The Pegasus Heterogeneous Multidatabase System

Rafi Ahmed, Philippe De Smedt, Weimin Du,
William Kent, Mohammad A. Ketabchi, Witold A. Litwin,
Abbas Rafii, and Ming-Chien Shan
Hewlett-Packard Laboratories

Benefiting from object-oriented data modeling and programming capabilities, Pegasus uses both type and function abstractions to resolve mapping and integration problems.

Pegasus, a heterogeneous multidatabase management system being developed by the Database Technology Department at Hewlett-Packard Laboratories, responds to the need for effective access and management of shared data across in a wide range of applications. Pegasus provides facilities for multidatabase applications to access and manipulate multiple autonomous heterogeneous distributed object-oriented, relational, and other information systems through a uniform interface. It is not just a front-end approach to multiple databases but a complete data management system that integrates various native and local databases.

The literature describes a number of heterogeneous database projects and systems. Litwin, Mark, and Roussopoulos[1] and Thomas et al.[2] survey prototype and commercial heterogeneous multidatabase management systems, and Gupta[3] provides a collection of papers on the subject.

A heterogeneous multidatabase system must support various database systems with different database models, languages, and services. One approach to reduce the number of mappings between diverse data systems is to define a common data model and language. For instance, Dataplex[4] maps the underlying data models to a relational data definition. Since a basic relational model is not sufficient to capture the integrated semantics[5] of underlying systems, the Amoco Distributed Database System[6] uses an extended relational data model to integrate relational, network, and hierarchical databases.

Multibase,[7,8] an ambitious project that has interesting similarities to and differences from Pegasus, provides a uniform and integrated interface for retrieving data from existing heterogeneous databases. Multibase uses a functional data

0018-9162/91/1200-0019$01.00 © 1991 IEEE

model to represent schema of various existing databases. A view mechanism defines the integration of local database schemas. The view mechanism also specifies the rules for resolving data mismatches.

Multibase uses the function abstraction in the Daplex schema language for writing integration rules and providing operations that database management systems (DBMSs) do not provide. The Multibase experience has indicated the need for a more extensible framework for dealing with the peculiarities of various DBMSs.

Pegasus takes advantage of object-oriented data modeling and programming capabilities. It uses both type and function abstractions to deal with mapping and integration problems. Function implementation can be defined in an underlying database language or a programming language. Data abstraction and encapsulation facilities in the Pegasus object model provide an extensible framework for dealing with various kinds of heterogeneities in the traditional database systems and nontraditional data sources ranging from simple text to complex multimedia systems.

Pegasus data model

Pegasus' object-oriented model serves as a framework for uniform interoperation of multiple data sources with different data management systems. The model, based on the Iris object-oriented model,[9] contains three basic constructs:

- *Types* have unique names and represent collections of objects that share common characteristics. Types are organized in a directed acyclic graph that supports generalization and specialization and provides multiple inheritance. A type may be declared to be a subtype of other types. A function defined on a given type is also defined on all its subtypes. Objects that are instances of a type are also instances of its supertypes.
- *Objects* are uniquely identified by their object identifiers. Some objects, such as integers, are self-identifying. Objects may gain and lose types dynamically. For example, an object representing a given person may be created as an instance of the Student type. Later, it may lose the Student type and acquire the Employee type.
- *Functions* are the manifestations of operations and provide mappings among objects. Properties of, relationships among, and computations on objects are expressed in terms of functions. Arguments and results of functions are typed. A type can thus be characterized by the roles it plays in the arguments and results of various functions.

The unifying data definition and data manipulation language of Pegasus is the Heterogeneous Object Structured Query Language. HOSQL is a functional as well as object-oriented language that provides declarative statements to manipulate multiple heterogeneous databases and to create types, functions, and objects in both Pegasus and underlying local databases. Specifications of types and functions can also be imported from underlying local databases and can then be integrated into the Pegasus native schemas, if so desired.

Databases, types, functions, and instances are defined by HOSQL statements of the form:

```
CREATE ObjectSpecification AS
  ObjectImplementation;
```

ObjectSpecification is Database or Type followed by a user-defined name or Function followed by a function name and the specification of its arguments and results. AS ObjectImplementation is an optional clause that specifies how the object is created.

Pegasus provides mapping facilities to generate a Pegasus schema that gives a local data source the appearance of a Pegasus database and thus lets the user access the database with HOSQL queries. The mapping facilities are modular, with a separate module for each local data model (such as relational and network models).

