Chapter 15

DATABASE INTEROPERABILITY

Up to this point we have considered technical issues related to (homogeneous) distributed database systems. As we discussed in Chapter 4, 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 distributed multidatabase systems which provide interoperability among a set of DBMSs. This is only one part of the more general *interoperability* problem. In recent years, new distributed applications have started to pose new requirements regarding the data source(s) they access. In parallel, the management of "legacy systems" and reuse of the data they generate have gained importance. The result has been a renewed consideration of the broader question of information system interoperability. The focus of this chapter is the narrower problem of database interoperability; however, we consider the more general question in the next chapter within the context of the World Wide Web and data management.

15.1 DATABASE INTEGRATION

Database integration involves the process by which information from participating databases can be conceptually integrated to form a single cohesive definition of a multidatabase. In other words, it is the process of designing the global conceptual schema. Recall from Chapter 4 that not all multidatabase architectures actually require the definition of this integrated view. Thus the discussions in this section are relevant only for those architectures that specify a global conceptual schema.

Recall from Chapter 5 that the design process in multidatabase systems is bottom-up. In other words, the individual databases actually exist, and designing the global conceptual schema involves integrating these component databases into a multidatabase. Database integration can occur in two steps (Figure 15.1): *schema translation* (or simply translation) and *schema integration*. In the first step, the component database schemas are translated to a common intermediate (InS_1, InS_2, . . . , InS_n) canonical representation. The use of a canonical representation

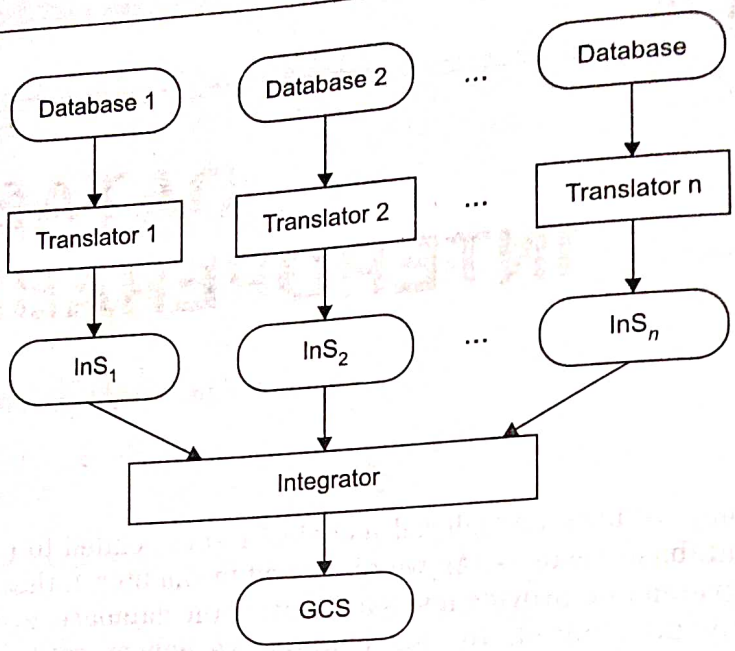


Figure 15.1. Database Integration Process

facilitates the translation process by reducing the number of translators that need to be written. The choice of the canonical model is important. As a principle, it should be one that is sufficiently expressive to incorporate the concepts available in all the databases that will later be integrated. Most of the recent studies use an object-oriented model for this purpose. This is necessary if one of the component databases that will be integrated is object-oriented. Even when this is not the case, an object model is generally viewed as the most appropriate canonical model. In this section, we will refrain from introducing object-orientation; we will discuss the role of object models and the object-oriented approach to interoperability in Section 15.4. For the time being, we will take a more traditional approach.

Clearly, the translation step is necessary only if the component databases are heterogeneous and each local schema may be defined using a different data model. In recent years, commercial interest has been on the integration of multiple relational databases where this translation step can be bypassed. There is some recent work on the development of system federation, in which systems with similar data models are integrated together (e.g., relational systems are integrated into one conceptual schema and, perhaps, object databases are integrated to another schema) and these integrated schemas are "combined" at a later stage (e.g., AURORA project [Yan, 1997], [Yan et al., 1997]). In this case, the translation step is delayed, providing increased flexibility for applications to access underlying data sources in a manner that is suitable for their needs.

In the second step, each intermediate schema is integrated into a global conceptual schema. In some methodologies, local external schemas are considered for integration rather than local conceptual schemas, since it may not be desirable to incorporate the entire local conceptual schema in the multidatabase.

Example 15.1

To facilitate our discussion of global schema design in multidatabase systems, we will use an example that is an extension of the engineering database we have been using throughout the book. To demonstrate both phases of the database integration process, we introduce some data model heterogeneity into our example.

Consider two organizations, each with their own database definitions. One is the (relational) database example that we have developed in Chapter 2. We repeat that definition in Figure 15.2 for completeness. The underscored attributes are the keys of the associated relations. We have made on modification in the PROJ relation by including attributes LOC and CNAME. LOC is the location of the project, whereas CNAME is the name of the client for whom the project is carried out. The second database also defined similar data, but is specified according to the entity-relationship (E-R) data model [Chen, 1976] as depicted in Figure 15.3.

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC, CNAME)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Figure 15.2. Relational Engineering Database Representation

We assume that the reader is familiar with the entity relationship data model. Therefore, we will not describe the formalism, except to make the following points regarding the semantics of Figure 15.3. This database is similar to the relational engineering database definition of Figure 15.2, with one significant difference: it also maintains data about the clients for whom the projects are conducted. The rectangular boxes in Figure. 15.3 represent, the entities modeled in the database, and the diamonds indicate a relationship between the entities to which they are connected. The type of relationship is indicated around the diamonds. For example, the CONTRACTED-BY relation is a many-to-one from the PROJECT entity to the CLIENT entity (e.g., each project has a single client, but each client can be many projects). Similarly, the WORKS-IN relationship indicates a many-to-many relationship between the two connected relations. The attributes of entities and the relationships are shown as elliptical circles.

15.1.1 Schema Translation

Schema translation is the task of mapping from one schema to another. This requires the specification of a target data model for the global conceptual schema definition. Schema translation may not be necessary in a heterogeneous database

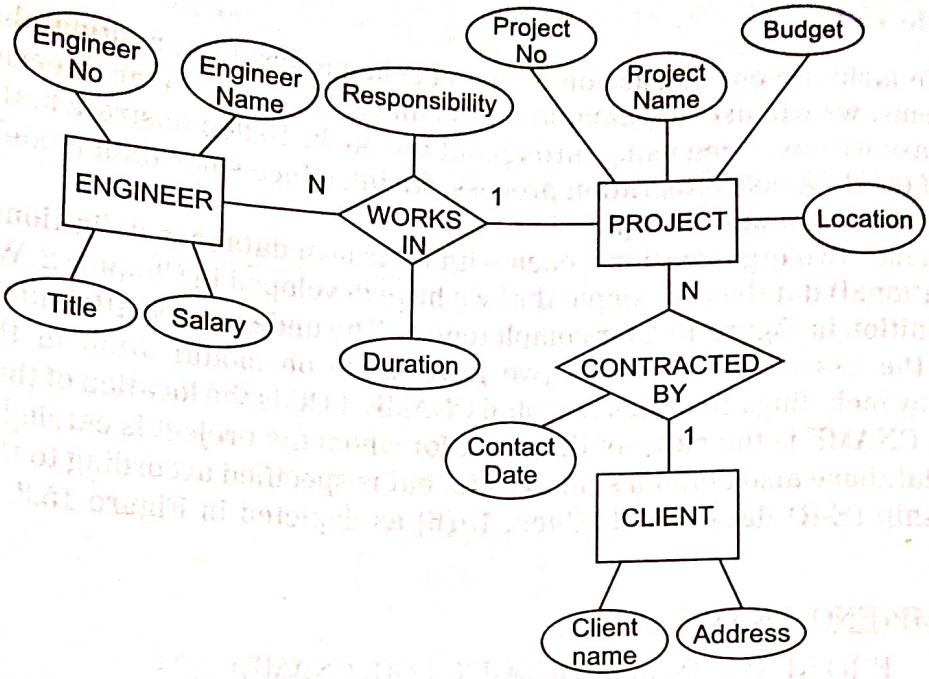


Figure 15.3. Entity-Relationship Database

if it can be accomplished during the integration stage. Combining the translation and integration steps [Brzezinski et al., 1984] provides the integrater with all the information about the entire global database at one time. Obviously, the integrater can make trade-offs between the different local schemas to determine which representation should be given precedence when conflicts arise. This requires that the integrater have knowledge of all the various trade-offs that must be made among several different schemas and their semantics, which may be different.

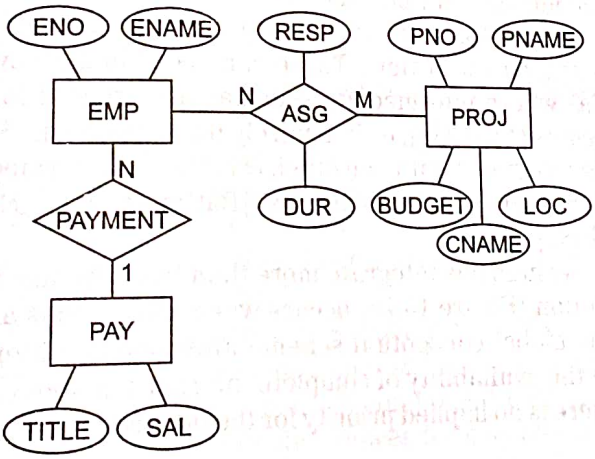
We will not discuss the specifics of translation between various data models; this can be found in many textbooks. The important point is that equivalences must be established between the concepts of the source model and those of the target model. The following example simply demonstrates the result of translation for the employee/engineering databases.

Example 15.2

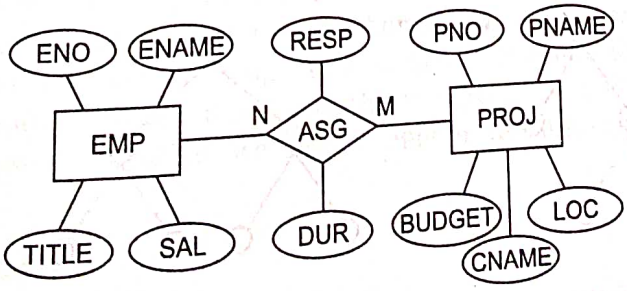
Among the two databases, the E-R model is more expressive; therefore we will use it as the canonical model. Translation of relational schemes to an E-R model requires consideration of each relation's role. The first difficulty is the determination of relations that represent entities versus those that represent relationships. This information may be easy to identify if there are specific relations that represent relationships as well as entities. Otherwise, these relationships may be identified from the foreign keys defined for each relation. Once this determination is made, the mapping is straightforward: relations that represent entities are modeled as entities, and relations that represent relationships are modeled as relationships.

A second difficulty relates to the nature of the relationships. Identification of the type of relationship (e.g., many-to-many) and relationship constraints require that semantic information be known about the relational implementation, since these are not intrinsic to the relational model. This typically requires consulting the system directory.

The relational model of the engineering database depicted in Figure 15.2 consists of four relations, three of which (EMP, PROJ and PAY) clearly correspond to entities, whereas one (ASG) corresponds to a relationship. The ENO and PNO attributes of ASG are foreign keys, which indicates that ASG is a relationship between EMP and PROJ. The type of relationship that ASG represents cannot be discerned from the relational schema definition. From our knowledge of the semantics of the database, we know it to be many-to-many. The handling of the PAY relation is more difficult. It can be treated as an entity, in which case it is necessary to establish a relationship between it and one of the other entities, probably EMP. Even though no such relation exists in Figure 15.2, it is possible to create a one-to-many relationship from PAY to EMP. The relationship needs to be one-to-many since each employee can have one salary, but a salary can belong to two employees who happen to have the same title. This is the PAYMENT relation in Figure 15.4a. Another alternative would be to treat salary as an attribute of an engineer entity (Figure 15.4b). This provides a cleaner E-R model but does not explicitly specify the relationship between the employee titles and their salaries.



