

THE DATAPHOR DATA ACCESS ENGINE

Providing a New Metaphor for Database Application Development

© Copyright December 2001 by Alphora

Bryn Rhodes

bryn@alphora.com

I. INTRODUCTION

Almost every application developed today will include some form of database access, whether it is a simple, single-user, flat-file database application with limited use, or an enterprise-wide accounting and employee management system, the issues facing the developer are the same: provide consistent, efficient access to the data being presented by the application.

Throughout the lifetime of any software, there are bound to be requirements changes, and this is particularly true of database applications. Because the current development model produces an application with many layers, each duplicating the information contained in the database description, extensive maintenance is required to respond to these changes. By centralizing the business model, and providing uniform access to the data, the Dataphor toolset provides a solution to the problems of existing applications.

In attempting to provide a centralized business model, however, one encounters several problems with existing Database Management Systems (DBMSs), ultimately leading back to a failure of these systems to correctly implement the relational model. The Dataphor Data Access Engine (DAE) acts as an insulating layer between the applications that consume the data, and the database systems which provide it.

II. CURRENT ISSUES IN DATABASE APPLICATION DEVELOPMENT

Database application development is the process of building software that utilizes a persistent data store. Under this broad definition, almost any application of computer technology today can be considered a database application. Certainly every application deals in some way with a set of data, and every application that does so has the same basic set of requirements. They must all provide consistent, efficient access to the data they manipulate. This is not always an easy goal to meet, particularly as the amount of data or users in a particular system increase. Database Management Systems (DBMS) are software systems specifically designed to handle these problems. The field of database theory is concerned with attempting to deal with these problems in a systematic and well-defined manner. By abstracting the common elements of data management, and devising a theory by which characteristics of data can be modeled, a foundation is laid for the development of these systems. Database application development can then be greatly simplified by building on top of systems which handle common problems of data management. The application developer is then free to concentrate on the problem of what data is to be stored, not how to store it.

The Relational Model of Data was introduced as a theory upon which such a system could be built, and many of today's systems claim to be Relational Database Management Systems (RDBMS). However, many of the problems facing database application developers today are a direct result of the failure of these systems to correctly implement the relational model. Still other problems arise out of the lack of a coherent model for database application development.

II.A. Issues Due to Flawed Implementation of the Relational Model

The first class of problems arises from the failure of existing systems to implement the relational model correctly. These problems can be further sub-divided into three main categories, those relating to the database access language, those relating to Logical Data Independence, and those relating to Physical Data Independence.

II.A.1. Database Access Language

The Relational Model provides a foundation for the construction of a database access language which would allow for the manipulation of sets of data called relations (or tables, informally) in much the same way that algebra is able to manipulate numbers [1]. This foundation is called the relational algebra, and provides a basis for building a system which is capable of realizing the goal of abstract data management. Most, if not all, commercially available DBMSs of today use a database language called Structured Query Language (SQL). Since its inception, SQL has been adopted almost completely as the language of choice for database access; however, there are two main problems with the language. First, the choice of nulls as the solution for missing information, and second, the ability to contain duplicate rows, undermine the languages ability to meet the goals of data management.

Missing information is a difficult problem, and much of the literature has been devoted to its solution. The SQL approach to missing information is the introduction of a flag, or null mark to indicate that some value is unknown. Despite numerous technical criticisms of this approach [3, 5, 6, 7, 8], it has become the standard method for dealing with missing information, and is embedded in SQL. One of the key problems with this approach is that if a column is allowed to be unknown, then the result of any operation

against that column is also unknown. By allowing nulls in a 'relation', one is no longer able to determine whether one row of a table is equivalent to another row of the same table, as the operation may return 'unknown.' For this reason, SQL views are not updatable, in general, because it is possible that the rows to be updated cannot be located. As a result, the application developer is then faced with a difficult choice, either present the data in terms of the base tables, or write the update logic manually.

SQL also allows for the existence of duplicate rows, i.e. a table is allowed to contain two rows which have the same value for all columns. This fundamentally violates the definition of a relation [2], and causes numerous problems ranging from performance and optimization to interpretation and redundancy. One optimization available to relational systems is the ability of the system to transform a given relational expression into an equivalent expression which may be more efficient. Such transformations are performed according to the laws of transformation for the relational algebra. However, these laws cannot be applied to tables which contain duplicate rows. The problem of interpretation is made clear by a case study in which a single query was expressed in nine semantically equivalent representations, and "received nine answers that differ only in the number of duplicates they contain" [8].