Each module provides mechanisms for specifying mapping between the data model of a data source and the data model of Pegasus, and for translating queries expressed in HOSQL into the language of the data source. Mapping mechanisms are supported by variants of the AS clause of the Create statement. The example below creates a type Employee that represents an employee entity in a relational database. The employee entity is identified by the primary key (PK) Empno in the Emprel relation in the Empdb relational database. The functions defined on the Employee type are mapped to the attributes of employee given in Emprel.

```
Create Type Employee AS
  Relational Empdb.Emprel (PK =
  Empno);
Create Function Eno (Employee e)
  -> Integer x
    As Relational Empdb.Emprel
      (x = Empno);
Create Function Name (Employee
  e) -> String n
    As Relational Empdb.Emprel
      (n = Name);
Create Function Skills (Employee
  e) -> String s Many
    As Relational Empdb.Emprel
      (s = Skill);
```

In the example above, the mapping and translation specifications can be fully or partially automated via special-purpose tools. Ahmed and Rafii[10] describe automatic mapping of relational schemas to Pegasus schemas.

HOSQL variables, which are references to objects in the result or argument of a function, can be used in queries and update statements. Variables range over the domains of types they refer to. An object can be retrieved into a variable, which can then be used to refer to the object. An HOSQL query can be expressed by following syntax:

```
SELECT list of variables or
  functions
FOR EACH list of all variables
  and their types
WHERE predicate expression;
```

The SELECT clause lists variables or functions. The FOR EACH clause quantifies and types all variables used in the SELECT and WHERE clauses. The WHERE clause contains a predicate expression that may involve nested functions, variables, constants, or subqueries.

Data integration

Other databases can be connected to a Pegasus database to provide access to multiple data sources. Figure 1 shows a Pegasus database system configuration.

A data source is typically a database,

although it can be of another type (such as a file system). Data sources not directly controlled by Pegasus are called local data sources. A local data source is represented in Pegasus by an imported schema that looks like a Pegasus schema, but the underlying data is in the local data source. A complete or partial mapping of a local schema can be visible through Pegasus. A native database is created in Pegasus, and both its schema and data are managed by Pegasus.

Before a data source can participate in a Pegasus multidatabase system, the external characteristics of its system must be registered with Pegasus. Registration is carried out at the system level in a given environment (platform and network). Registration describes data management systems, network protocols, network nodes, machine types, etc.

Attachment is an important data integration facility provided by Pegasus. Attachment logically extends a native database with other databases and creates unified Pegasus schemas. A sequence of mappings to one or more local databases can be labeled, stored, and later recalled in an attach statement. An attach statement activates a stored group of mappings and attempts to establish connections to the named databases. An attachment can expose a particular integrated view of the underlying data to a group of users.

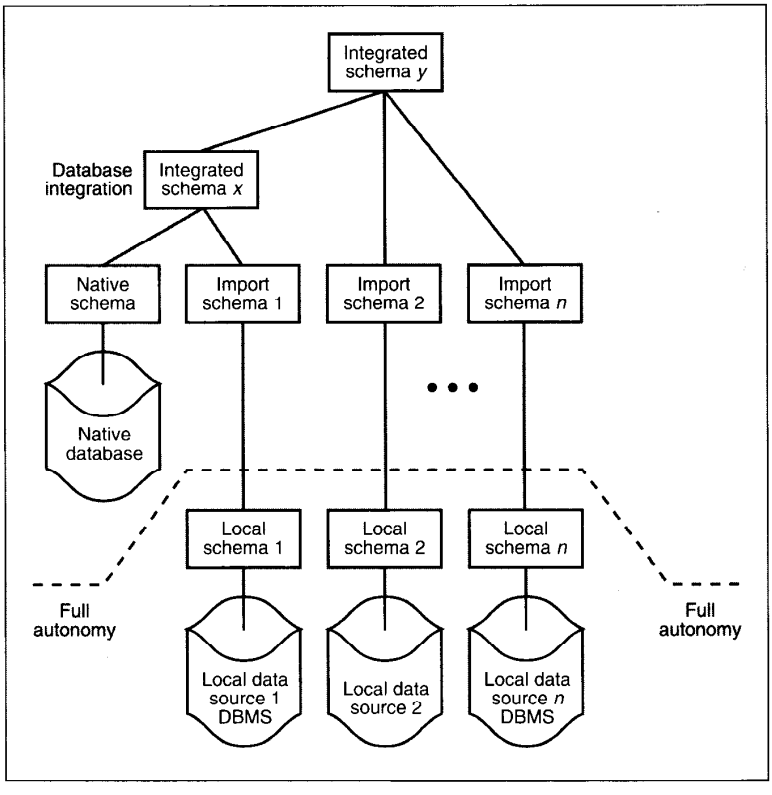


Figure 1. The Pegasus database system configuration.