(a)



(b)

Figure 15.4. E-R Equivalent of the Relational Engineering Databse

15.1.2 Schema Integration

Schema integration follows the translation process and generates the global conceptual schema by integrating the intermediate schemas. Schema integration is the process of *identifying* the components of a database which are related to one another, *selecting* the best representation for the global conceptual schema, and finally, *integrating* the components of each intermediate schema. Two components can be related as equivalent, one contained in the other, or as disjoint [Sheth et al., 1988a].

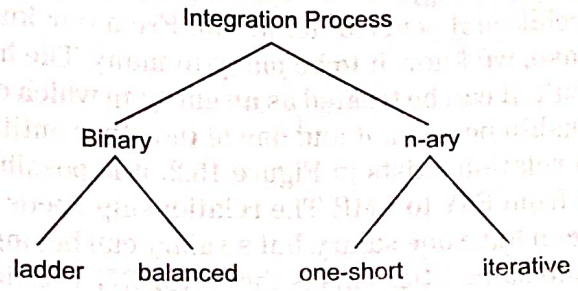


Figure 15.5. Taxonomy of Integration Methodologies

Integration methodologies can be classified as binary or *n*ary mechanisms [Batini et al., 1986] (Figure 15.5). Binary integration methodologies involve the manipulation of two schemas at a time. These can occur in a stepwise (ladder) fashion (Figure 15.6a) where intermediate schemas are created for integration with subsequent schemas [Pu, 1988], or in a purely binary fashion (Figure 15.6b), where each schema is integrated with one other, creating an intermediate schema for integration with other intermediate schemas ([Batini and Lenzirini, 1984] and [Dayal and Hwang, 1984]).

*N*ary integration mechanisms integrate more than two schemas at each iteration. One-pass integration (Figure 15.7a) occurs when all schemas are integrated at once, producing the global conceptual schema after one iteration. Benefits of this approach include the availability of complete information about all databases of integration time. There is no implied priority for the integration order of schemas,

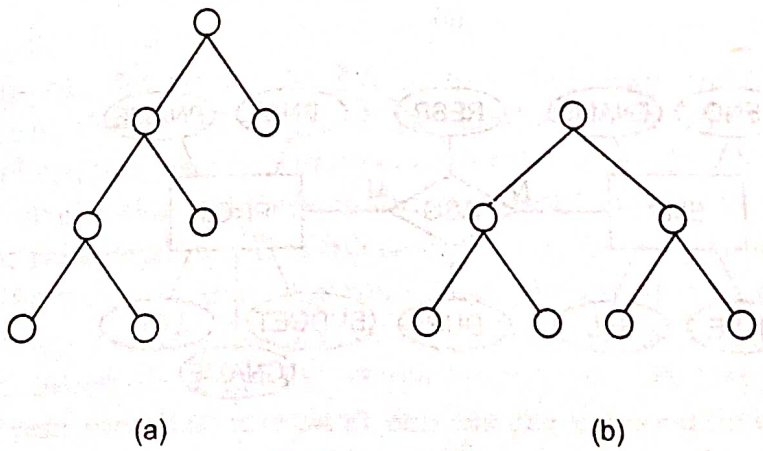


Figure 15.7. Binary Integration Methods

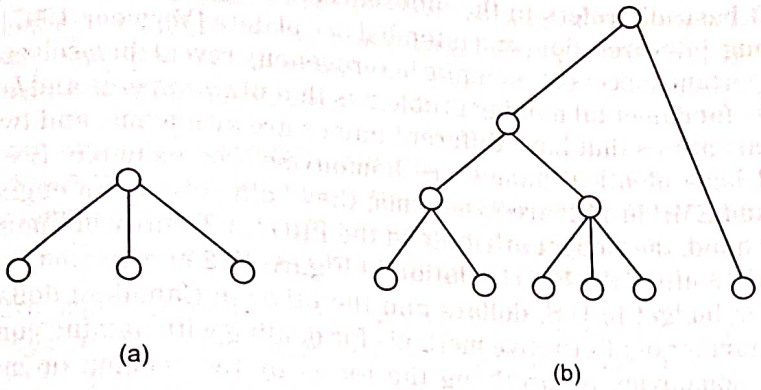


Figure 15.7. *N*ary Integration Methods

and the trade-offs, such as the best representation for data items or the most understandable structure, can be made between all schemas rather than between a few. Difficulties with this approach include increased complexity and difficulty of automation.

Iterative *n*ary integration (Figure 15.7b) offers more flexibility (typically more information is available) and is more general (the number of schemas can be varied depending on the integrater's preferences). Binary approaches are a special case of iterative *n*ary. They decrease the potential integration complexity and lead toward automation techniques, since the number of schemas to be considered at each step is more manageable. Integration by an *n*ary process enables the integrater to perform the operations on more than two schemas. For practical reasons, the majority of systems utilize binary methodology, but a number of researchers prefer the one-shot approach because complete information is available ([Elmasri et al., 1987] and [Yao et al., 1982b]). Tools have been developed to aid in this integration process (e.g., [Sheth et al., 1988a]).

Schema integration involves two tasks: homogenization and integration ([Yan et al., 1997] and [Yan, 1997]). *Homogenization* involves the determination of structural and semantic "problems" of each component database. These "problems" relate to structural and semantic differences from some norm. The norm could be a particular domain-specific ontology or the schema of another database. The idea is to make sure that component databases are comparable with each other in both structure and semantics once they are homogenized. *Integration* follows homogenization and involves merging the schemas of multiple databases to create a global conceptual schema. It is during this integration step that one of the methods discussed above is applied. We discuss each of these tasks below.

Homogenization

During this phase, both semantic heterogeneity and structural heterogeneity problems are resolved. semantic heterogeneity is a fairly loaded term without a clear

definition. It basically refers to the differences among the databases that relate to the meaning, interpretation, and intended use of data [Vermeer, 1997]. Arguably, the more important aspects of semantic heterogeneity reveal themselves as naming conflicts. The fundamental naming problem is that of *synonyms* and *homonyms*. Two identical entities that have different names are synonyms, and two different entities that have identical names are homonyms. For example, ENGINEER in figure 15.3 and EMP in 15.2 are synonyms; they both refer to an engineer entity. On the other hand, the Budget attribute in the PROJECT entity in Figure 15.3 and the Budget attribute of the PROJ relation in Figure 15.2 may be homonyms if one represents the budget in U.S. dollars and the other in Canadian dollars.
There are a number of alternative methods for dealing with naming conflicts. One is to resolve homonyms by prefixing the terms by the schema or model name [Elmasri et al., 1987]. It is not possible to resolve synonyms in a similar, simple fashion. The approach which is regarded as preferable is to use ontologies. An ontology is specific to a particular application domain and defines the terms, together with the semantics of those terms, that are acceptable in that domain. If every database schema for that domain uses a common ontology, then the naming conflicts are naturally resolved. There is significant effort underway in defining ontologies for various domains, but work is far from being complete.

Example 15.3

In the example that we are considering, integration will be performed on intermediate schemas in E-R notation. The intermediate schemas we will consider are depicted in Figures 15.3 and 15.4(b), which we will refer to as InS_1 and InS_2.

The synonyms between InS_1 and InS_2 are depicted in Figure 15.8, where the corresponding entries on the same row are synonyms (e.g., Salary and SAL). The only homonym is the title attribute that exists in both of the intermediate schemas. In InS_1, the attribute refers to the title of engineers,

InS_1	InS_2
ENGINEER	EMP
Engineer No	ENO
Engineer Name	ENAME
Salary	SAL
WORKS IN	ASG
Responsibility	RESP
Duration	DUR
PROJECTS	PROJ
Project No	PNO
Project Name	PNAME
Location	LOC

Figure 15.8. Synonyms in the Intermediate Schemas

so its domain is engineering titles, whereas in InS_2 it refers to the titles of all employees and therefore has a larger domain. Thus the title attributes in InS_1, and InS_2 form a homonym, since the same attribute name is used to mean two different things.

The naming conflicts in this example will be resolved by renaming entities, attributes, and relationships in the schemas. For simplicity, we will rename the schema of Figure 15.1 to conform to the gaining of Figure 15.3. We will rename the homonym TITLE attribute in the manner described above.

Structural conflicts occur in four possible ways: *as type conflicts, dependency conflicts*, *key conflicts*, or *behavioral conflicts* [Batini et al., 1986]. Type confilicts occur when the same object is represented by an attribute in one schema and by an entity in another. Dependency conflicts occur when different relationship modes (e.g., one-to-one versus many-to-many) are used to represent the same-thing in different schemas. Key conflicts occur when different candidate keys are available and different primary keys are selected in different schemas. Behavioral conflicts are implied by the modeling mechanism. For example, deleting the last item from one database may cause the deletion of the containing entity (i.e., deletion pf the last employee causes the dissolution of the department).

Example 15.4

We have two structural conflicts in the example we are considering. The first is a type conflict involving clients of projects. In the schema of Figure 15.3, the client of a project is modeled as an entity. In the schema of Figure 15.4, however, the client is included as an attributes of the PROJ entity.

The second structural conflict is a dependency conflict involving the WORKS_IN relationship in Figure 15.3 and the ASG relationship in Figure 15.4. In the former, the relationship is many-to-one from the ENGINEER to the PROJECT, whereas in the latter, the relationship is may-to-many. The resolution of these conflicts is discussed in Example 15.5.

Transformation of entities/attributes/relationships among each another is one way of handling structural conflicts ([Batini and Lenzirini, 1984], [Batini et al., 1986]). One can accomplish these transformations on an instance-by-instance basis. Figure 15.9 depicts the possible atomic transformation scenarios. The dashed lines indicate that a given attribute is an identifier (key) of the associated entity.

A non-key attribute can be transformed into an entity by creating an intermediate relationship connecting the new entity and a new attributes to represent it. Figure 15.9a depicts such a transformation of a non-key attribute A of entity E to separate entity that is related to E by a many-to-many relationship and is uniquely identified by a new key attribute, C. Figure 15.9b illustrates a key attribute translation where a key attribute is transformed into an entity that has an identifier C. C becomes the identifier of both the new entity A and the entity E, because the relationship between E and A is many-to-one. Figure 15.9c demonstrates the

case where identifier A is only a part of the complete identifier, which requires the non-standard reference back to the originating entity.

Example 15.5

In the example we are considering, there is one case where such a transformation would be necessary. In Figure 15.4, the attribute CHAIR is represented as an attribute and needs to be converted to an entity using the technique demonstrated in Figure 15.3a. The result is depicted in Figure 15.10.

Recall that there is a dependency conflict between the two schemas as well. In this example, we will resolve the conflict by choosing to accept the more general many-to-many relationship between the EMPLOYEE and PROJECT entities. Note that this is a design decision which affects alternative semantics of integration, and results in the loss of the more restrictive one-to-many constraint between EMPLOYEE and PROJECT.

