers.

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_9, © Springer Science+Business Media, LLC 2011

9.1 Issues in Multidatabase Query Processing

Query processing in a multidatabase system is more complex than in a distributed DBMS for the following reasons [Sheth and Larson, 1990]:

1. The computing capabilities of the component DBMSs may be different, which prevents uniform treatment of queries across multiple DBMSs. For example, some DBMSs may be able to support complex SQL queries with join and aggregation while some others cannot. Thus the multidatabase query processor should consider the various DBMS capabilities.
2. Similarly, the cost of processing queries may be different on different DBMSs, and the local optimization capability of each DBMS may be quite different. This increases the complexity of the cost functions that need to be evaluated.
3. The data models and languages of the component DBMSs may be quite different, for instance, relational, object-oriented, XML, etc. This creates difficulties in translating multidatabase queries to component DBMS and in integrating heterogeneous results.
4. Since a multidatabase system enables access to very different DBMSs that may have different performance and behavior, distributed query processing techniques need to adapt to these variations.

The autonomy of the component DBMSs poses problems. DBMS autonomy can be defined along three main dimensions: communication, design and execution [Lu et al., 1993]. Communication autonomy means that a component DBMS communicates with others at its own discretion,and, in particular, it may terminate its services at any time. This requires query processing techniques that are tolerant to system unavailability. The question is how the system answers queries when a component system is either unavailable from the beginning or shuts down in the middle of query execution. Design autonomy may restrict the availability and accuracy of cost information that is needed for query optimization. The difficulty of determining local cost functions is an important issue. The execution autonomy of multidatabase systems makes it difficult to apply some of the query optimization strategies we discussed in previous chapters. For example, semijoin-based optimization of distributed joins may be difficult if the source and target relations reside in different component DBMSs, since, in this case, the semijoin execution of a join translates into three queries: one to retrieve the join attribute values of the target relation and to ship it to the source relation's DBMS, the second to perform the join at the source relation, and the third to perform the join at the target relation's DBMS. The problem arises because communication with component DBMSs occurs at a high level of the DBMS API.

In addition to these difficulties, the architecture of a distributed multidatabase system poses certain challenges. The architecture depicted in Figure 1.17 points to an additional complexity. In distributed DBMSs, query processors have to deal only with data distribution across multiple sites. In a distributed multidatabase environment, on the other hand, data are distributed not only across sites but also across multiple

databases, each managed by an autonomous DBMS. Thus, while there are two parties that cooperate in the processing of queries in a distributed DBMS (the control site and local sites), the number of parties increases to three in the case of a distributed multi-DBMS: the multi-DBMS layer at the control site (i.e., the mediator) receives the global query, the multi-DBMS layers at the sites (i.e., the wrappers) participate in processing the query, and the component DBMSs ultimately optimize and execute the query.

9.2 Multidatabase Query Processing Architecture

Most of the work on multidatabase query processing has been done in the context of the mediator/wrapper architecture (see Figure 1.18). In this architecture, each component database has an associated wrapper that exports information about the source schema, data and query processing capabilities. A mediator centralizes the information provided by the the wrappers in a unified view of all available data (stored in a global data dictionary) and performs query processing using the wrappers to access the component DBMSs. The data model used by the mediator can be relational, object-oriented or even semi-structured (based on XML). In this chapter, for consistency with the previous chapters on distributed query processing, we continue to use the relational model, which is quite sufficient to explain the multidatabase query processing techniques.

The mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of component databases with "similar" data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different components, ranging from full-fledged relational DBMSs to simple files.

Assuming the mediator/wrapper architecture, we can now discuss the various layers involved in query processing in distributed multidatabase systems as shown in Figure 9.1. As before, we assume the input is a query on global relations expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution and heterogeneity are hidden. Three main layers are involved in multidatabase query processing. This layering is similar to that of query processing in homogeneous distributed DBMSs (see Figure 6.3). However, since there is no fragmentation, there is no need for the data localization layer.

The first two layers map the input query into an optimized distributed query execution plan (QEP). They perform the functions of query rewriting, query optimization and some query execution. The first two layers are performed by the mediator and use meta-information stored in the global directory (global schema, allocation and capability schema). Query rewriting transforms the input query into a query on local relations, using the global schema. Recall from Chapter 4 that there are two main

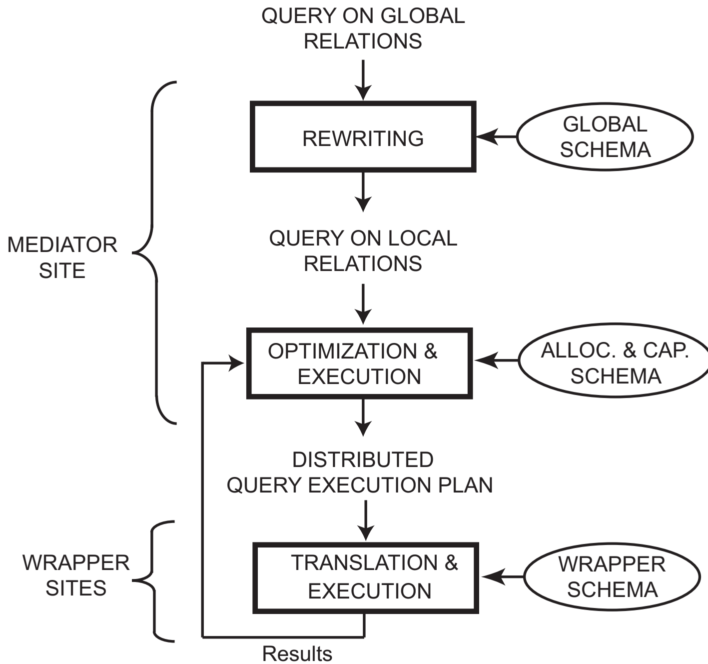

Fig. 9.1 Generic Layering Scheme for Multidatabase Query Processing

approaches for database integration: global-as-view (GAV) and local-as-view (LAV). Thus, the global schema provides the view definitions (i.e., mappings between the global relations and the local relations stored in the component databases) and the query is rewritten using the views.

Rewriting can be done at the relational calculus or algebra levels. In this chapter, we will use a generalized form of relational calculus called Datalog [Ullman, 1988] which is well suited for such rewriting. Thus, there is an additional step of calculus to algebra translation that is similar to the decomposition step in homogeneous distributed DBMSs.

The second layer performs query optimization and (some) execution by considering the allocation of the local relations and the different query processing capabilities of the component DBMSs exported by the wrappers. The allocation and capability schema used by this layer may also contain heterogeneous cost information. The distributed QEP produced by this layer groups within subqueries the operations that can be performed by the component DBMSs and wrappers. Similar to distributed DBMSs, query optimization can be static or dynamic. However, the lack of homogeneity in multidatabase systems (e.g., some component DBMSs may have unexpectedly long delays in answering) make dynamic query optimization more critical. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the next layer. This is illustrated by the arrow showing results coming from the next layer. Finally, this layer integrates the results coming from the

different wrappers to provide a unified answer to the user's query. This requires the capability of executing some operations on data coming from the wrappers. Since the wrappers may provide very limited execution capabilities, e.g., in the case of very simple component DBMSs, the mediator must provide the full execution capabilities to support the mediator interface.

The third layer performs *query translation and execution* using the wrappers. Then it returns the results to the mediator that can perform result integration from different wrappers and subsequent execution. Each wrapper maintains a *wrapper schema* that includes the local export schema (see Chapter 4) and mapping information to facilitate the translation of the input subquery (a subset of the QEP) expressed in a common language into the language of the component DBMS. After the subquery is translated, it is executed by the component DBMS and the local result is translated back to the common format.

The wrapper schema contains information describing how mappings from/to participating local schemas and global schema can be performed. It enables conversions between components of the database in different ways. For example, if the global schema represents temperatures in Fahrenheit degrees, but a participating database uses Celsius degrees, the wrapper schema must contain a conversion formula to provide the proper presentation to the global user and the local databases. If the conversion is across types and simple formulas cannot perform the translation, complete mapping tables could be used in the wrapper schema.

9.3 Query Rewriting Using Views

Query rewriting reformulates the input query expressed on global relations into a query on local relations. It uses the global schema, which describes in terms of views the correspondences between the global relations and the local relations. Thus, the query must be rewritten using views. The techniques for query rewriting differ in major ways depending on the database integration approach that is used, i.e., global-as-view (GAV) or local-as-view (LAV). In particular, the techniques for LAV (and its extension GLAV) are much more involved [Halevy, 2001]. Most of the work on query rewriting using views has been done using Datalog [Ullman, 1988], which is a logic-based database language. Datalog is more concise than relational calculus and thus more convenient for describing complex query rewriting algorithms. In this section, we first introduce Datalog terminology. Then, we describe the main techniques and algorithms for query rewriting in the GAV and LAV approaches.

9.3.1 Datalog Terminology

Datalog can be viewed as an in-line version of domain relational calculus. Let us first define *conjunctive queries*, i.e., select-project-join queries, which are the basis for

more complex queries. A conjuntive query in Datalog is expressed as a rule of the form:

$$Q(T) : -R_1(T_1), \ldots, R_n(T_n)$$

The atom $Q(T)$ is the *head* of the query and denotes the result relation. The atoms $R_1(T_1), \ldots, R_n(T_n)$ are the *subgoals* in the body of the query and denote database relations. Q and $R_1, \ldots, R_n$ are predicate names and correspond to relation names. $T, T_1, \ldots, T_n$ refer to the relation tuples and contain variables or constants. The variables are similar to domain variables in domain relational calculus. Thus, the use of the same variable name in multiple predicates expresses equijoin predicates. Constants correspond to equality predicates. More complex comparison predicates (e.g., using comparators such as $\neq$, $\leq$ and $<$) must be expressed as other subgoals. We consider queries which are *safe*, i.e., those where each variable in the head also appears in the body. Disjunctive queries can also be expressed in Datalog using unions, by having several conjuntive queries with the same head predicate.