Multiple data sources can be interoperated via Pegasus without having an integrated global schema. HOSQL statements may refer directly to the individual imported schemas. Integration in Pegasus is optional and deals with semantic and schematic heterogeneity among different databases, all of which have imported schemas in Pegasus.[11]

By default, Pegasus unifies the native schema and all imported schemas. The user-created types, functions, and objects in the native and imported databases are presumed to be distinct and disjoint. Names of types and functions may be prefixed by their database names to prevent ambiguities. Ambiguities involving the native database will be resolved in favor of the native database. In Pegasus, a single specification of system types and functions, as well as literal objects, is shared by native and imported databases.

The Pegasus prototype supports one fairly natural integration technique; it creates supertypes of types defined in underlying databases. Suppose the Pegasus schema contains types Programmer and Engineer, which might originate in different databases. Using the following command, we can create an abstract supertype Person, which acquires all the common functions from Programmer and Engineer:

```
Create Type Person supertype of
    Programmer, Engineer;
```

If both Programmer and Engineer have functions such as Name and Project, then the supertype Person acquires these functions. This mechanism can also be looked on as upward inheritance. The following query

```
Select Name (x), Project (x) For
    Each Person x;
```

will retrieve the names and projects of all programmers and engineers.

Simple mismatches of function names can be handled by using the Alias feature of HOSQL, allowing functions with different names to participate in upward inheritance as though they had the same name.

In general, semantic or behavioral differences among functions in different databases cannot be reconciled automatically. The Pegasus mechanisms for defining derived and foreign functions allow a database administrator to specify the appropriate reconciliation strategy. The body of a derived function is written with HOSQL statements. The body of a foreign function can be written in any general-purpose programming language and dynamically linked with Pegasus.

Domain mismatch. The domain mismatch problem arises when common concepts are treated in different ways by different domains in different spheres.[12] Consider the concept of money. The different domains correspond to different currencies in which money might be represented. A sphere is some scope in which a single domain, that is, a single currency, is used.

Currencies represent a relatively sim-

Schema and domain mappings

Consider two existing databases in an organization. As the figure shows, a personnel database Empdb stores information about full-time employees. A departmental database Progdb describes programmers and their projects.

In Empdb, employees are identified by their unique employee number (Eno). The designers of Progdb chose social security numbers (SSNs) to uniquely identify the programmers.

Some programmers are full-time employees and appear in both databases. Some employees are not programmers, and some programmers are not full-time employees. One possible approach for integrating these databases is to define a new Pegasus database that maps employee and programmer entities to Employee and Programmer types, respectively.

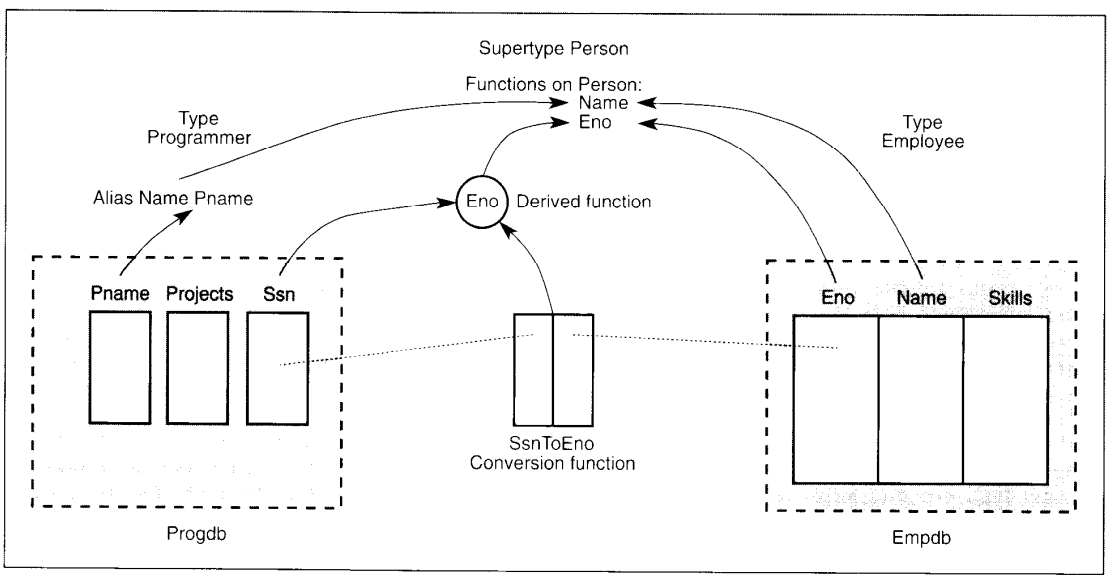
The functions defined on type Employee are mapped to attributes of employees in Empdb. Similarly, the functions defined on type Programmer are mapped to attributes of programmers in Progdb.