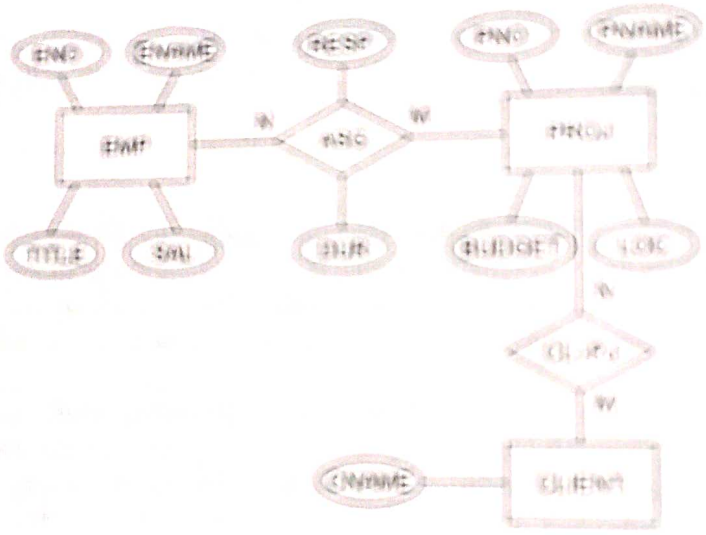


Figure 15.10 Attribute-to-Entity Transformation

The determination of synonyms and homonyms, as well as the identification of structural conflicts, requires specification of the relationship between the intermediate schemas. Two schemas can be related in four possible ways: they can be identical to one another, one can be a subset of the other, some components from one may occur in the other while retaining some unique features, or they could be completely different with no overlap. In the example that we have been considering, $fm.S_2$ is a subset of $fm.S_1$.

Integration

Integration involves the merging of the intermediate schemas and their restructuring. All schemas are merged into a single database schema and then restructured

case where identifier A is only a part of the complete identifier, which requires the non-standard reference back to the originating entity.

Example 15.5

In the example we are considering, there is one case where such a transformation would be necessary. In Figure 15.4, the attribute CNAME is represented as an attribute and needs to be converted to an entity using the technique demonstrated in Figure 15.9a. The result is depicted in Figure 15.10. Recall that there is a dependency conflict between the two schemas as well. In this example, we will resolve the conflict by choosing to accept the more general many-to-many relationship between the ENGINEER and PROJECT entities. Note that this is a design decision which reflects alternative semantics of integration, and results in the loss of the more restricting one-to-many constraint between ENGINEER and PROJECT.

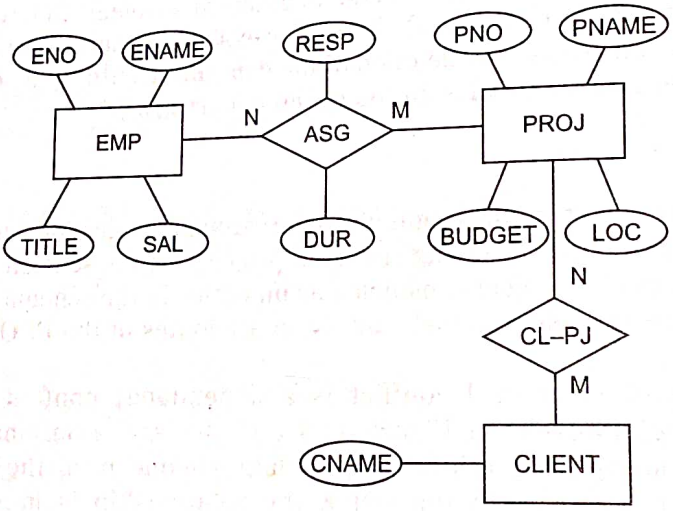


Figure 15.10. Attribute-to-Entity Transformation

The determination of synonyms and homonyms, as well as the identification of structural conflicts, requires specification of the relationship between the intermediate schemas. Two schemas can be related in four possible ways: they can be identical to one another, one can be a subset of the other, some components from one may occur in the other while retaining some unique features, or they could be completely different with no overlap. In the example that we have been considering, InS_2 is a subset of InS_1.

Integration

Integration involves the merging of the intermediate schemas and their restructuring. All schemas are merged into a single database schema and then restructured

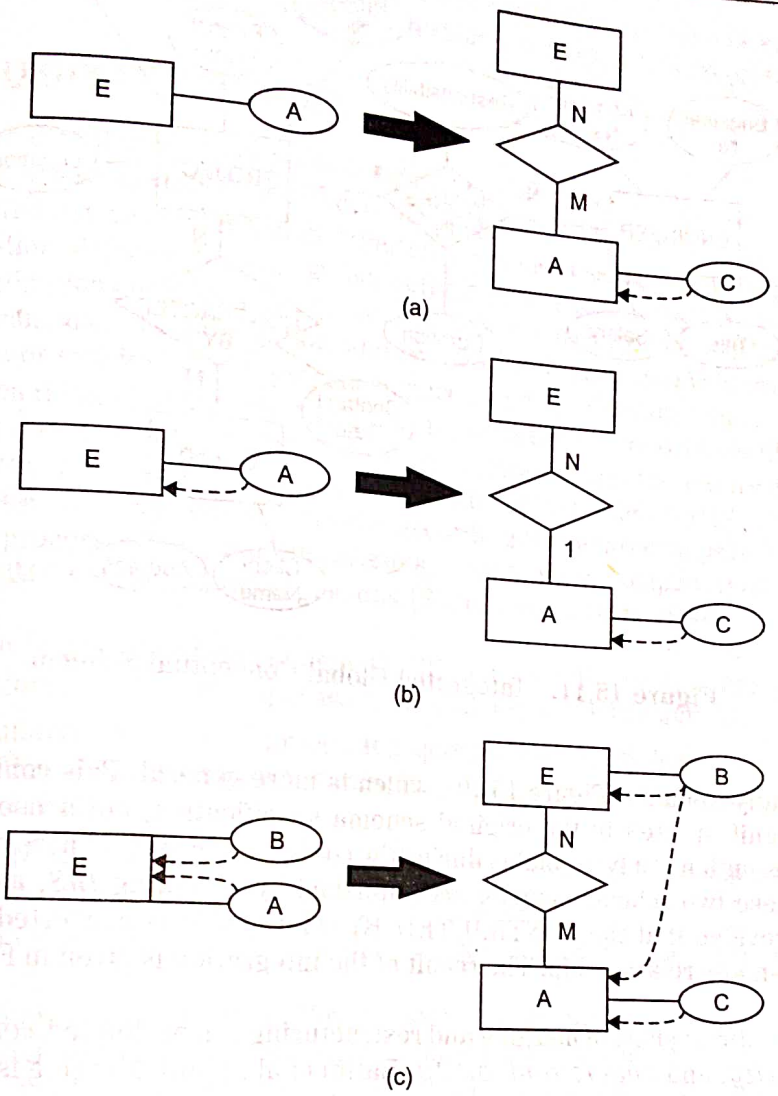


Figure 15.9. Atomic Conformation Alternatives (Adapted from: C. Batini, M. Isenzerini, and S.B. Navathe, Comparison of Methodologies for Database Scheme integration. *ACM Comp. Surveys*; December 1986; 18(4): 323–364.)

to create the "best" integrated schema. Merging requires that the information contained in the participating schemas be retained in the integrated schema.

Example 15.6

For this example, the integration step is straightforward. Since InS_2, is a subset of InS_1, we accept InS_2 as the integrated schema. The only compli- cation arises due to the different characteristics of the CONTRACTED_BY relationship in the two schemas. In InS_1, it is many-to-one from the PROJECT entity to the CLIENT entity. However, the same relationship in InS_2 is

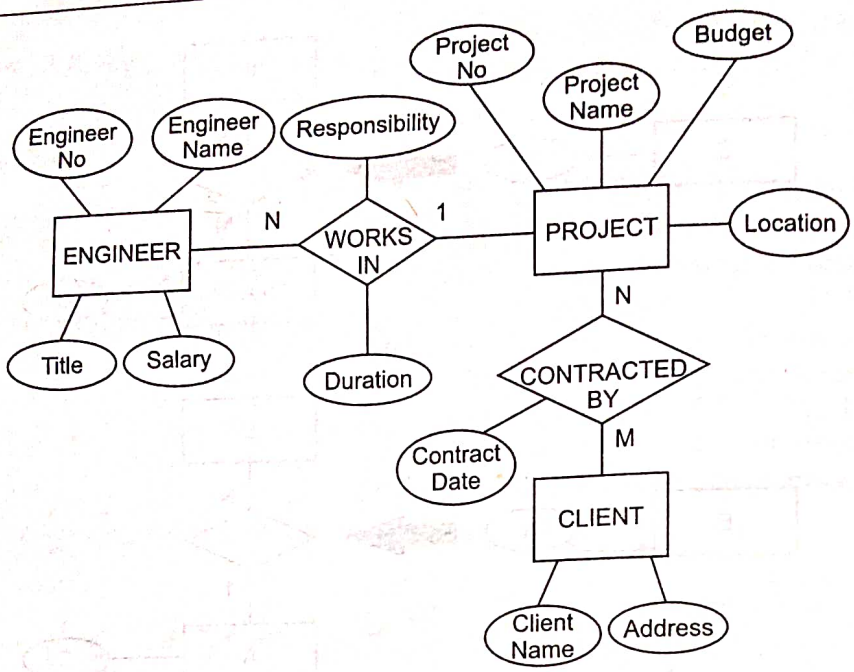


Figure 15.11. Integrated Global Conceptual Schema

many-to-many (Figure 15.10), which is more general. This conflict did not manifest itself in the original schema specification, but arose due to the resolution of a type conflict during the confirmation stage. Therefore, merging these two schemas can be accomplished by accepting InS_1 as the result, provided that the CONTRACTED_BY relationship is converted to a many-to-many relationship. The result of the integration is given in Figure 15.11.

Three dimensions of merging and restructuring can be defined: *completeness*, *minimatity*, and *understandability* [Batini et al., 1986]. Merging is *complete* if all the information from all the schemas is integrated into the common schema. To accomplish a complete merging, one may use *subsetting*, a technique that describes one entity in terms of another. The well-known concepts of generalization and specialization are special cases of subsetting. It is possible to devise special operators for this purpose [Motro and Buneman, 1981].

A merging is *non-minimal* when redundant relationship information is retained in an integrated schema because of a failure to detect containment where part of one intermediate schema may be included within another intermediate schema. Non-minimal schemas can also result from the translation process, due to the production of an intermediate schema which itself is not minimal.

Understandabitity is the final dimension for determining the best schema. Once all the elements are merged, the restructuring should facilitate an understandable schema. Unfortunately, quantifying exactly what makes something easily understandable is usually not possible, since the concept itself is highly subjective.

It may be necessary to make trade-offs between minimality and understandability, provided that the resulting merged and restructured schema is complete.

15.2 QUERY PROCESSING

Many of the distributed query processing and optimization techniques carry over to multidatabase systems, but there are important differences. Recall from Chapter 7 that we characterized distributed query processing in four steps: query decomposition, data localization, global optimization, and local optimization. This is a generalization of the local query processing steps in centralized DBMSs, which include decomposition, optimization, and execution [Gardarin and Valduriez, 1989). The nature of multidatabase systems requires slightly different steps.

The first thing to remember in this discussion is the nature of the multi-DBMS. In Chapter 4 (specifically in Figure 4.10), we indicated that the multi-DBMS is a layer of software that runs on top of component DBMSs. Each DBMS has its own query processors, which execute queries according to the three steps listed above.

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

1. The capability of component DBMSs may be different, which prevents uniform treatment of queries across multiple DBMSs and sites.
2. Similarly, the cost of processing queries may be different on different DBMSs. This increases the complexity of the cost functions that need to be evaluated.
3. There may be difficulties in moving data between DBMSs, since they may differ in their ability to read "moved" data.
4. The local optimization capability of each DBMS may be quite different.