Example 9.1. Let us consider relations EMP(ENO, ENAME, TITLE, CITY) and ASG(ENO, PNO, DUR) assuming that ENO is the primary key of EMP and (ENO, PNO) is the primary key of ASG. Consider the following SQL query:

```
SELECT ENO, TITLE, PNO
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND    TITLE = "Programmer" OR DUR = 24
```

The corresponding query in Datalog can be expressed as:

$$\begin{aligned} Q(\text{ENO}, \text{TITLE}, \text{PNO}) : -\ & \text{EMP}(\text{ENO}, \text{ENAME}, \text{"Programmer"}, \text{CITY}), \\ & \text{ASG}(\text{ENO}, \text{PNO}, \text{DUR}) \\ Q(\text{ENO}, \text{TITLE}, \text{PNO}) : -\ & \text{EMP}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}), \\ & \text{ASG}(\text{ENO}, \text{PNO}, 24) \end{aligned}$$

9.3.2 Rewriting in GAV

In the GAV approach, the global schema is expressed in terms of the data sources and each global relation is defined as a view over the local relations. This is similar to the global schema definition in tightly-integrated distributed DBMS. In particular, the local relations (i.e., relations in a component DBMS) can correspond to fragments. However, since the local databases pre-exist and are autonomous, it may happen that tuples in a global relation do not exist in local relations or that a tuple in a global relation appears in different local relations. Thus, the properties of completeness and disjointness of fragmentation cannot be guaranteed. The lack of completeness may yield incomplete answers to queries. The lack of disjointness may yield duplicate

results that may still be useful information and may not need to be eliminated. Similar to queries, view definitions can use Datalog notation.

Example 9.2. Let us consider the local relations EMP1(ENO, ENAME, TITLE, CITY), EMP2(ENO, ENAME, TITLE, CITY) and ASG1(ENO, PNO, DUR). The global relations EMP(ENO, ENAME, CITY) and ASG(ENO, PNO, TITLE, DUR) can be simply defined with the following Datalog rules:

$$\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}) :- \text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) \quad (r_1)$$
$$\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}) :- \text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) \quad (r_2)$$
$$\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR}) :- \text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}), \text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}) \quad (r_3)$$
$$\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR}) :- \text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}), \text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}) \quad (r_4)$$

♦

Rewriting a query expressed on the global schema into an equivalent query on the local relations is relatively simple and similar to data localization in tightly-integrated distributed DBMS (see Section 7.2). The rewriting technique using views is called *unfolding* [Ullman, 1997], and it replaces each global relation invoked in the query with its corresponding view. This is done by applying the view definition rules to the query and producing a union of conjunctive queries, one for each rule application. Since a global relation may be defined by several rules (see Example 9.2), unfolding can generate redundant queries that need to be eliminated.

Example 9.3. Let us consider the global schema in Example 9.2 and the following query Q that asks for assignment information about the employees living in "Paris":

$$Q(e, p) :- \text{EMP}(e, \text{ENAME}, \text{"Paris"}), \text{ASG}(e, p, \text{TITLE}, \text{DUR}).$$

Unfolding Q produces Q' as follows:

$$Q'(e, p) :- \text{EMP1}(e, \text{ENAME}, \text{TITLE}, \text{"Paris"}), \text{ASG1}(e, p, \text{DUR}). \quad (q_1)$$
$$Q'(e, p) :- \text{EMP2}(e, \text{ENAME}, \text{TITLE}, \text{"Paris"}), \text{ASG1}(e, p, \text{DUR}). \quad (q_2)$$

Q' is the union of two conjunctive queries labeled as q_1 and q_2. q_1 is obtained by applying rule r_3 or both rules r_1 and r_3. In the latter case, the query obtained is redundant with respect to that obtained with r_3 only. Similarly, q_2 is obtained by applying rule r_4 or both rules r_2 and r_4. ♦

Although the basic technique is simple, rewriting in GAV becomes difficult when local databases have limited access patterns [Calì and Calvanese, 2002]. This is the case for databases accessed over the web where relations can be only accessed using certain binding patterns for their attributes. In this case, simply substituting the global

relations with their views is not sufficient, and query rewriting requires the use of recursive Datalog queries.

9.3.3 Rewriting in LAV

In the LAV approach, the global schema is expressed independent of the local databases and each local relation is defined as a view over the global relations. This enables considerable flexibility for defining local relations.

Example 9.4. To facilitate comparison with GAV, we develop an example that is symmetric to Example 9.2 with EMP(ENO, ENAME, CITY) and ASG(ENO, PNO, TITLE, DUR) as global relations. In the LAV approach, the local relations EMP1(ENO, ENAME, TITLE, CITY), EMP2(ENO, ENAME, TITLE, CITY) and ASG1(ENO, PNO, DUR) can be defined with the following Datalog rules:

$$\begin{aligned} \text{EMP1(ENO, ENAME, TITLE, CITY)} :- &\text{EMP(ENO, ENAME, CITY)}, && (r_1) \\ &\text{ASG(ENO, PNO, TITLE, DUR)} \\ \text{EMP2(ENO, ENAME, TITLE, CITY)} :- &\text{EMP(ENO, ENAME, CITY)}, && (r_2) \\ &\text{ASG(ENO, PNO, TITLE, DUR)} \\ \text{ASG1(ENO, PNO, DUR)} :- &\text{ASG(ENO, PNO, TITLE, DUR)} && (r_3) \end{aligned}$$

Rewriting a query expressed on the global schema into an equivalent query on the views describing the local relations is difficult for three reasons. First, unlike in the GAV approach, there is no direct correspondence between the terms used in the global schema, (e.g., EMP, ENAME) and those used in the views (e.g., EMP1, EMP2, ENAME). Finding the correspondences requires comparison with each view. Second, there may be many more views than global relations, thus making view comparison time consuming. Third, view definitions may contain complex predicates to reflect the specific contents of the local relations, e.g., view EMP3 containing only programmers. Thus, it is not always possible to find an equivalent rewriting of the query. In this case, the best that can be done is to find a *maximally-contained* query, i.e., a query that produces the maximum subset of the answer [Halevy, 2001]. For instance, EMP3 could only return a subset of all employees, those who are programmers.

Rewriting queries using views has received much attention because of its relevance to both logical and physical data integration problems. In the context of physical integration (i.e., data warehousing), using materialized views may be much more efficient than accessing base relations. However, the problem of finding a rewriting using views is NP-complete in the number of views and the number of subgoals in the query [Levy et al., 1995]. Thus, algorithms for rewriting a query using views essentially try to reduce the numbers of rewritings that need to be considered. Three

main algorithms have been proposed for this purpose: the bucket algorithm [Levy et al., 1996b], the inverse rule algorithm [Duschka and Genesereth, 1997], and the MinCon algorithm [Pottinger and Levy, 2000]. The bucket algorithm and the inverse rule algorithm have similar limitations that are addressed by the MinCon algorithm.

The bucket algorithm considers each predicate of the query independently to select only the views that are relevant to that predicate. Given a query Q, the algorithm proceeds in two steps. In the first step, it builds a bucket b for each subgoal q of Q that is not a comparison predicate and inserts in b the heads of the views that are relevant to answer q. To determine whether a view V should be in b, there must be a mapping that unifies q with one subgoal v in V.

For instance, consider query Q in Example 9.3 and the views in Example 9.4. The following mapping unifies the subgoal EMP(e, ENAME, "Paris") of Q with the subgoal EMP(ENO, ENAME, CITY) in view EMP1:

$$e \rightarrow \text{ENO}, \text{“Paris”} \rightarrow \text{CITY}$$

In the second step, for each view V of the Cartesian product of the non-empty buckets (i.e., some subset of the buckets), the algorithm produces a conjuntive query and checks whether it is contained in Q. If it is, the conjuntive query is kept as it represents one way to anwer part of Q from V. Thus, the rewritten query is a union of conjunctive queries.

Example 9.5. Let us consider query Q in Example 9.3 and the views in Example 9.4. In the first step, the bucket algorithm creates two buckets, one for each subgoal of Q. Let us denote by b_1 the bucket for the subgoal EMP(e, ENAME, "Paris") and by b_2 the bucket for the subgoal ASG(e, p, TITLE, DUR). Since the algorithm inserts only the view heads in a bucket, there may be variables in a view head that are not in the unifying mapping. Such variables are simply primed. We obtain the following buckets:

$$\begin{aligned} b_1 = \{&\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}', \text{CITY}),\\ &\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}', \text{CITY})\}\\ b_2 = \{&\text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}')\} \end{aligned}$$

In the second step, the algorithm combines the elements from the buckets, which produces a union of two conjuntive queries:

$$Q'(e,p) :- \text{EMP1}(e, \text{ENAME}, \text{TITLE}, \text{“Paris”}), \text{ASG1}(e,p,\text{DUR}) \qquad (q_1)$$
$$Q'(e,p) :- \text{EMP2}(e, \text{ENAME}, \text{TITLE}, \text{“Paris”}), \text{ASG1}(e,p,\text{DUR}) \qquad (q_2)$$

The main advantage of the bucket algorithm is that, by considering the predicates in the query, it can significantly reduce the number of rewritings that need to be considered. However, considering the predicates in the query in isolation may yield the addition of a view in a bucket that is irrelevant when considering the join with

other views. Furthermore, the second step of the algorithm may still generate a large number of rewritings as a result of the Cartesian product of the buckets.

Example 9.6. Let us consider query Q in Example 9.3 and the views in Example 9.4 with the addition of the following view that gives the projects for which there are employees who live in Paris.

$$\text{PROJ1(PNO)} : -\text{EMP1(ENO, ENAME, "Paris")}, \text{ASG(ENO, PNO, TITLE, DUR)} \quad (r_4)$$

Now, the following mapping unifies the subgoal ASG(e, p, TITLE, DUR) of Q with the subgoal ASG(ENO, PNO, TITLE, DUR) in view PROJ1:

$$p \rightarrow PNAME$$

Thus, in the first step of the bucket algorithm, PROJ1 is added to bucket b_2. However, PROJ1 cannot be useful in a rewriting of Q since the variable ENAME is not in the head of PROJ1 and thus makes it impossible to join PROJ1 on the variable e of Q. This can be discovered only in the second step when building the conjunctive queries. ♦

The MinCon algorithm addresses the limitations of the bucket algorithm (and the inverse rule algorithm) by considering the query globally and considering how each predicate in the query interacts with the views. It proceeds in two steps like the bucket algorithm. The first step starts similar to that of the bucket algorithm, selecting the views that contain subgoals corresponding to subgoals of query Q. However, upon finding a mapping that unifies a subgoal q of Q with a subgoal v in view V, it considers the join predicates in Q and finds the minimum set of additional subgoals of Q that must be mapped to subgoals in V. This set of subgoals of Q is captured by a *MinCon description* (MCD) associated with V. The second step of the algorithm produces a rewritten query by combining the different MCDs. In this second step, unlike in the bucket algorithm, it is not necessary to check that the proposed rewritings are contained in the query because the way the MCDs are created guarantees that the resulting rewritings will be contained in the original query.

Applied to Example 9.6, the algorithm would create 3 MCDs: two for the views EMP1 and EMP2 containing the subgoal EMP of Q and one for ASG1 containing the subgoal ASG. However, the algorithm cannot create an MCD for PROJ1 because it cannot apply the join predicate in Q. Thus, the algorithm would produce the rewritten query Q' of Example 9.5. Compared with the bucket algorithm, the second step of the MinCon algorithm is much more efficient since it performs fewer combinations of MCDs than buckets.