Initially, programmers do not have employee numbers. To identify the relationship between programmers and employees, a conversion function (SsnToEno) is defined that maps a social security number to an employee number.

The mapping in this case is a simple one-to-one mapping. But more complex mapping algorithms can be encapsulated in this function. Given this function, you can derive a new Eno function for the Programmer type that uses the SsnToEno function to produce an employee number (if any) for a programmer. The new Eno function for programmers can be used in a query to find the skills of programmers who are full-time employees.

The alias statement handles function name mismatches. It is used here to unify the Name function for both Employees and Programmers. With this preparation, functions Name and Eno are defined on both Employee and Programmer types. One can define a new type Person that is a supertype of Employee and Programmer types. The new type will acquire the common functions defined on its subtypes. Therefore, a query that references the Name of persons retrieves names of both programmers and employees.

In the future, the query processor of Pegasus will be able to automatically use the association between employees and programmers to eliminate duplicate names for employees that are also programmers.



An integrated view of two databases.

ple sort of domain mismatch, involving computational conversions among literal data values. More complex discrepancies arise when the same concept is perceived as being populated, or partitioned, in different ways.

The concept of "job" might be common to several spheres, yet each sphere might have a different notion about specific jobs. One sphere might have engineer, secretary, and salesperson as jobs, while the jobs in another might include technician, designer, engineer, secretary, and customer representative. Another kind of mismatch arises if concepts are represented in one sphere as literal values but as persistent objects in another.

In a typical domain mismatch problem, information maintained in different spheres must be presented in some

globally unified form in an integrating sphere. For example, corporate headquarters (the integrating sphere) may wish to see the starting salaries for all jobs. Different divisions (the local spheres) may have different definitions of jobs, different algorithms for defining starting salaries, and different currencies in which they are expressed.

An ideal domain mapping is an invertible computation on a stable population of literal data values, such as unit conversion. But the populations may not be stable, requiring the mapping to be updated; for example, adoption of new letter grades at one school might require updating the mapping between other schools. Also, the mapping may not be invertible, perhaps being many-to-one.

The easiest solution puts the burden on users, requiring them to maintain the domains and mappings by appropriately creating and deleting objects and modifying mapping rules or data (see the sidebar). In this case, a mapping simply returns an error when it encounters an unfamiliar value. Suppose, for example, the Grade function for type Student1 and the Points function for type Student2 do not behave consistently. The types Student1 and Student2 might have different underlying databases. The user can define functions Map1 and Map2, which convert each to a common result:

```
Create Supertype Student of
  Student1, Student2;
Create Function Score(Student x) ->
  Real r AS
    IF Student1(x) THEN
      Map1(Grade(x))
    ELSE IF Student2(x) THEN
      Map2(Points (x))
    ELSE ERROR;
```

Schema mismatch. Schema mismatch occurs when the data elements of one database correspond to the schema elements of another database (that is, similar concepts are expressed differently in the schema). Depending on the model, these schema elements can be relations and attributes, entities and relationships, classes and methods, types and functions, etc. Our work is expressed in terms of the types and functions in the Pegasus object model.

Many schema mismatch problems are really domain mismatch problems, except that they involve schema elements, rather than data elements. Jobs, for example, are often modeled as types (that is, subtypes of Employee). Instead of Job(Sam) = 'Engineer', we know that Sam is an engineer because he is an instance of the type Engineer. For instance, consider a sphere, $StockSphere_1$, containing a stock market function called Activity with three arguments:

```
Activity (Char Company, Char
  Reading, Char Date) -> Real
  Price;
```

whose current extension is shown in Table 1.

Table 1. The $StockSphere_1$ data values.