In addition, the autonomy of these systems poses problems [Lu et al, 1993]. Communication autonomy means that a component DBMS 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 where a component system is either unavailable from the beginning or shuts down in the middle of query execution. There has not been much work in this area. Design autonomy may restrict the availability and accuracy of statistical information that is needed for query optimization. The difficulty of determining local cost functions is an issue that we will discuss shortly. The execution autonomy of multidatabase systems makes it difficult to apply some of the query optimization strategies. 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.

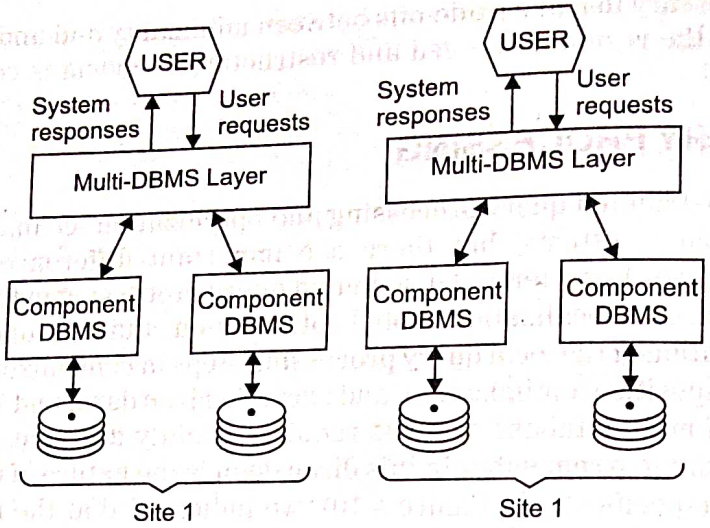


Figure 15.12. Structure of a Distributed Multi-DBMS

15.2.1 Query Processing Layers in Distributed Multi-DBMSs

With this structure in mind, we can now discuss the various steps involved in query processing in distributed multi-DBMSs (Figure 15.13). When a query is received at a site, the first thing that needs to be done is to "split" it into subqueries based on data distribution across multiple sites. At this step, it is only necessary to worry about the placement of data across the sites, rather than its storage across various databases. Therefore, the only information that is required is the typical data allocation information stored in a global directory. The site that receives the query and performs the splitting, called the *control site*, is ultimately responsible for successful completion of the task.

Each subquery is then sent to the site where it is to be processed. The multi-DBMS layer at each site further "fragments" the query for each DBMS that it controls. At this stage, the information within the directory is used. Each subquery is then translated into the language of the respective DBMS. Extensive information about the global query language and the individual languages used by the DBMSs needs to be maintained to facilitate translation. Even though this information can be kept within the directory, it is common to store it as an *auxiliary database* [Landers and Rosenberg, 1982].

The queries submitted to the component DBMSs are processed following decomposition, optimization, and execution steps. The decomposition step involves the simplification of a user query that is specified in some relational calculus and its translation to an equivalent relational algebra query over the conceptual schema. The optimization step involves the reordering of relational algebra operations, as well as determination of the best access paths to data. The resulting schedule is then executed by the run-time support processor.

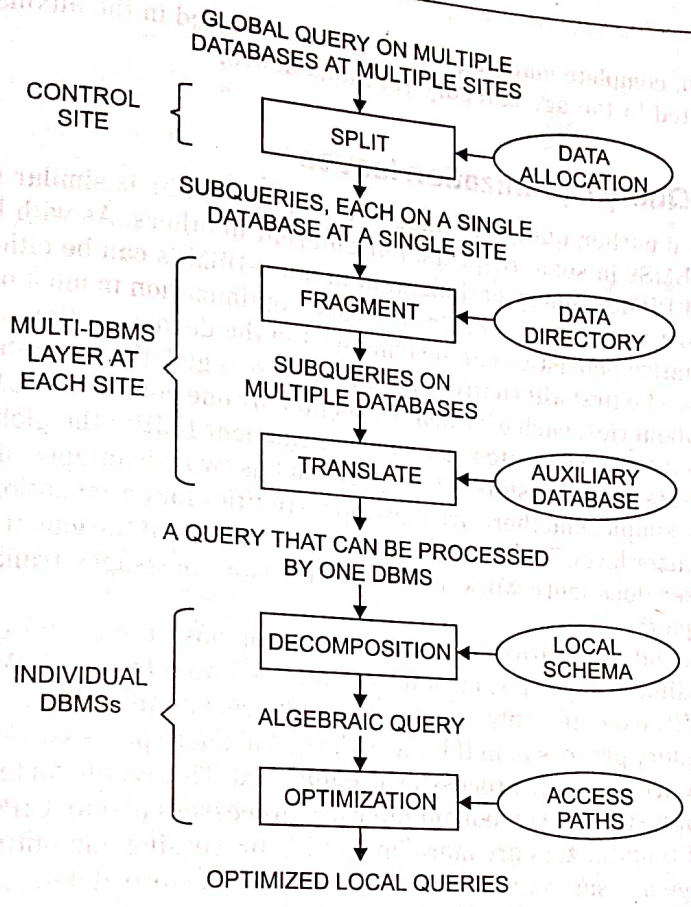


Figure 15.13. Query Processing Steps in Multidatabase Systems

As indicated above, specific translation information is stored in a separate auxiliary database. There is no overriding principle that dictates separation of the global directory from the auxiliary database. In fact, there are prototype heterogeneous systems (such as OMNIBASE [Rusinkiewicz et al., 1988], COSYS [Adiba and Portal, 1978], ADDS [Breitbart and Paolini, 1985], and MRDSM [Wong and Bazek, 1985]) that combine the two pieces of information into one database. We have separated them to highlight their different functionalities and to facilitate the incremental definition of the two databases. This separation serves to emphasize the distinction between distributed databases and distributed multidatabase systems.

The auxiliary database contains information describing how mappings from/to participating 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 auxiliary database 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 located in the auxiliary database, as illustrated in the age category relations above.

15.2.2 Query Optimization Issues

As indicated earlier, query optimization in multi-DBMSs is similar to that of distributed DBMSs in some respects, but different in others. As with homogeneous distributed DBMSs, query optimization in multi-DBMSs can be either heuristics-based or cost-based. We consider cost-based optimization in most of this section. Two alternative heuristics can be employed in the decomposition of a query into subqueries. The first alternative is to decompose a global query into the smallest possible subqueries, each of which is executed by one component DMBS. Multiple subqueries may be submitted to a given component DMBS; the global query processor collects partial results. This approach has two advantages: decomposition is relatively simple, and there are more opportunities for optimization at the global query optimizer level. The disadvantage, of course, is that the global query processor/optimizer does more work, and there are more messages transmitted to execute the query.

The second alternative heuristic is to decompose the global query into the largest possible subqueries, each of which is executed by one DBMS. Each component DMBS executes only one subquery, the results of which are collected by the global query processor. In this case, the global query processor/optimizer does less work, since inter-site processing is minimized. This results in fewer messages but more sophisticated component interface processors (CIPs). CIPs and the component DMBS optimizers are more involved in processing and optimization. CIPs can exchange messages among themselves, and can store, delete, and operate on temporary files.

If we follow our earlier characterization of cost-based query optimization in terms of search space, search algorithm, and cost functions, the differences are mainly in terms of cost functions. The absence of (or difficulty in obtaining) information about component DBMSs' operations poses problems in defining meaningful global cost function. Why is it necessary to define global cost functions? The main reason is that global queries involve inter-site operations, which require optimization at the multi-DBMS level. Of particular concern, of course, is the optimization of inter-DBMS joins. We consider that problem first, and then consider the issue of defining appropriate cost functions and obtaining information from the component DBMSs.

As discussed in the chapters on query optimization, most commercial cost-based 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 15.14a). 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 multi-DBMSs, these types of join execution plans are not necessarily the preferred

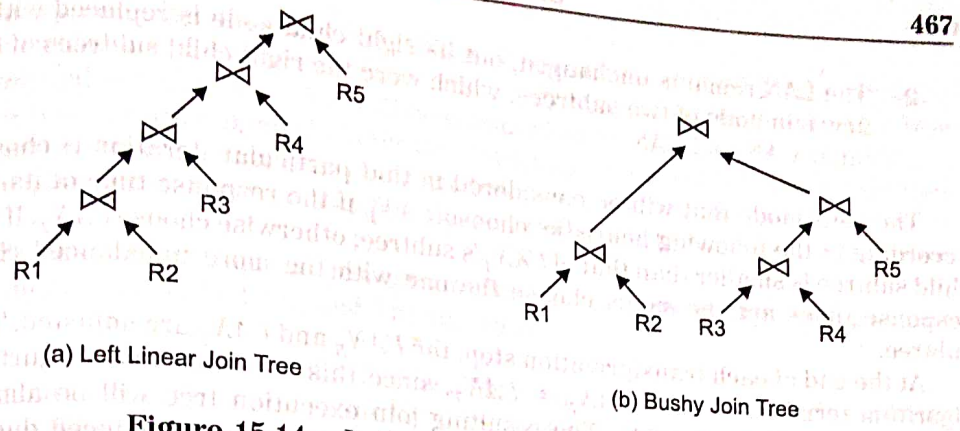


Figure 15.14. Left Linear versus Bushy Join Trees

ones, as they do not allow any parallelism in join execution — an intermediate join has to wait for the completion of the earlier join operation (See Figure 15.14b). This is particularly serious if the intermediate join is implemented using a sort-merge or hash join algorithm since, in this case, the intermediate join cannot even begin before the earlier one completes [Du et al., 1995]. Strictly speaking, this is also a problem in tightly coupled distributed DBMSs, but the issue is more serious in the case of multi-DBMSs, because in the latter systems, 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 commercial query optimizer to first generate a left linear join tree, and then convert it to a bushy one [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. Du et al [1995] propose 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. The algorithm maintains two pointers, called *upper anchor nodes* (UAN) on the tree. At the beginning, bottom UAN (UAN_B) is set to the grandparent of the leftmost root node (join 2 in Figure 15.14a), while top UAN (UAN_T) is set to the root (join 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.

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, developed specifically for the MDBS, can take these into consideration. Evrendilek et al [1997] propose such an algorithm, which generates an initial join execution graph. The algorithm proposes 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 [Evrendilek et al., 1997].

If optimizers are extensible, it is possible to extend a commercial optimizer rather than implementing one from scratch for the MDBS, or revising the output generated by one. This is done in the Garlic system [Haas et al., 1997], where a rule-based optimizer is extended with execution plans that are specific to the multidatabase environment. Garlic starts from the rule-based optimizer proposed by Lohman [1988] and extends the rules for new operators to create temporary relations to retrieve locally-stored data. It also creates the `PushDown` operator that pushes a portion of the work to the component DBMSs where they will be executed. The execution plans are represented, as usual, with 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.

Besides execution space considerations, the definition of a global cost function is a major issue. Global cost function definition, and the associated problem of obtaining cost-related information from component DBMSs, is perhaps the most-studied problem. A number of possible solutions have emerged, which we will 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 it 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. Treat the component DBMS as a black box, run some test queries on them, and from these determine the necessary cost information [Du et al., 1992], [Zhu and Larson, 1998].
2. Use previous knowledge about the component DBMSs, as well as their external characteristics, to subjectively determine the cost information [Zhu and Larson, 1998].
3. Monitor the run-time behavior of component DBMSs and dynamically collect the cost information [Lu et al., 1992].

Among these approaches, the first has attracted the most attention. We will discuss two proposals that follow this approach.

The Pegasus project [Du et al., 1992] addresses this problem by expressing the cost functions 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 is expressed as

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

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 that has been noted 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, 1998]. An alternative is proposed in the CORDS project [Zhu and Larson, 1996a], 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 are sample queries. In this case [Zhu and Larson, 1998], 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 underlying component DBMSs (e.g., the access methods that it supports). 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 [Zhu and Larson, 1996b] using the measured costs of sample queries. The regression coefficients are the cost function parameters.