II.A.2. Logical Data Independence

Logical data independence is the idea that all the tables (relations) available in a given database behave the same way, and are thus interchangeable. This fact is referred to as "The Principle of Interchangeability" [4]. The ability of a system to treat base and derived tables equally is critical to maintaining a consistent view of the database, and provides a necessary layer of abstraction between the consumer of the data, (the

application user) and the database management system. The inability of existing systems to update views provides a dramatic example of the difference between a table and a view; a difference which the relational model dictates should not exist. As a result, the application developer is forced to compensate in application code for a problem which should never have arisen in the first place.

II.A.3. Physical Data Independence

Physical data independence is the idea that the logical view of the database should remain unaffected by changes to the physical storage of the data. While existing systems certainly achieve a much higher degree of physical data independence than their forerunners, they fall short of the full practical benefits predicted by relational theory.

Existing systems map base tables directly to stored structures [7], which means that the physical representation of the database is constrained by the logical model. Many of the problems of performance, and their accompanying ‘solutions’ are due directly to this problem. If the database system allowed the mapping of logical tables to any storage mechanism, problems of performance could be resolved by rearranging the underlying physical storage, without affecting the logical model, and therefore the applications built against them. Because no such mapping layer exists, what is commonly termed ‘denormalization’, is all but required by existing systems.

II.B. Issues Due to Application Development Models

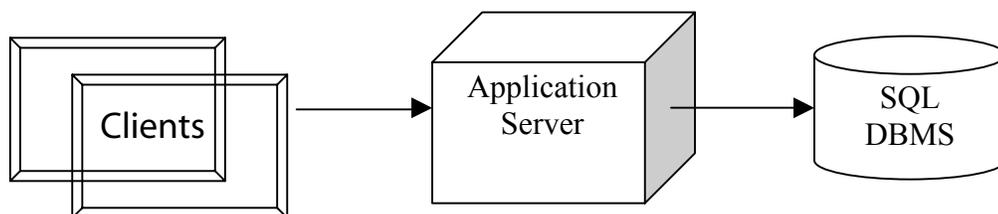
The second class of problems arises from the lack of a coherent model for database application development, and can be further divided into two main classes of problems. First, problems arising from the compensation code required by existing

database management systems, and second, problems arising from deficiencies in the call-level interface to the database system.

II.B.1. N-Tier Application Development

Current database application development is usually based on some variation of what is known as N-Tier development. As figure 1 illustrates, this model is based on the idea that there are several layers involved in any given application, beginning at one end with the persistent data store, and ending at the other with the presentation layer. The number of layers in between depends on the particular configuration employed, but usually involves at least one layer known as the Application Server. The main purpose of the Application Server is to attempt to consolidate the compensation code required by existing database management systems. The problem is that the Application Server is usually based on an object-oriented model, and so cannot provide all the benefits of a relational system. Application frontends are then required to manipulate data through an object model, which cannot serve as a foundation for data management [7]. The Application Server presents data through objects called “business-objects” which become the primary method of data manipulation for client applications. This model of database application development requires that the developer implement solutions for the same class of problems that the database management system was supposed to solve.

Figure 1. N-Tier Application Development



II.B.2. Call-Level Interface Deficiencies

The relational model provides a solid foundation for database manipulation and retrieval. However, when building presentation layer programs designed to allow users to input data into relational systems, several problems arise. First, the lack of an interface through which the application could request conditional information from the database server, and second, the difficulty of simulating navigational access and buffering data retrieval against a relational database system.

Database designs often include special definitions in the logical model intended to serve as the default value for columns if no value is provided. These defaults are typically coded into the client side application as well because the database server cannot answer the question “What would happen if I inserted a new row into this table?” By enabling the server to answer questions such as these, the duplication of the default in the client side can be avoided. The client simply requests that the server fill out an empty row with the default values for the columns of the table. This interface could also be used to verify values as they are entered, rather than waiting for the post against the database server. Database applications typically perform this type of validation, but through the enforcement of an equivalent client side constraint, rather than allowing the database server the ability to make the decisions.