Company	Reading	Date	Price
HP	Close	1/3/91	50
HP	Close	1/4/91	51
HP	High	1/3/91	52
HP	High	1/4/91	53
IBM	Close	1/4/91	51
IBM	High	1/4/91	54

Table 2. The $StockSphere_2$ data values.

Reading	Date	Price
Close	1/3/91	50
Close	1/4/91	51
High	1/3/91	52
High	1/4/91	53

Another sphere, $StockSphere_2$, might maintain the same data in separate functions for each company. For instance, for HP company, there is a function:

```
HP (Char Reading, Char Date) ->
  Real Price.
```

For IBM, there is another function:

```
IBM (Char Reading, Char Date) ->
  Real Price.
```

Table 2 shows the corresponding extension of the function HP.

In $StockSphere_1$, the domain of interest is a set of instances of company as data values. In $StockSphere_2$ on the other hand, the corresponding domain of interest is a set of functions.

To demonstrate the capability of Pegasus for reconciling the structural differences in schema of different spheres, we can define a function StockPrice that returns stock prices given a company, reading, and date.

```
Create function StockPrice(Char
  Company, Char Reading, Char
  Date) -> Real Price AS
    Select Price
    For Each Real Price, Char
      Company, Char Reading,
      Char Date
    Where Activity (Company,
      Reading, Date) = Price
    Union
    Select Price
    For Each Function f, Real
      Price, Char Company, Char
      Reading, Char Date
    Where FunctionName (f) =
      Company and f (Reading,
      Date) = Price;
```

In the above example, FunctionName is a system-defined function that returns the name of a function and *f* is a variable that ranges over all functions defined in the system. The result of the second select statement is to dynamically bind to *f* all functions whose names match the parameter Company. In the first select statement, this parameter is used simply as a data value. Clearly, we can define a function AverageStockPrice, if prices are returned from both spheres and the two prices are averaged.

Object identification. Object identification in single database systems is relatively simple. Most conventional object-oriented DBMSs have developed and adopted workable approaches. However, object identification in a heterogeneous multidatabase management system is difficult because logically different objects can have the same identifier in different data sources. The usual solution to the collision of object identifiers across multiple data sources is to introduce an independent system of globally unique identifiers that have to be mapped to the local identifiers of each participating local database.

A single logical object can have different identifiers in different data sources. The same object sometimes exists in multiple data sources. A given

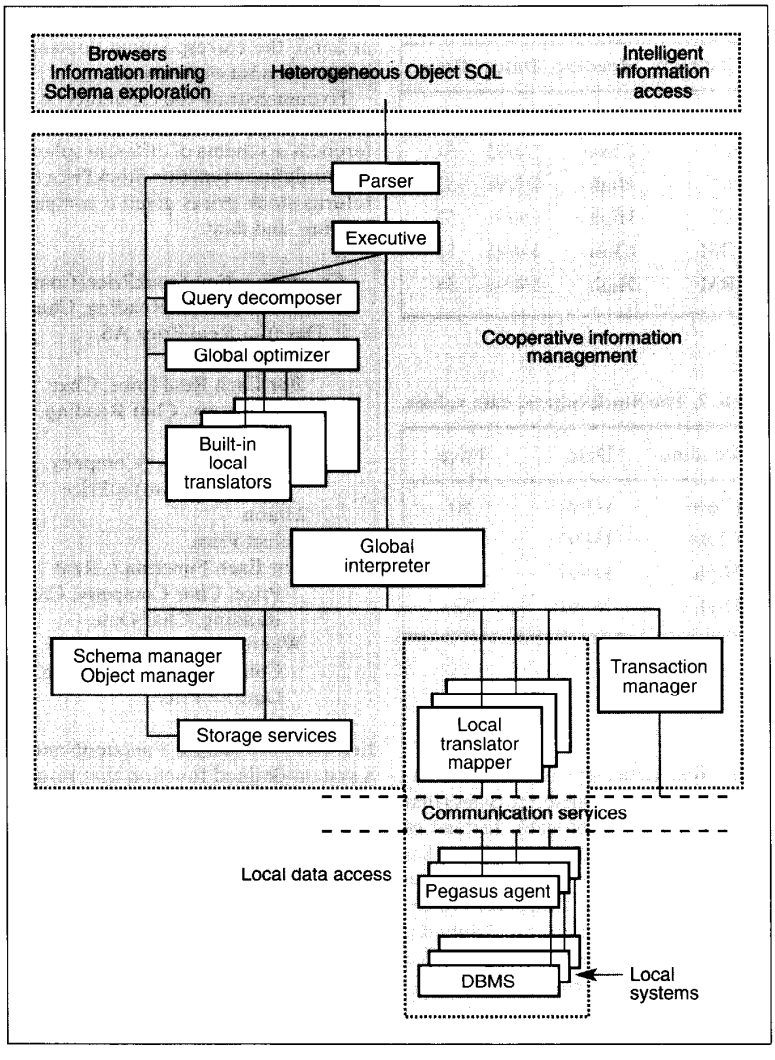


Figure 2. Functional layers of Pegasus.