15.3 TRANSACTION MANAGEMENT

Among all database interoperability problems, transaction management has probably been studied the most extensively. The challenge is to permit concurrent global updates to the component databases without violating their autonomy. In general, it is not possible to provide the same semantics as (homogeneous) distributed DBMSs without violating some autonomy.

Execution autonomy implies that the global transaction management functions are performed independent of the component transaction execution functions. In other words, the individual component DBMSs (more specifically, their transaction managers) are not modified to accommodate global updates. Design autonomy has the additional implication that the transaction managers of each DBMS may employ different concurrency control and commit protocols.

In this section, we describe the specific problems and review the existing work in the literature. First we discuss a transaction and computation model, and then we present an extension to the serializability theory that accommodates multidatabase systems.

15.3.1 Transaction and Computation Model

Let us first elaborate on the architectural aspects of multidatabase transaction processing. As described in Chapter 4 (specifically in Figure 4.10), the MDBS architecture involves a number of DBMSs, each with its own transaction manager (called local transaction managers or LTMs) and a multi-DBMS layer on top. The transaction manager of the multi-DBMS layer is called the global transaction manager (GTM) since it manages the execution of global transactions. Further, in a distributed multi-DBMS, the architecture of Figure 4.10 exists at each site. Thus our architectural model can be further abstracted, as in Figure 15.15, for the purposes of distributed transaction management.

In a multidatabase system, there are two types of transaction: local transac-

tions, which are submitted to each DBMS, and global transactions, which are submitted to the multi-DBMS layer. Local transactions execute on a single database, whereas global transactions access multiple databases. A global transaction is divided into a set of global subtransactions, each of which executes on one database. For a global transaction GT_i, its global subtransaction, which executes on database j, will be denoted as GST_{ij}. For a distributed transaction GT_i (which, by definition, must be global), the global subtransaction that executes at site k is denoted as GST_i^k.

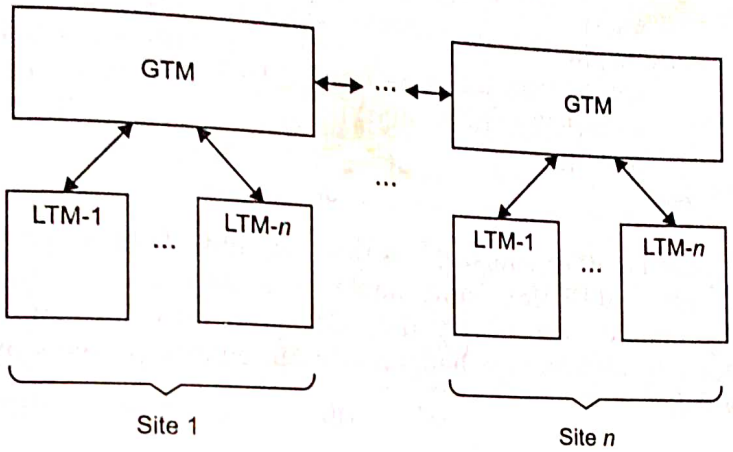


Figure 15.15. Distributed Multi-DBMS Transaction Management

Example 15.7

Consider the two databases that we designed in Section 15.1 (Figures 15.2 and 15.3). In this example, we ignore distribution for simplicity and without loss of generality. Let us denote the relational engineering database as 1, and the E-R engineering database as 2, and assume that these databases reside at the same site. Assume that a global transaction updates the salary of "J. Doe" by 15%. Let us denote this transaction as GT_1. First note that GT_1 may be specified on the global conceptual schema (if one is defined) which is specified in Example 14.11 (Figure 15.11). This global transaction will be subdivided into two subtransactions, as specified below, each executing on one of the databases. For the relational DBMS, we use an embedded SQL notation to specify the transaction; for the E-R DBMS, we use a straightforward algorithmic notation.

```
GST11:  EXEC SQL          SELECT TITLE INTO temp1
                          FROM EMP
                          WHERE ENAME = "J. Doe"
        if temp1 is empty then
            abort
```

```
            else begin
                EXEC SQL        UPDATE PAY
                                SET SAL = SAL * 1.15
                                WHERE ENAME = "J. Doe"
                commit
            endif
GST12:      read(ENGINEER.Salary) into temp
                where ENGINEER.Name = "J. Doe"
            if temp is empty then
                abort
            else begin
                ENGINEER-Salary ← temp * 1.15
                write(ENGINEER.Salary)
                commit
            end
```

There could be other transactions that may have been submitted directly to the component DBMSs. For example, the following local transactions LT_1 and LT_2, update, respectively, the salaries of all electrical engineers in database 1 by 50% and the budgets of maintenance projects by \$50,000 in database 3.

```
LT1:  EXEC SQL      UPDATE PAY
                    SET SAL = SAL * 1.5
                    WHERE TITLE = "Elect. Eng."
      Commit
LT2:  read(PROJECT.Budget) into temp
          where PROJECT.Name = "Maintenance"
      PROJECT.Budget ← temp + 50000
      write(PROJECT.Budget)
      Commit
```

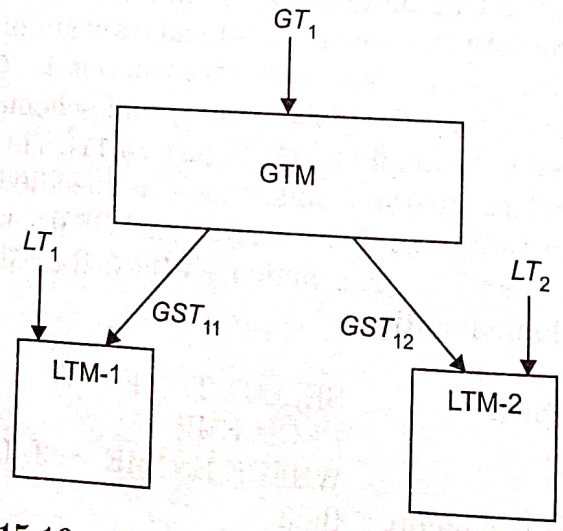


Figure 15.16. Transaction Execution Model Example

The execution of these transactions on the architectural model of Figure 15.15 is depicted in Figure 15.16.

15.3.2 Multidatabase Concurrency Control

There have been many proposals for ensuring consistency of concurrently executing transactions in a multidatabase environment. Recall from Chapters 10 and 11 that concurrency control algorithms maintain the consistency and isolation properties of transactions. Given the autonomy of the component DBMSs, it is not easy to maintain these properties. A number of rather strict conditions must be satisfied, which we will discuss shortly.

Concurrency control algorithms synchronize concurrent transactions by ordering their conflicting operations such that a serialization order can be maintained among transactions. Recall from Chapter 11 that two transactions conflict if they each have one operation that accesses the same data item, and one of these operations is a write (causing read-write or write-write conflict). It is not easy for the GTM to determine conflicts in a multi-DBMS. Two global transactions that are handled by the GTM may not appear to conflict at all. However, the existence of local transactions may cause conflicts at the component databases among transactions. *Indirect conflicts* of this type cannot be detected by the GTM and are a source of significant difficulty in multi-DBMSs.

Example 15.8

Consider data items x_1, x_2 stored at component database 1 of a multi-DBMS. For this example, we do no need to consider data at other component databases. Consider two global transactions:

GT_1: *Read*(x_1)
Update x_1
Write(x_1)
Commit

GT_2: *Read*(x_2)
Update x_2
Write(x_2)
Commit

These two transactions execute at site 1 and do not conflict. Thus, GTM may release both of them to DBMS-1. However, there may be a local transaction executing at this site unknown to GTM:

LT: *Read*(x_1)
Read (x_2)
$x_2 \leftarrow f(x_1)$ [Update x_2 based on x_1]
Write(x_2)
Commit

LT introduces a conflict between GT_1 and GT_2, which is not known to the GTM.

A series of conditions have been defined that specify when global transactions can safely update a multidatabase system ([Gligor and Popescu–Zeletin, 1986] and [Gligor and Luckenbaugh, 1984]). These conditions are helpful in determining the minimal functionality required of the various transaction managers.

The first condition for providing global concurrency control is to have the individual database managers guarantee local synchronization atomicity. This means that the local transaction managers are simply responsible for the correct execution of the transactions on their respective databases. If serializability is the correctness criterion used, each local transaction manager is responsible for maintaining that its schedule is serializable and recoverable. These schedules are made up of global subtransactions as well as local ones. Whether this condition can be enforced in an interoperable DBMS environment is questionable. As discussed in Chapter 10, many commercial systems allow less than serializable schedules. It may not be possible to enforce this requirement on all component DBMSs. Furthermore, the multi-DBMS has to deal with component DBMSs with different levels of transaction consistency enforcement.

The second condition requires that each LTM maintain the relative execution order of the subtransactions determined by the GTM. The global transaction manager, then, is responsible for coordinating the submission of the global subtransactions to the local transaction managers and coordinating their execution. If serializability is the correctness criterion used, the global transaction manager is responsible for the serializability of the global transaction execution schedules. Furthermore, the GTM is responsible for dealing with global deadlocks that occur among global transactions. Obviously, if the GTM awaits the result of one subtransaction before submitting the next, this ordering can be maintained. Whether this is possible without serializing transaction execution is a topic of much debate. There are methods that force component DBMSs to maintain the relative order. We discuss these in the next section.

In a distributed multi-DBMS, the global transaction manager is also responsible for the coordination of the distributed execution of global transactions. This involves a different execution paradigm than the one used in distributed DBMSs. In the latter, the transaction manager at the site where the transaction is submitted (called the *coordinating transaction manager*) can communicate directly with schedulers at its site and other sites. In distributed multi-DBMSs, however, this is not possible for two reasons. First, component DBMSs do not necessarily know how to communicate in a distributed environment. The earlier discussion on the functionality of the local transaction managers, together with our architectural discussions in Chapter 4, indicate that each component DBMS only knows how to communicate with an application program that executes on the same machine as itself. Second, global transaction managers usually have difficulty in scheduling transactions across multiple sites, so it may not be feasible for them to get even more involved with transaction scheduling across multiple DBMSs at one site. This would mean that a global transaction manager would send a global subtransaction

to another global transaction manager at another site and expect it to coordinate the execution of the global subtransaction. The global transaction manager at the other site may then further decompose the transaction into global subtransactions, depending on the organization of the local databases at its site. The condition that governs the execution of global transactions states that a global transaction should have only one global subtransaction executing at any one site.

Example 15.9.

Consider the following transaction, which, among other things, accesses two data items x and y stored at site 2.

GT_1: read(x)
⋮
write(x)
⋮
read(y)
⋮
write(y)
commit

Since GT_1 accesses, among others, two data items that are stored in site 2, it may be tempting (but incorrect, as we will demonstrate) for the coordinating global transaction manager to split it into the following two global subtransactions to be submitted to the global transaction manager of site 2:

GST_{11}^2: read(x)
⋮
write(x)
commit

and

GST_{12}^2: read(y)
⋮
write(y)
commit

Now consider a local transaction LT_1 that conflicts with either GST_{11}^2 or GST_{12}^2 (e.g., reading x or y). Then serializability of the global transaction GT_1 and the local transaction LT_1 would require that either $GT_1 \prec$ LT1 or $LT_1 \prec GT_1$. In terms of the subtransactions, this would mean that $GST_{11}^2 \prec LT_1 \prec GT_1$. However, it is possible to have an execution schedule, as, for example, $GST_{11}^2 \prec GST_{12}^2 \prec LT_1$ or $LT_1 \prec GST_{11}^2 \prec GST_{12}^2$. Certainly, this schedule is not serializable with respect to the local and global transactions.