9.4 Query Optimization and Execution

The three main problems of query optimization in multidatabase systems are heterogeneous cost modeling, heterogeneous query optimization (to deal with different capabilities of component DBMSs), and adaptive query processing (to deal with strong variations in the environment – failures, unpredictable delays, etc.). In this section, we describe the techniques for these three problems. We note that the result is a distributed execution plan to be executed by the wrappers and the mediator.

9.4.1 Heterogeneous Cost Modeling

Global cost function definition, and the associated problem of obtaining cost-related information from component DBMSs, is perhaps the most-studied of the three problems. A number of possible solutions have emerged, which we discuss below.

The first thing to note is that we are primarily interested in determining the cost of the lower levels of a query execution tree that correspond to the parts of the query executed at component DBMSs. If we assume that all local processing is "pushed down" in the tree, then we can modify the query plan such that the leaves of the tree correspond to subqueries that will be executed at individual component DBMSs. In this case, we are talking about the determination of the costs of these subqueries that are input to the first level (from the bottom) operators. Cost for higher levels of the query execution tree may be calculated recursively, based on the leaf node costs.

Three alternative approaches exist for determining the cost of executing queries at component DBMSs [Zhu and Larson, 1998]:

1. **Black Box Approach.** This approach treats each component DBMS as a black box, running some test queries on it, and from these determines the necessary cost information [Du et al., 1992; Zhu and Larson, 1994].
2. **Customized Approach.** This approach uses previous knowledge about the component DBMSs, as well as their external characteristics, to subjectively determine the cost information [Zhu and Larson, 1996a; Roth et al., 1999; Naacke et al., 1999].
3. **Dynamic Approach.** This approach monitors the run-time behavior of component DBMSs, and dynamically collects the cost information [Lu et al., 1992; Zhu et al., 2000, 2003; Rahal et al., 2004].

We discuss each approach, focusing on the proposals that have attracted the most attention.

9.4.1.1 Black box approach

In the black box approach, which is used in the Pegasus project [Du et al., 1992], the cost functions are expressed logically (e.g., aggregate CPU and I/O costs, selectivity factors), rather than on the basis of physical characteristics (e.g., relation cardinalities, number of pages, number of distinct values for each column). Thus, the cost functions for component DBMSs are expressed as

$$Cost = initialization\ cost + cost\ to\ find\ qualifying\ tuples \\ + cost\ to\ process\ selected\ tuples$$

The individual terms of this formula will differ for different operators. However, these differences are not difficult to specify a priori. The fundamental difficulty is the determination of the term coefficients in these formulae, which change with different component DBMSs. The approach taken in the Pegasus project is to construct a synthetic database (called a *calibrating database*), run queries against it in isolation, and measure the elapsed time to deduce the coefficients.

A problem with this approach is that the calibration database is synthetic, and the results obtained by using it may not apply well to real DBMSs [Zhu and Larson, 1994]. An alternative is proposed in the CORDS project [Zhu and Larson, 1996b], that is based on running probing queries on component DBMSs to determine cost information. Probing queries can, in fact, be used to gather a number of cost information factors. For example, probing queries can be issued to retrieve data from component DBMSs to construct and update the multidatabase catalog. Statistical probing queries can be issued that, for example, count the number of tuples of a relation. Finally, performance measuring probing queries can be issued to measure the elapsed time for determining cost function coefficients.

A special case of probing queries is sample queries [Zhu and Larson, 1998]. In this case, queries are classified according to a number of criteria, and sample queries from each class are issued and measured to derive component cost information. Query classification can be performed according to query characteristics (e.g., unary operation queries, two-way join queries), characteristics of the operand relations (e.g., cardinality, number of attributes, information on indexed attributes), and characteristics of the underlying component DBMSs (e.g., the access methods that are supported and the policies for choosing access methods).

Classification rules are defined to identify queries that execute similarly, and thus could share the same cost formula. For example, one may consider that two queries that have similar algebraic expressions (i.e., the same algebraic tree shape), but different operand relations, attributes, or constants, are executed the same way if their attributes have the same physical properties. Another example is to assume that join order of a query has no effect on execution since the underlying query optimizer applies reordering techniques to choose an efficient join ordering. Thus, two queries that join the same set of relations belong to the same class, whatever ordering is expressed by the user. Classification rules are combined to define query classes. The classification is performed either top-down by dividing a class into more

specific ones, or bottom-up by merging two classes into a larger one. In practice, an efficient classification is obtained by mixing both approaches. The global cost function is similar to the Pegasus cost function in that it consists of three components: initialization cost, cost of retrieving a tuple, and cost of processing a tuple. The difference is in the way the parameters of this function are determined. Instead of using a calibrating database, sample queries are executed and costs are measured. The global cost equation is treated as a regression equation, and the regression coefficients are calculated using the measured costs of sample queries [Zhu and Larson, 1996a]. The regression coefficients are the cost function parameters. Eventually, the cost model quality is controlled through statistical tests (e.g., F-test): if the tests fail, the query classification is refined until quality is sufficient. This approach has been validated over various DBMS and has been shown to yield good results [Zhu and Larson, 2000].

The above approaches require a preliminary step to instantiate the cost model (either by calibration or sampling). This may not be appropriate in MDBMSs because it would slow down the system each time a new DBMS component is added. One way to address this problem, as proposed in the Hermes project, is to progressively learn the cost model from queries [Adali et al., 1996b]. The cost model designed in the Hermes mediator assumes that the underlying component DBMSs are invoked by a function call. The cost of a call is composed of three values: the response time to access the first tuple, the whole result response time, and the result cardinality. This allows the query optimizer to minimize either the time to receive the first tuple or the time to process the whole query, depending on end-user requirements. Initially the query processor does not know any statistics about components DBMSs. Then it monitors on-going queries: it collects processing time of every call and stores it for future estimation. To manage the large amount of collected statistics, the cost manager summarizes them, either without loss of precision or with less precision at the benefit of lower space use and faster cost estimation. Summarization consists of aggregating statistics: the average response time is computed of all the calls that match the same pattern, i.e., those with identical function name and zero or more identical argument values. The cost estimator module is implemented in a declarative language. This allows adding new cost formulae describing the behavior of a particular component DBMS. However, the burden of extending the mediator cost model remains with the mediator developer.

The major drawback of the black box approach is that the cost model, although adjusted by calibration, is common for all component DBMSs and may not capture their individual specifics. Thus it might fail to estimate accurately the cost of a query executed at a component DBMS that exposes unforeseen behavior.

9.4.1.2 Customized Approach

The basis of this approach is that the query processors of the component DBMSs are too different to be represented by a unique cost model as used in the black-box approach. It also assumes that the ability to accurately estimate the cost of

local subqueries will improve global query optimization. The approach provides a framework to integrate the component DBMSs' cost model into the mediator query optimizer. The solution is to extend the wrapper interface such that the mediator gets some specific cost information from each wrapper. The wrapper developer is free to provide a cost model, partially or entirely. Then, the challenge is to integrate this (potentially partial) cost description into the mediator query optimizer. There are two main solutions.

A first solution is to provide the logic within the wrapper to compute three cost estimates: the time to initiate the query process and receive the first result item (called *reset_cost*), the time to get the next item (called *advance_cost*), and the result cardinality. Thus, the total query cost is:

$$Total_access_cost = reset_cost + (cardinality - 1) * advance_cost$$

This solution can be extended to estimate the cost of database procedure calls. In that case, the wrapper provides a cost formula that is a linear equation depending on the procedure parameters. This solution has been successfully implemented to model a wide range of heterogeneous components DBMSs, ranging from a relational DBMS to an image server [Roth et al., 1999]. It shows that a little effort is sufficient to implement a rather simple cost model and this significantly improves distributed query processing over heterogeneous sources.

A second solution is to use a hierarchical generic cost model. As shown in Figure 9.2, each node represents a cost rule that associates a query pattern with a cost function for various cost parameters.

The node hierarchy is divided into five levels depending on the genericity of the cost rules (in Figure 9.2, the increasing width of the boxes shows the increased focus of the rules). At the top level, cost rules apply by default to any DBMS. At the underlying levels, the cost rules are increasingly focused on: specific DBMS, relation, predicate or query. At the time of wrapper registration, the mediator receives wrapper metadata including cost information, and completes its built-in cost model by adding new nodes at the appropriate level of the hierarchy. This framework is sufficiently general to capture and integrate both general cost knowledge declared as rules given by wrapper developers and specific information derived from recorded past queries that were previously executed. Thus, through an inheritance hierarchy , the mediator cost-based optimizer can support a wide variety of data sources. The mediator benefits from specialized cost information about each component DBMS, to accurately estimate the cost of queries and choose a more efficient QEP [Naacke et al., 1999].

Example 9.7. Consider the following relations:

EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)

EMP is stored at component DBMS db_1 and contains 1,000 tuples. ASG is stored at component DBMS db_2 and contains 10,000 tuples. We assume uniform distribution

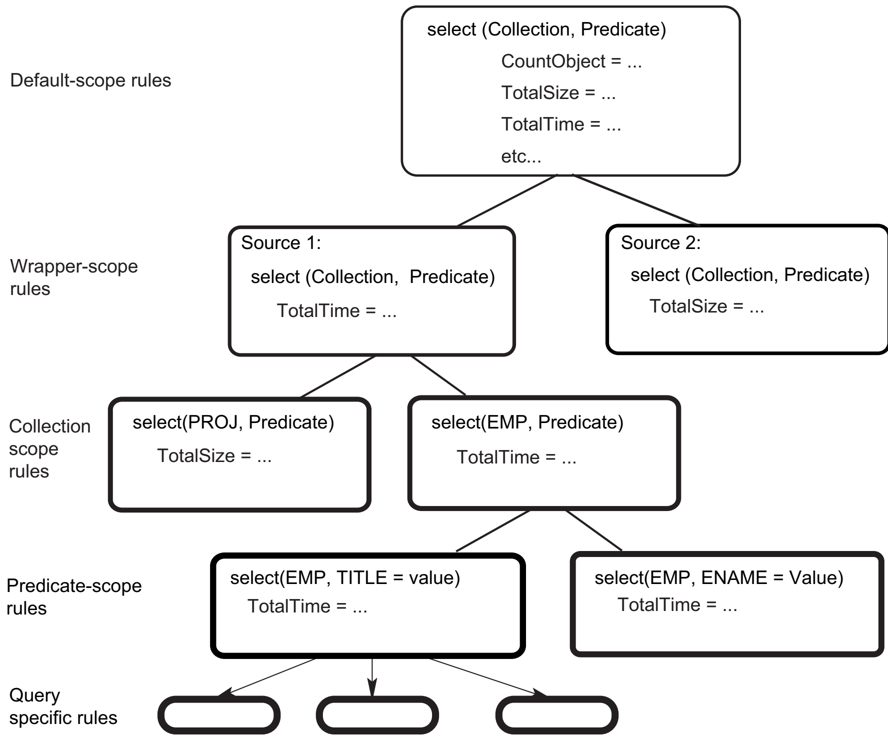

Fig. 9.2 Hierarchical Cost Formula Tree

of attribute values. Half of the ASG tuples have a duration greater than 6. We detail below some parts of the mediator generic cost model (we use superscripts to indicate the access method):