student might be attending several schools, or the same person might exist in separate databases as a student and as a teacher. They can be expected to have different object identifiers in the different databases. There is, in general, no fully automatic way to deal with this. Therefore, Pegasus allows the user to specify equivalences.

The specification of equivalences might be an algorithm that matches social security numbers or a user-constructed table of corresponding object identifiers. Pegasus will attempt to treat equivalent object identifiers as synonyms for the same object.

Certain local data sources do not provide unique object identification. These sources can be handled either by modeling everything as literals or by introducing user-specified object identifiers. In the latter case, object identifiers are constructed from user-imposed types and key properties, such as student type and student number. This requires handling of heterogeneous identifiers with different formats and lengths.

These problems are being investigated in the Pegasus project. Providing an independent system of globally unique identifiers will simplify the solution. The correctness criteria used in evaluating various solutions are

- *Uniqueness.* Objects must be distinguishable from one another in local or global context.
- *Stability.* Objects must retain their identities despite changes in properties.
- *Consistency.* Object identifiers must not conflict with one another if the system supports several kinds, such as logical, local, and global object identifiers.

Pegasus architecture

Figure 2 shows the Pegasus functional layers:

- The *intelligent information access layer* provides such services as information mining, browsers, schema exploration, and natural language interfaces.
- The *cooperative information management layer* deals with schema integration, global query processing, local query translation, and transaction management.
- The *local data access layer* manages schema mapping, local query and command translation, network communications, local system invocation, and data conversion and routing.

The cooperative information management layer is responsible for processing HOSQL statements and coordinating multidatabase transactions. It supports the global object model and manages integrated schema and mapping information.

Executive. The executive manages the interaction between Pegasus and its clients. HOSQL queries are passed to the query decomposer module in a canonical form that represents the parse tree of HOSQL functional expressions. The nodes of the parse tree include function calls, types, variables, and literals.

The tree is decomposed into a set of subqueries with an execution plan. The execution plan is annotated with the catalog information retrieved from the Pegasus storage services. The execution plan can be viewed as a tree consisting of operational primitives and operands.

Examples of operational primitives are commands to perform global joins, to pass parameters (data) between

DBMSs, and to synchronize steps executed in parallel. The operand nodes refer to the data in the native database or to a subquery against a local data source.

Optimizer. The optimizer module produces a more efficient alternative plan that is equivalent to the original plan. Several strategies are used by the optimizer. Executing a single DBMS query bypasses the optimization process and goes directly to the destination data source. The optimizer tries to reduce the invocations of local DBMSs by grouping together the subqueries that refer to the same DBMS in an execution plan. The grouping merges several subqueries into a single query by possibly adding more restriction clauses to the merged query. Another strategy is to reduce the size of intermediate data retrieved from a DBMS. Based on statistics and heuristics on the selectivity of queries and volume of data in various DBMSs, a cost-based optimization is used to determine join order, join method, intermediate data routing, buffering, etc.

Unlike distributed DBMSs, Pegasus has limited access to the statistical information about local DBMSs. Moreover, Pegasus has no control over optimizing subqueries sent to each DBMS. For example, Pegasus cannot enforce a particular access path to be used at a database site. Otherwise, it will violate site autonomy. Therefore, Pegasus emphasizes global optimization and tries to find the best possible decomposition and grouping of queries.

Local translator. After the query execution plan is determined, a subquery in the plan may be submitted to a built-in local translator depending on the type of a local system. A local translator uses the mapping information between a local schema and an equivalent imported schema to translate a Pegasus subquery into the language of the local database (for example, SQL for a relational DBMS).

Pegasus internally supports the local translators for important systems such as relational databases. Other translators can be provided outside cooperative information management in the local data access layer. In this case, the details of binding a function (or a query) to the commands of the underlying server can be provided externally.