The above discussion and example assume an execution model in which the multi-DBMS layer communicates with the component DBMSs by means of a high-level interface. This has been called the service request approach [Breitbart et al.,

1992a]. It has also been suggested that some component DBMSs may expose an interface that allows the GTM to submit transaction operations (e.g., read, write, commit, abort) one by one. If component DBMSs open up their interfaces like this, the GTM can have finer control on transaction execution.

How can one deal with concurrency control, given these rather difficult conditions? There have been numerous proposals; in fact, too numerous for us to review here. Some of these solutions are interesting, however, so we will discuss them briefly. A good survey of other solutions is given in [Breitbart et al., 1992a].

One approach has been to revise the set of acceptable solutions. Consider the schedule in Example 15.9. The careful reader will have noticed that the schedule $GST_{11}^2 \prec LT_1 \prec GST_{12}^2$ may not necessarily be wrong, even though it is not serializable. For example, if LT_1 conflicts with GT_1 by reading x, and if GST_{12}^2 never accesses x, the database would be consistent at the end of the execution schedule above, even though it may not be serializable. This is actually a known property of serializability theory: serializable schedules are only a subset of the correct execu-tion schedules. Thus serializability is quite conservative in the schedules it allows. In multidatabase systems this becomes even more significant. A corollary of the third condition that we discussed above is that a global subtransaction of a dis-tributed transaction should not be split further by the global transaction manager at the site to which it is submitted.

Example 15.10

Consider the same transaction GT_1 that we considered in Example 14.13, but this time assume that x and y are stored in different databases at the same site (say, site 2). GT_1 can be split into a number of global subtransactions, one of which is submitted to the global transaction manager at site 2 as follows:

$$\begin{aligned} GST_1^2: \ & \text{read}(x, y) \\ & \vdots \\ & \text{write}(x) \\ & \text{write}(y) \\ & \text{commit} \end{aligned}$$

Since x and y are in different databases, the global transaction manager itself may split GST_1^2 into GST_{11}^2 and GST_{12}^2 as defined in Example 14.13. Again notice that, if there is a local transaction LT_i that conflicts with either of these subtransactions, we would have a schedule that is not serializable but is correct.

This restriction is quite severe in multidatabase systems, since it makes it very difficult to find a computation model for executing transactions. Again considering the example above, where x and y are stored in different databases, to maintain serializability, the global transaction manager at site 2 has to hold exclusive access rights to data items at one local database (say, x) long after the transaction that accesses the data item (in this case, GST_{11}^2) may have completed.

This realization has caused some researchers to argue that serializability theory, as it is defined for distributed database systems, is unsuitable for distributed multidatabase systems [Du et al., 1989]. The outcome is a modification of the serializability theory such that the resulting class of schedules is a superset of the serializable schedules ([Du and Elmagarmid, 1989], [Breitbart and Silberschatz, 1988], [Barker and Özsu, 1990], [Mehrotra et al, 1991], and [Mehrotra et al, 1992]).

15.3.3 Multidatabase Recovery

Recovery protocols, as discussed in Chapter 12, deal with the atomicity and durability properties of transactions. The fundamental issue in this context is the implementation of atomic commitment protocols to ensure transaction atomicity. The autonomy of component DBMSs poses difficulties in this regard as well. We highlight the important issues in this section.

The most important consideration in the development of atomic commitment protocols for multi-DBMSs is whether or not the component DBMSs export a prepared-to-commit interface. If they do, the 2PC protocols discussed in Chapter 12 can be implemented. The remaining problems have to do with the differences in 2PC implementations across systems. There are standards, but, as is common, there are a number of them (e.g., LU6.2, OSI TP, RDA), and it will take additional effort to resolve the differences among them. In the end, however, advantages of interoperability will probably encourage vendors to provide a prepared-to-commit interface.

15.4 OBJECT ORIENTATION AND INTEROPERABILITY

Object orientation is expected to play a significant role in addressing some of the model and architectural issues in database interoperability. The modeling issues are relevant if the interoperability framework includes object DBMSs as well as relational ones. In this case, the global conceptual schema has to be object-oriented, since that is the most general model. We do not discuss this aspect of object-orientation in this chapter.

Two characteristics of object models are particularly important in addressing interoperability concerns. The first is encapsulation, which allows the differences in interfaces and implementations of component DBMSs to be hidden. In Section 15.1.1 we referred to this as schema translation. However, when object-orientation is brought to bear, "translator" does not accurately capture the role of this piece of software. A more appropriate term that has been proposed is *wrapper*, since the component DBMS is "wrapped" and its internal operations are hidden from other system components. We will discuss wrappers further in the next chapter when we discuss interoperability concerns related to World Wide Web data repositories. For the time being, it is sufficient to understand that wrappers encapsulate the data sources and provide a completely uniform interface to the outside world.

Example 15.11

Consider two component DBMSs, one relational and the other an old network model-based one. We don't need to know how network model DBMSs work,

except to note that they represent relationships between entities using labeled directed edges (similar to Entity-Relationship model), and the database access is by means of navigation along these edges. The relational DBMS provides an SQL interface, but the network DBMS only provides navigational primitives. The encapsulation property of object models enables the translation of the underlying database schemas to an object-oriented one. The wrappers also encapsulate the routines that translate the user query language (let's assume SQL) to the languages of the component DBMSs (Figure 15.17).

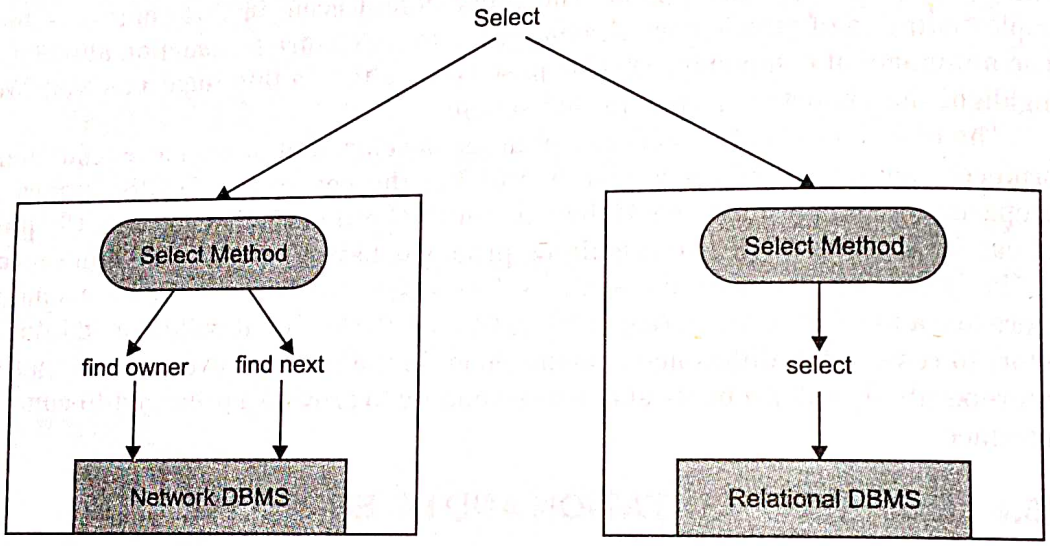


Figure 15.17. Encapsulation of Heterogeneous DBMSs

Another property of object-orientation that is useful for managing database interoperability is *specialization/generalization*, which is modeled by subtyping and supertyping, as discussed in Chapter 14. This allows for the creation of types which abstract the similarities of entities stored in different databases.

Example 15.12

Consider two **Employee** entities denned differently in two component databases. Assume that the definition in the first database assigns attributes **ENUM** and **SALARY**, and the second assigns attributes **ENUM** and **ADDRESS**. In the integrated (object-oriented) schema, these two entities could be modeled as subtypes of a more general **EMP** entity which abstracts the commonalities between the entity definitions (Figure 15.18).

In addition to these modeling advantages, there are a number of distributed object computing platforms whose main purpose is to facilitate the development of open systems by allowing applications to easily communicate with each other. Two such platforms are the Object Management Architecture (OMA) from Object Management Group (OMG) [Siegel, 1996] and Component Object Model (COM),

Distributed Component Object Model (DCOM) and Object Linking and Embedding (OLE) environment from Microsoft [Brockschmidt, 1995]. These platforms take the view that interoperability among information systems requires an environment that supports a common way to design each software piece (which is commonly referred to as *componentization*) whereby there are certain functions that can be expected of each component. The platforms provide an infrastructure that supports communication between components and between application programs and components, and incorporate services that are commonly needed by all distributed applications. The various platforms differ in how they accomplish these goals, and in the remainder, we discuss the two competing environments. We will present the fundamentals of each of these platforms and discuss how they can be used for DBMS interoperability.

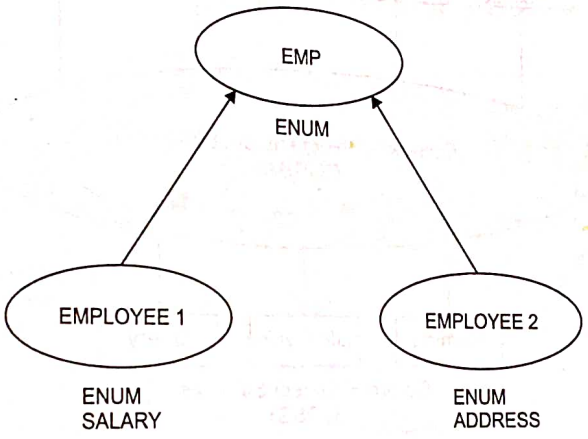


Figure 15.18. Abstraction of Heterogeneous Entities

15.4.1 Object Management Architecture

The Object Management Architecture (OMA) is an environment defined by the Object Management Group (OMG), which is a consortium of industry vendors committed to an object-oriented approach to building distributed systems. OMG puts forth standards and allows vendors to develop products that meet those standards. OMA defines a common object model, a common model of interaction by means of object invocations, and a set of common object services and facilities (Figure 15.19). OMA modules consist of the application objects; the Common Object Request Broker (CORBA), which directs requests and responses between objects; a set of common object services (COSS), which are the basic functions required for object management (e.g., naming, transactions, life cycle management); and a set of common facilities, which are generic object-oriented tools for various applications. Common facilities are divided into horizontal ones that are used by all applications (e.g., user interface, class browser) and vertical ones that are developed specifically for vertical market segments (e.g., healthcare, finance, telecommunications).

OMA's object model is a generic one that provides objects, values, operations, types, classes and subtype/supertype relationship among types. An object is an abstraction with a state and a set of operations. The operations have well-defined signatures, and the operations together with the signatures form the interface of each object. Each object and operation has a type. This is very similar to the object model that we described in Chapter 14. The communication between objects is by means of sending requests, whereby a request is an operation call with one or more parameters, any of which may identify an object (multi-targeting). The arguments and results are passed by value.

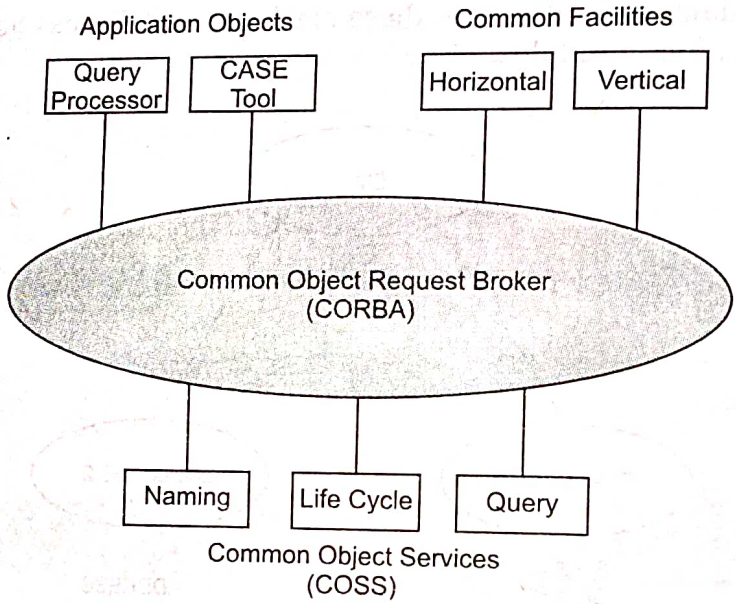


Figure 15.19. CORBA Architecture

Common Object Request Broker—CORBA

CORBA (Figure 15.20) is the key communication mechanism of OMA, in which objects communicate with each other via an Object Request Broker (ORB) that provides brokering services between clients and servers. Brokering involves target object location, message delivery, and method binding. Clients send a request to the ORB asking for certain services to be performed by whichever server can fulfill those needs. ORB finds the server, passes it the message from the client, and receives the result, which it then passes to the client.