Another difficult problem in database application development is providing a usable and intuitive interface for searching large amounts of data. Existing systems require that the application developer design the interface so as to avoid the return of large amounts of data, contributing to an unnatural “narrowing” search interface where the user must continuously refine the query until a small enough result is obtained to be

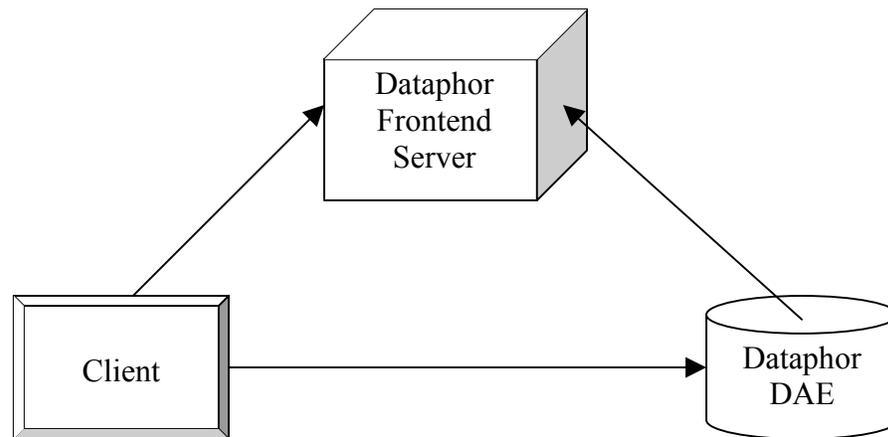
returned within a reasonable amount of time. By providing an interface through which the application could request a cursor that could be searched, without returning the entire result set, the application can provide a consistent, incremental search throughout the application.

III. DATAPHOR DATABASE APPLICATION DEVELOPMENT

These problems, and many others, led to the development of a set of tools called Dataphor, which is intended to solve all these problems, while also providing a foundation for the future of database management systems.

The Dataphor toolset introduces a new model for developing database applications. The DAE provides the infrastructure necessary for the implementation of three main facets of this model. First, the centralization of the business model, second, uniform access to any data source, and third, a rich call-level interface. All of these things are made possible by the introduction of a new database language, called D4.

Figure 2. Dataphor Application Development



III.A. Centralizing the Business Model

The business-model is a logical representation of all the data of interest to a particular entity or enterprise. This model includes not only what data is to be persisted,

but what that data means, and how it should be presented and manipulated. To describe the business model, the D4 language provides a complete set of structures for storing and defining data, collectively known as the Schema.

III.A.1. Schema Objects

The first of these is the Domain, which is, simply put, a named set of values, specifically scalar values. The Dataphor DAE includes several “built-in” domains, as well as allowing the developer to define any domain necessary. The D4 domain also includes several possible representations [4], and one physical representation. The physical representation of the domain is the actual byte arrangement of a value of the domain in physical storage, and is defined through the D4 host language. Physical representations are not exposed through the logical model. Possible representations, however, are the vehicle through which values are selected and operated upon. Each possible representation may consist of any number of components called properties, which may be of any D4 data type. For example, one possible representation of a DateTime data type is Seconds, which would be the representation of the given DateTime value as a number of Seconds. In this way, any given domain can be represented in any way that makes sense to the database designer. By separating the physical representation of a domain, and its logical representations, the database developer is able to make changes to the underlying storage, without affecting the behavior of the logical model.

Next, the D4 language provides for the creation of Operators, a named block of D4 executable code, which is capable of manipulating values of any data type. The

Dataphor DAE provides several “built-in” operators, such as the addition of two integers. Developers can use operators anywhere an expression or statement is valid in the language.

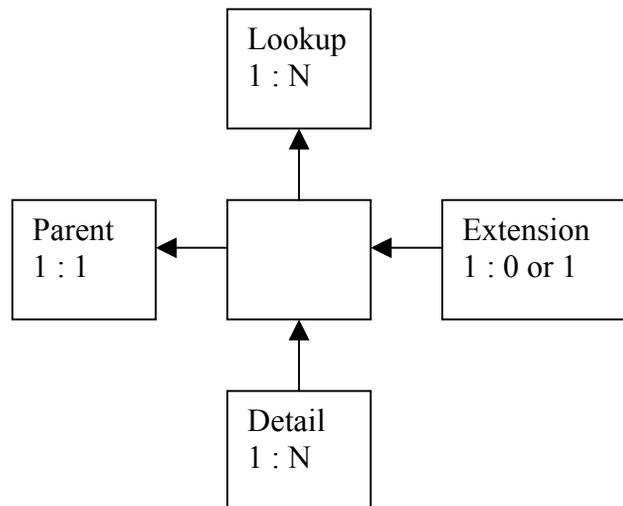
D4 provides two constructs for defining what data is to be available in the logical model, the table, and the view. The table is a named set of rows with a data type defined as a named set of columns, and the domains on which they are defined. The view is then defined as a named table which is the result of an arbitrary expression of the D4 language. For example, we might define a Customer table as capable of holding rows of the form {ID : integer, Name : string, TotalSpent : money} with a key of {ID}. Then we could define a view of Customer in which the TotalSpent column was greater than \$100.00. The resulting view would be indistinguishable from a base table in terms of behavior, i.e. they would both be relational expressions capable of appearing anywhere a relational expression could appear, and they would both exhibit the same updatability behavior, namely that any update which does not violate the meaning of the data is allowed to be performed.