Global interpreter. The executive passes the final plan to the global interpreter for execution. The global interpreter dispatches and synchronizes internal and external executables, such as the Pegasus schema manager and object manager and local interpreters. The global interpreter also implements its own primitive database operations such as join, union, filter, and move. These operations apply to the data retrieved from the other executables.

The global interpreter dispatches the transaction manager, which provides transaction-oriented facilities. As various heterogeneous servers begin to interoperate in a cooperative environment, the need for managing transactions that preserve some degree of isolation and maintain global data consistency among different systems becomes important.

The main source of difficulty in applying traditional transaction management techniques in these environments is the local autonomy of the local systems that participate in the transactions. In conventional distributed DBMSs, the execution coordinator communicates with the local databases to enforce data integrity through the well-known two-phase locking and two-phase commit protocol. This is possible because all the local databases that participate in the transaction observe the same transaction protocols.

In a heterogeneous environment, not all participants will have the same transaction protocol. Therefore, Pegasus is exploring new transaction management techniques, which can provide more flexibility.

Schema and object managers. The schema and object managers implement data definition operations, catalog management, object management, and schema integration services. All information about the mapping of schemas defined in a local system to an equivalent imported schema is kept in the catalog to be used for query processing and transaction management. The object manager, among other things, implements the data-definition operations of the model and maintains the user-defined mappings between the various object identifiers in different object-oriented DBMSs. The mapping information can be used to detect object equivalences in different local databases.

Local translator/mapper. Interface to local systems is implemented via local translator/mapper modules and Pegasus agents. A local translator/mapper module implements translation and mapping services not provided in cooperative information management. These modules can also provide additional semantics that are not provided in a local system. For instance, they can implement data exchange algorithms that are typically needed in a heterogeneous data environment.

During idle times, these modules can collect statistical data and communicate the results to the central site. They can also play a role in assisting global transaction management by providing missing functionalities such as two-phase commit or undo that might not be available in the local system.

Pegasus agents. A Pegasus agent is a process that runs in the same machine as a local DBMS. Its role is to represent Pegasus in the local site. The module is normally linked with the local system like one of its applications. It receives the translated commands from Pegasus, sends it to the local system, collects the results, and sends them back to Pegasus.

A Pegasus agent should match its buffer management policies with Pegasus to reduce communication and buffer management overhead. As a participant in a global query, a Pegasus agent may have to keep track of several active queries in its corresponding local DBMS. For instance, a Pegasus agent associated with a relational database may open multiple scans over tables accessed as part of a global query.

We believe that flexible, efficient, and general-purpose heterogeneous multidatabases are needed to support the trend toward the extensive use of computers and information as competitive tools in today's complex business world. However, designing and implementing such systems is a major undertaking.

Several problems must be solved before robust general-purpose heterogeneous multidatabase management systems become possible. The problems that must be resolved include distinguishing equal but logically different objects, consolidating different representations of the same object, materializing the views of existing applications,

resolving the semantic and schematic heterogeneity of information stored in multiple databases, maintaining consistency of data in the presence of multidatabase concurrent transactions, and doing all this efficiently.

The near-term plans of the Pegasus multidatabase management system include facilities for updating the local databases through Pegasus, providing flexible transaction management and concurrency control, defining relationships across local databases, and resolving domain mismatches by providing conversion functions. ■

References

1. W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," *ACM Computing Surveys*, Vol. 22, No. 3, Sept. 1990, pp. 267-293.

2. G. Thomas et al., "Heterogeneous Distributed Database Systems for Production Use," *ACM Computing Surveys*, Vol. 22, No. 3, Sept. 1990, pp. 237-266.

3. *Integration of Information Systems: Bridging Heterogeneous Databases*, A. Gupta, ed., IEEE Press, 1989.

4. C. Chung, "Dataplex: An Access to Heterogeneous Distributed Databases," *Comm. ACM*, Vol. 33, No. 1, Jan. 1990, pp. 70-80 (with corrigendum in *Comm. ACM*, Vol. 33, No. 4. p. 459).

5. C. Batini, M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys*, Vol. 18, No. 4, Dec. 1986, pp. 323-364.

6. Y. Breitbart, P.L. Olson and G.L. Thompson, "Database Integration in a Distributed Heterogeneous Database System," *Proc. Data Eng. Conf.*, IEEE CS Press, Los Alamitos, Calif., 1986, pp. 301-310.