$cost(R) = |R|$

$cost(\sigma_{predicate}(R)) = cost(R)$ (access to R by sequential scan (by default))

$cost(R \bowtie_{A}^{ind} S) = cost(R) + |R| * cost(\sigma_{A=v}(S))$ (using an index-based (ind) join with
the index on $S.A$)

$cost(R \bowtie_{A}^{nl} S) = cost(R) + |R| * cost(S)$ (using a nested-loop (nl) join)

Consider the following global query Q:

```
SELECT *
FROM   EMP, ASG
WHERE  EMP.ENO=ASG.ENO
AND    ASG.DUR>6
```

The cost-based query optimizer generates the following plans to process Q:

$$P_1 = \sigma_{\mathrm{DUR}>6}(\mathrm{EMP} \bowtie^{ind}_{\mathrm{ENO}} \mathrm{ASG})$$

$$P_2 = \mathrm{EMP} \bowtie^{nl}_{\mathrm{ENO}} \sigma_{\mathrm{DUR}>6}(\mathrm{ASG})$$

$$P_3 = \sigma_{\mathrm{DUR}>6}(\mathrm{ASG}) \bowtie^{ind}_{\mathrm{ENO}} \mathrm{EMP}$$

$$P_4 = \sigma_{\mathrm{DUR}>6}(\mathrm{ASG}) \bowtie^{nl}_{\mathrm{ENO}} \mathrm{EMP}$$

Based on the generic cost model, we compute their cost as:

$$\begin{aligned}
cost(P_1) &= cost(\sigma_{\mathrm{DUR}>6}(\mathrm{EMP} \bowtie^{ind}_{\mathrm{ENO}} \mathrm{ASG}) \\
&= cost(\mathrm{EMP} \bowtie^{ind}_{\mathrm{ENO}} \mathrm{ASG}) \\
&= cost(\mathrm{EMP}) + |\mathrm{EMP}| * cost(\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})) \\
&= |\mathrm{EMP}| + |\mathrm{EMP}| * |\mathrm{ASG}| = 10{,}001{,}000 \\
cost(P_2) &= cost(\mathrm{EMP}) + |\mathrm{EMP}| * cost(\sigma_{\mathrm{DUR}>6}(\mathrm{ASG})) \\
&= cost(\mathrm{EMP}) + |\mathrm{EMP}| * cost(\mathrm{ASG}) \\
&= |EMP| + |\mathrm{EMP}| * |\mathrm{ASG}| = 10{,}001{,}000 \\
cost(P_3) &= cost(P_4) = |\mathrm{ASG}| + \frac{|\mathrm{ASG}|}{2} * |\mathrm{EMP}| \\
&= 5{,}010{,}000
\end{aligned}$$

Thus, the optimizer discards plans P_1 and P_2 to keep either P_3 or P_4 for processing Q. Let us assume now that the mediator imports specific cost information about component DBMSs. db_1 exports the cost of accessing EMP tuples as:

$$cost(\sigma_{A=v}(R)) = |\sigma_{A=v}(R)|$$

db_2 exports the specific cost of selecting ASG tuples that have a given ENO as:

$$cost(\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})) = |\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})|$$

The mediator integrates these cost functions in its hierarchical cost model, and can now estimate more accurately the cost of the QEPs:

$$\begin{aligned}
cost(P_1) &= |\mathrm{EMP}| + |\mathrm{EMP}| * |\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})| \\
&= 1{,}000 + 1{,}000 * 10 \\
&= 11{,}000 \\
cost(P_2) &= |\mathrm{EMP}| + |\mathrm{EMP}| * |\sigma_{\mathrm{DUR}>6}(\mathrm{ASG})|
\end{aligned}$$

$$
\begin{aligned}
&= |\text{EMP}| + |\text{EMP}| * \frac{|\text{ASG}|}{2} \\
&= 5,001,000 \\
cost(P_3) &= |\text{ASG}| + \frac{|\text{ASG}|}{2} * |\sigma_{\text{ENO}=v}(\text{EMP})| \\
&= 10,000 + 5,000 * 1 \\
&= 15,000 \\
cost(P_4) &= |\text{ASG}| + \frac{|\text{ASG}|}{2} * |\text{EMP}| \\
&= 10,000 + 5,000 * 1,000 \\
&= 5,010,000
\end{aligned}
$$

The best QEP is now P_1 which was previously discarded because of lack of cost information about component DBMSs. In many situations P_1 is actually the best alternative to process Q_1. ♦

The two solutions just presented are well suited to the mediator/wrapper architecture and offer a good tradeoff between the overhead of providing specific cost information for diverse component DBMSs and the benefit of faster heterogeneous query processing.

9.4.1.3 Dynamic Approach

The above approaches assume that the execution environment is stable over time. However, in most cases, the execution environment factors are frequently changing. Three classes of environmental factors can be identified based on their dynamicity [Rahal et al., 2004]. The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory. The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and database schema. The third class for almost stable factors (every month to every year) includes DBMS type, database location, and CPU speed. We focus on solutions that deal with the first two classes.

One way to deal with dynamic environments where network contention, data storage or available memory change over time is to extend the sampling method [Zhu, 1995] and consider user queries as new samples. Query response time is measured to adjust the cost model parameters at run time for subsequent queries. This avoids the overhead of processing sample queries periodically, but still requires heavy computation to solve the cost model equations and does not guarantee that cost model precision improves over time. A better solution, called qualitative [Zhu

et al., 2000], defines the system contention level as the combined effect of frequently changing factors on query cost. The system contention level is divided into several discrete categories: high, medium, low, or no system contention. This allows for defining a multi-category cost model that provides accurate cost estimates while dynamic factors are varying. The cost model is initially calibrated using probing queries. The current system contention level is computed over time, based on the most significant system parameters. This approach assumes that query executions are short, so the environment factors remain rather constant during query execution. However, this solution does not apply to long running queries, since the environment factors may change rapidly during query execution.

To manage the case where the environment factor variation is predictable (e.g., the daily DBMS load variation is the same every day), the query cost is computed for successive date ranges [Zhu et al., 2003]. Then, the total cost is the sum of the costs for each range. Furthermore, it may be possible to learn the pattern of the available network bandwidth between the MDBMS query processor and the component DBMS [Vidal et al., 1998]. This allows adjusting the query cost depending on the actual date.

9.4.2 Heterogeneous Query Optimization

In addition to heterogeneous cost modeling, multidatabase query optimization must deal with the issue of the heterogeneous computing capabilities of component DBMSs. For instance, one component DBMS may support only simple select operations while another may support complex queries involving join and aggregate. Thus, depending on how the wrappers export such capabilities, query processing at the mediator level can be more or less complex. There are two main approaches to deal with this issue depending on the kind of interface between mediator and wrapper: query-based and operator-based.

1. **Query-based.** In this approach, the wrappers support the same query capability, e.g., a subset of SQL, which is translated to the capability of the component DBMS. This approach typically relies on a standard DBMS interface such as Open Database Connectivity (ODBC) and its extensions for the wrappers or SQL Management of External Data (SQL/MED) [Melton et al., 2001]. Thus, since the component DBMSs appear homogeneous to the mediator, query processing techniques designed for homogeneous distributed DBMS can be reused. However, if the component DBMSs have limited capabilities, the additional capabilities must be implemented in the wrappers, e.g., join queries may need to be handled at the wrapper, if the component DBMS does not support join.
2. **Operator-based.** In this approach, the wrappers export the capabilities of the component DBMSs through compositions of relational operators. Thus, there is more flexibility in defining the level of functionality between the mediator

and the wrapper. In particular, the different capabilities of the component DBMSs can be made available to the mediator. This makes wrapper construction easier at the expense of more complex query processing in the mediator. In particular, any functionality that may not be supported by component DBMSs (e.g., join) will need to be implemented at the mediator.

In the rest of this section, we present, in more detail, the approaches to query optimization.

9.4.2.1 Query-based Approach

Since the component DBMSs appear homogeneous to the mediator, one approach is to use a distributed cost-based query optimization algorithm (see Chapter 8) with a heterogeneous cost model (see Section 9.4.1). However, extensions are needed to convert the distributed execution plan into subqueries to be executed by the component DBMSs and into subqueries to be executed by the mediator. The hybrid two-step optimization technique is useful in this case (see Section 8.4.4): in the first step, a static plan is produced by a centralized cost-based query optimizer; in the second step, at startup time, an execution plan is produced by carrying out site selection and allocating the subqueries to the sites. However, centralized optimizers restrict their search space by eliminating bushy join trees from consideration. Almost all the systems use left linear join orders where the right subtree of a join node is always a leaf node corresponding to a base relation (Figure 9.3a). Consideration of only left linear join trees gives good results in centralized DBMSs for two reasons: it reduces the need to estimate statistics for at least one operand, and indexes can still be exploited for one of the operands. However, in multidatabase systems, these types of join execution plans are not necessarily the preferred ones as they do not allow any parallelism in join execution. As we discussed in earlier chapters, this is also a problem in homogeneous distributed DBMSs, but the issue is more serious in the case of multidatabase systems, because we wish to push as much processing as possible to the component DBMSs.

A way to resolve this problem is to somehow generate bushy join trees and consider them at the expense of left linear ones. One way to achieve this is to apply a cost-based query optimizer to first generate a left linear join tree, and then convert it to a bushy tree [Du et al., 1995]. In this case, the left linear join execution plan can be optimal with respect to total time, and the transformation improves the query response time without severely impacting the total time. A hybrid algorithm that concurrently performs a bottom-up and top-down sweep of the left linear join execution tree, transforming it, step-by-step, to a bushy one has been proposed [Du et al., 1995]. The algorithm maintains two pointers, called *upper anchor nodes* (UAN) on the tree. At the beginning, one of these, called the bottom UAN (UAN_B), is set to the grandparent of the leftmost root node (join with R_3 in Figure 9.3a), while the second one, called the top UAN (UAN_T), is set to the root (join with R_5). For each UAN the algorithm selects a *lower anchor node* (LAN). This is the node closest to the UAN and whose

right child subtree's response time is within a designer-specified range, relative to that of the UAN's right child subtree. Intuitively, the LAN is chosen such that its right child subtree's response time is **close** to the corresponding UAN's right child subtree's response time. As we will see shortly, this helps in keeping the transformed bushy tree balanced, which reduces the response time.