A Constraint is a truth-valued expression which defines the meaning of the data to the database system. Constraints allow the database system to make validation decisions, and reject data which does not satisfy the constraints defined. The D4 language allows for the expression of constraints at four different levels. First, the Domain Constraint, which constrains values of a domain, the Column Constraint, which is the domain on which the column is defined, the Table Constraint, which constrains what values are legal within the columns of a table or view, and the Database Constraint, which is any arbitrary expression of the D4 language which must evaluate to true at all

transaction boundaries. Constraints which are not possible to express in a language like SQL, such as “ensure that the sum of data in column A of table A is equal to the value of column B in table B” are easily expressed in D4, eliminating the need for business logic to be developed outside the database system, and hence the Application Server from the N-Tier development model.

References are a special case of Constraints which can be declaratively specified in the D4 language. A reference constraint ensures that the values of a set of columns in a given table reference an existing row in another table. References are the D4 equivalent of the “foreign-key” construct in SQL, and allow the database system to make intelligent decisions about presentation, as well as giving the developer a convenient mechanism for expressing a common database constraint. As figure 3 illustrates, the Dataphor DAE categorizes references according to the cardinality of the relationship defined.

Figure 3. Types of References



III.A.2. Meta Data

In addition to allowing the specification of the logical model, the D4 language provides a mechanism for “decorating” that model, i.e. providing application specific information called meta data that is irrelevant to the logical model, but should be preserved in a one-to-one correspondence with it. For example, a typical database application must provide user-friendly names for the columns of data being entered. Instead of placing this information in the client side application, the D4 language allows the developer to specify a “tag” which contains the interface specific display title for a given domain. The application then simply reads that information from the database server. In this way, the logical model can be extended within the database server, eliminating the need for specific code to be written in the application. This eliminates a maintenance point, and helps to centralize schema information. Another key point about meta data is that it is derived, just like type information, through the expression, so that all the meta data about a given domain is available anywhere that domain appears, in particular it is available through the call-level interface as the result data type, and so can be used by the application, without having to query system catalog to obtain it.

III.A.3. Expression Derivation

Using this rich dictionary of information made available by the DAE, the application is capable of deriving expressions for the presentation of data. This process is known as Expression Derivation, and is used by the Dataphor Frontend to dynamically build user-interfaces based solely on the definition of the business model. The main entities involved in expression derivation are Tables (and Views), and References. The expression derivation process consists of building an expression based on a given table,

and the references in which that table participates, which can serve as the basis for user manipulation of data in that table.

III.A.4. User Interface Derivation

Once the expression has been derived, and the data type of the expression result is known (including meta data), the process of User Interface Derivation can use this information to produce usable user interfaces dynamically. This eliminates another point of schema duplication, and therefore reduces application maintenance. In a traditional application, the developer would build a form in the client application which is “aware” that a particular table exists, and that it contains a given set of columns, and so on. If the definition of that table changes, the client application must also change to reflect it.