7. T.A. Landers and R.L. Rosenberg, "An Overview of Multibase — A Heterogeneous Database System," *Distributed Databases*, H.-J. Schneider, ed., North-Holland, Amsterdam, 1982, pp. 153-184.

8. U. Dayal and H.-Y. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," *IEEE Trans. Software Eng.*, Vol. SE-10, No. 6, Nov. 1984, pp. 628-645.

9. D.H. Fishman et al., "Iris: An Object-Oriented Database Management System," *ACM Trans. Office Information Systems*, Vol. 5, No. 1, Jan. 1987, pp. 48-69.

10. R. Ahmed and A. Rafii, "Relational Schema Mapping and Query Translation in Pegasus," *Proc. Workshop on Multidatabases and Semantic Interoperability*, 1990, pp. 22-25.

11. A. Rafii et al., "Integration Strategies in Pegasus Object Oriented Multidatabase System," *Proc. Hawaii Int'l Conf. System Sciences*, to be published 1992 by IEEE CS Press, Los Alamitos, Calif.

12. W. Kent, "Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language," *Proc. VLDB Conf.*, 1991, Morgan Kaufmann, San Mateo, Calif., pp. 147-160.

Rafi Ahmed is a member of technical staff at the Hewlett-Packard Laboratories, Palo Alto, California. His research interests include fluid mechanics, version management, temporal and heterogeneous databases, query languages, and query optimization.

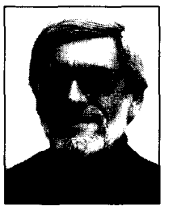
Ahmed received his MS degree in mathematics from the University of Bihar, India, and his MS and PhD degrees in computer science from the University of Florida.

Philippe De Smedt has been a member of the technical staff at the Hewlett-Packard Laboratories, Palo Alto, California, since 1985. He is responsible for integrating multimedia data sources on the Pegasus project. His research interests lie in performance evaluation, database technology, and multimedia systems.

De Smedt received an MS in computer science from the University of California at Berkeley and an MS in business and technology from Stanford University.

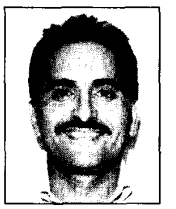
Weimin Du is a member of the technical staff at the Hewlett-Packard Laboratories, Palo Alto, California, where he conducts research into heterogeneous database systems. His research interests also include software engineering and programming languages.

Du received the BE and ME degrees in computer engineering from Hefei University of Technology, China, and MS and PhD degrees in computer science from Purdue University.



William Kent researches object-oriented data models in the Database Technology Department at the Hewlett-Packard Laboratories, Palo Alto, California. He played a leading role in developing the Iris prototype object-oriented database system and is active in the Pegasus project, investigating interoperability of heterogeneous database systems.

Kent authored the book *Data and Reality* and numerous papers in data analysis and design, conceptual schemas, entity-relationship models, the relational data model, and object orientation.



Mohammad A. Ketabchi is associate dean of the School of Engineering of Santa Clara University and an associate professor of computer engineering there. He is the founder and director of the Santa Clara University Object Technology Research Laboratory, and has been a consultant to Hewlett-Packard Laboratories, Palo Alto, California, since 1987. He has been active in the development, evaluation, and application of object-oriented systems since 1982.

Ketabchi obtained his PhD in computer science from the University of Minnesota, Minneapolis.

Witold A. Litwin, a professor at the University Paris-Dauphine, is a visiting researcher at the Hewlett-Packard Research Laboratories, Palo Alto, California, and Santa Clara University. His research interests are distributed databases, multidatabase systems, interoperability, and dynamic data structures.

Litwin has written more than 150 research papers, edited five books, and lectured at several universities.

Abbas Rafii is an engineer scientist in the Hewlett-Packard Laboratories, Palo Alto, California. He joined Hewlett-Packard in 1979 and has worked on several projects on computer system performance measurement and modeling, operating systems, and object-oriented databases. He was the technical leader of the Pegasus prototype team.

Rafii received his BS degree from the University of Tehran, and his MS and PhD degrees in electrical and computer engineering, respectively, from Stanford University.

Ming-Chien Shan manages the Pegasus project and Perseus project at the Hewlett-Packard Laboratories, Palo Alto, California. He has been with Hewlett-Packard since 1985 and was one of the original members of the Iris object-oriented DBMS team.

Ming-Chien received an MS degree in mathematics from Rutgers University and a PhD degree in computer science from University of California, Berkeley.

Readers can contact the authors at the Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94303-0971.