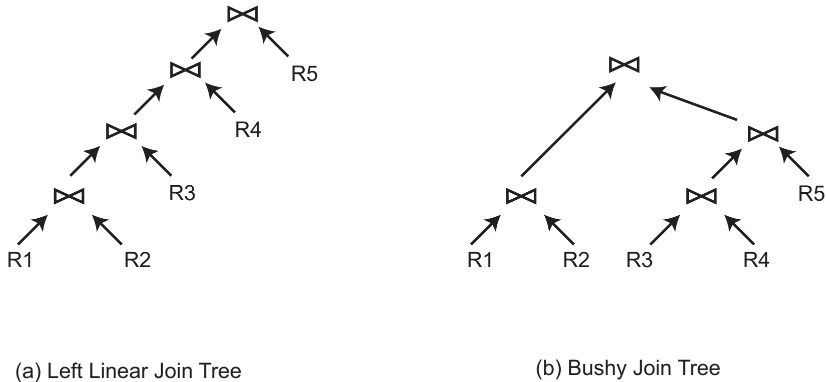

Fig. 9.3 Left Linear versus Bushy Join Tree

At each step, the algorithm picks one of the UAN/LAN pairs (strictly speaking, it picks the UAN and selects the appropriate LAN, as discussed above), and performs the following translation for the segment between that LAN and UAN pair:

1. The left child of UAN becomes the new UAN of the transformed segment.
2. The LAN remains unchanged, but its right child node is replaced with a new join node of two subtrees, which were the right child subtrees of the input UAN and LAN.

The UAN mode that will be considered in that particular iteration is chosen according to the following heuristic: choose UAN_B if the response time of its left child subtree is smaller than that of UAN_T's subtree; otherwise choose UAN_T. If the response times are the same, choose the one with the more unbalanced child subtree.

At the end of each transformation step, the UAN_B and UAN_T are adjusted. The algorithm terminates when $UAN_B = UAN_T$, since this indicates that no further transformations are possible. The resulting join execution tree will be almost balanced, producing an execution plan whose response time is reduced due to parallel execution of the joins.

The algorithm described above starts with a left linear join execution tree that is generated by a commercial DBMS optimizer. While this is a good starting point, it can be argued that the original linear execution plan may not fully account for the peculiarities of the distributed multidatabase characteristics, such as data replication. A special global query optimization algorithm [Evrendilek et al., 1997] can take

these into consideration. Starting from an initial join graph, the algorithm checks for different parenthesizations of this linear join execution order and produces a parenthesized order, which is optimal with respect to response time. The result is an (almost) balanced join execution tree. Performance evaluations indicate that this approach produces better quality plans at the expense of longer optimization time.

9.4.2.2 Operator-based Approach

Expressing the capabilities of the component DBMSs through relational operators allows tight integration of query processing between mediator and wrappers. In particular, the mediator/wrapper communication can be in terms of subplans. We illustrate the operator-based approach with planning functions proposed in the Garlic project [Haas et al., 1997a]. In this approach, the capabilities of the component DBMSs are expressed by the wrappers as planning functions that can be directly called by a centralized query optimizer. It extends the rule-based optimizer proposed by Lohman [1988] with operators to create temporary relations and retrieve locally-stored data. It also creates the *PushDown* operator that pushes a portion of the work to the component DBMSs where it will be executed. The execution plans are represented, as usual, as operator trees, but the operator nodes are annotated with additional information that specifies the source(s) of the operand(s), whether the results are materialized, and so on. The Garlic operator trees are then translated into operators that can be directly executed by the execution engine.

Planning functions are considered by the optimizer as enumeration rules. They are called by the optimizer to construct subplans using two main functions: *accessPlan* to access a relation, and *joinPlan* to join two relations using the access plans. These functions precisely reflect the capabilities of the component DBMSs with a common formalism.

Example 9.8. We consider three component databases, each at a different site. Component database db_1 stores relation EMP(ENO, ENAME, CITY). Component database db_2 stores relation ASG(ENO, PNAME, DUR). Component database db_3 stores only employee information with a single relation of schema EMPASG(ENAME, CITY, PNAME, DUR), whose primary key is (ENAME, PNAME). Component databases db_1 and db_2 have the same wrapper w_1 whereas db_3 has a different wrapper w_2.

Wrapper w_1 provides two planning functions typical of a relational DBMS. The accessPlan rule

$$\text{accessPlan}(R\text{: relation}, A\text{: attribute list}, P\text{: select predicate}) = \\ \text{scan}(R, A, P, db(R))$$

produces a scan operator that accesses tuples of R from its component database $db(R)$ (here we can have $db(R) = db_1$ or $db(R) = db_2$), applies select predicate P, and projects on the attribute list A. The joinPlan rule

joinPlan(R_1,R_2: relations, A: attribute list, P: join predicate) =
join (R_1,R_2, A, P)
condition: $db(R_1) \neq db(R_2)$

produces a join operator that accesses tuples of relations R_1 and R_2 and applies join predicate P and projects on attribute list A. The condition expresses that R_1 and R_2 are stored in different component databases (i.e., db_1 and db_2). Thus, the join operator is implemented by the wrapper.

Wrapper w_2 also provides two planning functions. The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
fetch(CITY="c")
condition: (CITY="c") $\subseteq P$

produces a fetch operator that directly accesses (entire) employee tuples in component database db_3 whose CITY value is "c". The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
scan(R,A,P)

produces a scan operator that accesses tuples of relation R in the wrapper and applies select predicate P and attribute project list A. Thus, the scan operator is implemented by the wrapper, not the component DBMS.

Consider the following SQL query submitted to mediator m:

```
SELECT ENAME, PNAME, DUR
FROM   EMPASG
WHERE  CITY = "Paris" AND DUR > 24
```

Assuming the GAV approach, the global view EMPASG(ENAME, CITY, PNAME, DUR) can be defined as follows (for simplicity, we prefix each relation by its component database name):

$$\text{EMPASG} = (db_1.\text{EMP} \bowtie db_2.\text{ASG}) \cup db_3.\text{EMPASG}$$

After query rewriting in GAV and query optimization, the operator-based approach could produce the QEP shown in Figure 9.4. This plan shows that the operators that are not supported by the component DBMS are to be implemented by the wrappers or the mediator. ♦

Using planning functions for heterogeneous query optimization has several advantages in multi-DBMSs. First, planning functions provide a flexible way to express precisely the capabilities of component data sources. In particular, they can be used to model non-relational data sources such as web sites. Second, since these rules are declarative, they make wrapper development easier. The only important development for wrappers is the implementation of specific operators, e.g., the scan operator of db_3 in Example 9.8. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

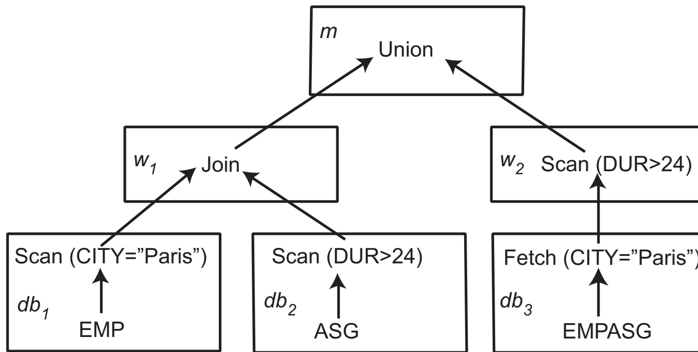

Fig. 9.4 Heterogeneous Query Execution Plan

The operator-based approach has also been successfully used in DISCO, a multi-DBMS designed to access multiple databases over the web [Tomasic et al., 1996, 1997, 1998]. DISCO uses the GAV approach and supports an object data model to represent both mediator and component database schemas and data types. This allows easy introduction of new component databases, easily handling potential type mismatches. The component DBMS capabilities are defined as a subset of an algebraic machine (with the usual operators such as scan, join and union) that can be partially or entirely supported by the wrappers or the mediator. This gives much flexibility for the wrapper implementors in deciding where to support component DBMS capabilities (in the wrapper or in the mediator). Furthermore, compositions of operators, including specific data sets, can be specified to reflect component DBMS limitations. However, query processing is more complicated because of the use of an algrebraic machine and compositions of operators. After query rewriting on the component schemas, there are three main steps [Kapitskaia et al., 1997].

1. **Search space generation.** The query is decomposed into a number of QEPs, which constitutes the search space for query optimization. The search space is generated using a traditional search strategy such as dynamic programming.
2. **QEP decomposition.** Each QEP is decomposed into a forest of *n wrapper QEPs* and a *composition QEP*. Each wrapper QEP is the largest part of the initial QEP that can be entirely executed by the wrapper. Operators that cannot be performed by a wrapper are moved up to the composition QEP. The composition QEP combines the results of the wrapper QEPs in the final answer, typically through unions and joins of the intermediate results produced by the wrappers.
3. **Cost evaluation.** The cost of each QEP is evaluated using a hierarchical cost model discussed in Section 9.4.1.

9.4.3 Adaptive Query Processing

Multidatabase query processing, as discussed so far, follows essentially the principles of traditional query processing whereby an optimal QEP is produced for a query based on a cost model, which is then executed. The underlying assumption is that the multidatabase query optimizer has sufficient knowledge about query runtime conditions in order to produce an efficient QEP and the runtime conditions remain stable during execution. This is a fair assumption for multidatabase queries with few data sources running in a controlled environment. However, this assumption is inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions.