The ORB performs the following functions, which we will describe in some detail below:

- Request dispatch to determine the identity of a method to be called.
- Parameter encoding to convey local representation of parameter values.
- Delivery of request and result messages to the proper objects (which may be at different sites).

- Synchronization of the requesters with the responses of the requests.
- Activation and deactivation of persistent objects.
- Exception handling to report various failures to requesters and servers.
- Security mechanisms to assure the secure conveyance of messages among objects.

The basic communication between two CORBA objects is accomplished by the ORB core. OMG does not place any restriction on how ORBs are implemented. In most ORB implementations, existing IPC (Inter-Process Communication) methods, such as Unix socket libraries, shared memory and multithreaded libraries, are used to achieve actual communication among clients, servers and the ORB. Yet ORB can be as simple as a library that supports communication among objects and their clients which are co-resident in the same process space.

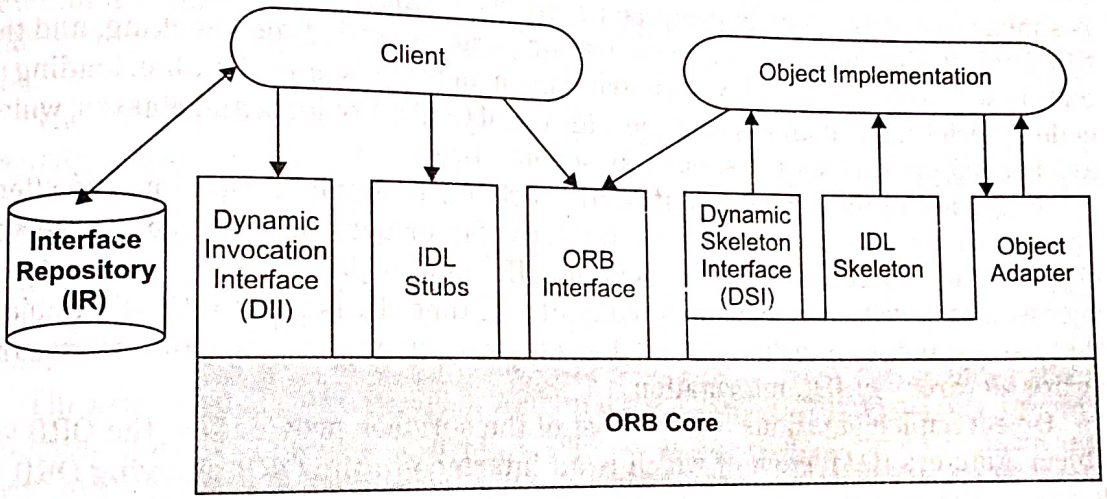


Figure 15.20. CORBA Architecture

To make a request, the client needs to know the operations that it is going to request of an object. In other words, it needs to know the interface of the object that will respond to the request. From here, the client can determine the reference of the target object which will service the request. This can be obtained either from the ORB as the reference of an existing object that is generally created by an object factory, or by using the Naming Services—one of the COSS modules. The target object reference, together with the requested operations, constitutes the request.

The interfaces are defined by means of the Interface Definition Language (IDL). IDL is a host language-independent, declarative language, not a programming language. It forces interfaces to be defined separately from object implementations. Objects can be constructed using different programming languages and still communicate with one another. IDL enables a particular object implementation to introduce itself to potential clients by "advertising" its interface. The IDL interface specifications are compiled to declarations in the programmer's own language. Language mappings determine how IDL features are mapped to facilities of a given programming language. There are standardized language mappings for C, C++,

Smalltalk, Ada and Java. IDL language mappings are those in which the abstractions and concepts specified in CORBA meet the "real world" of implementation.

IDL compilers generate client-side *stubs* and server-side *skeletons*. These are interface-specific code segments that cooperate to effectively exchange requests and results. A stub is a mechanism that effectively creates and issues requests on the client's behalf. A skeleton is a mechanism that delivers requests to CORBA object implementation. Communicating through stubs and skeletons is known as *static invocation*, since the linkage between the client and the server is established at compile time. An alternative is to use *dynamic invocation* through the Dynamic Invocation Interface (DII) at the client side, and the Dynamic Skeleton Interface (DSI) at the server side. DII allows clients to send requests to servers without the compile-time generation of stubs; DSI allows servers to be written without skeletons. Applications that establish static invocation bindings at compile time execute faster and are easier to program, since the programming interface is similar to ordinary object-oriented programs—a method is invoked on an identified object. Furthermore, static invocation permits static type checking, and the code is self-documenting. However, dynamic invocation is more flexible, leading to code genericity, and it allows runtime addition of CORBA objects and classes, which are needed for various tools, such as schema browsers.

For either of these modes, but particularly for dynamic invocation, the client application must have a way to know the types of interfaces supported by the server objects. The CORBA Interface Repository (IR) allows the IDL type system to be accessed and written programmatically at run time. IR is itself a CORBA object that has a standard interface. Using this interface, an application can traverse the entire hierarchy of IDL information.

Object implementations access most of the services provided by the ORB via object adapters (OA), each of which is an interface to the ORB allowing ORB to locate, activate, and invoke operations on an ORB object. The OA is a "glue" between CORBA object implementation and the ORB itself. It is an object that adapts the interface of another object to the interface expected by a caller. It uses delegation to allow a caller to invoke requests on an object even though the caller does not know the object's true interface. Until recently, only the Basic Object Adapter (BOA) was defined and had to be provided by all commercial ORBs, BOA is designed to be used with most of the object implementations and provides for generation and interpretation of object references, method invocation, registration, activation and deactivation of object implementations, selection of proper object implementation for a given object reference, and authentication. Recently OMG released a standard as an alternative to BOA. This standard, called the Persistent Object Adapter (POA), provides ORB portability. Some CORBA products have already started include POA as part of their basic system offering. There are expectations that, in the future, OMG will publish another standard for object DBMSs. Since object DBMSs provide some "ORB-like" services, such as object reference generation and management, this adapter will be tuned to integrate object DBMSs with ORB distribution and communication. Library object adapters will be tuned for implementations resident in the client's process space.

Earlier versions of CORBA (prior to Version 2.0) suffered from a lack of interoperability among various CORBA products, caused by the fact that earlier CORBA specification did not mandate any particular data formats or protocols for ORB communications. CORBA 2.0 specifies an interoperability architecture based on the *General Inter-ORB Protocol* (GIOP), which specifies transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport. CORBA 2.0 also mandates the *Internet Inter-ORB Protocol* (IIOP), which is an implementation of GIOP over TCP/IP transport. With IIOP, ORBs can interoperate with one another over the Internet.

Common Object Services

Object Services provide the main functions for implementing basic object functionality using ORB. Each object service has a well-defined interface definition and functional semantics that are orthogonal to other services. This orthogonality allows objects to use several object services at the same time without any confusion.

The set of services will eventually include naming, lifecycle, transaction, trader, security, event, concurrency, query, persistence, relationships, collections, time, properties, externalization, licensing, and change management. These services are at different phases of development. At the time of writing this chapter, standards (and, in most cases, products) are available for the first six listed above. For others, requests for proposals have been released, but no standards have yet been established.

The provision of these services, and their use by other CORBA objects, provide "plug-and-play" reusability to these objects. As an example, a client can move any object that supports Lifecycle Services by using the standard interface. If the object does not support the standard Lifecycle Services, then the user needs to know "move semantics" for the object and its corresponding interface.

Common Facilities

Common facilities consist of components that provide services for the development of application objects in a CORBA environment. Two classes of facilities have been identified. Horizontal facilities consist of those facilities that are used by all (or many) application objects. Examples of these facilities include user interfaces, systems management, and task management. Vertical facilities, on the other hand, are specialized components for selected application domains, such as health care, transportation, manufacturing, electronic commerce, and telecommunictions.

15.4.2 CORBA and Database Interoperability

As an object-oriented distributed computing platform, OMA, and in particular CORBA, can be helpful for database interoperability. The fundamental contribution is in terms of managing heterogeneity and, to a lesser extent, managing autonomy.

As discussed earlier, heterogeneity in a distributed system can occur at the hardware and operating system (which we can jointly call platform) level, communication level, DBMS level and semantic level. CORBA deals mainly with platform and communication heterogeneities. It also addresses DBMS heterogeneity by means of IDL interface definitions. However, the real problem of managing multiple DBMSs in the sense of a multidatabase system introduced earlier requires the development of a global layer that includes the global-level DBMS functionality. One issue with which CORBA cannot be helpful is semantic heterogeneity.

Using CORBA as the infrastructure affects the upper layers of a multidatabase system, since CORBA and COSS together provide basic database functionality to manage distributed objects. The most important database-related services included in COSS are Transaction Services, Backup and Recovery Services, Concurrency Services, and Query Services. If these services are available in the ORB implementation used, it is possible to develop the global layers of a multidatabase system on CORBA mainly by implementing the standard interfaces of these services for the involved objects. For example, by using a Transaction Service, implementing a global transaction manager occurs by implementing the interfaces defined in the Transaction Service specification for the involved DBMSs.

In this section, we discuss the design issues that must be resolved to use CORBA for database interoperability. This discussion is based, to a large extent, on the experiences with the MIND project ([Dogac et al., 1996a], [Dogac et al, 1996b], [Dogac et al., 1998b]).

A fundamental design issue is the granularity of the CORBA objects. In registering a DBMS to CORBA, a row in a relational DBMS, an object or a group of objects in an object DBMS, or a whole DBMS can be an individual CORBA object. The advantage of fine granularity objects is the finer control they permit. However, in this case, all the DBMS functionalities to process (e.g., querying and transactional control) and manage these objects have to be supported by the global system level (i.e., the multidatabase system). If, on the other hand, a whole DBMS is registered as a CORBA object, the functionality needed to process the entities is left to that DBMS.

Another consideration regarding granularity has to do with the capabilities of the particular ORB being used. In the case of ORBs that provide BOA, each insertion and deletion of classes necessitates recompiling of the IDL code and rebuilding the server. Thus, if the object granularity is fine, these ORBs incur significant overhead. A possible solution to this problem is to use DII. This prevents recompilation of the code and rebuilding of the server, but suffers the run-time performance overhead discussed earlier.

A second design issue is the definition of interfaces to the CORBA objects. Most commercial DBMSs support the basic transaction and query primitives, either through their Call Level Interface (CLI) library routines or their XA Interface library routines. This property makes it possible to define a generic database object interface through CORBA IDL to represent all the underlying DBMSs. CORBA allows, multiple implementations of an interface. Hence it is possible to encapsulate each of the local DBMSs by providing a different implementation of the generic database object.