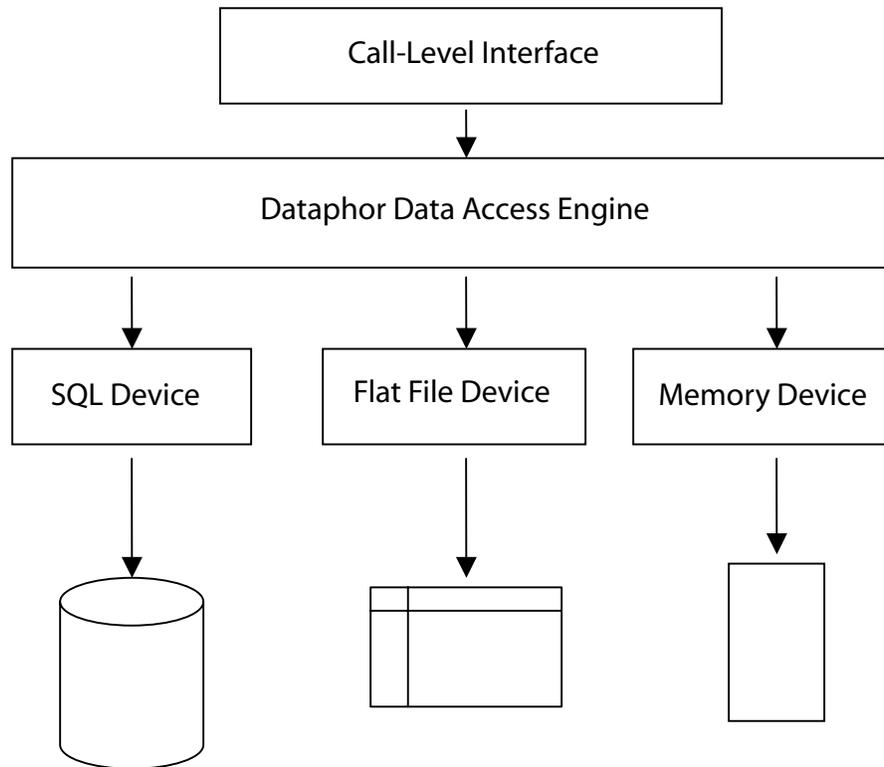
III.A.5. Minimizing Schema Duplication

All these features collectively allow a Dataphor Application to minimize the number of references to schema information contained in the DAE’s catalog. This not only provides a central repository for all the information about the business model, but reduces application development and maintenance time. The result is a dynamic and flexible application which changes easily in response to changes in the schema.

III.A.6. Uniform Access to Data

The Dataphor DAE provides a Storage Integration Architecture (SIA) which allows it to provide uniform relational access to any data source. By providing an extensible Device interface, the DAE is able to take advantage of the performance offered by existing systems, while compensating for any inadequacies present in those systems, and providing a single access point to all the data of a particular system. As shown in figure 4, the DAE uses Devices to persist data.

Figure 4. Dataphor Storage Integration Architecture



III.A.7. Rich Call-Level Interface

Each device is capable of answering the question “Do you support this expression.” If the device answers yes, the DAE hands off the expression at that point. Otherwise, it takes over and performs the necessary operations. The SIA also provides a mapping layer through the devices. A given view could actually be mapped to a stored table in the given device. This gives Dataphor applications an unprecedented level of physical data independence.

The Dataphor DAE provides a rich call-level interface, supporting what is called the Proposable interface. Through the use of this interface, applications are able to ask “what-if” style questions of the Dataphor DAE. For example, the Dataphor client application can ask what would happen if a new row were inserted into this table, and the

Dataphor DAE would respond by filling out an empty row with the values for defaults of each column in the table. This eliminates the need for the application developer to provide this functionality in the client side, while still giving the same level of usability from the end-users perspective. This also eliminates the duplication of the defaults, and it's accompanying maintenance point. In addition to Default, the DAE also provides column and row level Validate and Change interfaces which can be used to build more responsive and intuitive user interfaces.

IV. CONCLUSION

The Dataphor Toolset and the technologies on which it is based form a unique development solution. By providing a centralized repository for the business model, and uniform access to the information it contains, development is dramatically simplified, and maintenance time reduced. The Dataphor DAE provides an ideal environment for today's applications as well as an extensible foundation for the future of database development.

V. REFERENCES

1. Codd, E. F. (1969). *Derivability, Redundancy, and Consistency of Relations Stored in Large Shared Data Banks* (IBM Research Report RJ599).
2. Date, C. J. (2000). *An Introduction to Database Systems* (7th ed.). New York: Addison Wesley Longman.
3. Date, C. J. (1995). *Relational Database Writings 1991-1994*. New York: Addison Wesley Longman.
4. Date, C. J., & Darwen, H. (2000). *Foundation for Future Database Systems: The Third Manifesto* (2nd ed.). New York: Addison Wesley Longman.
5. Date, C. J., & Darwen, H. (1992). *Relational Database Writings 1989-1991*. New York: Addison Wesley Longman.
6. Date, C. J., & Warden, A. (1990). *Relational Database Writings 1985-1989*. New York: Addison Wesley Longman.
7. Date, C. J., Darwen, H., & McGoveran, D. (1998). *Relational Database Writings 1994-1997*. New York: Addison Wesley Longman.
8. Pascal, F. (2000). *Practical Issues in Database Management: A Reference for the Thinking Practitioner*. New York: Addison Wesley Longman, Inc.