Example 9.9. Consider the QEP in Figure 9.5 with relations EMP, ASG, PROJ and PAY at sites s_1, s_2, s_3, s_4, respectively. The crossed arrow indicates that, for some reason (e.g., failure), site s_2 (where ASG is stored) is not available at the beginning of execution. Let us assume, for simplicity, that the query is to be executed according to the iterator execution model [Graefe and McKenna, 1993], such that tuples flow from the left most relation,

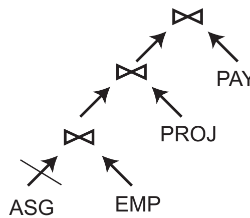

Fig. 9.5 Query Execution Plan with Blocked Data Source

Because of the unavailability of s_2, the entire pipeline is blocked, waiting for ASG tuples to be produced. However, with some reoganization of the plan, some other operators could be evaluated while waiting for s_2, for instance, to evaluate the join of EMP and PAY. ♦

This simple example illustrates that a typical static plan cannot cope with unpredictable data source unavailability [Amsaleg et al., 1996a]. More complex examples involve continuous queries [Madden et al., 2002b], expensive predicates [Porto et al., 2003] and data skew [Shah et al., 2003]. The main solution is to have some adaptive behavior during query processing, i.e., *adaptive query processing*. Adaptive query processing is a form of dynamic query processing, with a feedback loop between the execution environment and the query optimizer in order to react to unforeseen variations of runtime conditions. A query processing system is defined as adaptive if it receives information from the execution environment and determines its behavior according to that information in an iterative manner [Hellerstein et al., 2000; Gounaris et al., 2002b]. In the context of multidatabase systems, the execution environment

includes the mediator, wrappers and component DBMSs. In particular, wrappers should be able to collect information regarding execution within the component DBMSs. Obviously, this is harder to do with legacy DBMSs.

In this section, we first provide a general presentation of the adaptive query processing process. Then, we present, in more detail, the Eddy approach [Avnur and Hellerstein, 2000] that provides a powerful framework for adaptive query processing techniques. Finally, we discuss major extensions to Eddy.

9.4.3.1 Adaptive Query Processing Process

Adaptive query processing adds to the traditional query processing process the following activities: monitoring, assessing and reacting. These activities are logically implemented in the query processing system by sensors, assessment components, and reaction components, respectively. These components may be embedded into control operators of the QEP, e.g., the *Exchange* operator [Graefe and McKenna, 1993]. Monitoring involves measuring some environment parameters within a time window, and reporting them to the assessment component. The latter analyzes the reports and considers thresholds to arrive at an adaptive reaction plan. Finally, the reaction plan is communicated to the reaction component that applies the reactions to query execution.

Typically, an adaptive process specifies the frequency with which each component will be executed. There is a tradeoff between reactiveness, in which higher values lead to eager reactions, and the overhead caused by the adaptive process. A generic representation of the adaptive process is given by the function $f_{adapt}(E,T) \rightarrow Ad$, where E is a set of monitored environment parameters, T is a set of threshold values and Ad is a possibly empty set of adaptive reactions. The elements of E,T and Ad, called adaptive elements, obviously may vary in a number of ways depending on the application. The most important elements are the monitoring parameters and the adaptive reactions. We now describe them, following the presentation in [Gounaris et al., 2002b].

Monitoring parameters.

Monitoring query runtime parameters involves placing sensors at key places of the QEP and defining observation windows, during which sensors collect information. It also requires the specification of a communication mechanism to pass collected information to the assessment component. Examples of candidates for monitoring are:

- Memory size. Monitoring available memory size allows, for instance, operators to react to memory shortage or memory increase [Shah et al., 2003].
- Data arrival rates. Monitoring the variations in data arrival rates may enable the query processor to do useful work while waiting for a blocked data source.

- Actual statistics. Database statistics in a multidatabase environment tend to be inaccurate, if at all available. Monitoring the actual size of relations and intermediate results may lead to important modifications in the QEP. Furthermore, the usual data assumptions, in which the selectivity of predicates over attributes in a relation are considered to be mutually independent, can be abandoned and real selectivity values can be computed.
- Operator execution cost. Monitoring the actual cost of operator execution, including production rates, is useful for better operator scheduling. Furthermore, monitoring the size of the queues placed before operators may avoid overload situations [Tian and DeWitt, 2003b].
- Network throughput. In multidatabase query evaluation with remote data sources, monitoring network throughput may be helpful to define the data retrieval block size. In a lower throughput network, the system may react with larger block sizes to reduce network penalty.

Adaptive reactions.

Adaptive reactions modify query execution behavior according to the decisions taken by the assessment component. Important adaptive reactions are the following:

- Change schedule: modifies the order in which operators in the QEP get scheduled. *Query Scrambling* [Amsaleg et al., 1996a; Urhan et al., 1998a] reacts by a *change schedule* of the plan, e.g., to reorganize the QEP in Example 9.9, to avoid stalling on a blocked data source during query evaluation. Eddy adopts finer reaction where operator scheduling can be decided on a tuple basis.
- Operator replacement: replaces a physical operator by an equivalent one. For example, depending on the available memory, the system may choose between a nested loop join or a hash join. Operator replacement may also change the plan by introducing a new operator to join the intermediate results produced by a previous adaptive reaction. Query Scrambling, for instance, may introduce new operators to evaluate joins between the results of *change schedule* reactions.
- Operator behavior: modifies the physical behavior of an operator. For example, the symmetric hash join [Wilschut and Apers, 1991] or ripple join algorithms [Haas and Hellerstein, 1999b] constantly alternate the inner/outer relation roles between their input tuples.
- Data repartitioning: considers the dynamic repartitioning of a relation through multiple nodes using intra-operator parallelism [Shah et al., 2003]. Static partitioning of a relation tends to produce load imbalance between nodes. For example, information partitioned according to their associated geographical region (i.e., continent) may exhibit different access rates during the day because of the time differences in users' locations.

- Plan reformulation: computes a new QEP to replace an inefficient one. The optimizer considers actual statistics and state information, collected on the fly, to produce a new plan.

9.4.3.2 Eddy Approach

Eddy is a general framework for adaptive query processing. It was developed in the context of the Telegraph project with the goal of running queries on large volumes of online data with unpredictable input rates and fluctuations in the running environment.

For simplicity, we only consider select-project-join (SPJ) queries. Select operators can include expensive predicates [Hellerstein and Stonebraker, 1993]. The process of generating a QEP from an input SPJ query begins by producing a spanning tree of the query graph G modeling the input query. The choice among join algorithms and relation access methods favors adaptiveness. A QEP can be modeled as a tuple $Q = \langle D, P, C \rangle$, where D is a set of data sources, P is a set of query predicates with associated algorithms, and C is a set of ordering constraints that must be followed during execution. Observe that multiple valid spanning trees can be derived from G that obey the constraints in C, by exploring the search space composed of equivalent plans with different predicate orders. There is no need to find an optimal QEP during query compilation. Instead, operator ordering is done on the fly on a tuple-per-tuple basis (i.e., tuple routing). The process of QEP compilation is completed by adding the *Eddy operator* which is an n-ary physical operator placed between data sources in D and query predicates in P.

Example 9.10. Consider a three-relation query $Q = \sigma_p(R) \bowtie S \bowtie T$, where joins are equi-joins. Assume that the only access method to relation T is through an index on join attribute $T.A$, i.e., the second join can only be an index join over $T.A$. Assume also that σ_p is an expensive predicate (e.g., a predicate over the results of running a program over values of $R.B$). Under these assumptions, the QEP is defined as $D = \{R, S, T\}$, $P = \{\sigma_p(R), R \bowtie_1 S, S \bowtie_2 T\}$ and $C = \{S \prec T\}$. The constraint $\prec$ imposes S tuples to probe T tuples, based on the index on $T.A$.

Figure 9.6 shows a QEP produced by the compilation of query Q with Eddy. An ellipse corresponds to a physical operator (i.e., either the Eddy operator or an algorithm implementing a predicate $p \in P$). As usual, the bottom of the plan presents the data sources. In the absence of a scan access method, relation T access is wrapped by the index join implementing the second join, and, thus, does not appear as a data source. The arrows specify pipeline dataflow following a producer-consumer relationship. Finally, an arrow departing from the Eddy models the production of output tuples. ♦

Eddy provides fine-grain adaptiveness by deciding on the fly how to route tuples through predicates according to a scheduling policy. During query execution, tuples in data sources are retrieved and staged into an input buffer managed by the Eddy operator. Eddy responds to data source unavailability by simply reading from another data source and staging tuples in the buffer pool.

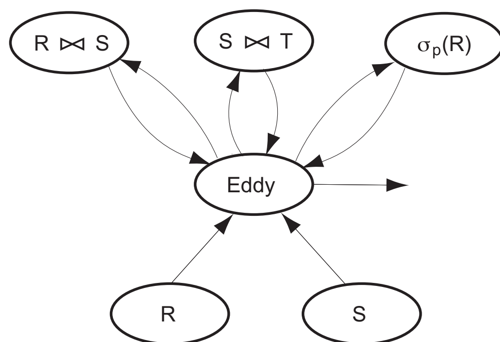

Fig. 9.6 A Query Execution Plan with Eddy.

The flexibility of choosing the currently available data source is obtained by relaxing the fixed order of predicates in a QEP. In Eddy, there is no fixed QEP and each tuple follows its own path through predicates according to the constraints in the plan and its own history of predicate evaluation.

The tuple-based routing strategy produces a new QEP topology. The Eddy operator together with its managed predicates form a circular dataflow in which tuples leave the Eddy operator to be evaluated by the predicates, which in turn bounce back output tuples to the Eddy operator. A tuple leaves the circular dataflow either when it is eliminated by a predicate evaluation or the Eddy operator realizes that the tuple has passed through all the predicates in its list. The lack of a fixed QEP requires each tuple to register the set of predicates it is eligible for. For example, in Figure 9.6, S tuples are eligible for the two join predicates but are not eligible for predicate $\sigma_p(R)$.

Let us now present, in more detail, how Eddy adaptively performs join ordering and scheduling.

Adaptive join ordering.

A fixed QEP (produced at compile time) dictates the join ordering and specifies which relations can be pipelined through the join operators. This makes query execution simple. When, as in Eddy, there is no fixed QEP, the challenge is to dynamically order pipelined join operators at run time, while tuples from different relations are flowing in. Ideally, when a tuple of a relation participating in a join arrives, it should be sent to a join operator (chosen by the scheduling policy) to be processed on the fly. However, most join algorithms cannot process some incoming tuples on the fly because they are asymmetric with respect to the way inner and outer tuples are processed. Consider the basic hash-based join algorithm, for instance: the inner relation is fully read during

the build phase to construct a hash table, whereas tuples in the outer relation are pipelined during the probe phase. Thus, an incoming inner tuple cannot be processed on the fly as it must be stored in the hash table and the processing will be possible when the entire hash table has been built. Similarly, the nested loop join algorithm is asymmetric as only the inner relation must be read entirely for each tuple of the outer relation. Join algorithms with some kind of asymmetry offer few opportunities for alternating input relations between inner and outer roles. Thus, to relax the order in which join inputs are consumed, symmetric join algorithms are needed where the role played by the relations in a join may change without producing incorrect results.