Another issue is the association mode between a client request and server method. CORBA provides three alternatives for this: one interface to one implementation, one interface to one of many implementations, and one interface to multiple implementations. If there exists only one implementation of an interface (i.e., there is only one component DBMS that implements that interface), all of the requests should be directed to a server that supports this single implementation. If there is more than one implementation of an interface (i.e., there are more than one DBMSs that can fulfill the request), ORB can direct the requests to a server that supports any one of the existing implementations. In both cases, implementations handle all operations defined in the interface, and after implementation selection, ORB always uses the same implementation for requests to a particular object. If each implementation of an interface does not handle all of the operations defined in the interface—that is, if each implementation provides only a part of the interface—the third method is used for associating a client request with a server method. In this case, ORB directs the requests to a server that supports an implementation of the interface that handles the invoked operation. The choice of the alternative is dependent both on the data location and the nature of the database access requests. If the requested data is contained in one database, then it is usually sufficient to use the second alternative and choose the DBMS that manages that data, since DBMSs registered to CORBA provide basic transaction management and query primitives for all the operations the interface definition specifies. If the request involves data from multiple databases, then the third alternative needs to be chosen.

CORBA defines three call communication modes between a client and a server—namely, synchronous, deferred synchronous, and one-way. Synchronous mode is blocked communication, where the client waits for the completion of the requested operation. Synchronous mode can be restrictive for clients who issue operations that can be executed in parallel with multiple objects. In deferred synchronous mode, the client continues its execution after server selection and keeps polling the server to get the results until the operation is completed. In one-way operation, a client sends a request without any intention of getting a reply. CORBA does not support asynchronous mode, since the only method of communication is via a request. This implies that if a client is to receive asynchronous messages, it should also act as a server that implements an object that can receive requests. In other words, asynchronous mode of operation can be achieved between two CORBA objects by sending one-way requests to each other. The only disadvantage of this peer-to-peer approach is the increased complexity of the client code. For objects of a multidatabase system, synchronous call mode is generally sufficient. Deferred synchronous mode or the peer-to-peer approach should be used when parallel execution is necessary. For example, in order to provide parallelism in query execution, the global query manager of a multidatabase should not wait for the query to complete after submitting it to a component DBMS.

When registering objects to CORBA, it is necessary to specify an activation policy for the implementation of each kind of object. This policy identifies how each implementation gets started. An implementation may support shared, unshared, server-per-method or persistent activation policies. While a server that uses a

shared activation policy can support more than one object, a server that uses an unshared activation policy can support only one object at a time for an implementation. In the server-per-method activation policy, a new server is used for each method invocation, the persistent activation policy is similar to the shared activation policy, except that the server is never started automatically. Some of the objects in a multidatabase system need to be concurrently active. This can be achieved either by using threads on a server that uses a shared activation policy or by using separate servers activated in the unshared mode for each object. Otherwise, since a server can only give service for one object at a time, client requests to other client requests to the objects owned by the same server should wait for the current request to complete. Further, if the server keeps transient data for the object throughout its life cycle, all requests to an object must be serviced by the same server. For example, if a global transaction manager is activated in shared mode, it would be necessary to preserve the transaction contexts in different threads. However, if the global transaction manager is activated in unshared mode, the same functionality can be obtained with a simpler implementation at the cost of having one process for each active transaction.

15.4.3 Distributed Component Object Model

An alternative to CORBA infrastructure is the Distributed Component Object Model/ Object Linking and Embedding (DCOM/OLE) environment from Microsoft. DOOM is similar in functionality to CORBA ORB, while OLE is the complete environment for componentization. DCOM/OLE is a single vendor proposal and, therefore, its contents are somewhat fluid and changing. DCOM's root is Microsoft's Component Object Model (COM). The transition from COM to DCOM is conceptually simple. We will discuss COM first and then present DCOM extensions.

COM object model is quite different than CORBA's; COM objects are really not "objects" in the sense denned in Chapter 14. The main differences with CORBA object model are the following:

- A COM (or OLE) object is one which supports one or more interfaces as defined by its class. Thus, there could be multiple interfaces to an object. All objects support one interface called `IUnknown`.
- COM objects have no identifiers.
- There is no inheritance defined among object classes. The relationship among them is defined by means of containment/delegation and aggregation.
- COM objects do not have state; applications obtain a pointer to interfaces that point to the methods that implement them.
- There are two definition languages: Interface Definition Language (IDL) for defining interface, and Object Definition Language (ODL) for describing object types.

All COM objects have the following characteristics:

1. They are identified by a Globally Unique Identifier (GUID). GUID is not an object identifier in the classical sense.
2. They must register with the Window's registry. This registration is then used by COM library functions to locate, start and stop the components.
3. They must publish and implement a set of interfaces. COM also defines a set of interfaces that must be supported by all COM objects, in addition to the particular interfaces they support in the context of the application in which they intend to participate.

COM objects exist in two forms: DLLs and EXEs. Regardless of the form it is in, a COM object must meet all the standards prescribed by the COM component architecture. DLLs must reside on the same platform as the client, they are often referred to as "local servers." EXE objects can be accessed via networks; they are often referred to as the "remote servers."

Clients access COM objects by means of the interface defined for each object. This is accomplished by indirection through an *Interface Function Table*, each of whose entries points to an interface implementation inside the COM object (Figure 15.21). There is one Interface Function Table per each interface that the object supports. The client obtains a pointer to the Interface Function Table that corresponds to the particular interface that it wishes to access, and invokes the interface functions contained therein. This method isolates clients from interface implementations.

The `IUnknown` interface has three methods: `QueryInterface`, by which the client can interrogate the COM objects as to whether it supports a particular interface, and `AddRef` and `Release` methods, which are for reference counting to be used for garbage collection.

DCOM is "COM with a longer wire." The move from COM to DCOM is straightforward, because most COM library functions support distributed and networked components. Further, Windows provides the code needed to locate and communicate with components over a network. This facility allows clients to locate components across the network either transparently or by requesting a component residing on a particular machine.

"Remoting" a component requires no change to the implementation of this component or the clients that access it; the communication is handled via Windows' registry and networking facilities. These facilities provide us with the elegant "longer wire" feel of COM/DCOM's way of building distributed software.

OLE adds componentization by encapsulating a COM object together with a class factory for object creation. In this sense, OLE objects are almost identical to COM objects. As discussed earlier, true reusability of "objects" requires a complete componentization framework. OLE provides this in the Microsoft environment.

An OLE server (also referred to as a COM server) performs a number of functions. It encapsulates a COM object and a class factory. In addition to the COM object interfaces that it supports, it provides an `IClassFactory` interface to

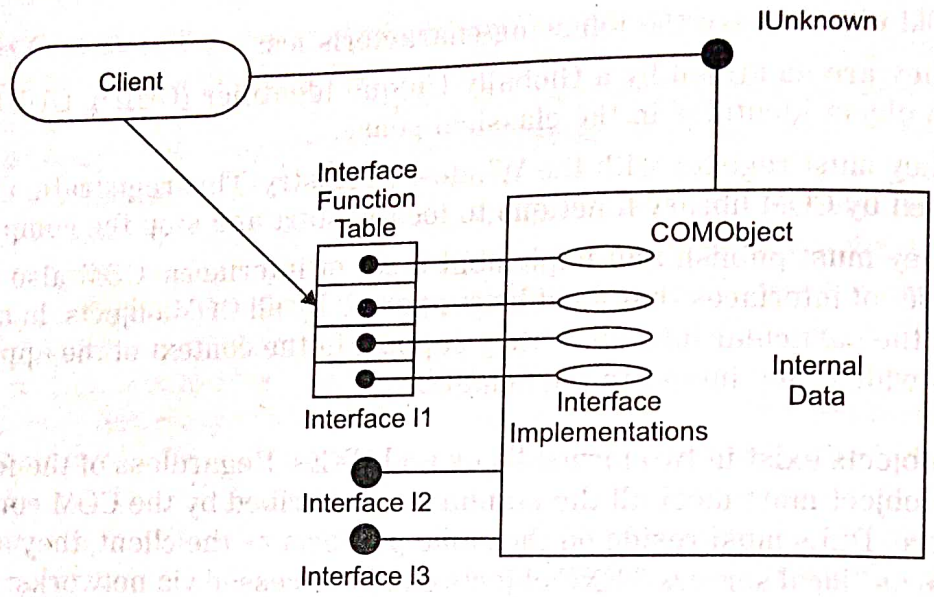


Figure 15.21. COM Objects and Interfaces

interact with the class factory. The functions that the server performs are the following: (a) it implements a class factory interface, (b) it registers the classes that it supports (there is an OLE registry for this purpose), (c) it initializes the COM library, (d) it verifies that the library version is compatible with the object version, (e) it implements a method for terminating itself when no clients are active, and (f) it terminates the use of the library when it is no longer needed.

COM/OLE is supported by the COM/OLE library, an API that provides component management services that are useful for all clients and components. This library guarantees that the most important and tedious operations are done in the same way for all components. Indeed, COM is a way of designing and building systems that can be used on any platform and with any language. However, Microsoft Windows so far provides the most support for COM programming in the form of the COM library. In the future, there may be ports of the environment to non-Windows environments.

15.4.4 COM/OLE AND DATABASE INTEROPERABILITY

Database interoperability in the COM/OLE environment is provided by OLE DB [Blakeley, 1996]. OLE DB extends the OLE environment to data repositories. It defines a uniform interface for all data repositories in the form of *rowsets*. In other words, all OLE DB data providers expose their data as rowsets, which are tabular representations of their data. COM objects that deal with repositories, therefore, communicate in terms of rowsets.

A rowset object (Figure 15.22) serves the function of a wrapper in providing a uniform interface. In its basic form, a rowset object has three interfaces in addition to `IUnknown`: `IRowset` provides methods for sequential iteration over the rows of a rowset, `IColumnsInfo` provides information about the columns of

the rowset, and `IAccessor` permits the definition of column bindings to client program variables. More elaborate OLE DB interfaces are defined for more inserting, deleting, and updating rows in a rowset, as well as more sophisticated ways of accessing a rowset (e.g., direct access, scrolling).

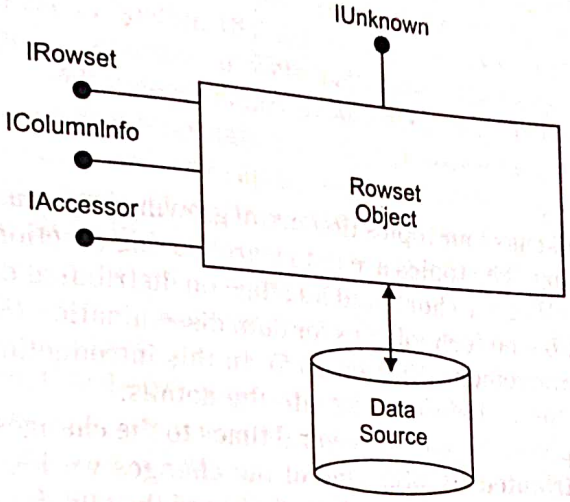


Figure 15.22. Rowset Object Abstraction

Using OLE DB, many of the design decisions discussed within the context of CORBA and database interoperability are made automatically for the system designer. There are commercial OLE DB providers for many of the relational DBMSs, which can be used to put together an interoperability framework for relational DBMSs. Similar providers are supposed to be available for object DBMSs as well.

REVIEW QUESTIONS

15.1 Explain the database integration process.
15.2 Give an example of entity-relationship database.
15.3 What do you mean by schema integration?
15.4 Explain attribute-to-entity transformation with an example.
15.5 What is the structure of a distributed multi-DBMS?
15.6 What are the query processing steps in multidatabase systems?
15.7 Explain distributed multi-DBMS transaction management.
15.8 Give an example of transaction execution model.
15.9 Explain multidatabase concurrency control.
15.10 Explain object orientation and interoperability.