The earliest example of a symmetric join algorithm is the symmetric hash join [Wilschut and Apers, 1991], which uses two hash tables, one for each input relation. The traditional build and probe phases of the basic hash join algorithm are simply interleaved. When a tuple arrives, it is used to probe the hash table corresponding to the other relation and find matching tuples. Then, it is inserted in its corresponding hash table so that tuples of the other relation arriving later can be joined. Thus, each arriving tuple can be processed on the fly. Another popular symmetric join algorithm is the ripple join [Haas and Hellerstein, 1999b], which can be viewed as a generalization of the nested loop join algorithm where the roles of inner and outer relation continually alternate during query execution. The main idea is to keep the probing state of each input relation, with a pointer that indicates the last tuple used to probe the other relation. At each toggling point, a change of roles between inner and outer relations occurs. At this point, the new outer relation starts to probe the inner input from its pointer position onwards, to a specified number of tuples. The inner relation, in turn, is scanned from its first tuple to its pointer position minus 1. The number of tuples processed at each stage in the outer relation gives the toggling rate and can be adaptively monitored.

Using symmetric join algorithms, Eddy can achieve flexible join ordering by controlling the history and constraints regarding predicate evaluation on a tuple basis. This control is implemented using two sets of *progress bits* carried by each tuple, which indicate, respectively, the predicates to which the tuple is ready to be evaluated by (i.e., the “ready bits”) and the set of predicates already evaluated (i.e., the “done bits”). When a tuple t is read into an Eddy operator, all done bits are zeroed and the predicates without ordering constraints, and to which t is eligible for, have their corresponding ready bits set. After each predicate evaluation, the corresponding done bit is set and the ready bits are updated, accordingly. When a join concatenates a pair of tuples, their done bits are ORed and a new set of ready bits are turned on. Combining progress bits and symmetric join algorithms allows Eddy to schedule predicates in an adaptive way.

Adaptive scheduling.

Given a set of candidate predicates, Eddy must adaptively select the one to which each tuple will be sent. Two main principles drive the choice of a predicate in Eddy: cost and selectivity. Predicate costs are measured as a function of the consumption

rate of each predicate. Remember that the Eddy operator holds tuples in its internal buffer, which is shared by all predicates. Low cost (i.e., fast) predicates finish their work quicker and request new tuples from the Eddy. As a result, low cost predicates get allocated more tuples than high cost predicates. This strategy, however, is agnostic with respect to predicate selectivity. Eddy's tuple routing strategy is complemented by a simple *lottery scheduling* mechanism that learns about predicate selectivity [Arpaci-Dusseau et al., 1999]. The strategy credits a ticket to a predicate whenever the latter gets scheduled a tuple. Once a tuple has been processed and is bounced back to the Eddy, the corresponding predicate gets its ticket amount decremented. Combining cost and selectivity criteria becomes easy. Eddy continuously runs a lottery among predicates currently requesting tuples. The predicate with higher count of tickets wins the lottery and gets scheduled.

Another interesting issue is the choice of the running tuple from the input buffer. In order to end query processing, all tuples in the input buffer must be evaluated. Thus, a difference in tuple scheduling may reflect user preferences with respect to tuple output. For example, Eddy may favor tuples with higher number of done bits set, so that the user receives first results earlier.

9.4.3.3 Extensions to Eddy

The Eddy approach has been extended in various directions. In the cherry picking approach [Porto et al., 2003], context is used instead of simple ticket-based scheduling. The relationship among expensive predicate input attribute values are discovered at runtime and used as the basis for adaptive tuple scheduling. Given a query Q with $D = \{R[A,B,C]\}$, $P = \{\sigma_p^1(R.A), \sigma_p^2(R.B), \sigma_p^3(R.C)\}$ and $C = \emptyset$, the main idea is to model the input attribute values of the expensive predicates in P as a hypergraph $G = (V,E)$, where V is a set of n node partitions, with n being the number of expensive predicates. Each partition corresponds to a single attribute of the input relation R that are input to a predicate in P and each node corresponds to a distinct value of that attribute. An hyperedge $e = \{a_i, b_j, c_k\}$ corresponds to a tuple of relation R. The degree of a node v_i corresponds to the number of hyperedges in which v_i takes part. With this modeling, efficiently evaluating query Q corresponds to eliminating as quickly as possible the hyperedges in G. An hyperedge is eliminated whenever a value associated with one of its nodes is evaluated by a predicate in P and returns false. Furthermore, node degrees model hidden attribute dependencies, so that when the result of a predicate evaluation over a value v_i returns false, all hyperedges (i.e., tuples) that v_i takes part in are also eliminated. An adaptive content-sensitive strategy to evaluate a query Q is proposed for this model. It schedules values to be evaluated by a predicate according to the *Fanout* of its corresponding node, computed as the product of the node degree in the hypergraph G with the ratio between the corresponding predicate selectivity and predicate unitary evaluation cost.

Another interesting extension is distributed Eddies [Tian and DeWitt, 2003b] to deal with distributed input data streams. Since a centralized Eddy operator may quickly become a bottleneck, a distributed approach is proposed for tuple routing.

Each operator decides on the next operator to route a tuple to based on its history of operator's evaluation (i.e., done bits) and statistics collected from the remaining operators. In a distributed setting, each operator may run at a different node in the network with a queue holding input tuples. The query optimization problem is specified by considering two new metrics for measuring stream query performance: average response time and maximum data rate. The former corresponds to the average time tuples take to traverse the operators in a plan, whereas the latter measures the maximum throughput the system can withstand without overloading. Routing strategies use the following parameters: operator's cost, selectivity, length of operator's input queue and probability of an operator being routed a tuple. The combination of these parameters yields efficient query evaluation. Using operator's cost and selectivity guarantee that low-cost and highly selective operators are given higher routing priority. Queue length provides information on the average time tuples are staged in queues. Managing operator's queue length allows the routing decision to avoid overloaded operators. Thus, by supporting routing policies, each operator is able to individually make routing decisions, thereby avoiding the bottlneck of a centralized router.

9.5 Query Translation and Execution

Query translation and execution is performed by the wrappers using the component DBMSs. A wrapper encapsulates the details of one or more component databases, each supported by the same DBMS (or file system). It also exports to the mediator the component DBMS capabilities and cost functions in a common interface. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non-SQL databases [Roth and Schwartz, 1997].

The main function of a wrapper is conversion between the common interface and the DBMS-dependent interface. Figure 9.7 shows these different levels of interfaces between the mediator, the wrapper and the component DBMSs. Note that, depending on the level of autonomy of the component DBMSs, these three components can be located differently. For instance, in the case of strong autonomy, the wrapper should be at the mediator site, possibly on the same server. Thus, communication between a wrapper and its component DBMS incurs network cost. However, in the case of a cooperative component database (e.g., within the same organization), the wrapper could be installed at the component DBMS site, much like an ODBC driver. Thus, communication between the wrapper and the component DBMS is much more efficient.

The information necessary to perform conversion is stored in the wrapper schema that includes the local schema exported to the mediator in the common interface (e.g., relational) and the schema mappings to transform data between the local schema and the component database schema and vice-versa. We discussed schema mappings in Chapter 4. Two kinds of conversion are needed. First, the wrapper must translate the input QEP generated by the mediator and expressed in a common interface

into calls to the component DBMS using its DBMS-dependent interface. These calls yield query execution by the component DBMS that return results expressed in the DBMS-dependent interface. Second, the wrapper must translate the results to the common interface format so that they can be returned to the mediator for integration. In addition, the wrapper can execute operations that are not supported by the component DBMS (e.g., the scan operation by wrapper w_2 in Figure 9.4).

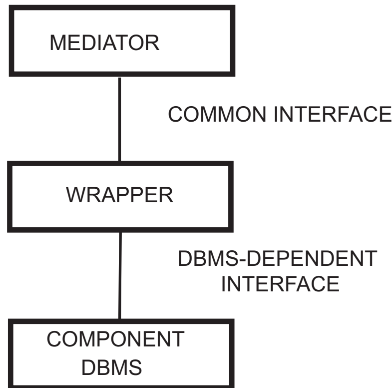

Fig. 9.7 Wrapper interfaces

As discussed in Section 9.4.2, the common interface to the wrappers can be query-based or operator-based. The problem of translation is similar in both approaches. To illustrate query translation in the following example, we use the query-based approach with the SQL/MED standard that allows a relational DBMS to access external data represented as foreign relations in the wrapper's local schema. This example, borrowed from [Melton et al., 2001], illustrates how a very simple data source can be wrapped to be accessed through SQL.

Example 9.11. We consider relation EMP(ENO, ENAME, CITY) stored in a very simple component database, in server *ComponentDB*, built with Unix text files. Each EMP tuple can then be stored as a line in a file, e.g., with the attributes separated by ":". In SQL/MED, the definition of the local schema for this relation together with the mapping to a Unix file can be declared as a foreign relation with the following statement:

```
CREATE  FOREIGN TABLE EMP
        ENO INTEGER,
        ENAME VARCHAR(30),
        CITY VARCHAR(20)
SERVER  ComponentDB
OPTIONS (Filename '/usr/EngDB/emp.txt', Delimiter ':')
```

Then, the mediator can send the wrapper supporting access to this relation SQL statements. For instance, the query:

```
SELECT ENAME
FROM   EMP
```

can be translated by the wrapper using the following Unix shell command to extract the relevant attribute:

```
cut -d: -f2 /usr/EngDB/emp
```

Additional processing, e.g., for type conversion, can then be done using programming code. ♦

Wrappers are mostly used for read-only queries, which makes query translation and wrapper construction relatively easy. Wrapper construction typically relies on CASE tools with reusable components to generate most of the wrapper code [Tomasic et al., 1997]. Furthermore, DBMS vendors provide wrappers for transparently accessing their DBMS using standard interfaces. However, wrapper construction is much more difficult if updates to component databases are to be supported through wrappers (as opposed to directly updating the component databases through their DBMS). The main problem is due to the heterogeneity of integrity constraints between the common interface and the DBMS-dependent interface. As discussed in Chapter 5, integrity constraints are used to reject updates that violate database consistency. In modern DBMSs, integrity constraints are explicit and specified as rules as part of the database schema. However, in older DBMSs or simpler data sources (e.g., files), integrity constraints are implicit and implemented by specific code in the applications. For instance, in Example 9.11, there could be applications with some embedded code that rejects insertions of new lines with an existing ENO in the EMP text file. This code corresponds to a unique key constraint on ENO in relation EMP but is not readily available to the wrapper. Thus, the main problem of updating through a wrapper is to guarantee component database consistency by rejecting all updates that violate integrity constraints, whether they are explicit or implicit. A software engineering solution to this problem uses a CASE tool with reverse engineering techniques to identify within application code the implicit integrity constraints which are then translated into validation code in the wrappers [Thiran et al., 2006].

Another major problem is wrapper maintenance. Query translation relies heavily on the mappings between the component database schema and the local schema. If the component database schema is changed to reflect the evolution of the component database, then the mappings can become invalid. For instance, in Example 9.11, the administrator may switch the order of the fields in the EMP file. Using invalid mappings may prevent the wrapper from producing correct results. Since the component databases are autonomous, detecting and correcting invalid mappings is important. The techniques to do so are those for mapping maintenance that we presented in Chapter 4.

9.6 Conclusion

Query processing in multidatabase systems is significantly more complex than in tightly-integrated and homogeneous distributed DBMSs. In addition to being distributed, component databases may be autonomous, have different database languages and query processing capabilities, and exhibit varying behavior. In particular, component databases may range from full-fledged SQL databases to very simple data sources (e.g., text files).

In this chapter, we addressed these issues by extending and modifying the distributed query processing architecture presented in Chapter 6. Assuming the popular mediator/wrapper architecture, we isolated the three main layers by which a query is successively rewritten (to bear on local relations) and optimized by the mediator, and then translated and executed by the wrappers and component DBMSs. We also discussed how to support OLAP queries in a multidatabase, an important requirement of decision-support applications. This requires an additional layer of translation from OLAP multidimensional queries to relational queries. This layered architecture for multidatabase query processing is general enough to capture very different variations. This has been useful to describe various query processing techniques, typically designed with different objectives and assumptions.

The main techniques for multidatabase query processing are query rewriting using multidatabase views, multidatabase query optimization and execution, and query translation and execution. The techniques for query rewriting using multidatabase views differ in major ways depending on whether the GAV or LAV integration approach is used. Query rewriting in GAV is similar to data localization in homogeneous distributed database systems. But the techniques for LAV (and its extension GLAV) are much more involved and it is often not possible to find an equivalent rewriting for a query, in which case a query that produces a maximum subset of the answer is necessary. The techniques for multidatabase query optimization include cost modeling and query optimization for component databases with different computing capabilities. These techniques extend traditional distributed query processing by focusing on heterogeneity. Besides heterogeneity, an important problem is to deal with the dynamic behavior of the component DBMSs. Adaptive query processing addresses this problem with a dynamic approach whereby the query optimizer communicates at run time with the execution environment in order to react to unforeseen variations of runtime conditions. Finally, we discussed the techniques for translating queries for execution by the components DBMSs and for generating and managing wrappers.

The data model used by the mediator can be relational, object-oriented or even semi-structured (based on XML). In this chapter, for simplicity, we assumed a mediator with a relational model that is sufficient to explain the multidatabase query processing techniques. However, when dealing with data sources on the Web, a richer mediator model such as object-oriented or semi-structured (e.g., XML-based) may be preferred. This requires significant extensions to query processing techniques.

9.7 Bibliographic Notes

Work on multidatabase query processing started in the early 1980's with the first multidatabase systems (e.g., [Brill et al., 1984; Dayal and Hwang, 1984] and [Landers and Rosenberg, 1982]). The objective then was to access different databases within an organization. In the 1990's, the increasing use of the Web for accessing all kinds of data sources triggered renewed interest and much more work in multidatabase query processing, following the popular mediator/wrapper architecture [Wiederhold, 1992]. A brief overview of multidatabase query optimization issues can be found in [Meng et al., 1993]. Good discussions of multidatabase query processing can be found in [Lu et al., 1992, 1993], in Chapter 4 of [Yu and Meng, 1998] and in [Kossmann, 2000].

Query rewriting using views is surveyed in [Halevy, 2001]. In [Levy et al., 1995], the general problem of finding a rewriting using views is shown to be NP-complete in the number of views and the number of subgoals in the query The unfolding technique for rewriting a query expressed in Datalog in GAV was proposed in [Ullman, 1997]. The main techniques for query rewriting using views in LAV are the bucket algorithm [Levy et al., 1996b], the inverse rule algorithm [Duschka and Genesereth, 1997], and the MinCon algorithm [Pottinger and Levy, 2000].

The three main approaches for heterogeneous cost modeling are discussed in [Zhu and Larson, 1998]. The black-box approach is used in [Du et al., 1992; Zhu and Larson, 1994]. The customized approach is developped in [Zhu and Larson, 1996a; Roth et al., 1999; Naacke et al., 1999]. The dynamic approach is used in [Zhu et al., 2000], [Zhu et al., 2003] and [Rahal et al., 2004].

The algorithm we described to illustrate the query-based approach to heterogeneous query optimization has been proposed in [Du et al., 1995]. To illustrate the operator-based approach, we described the popular solution with planning functions proposed in the Garlic project [Haas et al., 1997a]. The operator-based approach has been also used in DISCO, a multidatabase system to access component databases over the web [Tomasic et al., 1996, 1998].

Adaptive query processing is surveyed in [Hellerstein et al., 2000; Gounaris et al., 2002b]. The seminal paper on the Eddy approach which we used to illustrate adaptive query processing is [Avnur and Hellerstein, 2000]. Other important techniques for adaptive query processing are query scrambling [Amsaleg et al., 1996a; Urhan et al., 1998a], Ripple joins [Haas and Hellerstein, 1999b], adaptive partitioning [Shah et al., 2003] and Cherry picking [Porto et al., 2003]. Major extensions to Eddy are state modules [Raman et al., 2003] and distributed Eddies [Tian and DeWitt, 2003b].

A software engineering solution to the problem of wrapper creation and maintenance, considering integrity control, is proposed in [Thiran et al., 2006].

Exercises

Problem 9.1 (**). Can any type of global optimization be performed on global queries in a multidatabase system? Discuss and formally specify the conditions under which such optimization would be possible.

Problem 9.2 (*). Consider a marketing application with a ROLAP server at site s_1 which needs to integrate information from two customer databases, each at site s_2 within the corporate network. Assume also that the application needs to combine customer information with information extracted from Web data sources about cities in 10 different countries. For security reasons, a web server at site s_3 is dedicated to Web access outside the corporate network. Propose a multidatabase system architecture with mediator and wrappers to support this application. Discuss and justify design choices.

Problem 9.3 (**). Consider the global relations EMP(ENAME, TITLE, CITY) and ASG(ENAME, PNAME, CITY, DUR). City in ASG is the location of the project of name PNAME (i.e., PNAME functionnally determines CITY). Consider the local relations EMP1(ENAME, TITLE, CITY), EMP2(ENAME, TITLE, CITY), PROJ1(PNAME, CITY), PROJ2(PNAME, CITY) and ASG1(ENAME, PNAME, DUR). Consider query Q which selects the names of the employees assigned to a project in Rio de Janeiro for more than 6 months and the duration of their assignment.

(a) Assuming the GAV approach, perform query rewriting.
(b) Assuming the LAV approach, perform query rewriting using the bucket algorithm.
(c) Same as (b) using the MinCon algorithm.

Problem 9.4 (*). Consider relations EMP and ASG of Example 9.7. We denote by $|R|$ the number of pages to store R on disk. Consider the following statistics about the data:

$|\text{EMP}| = 1\,000$
$|\text{EMP}| = 100$
$|\text{ASG}| = 10\,000$
$|\text{ASG}| = 2\,000$
$selectivity(\text{ASG.DUR} > 36) = 1\%$

The mediator generic cost model is:

$cost(\sigma_{A=v}(R)) = |R|$

$cost(\sigma(X)) = cost(X)$ where X contains at least one operator.

$cost(R \bowtie_A^{ind} S) = cost(R) + |R| * cost(\sigma_{A=v}(S))$ using an indexed join algorithm.

$cost(R \bowtie_A^{nl} S) = cost(R) + |R| * cost(S)$ using a nested loop join algorithm.

Consider the MDBMS input query Q:

```
SELECT *
FROM   EMP, ASG
WHERE  EMP.ENO=ASG.ENO
AND    ASG.DUR>36
```

Consider four plans to process Q:

$$P_1 = EMP \bowtie^{ind}_{ENO} \sigma_{DUR>36}(ASG)$$
$$P_2 = EMP \bowtie^{nl}_{ENO} \sigma_{DUR>36}(ASG)$$
$$P_3 = \sigma_{DUR>36}(ASG) \bowtie^{ind}_{ENO} EMP$$
$$P_4 = \sigma_{DUR>36}(ASG) \bowtie^{nl}_{ENO} EMP$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 9.5 (*). Consider relations EMP and ASG of the previous exercice. Suppose now that the mediator cost model is completed with the following cost information issued from the component DBMSs.

The cost of accessing EMP tuples at db_1 is:

$$cost(\sigma_{A=v}(R)) = |\sigma_{A=v}(R)|$$

The specific cost of selecting ASG tuples that have a given ENO at D_2 is:

$$cost(\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})) = |\sigma_{\mathrm{ENO}=v}(\mathrm{ASG})|$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 9.6 (**). What are the respective advantages and limitations of the query-based and operator-based approaches to heterogeneous query optimization from the points of view of query expressiveness, query performance, development cost of wrappers, system (mediator and wrappers) maintenance and evolution?

Problem 9.7 (**). Consider Example 9.8 by adding, at a new site, component database db_4 which stores relations EMP(ENO, ENAME, CITY) and ASG(ENO, PNAME, DUR). db_4 exports through its wrapper w_3 join and scan capabilities. Let us assume that there can be employees in db_1 with corresponding assignments in db_4 and employees in db_4 with corresponding assignments in db_2.

(a) Define the planning functions of wrapper w_3.
(b) Give the new definition of global view EMPASG(ENAME, CITY, PNAME, DUR).
(c) Give a QEP for the same query as in Example 9.8.

Problem 9.8 (**). Consider three relations $R(A,B)$, $S(B,C)$ and $T(C,D)$ and query Q $(\sigma^1_p(R) \bowtie_1 S \bowtie_2 T)$, where $\bowtie_1$ and $\bowtie_2$ are natural joins. Assume that S has an index

on attribute *B* and *T* has an index on attribute *C*. Furthermore, σ_p^1 is an expensive predicate (i.e., a predicate over the results of running a program over values of *R.A*). Using the Eddy approach for adaptive query processing, answer the following questions:

(a) Propose the set *C* of constraints on *Q* to produce an Eddy-based *QEP*.
(b) Give a query graph *G* for *Q*.
(c) Using *C* and *G*, propose an Eddy-based QEP.
(d) Propose a second QEP that uses State Modules. Discuss the advantages obtained by using state modules in this QEP.

Problem 9.9 (**). Propose a data structure to store tuples in the Eddy buffer pool to help choosing quickly the next tuple to be evaluated according to user specified preference, for instance, produce first results earlier.

Problem 9.10 (**). Propose a predicate scheduling algorithm based on the Cherry picking approach introduced in Section 9.4.3.3.